Макс Шлее

bhv®

Qt 5.10

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ НА C++



- Кроссплатформенная реализация приложений для Windows, Mac OS X и Linux
- Разработка мобильных приложений для Android и iOS
- Программирование 2D- и 3D-графики, мультимедиа, веб-приложений, баз данных, сети, таймера, многопоточности, XML, QML и JavaScript
- 240 завершенных программ



Материалы на www.bhv.ru

Наиболее полное руководство

В ПОДЛИННИКЕ®

Макс Шлее

Qt 5.10
ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ НА C++

Санкт-Петербург
«БХВ-Петербург»
2018

УДК 004.438C++
ББК 32.973.26-018.1
Ш68

Шлее М.

Ш68 Qt 5.10. Профессиональное программирование на C++. — СПб.: БХВ-Петербург, 2018. — 1072 с.: ил. — (В подлиннике)

ISBN 978-5-9775-3678-3

Книга посвящена разработке приложений для Windows, Mac OS X, Linux, Android и iOS с использованием библиотеки Qt версии 5.10. Подробно рассмотрены возможности, предоставляемые этой библиотекой, и описаны особенности, выгодно отличающие ее от других библиотек. Описана интегрированная среда разработки Qt Creator и работа с технологией Qt Quick. Книга содержит исчерпывающую информацию о классах Qt 5, и так же даны практические рекомендации их применения, проиллюстрированные на большом количестве подробно прокомментированных примеров. Проекты примеров из книги размещены на сайте издательства.

Для программистов

УДК 004.438C++
ББК 32.973.26-018.1

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Екатерина Капалыгина*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Марины Дамбиевой*

Подписано в печать 30.03.18.
Формат 70×100$^{1}/_{16}$. Печать офсетная. Усл. печ. л. 86,43.
Тираж 1200 экз. Заказ № 8047.
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

Отпечатано в ОАО «Можайский полиграфический комбинат».
143200, г. Можайск, ул. Мира, 93.
www.oaompk.ru, www.оаомпк.рф тел.: (495) 745-84-28, (49638) 20-685

ISBN 978-5-9775-3678-3

© ООО "БХВ", 2018
© Оформление. ООО "БХВ-Петербург", 2018

Посвящается
любимой Аленушке,
моим детям,
родителям
и семейству Гоуз (Goes)

Оглавление

Предисловие Маттиаса Эттриха 1

Благодарности 3

Предисловие автора 4
Структура книги 5

Введение 13

ЧАСТЬ I. ОСНОВЫ QT 25

Глава 1. Обзор иерархии классов Qt 26
Первая программа на Qt 26
Модули Qt 27
Пространство имен Qt 29
Модуль *QtCore* 29
Модуль *QtGui* 30
Модуль *QtWidgets* 30
Модули *QtQuick* и *QtQML* 32
Модуль *QtNetwork* 32
Модули *QtXml* и *QtXmlPatterns* 32
Модуль *QtSql* 32
Модули *QtMultimedia* и *QtMultimediaWidgets* 32
Модуль *QtSvg* 32
Дополнительные модули Qt 32
Резюме 33

Глава 2. Философия объектной модели 35
Механизм сигналов и слотов 38
Сигналы 41
Слоты 43
Соединение объектов 44
Разъединение объектов 49
Переопределение сигналов 50

Организация объектных иерархий 51
Метаобъектная информация 53
Резюме 54

Глава 3. Работа с Qt 55
Интегрированная среда разработки 55
Программа Qt Assistant 55
Работа с qmake 55
Рекомендации для проекта с Qt 59
Метаобъектный компилятор MOC 60
Компилятор ресурсов RCC 61
Структура Qt-проекта 62
Методы отладки 62
Отладчик GDB (GNU Debugger) 63
Прочие методы отладки 66
Глобальные определения Qt 69
Информация о библиотеке Qt 71
Резюме 72

Глава 4. Библиотека контейнеров 74
Контейнерные классы 75
Итераторы 76
Итераторы в стиле Java 77
Итераторы в стиле STL 78
Ключевое слово *foreach* 80
Последовательные контейнеры 80
Вектор *QVector<T>* 82
Массив байтов *QByteArray* 83
Массив битов *QBitArray* 83
Списки *QList<T>* и *QLinkedList<T>* 84
Стек *QStack<T>* 85
Очередь *QQueue<T>* 86
Ассоциативные контейнеры 86
Словари *QMap<K,T>* и *QMultiMap<K,T>* 87
Хэши *QHash<K,T>* и *QMultiHash<K,T>* 89
Множество *QSet<T>* 90
Алгоритмы 91
Сортировка 92
Поиск 93
Сравнение 94
Заполнение значениями 94
Копирование значений элементов 94
Подсчет значений 95
Строки 95
Регулярные выражения 97
Произвольный тип *QVariant* 100
Модель общего использования данных 101
Резюме 102

ЧАСТЬ II. ЭЛЕМЕНТЫ УПРАВЛЕНИЯ 103

Глава 5. С чего начинаются элементы управления? 104
Класс *QWidget* 104
Размеры и координаты виджета 107
Механизм закулисного хранения 108
Установка фона виджета 108
Изменение указателя мыши 109
Стек виджетов 112
Рамки 112
Виджет видовой прокрутки 113
Резюме 115

Глава 6. Управление автоматическим размещением элементов 116
Менеджеры компоновки (layout managers) 116
Горизонтальное и вертикальное размещение 118
Класс *QBoxLayout* 118
Горизонтальное размещение *QHBoxLayout* 120
Вертикальное размещение *QVBoxLayout* 121
Вложенные размещения 122
Табличное размещение *QGridLayout* 123
Порядок следования табулятора 129
Разделители *QSplitter* 129
Резюме 130

Глава 7. Элементы отображения 132
Надписи 132
Индикатор выполнения 136
Электронный индикатор 139
Резюме 141

Глава 8. Кнопки, флажки и переключатели 142
С чего начинаются кнопки? Класс *QAbstractButton* 142
Установка текста и изображения 142
Взаимодействие с пользователем 142
Опрос состояния 143
Кнопки 143
Флажки 145
Переключатели 147
Группировка кнопок 148
Резюме 151

Глава 9. Элементы настройки 152
Класс *QAbstractSlider* 152
Изменение положения 152
Установка диапазона 152
Установка шага 153
Установка и получение значений 153

Ползунок ... 153
Полоса прокрутки ... 155
Установщик ... 156
Резюме ... 158

Глава 10. Элементы ввода ... 159
Однострочное текстовое поле ... 159
Редактор текста ... 161
Запись в файл ... 164
Расцветка синтаксиса (syntax highlighting) ... 165
С чего начинаются виджеты счетчиков? ... 171
Счетчик ... 171
Элемент ввода даты и времени ... 172
Проверка ввода ... 173
Резюме ... 175

Глава 11. Элементы выбора ... 176
Простой список ... 176
Вставка элементов ... 176
Выбор элементов пользователем ... 178
Изменение элементов пользователем ... 178
Режим пиктограмм ... 178
Сортировка элементов ... 179
Иерархические списки ... 180
Сортировка элементов ... 183
Таблицы ... 183
Выпадающий список ... 185
Вкладки ... 186
Виджет панели инструментов ... 187
Резюме ... 188

Глава 12. Интервью, или модель-представление ... 189
Концепция ... 190
Модель ... 190
Представление ... 192
Выделение элемента ... 193
Делегат ... 195
Индексы модели ... 197
Иерархические данные ... 198
Роли элементов ... 201
Создание собственных моделей данных ... 203
Промежуточная модель данных (Proxy model) ... 211
Модель элементно-ориентированных классов ... 213
Резюме ... 215

Глава 13. Цветовая палитра элементов управления ... 217
Резюме ... 220

ЧАСТЬ III. СОБЫТИЯ И ВЗАИМОДЕЙСТВИЕ С ПОЛЬЗОВАТЕЛЕМ 221

Глава 14. События 222
Переопределение специализированных методов обработки событий 224
События клавиатуры 225
Класс *QKeyEvent* 225
Класс *QFocusEvent* 227
Событие обновления контекста рисования. Класс *QPaintEvent* 227
События мыши 228
Класс *QMouseEvent* 228
Класс *QWheelEvent* 232
Методы *enterEvent()* и *leaveEvent()* 233
Событие таймера. Класс *QTimerEvent* 233
События перетаскивания (drag & drop) 233
Класс *QDragEnterEvent* 233
Класс *QDragLeaveEvent* 233
Класс *QDragMoveEvent* 233
Класс *QDropEvent* 234
Остальные классы событий 234
Класс *QChildEvent* 234
Класс *QCloseEvent* 234
Класс *QHideEvent* 234
Класс *QMoveEvent* 234
Класс *QShowEvent* 234
Класс *QResizeEvent* 234
Реализация собственных классов событий 236
Переопределение метода *event()* 236
Мультитач 239
Сохранение работоспособности приложения 244
Резюме 244

Глава 15. Фильтры событий 246
Реализация фильтров событий 246
Резюме 249

Глава 16. Искусственное создание событий 250
Резюме 253

ЧАСТЬ IV. ГРАФИКА И ЗВУК 255

Глава 17. Введение в компьютерную графику 256
Классы геометрии 256
Точка 256
Двумерный размер 257
Прямоугольник 259
Прямая линия 259
Многоугольник 260
Цвет 260
Класс *QColor* 260

Цветовая модель RGB 261
Цветовая модель HSV 262
Цветовая модель CMYK 263
Палитра 264
Предопределенные цвета 265
Резюме 266

Глава 18. Легенда о короле Артуре и контекст рисования 267
Класс *QPainter* 268
Перья и кисти 270
Перо 270
Кисть 271
Градиенты 272
Техника сглаживания (Anti-aliasing) 274
Рисование 275
Рисование точек 275
Рисование линий 275
Рисование сплошных прямоугольников 277
Рисование заполненных фигур 277
Запись команд рисования 281
Трансформация систем координат 281
Перемещение 282
Масштабирование 283
Поворот 283
Скос 283
Трансформационные матрицы 283
Графическая траектория (painter path) 284
Отсечения 285
Режим совмещения (composition mode) 286
Графические эффекты 289
Резюме 291

Глава 19. Растровые изображения 293
Форматы графических файлов 293
Формат BMP 293
Формат GIF 294
Формат PNG 294
Формат JPEG 294
Формат XPM 294
Контекстно-независимое представление 296
Класс *QImage* 296
Класс *QImage* как контекст рисования 303
Контекстно-зависимое представление 304
Класс *QPixmap* 305
Класс *QPixmapCache* 306
Класс *QBitmap* 306
Создание нестандартного окна виджета 307
Резюме 308

Глава 20. Работа со шрифтами 310
Отображение строки 312
Резюме 315

Глава 21. Графическое представление 316
Сцена 317
Представление 318
Элемент 319
События 321
Виджеты в графическом представлении 326
Резюме 328

Глава 22. Анимация 330
Класс *QMovie* 330
SVG-графика 332
Анимационный движок и машина состояний 333
Смягчающие линии 336
Машина состояний и переходы 341
Резюме 343

Глава 23. Работа с OpenGL 345
Основные положения OpenGL 345
Реализация OpenGL-программы 347
Разворачивание OpenGL-программ во весь экран 350
Графические примитивы OpenGL 350
Трехмерная графика 354
Резюме 358

Глава 24. Вывод на печать 359
Класс *QPrinter* 359
Резюме 364

Глава 25. Разработка собственных элементов управления 365
Примеры создания виджетов 365
Резюме 370

Глава 26. Элементы со стилем 371
Встроенные стили 373
Создание собственных стилей 377
Метод рисования простых элементов управления 378
Метод рисования элементов управления 378
Метод рисования составных элементов управления 378
Реализация стиля простого элемента управления 379
Использование каскадных стилей документа 382
Основные положения 383
Изменение подэлементов 385
Управление состояниями 386
Пример 387
Резюме 391

Глава 27. Мультимедиа ... 392
Звук ... 392
Воспроизведение WAV-файлов: класс *QSound* ... 393
Более продвинутые возможности воспроизведения звуковых файлов: класс *QMediaPlayer* ... 394
Видео и класс *QMediaPlayer* ... 401
Резюме ... 403

ЧАСТЬ V. СОЗДАНИЕ ПРИЛОЖЕНИЙ ... 405

Глава 28. Сохранение настроек приложения ... 406
Резюме ... 413

Глава 29. Буфер обмена и перетаскивание ... 414
Буфер обмена ... 414
Перетаскивание ... 415
Реализация drag ... 417
Реализация drop ... 419
Создание собственных типов перетаскивания ... 421
Резюме ... 426

Глава 30. Интернационализация приложения ... 428
Подготовка приложения к интернационализации ... 428
Утилита lupdate ... 430
Программа Qt Linguist ... 431
Утилита lrelease. Пример программы, использующей перевод ... 433
Смена перевода в процессе работы программы ... 435
Завершающие размышления ... 437
Резюме ... 438

Глава 31. Создание меню ... 439
«Анатомия» меню ... 439
Контекстные меню ... 443
Резюме ... 444

Глава 32. Диалоговые окна ... 445
Правила создания диалоговых окон ... 445
Класс *QDialog* ... 446
Модальные диалоговые окна ... 446
Немодальные диалоговые окна ... 447
Создание собственного диалогового окна ... 447
Стандартные диалоговые окна ... 451
Диалоговое окно выбора файлов ... 451
Диалоговое окно настройки принтера ... 454
Диалоговое окно выбора цвета ... 455
Диалоговое окно выбора шрифта ... 457
Диалоговое окно ввода ... 458
Диалоговое окно процесса ... 459
Диалоговые окна мастера ... 460

Диалоговые окна сообщений 462
Окно информационного сообщения 464
Окно предупреждающего сообщения 464
Окно критического сообщения 465
Окно сообщения о программе 466
Окно сообщения *About Qt* 466
Окно сообщения об ошибке 467
Резюме 467

Глава 33. Предоставление помощи 469
Всплывающая подсказка 469
Система помощи (Online Help) 471
Резюме 473

Глава 34. Главное окно, создание SDI- и MDI-приложений 475
Класс главного окна *QMainWindow* 475
Класс действия *QAction* 476
Панель инструментов 477
Доки 479
Строка состояния 480
Окно заставки 482
SDI- и MDI-приложения 484
SDI-приложение 484
MDI-приложение 488
Резюме 496

Глава 35. Рабочий стол (Desktop) 497
Область уведомлений 497
Виджет экрана 502
Класс сервиса рабочего стола 506
Резюме 506

ЧАСТЬ VI. ОСОБЫЕ ВОЗМОЖНОСТИ QT 507

Глава 36. Работа с файлами, каталогами и потоками ввода/вывода 508
Ввод/вывод. Класс *QIODevice* 508
Работа с файлами. Класс *QFile* 510
Класс *QBuffer* 512
Класс *QTemporaryFile* 512
Работа с каталогами. Класс *QDir* 512
Просмотр содержимого каталога 513
Информация о файлах. Класс *QFileInfo* 516
Файл или каталог? 516
Путь и имя файла 517
Информация о дате и времени 517
Получение атрибутов файла 517
Определение размера файла 517
Наблюдение за файлами и каталогами 518

Потоки ввода/вывода ... 520
Класс *QTextStream* ... 521
Класс *QDataStream* ... 522
Резюме ... 523

Глава 37. Дата, время и таймер ... 524
Дата и время ... 524
Класс даты *QDate* ... 524
Класс времени *QTime* ... 526
Класс даты и времени *QDateTime* ... 527
Таймер ... 527
Событие таймера ... 528
Класс *QTimer* ... 530
Класс *QBasicTimer* ... 532
Резюме ... 532

Глава 38. Процессы и потоки ... 533
Процессы ... 533
Потоки ... 536
Приоритеты ... 538
Обмен сообщениями ... 539
Сигнально-слотовые соединения ... 540
Отправка событий ... 544
Синхронизация ... 547
Мьютексы ... 547
Семафоры ... 549
Ожидание условий ... 550
Блокировка чтения/записи ... 550
Возникновение тупиковых ситуаций ... 551
Фреймворк *QtConcurrent* ... 551
Резюме ... 553

Глава 39. Программирование поддержки сети ... 555
Сокетное соединение ... 555
Модель «клиент-сервер» ... 556
Реализация TCP-сервера ... 557
Реализация TCP-клиента ... 562
Реализация UDP-сервера и UDP-клиента ... 566
Управление доступом к сети ... 570
Блокирующий подход ... 577
Режим прокси ... 580
Информация о хосте ... 580
Есть ли соединение с Интернетом? ... 581
Резюме ... 581

Глава 40. Работа с XML ... 582
Основные понятия и структура XML-документа ... 582
XML и Qt ... 584
Работа с DOM ... 584
Чтение XML-документа ... 585
Создание и запись XML-документа ... 587

Работа с SAX ... 589
Чтение XML-документа ... 589
Класс *QXmlStreamReader* для чтения XML ... 592
Использование XQuery ... 594
Резюме ... 597

Глава 41. Программирование баз данных ... 599
Основные положения SQL ... 599
Создание таблицы ... 600
Операция вставки ... 600
Чтение данных ... 600
Изменение данных ... 601
Удаление ... 601
Использование языка SQL в библиотеке Qt ... 601
Соединение с базой данных (второй уровень) ... 603
Исполнение команд SQL (второй уровень) ... 604
Классы SQL-моделей для интервью (третий уровень) ... 607
Модель запроса ... 607
Табличная модель ... 608
Реляционная модель ... 610
Резюме ... 611

Глава 42. Динамические библиотеки и система расширений ... 613
Динамические библиотеки ... 613
Динамическая загрузка и выгрузка библиотеки ... 614
Расширения (plug-ins) ... 617
Расширения для Qt ... 617
Поддержка собственных расширений в приложениях ... 619
Создание расширения для приложения ... 623
Резюме ... 625

Глава 43. Совместное использование Qt с платформозависимыми API ... 627
Совместное использование с Windows API ... 629
Совместное использование с Linux ... 632
Совместное использование с Mac OS X ... 632
Системная информация ... 637
Резюме ... 639

Глава 44. Qt Designer. Быстрая разработка прототипов ... 640
Создание новой формы в Qt Designer ... 640
Добавление виджетов ... 643
Компоновка (layout) ... 644
Порядок следования табулятора ... 645
Сигналы и слоты ... 646
Использование в формах собственных виджетов ... 648
Использование форм в проектах ... 648
Компиляция ... 651
Динамическая загрузка формы ... 651
Резюме ... 654

Глава 45. Проведение тестов 655
Создание тестов 656
Тесты с передачей данных 659
Создание тестов графического интерфейса 661
Параметры для запуска тестов 663
Резюме 664

Глава 46. Qt WebEngine 665
А зачем? 666
Быстрый старт 667
Создание простого веб-браузера 669
Ввод адресов 669
Управление историей 669
Загрузка страниц и ресурсов 670
Пишем веб-браузер: попытка номер два 670
Резюме 674

Глава 47. Интегрированная среда разработки Qt Creator 675
Первый запуск 676
Создаем проект «Hello Qt Creator» 677
Пользовательский интерфейс Qt Creator 681
Окна вывода 682
Окно проектного обозревателя 682
Секция компилирования и запуска 682
Редактирование текста 685
Как подсвечен ваш синтаксис? 685
Скрытие и отображение кода 686
Автоматическое дополнение кода 686
Поиск и замена 687
Комбинации клавиш для ускорения работы 691
Вертикальное выделение текста 691
Автоматическое форматирование текста 691
Комментирование блоков 692
Просмотр кода методов класса, их определений и атрибутов 692
Помощь, которая всегда рядом 693
Интерактивный отладчик и программный экзорцизм 694
Синтаксические ошибки 695
Ошибки компоновки 696
Ошибки времени исполнения 696
Логические ошибки 697
Трассировка 697
Команда *Step Over* 698
Команда *Step Into* 698
Команда *Step Out* 699
Контрольные точки 699
Окно переменных (Local and Watches) 700
Окно цепочки вызовов (Call Stack) 701
Резюме 701

Глава 48. Рекомендации по миграции программ из Qt 4 в Qt 5 703
Основные отличия Qt 5 от Qt 4 703
Подробности перевода на Qt 5 703
Виджеты 704
Контейнерные классы 704
Функция *qFindChildren<T>()* 705
Сетевые классы 705
WebKit 705
Платформозависимый код 705
Система расширений Plug-ins 705
Принтер *QPrinter* 706
Мультимедиа 706
Модульное тестирование 706
Реализация обратной совместимости Qt 5 с Qt 4 706
Резюме 709

ЧАСТЬ VII. ЯЗЫК СЦЕНАРИЕВ JAVASCRIPT 711

Глава 49. Основы поддержки сценариев JavaScript 712
Принцип взаимодействия с языком сценариев 713
Первый шаг использования сценария 716
Привет, сценарий 717
Резюме 718

Глава 50. Синтаксис языка сценариев 720
Зарезервированные ключевые слова 720
Комментарии 721
Переменные 721
Предопределенные типы данных 722
Целый тип 722
Вещественный тип 722
Строковый тип 723
Логический тип 723
Преобразование типов 723
Операции 725
Операторы присваивания 725
Арифметические операции 725
Поразрядные операции 726
Операции сравнения 726
Приоритет выполнения операций 727
Управляющие структуры 728
Условные операторы 728
Оператор *if... else* 728
Оператор *switch* 729
Оператор условного выражения 730
Циклы 730
Операторы *break* и *continue* 730
Цикл *for* 730

Цикл *while* ... 731
Цикл *do...while* ... 731
Оператор *with* ... 732
Исключительные ситуации ... 732
Оператор *try...catch* ... 732
Оператор *throw* ... 733
Функции ... 733
Встроенные функции ... 735
Объектная ориентация ... 735
Статические классы ... 738
Наследование ... 738
Перегрузка методов ... 741
Сказание о «джейсоне» ... 742
Резюме ... 743

Глава 51. Встроенные объекты JavaScript ... 744
Объект *Global* ... 744
Объект *Number* ... 744
Объект *Boolean* ... 744
Объект *String* ... 745
Замена ... 745
Получение символов ... 745
Получение подстроки ... 745
Объект *RegExp* ... 745
Проверка строки ... 746
Поиск позиции совпадений ... 746
Найденное совпадение ... 746
Объект *Array* ... 746
Дополнение массива элементами ... 747
Адресация элементов ... 747
Изменение порядка элементов массива ... 747
Преобразование массива в строку ... 748
Объединение массивов ... 748
Упорядочивание элементов ... 748
Многомерные массивы ... 748
Объект *Date* ... 749
Объект *Math* ... 750
Модуль числа ... 750
Округление ... 751
Определение максимума и минимума ... 751
Возведение в степень ... 751
Вычисление квадратного корня ... 751
Генератор случайных чисел ... 752
Тригонометрические методы ... 752
Вычисление натурального логарифма ... 752
Объект *Function* ... 753
Резюме ... 753

Глава 52. Классы поддержки JavaScript и практические примеры 754
Класс *QJSValue* 754
Класс *QJSEngine* 754
Практические примеры 755
«Черепашья» графика 755
Сигналы, слоты и функции 762
Полезные дополнительные функции 765
Резюме 769

ЧАСТЬ VIII. ТЕХНОЛОГИЯ QT QUICK 771

Глава 53. Знакомство с Qt Quick 772
А зачем? 772
Введение в QML 774
Быстрый старт 776
Использование JavaScript в QML 782
Резюме 783

Глава 54. Элементы 785
Визуальные элементы 785
Свойства элементов 788
Собственные свойства 790
Создание собственных элементов 793
Создание собственных модулей 795
Динамическое создание элементов 795
Элемент *Flickable* 796
Готовые элементы пользовательского интерфейса 797
Диалоговые окна 802
Резюме 805

Глава 55. Управление размещением элементов 806
Фиксаторы 806
Традиционные размещения 813
Размещение в виде потока 817
Резюме 819

Глава 56. Элементы графики 820
Цвета 820
Растровые изображения 821
Элемент *Image* 821
Элемент *BorderImage* 825
Градиенты 826
Шрифты 828
Рисование на элементах холста 828
Шейдеры и эффекты 833
Резюме 837

Глава 57. Пользовательский ввод 838
Область мыши 838
Сигналы 841

Ввод с клавиатуры ... 845
Фокус ... 846
«Сырой» ввод ... 848
Мультитач ... 850
Резюме ... 852

Глава 58. Анимация ... 853
Анимация при изменении свойств ... 853
Анимация для изменения числовых значений ... 855
Анимация с изменением цвета ... 856
Анимация с поворотом ... 857
Анимации поведения ... 858
Параллельные и последовательные анимации ... 860
Состояния и переходы ... 863
Состояния ... 863
Переходы ... 866
Модуль частиц ... 868
Резюме ... 872

Глава 59. Модель/Представление ... 873
Модели ... 873
Модель списка ... 873
XML-модель ... 874
JSON-модель ... 876
Представление данных моделей ... 877
Элемент *ListView* ... 877
Элемент *GridView* ... 880
Элемент *PathView* ... 882
Визуальная модель данных ... 884
Резюме ... 886

Глава 60. Qt Quick и C++ ... 888
Использование языка QML в C++ ... 888
Взаимодействие из C++ со свойствами QML-элементов и вызов их функций ... 889
Соединение QML-сигналов со слотами C++ ... 891
Использование компонентов языка C++ в QML ... 894
Экспорт объектов и виджетов из C++ в QML ... 895
Использование зарегистрированных объектов C++, их свойств и методов в QML ... 897
Реализация визуальных элементов QML на C++ ... 901
Класс *QQuickImageProvider* ... 904
Резюме ... 909

Глава 61. 3D-графика Qt 3D ... 910
Основы ... 910
Свет ... 911
Камера ... 912
3D-объекты ... 913
Материалы ... 916
Трансформация ... 919

Анимация ... 921
Qt 3D Studio ... 923
Резюме ... 924

ЧАСТЬ IX. МОБИЛЬНЫЕ ПРИЛОЖЕНИЯ И QT ... 927

Глава 62. Введение в мир мобильных приложений ... 928
Смартфоны меняют все ... 928
Виртуальные магазины приложений ... 930
Распространение приложений вне виртуального магазина ... 932
Qt и разработка мобильных приложений ... 932
Резюме ... 934

Глава 63. Подготовка к работе над мобильными приложениями ... 935
Подготовка среды для разработки iOS-приложений ... 935
Подготовка среды для разработки Android приложений ... 938
Запуск приложений на реальном устройстве ... 945
Резюме ... 946

Глава 64. Особенности разработки приложений для мобильных устройств ... 948
Анатомия файлов свойств для iOS- и Android-приложений ... 949
Файл свойств iOS-приложений ... 949
Файл свойств Android-приложений ... 951
Полноэкранный режим ... 955
iOS-реализация ... 956
Android-реализация ... 956
Автоматический поворот ... 956
Конфигурирование приложений для поддержки поворота ... 957
iOS-реализация ... 957
Android-реализация ... 958
Обработка поворота в приложениях ... 958
Сенсоры ... 960
Пользовательский ввод при помощи пальцев ... 964
Положение рук ... 967
Резюме ... 967

Глава 65. Пример разработки мобильного приложения ... 969
Обдумывание и планирование приложения ... 969
Название приложения ... 970
Значок приложения ... 970
Что будет в первой версии? ... 971
Пишем код ... 973
Добавление к приложению значков и стартовых экранов ... 980
iOS-реализация ... 980
Android-реализация ... 982
Резюме ... 983

Глава 66. Публикация в магазине мобильных приложений ... 984
Этапы работы для App Store ... 984
Регистрация ... 984

Настройки для запуска приложений на реальных устройствах 985
Создание электронной подписи 991
Создание страницы приложения 992
Загрузка и публикация приложения 998
Этапы работы для Google Play 1000
Регистрация 1001
Создание страницы приложения 1001
Создание электронной подписи 1005
Загрузка и публикация приложения 1007
Резюме 1009

ПРИЛОЖЕНИЯ 1011

Приложение 1. Настройка среды для работы над Qt-приложениями 1012
Настройка среды для Mac OS X 1012
Настройка среды для Windows 1013
Настройка среды для Ubuntu Linux 1015

Приложение 2. Таблица простых чисел 1018

Приложение 3. Таблицы семибитной кодировки ASCII 1021

Приложение 4. Описание архива с примерами 1024

Предметный указатель 1034

Любая достаточно передовая технология неотличима от магии.

Артур Кларк

Предисловие Маттиаса Эттриха

Let's start with a fictional story. Imagine ten years ago, someone came to me and asked: «Is it possible to write a feature-rich graphical application, and then compile and run this application natively on all different major operating systems? On Linux, on UNIX, on Windows, and on the Macintosh?» Back then — as a young computer scientist — I would probably have answered, «No, that's not possible. And if it was, the system would be very difficult to use, and limited by the weakest platform. Better choose one platform, or write your code several times.»

A few years later I discovered Qt — and how wrong I was!

Qt makes true cross-platform programming a reality, without limiting your choices and creativity. It gives users what users want: fast, native applications that look and feel just right. It gives developers what developers want: a framework that lets us write less code, and create more. A framework that makes programming fun again, no matter whether we do commercial work or contribute to Open Source projects.

Too good to be true? You don't believe me? Well, the proof is easy. I'll pass the word on to Max, who will tell you exactly how it's done. Max, your turn.

Before I leave, let me wish you good luck with your first Qt-steps. But be careful, it may very well turn into a lifetime addiction. Either way, I hope you will have as much fun using Qt as we have creating it for you.

Matthias Ettrich
October 1st, 2004, Oslo

Давайте пофантазируем. Представьте себе, будто бы 10 лет назад кто-то подошел ко мне и спросил: «Возможно ли создать многофункциональное приложение с графическим интерфейсом пользователя, а затем откомпилировать его и пользоваться на всех распространенных операционных системах? На Linux, UNIX, Windows, Macintosh?» В то время я был молодым программистом, и я бы, наверное, ответил: «Нет, это невозможно. А если это и было бы возможным, то такая система была бы очень трудна в обращении и ограничена возможностями самой слабой платформы. Лучше выбрать одну операционную систему или переписать свою программу несколько раз».

Несколько лет спустя я открыл для себя Qt — и понял, как я был не прав!

Qt делает платформонезависимое программирование действительностью, не ограничивая ваш выбор и творческие возможности. Qt предоставляет пользователям то, чего они хотят: быстрые программы, которые выглядят и работают должным образом. Qt предоставляет разработчикам программ то, чего они желают: среду, позволяющую писать меньше кода, создавая при этом больше. Благодаря этому программирование становится интереснее, и при этом неважно, является оно коммерческим или проектом с открытым исходным кодом (Open Source).

Слишком хорошо, чтобы быть правдой? Вы мне не верите? Ну что же, доказать это просто. Я передаю слово Максу, который расскажет вам подробно, как это делается. Макс, теперь твоя очередь.

Прежде чем я попрощаюсь, позвольте пожелать вам удачи в ваших первых шагах с Qt. Но осторожно, Qt может вызвать у вас зависимость на всю жизнь. В любом случае, я надеюсь, что вам будет так же интересно работать с Qt, как нам было интересно создавать ее для вас.

Маттиас Эттрих
1 октября 2004, Осло

Благодарности

Автор выражает глубокую признательность своей первой наставнице в области информатики — Татьяне Дмитриевне Оболенцевой — преподавателю Новосибирского филиала Московского технологического университета легкой промышленности, разбудившей в нем творческий потенциал. А также профессору, доктору Ульриху Айзэнекеру (Ulrich W. Eisenecker), который помог ему определиться в многообразном мире информатики.

Большую помощь в создании этой книги оказали самые близкие автору люди: Алена Шлее, родители Евгений и Галина Шлее, сестра Натали Гоуз.

Глубокую признательность и уважение испытывает автор ко всему коллективу издательства «БХВ-Петербург», а в особенности к Игорю Владимировичу Шишигину, Юрию Викторовичу Рожко, Андрею Геннадиевичу Смышляеву, Юрию Владимировичу Якубовичу, Евгению Евгеньевичу Рыбакову и Григорию Лазаревичу Добину за их поддержку и сотрудничество.

Особая благодарность Маттиасу Эттриху (Matthias Ettrich) — сотруднику фирмы Nokia и основателю KDE — за проявленный интерес и поддержку, оказанную при подготовке книги. Автор благодарит Кента Ханзена (Kent Hansen) и Андреаса Ардаль Ханссена (Andreas Aardal Hanssen) за проверку примеров книги, а также остальных сотрудников фирмы Nokia за замечательную библиотеку, которая вдохновила его на написание этой книги.

Я также выражаю благодарность моим читателям, присылавшим свои отклики, замечания и предложения: Виталию Улыбину, Александру Климову, Артуру Акопяну, Ирине Романенко, Вячеславу Гурковскому, Николаю Прокушину, Юрию Зинченко, Людмиле Брагиной, Алексею Старченко, Дмитрию Оленченко, Антону Матросову, Михаилу Кипа, Денису Песоцкому, Павлу Плотникову, Ярославу Васильеву, Михаилу Ермоленко, Виталию Венделю, Александру Басову и Николаю Прохоренок, Александру Матвееву, Стасу Койнову, Андрею Донцову, Ивану Ензхаеву, Александру Гилевичу, Александру Миргородскому, Максу Вальтеру, Ерну Белинину, Максиму Дзамбаеву, Денису Тену, Александру Марченко, Никите Липовичу, Артему Спиридонову, Ярославу Баранову, Семену Пейтонову, Олегу Белекову, Всеволоду Лукьянину, Александру Кузнецову, Ивану Ежову, Вадиму Сорочану, Алексею Пуц и Евгению Карныгину.

Предисловие автора

Занимайтесь любимым делом и тогда в вашей жизни не будет ни одного рабочего дня...

Здравствуйте, дорогие читатели! Рад сообщить вам, что наконец-то задуманное свершилось, и воплощенное в жизнь новое издание этой книги вышло в свет! Перед вами не просто знакомая вам всем по предыдущим изданиям книга о Qt, которая, надеюсь, гарантированно обеспечит вас дополнительными знаниями о том, как разрабатывать приложения. Новое издание постепенно и доходчиво введет вас в курс того, как вы сможете *применять* свои знания. И это очень важно, ведь на самом деле, как говорится: «Не в знаниях сила, а в их умелом применении!»

В этом издании значительно расширены темы по технологии Quick и добавлена новая глава о программировании трехмерной графики в QML: «3D-графика Qt 3D». Остальные главы предыдущего издания также тщательно обновлены и дополнены. Все главы книги содержат новую информацию и приведены в соответствие с самой актуальной на данный момент версией библиотеки Qt — 5.10.

Завершая подготовку этого издания книги, я счел необходимым добавить в нее также совершенно новую часть с пятью главами, посвященными разработке мобильных приложений для iOS и Android, после изучения которых у вас появятся возможности и шансы применить и эти новые знания себе во благо. Без разницы: будут это ваши собственные хобби-проекты, которым вы уделите свободное от работы время и которые помогут вам заработать дополнительную копейку, или же вы, овладев применением знаний, решите стать полностью независимым от работодателя и уйти от него, всецело посвятив себя только своим, личным проектам. И сделать их источником своего основного достатка. Как это в свое время сделал я.

Помните, теперь, с этой книгой, все задуманное вами станет возможным! Явным! Поэтому для воплощения своей мечты настоятельно рекомендую — внимательно читайте каждую страницу этой книги и не забывайте святую истину: «Пища к размышлению кормит только людей сообразительных!». Ради того, чтобы предоставить вам такую возможность, я провел многие бессонные ночи в бесконечных часах работы. Фактически прописался в университетской библиотеке, которая на время подготовки этой книги, можно так сказать, стала моим вторым домом. Сделано это было с одной целью — чтобы донести до вас всю необходимую информацию.

Итак, книга у вас в руках! Поздравляю! Но это не значит, что моя работа над ней закончилась. Я постарался сделать эту книгу более открытой и интересной для общения с вами и поэтому создал для нее домен: **www.qt-book.com**. И внедрил в каждую главу книги индивидуальные ссылки на соответствующие им веб-страницы, по которым вы, дорогие читатели,

можете оставлять свои отзывы о том, что вам понравилось и что нет, а также ваши предложения и пожелания о том, чтобы вам хотелось узнать дополнительно. Словом, пишите обо всем, что заинтересовало вас после прочтения глав книги. Система обратной связи хорошо зарекомендовала себя еще со времени выхода моей первой книги — она отлично работает и помогает мне выявить дополнительные моменты, которые интересны вам, а также с вашей помощью исправить некоторые неточности. Поэтому спасибо вам за ваши отклики и большой интерес, проявленный вами к моим предыдущим книгам! За все это я вам искренне благодарен.

Пожалуйста, помните, что, несмотря на внушительный объем книги, основной ее задачей является ознакомить вас с большим спектром возможностей библиотеки Qt 5 и подтолкнуть к тому, чтобы в дальнейшем вы могли «копать глубже» и находить нужную вам информацию самостоятельно.

Мне же остается в очередной раз пожелать вам счастливого путешествия по главам моей книги. И, конечно же, счастливых открытий в познании нашей любимой библиотеки Qt.

Структура книги

Книга состоит из девяти частей. Хочу сразу обратить ваше внимание вот на что: если вы уже имели опыт программирования с предыдущей версией Qt 4, то полезнее всего начать ознакомление с книгой с материала *главы 48*, которая описывает отличия Qt 5 от Qt 4 и содержит рекомендации по внесению изменений в код для переноса существующего кода на новую версию.

Часть I. Основы Qt

Основная задача этой части — описать новый подход при программировании с использованием Qt.

- **Глава 1. Обзор иерархии классов Qt.** Глава эта вводная, она знакомит с модульной архитектурой и классами Qt, а также с реализацией первой программы, созданной с помощью Qt.
- **Глава 2. Философия объектной модели.** В эту главу входит подробное описание механизма сигналов и слотов, организация объектов в иерархии, свойства объектов.
- **Глава 3. Работа с Qt.** Эта глава описывает процесс создания проектных файлов, которые можно переработать на любой платформе в соответствующие make-файлы, методы и средства отладки приложений.
- **Глава 4. Библиотека контейнеров.** Глава содержит описание классов, которые в состоянии хранить в себе элементы различных типов данных и манипулировать ими. Здесь описываются также различные категории итераторов. Контейнерные классы в Qt являются составной частью основного модуля, и знания о них необходимы на протяжении всей книги. Эта глава содержит также описание механизма «общих данных», дающего возможность экономично и эффективно использовать ресурсы. Все контейнерные классы: списки, словари, хэш-таблицы и др. — описаны в отдельности, особое внимание уделено классу строк `QString` и мощному механизму для анализа строк, именуемому «регулярное выражение». Здесь также осуществляется знакомство с классом `QVariant`, объекты которого способны содержать в себе данные разного типа.

Часть II. Элементы управления

Задача второй части — описание элементов, из которых строятся пользовательские интерфейсы. Эта часть дает навыки грамотного и обоснованного применения таких элементов.

- **Глава 5. С чего начинаются элементы управления?** Глава вводит понятие виджета как синонима элемента управления. Описываются три класса, от которых наследуются все элементы управления, и самые важные методы этих классов, такие как изменение размера, местоположения, цвета и др. Рассказывается, как управлять из виджета изменением изображения указателя мыши. Говорится и о классе `QStackedWidget`, который способен показывать в отдельно взятый момент времени только лишь один из содержащихся в нем виджетов.
- **Глава 6. Управление автоматическим размещением элементов.** Эта глава описывает классы для размещений (Layouts), позволяющие управлять различными вариантами размещения виджетов на поверхности другого виджета, знакомит с классом разделителя `QSplitter`. В качестве примера разрабатывается программа калькулятора.
- **Глава 7. Элементы отображения.** Глава описывает элементы управления, не принимающие непосредственного участия в действиях пользователя и служащие только для отображения информации. В группу таких элементов входят надписи, индикатор выполнения и электронный индикатор. Подробно рассматриваются основные особенности этих виджетов.
- **Глава 8. Кнопки, флажки и переключатели.** В этой главе после описания основных возможностей базового класса кнопок рассматриваются следующие типы интерфейсных элементов: обычные кнопки, флажки и переключатели. Делается акцент на особенностях их применения. Описывается возможность группировки таких интерфейсных элементов.
- **Глава 9. Элементы настройки.** Глава описывает группу виджетов, позволяющих выполнять не требующие большой точности настройки: ползунков, полос прокрутки, установщиков.
- **Глава 10. Элементы ввода.** В этой главе описывается группа виджетов, представляющих собой фундамент для ввода пользовательских данных. Детально рассматривается каждый виджет этой группы: однострочные и многострочные текстовые поля, счетчик, элемент ввода даты и времени. Описано использование класса `QValidator` для предотвращения неправильного ввода пользователя.
- **Глава 11. Элементы выбора.** Глава знакомит с группой виджетов, в которую входят списки, таблицы, вкладки, инструменты и др.
- **Глава 12. Интервью, или модель-представление.** Эта глава знакомит с подходом «модель-представление» и преимуществами, связанными с его использованием.
- **Глава 13. Цветовая палитра элементов управления.** Глава описывает процесс изменения цветов как для каждого виджета в отдельности, так и для всех виджетов приложения.

Часть III. События и взаимодействие с пользователем

Цель третьей части — подробно ознакомить с тонкостями применения событий при программировании с использованием библиотеки Qt.

- **Глава 14. События.** В этой главе разъясняется необходимость сосуществования двух механизмов, связанных с оповещением: сигналов и слотов и событий. После этого сле-

дует описание целого ряда классов событий для мыши, клавиатуры, таймера и др. Отдельно рассматривается каждый метод, который предназначен для получения и обработки этих событий. Рассмотрены механизмы работы с событиями множественных касаний «мультитач».

- **Глава 15. Фильтры событий.** Глава знакомит с очень мощным механизмом, дающим возможность объекту фильтра осуществлять перехват управлением событиями. Это позволяет объектам классов, унаследованных от класса `QObject`, реализовывать, например, один класс фильтра и устанавливать его в нужные объекты, что значительно экономит время на разработку, так как отпадает необходимость наследования или изменения класса, если при этом преследуется цель только переопределить методы для обработки событий.
- **Глава 16. Искусственное создание событий.** Здесь рассказывается о способах создания события искусственным образом, что может оказаться очень полезным, например, для имитации ввода пользователя.

Часть IV. Графика и звук

Задача четвертой части — познакомить с разнообразием возможностей, связанных с программированием компьютерной графики. Затрагивается также тема реализации приложений со звуком и мультимедиаприложений.

- **Глава 17. Введение в компьютерную графику.** Глава описывает основные классы геометрии, необходимые, прежде всего, для рисования. Дается понятие цвета и палитры.
- **Глава 18. Легенда о короле Артуре и контекст рисования.** Эта глава описывает перья и кисти, отсечения, градиентные заливки и многое другое. В ней содержатся примеры рисования различных графических примитивов — от точек до полигонов, рассказывается о записи команд рисования при помощи класса `QPicture`, о трансформации систем координат и о других аспектах, связанных с рисованием.
- **Глава 19. Растровые изображения.** Глава содержит подробное описание двух классов для растровых изображений: `QPixmap` и `QImage`. Рассматриваются преимущества подобного разделения растровых изображений на два класса и функциональные возможности этих классов. Вводится понятие «прозрачность» и перечисляются поддерживаемые графические форматы.
- **Глава 20. Работа со шрифтами.** В этой главе рассматривается использование шрифтов.
- **Глава 21. Графическое представление.** Глава описывает иерархию классов `QGraphicsScene`, предоставляющих интерфейс рисования высокого уровня. Эти классы можно применять там, где необходимо дать пользователю возможность манипулировать большим количеством графических изображений, — например, в качестве спрайтов для компьютерных игр.
- **Глава 22. Анимация.** Эта глава содержит описание класса `QMovie`, предназначенного для отображения анимированных изображений в GIF- и MNG-форматах, а также описывает возможности анимационного движка, машины состояний и использование масштабируемой графики и анимации в формате SVG.
- **Глава 23. Работа с OpenGL.** Глава описывает использование библиотеки OpenGL в Qt, где OpenGL привлекается в качестве дополнительного средства для вывода трехмерной графики. Подробно рассматриваются классы Qt, созданные для поддержки OpenGL. Для полноты проводится краткое знакомство с возможностями самого OpenGL: примитива-

ми, проекциями, пикселами и изображениями, трехмерной графикой и дисплейными списками.

- **Глава 24. Вывод на печать.** Глава рассказывает о возможностях, связанных с выводом на печатающее устройство: об использовании принтера в качестве контекста рисования, о настройке параметров печати и о многом другом.
- **Глава 25. Разработка собственных элементов управления.** Глава описывает факторы, которые необходимо учитывать при создании собственных виджетов. Например, обсуждается, какой из классов взять в качестве базового, и какие методы нуждаются в переопределении.
- **Глава 26. Элементы со стилем.** Эта глава рассказывает о механизме Look & Feel, позволяющем изменять внешний вид приложения и его поведение. Глава знакомит со встроенными стилями, демонстрирует их применение, а также описывает механизм создания своих собственных стилей и использование CSS (Cascading Style Sheets) для этой цели.
- **Глава 27. Мультимедиа.** Глава знакомит с возможностями воспроизведения звука и видео, предоставляемыми модулем `QtMultimedia`.

Часть V. Создание приложений

В пятой части описываются все необходимые составляющие для реализации профессиональных приложений.

- **Глава 28. Сохранение настроек приложения.** Здесь объясняется механизм сохранения измененных пользователем настроек и их восстановления при дальнейших загрузках приложения.
- **Глава 29. Буфер обмена и перетаскивание.** В этой главе демонстрируются возможности обмена данными между разными приложениями посредством буфера обмена и перетаскивания (drag & drop).
- **Глава 30. Интернационализация приложения.** Хорошее приложение предоставляет многоязыковую поддержку, обеспечивающую его комфортное использование в различных языковых средах. *Глава 30* описывает технику, связанную с интернационализацией и локализацией создаваемых приложений.
- **Глава 31. Создание меню.** Меню — это неотъемлемая часть каждого приложения для настольных и переносных компьютеров. *Глава 31* описывает процесс создания меню разных типов: строк меню, выпадающих меню, контекстных меню, а также ускорителей, предназначенных для быстрого доступа к отдельным пунктам меню.
- **Глава 32. Диалоговые окна.** В этой главе вводятся понятия модальных и немодальных диалоговых окон. Описываются стандартные диалоговые окна для выбора файлов, шрифтов, цвета и др. Объясняется, как применять простые диалоговые окна для выдачи сообщений и как создавать собственные диалоговые окна.
- **Глава 33. Предоставление помощи.** Предоставление подсказок в программах необходимо для облегчения работы пользователя. В *главе 33* рассмотрены различные варианты помощи и методы их реализации.
- **Глава 34. Главное окно, создание SDI- и MDI-приложений.** Эта глава описывает технику создания панелей инструментов для приложений и использования строк состояния, знакомит с анатомией главного окна приложения и возможностями класса для главного

окна приложения `QMainWindow`. Приведен пример создания полноценного текстового редактора: сначала как приложения SDI (Single Document Interface), а затем MDI (Multiple Document Interface).

- **Глава 35. Рабочий стол (Desktop).** Здесь рассматриваются основные приемы использования классов работы с рабочим столом операционной системы. Операционные системы предоставляют возможности размещения значков в области уведомлений (в Windows эта область находится в нижнем правом углу) и взаимодействия пользователя с приложением из этой области. Рассматривается класс `QSystemTrayIcon`, в котором реализованы механизмы работы с областью уведомлений. Кроме того, рассматривается класс `QDesktopWidget`, предоставляющий доступ к графической области рабочего стола.

Часть VI. Особые возможности Qt

Задача шестой части — подробно ознакомить с теми возможностями Qt, которые не обязательно связаны с программированием графики и пользовательского интерфейса, но очень важны, поскольку предоставляют программисту набор функциональных возможностей практически на все случаи жизни и тем самым позволяют добиться полной платформонезависимости.

- **Глава 36. Работа с файлами, каталогами и потоками ввода/вывода.** Здесь описываются возможности, предоставляемые Qt для чтения и записи файлов, а также для просмотра каталогов и получения подробной информации о файлах. Завершается глава примером реализации программы, осуществляющей поиск файлов в заданном каталоге (папке).
- **Глава 37. Дата, время и таймер.** Эта глава описывает область назначения и применения таймеров, а также знакомит с классами, предоставляющими информацию о текущей дате и времени и методы для работы с ними.
- **Глава 38. Процессы и потоки.** Глава рассказывает о назначении процессов, описывает использование многопоточности для параллельного выполнения задач, необходимые для этого классы и методы. Рассматриваются совместное использование данных и сложности, связанные с этим. Вводятся понятия мьютекса (mutex) и задач синхронизации, а также семафора, как обобщения мьютексов. Описываются высокоуровневые классы Qt, которые упрощают реализацию многопоточных приложений.
- **Глава 39. Программирование поддержки сети.** Глава знакомит с классами, позволяющими реализовывать как TCP/UDP-клиенты, так и серверы. После этого рассматривается специализированный класс для работы с сетью на более высоком уровне: `QNetworkAccessManager`.
- **Глава 40. Работа с XML.** Эта глава содержит краткий вводный курс в очень популярный формат для описания, хранения и обмена данными XML (eXtensible Markup Language). Анализируются преимущества и недостатки различных способов представления данных XML-документа. После небольшого введения в DOM (Document Object Model) объясняется, как можно осуществлять чтение и проводить операции с узлами DOM-представления XML-документа. Говорится также о чтении при помощи SAX (Simple API for XML) и о записи XML-документов.
- **Глава 41. Программирование баз данных.** Глава содержит краткий вводный курс в базы данных. Описываются процессы соединения с базой данных и ее открытия. Подробно говорится о классе `QSqlQuery` и исполнении SQL-команд (Structured Query Lan-

guage), получении, удалении и добавлении данных. Рассматривается возможность использования классов, базирующихся на технологии «Интервью».

- **Глава 42. Динамические библиотеки.** Эта глава рассказывает, как объединить используемый различными приложениями или их частями код в отдельные динамические библиотеки. Описывается процесс создания и загрузки динамических библиотек. Кроме того, рассказывается о системе расширений (plug-ins).
- **Глава 43. Совместное использование Qt с платформозависимыми API.** Глава описывает включение платформозависимых функций ОС Windows, Mac OS X и Linux в программы, базирующиеся на библиотеке Qt.
- **Глава 44. Qt Designer. Быстрая разработка прототипов.** После небольшого описания возможностей Qt Designer производится разработка приложения средствами, предоставляемыми этой средой.
- **Глава 45. Проведение тестов.** Тестирование — это залог правильной разработки программного обеспечения. *Глава 45* знакомит с возможностями, предоставляемыми Qt для проведения модульного тестирования.
- **Глава 46. Qt WebEngine.** Глава описывает модуль `QtWebEngine`, который предоставляет инструментарий для получения и отображения информации из Всемирной паутины (WWW, World Wide Web, или просто Web). Прочитав эту главу, вы научитесь быстро создавать веб-браузеры и другие веб-клиенты.
- **Глава 47. Интегрированная среда разработки Qt Creator.** Здесь вы познакомитесь с новой интегрированной средой разработки, узнаете, какие преимущества дает ее использование, и из каких компонентов она состоит. Каждый компонент отдельно описан, особое внимание уделяется встроенному интерактивному отладчику.
- **Глава 48. Рекомендации по миграции программ из Qt 4 в Qt 5.** Эта глава призвана познакомить читателя с основными изменениями, сделанными в Qt 5, и дать рекомендации для миграции программ на Qt 5. Описываются решения, которые способны обеспечить обратную совместимость с Qt 4.

Часть VII. Язык сценариев JavaScript

Задача седьмой части — ознакомить с языком сценариев JavaScript, который базируется на стандарте ECMA Script 4.0. С предоставлением поддержки этого языка в своих программах перед разработчиком открываются расширенные возможности.

- **Глава 49. Основы поддержки сценариев JavaScript.** Эта глава объясняет принцип работы и принцип взаимодействия языка JavaScript с Qt-программами и описывает, в каких случаях и какие преимущества дает использование языка сценариев.
- **Глава 50. Синтаксис языка сценариев.** Здесь описываются ключевые слова языка сценариев и приводятся примеры их использования: объявление переменных, операции присвоения, логические операции, циклы, определение функций, определение классов и многое другое.
- **Глава 51. Встроенные объекты JavaScript.** Эта глава описывает встроенные в JavaScript объекты. К ним относятся: `Object`, `Math`, `String`, `Boolean`, `RegExp`, `Number`, `Date` и т. д.
- **Глава 52. Классы поддержки JavaScript и практические примеры.** Здесь описаны классы, необходимые разработчику программного обеспечения на языке C++ для предоставления поддержки языка сценариев в своих программах, и приводится несколько примеров, подытоживающих материал предшествующих глав.

Часть VIII. Технология Qt Quick

Задача восьмой части — познакомить с новой технологией Qt Quick, которая предоставляет язык QML для создания графического пользовательского интерфейса.

- **Глава 53. Знакомство с Qt Quick.** Глава знакомит с набором инструментов, формирующих технологию Qt Quick, раскрываются преимущества, связанные с применением этой технологии. Завершается глава созданием первого проекта, выполненного с помощью Qt Quick и с использованием языка JavaScript.
- **Глава 54. Элементы.** Эта глава описывает анатомию, типы элементов и возможности взаимодействия элементов друг с другом. А также рассматриваются возможности создания собственных модулей, элементов и использование модулей с уже готовыми элементами.
- **Глава 55. Управление размещением элементов.** Здесь объясняется отличие привычных методов размещения элементов в библиотеке Qt при помощи классов размещения (layout) от нового подхода фиксации, используемого в языке QML. Кроме того, показаны различные методы использования техники фиксации на примерах.
- **Глава 56. Элементы графики.** В этой главе приведено описание возможности использования элементов растровых изображений, градиентов, шрифтов, цвета, рисования на элементах холста и использования эффектов шейдеров.
- **Глава 57. Пользовательский ввод.** Глава раскрывает возможности элементов, предназначенных для работы пользователя с клавиатурой и мышью, описывает механизмы использования сигналов, свойств их обработки, а также возможность использования механизмов обработки множественных касаний «мультитач».
- **Глава 58. Анимация.** В главе рассматриваются основные типы анимаций, свойств, поворота, поведения и т. п. Кроме того, рассмотрены последовательные и параллельные анимации, применение смягчающих линий, использование состояний и переходов и система частиц для создания анимаций с участием большого количества элементов.
- **Глава 59. Модель/Представление.** Здесь рассматриваются элементы отображения различных моделей данных, реализация делегатов и самих моделей данных.
- **Глава 60. Qt Quick и C++.** Эта глава посвящена внедрению компонентов, разработанных на языке C++, в язык QML (в технологию Qt Quick) и в противоположном направлении.
- **Глава 61. 3D-графика Qt 3D.** Глава представляет собой введение в создание трехмерной графики с помощью модулей группы Qt 3D. Рассматриваются источники света, камеры, анатомия 3D-объектов, их стандартные формы, а также использование материалов, проведение трансформаций и анимации 3D-объектов.

Часть IX. Мобильные приложения и Qt

Задача девятой части — снабдить вас всей необходимой информацией для создания мобильных приложений под платформы iOS и Android. Прочитав и овладев навыками этой части, вы научитесь воплощать свои идеи в продукты, которые могут быть доступны для огромной аудитории пользователей мобильных устройств.

- **Глава 62. Введение в мир мобильных приложений.** Эта глава проводит небольшой экскурс в историю появления такого явления, как смартфоны. Знакомит с самыми распространенными для платформ iOS и Android магазинами приложений и их возможно-

стями. А так же содержит информацию об особенностях использования Qt в разработке мобильных приложений.

- **Глава 63. Подготовка к работе над мобильными приложениями.** В этой главе мы установим и настроим на компьютере все необходимое, чтобы начать разрабатывать приложения для iOS и Android.
- **Глава 64. Особенности разработки приложений для мобильных устройств.** Подходы к разработке приложений для мобильных устройств отличаются от подходов к разработке приложений для настольных и переносных компьютеров. Эта глава делает акцент на существующей разнице и дает информацию о том, что необходимо учитывать при разработке мобильных приложений.
- **Глава 65. Пример разработки мобильного приложения.** В этой главе мы займемся разработкой примера реального приложения, в котором будут реализованы некоторые из описанных ранее подходов и технологий, присущих мобильным устройствам.
- **Глава 66. Публикация в магазине мобильных приложений.** В завершающей главе книги показано, что необходимо сделать, чтобы опубликовать приложение из *главы 65* в магазинах App Store и Google Play.

Приложения

Книга включает четыре приложения. В *приложении 1* описаны шаги, которые необходимо проделать для установки Qt на ОС Windows, Mac OS X и Linux (Ubunty), в *приложении 2* приведена таблица простых чисел, в *приложении 3* содержится таблица ASCII-кодировки, а в *приложении 4* — описание электронного архива с примерами.

Электронный архив

Электронный архив с примерами к этой книге можно скачать с FTP-сервера издательства «БХВ-Петербург» по ссылке **ftp://ftp.bhv.ru/9785977536783.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение 4*).

Введение

Путешествие в тысячу миль начинается с первого шага.
Древняя китайская мудрость

Сегодня практически невозможно представить себе приложение, не обладающее интерфейсом пользователя. Понятия *Software* (программный продукт), *Apps* (приложения) и *GUI* (Graphical User Interface, графический интерфейс пользователя) неразрывно связаны друг с другом.

Хотя каждая из операционных систем обладает всем необходимым для создания графического интерфейса пользователя, использование этих доступных «инструментов» требует больших затрат времени и практического опыта. Даже библиотеки, призванные облегчить процесс написания программ, не дают процессу создания программ и приложений той простоты и легкости, какой хотелось бы. Поэтому и сегодня разработчики по-прежнему тратят массу времени на реализацию интерфейса пользователя. Но самый большой недостаток, связанный с применением таких библиотек, — это платформозависимость.

В самом деле, если вы программируете только для какой-то одной конкретной операционной системы, например ОС Windows, то у вас, наверняка, возникнет вопрос — зачем мне испытывать что-то новое? И одна из причин, почему это все же стоит сделать, — реализация платформонезависимых приложений. Платформонезависимая реализация приложений — это будущее программной индустрии. С каждый днем она будет приобретать все более возрастающее значение. Только задумайтесь — зачем оставлять без внимания пользователей Mac OS X или мобильных устройств, базирующихся на Android, только лишь потому, что вы являетесь программистом для ОС Windows? Позволив своему приложению работать под разными ОС, вы заметно увеличите количество пользователей (клиентов). Выигрыш же от реализации платформонезависимых приложений налицо: значительно сокращается время разработки, поскольку вам не приходится писать код многократно — под каждую платформу, и, что не менее важно, отпадает необходимость знать нюансы каждой из платформ, для которой пишется программа. Не понадобится также во время разработки продукта формировать специальные подкоманды разработчиков для каждой платформы реализации — все это может значительно сократить не только время разработки, но и себестоимость вашего продукта. Вы сможете использовать самые передовые инструменты для отладки и совершенствования, улучшения кода программ — например, абсолютно бесплатную интегрированную среду разработки XCode для Mac OS X.

И вместе с тем заметно улучшится и качество ваших приложений, так как они будут тестироваться на нескольких платформах, а ошибки исправляться в одном и том же исходном коде программы.

Qt — это луч надежды для программистов, пишущих на языке C++, которые вынуждены сейчас выполнять тройную работу по реализации своих приложений для ОС Windows, Linux и Mac OS X. Выбор в пользу Qt избавит вас от этих проблем. Qt предоставляет поддержку большого числа операционных систем: Microsoft Windows, Mac OS X, Linux, FreeBSD и других клонов UNIX с X11, а также и для мобильных операционных систем iOS, Android, Windows Phone, Windows RT и BlackBerry. Более того, благодаря встраиваемому пакету Qt Embedded все возможности Qt доступны также и в интегрированных системах (Embedded Systems). Qt использует интерфейс API низкого уровня, что позволяет кроссплатформенным приложениям работать столь же эффективно, как и приложениям, разработанным специально для конкретной платформы.

Несмотря на то, что предоставляемая платформонезависимость является одной из самых заманчивых возможностей библиотеки, многие разработчики используют Qt и для создания приложений, работающих только на одной платформе. Делают они это из тех соображений, что им нравится инструментарий и идейный подход самой библиотеки, который предоставляет им дополнительную гибкость и скорость разработки. А учитывая, что требования к программному продукту с течением времени постоянно подвергаются изменениям, не составит большой сложности при появлении необходимости предоставить продукт и для какой-либо иной платформы.

Использование в разработке разных компиляторов C++ еще больше повышает правильность и надежность кода ваших программ, поскольку предупреждающие сообщения и сообщения об ошибках вы станете получать от разных компиляторов, что будет делать код вашей программы с каждым разом все более совершенным.

Для ускорения и упрощения создания пользовательских интерфейсов Qt предоставляет программу Qt Designer, позволяющую делать это в интерактивном режиме. Очень сильно повысить скорость создания пользовательских интерфейсов можно также и при помощи технологии Qt Quick с описательным языком QML, модули и инструменты которой являются неотъемлемой частью Qt.

На сегодняшний день Qt — это продукт, широко используемый разработчиками всего мира. Компаний, ориентированных на эту библиотеку, более четырех тысяч. В число активных пользователей Qt входят такие компании, как: Adobe, Amazon, AMD, Bosch, BMW, Blackberry, Canon, Cisco Systems, Disney, Intel, IBM, Panasonic, Parallels, Pioneer, Philips, Oracle, HP, Goober, Google, Mercedes, NASA, NEC, Neonway, Nokia, Rakuten, Samsung, Siemens, Sony, SUN, Tesla, Xerox, Xilinx, Yamaha и др.

Используя сегодня ту или иную программу, вы, возможно, даже и не догадываетесь, что при ее написании была задействована библиотека Qt. Приведу лишь несколько, на мой взгляд, самых ярких примеров:

- рабочий стол KDE Software Compilation 4 (**www.kde.org**), используемый в Linux и FreeBSD (рис. В.1);
- редактор трехмерной графики Autodesk Maya (**www.autodesk.com**) (рис. В.2);
- приложение Viber (**www.viber.com**) компании Rakuten, предназначенное для голосовой связи VoIP (Voice Over IP) — звонков на обычные телефоны через Интернет (рис. В.3);
- мессенджер Telegram (**www.telegram.org**) от Telegram LLP, предназначенный для обмена сообщениями и файлами через Интернет, а также и для голосовой связи VoIP (рис. В.4);
- программа Adobe Photoshop Album (**www.adobe.com**) для обработки растровых изображений (рис. В.5);
- сетевая карта мира Google Earth (**earth.google.com**), которая позволяет рассматривать интересующие нас участки поверхности нашей планеты с высоты до 200 м (рис. В.6);

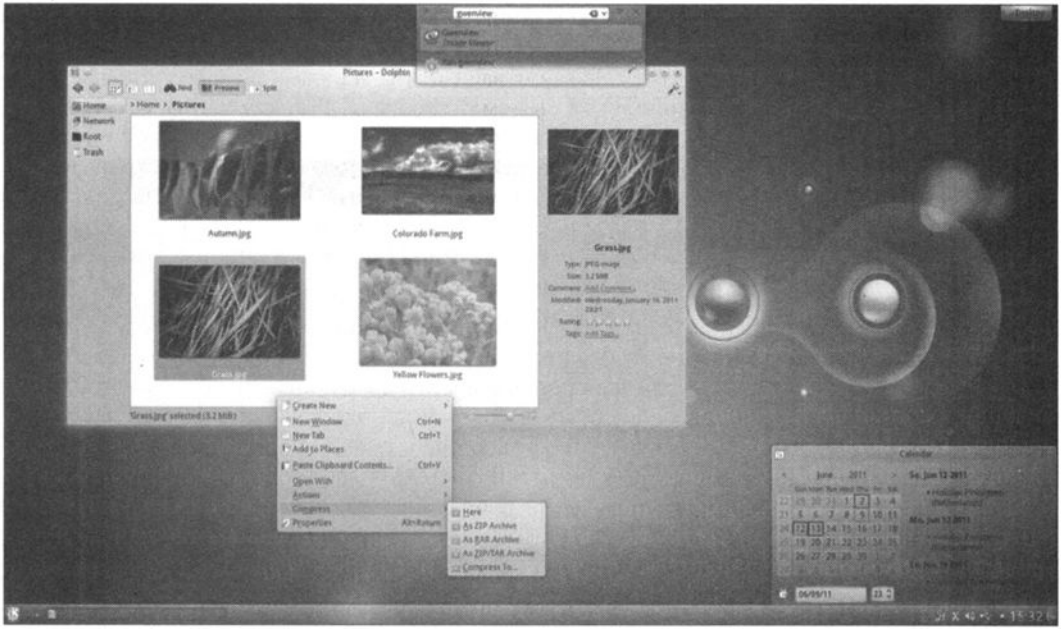


Рис. В.1. KDE Software Compilation 4 (взято с сайта **www.wikipedia.org**)

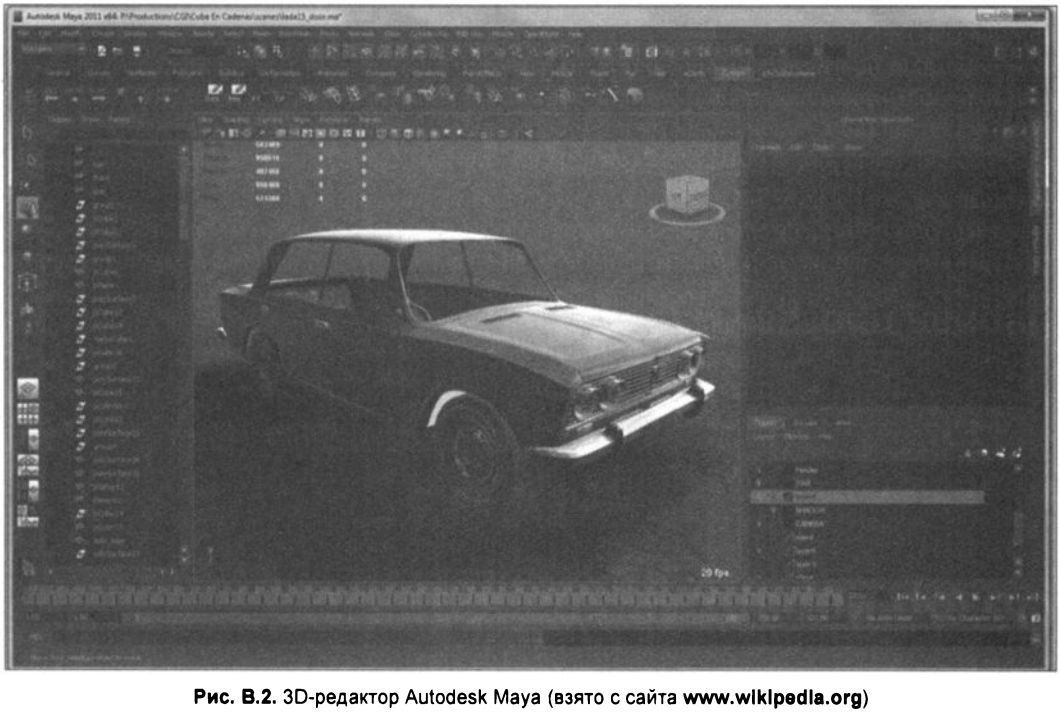


Рис. В.2. 3D-редактор Autodesk Maya (взято с сайта **www.wikipedia.org**)

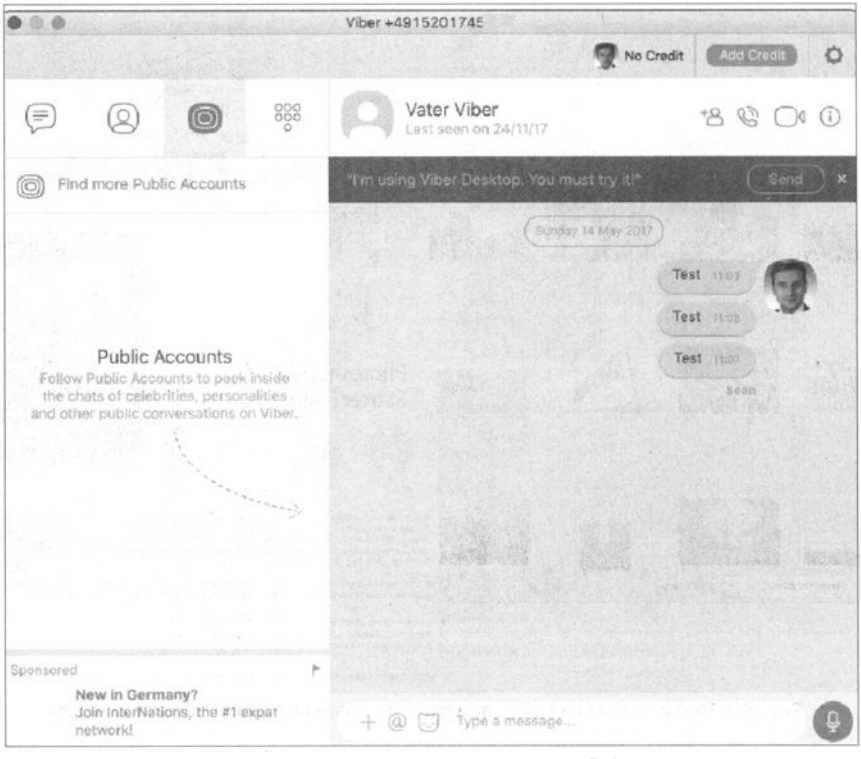


Рис. В.3. Приложение Viber от компании Rakuten

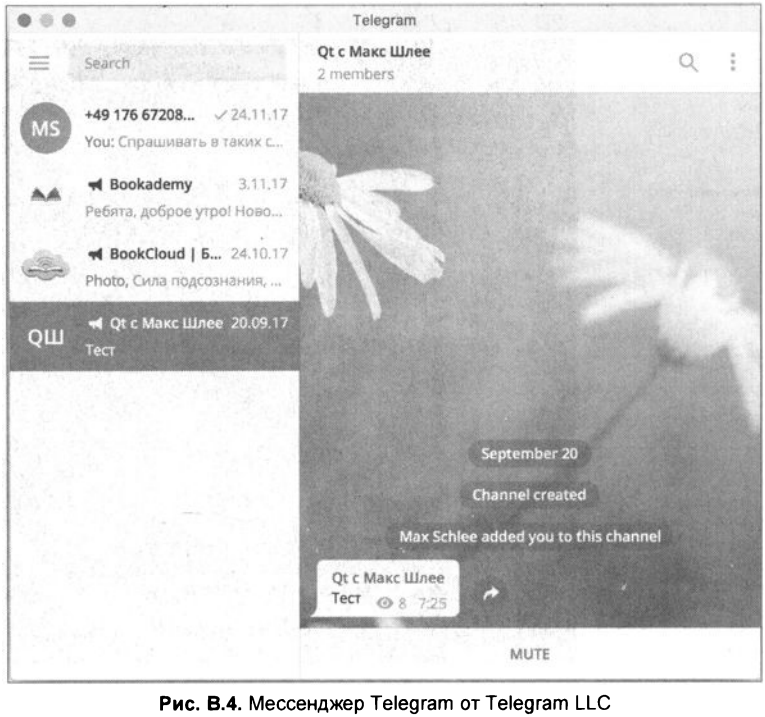


Рис. В.4. Мессенджер Telegram от Telegram LLC



Рис. В.5. Программа обработки растровых изображений Adobe Photoshop Album

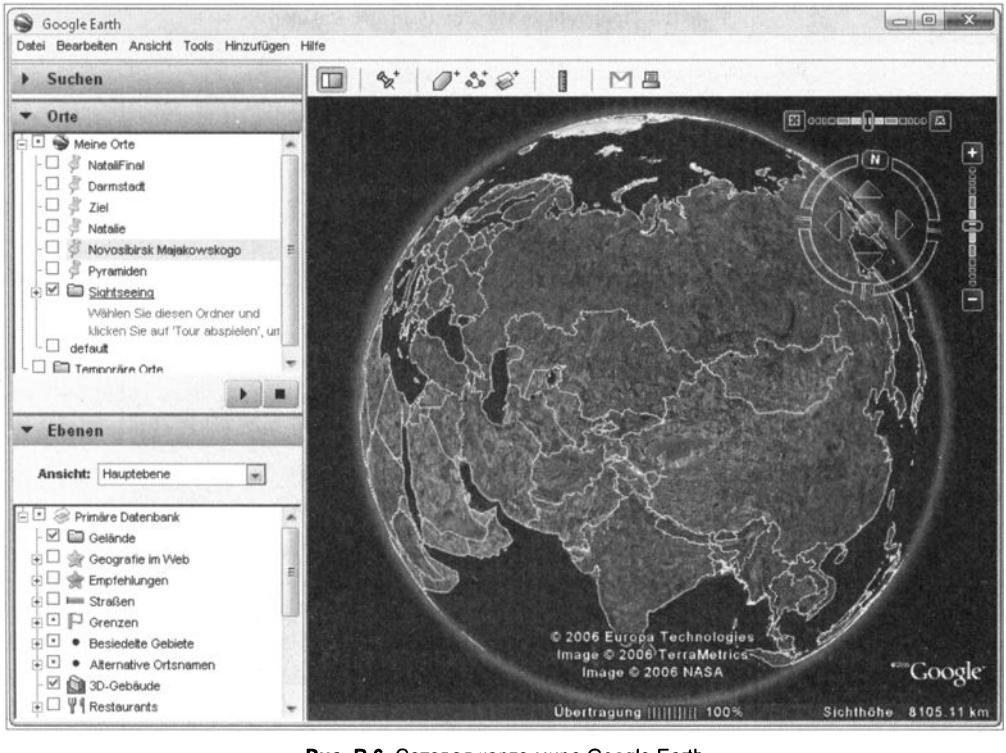


Рис. В.6. Сетевая карта мира Google Earth

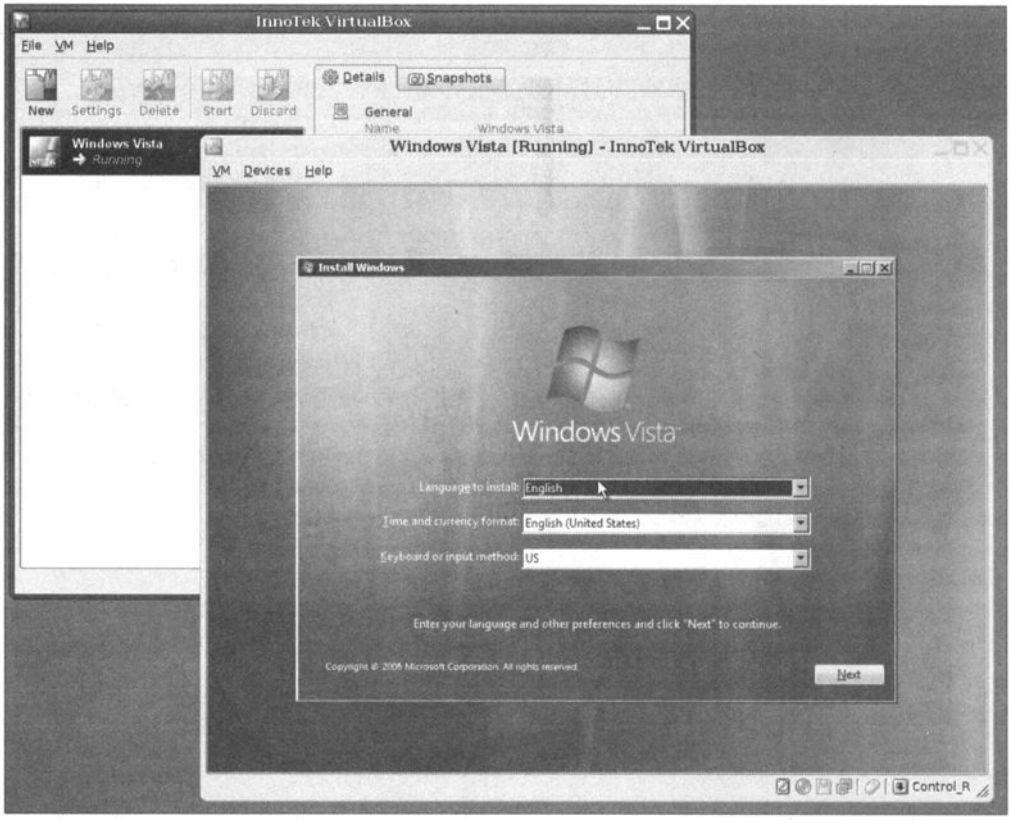


Рис. В.7. Эмулятор VirtualBox (взято с сайта **www.virtualbox.org**)

- программа для виртуализации операционных систем VirtualBox (**www.virtualbox.org**) от Sun Microsystems (рис. В.7);
- свободный проигрыватель VLC media player (**www.videolan.org/vlc/**), начиная с версии 0.9 (рис. В.8);
- программа для виртуализации операционных систем Parallels (**www.parallels.com**) от компании Parallels (рис. В.9);
- программа Kindle (рис. В.10) от компании Amazon (**www.amazon.com**), разработанная для загрузки, просмотра и чтения электронных книг, газет и журналов, купленных в магазине Kindle-Shop;
- программы официальных клиентов виртуальных валют Bitcoin (**www.bitcoin.org**) (рис. В.11) и Litecoin (**www.litecoin.org**).

Я не только пишу о библиотеке Qt, но и весьма интенсивно использую ее сам. За последние годы с моим личным участием было реализовано на Qt уже более 50 действующих проектов приложений, которые можно найти на странице моей компании и других компаний, в которых я когда-то работал, а также на Apple App Store, Google Play, Amazon Appstore и BlackBerry World. В связи с этим упомяну некоторые из таких Qt-проектов:

- программа PhotoGUN (**www.neonway.ru/photogun/**), предназначенная для пакетной обработки фотографий (рис. В.12);

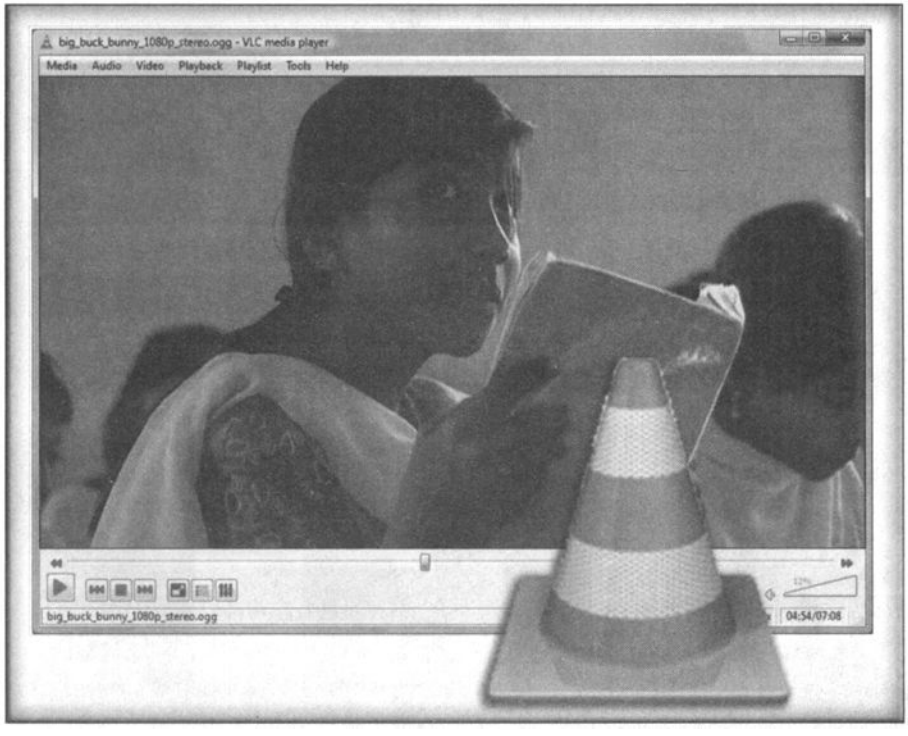


Рис. В.8. Проигрыватель VLC media player



Рис. В.9. Эмулятор Parallels

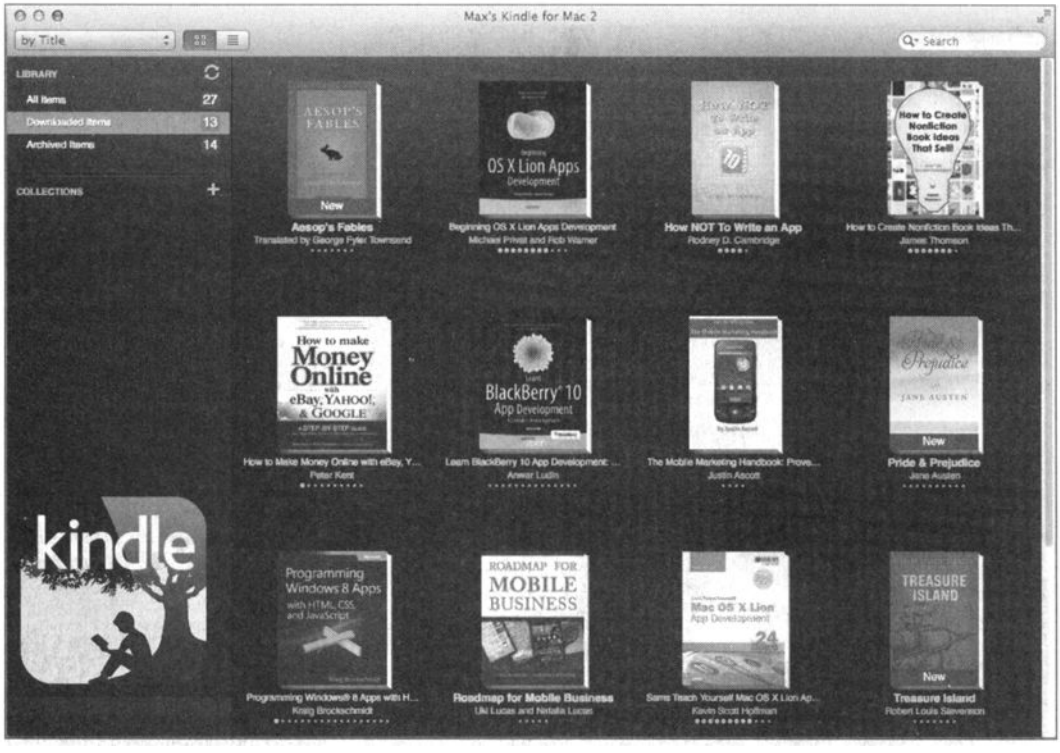


Рис. В.10. Программа Amazon Kindle

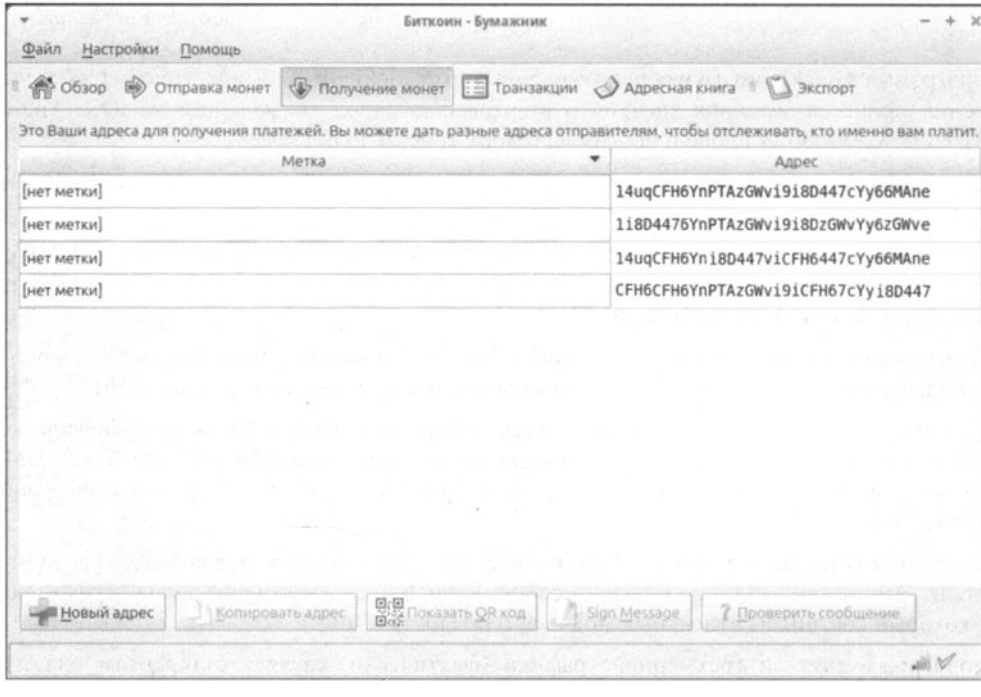


Рис. В.11. Клиент для виртуальной валюты Bitcoin

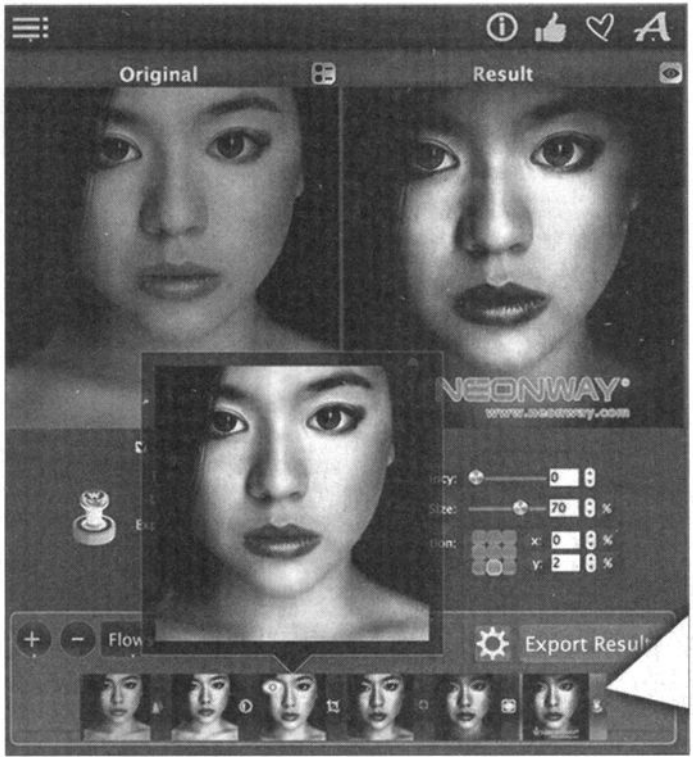


Рис. В.12. Программа PhotoGUN от Neonway



Рис. В.13. Программа QuickIcons от Neonway

- программа QuickIcons (**www.neonway.com/apps/quickicons/**) — для упрощения и ускорения процесса создания значков и стартовых экранов приложений на iOS, Android, Mac OS X, Windows Phone и других платформ (рис. В.13);
- ChordsMaestro (рис. В.14) — программа для обучения игре на семи музыкальных инструментах (**www.neonway.ru/chordsmaestro/**);
- программа FLEXXITY (**www.dft-film.com/archive/flexxity_archive.php**) от DigitalFilmTechnology Weiterstadt, предназначенная для редактирования, применения эффектов и реставрации видеоматериала (рис. В.15);
- программа ChatCube от Goober Networks, Inc. — для обеспечения текстовой, голосовой и видеосвязи через Интернет между компьютерами и смартфонами (рис. В.16).

В настоящее время Qt все чаще задействуется в разработке графических интерфейсов пользователя для автомобилей. Его использовали такие знаменитые фирмы, как Tesla, BMW, Mercedes и др. На рис. В.17 показана модель «Мерседес», представленная на конференции Qt World Summit 2017.

Многие привыкли считать, что Qt — лишь средство для создания только интерфейса пользователя. Это не так — Qt представляет собой полный инструментарий для программирования, который состоит из отдельных модулей и предоставляет:

- поддержку двух- и трехмерной графики (фактически, являясь стандартом для платформонезависимого программирования на OpenGL), а также имеет свою собственную альтернативную реализацию и собственный модуль Qt 3D;

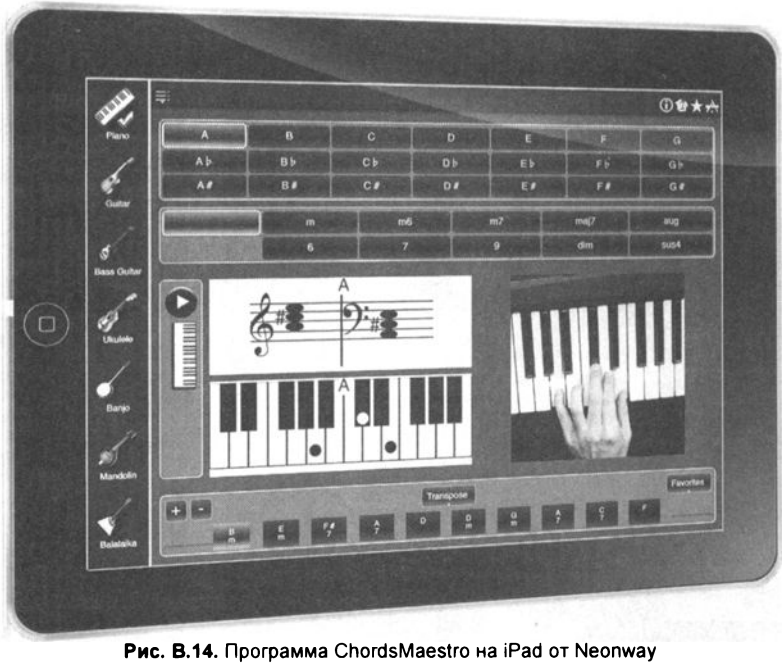


Рис. В.14. Программа ChordsMaestro на iPad от Neonway

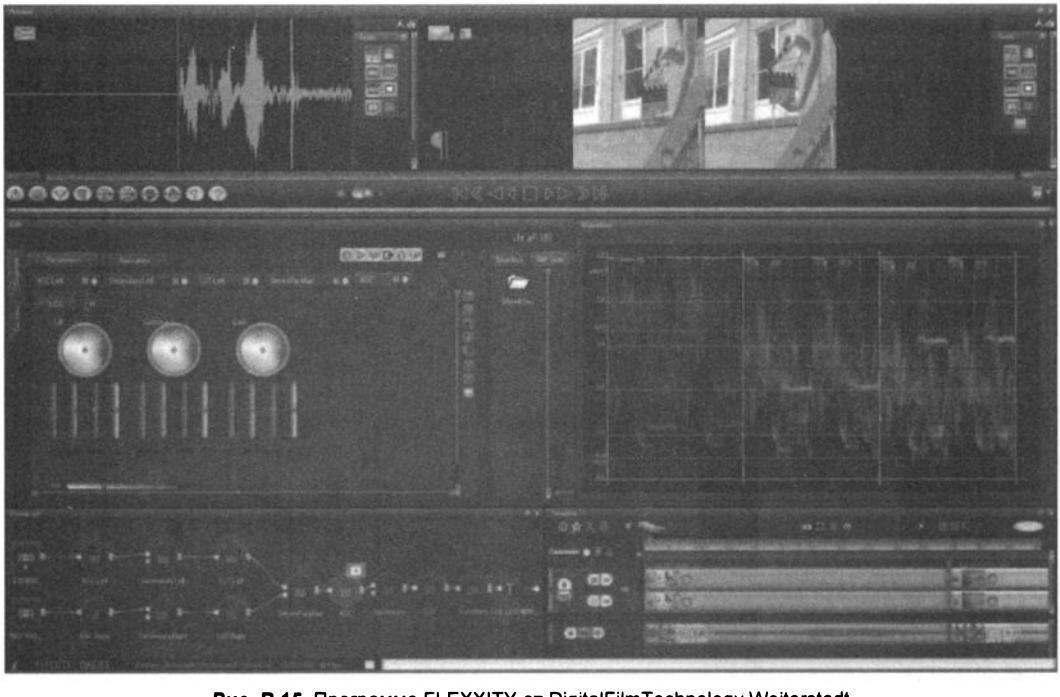


Рис. В.15. Программа FLEXXITY от DigitalFilmTechnology Weiterstadt



Рис. В.16. Программа ChatCube от Goober Networks, Inc.

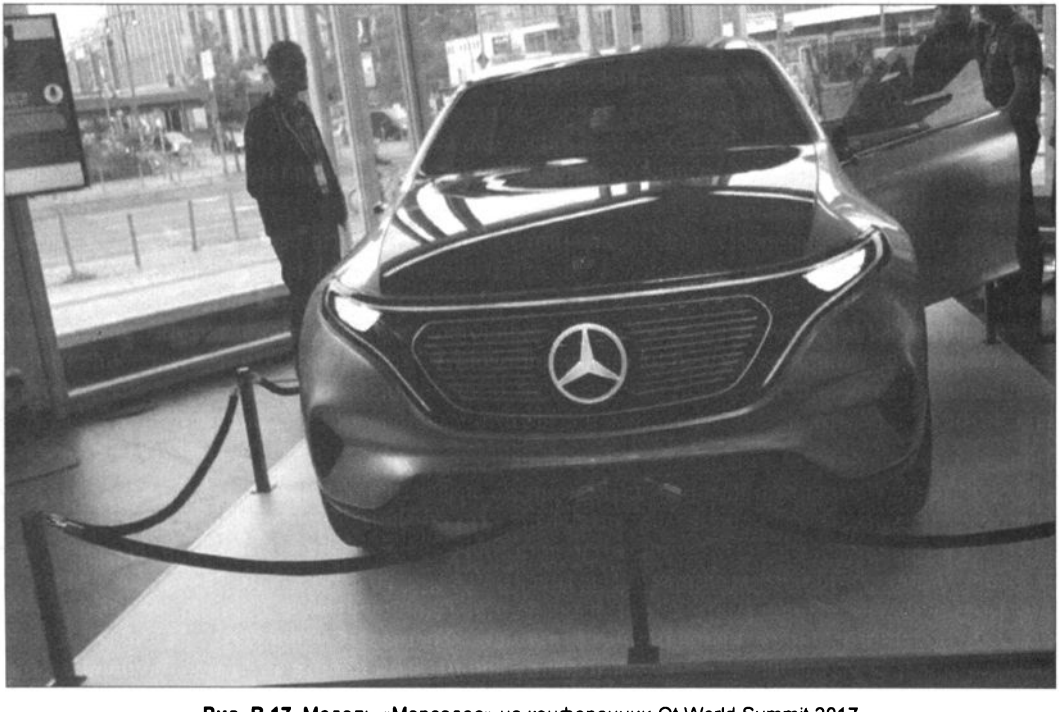


Рис. В.17. Модель «Мерседес» на конференции Qt World Summit 2017

- возможность интернационализации, которая позволяет значительно расширить рынок сбыта ваших программ;
- использование форматов JSON (JavaScript Object Notation) и XML (eXtensible Markup Language);
- STL-совместимую библиотеку контейнеров;
- поддержку стандартных протоколов ввода/вывода;
- классы для работы с сетью;
- поддержку программирования баз данных, включая Oracle, Microsoft SQL Server, IBM DB2, MySQL, SQLite, Sybase, PostgreSQL;
- и многое другое.

Qt — полностью объектно-ориентированная библиотека. Новая концепция ведения межобъектных коммуникаций, именуемая «сигналы и слоты», полностью заменяет применявшуюся ранее не вполне надежную модель обратных вызовов. Имеется также возможность обработки событий — например, нажатия клавиш клавиатуры, перемещения мыши и т. д.

Предоставляемая система расширений (plug-ins) позволяет создавать модули, расширяющие функциональные возможности создаваемых приложений. Эти расширения пользователи вашей программы могут получать не только от вас, но и от других разработчиков.

Несмотря на то, что библиотека Qt изначально создавалась для языка программирования C++, это вовсе не означает, что ее использование невозможно в других языках. Напротив, во многих языках программирования существуют модули для работы с этой библиотекой — например: Qt# в C#, PerlQt в Perl, PyQt в Python, PHP, Ruby и т. д.

Программы, реализованные с помощью Qt, могут использовать язык сценариев JavaScript. Эта технология позволяет пользователям вашего приложения расширить его возможности без изменения исходного кода и без перекомпоновки самого приложения изменить «поведение» приложения.

Qt прекрасно документирована, благодаря чему с помощью программы Qt Asisstant вы всегда можете почерпнуть о ней любую интересующую вас информацию. А если и этого окажется недостаточно, то не забывайте, что Qt — библиотека с открытым исходным кодом (Open Source), и вы всегда можете взглянуть в него и детально разобраться в том, как работает та или иная часть этой библиотеки.

И если быть предельно кратким, то библиотеку Qt можно охарактеризовать в трех словах: Простота + Быстрота + Мощность.

Добро пожаловать в мир Qt 5!

Макс Шлее
Дармштадт 9 декабря 2017 г.



ЧАСТЬ I

Основы Qt

Вы не обязаны быть великим, чтобы начать, но обязаны начать, чтобы стать великим.

Джо Сабах

Глава 1. Обзор иерархии классов Qt

Глава 2. Философия объектной модели

Глава 3. Работа с Qt

Глава 4. Библиотека контейнеров

ГЛАВА 1

Обзор иерархии классов Qt

Если вы хотите знать территорию — нужно сначала изучить карту.

Тони Бьюзен

Первая программа на Qt

Для того чтобы написать и запустить первую программу на Qt, нужно обязательно установить саму библиотеку Qt и все необходимое для ее работы. И если вы этого еще не сделали, то о том, как это сделать, рассказано в *приложении 1* к этой книге.

Итак, теперь у вас для работы все готово! И это значит, что, как и заведено в самом начале знакомства, настало, наконец, время поздороваться, и, чтобы никого не оставить без внимания, мы обратимся, не больше и не меньше, а сразу ко всему миру. Давайте для этого напишем короткую программу «Hello, World» («Здравствуй, Мир»), результат выполнения которой показан на рис. 1.1.

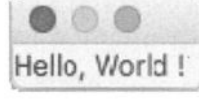


Рис. 1.1. Окно программы «Hello, World»

Написание подобного рода программ стало уже традицией при знакомстве с новым языком или библиотекой. И хотя такой пример не в состоянии продемонстрировать весь потенциал и возможности самой библиотеки, он дает представление о базовых понятиях и позволяет оценить объем и сложность процесса реализации программ, использующих ту или иную библиотеку. Кроме того, на этом примере можно убедиться, что все необходимое для компиляции и компоновки установлено правильно.

Листинг 1.1. Программа «Hello, World» (файл hello.cpp)

```
#include <QtWidgets>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QLabel lbl("Hello, World !");
    lbl.show();
    return app.exec();
}
```

Электронный архив

Напомню, что электронный архив с примерами к этой книге можно скачать с FTP-сервера издательства «БХВ-Петербург» по ссылке **ftp://ftp.bhv.ru/9785977536783.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение 4*). Файлы, упомянутые в названиях листингов, находятся в папках с номерами соответствующих глав.

В первой строке листинга 1.1 подключается заголовочный файл `QtWidgets`, представляющий собой модуль, включающий в себя заголовочные файлы для используемых в нашей программе классов: `QApplication` и `QLabel`. Конечно, мы могли бы обойтись и без модуля `QtWidgets`, а непосредственно подключить заголовочные файлы для поддержки классов `QApplication` и `QLabel`, но при большом количестве включаемых классов, задействованных в программе, читаемость самой программы заметно ухудшается. Кроме того, подключение заголовочного модуля `QtWidgets` ускоряет саму работу с кодом и, благодаря механизму предварительной компиляции заголовочных файлов (Precompiled Headers), не должно отразиться на скорости компиляции самой программы — конечно в том случае, если ваш компилятор этот механизм поддерживает.

Теперь давайте рассмотрим наш пример подробнее. В нем сначала создается объект класса `QApplication`, который и осуществляет управление приложением. Для его создания в конструктор этого класса необходимо передать два аргумента. Первый аргумент представляет собой информацию о количестве аргументов в командной строке, из которой происходит обращение к программе, а второй — это указатель на массив символьных строк, содержащих аргументы, по одному в строке. Любая использующая Qt-программа с графическим интерфейсом должна создавать только один объект этого класса, и он должен быть создан до использования операций, связанных с пользовательским интерфейсом.

Затем создается объект класса `QLabel`. После создания элементы управления Qt по умолчанию невидимы, и для их отображения необходимо вызвать метод `show()`. Объект класса `QLabel` является *основным управляющим элементом приложения*, что позволяет завершить работу приложения при закрытии окна элемента. Если вдруг окажется, что в созданном приложении имеется сразу несколько независимых друг от друга элементов управления, то при закрытии окна последнего такого элемента управления завершится и само приложение. Это правильно, иначе приложение осталось бы в памяти компьютера и расходовало бы его ресурсы.

Наконец, в последней строке программы приложение запускается вызовом `QApplication::exec()`. С его запуском приводится в действие цикл обработки событий, определенный в классе `QCoreApplication`, являющемся базовым для `QGuiApplication`, от которого унаследован класс `QApplication`. Этот цикл передает получаемые от системы события на обработку соответствующим объектам. Он продолжается до тех пор, пока либо не будет вызван статический метод `QCoreApplication::exit()`, либо не закроется окно последнего элемента управления. По завершении работы приложения метод `QApplication::exec()` возвращает значение целого типа, содержащее код, информирующий о его завершении.

Модули Qt

У программистов, начинающих изучение классов новой библиотеки, из-за большого объема информации, которую надо усвоить, зачастую создается ощущение перенасыщения. Но иерархия классов `Qt` имеет четкую внутреннюю структуру, которую важно сразу понять, чтобы потом уметь хорошо и интуитивно в этой библиотеке ориентироваться.

Библиотека Qt — это множество классов (около 1500), которые охватывают большую часть функциональных возможностей операционных систем, предоставляя разработчику мощные

механизмы, расширяющие и вместе с тем упрощающие разработку приложений. При этом не нарушается идеология операционной системы. Qt не является единым целым — она разбита на модули (табл. 1.1).

Таблица 1.1. *Некоторые модули Qt*

Библиотека	Обозначение в проектном файле	Назначение
`QtCore`	`core`	Основополагающий модуль, состоящий из классов, не связанных с графическим интерфейсом (см. *части I, IV*)
`QtGui`	`gui`	Модуль базовых классов для программирования графического интерфейса
`QtWidgets`	`widgets`	Модуль, дополняющий `QtGui` «строительным материалом» для графического интерфейса в виде виджетов на C++ (см. *части II, III, IV, V*)
`QtQuick`	`quick`	Модуль, содержащий описательный фреймворк для быстрого создания графического интерфейса (см. *часть VIII*)
`QtQML`	`qml`	Модуль, содержащий движок для языка QML и JavaScript (см. *часть VIII*)
`QtNetwork`	`network`	Модуль для программирования сети (см. *главу 39*)
`QtSql`	`sql`	Модуль для программирования баз данных (см. *главу 41*)
`QtSvg`	`svg`	Модуль для работы с SVG (Scalable Vector Graphics, масштабируемая векторная графика) (см. *главу 22*)
`QtXml`	`xml`	Модуль поддержки XML, классы, относящиеся к SAX и DOM (см. *главу 40*)
`QtXmlPatterns`	`xmlpatterns`	Модуль поддержки XPath, XQuery, XSLT и XmlShemaValidator (см. *главу 40*)
`QtMultimedia`	`multimedia`	Модуль мультимедиа. Собрание классов для работы со звуком, видео, камерой и радио (см. *главу 27*)
`QtMultimediaWidgets`	`multimediawidgets`	Модуль с виджетами для модуля `QtMultimedia` (см. *главу 27*)
`QPrintSupport`	`printsupport`	Модуль для работы с принтером (см. *главу 24*)
`QtTest`	`test`	Модуль, содержащий классы для тестирования кода (см. *главу 45*)

Любая Qt-программа так или иначе должна использовать хотя бы один из модулей нашего примера из листинга 1.1 — это три модуля: `QtCore`, `QtGui` и `QtWidgets`, они присутствуют во всех программах с графическим интерфейсом и поэтому определены в программе создания make-файлов (см. *главу 3*) по умолчанию. Для использования других модулей в своих проектах необходимо перечислить их в проектном файле (см. *главу 3*). Например, чтобы добавить модули, нужно написать:

```
QT += widgets network sql
```

А чтобы исключить модуль из проекта:

```
QT -= gui
```

Наиболее значимый из приведенных в табл. 1.1 модулей — это `QtCore`, так как он является базовым для всех остальных модулей. Далее идут модули, которые непосредственно зависят от `QtCore`, это: `QtNetwork`, `QtGui`, `QtSql` и `QtXml`.

Для каждого модуля Qt предоставляет отдельный заголовочный файл, содержащий заголовочные файлы всех классов этого модуля. Название такого заголовочного файла соответствует названию самого модуля. Например, для включения модуля `QtWidgets` нужно добавить в программу строку, как мы это уже сделали в листинге 1.1:

```
#include <QtWidgets>
```

Пространство имен Qt

Пространство имен Qt содержит ряд типов перечислений и констант, которые часто применяются при программировании. Если вам необходимо получить доступ к какой-либо константе этого пространства имен, то вы должны указать префикс `Qt` (например, не `red`, а `Qt::red`). Если вы все-таки хотите опускать префикс `Qt`, то необходимо в начале файла с исходным кодом добавить следующую директиву:

```
using namespace Qt;
```

Модуль *QtCore*

Как уже было сказано ранее, базовым является модуль `QtCore`. При этом он является базовым для приложений и не содержит классов, относящихся к интерфейсу пользователя. Если вы собираетесь реализовать консольное приложение, то, вполне возможно, можете ограничиться одним этим модулем. В модуль `QtCore` входят более 200 классов, вот некоторые из них:

- контейнерные классы: `QList`, `QVector`, `QMap`, `QVariant`, `QString` и т. д. (см. *главу 4*);
- классы для ввода и вывода: `QIODevice`, `QTextStream`, `QFile` (см. *главу 36*);
- классы процесса `QProcess` и для программирования многопоточности: `QThread`, `QWaitCondition`, `QMutex` (см. *главу 38*);
- классы для работы с таймером: `QBasicTimer` и `QTimer` (см. *главу 37*);
- классы для работы с датой и временем: `QDate` и `QTime` (см. *главу 37*);
- класс `QObject`, являющийся *краеугольным камнем* объектной модели Qt (см. *главу 2*);
- базовый класс событий `QEvent` (см. *главу 14*);
- класс для сохранения настроек приложения `QSettings` (см. *главу 28*);
- класс приложения `QCoreApplication`, из объекта которого, если требуется, можно запустить цикл событий;
- классы поддержки анимации: `QAbstractAnimation`, `QVariantAnimation` и т. д. (см. *главу 22*);
- классы для машины состояний: `QStateMachine`, `QState` и т. д. (см. *главу 22*);
- классы моделей интервью: `QAbstractItemModel`, `QStringListModel`, `QAbstractProxyModel` (см. *главу 12*).

Модуль содержит также механизмы поддержки файлов ресурсов (см. *главу 3*).

Давайте немного остановимся на классе `QCoreApplication`. Объект класса приложения `QCoreApplication` можно образно сравнить с сосудом, содержащим объекты, подсоединенные к контексту операционной системы. Срок жизни объекта класса `QCoreApplication` соответствует продолжительности работы всего приложения, и он остается доступным в любой момент работы программы. Объект класса `QCoreApplication` должен создаваться в приложении только один раз. К задачам этого объекта можно отнести:

- управление событиями между приложением и операционной системой;
- передачу и предоставление аргументов командной строки.

Кроме того, `QCoreApplication` можно унаследовать, чтобы перезаписать некоторые методы, а также задействовать сам объект для дополнительных глобальных данных, используемых внутри приложения. Такой подход может избавить вас от нежелательного использования шаблона проектирования Singleton.

Модуль *QtGui*

Этот модуль предоставляет классы интеграции с оконной системой, с OpenGL и OpenGL ES. Он содержит класс `QWindow`, который является элементарной областью с возможностью получения событий пользовательского ввода, изменения фокуса и размеров, а так же позволяющий производить графические операции и рисование на своей поверхности.

Класс приложения этого модуля — `QGuiApplication`. Он содержит механизм цикла событий и обладает так же возможностями:

- получения доступа к буферу обмена (см. *главу 29*);
- инициализации необходимых настроек приложения — например, палитры для расцветки элементов управления (см. *главу 13*);
- управления формой курсора мыши.

Модуль *QtWidgets*

Этот модуль содержит около 300 классов виджетов, представляющих собой «строительный материал» для программирования графического интерфейса пользователя. Вот некоторые из них:

- класс `QWidget` — это базовый класс для всех элементов управления библиотеки Qt. По своему внешнему виду он не что иное, как заполненный четырехугольник, но за этой внешней простотой скрывается большой потенциал непростых функциональных возможностей. Этот класс насчитывает 254 метода и 53 свойства. В *главе 5* ему уделено особое внимание;
- классы для автоматического размещения элементов: `QVBoxLayout`, `QHBoxLayout` (см. *главу 6*);
- классы элементов отображения: `QLabel`, `QLCDNumber` (см. *главу 7*);
- классы кнопок: `QPushButton`, `QCheckBox`, `QRadioButton` (см. *главу 8*);
- классы элементов установок: `QSlider`, `QScrollBar` (см. *главу 9*);
- классы элементов ввода: `QLineEdit`, `QSpinBox` (см. *главу 10*);
- классы элементов выбора: `QComboBox`, `QToolBox` (см. *главу 11*);
- классы меню: `QMainWindow` и `QMenu` (см. *главы 31* и *34*);

- классы окон сообщений и диалоговых окон: `QMessageBox`, `QDialog` (см. *главу 32*);
- классы для рисования: `QPainter`, `QBrush`, `QPen`, `QColor` (см. *главу 18*);
- классы для растровых изображений: `QImage`, `QPixmap` (см. *главу 19*);
- классы стилей (см. *главу 26*) — как отдельному элементу, так и всему приложению может быть присвоен определенный стиль, изменяющий их внешний облик;
- класс приложения `QApplication`, который предоставляет цикл событий.

Давайте рассмотрим немного поподробнее последний класс — класс `QApplication`, с которым мы встречались в самом первом примере. Все, что было сказано ранее о классе `QCoreApplication`, относится также и к этому классу, поскольку он является его наследником. Объект класса `QApplication` представляет собой центральный контрольный пункт Qt-приложений, имеющих пользовательский интерфейс на базе виджетов. Этот объект используется для получения событий клавиатуры, мыши, таймера и других событий, на которые приложение должно реагировать соответствующим образом. Например, окно даже самого простого приложения может быть изменено по величине или быть перекрыто окном другого приложения, и на все подобные события необходима правильная реакция.

Класс `QApplication` напрямую унаследован от `QGuiApplication` и дополняет его следующими возможностями:

- установка стиля приложения. Таким способом можно устанавливать *виды и поведения* (Look & Feel) приложения, включая и свои собственные (см. *главу 26*);
- получение указателя на объект *рабочего стола* (desktop);
- управление глобальными манипуляциями с мышью (например, установка интервала двойного щелчка кнопкой мыши) и регистрация движения мыши в пределах и за пределами окна приложения;
- обеспечение правильного завершения работающего приложения при завершении работы операционной системы (см. *главу 28*).

Бывает так, что приложение может быть неактивным, а есть необходимость обратить на себя внимание пользователя. Для этой цели класс `QApplication` предоставляет статический метод `alert()`. Его вызов приведет к подскакиванию значка приложения на док-панели в Mac OS X (рис. 1.2) и его пульсации на панели задач в ОС Windows (рис. 1.3).

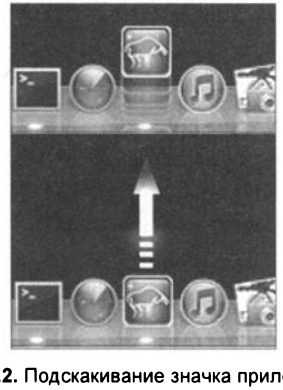


Рис. 1.2. Подскакивание значка приложения на док-панели в Mac OS X

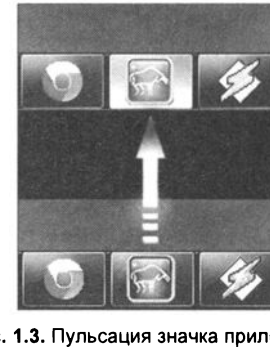


Рис. 1.3. Пульсация значка приложения на панели задач в ОС Windows

Модули *QtQuick* и *QtQML*

Это альтернатива виджетам — модули, представляющие собой набор технологий для быстрой разработки графических интерфейсов нового поколения на базе описательного языка QML, языка программирования JavaScript и всех остальных возможностей библиотеки Qt (см. *главу 53*).

Модуль *QtNetwork*

Сетевой модуль `QtNetwork` предоставляет инструментарий для программирования TCP- и UDP-сокетов (классы `QTcpSocket` и `QUdpSocket`), а также для реализации программ-клиентов, использующих HTTP- и FTP-протоколы (класс `QNetworkAccessManager`). Этот модуль описывается в *главе 39*.

Модули *QtXml* и *QtXmlPatterns*

Модуль `QtXml` предназначен для работы с базовыми возможностями XML посредством SAX2- и DOM-интерфейсов, которые определяют классы Qt (см. *главу 40*). А модуль `QtXmlPatterns` идет дальше и предоставляет поддержку для дополнительных технологий XML — таких как: XPath, XQuery, XSLT и XmlShemaValidator.

Модуль *QtSql*

Этот модуль предназначен для работы с базами данных. В него входят классы, предоставляющие возможность для манипулирования значениями баз данных (см. *главу 41*).

Модули *QtMultimedia* и *QtMultimediaWidgets*

Модуль `QtMultimedia` обладает всем необходимым для создания приложений с поддержкой мультимедиа. Он поддерживает как низкий уровень, необходимый для более детальной специализированной реализации, так и высокий уровень, делающий возможным проигрывать видео- и звуковые файлы при помощи всего нескольких строк программного кода. Модуль `QtMultimediaWidgets` содержит полезные элементы в виде виджетов, которые позволяют экономить время для реализации. Более подробно с этими модулями можно ознакомиться в *главе 27*.

Модуль *QtSvg*

Модуль поддержки графического векторного формата SVG, базирующегося на XML. Этот формат предоставляет возможность не только для вывода одного кадра векторного изображения, но может быть использован и для векторной анимации (см. *главу 22*).

Дополнительные модули Qt

Помимо важных модулей, Qt предоставляет так же и дополнительные модули, которые могут понадобиться не всем, а более узкому кругу разработчиков (табл. 1.2). Некоторые из этих модулей могут быть установлены только после их выделения в программе установки эксплуатации и обслуживания Qt (MaintenanceTool).

Таблица 1.2. *Некоторые из дополнительных модулей Qt*

Библиотека	Обозначение в проектном файле	Назначение
QtWebEngineCore	webenginecore	Позволяет очень просто интегрировать в приложение возможости веб
QtWebEngineWidgets	webenginewidgets	Предоставляет готовые к интеграции в приложение элементы в виде виджетов с возможностью также расширять элементы веб своими собственными виджетами (см. *главу 46*)
Qt 3D	3dcore, 3drenderer, 3dinput, 3dlogic, 3dextras, 3danimation, 3dquickscene2d	Представляет собой целую коллекцию из 7 модулей: Qt3DAnimation, Qt3DCore, Qt3DExtras, Qt3DInput, Qt3DLogic, Qt3DRender и Qt3DScene2D. Цель этих модулей — предоставить механизмы для упрощения программирования трехмерной графики
QtBluetooth	bluetooth	Содержит классы для использования беспроводной технологии Bluetooth
QtLocation	location	Предоставляет классы геолокации для определения текущего местоположения
QtSensors	sensors	Обеспечивает доступ к сенсорам мобильных устройств — таким, как, например, сенсор ориентации и акселерометр. В настоящее время этот модуль поддерживает платформы iOS, Android, SailFish и WinRT
QtCharts	charts	Реализует возможности для отображения данных в виде стильных диаграмм разного типа сложности и представлений
QtDataVisualization	datavisualization	Отображение данных в виде диаграмм в трехмерном пространстве
QtVirtualKeyboard	virtualkeyboardplugin	Собственная реализация виртуальной клавиатуры для целого ряда языков. Предполагает использование на настольных компьютерах
QtRemoteObjects	remoteobjects	Поддержка межпроцессного взаимодействия (*IPC*). В простой форме обеспечивает обмен информацией между приложениями, находящимися как на одном, так и на удаленных компьютерах

Резюме

Библиотека Qt не является монолитной библиотекой, она разбита на отдельные модули: QtCore, QtGui, QtWidgets, QtQuick, QtQML, QtMultimedia, QtNetwork, QtSql, QtXml и QtSvg. Каждый модуль имеет свое назначение — например, программирование интерфейса пользователя, графики, баз данных и др. Классы модулей предоставляют разработчику механизмы, расширяющие возможности программистов и, вместе с тем, упрощающие создание приложений. Вершиной модульной иерархии является модуль QtCore, который позволяет реализовывать приложения без графического интерфейса пользователя (консольные приложения). Объект класса QCoreApplication должен быть создан в приложении только один раз.

Для реализации приложений с графическим интерфейсом пользователя необходимы модули `QtWidgets` или `QtQuick`. Классы `QGuiApplication` и `QApplication` являются стержнем для Qt-приложений с графическим интерфейсом. Объект одного из этих классов не должен создаваться в приложении больше одного раза.

Библиотека Qt предоставляет так же и дополнительные модули. Некоторые из этих модулей могут быть установлены по желанию разработчика.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/01-510/** или с помощью следующего QR-кода (рис. 1.4):



Рис. 1.4. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 2

Философия объектной модели

> Те, кого первое знакомство с квантовой теорией не повергло в шок, скорее всего, вовсе ее не поняли.
> *Макс Борн*

Объектная модель Qt подразумевает, что все построено на объектах. Фактически, класс `QObject` — это основной, базовый класс. Подавляющее большинство классов Qt являются его наследниками. Классы, имеющие сигналы и слоты, должны быть унаследованы от этого класса.

Множественное наследование

При множественном наследовании важно помнить, что при определении класса имя класса `QObject` (или унаследованного от него) должно стоять первым, чтобы MOC (Meta Object Compiler, метаобъектный компилятор) мог его правильно распознать. Другой порядок приведет к ошибке при компиляции. В листинге 2.1 приведен правильный порядок для множественного наследования.

Листинг 2.1. Порядок наследования

```
class MyClass : public QObject, public AnotherClass {
...
};
```

Еще о множественном наследовании

При множественном наследовании также важно учитывать, что от класса `QObject` должен быть унаследован только один из базовых классов. Другими словами, нельзя производить наследование сразу от нескольких классов, наследующих класс `QObject`.

Класс `QObject` содержит в себе поддержку:

- сигналов и слотов (signal/slot);
- таймера;
- механизма объединения объектов в иерархии;
- событий и механизма их фильтрации;
- организации объектных иерархий;
- метаобъектной информации;

- приведения типов;
- свойств.

Сигналы и *слоты* — это средства, позволяющие эффективно производить обмен информацией о событиях, вырабатываемых объектами. О них мы подробно поговорим позже в этой главе.

Таймер дает возможность каждому из классов, унаследованных от класса `QObject`, не создавать дополнительно объект таймера. Тем самым экономится время на разработку. Подробнее о таймерах говорится в *главе 37*.

Механизм объединения объектов в иерархические структуры позволяет резко сократить временные затраты при разработке приложений, не заботясь об освобождении памяти создаваемых объектов, поскольку объекты-предки сами отвечают за уничтожение своих потомков.

Механизм фильтрации событий позволяет осуществить их перехват. Фильтр событий может быть установлен в любом классе, унаследованном от класса `QObject`, благодаря чему можно изменять реакцию объектов на происходящие события без изменения исходного кода класса (см. *главу 15*).

Метаобъектная информация включает в себя информацию о наследовании классов, что позволяет определять, являются ли классы непосредственными наследниками, а также узнать имя класса.

Для *приведения типов* Qt предоставляет шаблонную функцию `qobject_cast<T>()`, базирующуюся на метаинформации, создаваемой метаобъектным компилятором MOC (см. *главу 3*) для классов, унаследованных от `QObject`.

Свойства — это поля, для которых обязательно должны существовать методы чтения. С их помощью можно получать доступ к атрибутам объектов извне — например, из языка сценариев Qt Script (см. *часть VII*). Свойства также широко задействованы в визуальной среде разработки пользовательского интерфейса Qt Designer (см. *главу 44*), механизм которой реализован в Qt при помощи директив препроцессора. Задается свойство использованием макроса `Q_PROPERTY`. Определение свойства в общем виде выглядит следующим образом:

```
Q_PROPERTY(type name
           READ getFunction
           [WRITE setFunction]
           [RESET resetFunction]
           [DESIGNABLE bool]
           [SCRIPTABLE bool]
           [STORED bool]
          )
```

Сначала задаются тип и имя свойства, затем — имя метода чтения (`READ`). Определение остальных параметров не является обязательным. Третий параметр задает имя метода записи (`WRITE`), четвертый — имя метода сброса значения (`RESET`), пятый (`DESIGNABLE`) является логическим (булевым) значением, говорящим, должно ли свойство появляться в инспекторе свойств Qt Designer. Шестой параметр (`SCRIPTABLE`) — также логическое значение, которое управляет тем, будет ли свойство доступно для языка сценариев Qt Script. Последний, седьмой, параметр (`STORED`) управляет сериализацией, т. е. тем, будет ли свойство запоминаться во время сохранения объекта.

Итак, теперь, когда вы познакомились с понятием «свойство» (хотя в ближайшее время этот механизм нам и не понадобится), давайте все равно в качестве простого примера определим в классе свойство для управления режимом только чтения (листинг 2.2).

Листинг 2.2. Определение свойства для управления режимом только чтения

```
class MyClass : public QObject {
Q_OBJECT
Q_PROPERTY(bool readOnly READ isReadOnly WRITE setReadOnly)

private:
    bool m_bReadOnly;

public:
    MyClass(QObject* pobj = 0) : QObject(pobj)
                               , m_bReadOnly(false)
    {
    }

public:
    void setReadOnly(bool bReadOnly)
    {
        m_bReadOnly = bReadOnly;
    }

    bool isReadOnly() const
    {
        return m_bReadOnly;
    }
}
```

Класс `MyClass`, показанный в листинге 2.2, наследуется от класса `QObject`. Мы определяем атрибут `m_bReadOnly`, в котором будут запоминаться значения состояния. Этот атрибут инициализируется в конструкторе значением `false`. Для получения и изменения значения атрибута в классе `MyClass` определены методы `isReadOnly()` и `setReadOnly()`. Эти методы регистрируются в макросе `Q_PROPERTY`. Метод `isReadOnly()` служит для получения значения, поэтому указывается в секции `READ`, а метод `setReadOnly()` — для изменения значения, поэтому пишется в секции `WRITE`.

Из программы мы можем изменить значение нашего свойства следующим образом:

```
pobj->setProperty("readOnly", true);
```

А так можно получить текущее значение:

```
bool bReadOnly = pobj->property("readOnly").toBool();
```

Чтобы узнать сразу все свойства любого объекта и их значения, можно поступить следующим образом. Получить при помощи метода `propertyCount()` класса `QMetaObject` количество свойств. Затем в цикле получить для каждого индекса объект свойства и вызвать из него метод `typeName()` — для типа свойства и метод `name()` — для имени свойства. Значение свойства хранится в типе `QVariant`, который является универсальным классом, предназначенным для хранения любых типов (см. *главу 4*), его значение мы получаем вызовом метода `property()` из самого объекта. Этот метод был рассмотрен ранее. Вот пример того, как можно реализовать чтение всех свойств объекта:

```
const QMetaObject* pmo = pobj->metaObject();
for (int i = 0; i < pmo->propertyCount(); ++i) {
    const QMetaProperty mp = pmo->property(i);
    qDebug() << "Property#:" << i;
    qDebug() << "Type:" << mp.typeName();
    qDebug() << "Name:" << mp.name();
    qDebug() << "Value:" << pobj->property(mp.name());
}
```

Механизм сигналов и слотов

Элементы графического интерфейса определенным образом реагируют на действия пользователя и посылают сообщения. Существует несколько вариантов такого решения.

Старая концепция *функций обратного вызова* (callback functions), лежащая в основе X Window System, основана на использовании обычных функций, которые должны вызываться в результате действий пользователя. Применение такой концепции значительно усложняет исходный код программы, делая его менее понятным. Кроме того, здесь отсутствует возможность производить проверку типов возвращаемых значений, потому что во всех случаях возвращается указатель на пустой тип `void`. Например, для того чтобы сопоставить код с кнопкой, необходимо передать в функцию указатель на кнопку. Если пользователь нажимает на кнопку, функция будет вызвана. Сами библиотеки не проверяют, были ли аргументы, переданные в функцию, требуемого типа, а это часто является причиной сбоев. Другой недостаток функций обратного вызова заключается в том, что элементы графического интерфейса пользователя тесно связаны с функциональными частями программы, и это, в свою очередь, заметно усложняет разработку классов независимо друг от друга. Одним из ярких представителей этой концепции является библиотека Motif.

Важно помнить, что Motif и Windows API предназначены для процедурного программирования, и с реализацией объектно-ориентированных проектов у них наверняка появятся трудности.

Существуют, впрочем, специальные библиотеки классов языка C++, облегчающие программирование для ОС Windows. Одной из самых первых таких библиотек (и до сих пор на удивление находящихся в применении у целого ряда индивидуальных разработчиков и компаний) является Microsoft Foundation Classes (MFC). Назвать ее объектно-ориентированной можно лишь с большой натяжкой, поскольку она создавалась людьми, не подозревающими о существовании самых элементарных принципов объектно-ориентированного подхода. Одна из главных фундаментальных заповедей объектно-ориентированного подхода — это *инкапсуляция*, которая запрещает оставлять атрибуты классов незащищенными (ведь тогда объекты могут читать и изменять данные без ведома объекта-владельца), но, несмотря на это, во многих MFC-классах такое требование не соблюдено. Сама библиотека MFC является надстройкой, предоставляющей доступ к функциям Windows, реализованным на языке C, что заставляет разработчиков время от времени использовать устаревшие структуры, не вписывающиеся в рамки концепции объектно-ориентированного подхода. Интересно также отметить, что сама Microsoft для реализации широко известной программы Microsoft Word не использует MFC вообще.

При использовании MFC для обеспечения связей сообщения и методов обработки задействуются специальные макросы — так называемые *карты сообщений* (листинг 2.3). Они очень сильно загромождают исходный код программы, заметно снижая ее читаемость.

Листинг 2.3. Фрагмент программы, реализованной с помощью MFC

```
class CPhotoStylerApp : public CWinApp {
public:
    CPhotoStylerApp();
public:
    virtual BOOL InitInstance();

    afx_msg void OnAppAbout();
    afx_msg void OnFileNew();

    DECLARE_MESSAGE_MAP()
};
BEGIN_MESSAGE_MAP(CPhotoStylerApp, CWinApp)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
    ON_COMMAND(ID_FILE_NEW, OnFileNew)
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

Конструкции, подобные показанной в листинге 2.3, очень неудобны для человеческого восприятия и приводят в замешательство при проведении анализа кода программы. Пусть многие рассказывают об удобстве использования средств для автоматической генерации подобного кода, но созданы они были не от хорошей жизни. Так, непродуманность самой библиотеки вынуждает разработчика при внесении незначительных изменений модифицировать код самой программы сразу в нескольких местах. Например, для того чтобы добавить в диалоговое окно текстовое поле, необходимо провести целый ряд операций. Во-первых, нужно создать в классе диалога атрибут, предназначенный для хранения значений, вводимых в текстовом поле. Во-вторых, надо задать идентификатор ресурса текстового поля. В-третьих, поставить идентификатор ресурса и атрибут в методе `DoDataExchange()` в соответствие друг с другом при помощи метода `DDX_Text()`, после чего сможет осуществляться обмен данными между текстовым полем и атрибутом. В-четвертых, этим обменом необходимо управлять, передавая в методе `UpdateData()` значения булевого типа `true` или `false`. И лишь с помощью средств автоматического создания кода можно частично избавиться от такой проблемы, заставив выполнить эти изменения за вас и получив взамен другие недостатки, — например, дополнительное засорение кода программы ненужной информацией и возможное несовпадение созданного кода с утвержденными для проекта требованиями по форматированию и нотации (если не используется венгерская нотация). Я не противник обоснованного применения подобного рода средств, но, на мой взгляд, они не должны использоваться в качестве средства устранения изъянов плохого дизайна самой библиотеки.

В этой ситуации часть вины скрыта в самом языке C++. Дело в том, что C++ не создавался как средство для написания пользовательского интерфейса, и поэтому он не предоставляет соответствующей поддержки, делающей программирование в этой области более удобным. Например, если бы работа по передаче событий реализовывалась средствами самого языка, то отпадала бы необходимость в использовании подобного рода макросов. До настоящего времени не удавалось сделать ничего подобного, именно поэтому библиотека Qt явилась «как гром среди ясного неба». Потому что, в отличие от большинства других библиотек

программирования, Qt расширяет язык C++ дополнительными ключевыми словами для выполнения этой задачи.

Проблема расширения языка C++ решена в Qt с помощью специального препроцессора MOC (Meta Object Compiler, метаобъектный компилятор). Он анализирует классы на наличие в их определении специального макроса `Q_OBJECT` и внедряет в отдельный файл всю необходимую дополнительную информацию. Это происходит автоматически, без непосредственного участия разработчика. Подобная операция автоматического создания кода не противоречит привычному процессу программирования на C++ — ведь стандартный препроцессор перед компиляцией самой программы тоже создает промежуточный код, содержащий исполненные команды препроцессора. Подобным образом действует и MOC, записывая всю необходимую дополнительную информацию в отдельный файл, содержимое которого не требует внимания разработчика. Макрос `Q_OBJECT` должен располагаться сразу на следующей строке после ключевого слова `class` с определением имени класса. Очень важно помнить, что *после макроса не должно стоять точки с запятой*. Внедрять макрос в определение класса имеет смысл в тех случаях, когда созданный класс использует механизм сигналов и слотов или если ему необходима информация о свойствах.

Механизм сигналов и слотов полностью замещает старую модель функций обратного вызова, он очень гибок и полностью объектно-ориентирован. Сигналы и слоты — это краеугольный концепт программирования с использованием Qt, позволяющий соединить вместе несвязанные друг с другом объекты. Каждый унаследованный от `QObject` класс способен отправлять и получать сигналы. Эта особенность идеально вписывается в концепцию объектной ориентации и не противоречит человеческому восприятию. Представьте себе ситуацию: у вас звонит телефон, и вы реагируете на это снятием трубки. На языке сигналов и слотов подобную ситуацию можно описать следующим образом: объект «телефон» выслал сигнал «звонок», на который объект «человек» отреагировал слотом «снятия трубки».

Использование механизма сигналов и слотов дает программисту следующие преимущества:

- каждый класс, унаследованный от `QObject`, может иметь любое количество сигналов и слотов;
- сообщения, посылаемые посредством сигналов, могут иметь множество аргументов любого типа;
- сигнал можно соединять с различным количеством слотов. Отправляемый сигнал поступит ко всем подсоединенным слотам;
- слот может принимать сообщения от многих сигналов, принадлежащих разным объектам;
- соединение сигналов и слотов можно производить в любой точке приложения;
- сигналы и слоты являются механизмами, обеспечивающими связь между объектами. Более того, эта связь может выполняться между объектами, которые находятся в различных потоках (см. *главу 38*);
- при уничтожении объекта происходит автоматическое разъединение всех сигнально-слотовых связей. Это гарантирует, что сигналы не будут отправляться к несуществующим объектам.

Нельзя не упомянуть и о недостатках, связанных с применением сигналов и слотов:

- сигналы и слоты не являются частью языка C++, поэтому перед компиляцией программы требуется запуск дополнительного препроцессора;
- отправка сигналов происходит немного медленнее, чем обычный вызов функции, который осуществляется при использовании механизма функций обратного вызова;

- существует необходимость в наследовании класса `QObject`;
- в процессе компиляции не производится никаких проверок: имеется ли сигнал или слот в соответствующих классах или нет, совместимы ли сигнал и слот друг с другом и могут ли они быть соединены вместе. Об ошибке станет известно лишь тогда, когда приложение будет запущено в отладчике или на консоли. Вся эта информация выводится на консоль, поэтому, чтобы увидеть ее в Windows, в проектном файле (см. *главу 3*) необходимо в секции `CONFIG` добавить опцию `console` (для Mac OS X и Linux никаких дополнительных изменений проектного файла не требуется).

> ***Альтернативная форма сигнально-слотовых соединений***
>
> Использование альтернативной формы сигнально-слотовых соединений устраняет этот недостаток, позволяя выявлять ошибки соединений сигналов со слотами на этапе компиляции программы. Об этом читайте далее в *разд. «Соединение объектов»*.

Сигналы

Сигналы (signals) окружают нас в повседневной жизни везде: звонок будильника, жест регулировщика, а также и в не повседневной — примером может служить индейский сигнальный костер и т. п. В программировании с использованием Qt под понятием сигнала подразумеваются методы, которые в состоянии осуществлять пересылку сообщений. Причиной для появления сигнала может стать сообщение об изменении состояния управляющего элемента — например, о перемещении ползунка. На подобные изменения присоединенный объект, отслеживающий такие сигналы, может соответственно отреагировать, что, впрочем, и не обязательно. Это очень важный момент — он говорит о том, что соединяемые объекты могут быть абсолютно независимы и реализованы отдельно друг от друга. Такой подход позволяет объекту, отправляющему сигналы, не беспокоиться о том, что впоследствии будет происходить с этими сигналами. Объект, отправляющий сигналы, может даже и не догадываться, что их принимают и обрабатывают другие объекты. Благодаря такому разделению можно разбить большой проект на компоненты, которые будут разрабатываться разными программистами по отдельности, а потом соединяться при помощи сигналов и слотов вместе. Это делает код очень гибким и легко расширяемым — если один из компонентов устареет или должен будет реализован иначе, то все другие компоненты, участвующие в коммуникации с этим компонентом, и сам проект в целом, не изменятся. Новый компонент после разработки встанет на место старого и будет подключен к основной программе при помощи тех же самых сигналов и слотов. Это делает библиотеку Qt особенно привлекательной для реализации компонентно-ориентированных приложений. Однако не забывайте, что большое количество взаимосвязей приводит к возникновению сильно связанных систем, в которых даже незначительные изменения могут привести к непредсказуемым последствиям.

Сигналы определяются в классе, как и обычные методы, только без реализации. С точки зрения программиста они являются лишь прототипами методов, содержащихся в заголовочном файле определения класса. Всю дальнейшую заботу о реализации кода для этих методов берет на себя MOC. Методы сигналов не должны возвращать каких-либо значений, и поэтому перед именем метода всегда должен стоять возвращаемый параметр `void`.

Сигнал не обязательно соединять со слотом. Если соединения не произошло, то он просто не будет обрабатываться. Подобное разделение отправляющих и получающих объектов исключает возможность того, что один из подсоединенных слотов каким-то образом сможет помешать объекту, отправившему сигналы.

Библиотека предоставляет большое количество уже готовых сигналов для существующих элементов управления. В основном, для решения поставленных задач хватает этих сигналов, но иногда возникает необходимость реализации новых сигналов в своих классах. Пример определения сигнала приведен в листинге 2.4.

Листинг 2.4. Определение сигнала

```
class MySignal {
Q_OBJECT
...
signals:
    void doIt();
...
};
```

Обратите внимание на метод сигнала `doIt()`. Он не имеет реализации — эту работу принимает на себя MOC, обеспечивая примерно такую реализацию:

```
void MySignal::doIt()
{
    QMetaObject::activate(this, &staticMetaObject, 0, 0);
}
```

Из сказанного становится ясно, что не имеет смысла определять сигналы как `private`, `protected` или `public`, поскольку они играют роль вызывающих методов.

Выслать сигнал можно при помощи ключевого слова `emit`. Ввиду того, что сигналы играют роль вызывающих методов, конструкция отправки сигнала `emit doIt()` приведет к обычному вызову метода `doIt()`. Сигналы могут отправляться из классов, которые их содержат. Например, в листинге 2.4 сигнал `doIt()` может отсылаться только объектами класса `MySignal` и никакими другими. Чтобы иметь возможность отослать сигнал программно из объекта этого класса, следует добавить метод `sendSignal()`, вызов которого заставит объект класса `MySignal` отправлять сигнал `doIt()`, как это показано в листинге 2.5.

Листинг 2.5. Реализация сигнала

```
class MySignal {
Q_OBJECT
public:
    void sendSignal()
    {
        emit doIt();
    }
signals:
    void doIt();
};
```

Сигналы также имеют возможность высылать информацию, передаваемую в параметре. Например, если возникла необходимость передать в сигнале строку текста, то можно реализовать это, как показано в листинге 2.6.

Листинг 2.6. Реализация сигнала с параметром

```
class MySignal : public QObject {
Q_OBJECT
public:
    void sendSignal()
    {
        emit sendString("Information");
    }
signals:
    void sendString(const QString&);
};
```

Слоты

Слоты (slots) — это методы, которые присоединяются к сигналам. По сути, они являются обычными методами. Самое большое различие слотов и обычных методов состоит в возможности слотов принимать сигналы. Как и обычные методы, слоты определяются в классе как `public`, `private` или `protected`. Если необходимо сделать так, чтобы слот мог соединяться только с сигналами сторонних объектов, но не вызываться как обычный метод со стороны, этот слот необходимо объявить как `protected` или `private`. Во всех других случаях объявляйте их как `public`. В объявлениях перед каждой группой слотов должно стоять соответственно: `private slots:`, `protected slots:` или `public slots:`. Слоты могут быть и виртуальными.

> ***Соединение сигнала с виртуальным слотом***
>
> Соединение сигнала с виртуальным слотом примерно в десять раз медленнее, чем с невиртуальным. Поэтому не стоит делать слоты виртуальными, если нет особой необходимости.

Правда, есть небольшие ограничения, отличающие обычные методы от слотов. В слотах нельзя использовать параметры по умолчанию — например: `slotMethod(int n = 8)`, или определять слоты как `static`.

Классы библиотеки содержат целый ряд уже реализованных слотов. Но определение слотов для своих классов — это частая процедура. Реализация слота показана в листинге 2.7.

Листинг 2.7. Реализация слота

```
class MySlot : public QObject {
 Q_OBJECT
public:
 MySlot();
public slots:
 void slot()
 {
   qDebug() << "I'm a slot";
 }
};
```

Внутри слота вызовом метода `sender()` можно узнать, от какого объекта был выслан сигнал. Он возвращает указатель на объект типа `QObject`. Например, в этом случае на консоль будет выведено имя объекта, выславшего сигнал:

```
void slot()
{
    qDebug() << sender()->objectName();
}
```

Соединение объектов

Соединение объектов осуществляется при помощи статического метода `connect()`, который определен в классе `QObject`. В общем виде вызов метода `connect()` выглядит следующим образом:

```
QObject::connect(const QObject*     sender,
                 const char*        signal,
                 const QObject*     receiver,
                 const char*        slot,
                 Qt::ConnectionType type = Qt::AutoConnection
                );
```

Ему передаются пять следующих параметров:

- `sender` — указатель на объект, отправляющий сигнал;
- `signal` — это сигнал, с которым осуществляется соединение. Прототип (имя и аргументы) метода сигнала должен быть заключен в специальный макрос `SIGNAL(method())`;
- `receiver` — указатель на объект, который имеет слот для обработки сигнала;
- `slot` — слот, который вызывается при получении сигнала. Прототип слота должен быть заключен в специальный макрос `SLOT(method())`;
- `type` — управляет режимом обработки. Имеются три возможных значения:
  - `Qt::DirectConnection` — сигнал обрабатывается сразу вызовом соответствующего метода слота;
  - `Qt::QueuedConnection` — сигнал преобразуется в событие (см. *главу 14*) и ставится в общую очередь для обработки;
  - `Qt::AutoConnection` — это автоматический режим, который действует следующим образом: если отсылающий сигнал объект находится в одном потоке с принимающим его объектом, то устанавливается режим `Qt::DirectConnection`, в противном случае — режим `Qt::QueuedConnection`. Этот режим (`Qt::AutoConnection`) определен в методе `connection()` по умолчанию. Вам вряд ли придется изменять режимы «вручную», но полезно знать, что такая возможность есть.

Существует альтернативный вариант метода `connect()`, преимущество которого заключается в том, что все ошибки соединения сигналов со слотами выявляются на этапе компиляции программы, а не при ее исполнении, как это происходит в классическом варианте метода `connect()`. Прототип альтернативного метода выглядит так:

```
QObject::connect(const QObject*     sender,
                 const QMetaMethod& signal,
                 const QObject*     receiver,
```

```
                const QMetaMethod& slot,
                Qt::ConnectionType type = Qt::AutoConnection
               );
```

Параметры этого метода полностью аналогичны предыдущему за исключением тех параметров, которые были объявлены в предыдущем методе как `const char*`. Вместо них используются указатели на методы сигналов и слотов классов напрямую. Благодаря именно этому, если вы вдруг ошибетесь с названием сигнала или слота, ваша ошибка будет выявлена сразу в процессе компиляции программы. К недостаткам альтернативного метода можно отнести то, что при каждом соединении нужно явно указывать имена классов для сигнала и слота и следить за совпадением их параметров.

Следующий пример демонстрирует то, как может быть осуществлено соединение объектов в программе с помощью первого метода `connect()`.

```
void main()
{
    ...
    QObject::connect(pSender, SIGNAL(signalMethod()),
                     pReceiver, SLOT(slotMethod())
                    );
    ...
}
```

А вот так выглядит аналогичное соединение при помощи альтернативного метода `connect()`:

```
QObject::connect(pSender, &SenderClass::signalMethod,
                 pReceiver, &ReceiverClass::slotMethod
                );
```

Далее мы будем использовать первый вариант метода `connect()`.

Если вызов происходит из класса, унаследованного от `QObject`, тогда `QObject::` можно опустить:

```
MyClass::MyClass() : QObject()
{
    ...
    connect(pSender, SIGNAL(signalMethod()),
            pReceiver, SLOT(slotMethod())
           );
    ...
}
```

В случае если слот содержится в классе, из которого производится соединение, то можно воспользоваться сокращенной формой метода `connect()`, опустив третий параметр (`pReceiver`), указывающий на объект-получатель. Другими словами, если в качестве объекта-получателя должен стоять указатель `this`, его можно просто не указывать:

```
MyClass::MyClass() : QObject()
{
    connect(pSender, SIGNAL(signalMethod()), SLOT(slot()));
}
void MyClass::slot()
```

```
{
   qDebug() << "I'm a slot";
}
```

Метод `connect()` после вызова возвращает объект класса `Connection`, с помощью которого можно определить, произошло соединение успешно или нет. Этим обстоятельством можно воспользоваться, например, для того, чтобы в случае, если в методе будет допущена какая-либо ошибка, то аварийно завершить программу вызовом макроса `Q_ASSERT()`.

Класс `Connection` содержит оператор неявного преобразования к типу `bool`, поэтому мы используем в примере переменную `bOk` этого типа. Подобный код может выглядеть следующим образом.

```
bool bOk = true;
bOk &= connect(pcmd1, SIGNAL(clicked()), pobjReceiver1, SLOT(slotButton1Clicked()));
bOk &= connect(pcmd2, SIGNAL(clicked()), pobjReceiver2, SLOT(slotButton2Clicked()));
Q_ASSERT(bOk);
```

Иногда возникают ситуации, когда объект не обрабатывает сигнал, а просто передает его дальше. Для этого необязательно определять слот, который в ответ на получение сигнала (при помощи `emit`) отсылает свой собственный. Можно просто соединить сигналы друг с другом. Отправляемый сигнал должен содержаться в определении класса:

```
MyClass::MyClass() : QObject()
{
    connect(pSender, SIGNAL(signalMethod()), SIGNAL(mySignal()));
}
```

Отправку сигналов можно на некоторое время заблокировать, вызвав метод `blockSignals()` с параметром `true`. Объект будет «молчать», пока блокировка не будет снята тем же методом `blockSignals()` с параметром `false`. При помощи метода `signalsBlocked()` можно узнать текущее состояние блокировки сигналов.

Соединение сигнала с лямбда-функцией напрямую

Если ваш компилятор поддерживает стандарт C++11, то вы можете даже соединить сигнал напрямую с лямбда-функцией. Следующее соединение после нажатия кнопки `pcmd` произведет скрытие виджета `pwgt`:

```
connect(pcmd, &QPushButton::clicked, [=](){pwgt->hide();});
```

Окна программы, показанные на рис. 2.1, демонстрируют механизм сигналов и слотов в действии. Для этого примера создается приложение (листинги 2.8–2.10), в первом окне которого (рис. 2.1, *справа*) находится кнопка нажатия, а во втором (рис. 2.1, *слева*) — виджет надписи. При щелчке в правом окне на кнопке **ADD** (Добавить) происходит увеличение отображаемого в левом окне значения на единицу. Как только значение достигнет пяти, произойдет выход из приложения.



Рис. 2.1. Программа-счетчик: демонстрация работы механизма сигналов и слотов

Листинг 2.8. Основная программа приложения (файл main.cpp)

```
#include <QtWidgets>
#include "Counter.h"

int main (int argc, char** argv)
{
    QApplication app(argc, argv);

    QLabel      lbl("0");
    QPushButton cmd("ADD");
    Counter     counter;

    lbl.show();
    cmd.show();

    QObject::connect(&cmd, SIGNAL(clicked()),
                     &counter, SLOT(slotInc())
                    );

    QObject::connect(&counter, SIGNAL(counterChanged(int)),
                     &lbl, SLOT(setNum(int))
                    );

    QObject::connect(&counter, SIGNAL(goodbye()),
                     &app, SLOT(quit())
                    );

    return app.exec();
}
```

В основной программе приложения (листинг 2.8) создается объект надписи `lbl`, нажимающаяся кнопка `cmd` и объект счетчика `counter` (описание которого приведено в листингах 2.9–2.10). Далее сигнал `clicked()` соединяется со слотом `slotInc()`. При каждом нажатии на кнопку вызывается метод `slotInc()`, увеличивающий значение счетчика на 1. Он должен быть в состоянии сообщать о подобных изменениях, чтобы элемент надписи отображал всегда только действующее значение. Для этого сигнал `counterChanged(int)`, передающий в параметре актуальное значение счетчика, соединяется со слотом `setNum(int)`, способным принимать это значение.

Следите за совпадением типов сигналов со слотами!

При соединении сигналов со слотами, передающими значения, важно следить за совпадением их типов. Например, сигнал, передающий в параметре значение `int`, не должен соединяться со слотом, принимающим `QString`:

```
connect(pobj1, SIGNAL(sig(int)), pobj2, SLOT(slt(int))); // ПРАВИЛЬНО!
connect(pobj1, SIGNAL(sig(int)), pobj2, SLOT(slt(QString))); // НЕПРАВИЛЬНО!
```

Можно игнорировать в слоте значения, передаваемые сигналом

Так же важно понимать, что можно игнорировать в слоте значения, передаваемые сигналом. Так, допустимо, например, соединить сигнал, высылающий значение `int`, со слотом,

который не принимает параметров. Но нельзя сделать наоборот и соединить сигнал, который не высылает никаких значений, со слотом, который принимает параметры:

```
connect(pobj1, SIGNAL(sig(int)), pobj2, SLOT(slt())); // МОЖНО!
connect(pobj1, SIGNAL(sig()), pobj2, SLOT(slt(int))); // НЕЛЬЗЯ!
```

Не указывайте вместе с типом имя переменной!

Обратите внимание, что в качестве аргумента выступает только тип `int` без указания имени переменной. Если вы укажете вместе с типом имя переменной, то соединение работать не будет. При этом, у вас может уйти много времени на поиск ошибки:

```
connect(pobj1, SIGNAL(sig(int n)), pobj2, SLOT(slt(int n))); // ОШИБКА!
```

Наконец, сигнал `goodbye()`, символизирующий конец работы счетчика, соединяется со слотом объекта приложения `quit()`, который осуществляет завершение работы приложения, после нажатия кнопки **ADD** в пятый раз. Наше приложение состоит из двух окон, и после закрытия последнего окна его работа автоматически завершится.

Листинг 2.9. Счетчик (файл Counter.h)

```
#pragma once

#include <QObject>

// ============================================================================
class Counter : public QObject {
    Q_OBJECT
private:
    int m_nValue;

public:
    Counter();

public slots:
    void slotInc();

signals:
    void goodbye       (   );
    void counterChanged(int);
};
```

Как видно из листинга 2.9, в определении класса счетчика содержатся два сигнала: `goodbye()`, сообщающий о конце работы счетчика, и `counterChanged(int)`, передающий актуальное значение счетчика, а также слот `slotInc()`, увеличивающий значение счетчика на единицу.

Листинг 2.10. Счетчик (файл counter.cpp)

```
#include "Counter.h"

// ----------------------------------------------------------------------------
Counter::Counter() : QObject()
                   , m_nValue(0)
{
}
```

```
// ------------------------------------------------------------------------
void Counter::slotInc()
{
    emit counterChanged(++m_nValue);

    if (m_nValue == 5) {
        emit goodbye();
    }
}
```

В листинге 2.10 метод слота `slotInc()` отправляет два сигнала: `counterChanged()` и `goodbye()`. Сигнал `goodbye()` отправляется при значении атрибута `m_nValue`, равном 5.

Слот, не имеющий параметров, можно соединить с сигналом, имеющим параметры. Это удобно, когда сигналы поставляют больше информации, чем требуется для объекта, получающего сигнал. В этом случае в слоте можно не указывать параметры:

```
MyClass::MyClass() : QObject()
{
    connect(pSender, SIGNAL(signalMethod(int)), SIGNAL(mySignal()));
}
```

Если вы не уверены, пригодится ли параметр сигнала в будущем, то лучше определить слот с параметром и проигнорировать его внутри слота. Зато потом, когда возникнет необходимость, вам не потребуется менять прототип слота.

Разъединение объектов

Если есть возможность соединения объектов, то должна существовать и возможность их разъединения. В Qt при уничтожении объекта все связанные с ним соединения уничтожаются автоматически, но в редких случаях может возникнуть необходимость в уничтожении этих соединений «вручную». Для этого существует статический метод `disconnect()`, параметры которого аналогичны параметрам статического метода `connect()`. В общем виде этот метод выглядит таким образом:

```
QObject::disconnect(sender, signal, receiver, slot);
```

Следующий пример демонстрирует, как может быть выполнено разъединение объектов в программе:

```
void main()
{
    ...
    QObject::disconnect(pSender, SIGNAL(signalMethod()),
                        pReceiver, SLOT(slotMethod())
                       );
    ...
}
```

Существуют два сокращенных, не статических варианта: `disconnect(signal, receiver, slot)` и `disconnect(receiver, slot)`.

Переопределение сигналов

Если есть необходимость сократить в классе количество слот-методов и отреагировать действием на разные сигналы в одном слоте, то следует воспользоваться классом `QSignalMapper`. С его помощью можно переопределить сигналы и сделать так, чтобы в слот отправлялись значения типов `int`, `QString` или `QWidget`. Рассмотрим этот механизм на примере использования значений типа `QString`. Итак, предположим, что у нас в программе есть две кнопки: при нажатии на первую кнопку нам нужно отобразить сообщение `Button1 Action`, а при нажатии на вторую — `Button2 Action`. Мы, конечно же, могли бы реализовать в классе два разных слота, которые были бы соединены с сигналами `clicked()` каждой из двух кнопок и выводили бы каждый свое сообщение. Но мы поступим иначе и для этого воспользуемся классом `QSignalMapper` (листинг 2.11).

Листинг 2.11. Пример переопределения сигналов

```
MyClass::MyClass(QWidget* pwgt)
{
...
    QSignalMapper* psigMapper = new QSignalMapper(this);
    connect(psigMapper, SIGNAL(mapped(const QString&)),
            this,       SLOT(slotShowAction(const QString&))
           );

    QPushButton* pcmd1 = new QPushButton("Button1");
    connect(pcmd1, SIGNAL(clicked()), psigMapper, SLOT(map()));
    psigMapper->setMapping(pcmd1, "Button1 Action");

    QPushButton* pcmd2 = new QPushButton("Button2");
    connect(pcmd2, SIGNAL(clicked()), psigMapper, SLOT(map()));
    psigMapper->setMapping(pcmd1, "Button2 Action");
...
}
void MyClass::slotShowAction(const QString& str)
{
    qDebug() << str;
}
```

В листинге 2.11 мы создаем объект класса `QSignalMapper` и соединяем его сигнал `mapped()` с единственным слотом `slotShowAction()`, который принимает объекты `QString`. Класс `QSignalMapper` предоставляет слот `map()`, с которым должен быть соединен каждый объект, сигнал которого должен быть переопределен. При помощи метода `QSignalMapper::setMapping()` мы устанавливаем конкретное значение, которое должно быть направлено в слот при получении сигнала, — в нашем случае сигнала `clicked()`.

Вот и все... Остается только отметить, что сигнальными переопределениями не стоит злоупотреблять, потому что чрезмерное увлечение этими конструкциями может заметно снизить читаемость программного кода.

Организация объектных иерархий

Организация объектов в иерархии снимает с разработчика необходимость самому заботиться об освобождении памяти от созданных объектов.

Конструктор класса `QObject` выглядит следующим образом:

```
QObject(QObject* pobj = 0);
```

В его параметре передается указатель на другой объект класса `QObject` или унаследованного от него класса. Благодаря этому параметру существует возможность создания объектов-иерархий. Он представляет собой указатель на объект-предок. Если в первом параметре передается значение, равное нулю, или ничего не передается, то это значит, что у создаваемого объекта нет предка, и он будет являться объектом верхнего уровня и находиться на вершине объектной иерархии. Объект-предок задается в конструкторе при создании объекта, но впоследствии его можно в любой момент исполнения программы изменить на другой при помощи метода `setParent()`.

Созданные объекты по умолчанию не имеют имени. При помощи метода `setObjectName()` можно присвоить объекту имя. Имя объекта не имеет особого значения, но может быть полезно при отладке программы. Для того чтобы узнать имя объекта, можно вызвать метод `objectName()`, как показано в листинге 2.12.

Листинг 2.12. Пример создания объектной иерархии

```
QObject* pobj1 = new QObject;
QObject* pobj2 = new QObject(pobj1);
QObject* pobj4 = new QObject(pobj2);
QObject* pobj3 = new QObject(pobj1);
pobj2->setObjectName("the first child of pobj1");
pobj3->setObjectName("the second child of pobj1");
pobj4->setObjectName("the first child of pobj2");
```

В первой строке листинга 2.12 создается объект верхнего уровня (объект без предка). При создании объекта `pobj2` в его конструктор передается в качестве предка указатель на объект `pobj1`. Объект `pobj3` имеет в качестве предка `pobj1`, а объект `pobj4` имеет предка `pobj2`. Полученная объектная иерархия показана на рис. 2.2.

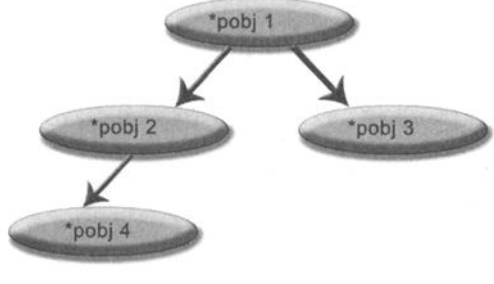


Рис. 2.2. Схема получившейся объектной иерархии

При уничтожении созданного объекта (при вызове его деструктора) все присоединенные к нему объекты-потомки уничтожаются автоматически. Такая особенность рекурсивного уничтожения объектов значительно упрощает программирование, поскольку при этом не

нужно заботиться об освобождении ресурсов памяти. Именно поэтому необходимо создавать объекты, а особенно объекты неверхнего уровня, динамически, при помощи оператора `new`, иначе удаление объекта приведет к ошибке при исполнении программы.

> ***Все объекты должны создаваться в памяти динамически!***
>
> Одна из самых распространенных ошибок программистов, пишущих на языке C++, при программировании с использованием библиотеки Qt — это самостоятельный контроль процесса выделения/освобождения памяти для объекта и нединамическое создание элементов управления. При программировании с Qt важно помнить, что все объекты должны создаваться в памяти динамически, с помощью оператора `new`. Исключение из этого правила могут составлять только объекты, не имеющие предков.

Для получения информации об объектной иерархии существуют два метода: `parent()` и `children()`. С помощью метода `parent()` можно определить объект-предок. Согласно рис. 2.2, вызов `pobj2->parent()` вернет указатель на объект `obj1`. Для объектов верхнего уровня этот метод вернет значение 0. Чтобы вывести на консоль всю цепь имен объектов-предков какого-либо из объектов, можно поступить так, как показано в листинге 2.13 (проделаем это для объекта `pobj4` из листинга 2.12).

Листинг 2.13. Вывод имен объектов предков

```
for (QObject* pobj = pobj4; pobj; pobj = pobj->parent()) {
    qDebug() << pobj->objectName();
}
```

И на экране вы увидите:

```
the first child of pobj2
the first child of pobj1
```

Метод `children()` возвращает константный указатель на список объектов-потомков. Для приведенного ранее примера (см. листинг 2.11 и рис. 2.2) метод `pobj1->children()` возвратит указатель на список `QObjectList`, содержащий два элемента: указатели `pobj2` и `pobj3`.

Поиск нужного объекта-потомка можно осуществлять при помощи метода `findChild()`. В параметре этого метода необходимо передать имя искомого объекта. Например, следующий вызов возвратит указатель на объект `pobj4`:

```
QObject* pobj = pobj1->findChild<QObject*>("the first child of pobj2");
```

Для расширенного поиска существует метод `findChildren()`, возвращающий список указателей на объекты. Все параметры метода не обязательны, может передаваться либо строка имени, либо регулярное выражение (см. *главу 4*), а вызов метода без параметров приведет к тому, что он вернет список указателей на все объекты-потомки. Поиск производится рекурсивно. Следующий вызов возвратит список указателей на все объекты, имя которых начинается с букв `th`, в нашем случае их три:

```
QList<QObject*> plist = pobj1->findChildren<QObject*>(QRegExp("th*"));
```

Для того чтобы возвратить все объекты потомков указанного типа, независимо от их имени, нужно просто не указывать аргумент:

```
QList<QObject*> plist = pobj1->findChildren<QObject*>();
```

В данном конкретном случае эта функция вернет нам указатели на объекты `pobj2`, `pobj3` и `pobj4`.

Для отладки программы полезен метод `dumpObjectInfo()`, который показывает следующую информацию, относящуюся к объекту:

- имя объекта;
- класс, от которого был создан объект;
- сигнально-слотовые соединения.

Вся эта информация поступает в стандартный поток вывода `stdout`.

При отладке можно воспользоваться и методом `dumpObjectTree()`, предназначенным для отображения объектов-потомков в виде иерархии. Например, вызов `dumpObjectTree()` для нашего объекта `pobj1` из листинга 2.11 покажет:

```
QObject::
    QObject::the first child of pobj1
        QObject::the first child of pobj2
    QObject::the second child of pobj1
```

Метаобъектная информация

Каждый объект, созданный от класса `QObject` или от унаследованного от него класса, располагает структурой данных, называемой *метаобъектной информацией* (класс `QMetaObject`). В ней хранится информация о сигналах, слотах (включая указатели на них), о самом классе и о наследовании. Получить доступ к этой информации можно посредством метода `QObject::metaObject()`. Таким образом, для того чтобы узнать, например, имя класса объекта, от которого он был создан, можно поступить следующим образом:

```
qDebug() << pobj1->metaObject()->className();
```

А для того чтобы сравнить имя класса с известным, можно поступить так:

```
if (pobj1->metaObject()->className() == "MyClass") {
    // Выполнить какие-либо действия
}
```

Для получения информации о наследовании классов существует метод `inherits(const char*)`, который определен непосредственно в классе `QObject` и возвращает значение `true`, если класс объекта унаследован от указанного в этом методе класса либо создан от этого класса, иначе метод возвращает значение `false`. Например:

```
if(pobj->inherits("QWidget")) {
    QWidget* pwgt = static_cast<QWidget*>(pobj);
    // Выполнить какие-либо действия с pwgt
}
```

Метаобъектную информацию использует и операция приведения типов `qobject_cast<T>`. Таким образом, при помощи метода `inherits()` пример можно изменить:

```
QWidget* pwgt = qobject_cast<QWidget*>(pobj);
if(pwgt) {
    // Выполнить какие-либо действия с pwgt
}
```

К метаобъектной информации относится также и метод `tr()`, предназначенный для интернационализации программ. Подробнее интернационализация описана в *главе 30*.

Резюме

В этой главе мы узнали о сущности сигналов и слотов. Это достойная альтернатива используемым до сих пор средствам для реализации связей между компонентами графического интерфейса пользователя, позволяющая значительно повысить читабельность программного кода.

Сигналы и слоты могут быть соединены друг с другом, причем сигнал может быть соединен с большим количеством слотов. Слот, в свою очередь, тоже может быть соединен со многими сигналами. В случае, когда слот не делает ничего, кроме отправки полученного сигнала дальше, можно вообще обойтись без него, а просто соединить сигналы друг с другом. Методы сигналов должны быть обозначены в определении класса специальным словом `signals`, а слоты — словом `slots`. При этом слоты являются обыкновенными методами языка C++ и в их определении могут присутствовать модификаторы `public`, `protected`, `private`. Реализацию кода для сигналов берет на себя MOC. Отправка сигнала производится при помощи ключевого слова `emit`. Класс, содержащий сигналы и слоты, должен быть унаследован от класса `QObject` или от класса, унаследованного от этого класса. Сигнально-слотовые соединения всегда можно удалить (отсоединить), воспользовавшись методом `disconnect()`, но это бывает нужно крайне редко, т. к. при удалении объекта автоматически уничтожаются все его соединения. Соединение объектов производится при помощи статического метода `QObject::connect()`.

`QObject` — класс, по сути являющийся основным классом при программировании с использованием Qt. Конструктор класса `QObject` имеет два параметра: первый используется для создания объектных иерархий, а второй — для присвоения объекту имени. Свойства объектов важны, т. к. позволяют получать информацию о классе и об объекте в процессе исполнения программы. Все объекты класса `QObject` или унаследованных от него классов должны создаваться динамически оператором `new`, а об освобождении памяти созданной объектной иерархии программист может не беспокоиться.

Поскольку концепт сигналов и слотов, а также информацию о наследственности, невозможно было реализовать средствами самого языка C++, был создан специальный препроцессор, называемый MOC (метаобъектный компилятор), задача которого — создавать для заголовочных файлов дополнительные CPP-файлы, подлежащие компиляции и присоединению их объектного кода к исполняемому коду программы. Для того чтобы MOC мог распознать классы, нуждающиеся в подобной переработке, такой класс должен содержать макрос `Q_OBJECT`.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/02-510/** или с помощью следующего QR-кода (рис. 2.3):

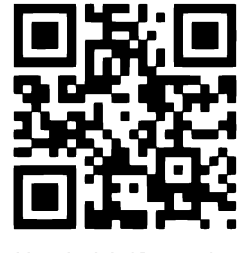


Рис. 2.3. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 3

Работа с Qt

Чтобы узнать истину, нужно узнать причины.
Фрэнсис Бэкон

Прежде чем начать изучение собственно библиотеки Qt и перейти к следующим главам, вам надо обязательно ознакомиться с инструментами и средствами, необходимыми для работы с Qt.

Интегрированная среда разработки

Существует много различных интегрированных сред разработки (IDE, Integrated Development Environment), которые можно очень эффективно использовать при создании Qt-проектов. К ним относятся Microsoft Visual Studio, XCode, IBM Eclipse и др. Но, пожалуй, самая яркая среда разработки — это Qt Creator (рис. 3.1). Эта IDE включена в пакет поставки Qt. Ей посвящена в книге отдельная глава, поэтому не буду забегать вперед, а всех заинтересованных читателей, которым не терпится поскорее узнать о ней, отсылаю к *главе 47*.

Программа Qt Assistant

Документация — это то, чем чаще всего пользуется разработчик. Существует и средство, обеспечивающее ему быстрый поиск нужной информации. Таким средством является программа Qt Assistant, похожая по принципу работы на веб-браузер. Qt Assistant предоставляет возможность поиска текста во всех доступных документах о Qt. Для того чтобы установить путь к местоположению той или иной документации, можно воспользоваться параметром `-docPath`. На рис. 3.2 показано окно программы Qt Assistant.

Программа Qt Assistant так же встроена в программу Qt Creator. Это позволяет ее использовать, не покидая окна IDE, и получать контекстную помощь прямо в окне редактора кода.

Работа с qmake

Ни один программист для компиляции своей программы не будет каждый раз задавать параметры компоновки и передавать пути к библиотекам вручную. Гораздо удобнее создать make-файл (makefile), который возьмет на себя всю работу по настройке компилятора и компоновщика.

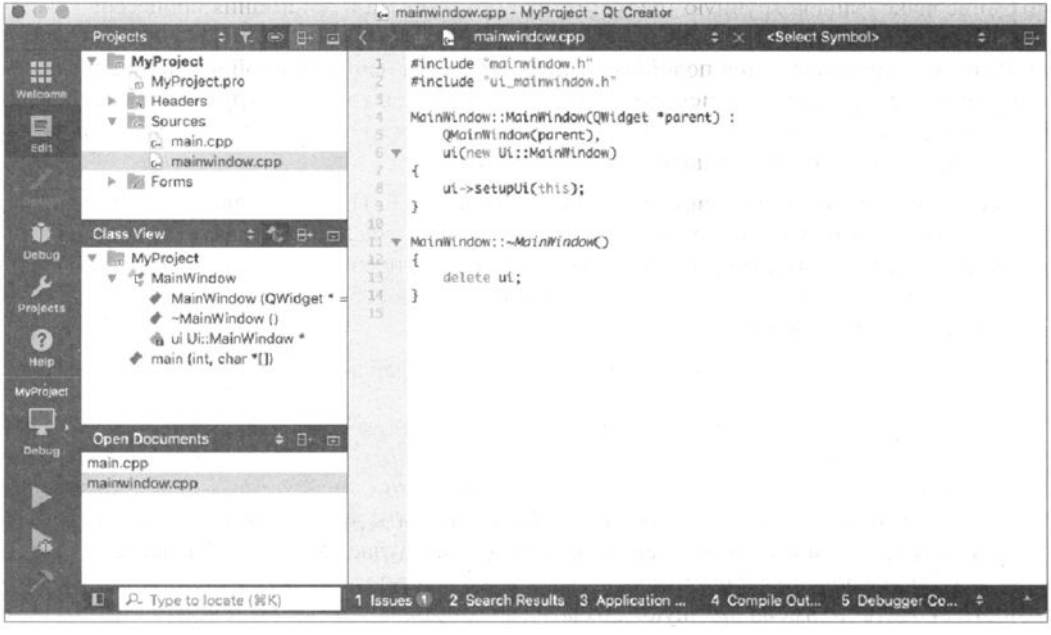


Рис. 3.1. Окно интегрированной среды разработки Qt Creator

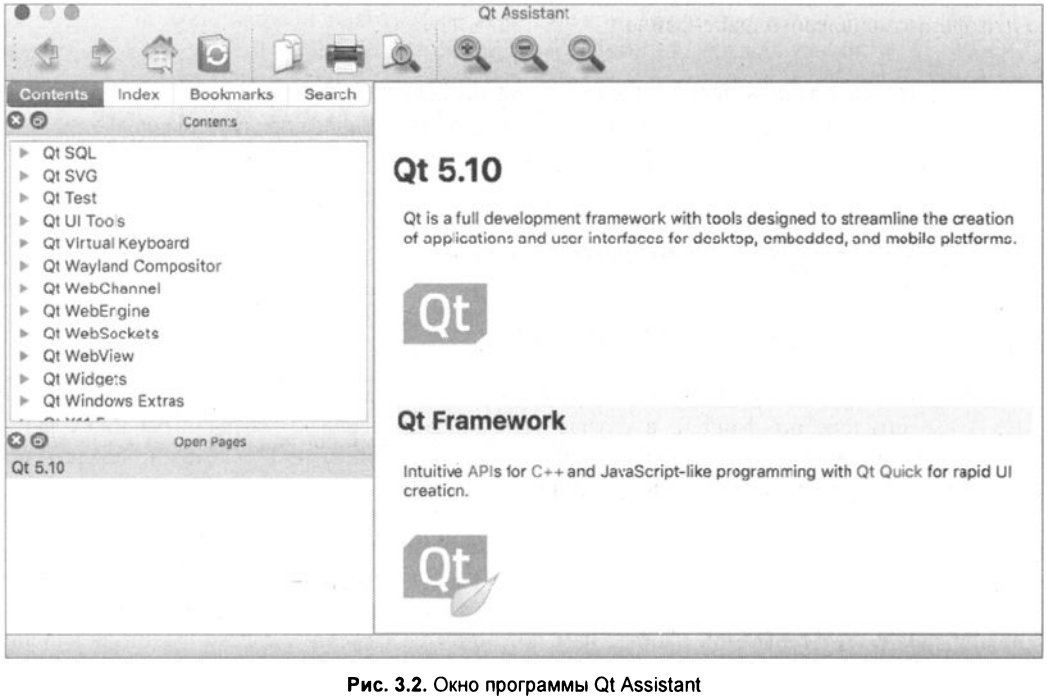


Рис. 3.2. Окно программы Qt Assistant

Создание make-файлов вручную требует от их автора опыта и понимания процессов компоновки приложения, причем в зависимости от платформы вид этих файлов будет различаться. Раньше техника создания подобных файлов являлась неотъемлемой частью программирования, но теперь многое изменилось. И дело совсем не в том, что структура make-файлов стала проще, скорее наоборот — она стала сложнее. Просто появились специальные утилиты — генераторы, которые выполнят эту работу за вас.

Утилита qmake вошла в поставку Qt, начиная с версии 3.0. Примечательно, что эта утилита так же хорошо переносима, как и сама Qt. Программа qmake при создании make-файлов интерпретирует файлы проектов, которые имеют расширение pro и содержат различные параметры. Самое удивительное, что утилита qmake способна создавать не только make-файлы, но и сами pro-файлы.

Допустим, вы указали, что в каталоге есть исходные файлы C++, выполнив следующую команду:

```
qmake -project
```

В результате будет реализована попытка автоматического создания pro-файла. Это удобно, поскольку на первых порах вам не понадобится вникать во все тонкости создания pro-файлов. Также это может оказаться полезным в том случае, если вы обладаете большим количеством файлов, к которым требуется создать pro-файл, — тогда у вас отпадет необходимость вносить их имена вручную. Создать из pro-файла make-файл совсем нетрудно, для этого нужно просто выполнить команду:

```
qmake file.pro -o Makefile
```

Как нетрудно догадаться, `file.pro` — это имя pro-файла, а `Makefile` — имя для создаваемого платформозависимого make-файла.

Создание файлов при помощи qmake на Mac OS X

На Mac OS X есть две основные возможности создания файлов при помощи qmake:

- первая — это создание make-файла для GNU C++:

  ```
  qmake -spec macx-clang -o Makefile
  ```

- вторая — создание проектных файлов для XCode:

  ```
  qmake -spec macx-xcode -o MyPrj
  ```

Первая опция является опцией по умолчанию.

Если бы мы выполнили команду без параметров, то утилита qmake попыталась бы найти в текущем каталоге pro-файл и, в случае успеха, автоматически создала бы make-файл. Таким образом, имея в распоряжении только исходные файлы на C++, можно создать исполняемую программу, выполнив всего лишь три команды:

```
qmake -project
qmake
make
```

Конечно, для более серьезной работы нам понадобится изменять содержимое pro-файлов, что позволит осуществлять более тонкую настройку наших проектов. Табл. 3.1 содержит некоторые опции pro-файла, полный список которых можно получить в официальной документации Qt, поставляемой вместе с самой библиотекой (для этого вы можете просто запустить программу Qt Assistant).

Таблица 3.1. Некоторые опции для файла проекта

Опция	Назначение
HEADERS	Список созданных заголовочных файлов
SOURCES	Список созданных файлов реализации (с расширением cpp)
FORMS	Список файлов с расширением ui. Эти файлы создаются программой Qt Designer и содержат описание интерфейса пользователя в формате XML (см. *главы 40* и *44*)
TARGET	Имя приложения. Если это поле не заполнено, то название программы будет соответствовать имени проектного файла
LIBS	Задает список библиотек, которые должны быть подключены для создания исполняемого модуля
CONFIG	Задает опции, которые должен использовать компилятор
DESTDIR	Задает путь, куда будет помещен готовый исполняемый модуль
DEFINES	Здесь можно передать опции для компилятора. Например, это может быть опция помещения отладочной информации для отладчика debuger в исполняемый модуль
INCLUDEPATH	Путь к каталогу, где содержатся заголовочные файлы. Этой опцией можно воспользоваться в том случае, если уже есть готовые заголовочные файлы, и вы хотите использовать их (подключить) в текущем проекте
DEPENDPATH	В этом разделе указываются зависимости, необходимые для компиляции
SUBDIRS	Задает имена подкаталогов, которые содержат pro-файлы
TEMPLATE	Задает разновидность проекта. Например: `app` — приложение, `lib` — библиотека, `subdirs` — подкаталоги
TRANSLATIONS	Задает файлы переводов, используемые в проекте (см. *главу 31*)

Давайте пристальнее рассмотрим «анатомию» проектных файлов. Итак, pro-файл может выглядеть следующим образом:

```
TEMPLATE = app
HEADERS += file1.h \
           file2.h
SOURCES += main.cpp  \
           file1.cpp \
           file2.cpp
TARGET = file
CONFIG += qt warn_on release
```

В первой строке задается тип программы. В нашем случае это приложение, поэтому `TEMPLATE = app` (если бы нам нужно было создать библиотеку, то `TEMPLATE` следовало бы присвоить значение `lib`). Во второй строке (`HEADERS`) перечисляются все заголовочные файлы, принадлежащие проекту. В опции `SOURCES` перечисляются все файлы реализации проекта. Строка `TARGET` определяет имя программы, а строка `CONFIG` — опции, которые должен использовать компилятор в соответствии с подсоединяемыми библиотеками. Например, в рассматриваемом случае:

- `qt` указывает, что это Qt-приложение и используется библиотека Qt;
- `warn_on` означает, что компилятор должен выдавать как можно больше предупреждающих сообщений;

- `release` указывает, что приложение должно быть откомпилировано в окончательном варианте, без отладочной информации.

Как видно из примера, программе qmake не требуется много информации, поскольку она опирается на файл локальной конфигурации, который определен системной конфигурацией. Такой файл очень важен еще и потому, что один и тот же вызов утилиты qmake приведет к созданию различных make-файлов в зависимости от того, на какой платформе она была вызвана. Это один из очень важных шагов в сторону платформонезависимости самих проектных файлов.

Проектный файл может быть использован для компиляции проектов, расположенных в разных каталогах. Наглядным примером может служить pro-файл Examples.pro каталога примеров, содержащихся в электронном архиве, расположенном на FTP-сервере издательства «БХВ-Петербург» и доступном по ссылке **ftp://ftp.bhv.ru/9785977536783.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение 4*). Этот файл выглядит примерно так:

```
TEMPLATE = subdirs
SUBDIRS = Example1 Example2 .... ExampleN
```

> ***Удаление объектных файлов***
>
> Для удаления объектных файлов проекта служит опция `clean`, а для удаления объектных файлов, созданных проектом исполняемых модулей, и созданных make-файлов, существует опция `distclean`. Например: `make distclean`.

Рекомендации для проекта с Qt

При реализации файлы классов лучше всего разбивать на две отдельные части. Часть определения класса помещается в заголовочный файл с расширением h, а реализация класса — в файл с расширением cpp. Важно помнить, что в заголовочном файле с определением класса должна содержаться директива препроцессора `#ifndef`. Смысл этой директивы состоит в том, чтобы избежать конфликтов в случае, когда один и тот же заголовочный файл будет включаться в исходные файлы более одного раза.

```
#ifndef _MyClass_h_
#define _MyClass_h_
class MyClass {
...
};
#endif //_MyClass_h_
```

Эту конструкцию можно так же заменить на эквивалентную с использованием прагмы, тогда код заголовочного файла будет смотреться более компактно:

```
#pragma once
class MyClass {
...
};
```

По традиции заголовочный файл, как правило, носит имя содержащегося в нем класса. В заголовочных файлах, в целях более быстрой компиляции, для указателей на типы данных используется предварительное объявление для типа данных, а не прямое включение посредством директивы `#include`. В начале определения класса содержится макрос

Q_OBJECT для MOC — это необходимо, если ваш класс использует сигналы и слоты, а в других случаях, если у вас нет нужды в метаинформации, этим макросом можно пренебречь. Но нужно учитывать то обстоятельство, что из-за отсутствия метаинформации нельзя будет использовать приведение типа qobject_cast<T>(obj).

```
class MyClass : public QObject {
Q_OBJECT
public:
    MyClass();
...
};
```

Основная программа должна быть реализована в отдельном файле, который является «стартовой площадкой» приложения. Такому файлу принято давать имя main.cpp. Это удобно еще и потому, что проект может состоять из сотен файлов, и если следовать указанному правилу, то найти отправную точку всего проекта не составит труда.

Соблюдение приведенных здесь рекомендаций может сослужить хорошую службу. Проектам свойственно со временем расширяться, поэтому неплохо с самого начала придерживаться определенной структуры, чтобы потом чувствовать себя в своих и чужих проектах, придерживающихся принятой в Qt структуры, «как рыба в воде».

Метаобъектный компилятор MOC

Метаобъектный компилятор (MOC, Meta Object Compiler), по сути дела, является не компилятором, а препроцессором, который исполняется в ходе компиляции приложения, создавая, в соответствии с определением класса, дополнительный код на языке C++. Это происходит из-за того, что определения сигналов и слотов в исходном коде программы недостаточно для компиляции. Сигнально-слотовый код должен быть преобразован в код, понятный для компилятора C++. Код сохраняется в файле с прототипом имени: moc_<filename>.cpp.

> ***НЕ ВКЛЮЧАЙТЕ MOC-ФАЙЛЫ В КОНЕЦ ОСНОВНОГО ФАЙЛА!***
>
> Созданные moc-файлы не стоит включать с помощью команды препроцессора #include "main.moc" в конец основного файла. Например, так:
>
> ```
> #include <QtGui>
> int main(int argc, char** argv)
> {
>     QApplication app(argc, argv);
>     ...
>     return app.exec();
> }
> #include "main.moc"
> ```
>
> Лучше, если они будут отдельно откомпилированы и подсоединены компоновщиком к основной программе. Хотя при написании демонстрационных программ этим правилом можно пренебречь, чтобы разместить весь код в одном файле main.cpp.

Если вы работаете с файлами проекта, то о существовании MOC можете и не догадываться, ведь в этом случае управление MOC автоматизировано. Для создания moc-файла вручную можно воспользоваться следующей командой:

```
moc -o proc.moc proc.h
```

После ее исполнения MOC создаст дополнительный файл proc.moc.

Для каждого класса, унаследованного от QObject, MOC предоставляет объект класса, унаследованного от QMetaObject. Объект этого класса содержит информацию о структуре объекта — например, сигнально-слотовые соединения, имя класса и структуру наследования.

Компилятор ресурсов RCC

Почти любая программа так или иначе обращается к сторонним ресурсам — растровым изображениям, файлам перевода и т. п. Такие обращения не являются достаточно надежным и эффективным способом, поскольку сторонние ресурсы могут быть удалены или недоступны по каким-либо иным причинам, что, несомненно, может отразиться на правильной работе программы, ее внешнем облике и работоспособности. Компилятор ресурсов предоставляет возможность внедрения таких файлов в исполняемые модули, для того чтобы приложение получало доступ к требуемым ресурсам в процессе его исполнения. Существуют специальные соглашения об именовании, благодаря которым можно однозначно обращаться к таким ресурсам. Все необходимые для использования файлы (ресурсы) должны быть вместе с их путями описаны в специальном файле с расширением qrc (Qt Resource Collection, коллекция ресурсов Qt). Это описание выполняется в нотации XML. Например:

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/open.png</file>
    <file>images/quit.png</file>
</qresource>
</RCC>
```

Файл нашего примера будет подвергнут анализу компилятором ресурсов (утилитой rcc) для создания из файлов open.png и quit.png одного исходного файла C++, содержащего все их данные, которые будут компилироваться и компоноваться вместе с остальными файлами проекта. Все данные ресурса хранятся в файле C++ в виде одного большого массива.

Такой подход дает уверенность в том, что необходимые ресурсы всегда доступны, что поможет избежать проблем неправильной установки необходимых для исполняемой программы файлов. Сам же qrc-файл должен быть указан в pro-файле в секции RESOURCES, для того чтобы утилита qmake учла информацию из файла ресурса. Например:

```
RESOURCES = images.qrc
```

Для того чтобы воспользоваться файлом open.png, а точнее, представленным в нем растровым изображением, можно поступить следующим образом:

```
plbl->setPixmap(QPixmap(":/images/open.png"));
```

В нашем случае все растровые изображения размещены в каталоге images, и это идеальный случай. Но не всегда представляется возможным размещать файлы ресурсов там, где удобно. Пути для обращения к этим файлам не всегда короткие, что накладывает дополнительные трудности в прописывании длинных путей для обращения к файлам ресурсов. Это неудобство решается использованием синонимов, которые прописываются в xml-файле в теге <file> при помощи опции alias, например:

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file alias="open.png">../../../very/long/path/images/open.png</file>
    <file alias="quit.png">../../../very/long/path/images/quit.png</file>
</qresource>
</RCC>
```

Теперь мы можем обращаться к нашим файлам ресурсов из программы следующим образом:

```
plbl->setPixmap(QPixmap(":/open.png"));
```

Без использования синонимов обращение к файлу ресурса выглядело бы так:

```
plbl->setPixmap(QPixmap(":/very/long/path/images/open.png"));
```

Согласитесь, вариант обращения с использованием синонимов выглядит гораздо симпатичнее.

> ***Класс QResource для чтения файла, находящегося в ресурсе***
>
> Модуль `QtCore` предоставляет класс `QResource` для чтения файла, находящегося в ресурсе, напрямую, без дополнительных промежуточных операций. Для этого нужно создать объект и в конструктор класса `QResource` передать имя файла. Затем при помощи метода `data()` можно получить указатель на массив данных переданного файла.

Структура Qt-проекта

Мы рассмотрели утилиты для Qt-проекта по отдельности. Теперь давайте соберем их все вместе, чтобы понять, как они взаимодействуют. Структура проекта Qt очень проста — помимо файлов исходного кода на языке C++ обычно имеется файл проекта с расширением pro. Из него вызовом утилиты qmake и создается make-файл. Этот make-файл содержит в себе все необходимые инструкции для создания готового исполняемого модуля (рис. 3.3).

В make-файле содержится вызов MOC для создания дополнительного кода C++ и необходимых заголовочных файлов. Если проект содержит qrc-файл, то будет также создан файл C++, содержащий данные ресурсов. После этого все исходные файлы компилируются C++-компилятором в файлы объектного кода, которые объединяются компоновщиком link в готовый исполняемый модуль.

Методы отладки

Нет программ без ошибок и дефектов (bug, ошибка или, попросту, баг). И в процессе разработки программ часто возникают проблемы с их обнаружением. Разработчикам приходится тратить немалую часть рабочего времени на то, чтобы найти и устранить имеющиеся баги. К средствам, помогающим снизить их количество, можно отнести:

- предоставление исходного кода для просмотра другими разработчиками (code review);
- создание классов для автоматизированных тестов, подробно описанных в *главе 45*.

Ошибки можно минимизировать, но, в любом случае, полностью их избежать не удастся, и если в вашу программу вдруг закрался коварный баг, то самым первым средством, помогающим в нелегком труде его поиска, станет *отладчик*. Роль отладчика заключается в предоставлении оболочки, в которой можно отслеживать изменение данных во время выполнения программы, благодаря чему можно узнать, почему созданная вами программа ведет себя не так, как вы это задумывали. Благодаря платформонезависимости Qt разработчик может для отладки своих программ использовать любой из полюбившихся ему отладчиков — например, GDB или отладчик, встроенный в Microsoft Visual Studio. Если вы еще серьезно не сталкивались с процессом отладки программ, то рекомендую начать с отладчика без графического пользовательского интерфейса, поскольку подача команд в диалоговом

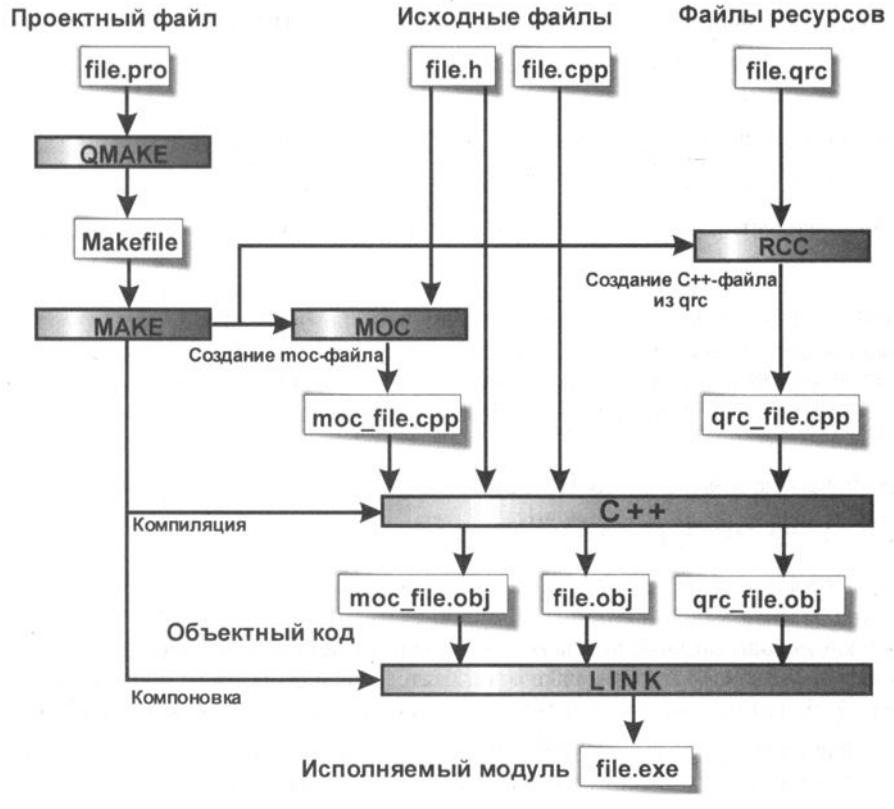


Рис. 3.3. Схема создания исполняемого модуля

режиме поможет вам понять работу отладчика как такового и в будущем по достоинству оценить отладчики, обладающие графическим интерфейсом. Поэтому мы подробнее рассмотрим отладчик GDB, доступный как для Windows, так и для Linux, и Mac OS X.

Отладчик GDB (GNU Debugger)

GDB — это самое привычное средство для отладки программ в ОС UNIX. Работа с этим отладчиком обычно осуществляется из командной строки, хотя можно воспользоваться и оболочками, предоставляющими возможность работы с этим отладчиком в интерактивном режиме, — вот некоторые из них: XXGDB, DDD, KDBG. Пожалуй, идеальной средой для работы с отладчиком в интерактивном режиме является IDE, в этом случае все необходимое находится «под рукой», поэтому среда разработки Qt Creator тоже имеет возможность использования этого отладчика. Если вы не собираетесь работать с отладчиком напрямую и предпочтете использовать IDE, то дальнейшее описание GDB можете пропустить. Но если вы хотите разобраться в процессе отладки, то давайте создадим программу, заведомо содержащую проблемный код (листинг 3.1).

Листинг 3.1. Проблемный код

```
void bug()
{
    int  n  = 3;
    int* pn = &n;
```

```
    // Ошибка!
    delete pn;
}

int main()
{
    bug();
    return 0;
}
```

В листинге 3.1 из функции `main()` осуществляется вызов функции `bug()`, в которой создается и инициализируется переменная `n`, а после присвоения указателю `pn` ее адреса вызовом оператора `delete` выполняется попытка освобождения памяти, используемой переменной `n`. Поскольку память не была выделена динамически, эта операция неизбежно приведет к ошибке.

Откомпилируем программу с параметром `-g`, чтобы в исполняемый файл была включена информация, необходимая для отладчика:

```
g++ -g bug.cpp -o bug
```

> ***Примечание***
>
> В рассматриваемом случае я не стал создавать традиционного для Qt pro-файла и ограничился вызовом компилятора напрямую. Здесь это проще. В случаях же с pro-файлами Qt, для того чтобы получить исполняемый файл с включенной отладочной информацией, переменная pro-файла `CONFIG` должна содержать значения `debug` или `debug_and_release`.

Отладчик можно запустить следующим образом, указав имя программы, предназначенной для отладки:

```
gdb bug.exe
```

> ***Поиск проблемного места при помощи core-файла***
>
> Кроме имени программы, в GDB можно дополнительно передавать и core-файл, сгенерированный операционной системой после аварийного завершения программы. Это очень удобно, так как можно не загружать программу на выполнение в отладчик, а найти проблемное место при помощи core-файла. К сожалению, ОС Windows не генерирует подобных файлов, поэтому в дальнейшем будут описаны приемы работы с отладчиком GDB, применимые на обеих ОС (Windows и Linux).

После этого отладчик отобразит строку приглашения следующего вида:

(gdb)

Запустим теперь саму программу под отладчиком. Для этого нужно ввести команду `run`:

(gdb) `run`

В результате мы получим сообщения, сигнализирующие о ненормальном завершении программы. Для того чтобы разобраться в проблеме, нужно просмотреть стек программы. Это делается при помощи команды `where`:

(gdb) `where`

Вывод отладчика будет примерно следующим:

```
#0  0x77f767ce in _libmsvcrt_a_iname ()
...
```

```
#6  0x00401305 in operator delete(void*) ()
#7  0x004012ae in bug() () at bug.cpp:5
#8  0x004012df in main () at bug.cpp:10
```

Заметьте, что функция `main()` вызвала в десятой строке программы (см. листинг 3.1) функцию `bug()`, а вызов пятой строки этой функции создал проблему.

С помощью команды `up` можно подняться по стеку программы на определенное количество уровней. Давайте поднимемся на один уровень — это будет соответствовать функции `bug()`:

(gdb) `up 1`

Отладчик покажет следующее:

```
#7  0x004012ae in bug() () at bug.cpp:5
5           delete pn;
```

Для того чтобы узнать значение какой-либо локальной переменной функции, нужно подать команду `print`. Давайте проделаем это для переменной `n`:

(gdb) `print n`

В ответ отладчик покажет ее значение:

```
$1 = 3
```

Установка *контрольных точек* (break points) осуществляется в отладчике при помощи команды `break`. Установим точку в функции `bug()` и перезапустим нашу программу:

(gdb) `break bug`

(gdb) `run`

Отладчик остановится на заданной нами контрольной точке и покажет следующее:

```
Breakpoint 1, bug() () at bug.cpp:3
3           int  n  = 3;
```

Для того чтобы перейти на следующую строку, воспользуемся командой `next`. Эта команда выполняет код построчно без перехода внутрь тела функции:

(gdb) `next`

```
4           int* pn = &n;
```

Отсюда видно, что отладчик перешел с третьей строки на четвертую. Если понадобится выполнить строки кода, включая строку внутри тела функции, то для этого нужно было бы воспользоваться командой `step`. Например, мы могли бы установить контрольную точку в функции `main()` и при помощи команды `step` войти внутрь функции `bug()`. В табл. 3.2 сведены наиболее часто используемые команды отладчика.

Таблица 3.2. *Некоторые команды отладчика GDB*

Команда	Описание
`quit`	Выход из отладчика
`help`	Вывод справочной информации. Если дополнительным параметром указана какая-либо команда, то выводится полная справочная информация по этой команде
`run`	Запуск программы

Таблица 3.2 (окончание)

Команда	Описание
attach	Присоединение отладчика к запущенному процессу с указанным идентификатором
detach	Отсоединение отладчика от присоединенного процесса
break	Установка контрольной точки. Вызов команды без параметра установит точку на следующей исполняемой инструкции. В качестве параметра можно передавать имя функции, номер строки и смещение. Если требуется, можно указать имя конкретного исходного файла в виде <имя файла>:<номер строки> или <имя файла>:<имя функции>
tbreak	Аналогична команде break с той лишь разницей, что контрольная точка будет удалена после ее достижения
clear	Удаление контрольной точки. Вызов команды без параметра удалит контрольную точку на следующей исполняемой инструкции. В качестве параметра можно передавать имя функции, номер строки и смещение. Если требуется, то можно указать имя конкретного исходного файла в виде: <имя файла>:<номер строки> или <имя файла>:<имя функции>
delete	Удаление всех контрольных точек
disable	Отключение всех контрольных точек
enable	Включение всех контрольных точек
continue	Продолжение выполнения программы. Дополнительным параметром можно указать количество игнорирований контрольной точки
next	Выполнение следующей строки исходного кода программы. Дополнительным параметром можно задать количество выполняемых строк
step	Выполнение следующей строки исходного кода программы. Дополнительным параметром можно задать количество выполняемых строк. В отличие от next, при вызове функции происходит вход в нее и остановка
until	Продолжение выполнения программы до выхода из функции

Прочие методы отладки

Одним из стандартных приемов отладки является вставка в исходный код операторов вывода, что позволяет увидеть значения переменных и сравнить их с ожидаемыми значениями. Такой способ отладки часто используется разработчиками, поскольку ничего не стоит поместить эти операторы или оформить их в виде отдельного дамп-метода. В Qt примером такого подхода является метод QObject::dumpObjectInfo(), который выводит на экран метаинформацию объекта.

Qt предоставляет макросы и функции для отладки, при помощи которых можно встраивать в саму программу различного рода проверки и вывод тестовых сообщений.

В заголовочном файле QtGlobal содержатся определения двух макросов Q_ASSERT() и Q_CHECK_PTR():

- Q_ASSERT() принимает в качестве аргумента значение булевого типа и выводит предупреждающее сообщение, если это значение не равно true;
- Q_CHECK_PTR() принимает указатель и выводит предупреждающее сообщение, если переданный указатель равен 0, а это означает, что либо указатель не был инициализирован, либо операция по выделению памяти прошла неудачно.

Qt предоставляет глобальные функции `qDebug()`, `qWarning()` и `qFatal()`, которые также определены в заголовочном файле QtGlobal. Их применение похоже на функцию `printf()`. Как и в `printf()`, в эти функции передаются форматированная строка и различные параметры. В Microsoft Visual Studio вывод этих функций выполняется в окно отладчика, а в ОС Linux — в стандартный поток вывода ошибок.

> ***Особенность функции `qFatal()`***
>
> Вызов функции `qFatal()` после вывода сообщения сразу завершает работу всего приложения.

Если потребуется перенаправить поток вывода сообщения, нужно создать и установить свою собственную функцию для управления выводом. Устанавливается она при помощи функции `qInstallMessageHandler()`. Этой функции в качестве аргумента передается адрес на функцию, управляющую сообщениями и имеющую следующий прототип:

```
void fct(QtMsgType type, const QMessageLogContext& context, const QString& msg)
```

На месте `fct` должно стоять имя функции. Первый аргумент представляет собой тип сообщения, принимающего одно из значений перечисления `QtMsgType`: `QtDebugMsg`, `QtWarningMsg` или `QtFatalMsg`. Второй аргумент — дополнительная информация о сообщении, а третий — само сообщение. Проиллюстрируем сказанное фрагментом кода (листинг 3.2).

Листинг 3.2. Перенаправление потока вывода сообщений

```
void messageToFile(QtMsgType                type,
                   const QMessageLogContext& context,
                   const QString&           msg
                  )
{
    QFile file("protocol.log");
    if(!file.open(QIODevice::WriteOnly | QIODevice::Text | QIODevice::Append))
        return;

    QString strDateTime =
        QDateTime::currentDateTime().toString("dd.MM.yy-hh:mm");

    QTextStream out(&file);
    switch (type) {
     case QtDebugMsg:
         out << strDateTime << "Debug: " << msg
             << ", " << context.file << endl;
         break;
     case QtWarningMsg:
         out << strDateTime << "Warning: " << msg
             << ", " << context.file << endl;
         break;
     case QtCriticalMsg:
         out << strDateTime << "Critical: " << msg
             << ", " << context.file << endl;
         break;
```

```
    case QtFatalMsg:
        out << strDateTime << "Fatal: " << msg
            << ", " << context.file << endl;
        abort();
    }
}

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    qInstallMessageHandler(messageToFile);
...
}
```

Теперь все сообщения `qDebug()`, `qWarning()` и `qFatal()` будут записываться в файл protocol.log, а не выводиться на консоль. Это очень удобно для изучения ошибок и странного поведения программы, которые произошли на стороне тестера и пользователей. Теперь вы всегда сможете попросить прислать вам файл protocol.log для более углубленного исследования.

> ***Возврат к выводу сообщений на консоль***
>
> Если в программе понадобится снова выводить сообщения на консоль, то нужно просто вызвать глобальную функцию `qInstallMessageHandler()` и передать ей значение `0;`.

Для облегчения процесса отладки рекомендуется присваивать всем объектам имена. Тогда эти объекты можно будет всегда найти, вызвав метод `QObject::objectName()`. Это даст возможность в процессе работы программы воспользоваться методом `QObject::dumpObjectInfo()`, который позволяет отобразить внутреннюю информацию объекта.

При отладке можно также воспользоваться установкой фильтра событий для объекта класса `QCoreApplication` — в этом случае такой фильтр будет являться наипервейшим объектом, получающим и обрабатывающим события всех объектов приложения (см. *главу 15*).

Самый простой способ операции вывода в Qt — использование объекта класса `QDebug`. Этот объект очень напоминает стандартный объект потока вывода в C++ `cout`. Например, вывести сообщение в отладчике или на консоли с помощью функции `qDebug()` можно следующим образом:

```
qDebug() << "Test";
```

Эта функция создает объект класса потока `QDebug`, передавая в его конструктор упомянутый ранее аргумент `QtDebugMsg`. Можно было бы, конечно, поступить и так:

```
QDebug(QtDebugMsg) << "Test";
```

Но, как вы видите, предыдущая строка выглядит более компактно, поэтому рекомендую пользоваться именно ей.

Важно понимать, что вывод информации с помощью функции `qDebug()` происходит при отладочных и релизных компоновках. Если вывод информации в релизной версии не желателен, и все сообщения должны отображаться только в отладочной версии программы, то можно реализовать пустую функцию для управления выводом `dummyOutput()` и установить ее в `qInstallMessageHandler()`, как показано в листинге 3.3.

Листинг 3.3. Скрытие информации, выводимой функциями qDebug(), qWarning() и qFatal()

```
void dummyOutput(QtMsgType, const QMessageLogContext&, const QString&)
{
}

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
#ifndef QT_DEBUG
    qInstallMessageHandler(dummyOutput);
#endif
}
```

При этом следует использовать вместо функции `qDebug()` следующую запись:

```
qDebug() << "Test1" << 123 << "Test2" << 456;
```

Теперь, делая релиз своей программы, вы можете быть совершенно уверены в том, что весь вывод функциями `qDebug()`, `qWarning()` и `qFatal()` информации, предназначенной для отладки, в релизной версии будет скрыт от посторонних глаз.

В отладочном выводе можно также использовать манипуляторы — такие как, например, `hex` и `uppercasedigits`. Первый отвечает за то, чтобы все целочисленные значения выводились в шестнадцатеричной форме, а второй следит за тем, чтобы все строчные буквы преобразовывались в заглавные. В качестве примера реализуем небольшую программу вывода значения переменной `n` в шестнадцатеричном виде и с заглавными буквами:

```
int n = 7777;
qDebug() << "DEC:" << n << "= HEX:" << hex << uppercasedigits << n;
```

Вот что мы увидим на экране:

DEC: 7777 =HEX: 1E61

После каждой секции вывода `qDebug()` помещается пробел. Например:

```
qDebug() << 1 << 2 << 3 << 4;
```

Вывод выглядит так:

1 2 3 4

При помощи метода `nospace()` класса `QDebug` можно воспрепятствовать этому:

```
qDebug().nospace() << 1 << 2 << 3 << 4;
```

Теперь вывод будет выглядеть так:

1234

Глобальные определения Qt

Qt содержит в заголовочном файле QtGlobal некоторые макросы и функции, которые могут быть очень полезны при написании программ.

Шаблонные функции `qMax(a, b)` и `qMin(a, b)` используются для определения максимального и минимального из двух переданных значений:

```
int n = qMax<int>(3, 5); // n = 5
int n = qMin<int>(3, 5); // n = 3
```

Функция `qAbs(a)` возвращает абсолютное значение:

```
int n = qAbs(-5); // n = 5
```

Функция `qRound()` округляет передаваемое число до целого:

```
int n = qRound(5.2);  // n = 5
int n = qRound(-5.2); // n = -5
```

Функция `qBound()` возвращает значение, находящееся между минимумом и максимумом:

```
int n = qBound(2, 12, 7); // n = 7
```

А вот еще одна интересная функция. Дело в том, что сравнение двух значений с плавающей точкой на точное равенство является в программировании одной из частых ошибок. Функция `qFuzzyCompare()` берет в этом случае всю ответственность за правильное сравнение на себя. Она принимает два значения типа `double` или `float` и возвращает логическое значение `true`, если переменные считаются равными, в противном случае она возвращает значение `false`. Само сравнение осуществляется в относительной манере, когда точность для сравнения увеличивается с уменьшением численных значений сравниваемых величин. Поэтому единственное значение, которое представляет сложность для этой функции, — это нулевое значение. Но есть решение и для этой задачи. Нужно просто сделать так, чтобы сравниваемые значения были либо равны, либо больше 1,0. Например:

```
double dValue1 = 0.0;
double dValue2 = myFunction();
if (qFuzzyCompare(1 + dValue1, 1 + dValue2)) {
    // Значения равны
}
```

ФУНКЦИЯ `qFuzzyCompare()` И МОДУЛЬНЫЕ ТЕСТЫ

Эта функция также может быть очень полезной для написания модульных тестов, описанных в *главе 45*.

В табл. 3.3 приведен список типов Qt, которые можно использовать при программировании.

Таблица 3.3. *Таблица целых типов Qt*

Тип Qt	Эквивалент C++	Размер
qint8	signed char	8 битов
quint8	unsigned char	8 битов
qint16	short	16 битов
quint16	unsigned short	16 битов
qint32	Int	32 бита
quint32	unsigned int	32 бита
qint64	__int64 **или** long long	64 бита
quint64	unsigned __int64 **или** unsigned long long	64 бита

Таблица 3.3 (окончание)

Тип Qt	Эквивалент C++	Размер
qlonglong	То же самое, что и qint64	64 бита
qulonglong	То же самое, что и quint64	64 бита

Как видно из табл. 3.3, самыми спорными являются типы qint64 и quint64. Давайте проверим правильность указанных для них в таблице значений битов, а заодно их минимальные и максимальные значения:

```
qDebug() << "Number of bits for qint64 ="  << (sizeof(qint64) * 8);
qDebug() << "Minimum of qint64 = -"        << ~(~quint64(0) >> 1);
qDebug() << "Maximum of qint64 ="          << (~quint64(0) >> 1);
qDebug() << "Number of bits for quint64 =" << (sizeof(quint64) * 8);
qDebug() << "Minimum of quint64 ="         << 0;
qDebug() << "Maximum of quint64 ="         << ~quint64(0);
```

Результат выполнения:

```
Number of bits for qint64 = 64
Minimum of qint64 = -9223372036854775808
Maximum of qint64 = 9223372036854775807
Number of bits for quint64 = 64
Minimum of quint64 = 0
Maximum of quint64 = 18446744073709551615
```

Информация о библиотеке Qt

Иногда бывает очень полезно получить информацию о той библиотеке, которая используется на вашем компьютере в настоящий момент. Например, вам захотелось узнать, в каком каталоге Qt хранит свои файлы расширений (plug-ins), или вам необходимо уточнить текущую версию Qt, и т. п. За получение такой информации отвечает класс QLibraryInfo и предоставляет для этого целый ряд статических методов. Продемонстрируем их применение на небольшом примере (листинг 3.4).

Листинг 3.4. Использование статических функций класса QLibraryInfo

```
#include <QtCore>
int main(int argc, char** argv)
{
    qDebug() << "License Products:"
             << QLibraryInfo::licensedProducts();
    qDebug() << "Licensee:"
             << QLibraryInfo::licensee();
    qDebug() << "Is Debug Build:"
             << QLibraryInfo::isDebugBuild();

    qDebug() << "Locations";
    qDebug() << "  Headers:"
             << QLibraryInfo::location(QLibraryInfo::HeadersPath);
```

```
    qDebug() << "  Libraries:"
             << QLibraryInfo::location(QLibraryInfo::LibrariesPath);
    qDebug() << "  Binaries:"
             << QLibraryInfo::location(QLibraryInfo::BinariesPath);
    qDebug() << "  Prefix"
             << QLibraryInfo::location(QLibraryInfo::PrefixPath);
    qDebug() << "  Documentation: "
             << QLibraryInfo::location(QLibraryInfo::DocumentationPath);
    qDebug() << "  Plugins:"
             << QLibraryInfo::location(QLibraryInfo::PluginsPath);
    qDebug() << "  Data:"
             << QLibraryInfo::location(QLibraryInfo::DataPath);
    qDebug() << "  Settings:"
             << QLibraryInfo::location(QLibraryInfo::SettingsPath);

    qDebug() << "  Examples:"
             << QLibraryInfo::location(QLibraryInfo::ExamplesPath);
}
```

Вот вывод этой программы для версии Qt 5.10.0, установленной на компьютере с операционной системой Windows:

```
License Products: "OpenSource"
Licensee: "Open Source"
Is Debug Build: false
Locations
  Headers: "C:\Qt\5.10.0\include"
  Libraries: "C:\Qt\5.10.0\lib"
  Binaries: "C:\Qt\5.10.0\bin"
  Prefix "C:\Qt\5.10.0"
  Documentation:  "C:\Qt\5.10.0\doc"
  Plugins: "C:\Qt\5.10.0\plugins"
  Data: "C:\Qt\5.10.0"
  Settings: "C:\Qt\5.10.0"
  Examples: "C:\Qt\5.10.0\examples"
```

Резюме

В этой главе мы узнали, как выглядит типичный проект с использованием Qt, и какие процессы протекают «за кулисами» при создании готового исполняемого программного модуля.

Мы познакомились с утилитой qmake, берущей на себя всю работу по созданию из файлов проекта make-файлов для любой платформы. Мы подробнее узнали о препроцессоре MOC, который при запуске создает дополнительный код поддержки сигналов и слотов, и провели небольшой экскурс в глобальные определения Qt. Мы также рассмотрели возможности и методы отладки Qt-программ, специальные макросы, предназначенные для этих целей, и особенности применения отладчика GDB.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/03-510/** или с помощью следующего QR-кода (рис. 3.4):



Рис. 3.4. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 4

Библиотека контейнеров

Ты Дерево, Животное или Минерал?
Льюис Кэрролл, «Алиса в Зазеркалье»

Одна из самых распространенных задач в программировании заключается в организации обработки групп элементов. Реализация и отладка программ с использованием подобного рода структур отнимает у разработчиков много времени, так как они каждый раз вынуждены решать одни и те же задачи. Для решения этой проблемы *библиотека контейнеров* предоставляет набор часто используемых классов, на правильную работоспособность которых можно положиться. Это позволяет разработчику сконцентрироваться на реализации самого приложения и не вникать в детали реализации используемых контейнерных классов.

Qt предоставляет библиотеку контейнеров, именуемую Tulip и являющуюся составной частью ядра Qt, поэтому ее понимание очень важно для дальнейшего изучения Qt. Эта библиотека не только очень похожа на STL (Standard Template Library, стандартная библиотека шаблонов), но и совместима с ней. В данном случае разработчик может свободно выбирать, чем ему лучше воспользоваться: Tulip или STL. Предпочтительнее выбрать первую библиотеку, потому что Tulip является частью Qt и активно используется в ней. Вторым аргументом в пользу применения Tulip может служить утверждение, что эта библиотека в соответствии со спецификой классов Qt оптимизирована по производительности и расходу памяти. Нелишне будет отметить, что использование в программах шаблонных классов, на основе которых построены контейнеры, заметно увеличивает размер исполняемых программ. Это связано с тем, что в каждом объектном файле реализации класса, использующего контейнеры, находится созданный компилятором код контейнеров с нужными типами, и этот код может повторяться. Библиотека Tulip создавалась именно с учетом этих обстоятельств и, кроме того, оптимизирована для заметного уменьшения размера объектного кода.

Реализация классов Tulip расположена в модуле `QtCore`. В основе библиотеки Tulip (как и в STL) лежат три понятия:

- контейнерные классы (контейнеры);
- алгоритмы;
- итераторы.

Их взаимосвязь показана на рис. 4.1.

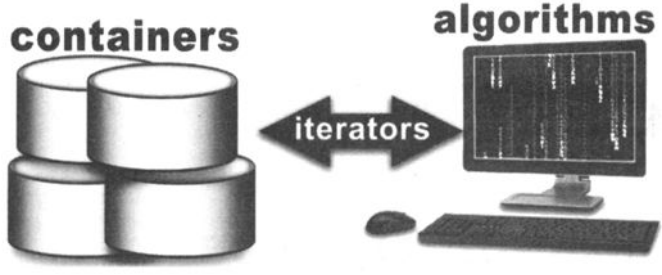


Рис. 4.1. Взаимосвязь контейнеров, итераторов и алгоритмов

Контейнерные классы

Контейнерные классы — это классы, которые в состоянии хранить в себе элементы различных типов данных. Почти все контейнерные классы в Qt реализованы как шаблонные и, таким образом, они могут хранить данные любого типа. Основная идея шаблона состоит в создании родового класса, который определяется при создании объекта этого класса. Классы контейнеров могут включать целые серии других объектов, которые, в свою очередь, тоже могут являться контейнерами.

Чтобы правильно подобрать контейнер для конкретного случая, очень важно правильно понимать различия разновидностей контейнеров. От этого в значительной степени зависит скорость работы кода и эффективность использования памяти. Qt предоставляет две категории разновидностей классов контейнеров: *последовательные* (sequence containers) и *ассоциативные* (associative containters).

Последовательные контейнеры — это упорядоченные коллекции, где каждый элемент занимает определенную позицию. Позиция элемента зависит от места его вставки. К последовательным контейнерам относятся: *вектор* (vector), *список* (list), *стек* (stack) и *очередь* (queue). Соответственно, Qt содержит пять классов этой категории:

- `QVector<T>` — вектор;
- `QList<T>` — список;
- `QLinkedList<T>` — двусвязный список;
- `QStack<T>` — стек;
- `QQueue<T>` — очередь.

Ассоциативные контейнеры — это коллекции, в которых позиция элемента зависит от его значения, то есть после занесения элементов в коллекцию порядок их следования будет задаваться их значениями. К ассоциативным контейнерам относятся: *множество* (set), *словарь* (map) и *хэш* (hash). Классы этой категории:

- `QSet<T>` — множество;
- `QMap<K,T>` — словарь;
- `QMultiMap<K,T>` — мультисловарь;
- `QHash<K,T>` — хэш;
- `QMultiHash<K,T>` — мультихэш.

Во всех контейнерах этих двух групп доступны операции, приведенные в табл. 4.1. Обратите, пожалуйста, внимание на некоторые исключения для класса `QSet<T>`.

Таблица 4.1. *Операторы и методы, определенные во всех контейнерных классах*

Оператор, метод	Описание
`==` и `!=`	Операторы сравнения, равно и не равно
`=`	Оператор присваивания
`[]`	Оператор индексации. Исключение составляют только классы `QSet<T>` и `QLinkedList<T>` — в них этот оператор не определен
`begin()` и `constBegin()`	Методы, возвращающие итераторы, установленные на начало последовательности элементов контейнера. Для класса `QSet<T>` возвращаются только константные итераторы
`end()` и `constEnd()`	Методы, возвращающие константные итераторы, установленные на конец последовательности элементов контейнера
`clear()`	Удаление всех элементов контейнера
`insert()`	Операция вставки элементов в контейнер
`remove()`	Операция удаления элементов из контейнера
`size()` и `count()`	Оба метода идентичны — возвращают количество элементов контейнера, но применение первого предпочтительно, так как соответствует STL
`value()`	Возвращает значение элемента контейнера. В `QSet<T>` этот метод не определен
`empty()` и `isEmpty()`	Возвращают `true`, если контейнер не содержит ни одного элемента. Оба метода идентичны, но применение первого предпочтительно, так как соответствует STL

Сохраняйте в контейнерах указатели на объекты, наследуемые от класса QObject

Важно не забывать о том, что классы, унаследованные от класса `QObject`, не имеют доступного конструктора копирования и оператора присваивания, поскольку они находятся в секции `private`. Следовательно, их объекты не могут храниться в контейнерах, поэтому нужно сохранять в контейнерах не сами объекты, наследуемые от класса `QObject`, а указатели на них.

Для проверки наличия в контейнере элементов пользуйтесь методом empty()

Если вам нужно проверить, содержит контейнер элементы или нет, воспользуйтесь для этого методом `empty()`, а не методом `size()`. Причина проста: метод `empty()` для всех контейнеров выполняется за один шаг, а `size()` может потребовать для контейнера списка линейных затрат, что способно существенно снизить скорость вашего алгоритма.

Итераторы

Наверняка вам потребуется перемещаться по элементам контейнера. Для этих целей предназначены *итераторы*. Итераторы позволяют абстрагироваться от структуры данных контейнеров, то есть, если в какой-либо момент вы решите, что применение другого типа контейнера было бы гораздо эффективнее, то все, что вам надо будет сделать, — это просто заменить тип контейнера нужным. На остальном коде, использующем итераторы, это никак не отразится.

Qt предоставляет два стиля итераторов:

- итераторы в стиле Java;
- итераторы в стиле STL.

В качестве альтернативы существует вариант обхода элементов при помощи ключевого слова `foreach`.

Итераторы в стиле Java

Итераторы в стиле Java очень просты в использовании. Они были разработаны специально для программистов, не имеющих опыта работы с контейнерами STL. Основным их отличием от последних является то обстоятельство, что они указывают не на сам элемент, а на двух его соседей. Таким образом, вначале итератор укажет на положение перед первым элементом контейнера, а с каждым вызовом метода `next()` (см. табл. 4.2) будет перемещать указатель на одну позицию вперед. Но на самом деле итераторы в стиле Java представляют собой объекты, а не указатели. Их применение в большинстве случаев делает код более компактным, чем при использовании итераторов в стиле STL:

```
QList<QString> list;
list << "Within Temptation" << "Anubis" << "Mantus";
QListIterator<QString> it(list);
while(it.hasNext()) {
    qDebug() << "Element:" << it.next();
}
```

В табл. 4.2 приведены методы класса `QListIterator`, применимые также для классов `QLinkedListIterator`, `QVectorIterator`, `QHashIterator`, `QMapIterator` и `QSetIterator`. Эти итераторы являются константными, соответственно изменение значений элементов, их вставка и удаление невозможны.

Таблица 4.2. *Методы `QListIterator`, `QLinkedListIterator`, `QVectorIterator`, `QHashIterator`, `QMapIterator`, `QSetIterator`*

Метод	Описание
`toFront()`	Перемещает итератор на начало списка
`toBack()`	Перемещает итератор на конец списка
`hasNext()`	Возвращает значение `true`, если итератор не находится в конце списка
`next()`	Возвращает значение следующего элемента списка и перемещает итератор на следующую позицию
`peekNext()`	Просто возвращает следующее значение без изменения позиции итератора
`hasPrevious()`	Возвращает значение `true`, если итератор не находится в начале списка
`previous()`	Возвращает значение предыдущего элемента списка и перемещает итератор на предыдущую позицию
`peekPrevious()`	Просто возвращает предыдущее значение без изменения позиции итератора
`findNext(const T&)`	Поиск заданного элемента в прямом направлении
`findPrevious(const& T)`	Поиск заданного элемента в обратном направлении

Если необходимо производить изменения в процессе прохождения итератором элементов, то для этого следует воспользоваться изменяющимися (mutable) итераторами. Их классы называются аналогично, но с добавлением фрагмента `Mutable`: `QMutableListIterator`, `QMutableHashIterator`, `QMutableLinkedListIterator`, `QMutableMapIterator` и `QmutableVectorIterator`. Метод `remove()` удаляет текущий элемент, а метод `insert()` производит вставку элемента на текущую позицию. При помощи метода `setValue()` можно присвоить элементу другое значение.

Давайте присвоим элементу списка `"Boney M"` значение `"Rolling Stones"`:

```
QList<QString> list;
list << "Beatles" << "ABBA" << "Boney M";
QMutableListIterator<QString> it(list);
while(it.hasNext()) {
    if (it.next() == "Boney M") {
        it.setValue("Rolling Stones");
    }
    qDebug() << it.peekPrevious();
}
```

Основным недостатком итераторов в стиле Java является то, что их применение, как правило, заметно увеличивает объем созданного объектного модуля, в сравнении с использованием итераторов стиля STL, которые мы сейчас и рассмотрим.

Итераторы в стиле STL

Итераторы в стиле STL немного эффективнее итераторов Java-стиля и могут быть использованы совместно с алгоритмами STL. Пожалуй, для разработчиков на языке C++ — это самый привычный тип итераторов. Итераторы в стиле STL можно представить как некоторые обобщенные указатели, ссылающиеся на элементы контейнера.

Вызов метода `begin()` из объекта контейнера возвращает итератор, указывающий на первый его элемент, а вызов метода `end()` возвращает итератор, указывающий на конец контейнера. Обратите внимание: именно на конец контейнера, а не на последний элемент, то есть на позицию, на которой мог бы быть размещен следующий элемент. Другими словами, этот итератор не указывает на элемент, а служит только для обозначения достижения конца контейнера (рис. 4.2).

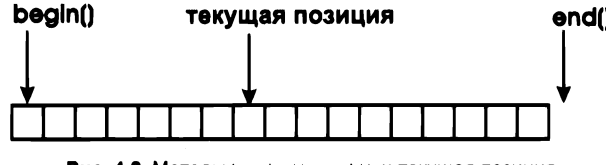


Рис. 4.2. Методы `begin()`, `end()` и текущая позиция

Операторы `++` и `--` объекта итератора производят перемещения на следующий или предыдущий элемент соответственно. Доступ к элементу, на который указывает итератор, можно получить при помощи операции разыменования `*`. Например:

```
QVector<QString> vec;
vec << "In Extremo" << "Blackmore's Night" << "Cultus Ferox";
QVector<QString>::iterator it = vec.begin();
```

```
for (; it != vec.end(); ++it) {
    qDebug() << "Element:" << *it;
}
```

На экране будет отображено:

```
Element: "In Extremo"
Element: "Blackmore's Night"
Element: "Cultus Ferox"
```

Обратите внимание, что для увеличения итератора `it` в цикле служит операция преинкрементации: `++it`, что позволяет избежать на каждом витке цикла сохранения старого значения, как скрытно осуществляется в инкрементации, и это делает цикл более эффективным.

При прохождении элементов в обратном порядке при помощи оператора `--` необходимо помнить, что он не симметричен с прохождением при помощи оператора `++`. Поэтому цикл должен в этом случае выглядеть следующим образом:

```
QVector<QString>::iterator it = vec.end();
for (;it != vec.begin();) {
    --it;
    qDebug() << "Element:" << *it;
}
```

На экране будет отображено:

```
Element: "Cultus Ferox"
Element: "Blackmore's Night"
Element: "In Extremo"
```

Если вы собираетесь только получать значения элементов, не изменяя их, то гораздо эффективнее использовать константный итератор `const_iterator`. При этом вам нужно будет пользоваться вместо методов `begin()` и `end()` методами `constBegin()` и `constEnd()`. Таким образом, наш пример примет следующий вид:

```
QVector<QString> vec;
vec << "In Extremo" << "Blackmore's Night" << "Cultus Ferox";
QVector<QString>::const_iterator it = vec.constBegin();
for (; it != vec.constEnd(); ++it) {
    qDebug() << "Element:" << *it;
}
```

Примечательно также, что эти итераторы можно использовать со стандартными алгоритмами STL, определенными в заголовочном файле algorithm. Например, для сортировки вектора посредством STL-алгоритма `sort()` можно поступить следующим образом:

```
QVector<QString> vec;
vec << "In Extremo" << "Blackmore's Night" << "Cultus Ferox";
std::sort(vec.begin(), vec.end());
qDebug() << vec;
```

На экране будет отображено:

```
QVector("Blackmore's Night", "Cultus Ferox", "In Extremo")
```

Qt предоставляет и свои собственные алгоритмы, которые мы рассмотрим далее в этой главе.

Ключевое слово *foreach*

Разумеется, в языке C++ нет такого ключевого слова, оно было создано искусственно, посредством препроцессора, и представляет собой разновидность цикла, предназначенного для перебора всех элементов контейнера. Этот способ является альтернативой константному итератору. Например:

```
QList<QString> list;
list << "Subway to sally" << "Rammstein" << "After Forever";
foreach(QString str, list) {
    qDebug() << "Element:" << str;
}
```

В `foreach`, как и в циклах, можно использовать ключевые слова `break`, `continue`, а также вкладывать циклы друг в друга.

> ***Изменение значений элементов в цикле foreach на оригинальном контейнере не отражается***
>
> Qt делает копию контейнера при входе в цикл `foreach`, поэтому если вы будете менять значение элементов в цикле, то на оригинальном контейнере это никак не отразится.

Последовательные контейнеры

Последовательные контейнеры представляют собой упорядоченные коллекции, где каждый элемент занимает определенную позицию. Операции, доступные для всех последовательных контейнеров, приведены в табл. 4.3.

Таблица 4.3. *Общие методы последовательных контейнеров*

Оператор/метод	Описание
`+`	Объединяет элементы двух контейнеров
`+=`	Добавляет элемент в контейнер (то же, что и <<)
`<<`	Добавляет элемент в контейнер
`at()`	Возвращает указанный элемент
`back()` и `last()`	Возвращают ссылку на последний элемент. Эти методы предполагают, что контейнер не пуст. Оба метода: `back()` и `last()` — идентичны, но применение первого предпочтительнее, так как он соответствует STL
`contains()`	Проверяет, содержится ли переданный в качестве параметра элемент в контейнере
`erase()`	Удаляет элемент, расположенный на позиции итератора, передаваемого в качестве параметра
`front()` и `first()`	Возвращают ссылку на первый элемент контейнера. Методы предполагают, что контейнер не пуст. Оба метода: `front()` и `first()` — идентичны, но применение первого более предпочтительно, так как он соответствует STL
`indexOf()`	Возвращает позицию первого совпадения найденного в контейнере элемента в соответствии с переданным в метод значением. Внимание: в контейнере `QLinkedList` этот метод отсутствует

Таблица 4.3 *(окончание)*

Оператор/метод	Описание
`lastIndexOf()`	Возвращает позицию последнего совпадения найденного в контейнере элемента в соответствии с переданным в метод значением. Внимание: в контейнере `QLinkedList` этот метод отсутствует
`mid()`	Возвращает контейнер, содержащий копии элементов, задаваемых начальной позицией и количеством
`pop_back()`	Удаляет последний элемент контейнера
`pop_front()`	Удаляет первый элемент контейнера
`push_back()` и `append()`	Методы добавляют один элемент в конец контейнера. Оба метода идентичны, но применение первого предпочтительно, так как он соответствует STL
`push_front()` и `prepend()`	Методы добавляют один элемент в начало контейнера. Оба метода идентичны, но применение первого предпочтительно, так как он соответствует STL
`replace()`	Заменяет элемент, находящийся на заданной позиции, значением, переданным как второй параметр

Пример:

```
QVector<QString> vec;
vec.append("In Extremo");
vec.append("Blackmore's Night");
vec.append("Cultus Ferox");
qDebug() << vec;
```

На экране вы увидите:

```
QVector("In Extremo", "Blackmore's Night", "Cultus Ferox")
```

Как мы видим из табл. 4.3, основными операциями являются получение доступа к элементу, вставка/удаление элемента, добавление элемента в конец и добавление элемента в начало.

Табл. 4.4 показывает, насколько быстро выполняются эти операции для каждого из контейнеров в отдельности. Пользуясь этой таблицей, важно в зависимости от поставленной задачи подобрать контейнер, который будет работать быстро.

Таблица 4.4. *Скорость выполнения операций для последовательных контейнеров*

Контейнер	Доступ	Вставка/Удаление	Добавление в конец	Добавление в начало
`QList` `QQueue`	Быстро	Медленно	Быстро	Быстро
`QVector` `QStack`	Быстро	Медленно	Быстро	Медленно
`QLinkedList`	Медленно	Быстро	Быстро	Быстро

Вектор QVector<T>

Вектор — это структура данных, очень похожая на обычный массив (рис. 4.3). Однако использование класса вектора предоставляет некоторые преимущества по сравнению с обычным массивом — например, можно получить количество содержащихся в нем элементов или динамически изменить его размер. Кроме того, этот контейнер, по сравнению с другими, наиболее экономично расходует память.



Рис. 4.3. Структура вектора

> ***ПРАВИЛЬНО ВЫБИРАЙТЕ ТИП КОНТЕЙНЕРА!***
>
> Постарайтесь не использовать методы `push_front()`, `prepend()`, `pop_front()` и `remove()`, так как вставка и удаление с извлечением элементов в начале вектора очень неэффективны (см. табл. 4.4) и могут снизить быстродействие вашего алгоритма и программы в целом. Это же касается и операции вставки, поэтому также желательно не прибегать к методу `insert()` (см. табл. 4.4). Если использованием указанных методов нельзя пренебречь, то задайте себе вопрос, правильно ли вы выбрали тип контейнера и не лучше ли было бы использовать какой-либо другой — например, `QLinkedList<T>`.

Для добавления элементов в конец последовательного контейнера (см. табл. 4.3) можно воспользоваться методом `push_back()`. К элементам вектора можно обратиться как посредством оператора индексации `[]` (см. табл. 4.1), так и при помощи итератора. Например:

```
QVector<int> vec;
vec.push_back(10);
vec.push_back(20);
vec.push_back(30);

qDebug() << vec;
```

На консоли будет отображено следующее:

```
QVector(10, 20, 30)
```

Размер вектора можно задать в конструкторе при его создании. По умолчанию только что созданный вектор будет иметь размер, равный 0, так как он не содержит ни одного элемента. Изменить его размер можно либо добавив к нему элементы, либо вызвав метод `resize()` (табл. 4.5).

Таблица 4.5. *Некоторые методы контейнера* `QVector<T>`

Метод	Описание
`data()`	Возвращает указатель на данные вектора (то есть на обычный массив)
`fill()`	Присваивает одно и то же значение всем элементам вектора
`reserve()`	Резервирует память для количества элементов в соответствии с переданным значением
`resize()`	Устанавливает размер вектора в соответствии с переданным значением
`toList()`	Возвращает объект `QList` с элементами, содержащимися в векторе
`toStdVector()`	Возвращает объект `std::vector` с элементами, содержащимися в векторе

Массив байтов *QByteArray*

Этот класс очень похож на QVector<T>, но разница заключается в том, что это не шаблонный класс, и в нем допускается хранение только элементов, имеющих размер в один байт. Объекты типа QByteArray можно использовать везде, где требуется промежуточное хранение данных. Количество элементов массива можно задать в конструкторе, а доступ к ним получать при помощи оператора []:

```
QByteArray arr(3);
arr[0] = arr[1] = 0xFF;
arr[2] = 0x0;
```

К данным объектов класса QByteArray можно также применять операцию сжатия и обратное преобразование. Это достигается при помощи двух глобальных функций: qCompress() и qUncompress(). Просто сожмем и разожмем данные:

```
QByteArray a           = "Test Data";
QByteArray aCompressed = qCompress(a);
qDebug() << qUncompress(aCompressed);
```

На экране появится: "Test Data".

Случается, что нужно преобразовывать бинарные данные в текстовые. Например, вам требуется записать растровое изображение в текст XML-файла (см. *главу 40*). Класс QByteArray предоставляет для этих целей два метода: toBase64() и fromBase64(). Из названий методов видно, что бинарные данные будут преобразовываться в формат Base64. Этот формат был специально разработан для передачи бинарных данных в текстовой форме. Приведем небольшой пример, а для того чтобы проводимые преобразования были понятны, мы применим их к обычной строке текста:

```
QByteArray a       = "Test Data";
QByteArray aBase64 = a.toBase64();
qDebug() << aBase64;
```

На экране мы увидим: "VGVzdCBEYXRh".

Теперь проведем обратное преобразование при помощи статического метода fromBase64():

```
qDebug() << QByteArray::fromBase64(aBase64);
```

На экране появится: "Test Data".

Чтобы бинарные данные занимали меньше места в текстовом файле, их можно перед кодированием в Base64 сжать.

Массив битов *QBitArray*

Этот класс управляет битовым (или булевым) массивом. Каждое из сохраняемых значений занимает только один бит, не расходуя лишней памяти. Значения упаковываются в байты с помощью класса QByteArray. Этот тип используется для хранения большого количества переменных типа bool.

Для операций с битами этот класс предоставляет методы: для чтения testBit() и для записи setBit(). Наряду с этими методами существует также оператор [], с помощью которого можно обращаться к каждому биту в отдельности:

```
QBitArray bits(3);
bits[0] = bits[1] = true;
bits[2] = false;
```

Списки QList<T> и QLinkedList<T>

Список — это структура данных, представляющая собой упорядоченный набор связанных друг с другом элементов. Преимущество списков перед векторами и очередями состоит в том, что вставка и удаление элементов в любой позиции происходит эффективнее, поскольку для выполнения этих операций изменяется только минимальное количество указателей, исключение составляет лишь вставка элемента в центр списка. Но есть и недостаток — списки плохо приспособлены для поиска определенного элемента по индексу, и для этой цели лучше использовать векторы.

> ***Чтобы узнать, пуст список или нет, используйте методы empty() или isEmpty()***
>
> Постарайтесь как можно реже опрашивать количество элементов списка вызовом метода `size()`, так как при каждом его вызове будет осуществляться их подсчет, а это может заметно сказаться на скорости программы. В тех же случаях, когда необходимо узнать, пуст список или нет, используйте без раздумий только методы `empty()` или `isEmpty()`.

> ***Вставка и удаление с извлечением элементов очень неэффективны!***
>
> Постарайтесь не использовать для `QList<T>` методы `removeAt()` и `insert()`, так как вставка и удаление с извлечением элементов очень неэффективны (см. табл. 4.4). Для этих операций лучше использовать описанный далее класс `QLinkedList`.

Списки реализует класс `QList<T>`. В общем виде этот класс представляет собой массив указателей на элементы (рис. 4.4).

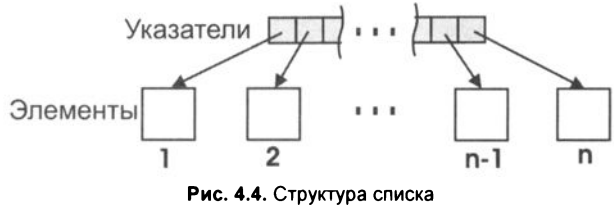


Рис. 4.4. Структура списка

Специфические операции для списков приведены в табл. 4.6.

Таблица 4.6. *Некоторые методы контейнера* `QList<T>`

Метод	Описание
`move()`	Перемещает элемент с одной позиции на другую
`removeFirst()`	Удаляет первый элемент списка
`removeLast()`	Удаляет последний элемент списка
`swap()`	Меняет местами два элемента на указанных позициях
`takeAt()`	Возвращает элемент на указанной позиции и удаляет его
`takeFirst()`	Удаляет первый элемент и возвращает его
`takeLast()`	Удаляет последний элемент и возвращает его
`toSet()`	Возвращает контейнер `QSet<T>` с данными, содержащимися в объекте `QList<T>`
`toStdList()`	Возвращает стандартный список STL `std::list<T>` с элементами, содержащимися в объекте `QList<T>`

Таблица 4.6 *(окончание)*

Метод	Описание
`toVector()`	Возвращает объект вектора `QVector<T>` с элементами, содержащимися в объекте `QList<T>`

Если вы не собираетесь изменять значения элементов, то, из соображений эффективности, не рекомендуется использовать оператор индексации `[]`. Вместо этого лучше воспользоваться методом `at()`, так как этот метод возвращает константную ссылку на элемент.

Одна из самых распространенных операций — обход списка для последовательного получения значений каждого элемента списка. Например:

```
QList<int> list;
list << 10 << 20 << 30;

QValueList<int>::iterator it = list.begin();
while (it != list.end()) {
    qDebug() << "Element:" << *it;
    ++it;
}
```

На консоли будет отображено следующее:

```
Element:10
Element:20
Element:30
```

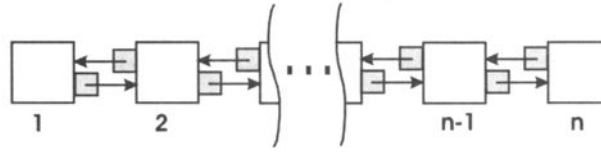


Рис. 4.5. Структура двусвязного списка

В классе `QList<T>` недостаточно эффективно работает вставка элементов, поэтому если вы работаете с большими списками и/или вам часто требуется вставлять элементы, то эффективнее использовать двусвязные списки `QLinkedList<T>`. Хотя этот контейнер расходует больше памяти, чем `QList<T>`, как это видно из его структуры (рис. 4.5), зато операции вставки и удаления сводятся к переопределению четырех указателей, независимо от позиции удаляемого или вставляемого элемента.

Примечание

В этом классе не определены операции для индексного доступа: `[]` и `at()`.

Стек *QStack<T>*

Стек `QStack<T>` реализует структуру данных, работающую по принципу LIFO (Last In First Out, последним пришел — первым ушел), т. е. из стека первым удаляется элемент, который был вставлен позже всех остальных (рис. 4.6).

Класс `QStack<T>` представляет собой реализацию стековой структуры. Этот класс унаследован от класса `QVector<T>`. Процесс помещения элементов в стек обычно называется *проталкиванием* (pushing), а извлечение из него верхнего объекта — *выталкиванием* (poping). Каждая операция проталкивания увеличивает размер стека на 1, а каждая операция выталкивания — уменьшает на 1. Для этих операций в классе `QStack<T>` определены методы `push()` и `pop()`. Метод `top()` возвращает ссылку на элемент вершины стека. Следующий пример демонстрирует использование класса стека.

```
QStack<QString> stk;
stk.push("Era");
stk.push("Corvus Corax");
stk.push("Gathering");

while (!stk.empty()) {
   qDebug() << "Element:" << stk.pop();
}
```

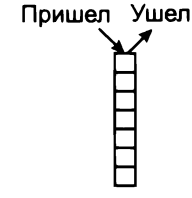


Рис. 4.6. Принцип работы стека

На консоли будет отображено следующее:

```
Element:"Gathering"
Element:"Corvus Corax"
Element:"Era"
```

Очередь *QQueue<T>*

Очередь реализует структуру данных, работающую по принципу FIFO (First In First Out, первым пришел — первым ушел), то есть из очереди удаляется не последний вставленный элемент, а тот, который был вставлен в очередь раньше всех остальных (рис. 4.7). Реализована очередь в классе `QQueue<T>`, который унаследован от класса `QList<T>`.

Следующий пример демонстрирует принцип использования очереди:

```
QQueue<QString> que;
que.enqueue("Era");
que.enqueue("Corvus Corax");
que.enqueue("Gathering");

while (!que.empty()) {
   qDebug() << "Element:" <<que.dequeue();
}
```

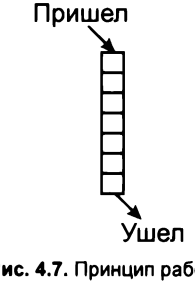


Рис. 4.7. Принцип работы очереди

На экране должно появиться:

```
Element:"Era"
Element:"Corvus Corax"
Element:"Gathering"
```

Ассоциативные контейнеры

Задача ассоциативных контейнеров заключается в сохранении соответствий ключа и значения. Это позволяет обращаться к элементам не по индексу, а при помощи ключа. Для всех контейнеров этого типа (за некоторыми исключениями для контейнера `QSet<T>`) доступны методы, приведенные в табл. 4.7.

Таблица 4.7. Общие методы ассоциативных контейнеров

Метод	Описание
`contains()`	Возвращает значение `true`, если контейнер содержит элемент с заданным ключом. Иначе возвращается значение `false`
`erase()`	Удаляет элемент из контейнера в соответствии с переданным итератором
`find()`	Осуществляет поиск элемента по значению. В случае успеха возвращает итератор, указывающий на этот элемент, а в случае неудачи итератор указывает на метод `end()`
`insertMulti()`	Вставляет в контейнер новый элемент. Если элемент уже присутствует в контейнере, то создается новый элемент. Этот метод отсутствует в классе `QSet<T>`
`insert()`	Вставляет в контейнер новый элемент. Если элемент уже присутствует в контейнере, он замещается новым элементом. Этот метод отсутствует в классе `QSet<T>`
`key()`	Возвращает первый ключ в соответствии с переданным в этот метод значением. Этот метод отсутствует в классе `QSet<T>`
`keys()`	Возвращает список всех ключей, находящихся в контейнере. Этот метод отсутствует в классе `QSet<T>`
`take()`	Удаляет элемент из контейнера в соответствии с переданным ключом и возвращает копию его значения. Этот метод отсутствует в классе `QSet<T>`
`unite()`	Добавляет элементы одного контейнера в другой
`values()`	Возвращает список всех значений, находящихся в контейнере

Словари *QMap<K,T>* и *QMultiMap<K,T>*

«Программные» словари, в принципе, похожи на словари обычные, используемые нами в повседневной жизни. Они хранят элементы одного и того же типа, индексируемые ключевыми значениями. Главное достоинство словаря в том, что он позволяет быстро получать значение, ассоциированное с заданным ключом. Ключи должны быть уникальными (рис. 4.8), за исключением мультисловаря, который допускает дубликаты (рис. 4.9).

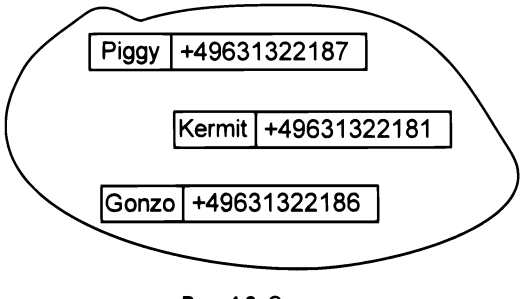


Рис. 4.8. Словарь

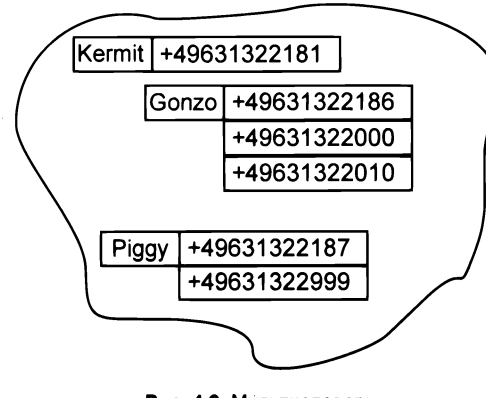


Рис. 4.9. Мультисловарь

В контейнеры этого типа заносятся элементы вместе с ключами, по которым их можно найти и которыми могут выступать значения любого типа. В случае со словарем `QMap<K,T>` необходимо следить за тем, чтобы не было занесено двух разных элементов с одинаковым ключом, — ведь тогда один из этих элементов невозможно будет отыскать. То есть, каждый ключ словаря `QMap<K,T>` должен быть уникален.

При создании объекта класса `QMap<K,T>` нужно передать его размер в конструктор. Этот размер не является, как это принято в других контейнерных классах, размером, ограничивающим максимальное количество элементов, а представляет собой количество позиций. Количество позиций должно быть больше количества элементов, ожидаемых для хранения, иначе поиск элементов в словаре будет проводиться недостаточно эффективно. Желательно, чтобы это значение относилось к разряду простых чисел (см. *приложение 2*), так как в этом случае размещение элементов будет более удобным. Табл. 4.8 содержит некоторые из методов класса `QMap<K,T>`.

Таблица 4.8. *Некоторые методы контейнера* `QMap<K,T>`

Метод	Описание
`lowerBound()`	Возвращает итератор, указывающий на первый элемент с заданным ключом
`toStdMap()`	Возвращает стандартный словарь STL с элементами, находящимися в объекте `QMap<T>`
`upperBound()`	Возвращает итератор, указывающий на последний элемент с заданным ключом

Одним из самых частых способов обращения к элементам словаря является использование ключа в операторе `[]`. Но можно обойтись и без него, так как ключ и значение можно получить с помощью методов итератора `key()` и `value()`, например:

```
QMap<QString, QString> mapPhonebook;
mapPhonebook["Piggy"]  = "+49 631322187";
mapPhonebook["Kermit"] = "+49 631322181";
mapPhonebook["Gonzo"]  = "+49 631322186";

QMap<QString, QString>::iterator it = mapPhonebook.begin();
for (;it != mapPhonebook.end(); ++it) {
    qDebug() << "Name:" << it.key()
             << " Phone:" << it.value();
}
```

На консоли будет отображено следующее:

```
Name:Gonzo Phone:+49 631322186
Name:Kermit Phone:+49 631322181
Name:Piggy Phone:+49 631322187
```

Особое внимание следует обратить на использование оператора `[]`, который может применяться как для вставки, так и для получения значения элемента. Но надо быть осторожным, поскольку задание ключа, для которого элемент не существует, приведет к тому, что элемент будет создан. Чтобы избежать этого, нужно проверять существование элемента, привязанного к ключу. Подобную проверку можно осуществить при помощи метода `contains()`. Например:

```
if(mapPhonebook.contains("Kermit")) {
    qDebug() << "Phone:" << mapPhonebook["Kermit"];
}
```

На практике случается так, что нужно внести сразу несколько телефонных номеров для одного и того же человека, — например, его домашний, рабочий и мобильный телефоны. Для этого обычный словарь QMap<K,T> уже не подходит, и нам будет необходимо воспользоваться мультисловарем QMultiMap<K,T>. Пример такого словаря был показан на рис. 4.9, давайте снабдим его программным кодом и узнаем телефонные номера для Piggy:

```
QMultiMap<QString, QString> mapPhonebook;
mapPhonebook.insert("Kermit", "+49 631322181");
mapPhonebook.insert("Gonzo", "+49 631322186");
mapPhonebook.insert("Gonzo", "+49 631322000");
mapPhonebook.insert("Gonzo", "+49 631322010");
mapPhonebook.insert("Piggy", "+49 631322187");
mapPhonebook.insert("Piggy", "+49 631322999");

QMultiMap<QString, QString>::iterator it =
                                mapPhonebook.find("Piggy");
for (; it != mapPhonebook.end() && it.key() == "Piggy"; ++it) {
    qDebug() << it.value() ;
}
```

Хэши QHash<K,T> и QMultiHash<K,T>

Функциональность хэшей очень похожа на функциональность словаря QMap<K,T> с той лишь разницей, что вместо сортировки по ключу этот класс использует хэш-таблицу. Такой подход позволяет ему осуществлять поиск ключевых значений гораздо быстрее, чем это делает словарь QMap<K,T>.

Так же, как и в случае со словарем QMap<K,T>, следует соблюдать осторожность при использовании оператора индексации [], поскольку задание ключа, для которого элемент не существует, приведет к тому, что элемент будет создан. Поэтому важно проверять существование элемента, привязанного к ключу, при помощи метода contains() контейнера.

Если вы намереваетесь разместить в хэш QHash<K,T> объекты собственных классов, то вам необходимо будет реализовать оператор сравнения == и специализированную функцию qHash() для вашего класса. Вот пример реализации оператора сравнения:

```
inline bool operator==(const MyClass& mc1, const MyClass& mc2)
{
    return (mc1.firstName() == mc2.firstName()
            && mc1.secondName() == mc2.secondName()
           );
}
```

Функция qHash()возвращает число, которое должно быть уникальным для каждого находящегося в хэше элемента. Например:

```
inline uint qHash(const MyClass& mc)
{
    return qHash(mc.firstName()) ^ qHash(mc.secondName());
}
```

Класс QMultiHash<K,T> унаследован от QHash<K,T>. Он позволяет размещать значения с одинаковыми ключами и, в целом, похож на класс QMultiMap<K,T>, но учитывает специфику

своего родительского класса. Методы, присущие только для этих контейнеров, приведены в табл. 4.9.

Таблица 4.9. *Некоторые методы контейнеров* `QHash<K,T>` *и* `QMultiHash<K,T>`

Метод	Описание
`capacity()`	Возвращает размер хэш-таблицы
`reserve()`	Задает размер хэш-таблицы
`squeeze()`	Уменьшает объем внутренней хэш-таблицы для уменьшения используемого объема памяти

Множество *QSet<T>*

Как заметил немецкий математик Георг Кантор: «Множество — это есть многое, мысленно подразумеваемое нами как единое». Это «единое», в контексте Tulip, есть не что иное, как контейнер `QSet<T>`, который записывает элементы в некотором порядке и предоставляет возможность очень быстрого просмотра значений и выполнения с ними операций, характерных для множеств, — таких как: объединение, пересечение и разность. Необходимым условием является уникальность ключей.

Класс `QSet<T>` базируется на использовании хэш-таблицы `QHash<K,T>`, но является вырожденным ее вариантом, так как с ключами не связываются никакие значения. Главная задача этого класса заключается в хранении ключей. Контейнер `QSet<T>` можно использовать в качестве неупорядоченного списка для быстрого поиска данных. Пример множеств показан на рис. 4.10, где изображены два множества, состоящие из трех элементов каждое.

Операции, которые можно проводить с множествами, проиллюстрированы на рис. 4.11.

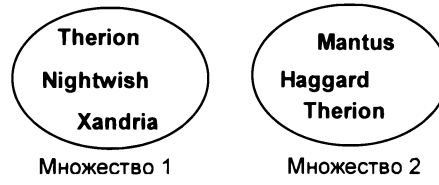


Рис. 4.10. Два множества

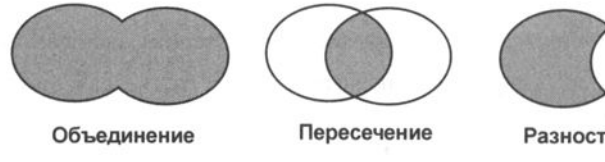


Рис. 4.11. Некоторые операции над множествами

Создадим два множества и запишем в них элементы в соответствии с рис. 4.10.

```
QSet<QString> set1;
QSet<QString> set2;
set1 << "Therion" << "Nightwish" << "Xandria";
set2 << "Mantus" << "Haggard" << "Therion";
```

Произведем операцию объединения (см. рис. 4.11, *слева*) этих двух множеств, а для того, чтобы элементы множеств остались неизмененными, введем промежуточное множество setResult:

```
QSet<QString> setResult = set1;
setResult.unite(set2);
qDebug() << "Объединение = " << setResult.toList();
```

На экране должно быть показано следующее:

```
Объединение = ("Xandria", "Haggard", "Mantus", "Nightwish", "Therion")
```

Теперь произведем операцию пересечения (см. рис. 4.11, *в центре*):

```
setResult = set1;
setResult.intersect(set2);
qDebug() << "Пересечение set1 с set2 = " << setResult.toList();
```

Поскольку два множества имеют только один одинаковый элемент, то на экране мы увидим:

```
Пересечение set1 с set2 =  ("Therion")
```

И последняя операция, которую мы произведем, будет операция разности двух множеств (см. рис. 4.11, *справа*):

```
setResult = set1;
setResult.subtract(set2);
qDebug() << "Разность set1 с set2 = " << setResult.toList();
```

Множество set1 отличается от множества set2 двумя элементами, поэтому на экране должно отобразиться:

```
Разность set1 с set2 =  ("Xandria", "Nightwish")
```

В табл. 4.10 сведены методы для контейнера QSet<T>.

Таблица 4.10. *Некоторые методы контейнера* `QSet<T>`

Метод	Описание
`intersect()`	Удаляет элементы множества, не присутствующие в переданном множестве
`reserve()`	Задает размер хэш-таблицы
`squeeze()`	Уменьшает объем внутренней хэш-таблицы для уменьшения используемого объема памяти
`subtract()`	Удаляет все элементы множества, присутствующие в переданном множестве
`toList()`	Возвращает объект контейнера QList<T>, содержащий элементы из объекта контейнера QSet<T>
`unite()`	Объединяет элементы множеств

Алгоритмы

Алгоритмы определены в заголовочном файле QtAlgorithms и предоставляют операции, применяемые к контейнерам, — например: сортировку, поиск, преобразование данных и т. д. Следует отметить, что алгоритмы реализованы не в виде методов контейнерных

классов, а в виде шаблонных функций, что позволяет использовать их как для любого контейнерного класса Tulip, так и для обычных массивов. Например, для копирования элементов из одного массива в другой можно задействовать алгоритм `qCopy()`:

```
QString values[] = {"Xandria", "Therion", "Nightwish", "Haggard"};
const int n = sizeof(values) / sizeof(QString);
QString copyOfValues[n];
qCopy(values, values + n, copyOfValues);
```

При копировании контейнеров важно убедиться, что целевой контейнер имеет размер, достаточный для размещения копии. В нашем примере мы позаботились о том, чтобы целевой контейнер имел такой же размер, как контейнер-источник.

В табл. 4.11 приведены все алгоритмы, предоставляемые Tulip. Qt предоставляет только самые основные алгоритмы, но если их вдруг окажется недостаточно, всегда можно воспользоваться алгоритмами STL.

Таблица 4.11. *Алгоритмы*

Алгоритм	Описание
`qBinaryFind()`	Двоичный поиск заданных значений
`qCopy()`	Копирование элементов, начиная с первого
`qCopyBackward()`	Копирование элементов, начиная с последнего
`qCount()`	Подсчет элементов контейнера
`qDeleteAll()`	Удаление всех элементов. Необходимо, чтобы элементы контейнера не были константными указателями
`qEqual()`	Сравнение. Необходимо, чтобы для размещенных объектов был определен оператор ==
`qFill()`	Присваивает всем элементам контейнера заданное значение
`qFind()`	Поиск заданных значений
`qLowerBound()`	Нахождение первого элемента со значением, большим либо равным заданному
`qUpperBound()`	Нахождение первого элемента со значением, строго большим заданного
`qSort()`	Сортировка элементов
`qStableSort()`	Сортировка элементов с сохранением порядка следования равных элементов
`qSwap()`	Перемена двух значений местами

Сортировка

Сортировку осуществляет функция-алгоритм `qSort()` (см. табл. 4.10). Для сортировки необходимо, чтобы к типам элементов контейнера можно было применить операторы сравнения, так как они нужны для принятия решения самим алгоритмом. Например, для `QString` эти операторы доступны. Произведем сортировку для списка с элементами `QString`:

```
QList<QString> list;
list << "Within Temptation" << "Anubis" << "Mantus";
```

```
qSort(list);
qDebug() << "Sorted list=" << list;
```

На экране мы увидим следующее:

```
Sorted list=("Anubis", "Mantus", "Within Temptation")
```

Можно так же задать границы и условие для проведения сортировки, например:

```
qSort(list.begin(), list.end(), caseLessThan);
```

В этом случае мы производим сортировку в алфавитном порядке с учетом заглавных и строчных букв в границах от начала до конца списка. Для отключения учета заглавных и строчных букв нужно воспользоваться третьим параметром: `caseInsensitiveLessThan`.

Для сортировки чисел в порядке убывания в качестве третьего параметра нужно использовать `QGreater<T>`, например:

```
QList<int> list;
list << 1 << 2 << 3 << 4 << 5 << 6;
qSort(list.begin(), list.end(), qGreater<int>());
qDebug() << "Sorted list=" << list;
```

На экране мы увидим:

```
Sorted list=(6, 5, 4, 3, 2, 1)
```

Можно на третьем месте алгоритма `qSort()` для задания условия сортировки использовать и собственную функцию:

```
bool lessThan(const QString& str1, const QString& str2)
{
    return QString::compare(str1, str2, Qt::CaseInsensitive) < 0;
}

QList<QString> list;
list << "Within Temptation" << "Anubis" << "anubis" << "Mantus";
qSort(list.begin(), list.end(), lessThan);
qDebug() << list;
```

На экране появится:

```
("anubis", "Anubis", "Mantus", "Within Temptation")
```

Поиск

За поиск элементов отвечает функция-алгоритм `qFind()` (см. табл. 4.10). Эта функция возвращает итератор, установленный на первый найденный элемент или на `end()`, если элемент не найден.

```
QList<QString> list;
list << "Within Temptation" << "Anubis" << "Mantus";
QList<QString>::iterator it =
    qFind(list.begin(), list.end(), "Anubis");
if (it != list.end()) {
    qDebug() << "Found=" << *it;
}
```

```
else {
    qDebug() << "Not Found";
}
```

На экране появится:

```
Found=Anubis
```

Сравнение

Иногда возникает необходимость в сравнении содержимого контейнеров различных типов. Это можно осуществить при помощи функции-алгоритма `qEqual()` (см. табл. 4.10). Как и в случае сортировки, для элементов контейнера должны быть применимы операторы сравнения:

```
QList<QString> list;
list << "Within Temptation" << "Anubis" << "Mantus";

QVector<QString> vec(3);
vec[0] = "Within Temptation";
vec[1] = "Anubis";
vec[2] = "Mantus";

qDebug() << "Equal="
         << qEqual(list.begin(), list.end(), vec.begin());
```

На экране вы увидите:

```
Equal=true
```

Если вы измените в одном из контейнеров какую-нибудь из строк, например `Mantus` на `Mantux`, то функция `qEqual()` возвратит значение `false`.

Заполнение значениями

В некоторых случаях может понадобиться присвоить значения элементам какой-либо части контейнера. Для этих целей существует алгоритм `qFill()` (см. табл. 4.10). Присвоим всем элементам списка значение `Beatles`:

```
QList<QString> list;
list << "Within Temptation" << "Anubis" << "Mantus";
qFill(list.begin(), list.end(), "Beatles");
qDebug() << list;
```

На экране должно появиться:

```
("Beatles", "Beatles", "Beatles")
```

Копирование значений элементов

Для того чтобы скопировать значения элементов из одного контейнера в другой, можно использовать алгоритм `qCopy()`. Вот так можно, например, скопировать все значения элементов из контейнера списка в контейнер вектора:

```
QList<QString> list;
list << "Within Temptation" << "Anubis" << "Mantus";
```

```
QVector<QString> vec(3);
qCopy(list.begin(), list.end(), vec.begin());
qDebug() << vec;
```

На экране появится:

```
QVector("Within Temptation", "Anubis", "Mantus")
```

Подсчет значений

Для того чтобы подсчитать количество элементов контейнера с определенным значением, можно воспользоваться алгоритмом `qCount()` (см. табл. 4.10). Подсчитаем, например, количество строк `"Mantus"` в списке:

```
QList<QString> list;
list << "Within Temptation" << "Mantus" << "Anubis" << "Mantus";
int n = 0;
qCount(list, "mantus", n);
qDebug() << n;
```

На экране появится цифра 2.

Строки

Практически все приложения оперируют текстовыми данными. В Qt реализован класс `QString`, объекты которого могут хранить строки символов в формате Unicode, где каждый символ занимает два байта. Принцип хранения данных аналогичен классу `QVector`, единственное различие состоит в том, что элементы всегда относятся к символьному типу `QChar`, то есть можно сказать: *строка* — это контейнер для хранения символов. Класс `QString` предоставляет целую серию методов и операторов, позволяющих проводить со строками разного рода операции, — например: соединять строки, осуществлять поиск подстрок, преобразовывать их в верхний или нижний регистр и многое другое.

Строки можно сравнивать друг с другом при помощи операторов сравнения `==`, `!=`, `<`, `>`, `<=` и `>=`. Результат сравнения зависит от регистра символов, например:

```
QString str = "Lo";
bool b1  = (str == "Lo"); // b1 = true
bool b2  = (str != "LO"); // b2 = true
```

При помощи метода `isEmpty()` можно узнать, не пуста ли строка. Того же результата можно добиться, проверив длину строки методом `length()`. В классе `QString` имеются различия между пустыми и нулевыми строками — так, строка, созданная при помощи конструктора по умолчанию, представляет собой нулевую строку. Например:

```
QString str1 = "";
QString str2;
str1.isNull(); // false
str2.isNull(); // true
```

Объединение строк является одной из самых распространенных операций. Провести его можно разными способами: скажем, при помощи операторов `+=` и `+` или вызовом метода `append()`. Например:

```
QString str1 = "Lo";
QString str2 = "stris";
QString str3 = str1 + str2; // str3 = "Lostris"
str1.append(str2); //str1 = "Lostris"
```

Для замены определенной части строки другой класс `QString` предоставляет метод `replace()`. Например:

```
QString str = "Lostris";
str.replace("stris", "gic"); // str = "Logic"
```

Для преобразования данных строки в верхний или нижний регистр используются методы `toLower()` или `toUpper()`. Например:

```
QString str1 = "LoStRiS";
QString str2 = str1.toLower(); // str2 = "lostris"
QString str3 = str1.toUpper(); // str3 = "LOSTRIS"
```

При помощи метода `setNum()` можно конвертировать числовые значения в строковые. Того же результата можно добиться вызовом статического метода `number()`. Например:

```
QString str = QString::number(35.123);
```

Аналогичного результата можно добиться также и при помощи текстового потока Qt. Например:

```
QString str;
QTextStream(&str) << 35.123;
```

Преобразование из строкового в числовое значение производится методами, содержащими в своем имени название типа. В этих методах вторым параметром можно передавать ссылку на переменную булевого типа для получения информации о том, успешно ли была проведена операция. Например:

```
bool ok;
QString str = "234";
double  d   = str.toDouble(&ok);
int     n   = str.toInt(&ok);
```

Строка может быть разбита на массив строк при помощи метода `split()` класса `QStringList`. Следующий пример создаст список из двух строк: `Ringo` и `Star`:

```
QString str = "Ringo Star";
QStringList list = str.split(" ");
```

Операция объединения списка строк в одну строку производится при помощи метода `join()`. Например, объединить список из двух элементов (`Ringo` и `Star`) в одну строку, разделив их знаком пробела, можно следующим образом:

```
str = list.join(" "); //"Ringo Star"
```

В табл. 4.12 приведены некоторые методы класса `QString`, которые могут так же оказаться очень полезными.

Новый класс для работы со строками

Начиная с версии 5.10, в Qt появился класс `QStringView`, специально ориентированный на UTF-16. Этот класс имеет методы, схожие с классом `QString`. Если в программе вы часто используете методы класса `QString` для конвертирования данных в/из UTF-16, то целесообразнее будет использовать для строк новый класс `QStringView`.

Таблица 4.12. *Некоторые методы класса* `QString`

Символ	Описание
`endsWith()`	Принимает в качестве параметра строку и возвращает значение булевого типа `true`, если строка заканчивается этой строкой. В противном случае возвращается значение `false`
`startsWith()`	Принимает в качестве параметра строку и возвращает значение булевого типа `true`, если строка начинается с этой строки. В противном случае возвращается значение `false`
`contains()`	Принимает в качестве аргумента строку или регулярное выражение и возвращает значение булевого типа `true` в случае нахождения совпадения внутри строки. В противном случае возвращается значение `false`
`indexOf()`	Производит поиск строки или регулярного выражения с начала и в случае нахождения возвращает позицию. Если регулярное выражение или строка не найдены, то возвращается значение `-1`
`lastIndexOf()`	Производит поиск строки или регулярного выражения с конца и в случае нахождения возвращает позицию. Если регулярное выражение или строка не найдены, то возвращается значение `-1`
`left()`	Возвращает часть строки с указанным количеством символов слева
`right()`	Возвращает часть строки с указанным количеством символов справа
`mid()`	Возвращает часть строки с указанным количеством символов, начиная с заданной позиции
`simplified()`	Удаляет в строке повторяющиеся знаки пробелов
`leftJustified()`	Дополняет строку слева заданным символом. Принимает два аргумента: количество символов заполнения и сам символ
`rightJustified()`	Дополняет строку справа заданным символом. Принимает два аргумента: количество символов заполнения и сам символ

Регулярные выражения

Для работы с регулярными выражениями Qt предоставляет класс `QRegExp`. Регулярные выражения — это мощное средство анализа и обработки строк. Они содержат в себе шаблон, предназначенный для поиска в строке. Это позволяет быстро и гибко извлекать совпавший с шаблоном текст. Но следует заметить, что работа с регулярными выражениями производится медленнее методов, определенных в классе `QString`, и поэтому их применение должно быть обоснованным. Табл. 4.13 содержит основные шаблонные символы, поддерживаемые классом `QRegExp`.

Таблица 4.13. *Шаблоны регулярных выражений*

Символ	Описание	Пример
`.`	Любой символ	`a.b`
`$`	Должен быть конец строки	`Abc$`
`[]`	Любой символ из заданного набора	`[abc]`
`-`	Определяет диапазон символов в группе `[]`	`[0-9A-Za-z]`

Таблица 4.13 (окончание)

Символ	Описание	Пример
^	В начале набора символов означает любой символ, не вошедший в набор	[^def]
*	Символ должен встретиться в строке ни разу или несколько раз	A*b
+	Символ должен встретиться в строке минимум 1 раз	A+b
?	Символ должен встретиться в строке 1 раз или не встретиться вообще	A?b
{n}	Символ должен встретиться в строке указанное число раз	A{3}b
{n,}	Допускается минимум n совпадений	a{3,}b
{,n}	Допускается до n совпадений	a{,3}b
{n,m}	Допускается от n до m совпадений	a{2,3}b
|	Ищет один из двух символов	ac|bc
\b	В этом месте присутствует граница слова	a\b
\B	Границы слова нет в этом месте	a\Bd
( )	Ищет и сохраняет в памяти группу найденных символов	(ab|ac)ad
\d	Любое число	
\D	Все, кроме числа	
\s	Любой тип пробелов	
\S	Все, кроме пробелов	
\w	Любая буква, цифра или знак подчеркивания	
\W	Все, кроме букв	
\A	Начало строки	
\b	Целое слово	
\B	Не слово	
\Z	Конец строки (совпадает с символом конца строки или перед символом перевода каретки)	
\z	Конец строки (совпадает только с концом строки)	

Для того чтобы найти один из нескольких символов, нужно поместить их в квадратные скобки. Например, [ab] будет совпадать с a или b. Чтобы не писать все символы подряд, можно указать диапазон — например, [A-Z] совпадает с любой буквой в верхнем регистре, [a-z] — с любой буквой в нижнем регистре, а [0-9] — с любой цифрой. Можно совмещать такие записи — например, [a-z7] будет совпадать с любой буквой в нижнем регистре и с цифрой 7.

Также можно исключать символы, поставив перед ними знак ^. Например, [^0-9] будет соответствовать всем символам, кроме цифр.

Указанные в табл. 4.11 величины в фигурных скобках называются *пределами*. Пределы позволяют точно задать количество раз, которое символ должен повторяться в тексте. Напри-

мер, a{4,5} будет совпадать с текстом, если буква a встретится в нем не менее 4-х, но не более 5-ти раз подряд. Так, в следующем отрывке задано регулярное выражение для IP-адреса — им можно воспользоваться, например, для того, чтобы проверить строку на содержание в ней IP-адреса:

```
QRegExp reg("[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}");
QString str("this is an ip-address 123.222.63.1 lets check it");
qDebug() << (str.contains(reg) > 0); // true
```

Обратите внимание, что для указания символа точки в регулярном выражении перед ним стоит обратная косая черта (\), а в соответствии с правилами языка C++ для ее задания в строке она должна удваиваться. Если бы косой черты не было, то точка имела бы в соответствии с табл. 4.11 значение «любой символ», и регулярное выражение распознавало бы, например, строку 1z2y3x4, как IP-адрес, что, разумеется, неправильно.

Шаблоны можно комбинировать при помощи символа |, задавая ветвления в регулярном выражении. Регулярное выражение с двумя ветвями совпадает с подстрокой, если совпадает одна из ветвей. Например:

```
QRegExp rxp("(.com|.ru)");
int n1 = rxp.indexIn("www.bhv.ru"); // n1 = 7 (совпадение на 7-й позиции)
int n2 = rxp.indexIn("www.bhv.de");  // n2 =  -1 (совпадений не найдено)
```

Указанные в табл. 4.11 символы с обратной косой чертой (обратным слэшем) позволяют значительно упростить регулярные выражения. Например, регулярное выражение [a-zA-Z0-9_] идентично выражению \w.

Для определения правильности ввода адреса электронной почты (E-mail) можно использовать следующее регулярное выражение, указанное в объекте regEmail:

```
QRegExp regEmail("([a-zA-Z0-9_\\-\\.]+)@([a-zA-Z0-9_.-])+\\.([a-zA-Z]{2,4}|[0-
9]{1,3})");
QString strEmail1 = "Max.Schlee@neonway.com";
QString strEmail2 = "Max.Schlee#neonway.com";
QString strEmail3 = "Max.Schlee@neonway";
bool b1 = regEmail.exactMatch(strEmail1); //b1 = true
bool b2 = regEmail.exactMatch(strEmail2); //b2 = false
bool b3 = regEmail.exactMatch(strEmail3); //b3 = false
```

Вот пример того, как можно определить, является ли строка положительным числом от 0 до 999. Если строка им не является, то метод indexIn() вернет значение -1, так как строка не содержит позиции, которая удовлетворяла бы регулярному выражению, в противном случае этот метод вернет значение 0.

```
QRegExp reg("^\\d+\\d?\\d?$");
qDebug() << reg.indexIn("567");  // 0
qDebug() << reg.indexIn("3GB");  // -1
qDebug() << reg.indexIn("111B"); // -1
qDebug() << reg.indexIn("010");  // 0
qDebug() << reg.indexIn("10");   // 0
qDebug() << reg.indexIn("2");    // 0
qDebug() << reg.indexIn("-2");   // -1
```

Регулярные выражения можно использовать и для списков строк. Так, при помощи метода filter() класса QStringList можно отфильтровать список строк для того, чтобы получить

новый список. Например, возвращаясь к предыдущему примеру, давайте отфильтруем список строк и получим новый, который содержит только строки с положительным числом от 0 до 999.

```
QStringList lst;
lst << "576" << "3GB" << "111B" << "010" << "10" << "2" << "-2";
QStringList lstNumbers = lst.filter(QRegExp("^\\d+\\d?\\d?$"));
qDebug() << lstNumbers;
```

На экране увидим: `("576", "010", "10", "2")`

Произвольный тип *QVariant*

Объекты класса `QVariant` могут содержать данные разного типа, включая контейнеры. К этим типам относятся: `int`, `unsigned int`, `double`, `bool`, `QString`, `QStringList`, `QImage`, `QPixmap`, `QBrush`, `QColor`, `QRegExp` и др. Важно учитывать то обстоятельство, что частое применение этого типа может отрицательно отразиться на скорости программы и эффективности использования памяти, а также может заметно снизить читабельность самой программы. Поэтому объекты класса `QVariant` не следует использовать без особой на то необходимости.

Для создания объектов класса `QVariant` необходимо передать в конструктор переменную нужного типа. Например:

```
QVariant v1(34);
QVariant v2(true);
QVariant v2("Lostris");
```

Метод `type()` позволяет узнать тип записанных в объекте `QVariant` данных. Этот метод возвращает целочисленный идентификатор типа. Чтобы преобразовать его в строку, следует передать его в статический метод `typeToName()`. Того же результата можно добиться и вызовом метода `typeName()`, который возвращает информацию о типе в виде строки:

```
QVariant v(5.0);
qDebug() << QVariant::typeToName(v.type()); // =>double
```

Чтобы получить из объекта `QVariant` данные нужного типа, существует серия специальных методов `toT()`, где `T` — это имя типа. Метод `toT()` создает новый объект типа `T` и копирует данные из объекта `QVariant` в нужный объект. Например:

```
QVariant v2(23);
int a = v2.toInt() + 5; // a = 28
```

> ***Ограничение объекта QVariant***
>
> Ввиду того, что `QVariant` реализован в `QtCore`, соответствующих методов `toT()` для классов `QColor`, `QImage` и `QPixmap` и др., находящихся в модуле `QtGui`, не предоставляется.

Вместо методов `toT()` для приведения к нужному типу можно использовать также шаблонный метод `value<T>()`. Наш пример с преобразованием объекта `QVariant` к целому типу можно представить тогда следующим образом:

```
QVariant v2(23);
int a = v2.value<int>() + 5; //a = 28
```

или например для объекта `QPixmap`

```
QPixmap pix(":/myimg.png"); // создаем  объект QPixmap
QVariant vPix = pix; // сконвертируем его в QVariant, неявным вызовом
QPixmap::operator QVariant()
QPixmap pix2 = vPix.value<QPixmap>(); // получим объект QPixmap из QVariant обратно
```

Модель общего использования данных

Из соображений эффективности во многих классах Qt стараются избежать копирования данных — вместо этого используется ссылка на нужные данные (рис. 4.12). Этот принцип получил название *общее использование данных* (shared data). В Qt применяется модель неявных общих данных. В такой модели вызов конструктора копирования или оператора присваивания не приведет к копированию данных, а только увеличит счетчик ссылок на эти данные на 1. Соответственно, при удалении элемента счетчик ссылок уменьшится на 1. Если значение счетчика ссылок становится равным 0, то данные уничтожаются. Копирование данных происходит только при изменениях — соответственно, значение счетчика ссылок при этом уменьшается.

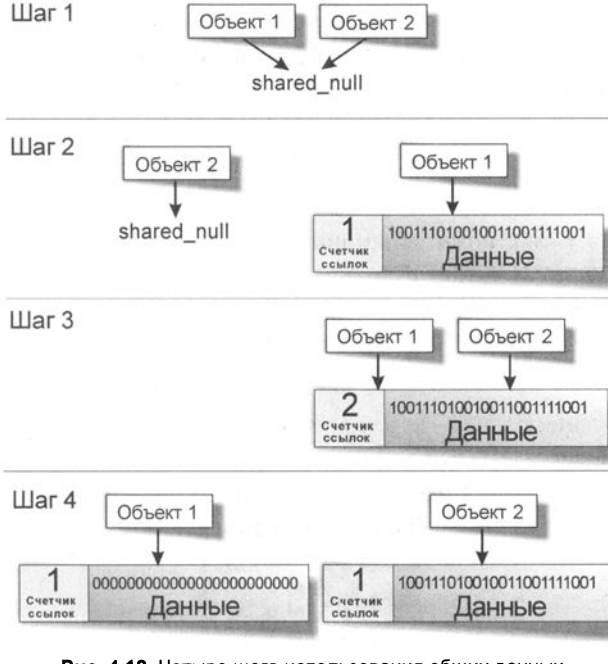


Рис. 4.12. Четыре шага использования общих данных

На рис. 4.12 на первом шаге создаются два объекта, и так как данные им не были присвоены, то они оба указывают на `shared_null` (общий ноль). На втором шаге первому объекту присваиваются данные, и счетчик ссылок становится равным единице. На третьем шаге второму объекту присваивается первый объект, и они теперь оба указывают на одни и те же данные, счетчик ссылок при этом увеличивается на единицу. На четвертом шаге производится изменение данных первого объекта, что приводит к созданию для него отдельной копии, а счетчик ссылок старых данных уменьшается на один, поскольку на один объект, ис-

пользующий эти данные, стало меньше. Если бы мы пятым шагом изменили данные второго объекта, то после создания копии для новых данных счетчик ссылок старых данных уменьшился бы до значения 0, и это привело бы к освобождению памяти и уничтожению старых данных.

Проиллюстрируем изображенную на рис. 4.12 ситуацию программным кодом:

```
QString str1;           // Ссылается на shared_null
QString str2;           // Ссылается на shared_null
str1 = "Новая строка"   // Ссылается на данные, счетчик ссылок = 1
str2 = str1;            // str1 и str2 указывают на одни и те же данные
                        // счетчик ссылок = 2
str1 += " добавление"; // Производится копирование данных для str1
```

Резюме

В этой главе мы узнали, что контейнер — это объект, предназначенный для хранения и управления содержащимися в нем элементами. Он заботится о выделении и освобождении памяти, а также отвечает за добавление и удаление элементов. Контейнерные классы подразделяются на последовательные и ассоциативные. К последовательным контейнерам относятся вектор, список, стек и очередь, к ассоциативным — множество, словарь и хэш.

Для прохождения по элементам контейнера используются итераторы. Qt предоставляет итераторы в стилях Java и STL. В качестве альтернативы для прохождения по элементам контейнера можно также воспользоваться макросом `foreach`.

При помощи алгоритмов можно проводить такие операции над содержимым контейнеров, как сортировка, поиск и многое другое.

Класс `QString` представляет собой реализацию строк и содержит целый ряд методов для проведения различного рода операций с ними.

Регулярные выражения представляют собой мощный механизм для проверки строк на соответствие шаблону.

Объекты класса `QVariant` могут содержать данные разного типа, включая также и контейнеры.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/04-510/** или с помощью следующего QR-кода (рис. 4.13):



Рис. 4.13. QR-код для доступа на страницу комментариев к этой главе



ЧАСТЬ II

Элементы управления

Никто не может вернуться в прошлое и изменить свой старт. Но каждый может стартовать сейчас и изменить свой финиш.

Рой Джонс

Глава 5. С чего начинаются элементы управления?

Глава 6. Управление автоматическим размещением элементов

Глава 7. Элементы отображения

Глава 8. Кнопки, флажки и переключатели

Глава 9. Элементы настройки

Глава 10. Элементы ввода

Глава 11. Элементы выбора

Глава 12. Интервью, или модель-представление

Глава 13. Цветовая палитра элементов управления



ГЛАВА 5

С чего начинаются элементы управления?

Кто скажет, где кончается одно и начинается другое?
Сунь Цзы

Практически любая программа имеет графический интерфейс пользователя (GUI, Graphical User Interface). Виджеты (widgets) — это «строительный материал» для его создания. *Виджет* — это не просто область, отображаемая на экране, это компонент, способный выполнять различные действия, например реагировать на поступающие сигналы и события или отправлять сигналы другим виджетам. Qt предоставляет полный арсенал виджетов: от кнопок меню до диалоговых окон, необходимых для создания профессиональных приложений. Если вам окажется недостаточно этих виджетов, то можно создать свои собственные, наследуя классы уже существующих.

Иерархия, показанная на рис. 5.1, содержит классы виджетов. Во второй части книги приведено описание большинства из них. Классы, не описанные в этой части, можно найти в других главах книги: `QMenu` (см. *главу 31*), `QGLWidget` (см. *главу 23*), `QMainWindow` (см. *главу 34*), `QGraphicsView` (см. *главу 21*).

Класс *QWidget*

Класс `QWidget` является фундаментальным для всех классов виджетов. Его интерфейс содержит 254 метода, 53 свойства и массу определений, необходимых каждому из виджетов, например, для изменения размеров, местоположения, обработки событий и др. Сам класс `QWidget`, как видно из рис. 5.1, унаследован от класса `QObject`, а значит, может использовать механизм сигналов/слотов и механизм объектной иерархии. Благодаря этому виджеты могут иметь потомков, которые отображаются внутри предка. Это очень важно, так как каждый виджет может служить контейнером для других виджетов, — то есть в Qt нет разделения между элементами управления и контейнерами. Виджеты в контейнерах могут выступать в роли контейнеров для других виджетов, и так до бесконечности. Например, диалоговое окно содержит кнопки **Ok** и **Cancel** (Отмена) — следовательно, оно является контейнером. Это удобно еще и потому, что если виджет-предок станет недоступным или невидимым, то виджеты-потомки автоматически примут его состояние.

Виджеты без предка называются *виджетами верхнего уровня* (top-level widgets) и имеют свое собственное окно. Все виджеты без исключения могут быть виджетами верхнего уровня. Позиция виджетов-потомков внутри виджета-предка может изменяться методом `setGeometry()` вручную или автоматически, с помощью специальных классов компоновки

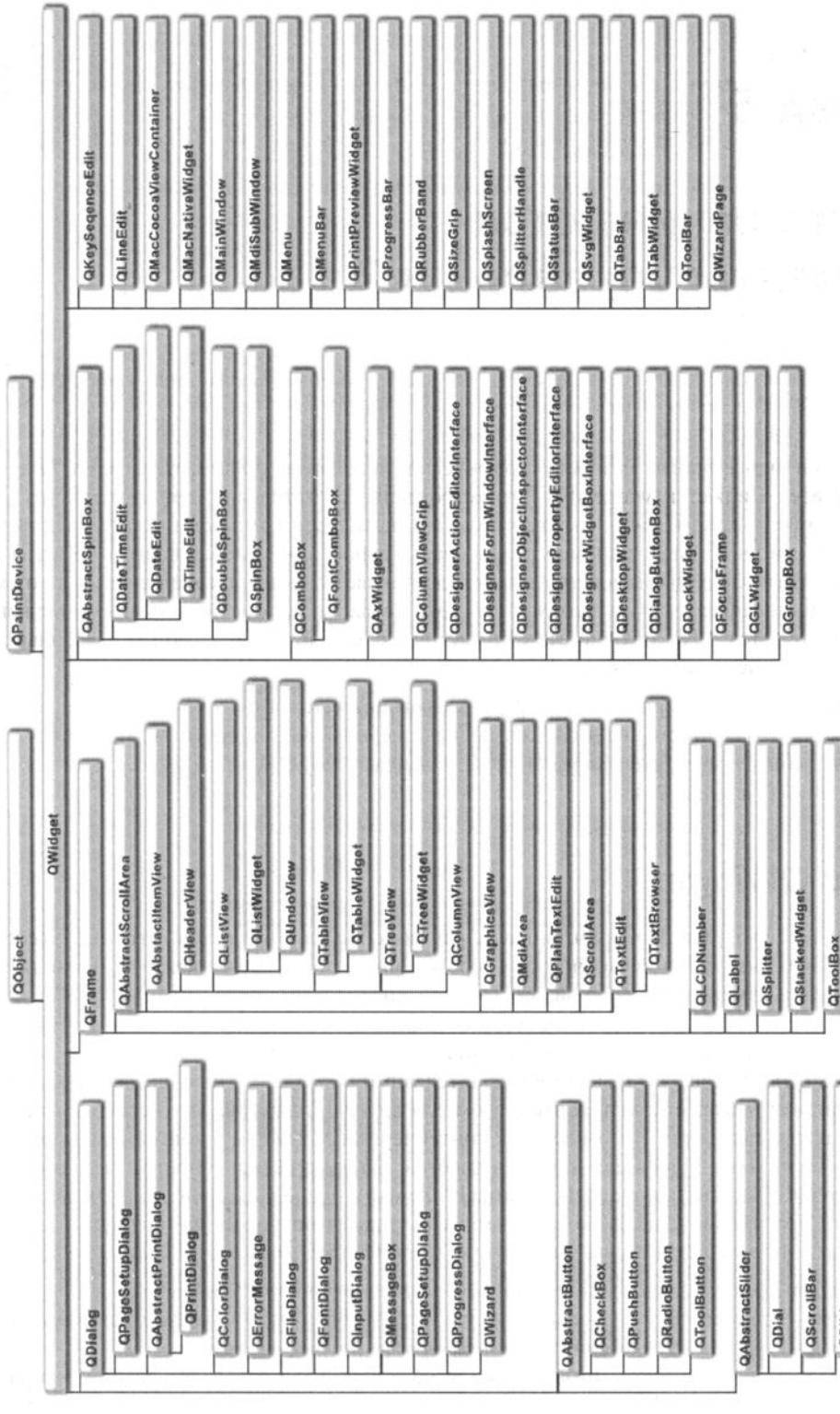


Рис. 5.1. Иерархия классов виджетов

(layouts) (см. *главу 6*). Для отображения виджета на экране вызывается метод `show()`, а для скрытия — метод `hide()`.

> ***После создания виджета верхнего уровня нужно вызвать метод show()***
>
> Не забывайте, что после создания виджета верхнего уровня, чтобы показать его на экране, нужно вызвать метод `show()`, иначе и его окно, и виджеты-потомки будут невидимыми.

Класс `QWidget` и большинство унаследованных от него классов имеют конструктор с двумя параметрами:

```
QWidget(QWidget* pwgt = 0, Qt::WindowFlags f = 0)
```

Из определения видно, что не обязательно передавать параметры в конструктор, так как они равны нулю по умолчанию. А это значит, что если конструктор вызывается без аргументов, то созданный виджет станет виджетом верхнего уровня. Второй параметр: `Qt::WindowFlags` служит для задания свойств окна, и с его помощью можно управлять внешним видом окна и режимом отображения (чтобы окно не перекрывалось другими окнами и т. д.). Чтобы изменить внешний вид окна, необходимо во втором параметре конструктора передать значения модификаторов, объединенные с типом окна (рис. 5.2) побитовой операцией ИЛИ, обозначенной символом `|`. Аналогичного результата можно добиться вызовом метода `setWindowFlags()`. Например:

```
wgt.setWindowFlags(Qt::Window | Qt::WindowTitleHint |
                   Qt::WindowStaysOnTopHint);
```

На рис. 5.2 изображены некоторые из вариантов применения этих значений.



Рис. 5.2. Вид окон виджетов верхнего уровня

ЧТОБЫ ОКНО ВСЕГДА НАХОДИЛОСЬ НА ПЕРЕДНЕМ ПЛАНЕ!

Значение `Qt::WindowStaysOnTopHint` не изменяет внешний вид окна, а лишь рекомендует, чтобы окно всегда находилось на переднем плане и не перекрывалось другими окнами.

При помощи слот-метода `setWindowTitle()` устанавливается надпись заголовка окна. Но это имеет смысл только для виджетов верхнего уровня. Например:

```
wgt.setWindowTitle("My Window");
```

Слот `setEnabled()` устанавливает виджет в *доступное* (enabled) или *недоступное* (disabled) состояние. Параметр `true` соответствует доступному, а `false` — недоступному состоянию. Чтобы узнать, в каком состоянии находится виджет, вызовите метод `isEnabled()`.

При создании собственных классов виджетов важно, чтобы виджет был в состоянии обрабатывать события (см. *главу 14*). Например, для обработки событий мыши необходимо перезаписать хотя бы один из следующих методов: `mousePressEvent()`, `mouseMoveEvent()`, `mouseReleaseEvent()` или `mouseDoubleClickEvent()`.

Размеры и координаты виджета

Виджет представляет собой прямоугольную область (рис. 5.3). Существует целый ряд методов, с помощью которых можно узнать местонахождение виджета и его размеры. Методы `size()`, `height()` и `width()` возвращают размеры виджета. При этом, если вызовы `height()` и `width()` вернут значения высоты и ширины целого типа, соответственно, то вызов метода `size()` вернет объект класса `QSize` (см. *главу 17*), хранящий ширину и высоту виджета.

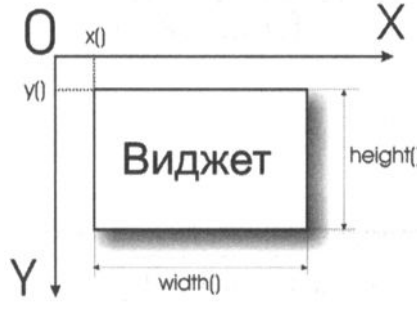


Рис. 5.3. Виджет в области экрана (или предка)

Методы `x()`, `y()` и `pos()` служат для определения координат виджета. Первые два метода возвращают целые значения координат по осям *X* и *Y*, а метод `pos()` — объект класса `QPoint` (см. *главу 17*), хранящий обе координаты.

Метод `geometry()` возвращает объект класса `QRect` (см. *главу 17*), описывающий положение и размеры виджета.

Положение виджета можно изменить методом `move()`, а его размеры — методом `resize()`. Например:

```
pwgt->move(5, 5);
pwgt->resize(260, 330);
```

Одновременно изменить и положение, и размеры виджета можно, вызвав метод `setGeometry()`. Первый параметр этого метода задает координату левого верхнего угла

виджета по оси *X*, второй — по оси *Y*, третий задает ширину, а четвертый — высоту. Например, следующий вызов эквивалентен двум ранее приведенным вызовам `move()` и `resize()`:

```
pwgt->setGeometry(5, 5, 260, 330);
```

Механизм закулисного хранения

Техника *закулисного хранения* (Backing Store) заключается в запоминании в памяти компьютера растровых изображений для всех виджетов окна в любое время, что позволяет очень быстро помещать нужную часть хранимой области без вызова системой событий рисования (PaintEvent). При этом не играет роли, насколько сложно рисование самого виджета. Вызов события рисования для виджета выполняется только в тех случаях, когда это действительно необходимо, — например, для изменения фона. Эта техника позволяет значительно повысить производительность.

Установка фона виджета

Виджету можно задать фон, причем это может быть цвет или растровое изображение. Для заполнения сплошным цветом или растровым изображением необходимо сначала создать объект палитры (см. *главу 13*), а затем вызовом метода `setPalette()` установить его в виджете.

Виджет имеет важное свойство `autoFillBackground`, которое по умолчанию равно `false`. Вследствие этого все потомки виджета не заполняются фоном и, соответственно, невидимы. Установив это свойство равным `true`, вы тем самым заставите виджет заполнять фон автоматически, что сделает его видимым. Например:

```
wgt.setAutoFillBackground(true);
```

В листинге 5.1 создается виджет верхнего уровня `wgt`, который передается двум другим виджетам (указатели `pwgt1` и `pwgt2`) в качестве предка.

Листинг 5.1. Создание виджета верхнего уровня (файл main.cpp)

```
#include <QtWidgets>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QWidget* pwgt1 = new QWidget(&wgt);
    QPalette pal1;
    pal1.setColor(pwgt1->backgroundRole(), Qt::blue);
    pwgt1->setPalette(pal1);
    pwgt1->resize(100, 100);
    pwgt1->move(25, 25);
    pwgt1->setAutoFillBackground(true);

    QWidget* pwgt2 = new QWidget(&wgt);
    QPalette pal2;
```

```
    pal2.setBrush(pwgt2->backgroundRole(), QBrush(QPixmap(":/stone.jpg")));
    pwgt2->setPalette(pal2);
    pwgt2->resize(100, 100);
    pwgt2->move(75, 75);
    pwgt2->setAutoFillBackground(true);

    wgt.resize(200, 200);
    wgt.show();

    return app.exec();
}
```

Первому виджету методом `setPalette()` передается объект палитры, который устанавливает в нем сплошной цвет фона (голубой). После изменения его размеров (методом `resize()`) и перемещения в области виджета-предка (методом `move()`) с помощью метода `setAutioFillBackround()` свойству `autoFillBackground` присваивается значение `true`, чтобы виджет стал видимым.

Со вторым виджетом-потомком (указатель `pwgt2`) выполняются те же операции, что и с первым. Разница заключается в том, что во втором виджете в качестве фона устанавливается растровое изображение из файла stone.jpg при помощи объекта палитры `pal2`.

В результате один виджет заполнен сплошным цветом, а другой — растровым изображением (рис. 5.4).

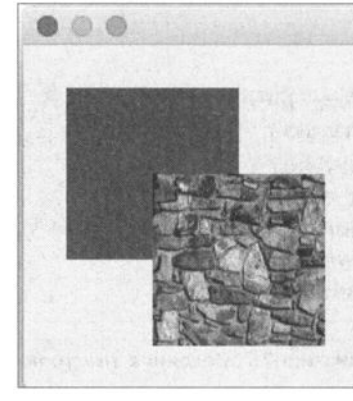


Рис. 5.4. Виджеты с фоном

Изменение указателя мыши

Класс указателя (курсора) мыши `QCursor` определен в файле QCursor. Указатель представляет собой небольшую растровую картинку, информирующую пользователя о позиции мыши на экране. В зависимости от местоположения, внешний вид указателя может меняться. В большинстве случаев это стрелка, но, скажем, при попадании на границу окна он может превратиться в двунаправленную стрелку, информируя пользователя, что размеры окна могут быть изменены.

Установить изображение указателя можно методом `setCursor()`, передав ему одно из приведенных в табл. 5.1 значений.

Таблица 5.1. *Значения* `CursorShape` *пространства имен Qt*

Значение	Вид	Описание
`ArrowCursor`		Стандартный указатель стрелки, он появляется поверх большинства виджетов. Служит для указания, выбора или перемещения объекта

Таблица 5.1 *(окончание)*

Значение	Вид	Описание
`UpArrowCursor`		Стрелка, показывающая вверх. Применение этого указателя зависит от конкретной ситуации
`CrossCursor`		Крестообразный указатель служит для выделения прямоугольных областей. Может появиться над любым виджетом, допускающим эту операцию
`WaitCursor`		Указатель ожидания — появляется над любым виджетом или позицией при выполнении операции в фоновом режиме
`IbeamCursor`		I-образный текстовый указатель (I-beam cursor) представляет собой вертикальную линию. Появляется над текстом для его изменения, выбора и перемещения
`PointingHandCursor`		Указатель в виде руки — появляется над гипертекстовыми ссылками
`ForbiddenCursor`		Указатель невозможности входа — появляется над объектом-приемником при проведении операций перетаскивания, сигнализируя о том, что принимающая сторона не в состоянии принять перетаскиваемый объект
`WhatsThisCursor`		Указатель с вопросом — появляется поверх большинства виджетов для получения контекстно-зависимой помощи
`SizeVerCursor`		Указатель изменения вертикального размера окна — появляется поверх регулируемой границы окна
`SizeHorCursor`		Указатель изменения горизонтального размера окна — появляется поверх регулируемой границы окна
`SizeBDiagCursor`		Указатель изменения размеров окна по диагонали — появляется поверх регулируемой границы окна
`SizeFDiagCursor`		Указатель изменения размеров окна по другой диагонали — появляется поверх регулируемой границы окна
`SizeAllCursor`		Указатель для изменения местоположения окна — сигнализирует о готовности окна быть перемещенным
`SplitVCursor`		Указатель изменения высоты для разделенных виджетов — появляется над границей между двумя разделенными виджетами. Разделение виджетов описано в *главе 6*
`SplitHCursor`		Указатель изменения ширины для разделенных виджетов — появляется над границей между двумя разделенными виджетами. Разделение виджетов описано в *главе 6*
`OpenHandCursor`		Указатель в виде разжатой руки — сигнализирует о возможности перемещения частей изображения в видимой области
`ClosedHandCursor`		Указатель в виде сжатой руки — сигнализирует о готовности перемещения частей изображения в видимой области
`BlankCursor`	Пустой указатель	Пустой указатель — говорит о невозможности использования мыши

Установка изображения указателя для всего приложения

Вызов статического метода `QGuiApplication::setOverrideCursor()` устанавливает изображение указателя для всего приложения. Это может понадобиться, например, для информирования пользователя о том, что приложение выполняет интенсивную, продолжительную по времени операцию и не в состоянии реагировать на команды. В этот момент все вид-

жеты должны отображать указатель мыши в виде песочных часов, для чего вызывается метод `QGuiApplication::setOverrideCursor(Qt::WaitCursor)`. Когда приложение снова будет в состоянии выполнять команды пользователя, вызовом статического метода `QGuiApplication::restoreOverrideCursor()` указателю мыши возвращается его прежний вид.

В классе `QCursor` содержится метод `pos()`, который возвращает текущую позицию указателя мыши относительно левого верхнего угла экрана. При помощи метода `setPos()` можно перемещать указатель мыши.

Чтобы создать собственное изображение указателя мыши, нужны два растровых изображения типа `QBitmap`. Эти изображения должны иметь одинаковые размеры, а одно из них представлять собой битовую маску. В тех местах, на которых маска будет иметь цвет `color1`, будет нарисовано само изображение указателя, а в местах, где маска будет иметь цвет `color0`, — изображение будет прозрачно.

Более простой способ — это использование объекта класса `QPixmap`. Результат выполнения программы (листинг 5.2), показанный на рис. 5.5, демонстрирует эту возможность.

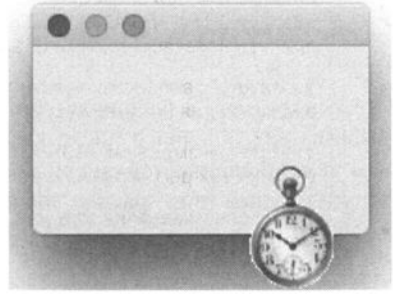


Рис. 5.5. Использование собственного изображения для указателя мыши

Листинг 5.2. Изменение указателя мыши

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;
    QPixmap      pix(":/clock.png");
    QCursor      cur(pix);

    wgt.setCursor(cur);
    wgt.resize(180, 100);
    wgt.show();

    return app.exec();
}
```

В листинге 5.2 сначала создается виджет `wgt`, затем — объект растрового изображения `pix`, в конструктор которого передается имя PNG-файла, представляющего собой растровое изображение и битовую маску (подробное описание этого формата и возможностей класса `QPixmap` можно найти в *главе 19*). Для создания указателя мыши объект растрового изобра-

жения передается в конструктор класса `QCursor` и с помощью метода `setCursor()` устанавливается в виджете.

Стек виджетов

Класс `QStackedWidget` унаследован от класса `QFrame` и представляет собой виджет, который показывает в отдельно взятый промежуток времени только одного из своих потомков.

Виджеты добавляются в стек при помощи метода `addWidget()`. Он принимает указатель на виджеты и возвращает присвоенный виджету идентификационный номер. Удаление виджетов из стека осуществляется вызовом метода `removeWidget()`, в который передается указатель на виджет.

Передавая указатель на виджет в слот `setCurrentWidget()` или идентификационный номер виджета в слот `setCurrentIndex()`, вы делаете его видимым. Идентификационный номер виджета можно узнать вызовом метода `indexOf()`, передав ему указатель на виджет.

Рамки

Класс `QFrame` унаследован от класса `QWidget` и расширяет его возможностью отображения рамки. Этот класс является базовым для большого числа классов виджетов (см. рис. 5.1). Стиль рамки может быть разным, и устанавливается он с помощью метода `setFrameStyle()`, которому передаются флаги формы и флаги теней рамки. Значения соединяются друг с другом побитовой операцией `|` (ИЛИ).

Существуют три флага теней (табл. 5.2): `QFrame::Raised`, `QFrame::Plain` и `QFrame::Sunken`. С их помощью достигается эффект вогнутости или выпуклости рамки.

Таблица 5.2. *Примеры рамок*

Флаги	Вид	Флаги	Вид
`Box | Plain`		`HLine | Plain`	
`Box | Raised`		`HLine | Raised`	
`Box | Sunken`		`HLine | Sunken`	
`Panel | Plain`		`VLine | Plain`	
`Panel | Raised`		`VLine | Raised`	
`Panel | Sunken`		`VLine | Sunken`	
`WinPanel | Plain`		`StyledPanel | Plain`	
`WinPanel | Raised`		`StyledPanel | Raised`	
`WinPanel | Sunken`		`StyledPanel | Sunken`	

Для задания внешнего вида рамки можно воспользоваться одной из пяти основных форм (см. табл. 5.2): `QFrame::Box`, `QFrame::Panel`, `QFrame::WinPanel`, `QFrame::HLine` или `QFrame::VLine`. Если нужно, чтобы рамка вообще не отображалась, то тогда в метод `setFrameStyle()` передается значение `QFrame::NoFrame`.

Методом `setContentsMargin()` класса `QWidget` устанавливается расстояние от рамки до содержимого виджета, а методами `setLineWidth()` и `setMidLineWidth()` можно изменять толщину самой рамки.

```
QFrame pfrm = new QFrame;
pfrm->setFrameStyle(QFrame::Box | QFrame::Sunken);
pfrm->setLineWidth(3);
```

В этом примере создается виджет рамки, в котором методом `setFrameStyle()` устанавливается нужный стиль рамки, а методом `setLineWidth()` — ее толщина.

Виджет видовой прокрутки

Базовый класс для видовой прокрутки `QAbstractScrollArea` унаследован от класса `QFrame` и представляет собой окно для просмотра только части информации. Сам виджет видовой прокрутки реализует класс `QScrollArea`.

Этот виджет может размещать виджеты потомков, а если хотя бы один из них выйдет за границы окна просмотра, то автоматически появляются вертикальная и/или горизонтальная полосы прокрутки. С их помощью можно перемещать части виджета в область просмотра. Если вы хотите, чтобы полосы прокрутки были видны всегда, то нужно передать в методы управления поведением полос значение `Qt::ScrollBarAlwaysOn`. Например:

```
QScrollArea sa;
sa.setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
sa.setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
```

Как видно из рис. 5.6, виджет видовой прокрутки является совокупностью сразу нескольких виджетов, работающих вместе. Указатели на эти виджеты можно получить обозначенными на рисунке методами. Осуществить доступ к виджету области просмотра можно посредством метода `QAbstractScrollArea::viewport()`. Методы `verticalScrollBar()` и `horizontalScrollBar()` возвращают указатели на виджеты соответственно вертикальной и горизонтальной полосы прокрутки класса `QScrollBar`. Метод `cornerWidget()` возвращает указатель на виджет, находящийся в правом нижнем углу.

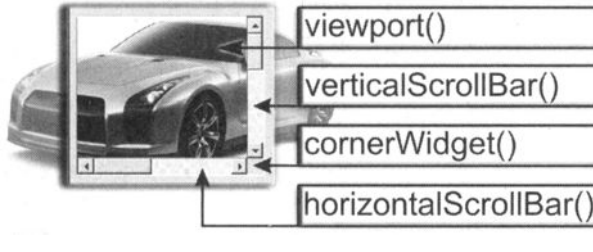


Рис. 5.6. Структура виджета видовой прокрутки

Установить виджет в виджете видовой прокрутки можно при помощи метода `setWidget()`, передав ему указатель на него. Эта операция автоматически сделает переданный виджет потомком виджета области просмотра. Указатель установленного виджета всегда можно получить методом `widget()`. Удаление виджета из `QScrollArea` осуществляется вызовом метода `removeChild()`.

На рис. 5.7 показан результат работы программы применения видовой прокрутки (листинг 5.3). Программа позволяет перемещать части изображения в видимую область окна.

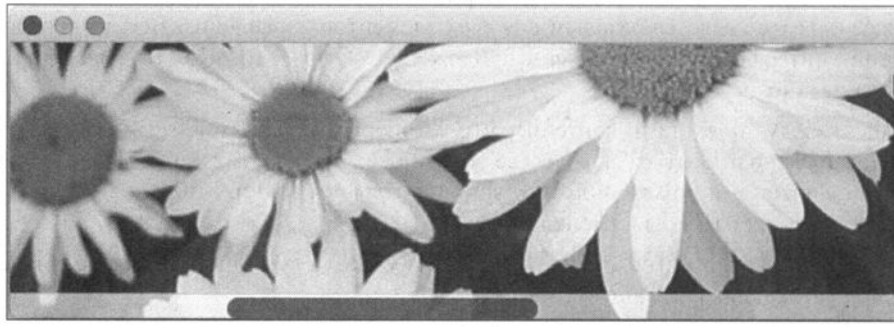


Рис. 5.7. Пример виджета видовой прокрутки

Листинг 5.3. Применение видовой прокрутки (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QScrollArea  sa;

    QWidget*     pwgt = new QWidget;
    QPixmap      pix(":/img.jpg");

    QPalette pal;
    pal.setBrush(pwgt->backgroundRole(), QBrush(pix));
    pwgt->setPalette(pal);
    pwgt->setAutoFillBackground(true);
    pwgt->setFixedSize(pix.width(), pix.height());

    sa.setWidget(pwgt);
    sa.resize(350, 150);
    sa.show();

    return app.exec();
}
```

В листинге 5.3 создается виджет видовой прокрутки `sa`. Затем создается обычный виджет (указатель `pwgt`) и объект растрового изображения `pix`. Объект растрового изображения инициализируется файлом img.jpg, который затем устанавливается вызовом метода `setPalette()` в качестве заднего фона виджета. Для того чтобы виджет был виден, метод

`setAutoFillBackground()` включает режим автоматического заполнения фона. Размеры виджета (указатель `pwgt`) приводятся в соответствие с размерами растрового изображения вызовом метода `setFixedSize()`. Затем виджет видовой прокрутки `sa` вызовом метода `setWidget()` добавляет в свое окно созданный нами виджет.

Резюме

В центре создания пользовательского интерфейса стоит понятие *виджет* (элемент управления). Класс `QWidget` является базовым для всех элементов управления. Все, из чего в основном состоит интерфейс пользователя в приложениях Qt, — это объекты класса `QWidget` и унаследованных от него классов.

Разделение между виджетами-контейнерами и просто виджетами отсутствует — любой виджет может использоваться в качестве контейнера для других виджетов. К основным операциям с виджетами, помимо *показа* (show) и *скрытия* (hide), относятся методы, позволяющие изменять их размеры и расположение.

Позиция и размеры виджетов внутри виджета-предка могут при использовании классов компоновки устанавливаться автоматически.

Виджеты верхнего уровня имеют свое собственное окно, которое можно по-разному декорировать, например изменять рамку окна.

В каждом виджете можно устанавливать указатели мыши. Qt предоставляет ряд предопределенных изображений таких указателей, которые можно использовать для установки. Имеется также возможность создавать свои собственные указатели из растровых изображений.

Класс `QFrame` унаследован от класса `QWidget` и представляет собой прямоугольник с рамкой, стиль которой можно менять.

Класс `QStackedWidget` показывает в отдельно взятый промежуток времени только одного из потомков. Этим свойством пользуются тогда, когда есть много виджетов и нужно, чтобы только один из них в определенное время был видимым.

Виджет видовой прокрутки предоставляет окно для просмотра только части информации. Этот виджет применяется для отображения информации, размеры которой превышают выделенную для нее область просмотра.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/05-510/** или с помощью следующего QR-кода (рис. 5.8):

Рис. 5.8. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 6

Управление автоматическим размещением элементов

Порядок и беспорядок зависят от организации.
Сунь Цзы

Классы *компоновки виджетов* (Layouts) являются одной из сильных сторон Qt (не надо путать компоновку виджетов и компоновку приложения — это совсем разные вещи). По сути, это контейнеры, которые после изменения размеров окна автоматически приводят в соответствие размеры и координаты виджетов, находящихся в нем. Хотя они ничего не добавляют к функциональной части самой программы, тем не менее, они очень важны для внешнего вида окон приложения. Компоновка определяет расположение различных виджетов относительно друг друга.

Конечно, можно вручную размещать виджеты в окнах приложения, но это существенно усложняет разработку. Ведь тогда, чтобы заново упорядочить элементы, нужно будет отлавливать и обрабатывать изменение размеров окна приложения. Подобные ситуации хорошо известны программистам на языке Visual Basic, которые вынуждены писать для этого сложные методы.

Еще один из недостатков размещения вручную состоит в том, что если приложение поддерживает несколько языков, то, поскольку слова в разных языках имеют разную длину, необходим механизм, который мог бы в процессе работы программы динамически подправлять и изменять размеры и координаты виджетов, — иначе части текста на другом языке могут оказаться «отрезанными». Классы компоновки библиотеки Qt выполняют эту непростую работу за вас. Более того, классы компоновки могут инвертировать направление размещения элементов, что может быть полезно для пишущих справа налево, например, в Израиле или в странах арабского Востока.

Qt предоставляет так называемые *менеджеры компоновки*, позволяющие организовать размещение виджетов на поверхности другого виджета. Основу их работы определяет возможность каждого виджета сообщать о том, сколько ему необходимо места, может ли он быть растянут по вертикали и/или горизонтали и т. п.

Менеджеры компоновки (layout managers)

Менеджеры компоновки предоставляют возможности для горизонтального, вертикального и табличного размещения не только виджетов, но и встроенных компоновок. Это позволяет конструировать довольно сложные размещения.

Фундаментом для всей группы менеджеров компоновки является класс `QLayout` — абстрактный класс, унаследованный сразу от двух классов: `QObject` и `QLayoutItem` (рис. 6.1). Этот класс определен в заголовочном файле QLayout.

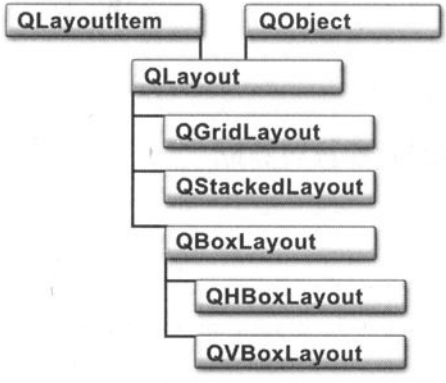


Рис. 6.1. Иерархия классов менеджеров компоновки

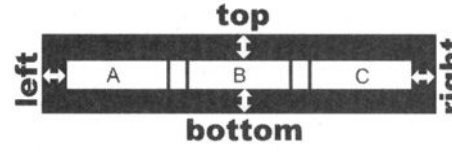


Рис. 6.2. Размещение виджетов по горизонтали

Создать собственного менеджера компоновки

Создание своего собственного класса компоновки — явление очень редкое, так как практически все задачи размещения можно решить стандартными классами размещения, предоставляемыми Qt. Но если вам понадобится создать свой собственный менеджер компоновки, то можно унаследовать класс `QLayout`, реализовав методы `addItem()`, `count()`, `setGeometry()`, `takeAt()` и `itemAt()`.

От класса `QLayout` унаследованы классы `QGridLayout` и `QBoxLayout` (см. рис. 6.1). Класс `QGridLayout` управляет табличным размещением, а от `QBoxLayout` унаследованы два класса: `QHBoxLayout` и `QVBoxLayout` — для горизонтального и вертикального размещения.

По умолчанию между виджетами остается небольшое расстояние, необходимое для их визуального разделения. Задать его можно с помощью метода `setSpacing()`, передав в него нужное значение в пикселах. Методом `setContentsMargins()` можно установить отступ виджетов от границ сторон компоновки — их четыре: слева (`left`), сверху (`top`), справа (`right`) и снизу (`bottom`). Рис. 6.2 показывает расположение сторон на примере горизонтального размещения метода `setContentsMargins()`. Промежутки между кнопками **A**, **B** и **B**, **C** на рис. 6.2 задает метод `setSpacing()`.

При помощи метода `addWidget()` выполняется добавление виджетов в компоновку, а с помощью метода `addLayout()` можно добавлять встроенные менеджеры компоновки. Если понадобится удалить какой-либо виджет из компоновки, то следует воспользоваться методом `removeWidget()`, передав ему указатель на этот виджет.

Объектная иерархия виджетов и объектов размещения отделены друг от друга

Виджеты — это потомки других виджетов, а объекты размещения — потомки других объектов размещения.

Объекты размещения отвечают за правильное размещение виджетов и присвоение им нужных виджетов-предков. Так что вам не нужно беспокоиться о том, чтобы присваивать объекты предков, поскольку это будет сделано за вас автоматически.

Горизонтальное и вертикальное размещение

Для горизонтального или вертикального размещения можно воспользоваться классом `QBoxLayout` или унаследованными от него классами `QHBoxLayout` и `QVBoxLayout`.

Классы `QHBoxLayout` и `QVBoxLayout`, унаследованные от `QBoxLayout`, отличаются от него тем, что в их конструктор не передается параметр, говорящий о способе размещения, — горизонтальный он или вертикальный, так как порядок размещения заложен уже в самом классе. Эти классы сами выполняют горизонтальное или вертикальное размещение: слева направо или сверху вниз.

Класс *QBoxLayout*

Объект класса `QBoxLayout` может управлять как горизонтальным, так и вертикальным размещением. Для того чтобы задать способ размещения, первым параметром конструктора должно быть одно из следующих значений:

- `LeftToRight` — горизонтальное размещение, заполнение осуществляется слева направо;
- `RightToLeft` — горизонтальное размещение, заполнение выполняется справа налево;
- `TopToBottom` — вертикальное размещение, заполнение осуществляется сверху вниз;
- `BottomToTop` — вертикальное размещение, заполнение выполняется снизу вверх.

Этот класс расширяет класс `QLayout` методами вставки на заданную позицию: виджета — `insertWidget()`, встроенной компоновки — `insertLayout()`, расстояния между виджетами — `insertSpacing()` и фактора растяжения — `insertStretch()`.

К компоновке при помощи метода `addSpacing()` можно добавить заданное расстояние между двумя виджетами.

Класс `QBoxLayout` определяет свой собственный метод `addWidget()` для добавления виджетов в компоновку с возможностью указания в дополнительном параметре фактора растяжения (по умолчанию этот параметр равен нулю). Демонстрация этой возможности показана на рис. 6.3, а текст соответствующей программы приведен в листинге 6.1.



Рис. 6.3. Кнопки с факторами растяжений

Листинг 6.1. Добавление виджетов в компоновку с указанием в дополнительном параметре фактора растяжения (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QPushButton* pcmdA = new QPushButton("A");
    QPushButton* pcmdB = new QPushButton("B");
    QPushButton* pcmdC = new QPushButton("C");
```

```
    //Layout setup
    QBoxLayout* pbxLayout = new QBoxLayout(QBoxLayout::LeftToRight);
    pbxLayout->addWidget(pcmdA, 1);
    pbxLayout->addWidget(pcmdB, 2);
    pbxLayout->addWidget(pcmdC, 3);
    wgt.setLayout(pbxLayout);

    wgt.resize(450, 40);
    wgt.show();

    return app.exec();
}
```

В листинге 6.1 создаются три кнопки: **A**, **B** и **C**, которые помещаются в компоновку с помощью метода `QBoxLayout::addWidget()`, причем во втором параметре этого метода указывается параметр растяжения. При создании объекта компоновки в конструктор передается параметр `QBoxLayout::LeftToRight`, который задает горизонтальное размещение элементов слева направо. Вызов метода `QWidget::setLayout()` устанавливает компоновку в виджете `wgt`.

Возможно, что приведенный в листинге 6.1 пример произведет шокирующий эффект, поскольку создаваемые кнопки (указатели `pcmdA`, `pcmdB` и `pcmdC`) не имеют объекта-предка, а это значит, что некому будет позаботиться об освобождении памяти, выделенной для этих виджетов. Так что же все-таки происходит? Неужели программа и вправду содержит ошибку, которая может привести к утечке памяти (memory leak)?

На самом деле беспокоиться не о чем, так как за присвоение виджета-предка отвечает сама компоновка. При вызове метода `setLayout()` всем помещенным в компоновку виджетам будет присвоен виджет предка — в нашем случае это `wgt`.

Факторы растяжения можно самостоятельно добавлять в компоновки, для чего существует метод `addStretch()`. В этом случае фактор растяжения образно можно сравнить с пружиной, которая находится между виджетами и может иметь различную упругость в соответствии с задаваемым параметром. На рис. 6.4 показан пример добавления фактора растяжения между двумя виджетами для расположения виджетов по краям окна.

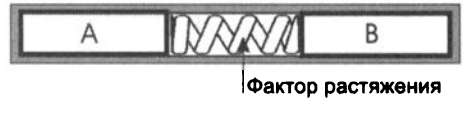


Рис. 6.4. Добавление фактора растяжения между виджетами **A** и **B**

Результат выполнения программы (листинг 6.2), показанный на рис. 6.5, демонстрирует добавление фактора растяжения и представляет собой небольшую модификацию предыду-

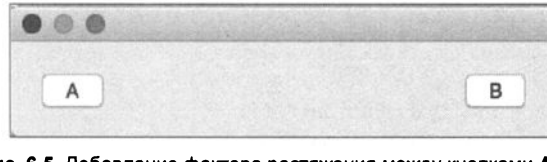


Рис. 6.5. Добавление фактора растяжения между кнопками **A** и **B**

щей программы (см. листинг 6.1), выполнение которой приведено на рис. 6.3, — только вместо одной из кнопок здесь добавляется фактор растяжения.

Листинг 6.2. Вместо одной из кнопок добавляется фактор растяжения (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QPushButton* pcmdA = new QPushButton("A");
    QPushButton* pcmdB = new QPushButton("B");

    //Layout setup
    QBoxLayout* pbxLayout = new QBoxLayout(QBoxLayout::LeftToRight);
    pbxLayout->addWidget(pcmdA);
    pbxLayout->addStretch(1);
    pbxLayout->addWidget(pcmdB);
    wgt.setLayout(pbxLayout);

    wgt.resize(350, 40);
    wgt.show();

    return app.exec();
}
```

В листинге 6.2 создается виджет класса `QWidget` и две кнопки: **A** и **B**. После создания объекта компоновки для горизонтального размещения в него вызовом метода `QLayout::addWidget()` добавляется первая кнопка (указатель `pcmdA`). Затем вызов метода `QBoxLayout::addStretch()` добавляет фактор растяжения, после чего добавляется вторая кнопка (указатель `pcmdB`).

Горизонтальное размещение *QHBoxLayout*

Объекты класса `QHBoxLayout` упорядочивают все виджеты только в горизонтальном порядке — слева направо. Его применение аналогично использованию класса `QBoxLayout`, но передавать в конструктор дополнительный параметр, задающий горизонтальный порядок размещения, не нужно. Окно программы, которая упорядочивает виджеты при помощи объекта класса `QHBoxLayout` (листинг 6.3), показано на рис. 6.6.

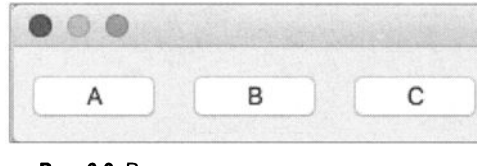


Рис. 6.6. Размещение кнопок по горизонтали

Листинг 6.3. Упорядочивание виджетов при помощи объекта класса QHBoxLayout (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QPushButton* pcmdA = new QPushButton("A");
    QPushButton* pcmdB = new QPushButton("B");
    QPushButton* pcmdC = new QPushButton("C");

    //Layout setup
    QHBoxLayout* phbxLayout = new QHBoxLayout;
    phbxLayout->setContentsMargins(10, 10, 10, 10);
    phbxLayout->setSpacing(20);
    phbxLayout->addWidget(pcmdA);
    phbxLayout->addWidget(pcmdB);
    phbxLayout->addWidget(pcmdC);
    wgt.setLayout(phbxLayout);

    wgt.show();

    return app.exec();
}
```

В листинге 6.3 создаются три кнопки: **A**, **B** и **C** (указатели `pcmdA`, `pcmdB` и `pcmdC`). Затем создается объект класса `QHBoxLayout` для горизонтального размещения дочерних виджетов. Метод `QLayout::setContentsMargins()` устанавливает толщину рамки в 10 пикселов со всех четырех сторон. Метод `QLayout::setSpacing()` задает расстояние между виджетами, равное 20 пикселам. Три вызова метода `QLayout::addWidget()` добавляют виджеты кнопок в компоновку.

Вертикальное размещение *QVBoxLayout*

Компоновка `QVBoxLayout` унаследована от `QBoxLayout` и упорядочивает все виджеты только по вертикали — сверху вниз. В остальном она ничем не отличается от классов `QBoxLayout` и `QHBoxLayout`. Если заменить в листинге 6.3 имя класса `QHBoxLayout` на `QVBoxLayout`, то в результате получится программа, окно которой показано на рис. 6.7.

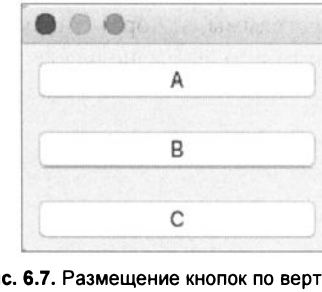


Рис. 6.7. Размещение кнопок по вертикали

Вложенные размещения

Размещая одну компоновку внутри другой, можно создавать размещения практически любой сложности. Для организации вложенных размещений существует метод `addLayout()`, в который вторым параметром передается фактор растяжения для добавляемой компоновки.

На рис. 6.8 показан пример вложенного размещения двух менеджеров компоновки (листинг 6.4) — в компоновку `QVBoxLayout` помещается компоновка `QHBoxLayout`.

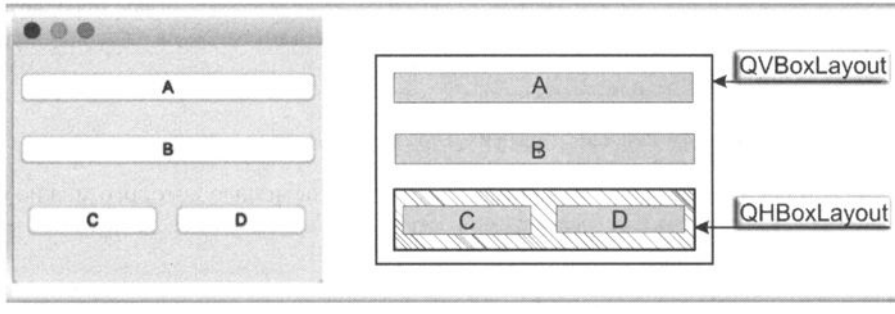


Рис. 6.8. Вложенное размещение

Листинг 6.4. Вложенное размещение двух менеджеров компоновки (файл Layout.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QPushButton* pcmdA = new QPushButton("A");
    QPushButton* pcmdB = new QPushButton("B");
    QPushButton* pcmdC = new QPushButton("C");
    QPushButton* pcmdD = new QPushButton("D");

    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    QHBoxLayout* phbxLayout = new QHBoxLayout;
    phbxLayout->setContensMargins(5, 5, 5, 5);
    phbxLayout->setSpacing(15);
    phbxLayout->addWidget(pcmdC);
    phbxLayout->addWidget(pcmdD);

    pvbxLayout->setContentsMargins(5, 5, 5, 5);
    pvbxLayout->setSpacing(15);
    pvbxLayout->addWidget(pcmdA);
    pvbxLayout->addWidget(pcmdB);
    pvbxLayout->addLayout(phbxLayout);
    wgt.setLayout(pvbxLayout);
    wgt.show();

    return app.exec();
}
```

В листинге 6.4 приводится фрагмент программы, окно которой показано на рис. 6.8. Программа будет читаться гораздо лучше, если сначала создать все виджеты, а затем объекты менеджеров компоновки, что мы и делаем. Используя метод для установки толщины рамки `setContentsMargins()`, мы устанавливаем ее равной пяти пикселам для обеих компоновок со всех четырех сторон компоновки, а с помощью метода `setSpacing()` устанавливаем расстояние между виджетами в 15 пикселов, также для обеих компоновок. В горизонтальную компоновку мы добавляем виджеты кнопок `pcmdC` и `pcmdD`. Затем виджеты кнопок `pcmdA` и `pcmdB` по очереди передаются в метод `QLayout::addWidget()` вертикальной компоновки `pvbxLayout`, после чего при помощи метода `QBoxLayout:: addLayout()` в нее передается объект горизонтальной компоновки `phbxLayout`. Вызов метода `QWidget::setLayout()` устанавливает вертикальную компоновку `pvbxLayout` в виджете `wgt`.

Табличное размещение *QGridLayout*

Для табличного размещения служит класс `QGridLayout`, с помощью которого можно быстро создавать сложные по структуре размещения. Таблица состоит из ячеек, позиции которых задаются строками и столбцами.

> ***Создание таблицы из двух столбцов***
>
> Если вам нужна таблица, которая состоит из двух столбцов, то можно воспользоваться классом `QFormLayout`, — это поможет реализовать более компактный код, чем с использованием класса `QGridLayout`. Подобная ситуация встречается часто, например, при создании диалоговых окон, в которых в первом столбце стоят объясняющие надписи, а во втором — виджеты для ввода информации. Добавление виджетов осуществляется вызовом метода `addRow()`, в который передаются сразу два виджета: виджет надписи и функциональный виджет.

Добавить виджет в таблицу можно с помощью метода `addWidget()`, передав ему позицию ячейки, в которую помещается виджет. Иногда необходимо, чтобы виджет занимал сразу несколько позиций, чего можно достичь тем же методом `addWidget()`, указав в дополнительных параметрах количество строк и столбцов, которые будет занимать виджет. В последнем параметре можно задать выравнивание (см. табл. 7.1 в *главе 7*), например, по центру:

```
playout->addWidget(widget, 17, 1, Qt::AlignCenter);
```

Фактор растяжения устанавливается методами `setRowStretch()` и `setColumnStretch()`, но не для каждого виджета в отдельности, а для строки или столбца. Расстояние между виджетами также устанавливается для столбцов или строк методом `setSpacing()`.

Пример, показанный на рис. 6.9 (листинг 6.5), размещает четыре кнопки: **A**, **B**, **C** и **D** в таблице размером 2 на 2 ячейки.

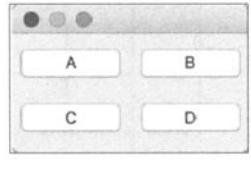


Рис. 6.9. Размещение кнопок в табличном порядке

Листинг 6.5. Размещение кнопок в табличном порядке (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
```

```
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QPushButton* pcmdA = new QPushButton("A");
    QPushButton* pcmdB = new QPushButton("B");
    QPushButton* pcmdC = new QPushButton("C");
    QPushButton* pcmdD = new QPushButton("D");

    QGridLayout* pgrdLayout = new QGridLayout;
    pgrdLayout->setContentsMargins(5, 5, 5, 5);
    pgrdLayout->setSpacing(15);
    pgrdLayout->addWidget(pcmdA, 0, 0);
    pgrdLayout->addWidget(pcmdB, 0, 1);
    pgrdLayout->addWidget(pcmdC, 1, 0);
    pgrdLayout->addWidget(pcmdD, 1, 1);
    wgt.setLayout(pgrdLayout);

    wgt.show();

    return app.exec();
}
```

В листинге 6.5 создается компоновка табличного размещения `pgrdLayout`. Метод `setContentsMargins()` устанавливает отступ от границы в 5 пикселов со всех сторон. Вызов метода `setSpacing()` установит расстояние в 15 пикселов между виджетами. Виджеты кнопок добавляются в компоновку вызовом метода `addWidget()`, последние два параметра которого указывают те строку и столбец, в которых должен быть расположен виджет.

Приложение (листинг 6.6), выполнение которого показано на рис. 6.10, демонстрирует применение табличного размещения на примере калькулятора. В примере задействованы классы стека `QValueStack` и регулярного выражения `QRegExp`, описанные в *главе 4*.

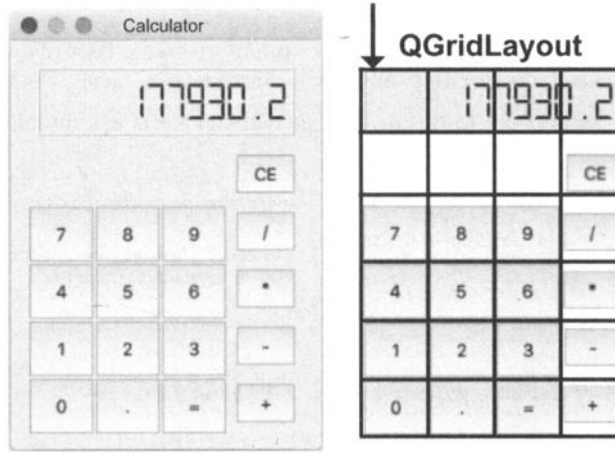


Рис. 6.10. Результат выполнения программы, использующей табличное размещение

Листинг 6.6. Применение табличного размещения на примере калькулятора (файл main.cpp)

```
#include <QApplication>
#include "Calculator.h"

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    Calculator   calculator;

    calculator.setWindowTitle("Calculator");
    calculator.resize(230, 200);

    calculator.show();

    return app.exec();
}
```

В программе, приведенной в листинге 6.6, создается виджет калькулятора `calculator` (см. листинги 6.7–6.11). После изменения размера методом `resize()` вызов метода `show()` отображает калькулятор на экране.

В определении класса `Calculator`, приведенном в листинге 6.7, описываются атрибуты: `m_plcd` — указатель на виджет электронного индикатора, `m_stk` — стек для проведения операций вычисления и `m_strDisplay` — строка, в которую мы будем записывать символы нажатых пользователем кнопок. Метод `createButton()` предназначен для создания кнопок калькулятора, а метод `calculate()` — для вычисления выражений, находящихся в стеке `m_stk`. Слот `slotButtonClicked()` вызывается при нажатии на любую из кнопок калькулятора.

Листинг 6.7. Определение класса `Calculator` (файл Calculator.h)

```
#pragma once

#include <QWidget>
#include <QStack>

class QLCDNumber;
class QPushButton;

// ============================================================================
class Calculator : public QWidget {
    Q_OBJECT
private:
    QLCDNumber*     m_plcd;
    QStack<QString> m_stk;
    QString         m_strDisplay;

public:
    Calculator(QWidget* pwgt = 0);
```

```
    QPushButton* createButton(const QString& str);
    void         calculate   (                  );

public slots:
    void slotButtonClicked();
};
```

При создании электронного индикатора (листинг 6.8) в его конструктор передается количество сегментов, равное 12. Флаг `QLCDNumber::Flat`, переданный в метод `setSegmentStyle()`, задает сегментам индикатора плоский стиль. Метод `setMinimumSize()` переустанавливает минимально возможные размеры виджета индикатора. В массиве `aButtons` определяются надписи для кнопок калькулятора. Виджет электронного индикатора помещается в компоновку вызовом метода `addWidget()`, первые два параметра которого задают его расположение, а последние два — количество занимаемых им строк и столбцов табличной компоновки.

Кнопка **CE**, после своего создания методом `createButton()`, помещается в компоновку методом `addWidget()` на позиции (1,3) (то есть на пересечении второй строки и четвертого столбца — они нумеруются с нуля). Все остальные виджеты кнопок создаются и помещаются в компоновку в цикле с помощью методов `createButton()` и `addWidget()`.

Листинг 6.8. Конструктор `Calculator` (файл Calculator.cpp)

```
Calculator::Calculator(QWidget* pwgt/*= 0*/) : QWidget(pwgt)
{
    m_plcd = new QLCDNumber(12);
    m_plcd->setSegmentStyle(QLCDNumber::Flat);
    m_plcd->setMinimumSize(150, 50);

    QChar aButtons[4][4] = {{'7', '8', '9', '/'},
                            {'4', '5', '6', '*'},
                            {'1', '2', '3', '-'},
                            {'0', '.', '=', '+'}
                           };

    //Layout setup
    QGridLayout* ptopLayout = new QGridLayout;
    ptopLayout->addWidget(m_plcd, 0, 0, 1, 4);
    ptopLayout->addWidget(createButton("CE"), 1, 3);

    for (int i = 0; i < 4; ++i) {
        for (int j = 0; j < 4; ++j) {
           ptopLayout->addWidget(createButton(aButtons[i][j]), i + 2, j);
        }
    }
    setLayout(ptopLayout);
}
```

Метод `createButton()`, приведенный в листинге 6.9, получает строку с надписью и создает нажимающуюся кнопку. После этого вызовом метода `setMinimumSize()` для кнопки уста-

навливаются минимально возможные размеры, а сигнал `clicked()` соединяется со слотом `slotButtonClicked()` вызовом `connect()`.

Листинг 6.9. Метод `createButton()` (файл Calculator.cpp)

```
QPushButton* Calculator::createButton(const QString& str)
{
    QPushButton* pcmd = new QPushButton(str);
    pcmd->setMinimumSize(40, 40);
    connect(pcmd, SIGNAL(clicked()), SLOT(slotButtonClicked()));
    return pcmd;
}
```

Назначение метода `calculate()` (листинг 6.10) состоит в вычислении выражения, содержащегося в стеке `m_stk`. Переменная `dOperand2` получает снятое с вершины стека значение, преобразованное к типу `qreal`. Строковая переменная `strOperation` получает символ операции. Переменная `fOperand1` из стека получает последнее значение, которое также преобразуется к типу `qreal`. В операторах `if` символ операции сравнивается с четырьмя допустимыми и, в случае совпадения, выполняется требуемая операция, результат которой сохраняется в переменной `fResult`. После этого вызовом метода `dislay()` значение переменной `fResult` отображается на электронном индикаторе (указатель `m_plcd`).

Листинг 6.10. Метод `calculate()` (файл Calculator.cpp)

```
void Calculator::calculate()
{
    qreal   fOperand2    = m_stk.pop().toFloat();
    QString strOperation = m_stk.pop();
    qreal   fOperand1    = m_stk.pop().toFloat();
    qreal   fResult      = 0;

    if (strOperation == "+") {
        fResult = fOperand1 + fOperand2;
    }
    if (strOperation == "-") {
        fResult = fOperand1 - fOperand2;
    }
    if (strOperation == "/") {
        fResult = fOperand1 / fOperand2;
    }
    if (strOperation == "*") {
        fResult = fOperand1 * fOperand2;
    }
    m_plcd->display(fResult);
}
```

В слоте `slotButtonClicked()` (листинг 6.11) осуществляется преобразование виджета, выславшего сигнал, к типу `QPushButton`, после чего переменной `str` присваивается текст

надписи на кнопке. Если надпись равна CE, то выполняется операция сброса — очистка стека и установка значения индикатора в 0. Если была нажата цифра или точка, то выполняется ее добавление в конец строки m_strDisplay, она отображается индикатором с последующей актуализацией. При нажатии любой другой кнопки мы считаем, что была нажата кнопка операции. Если в стеке находится менее двух элементов, то отображаемое число и операция заносятся в стек. Иначе в стек заносится отображаемое значение и вызывается метод calculate() для вычисления находящегося в стеке выражения. После этого стек очищается с помощью метода clear() и в него записывается значение результата, отображаемое индикатором, и следующая операция. Если выполняется операция =, то она не будет добавляться в стек.

Листинг 6.11. Метод slotButtonClicked() (файл Calculator.cpp)

```
void Calculator::slotButtonClicked()
{
    QString str = ((QPushButton*)sender())->text();

    if (str == "CE") {
        m_stk.clear();
        m_strDisplay = "";
        m_plcd->display("0");
        return;
    }
    if (str.contains(QRegExp("[0-9]"))) {
        m_strDisplay += str;
        m_plcd->display(m_strDisplay.toDouble());
    }
    else if (str == ".") {
        m_strDisplay += str;
        m_plcd->display(m_strDisplay);
    }
    else {
        if (m_stk.count() >= 2) {
            m_stk.push(QString().setNum(m_plcd->value()));
            calculate();
            m_stk.clear();
            m_stk.push(QString().setNum(m_plcd->value()));
            if (str != "=") {
                m_stk.push(str);
            }
        }
        else {
            m_stk.push(QString().setNum(m_plcd->value()));
            m_stk.push(str);
            m_strDisplay = "";
        }
    }
}
```

Порядок следования табулятора

Пользователь может взаимодействовать с виджетами при помощи мыши и клавиатуры. В последнем случае для выбора нужного виджета используется клавиша табуляции — <Tab>, при нажатии которой происходит переход фокуса согласно установленному порядку от одного виджета к другому. Иногда возникает необходимость в изменении этого порядка, который по умолчанию соответствует очередности установки дочерних виджетов в виджете предка. На рис. 6.11 цифрами изображен порядок смены фокуса с помощью табулятора: при появлении диалогового окна с тремя кнопками фокус будет установлен на кнопке **C** и после нажатия на клавишу табуляции он перейдет на кнопку **B**, а затем — на кнопку **A**.

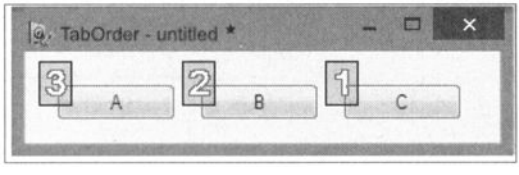


Рис. 6.11. Порядок смены фокуса

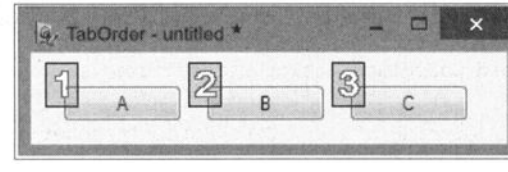


Рис. 6.12. Измененный порядок смены фокуса

Изменить порядок смены фокуса можно с помощью статического метода `QWidget::setTabOrder()`, получающего в качестве параметров два указателя на виджеты. Следующие вызовы изменят порядок следования табулятора, показанный на рис. 6.11, на более логичный порядок, представленный на рис. 6.12:

```
QWidget::setTabOrder(A, B);
QWidget::setTabOrder(B, C);
```

Разделители *QSplitter*

Разделители придуманы для одновременного просмотра различных частей текстовых или графических объектов. В некоторых случаях применение разделителя более предпочтительно, чем размещение с помощью классов компоновки, так как появляется возможность изменения размеров виджетов. Конкретный тому пример — это всем известная программа Проводник (Windows Explorer из ОС Windows) или, например, разделение рабочего окна с текстом на две части в MS Word. Разделители реализованы в классе `QSplitter`, определение которого находится в заголовочном файле QSplitter. С помощью виджета разделителя можно располагать виджеты как горизонтально, так и вертикально. Между виджетами отображается черта разделителя, которую можно двигать с помощью мыши, тем самым изменяя размеры виджетов.

Если необходимо, чтобы виджеты разделителя были проинформированы об изменении размеров, то тогда нужно вызывать метод `setOpaqueResize()`, передав ему значение `true`.

Пример, показанный на рис. 6.13, разделяет два виджета класса `QTextEdit` (листинг 6.12).

Листинг 6.12. Создание виджета вертикального разделителя (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
```

```
    QSplitter    spl(Qt::Vertical);
    QTextEdit*   ptxt1 = new QTextEdit;
    QTextEdit*   ptxt2 = new QTextEdit;
    spl.addWidget(ptxt1);
    spl.addWidget(ptxt2);

    ptxt1->setPlainText("Line1\n"
                        "Line2\n"
                        "Line3\n"
                        "Line4\n"
                        "Line5\n"
                        "Line6\n"
                        "Line7\n"
                       );
    ptxt2->setPlainText(ptxt1->toPlainText());

    spl.resize(200, 220);
    spl.show();

    return app.exec();
}
```

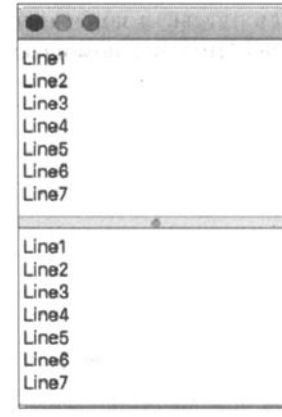


Рис. 6.13. Разделитель

В листинге 6.12 в конструктор класса `QSlitter` передается флаг `Qt::Vertical` и тем самым создается виджет вертикального разделителя.

> ***Создание горизонтального разделителя***
>
> Для создания горизонтального разделителя в конструктор нужно передать значение `QSplitter::Horizontal`. Того же эффекта можно добиться, если передавать флаги `QSplitter::Horizontal` и `QSplitter::Vertical` в метод `setOrientation()`.

После создания виджетов класса `QTextEdit` они добавляются в виджет разделителя, для чего их указатели передаются в метод `QSplitter::addWidget()`. Текст в виджетах `QTextEdit` устанавливается методом `setPlainText()`.

Резюме

Объекты компоновки виджетов при изменении размеров окна автоматически выполняют в нем размещение виджетов. Qt предоставляет менеджеры компоновки, которые обладают возможностью горизонтального, вертикального и табличного размещения. Эти способы можно комбинировать для создания сложных размещений. Классы менеджеров компоновки базируются сразу на двух классах: `QObject` и `QLayoutItem`. Объекты компоновки предоставляют возможность установки фактора растяжения для управления соотношением размеров виджетов. Кроме того, реализованы методы для установки расстояний между виджетами и настройки размера отступа виджетов от границы компоновки.

Виджет разделителя предоставляет возможность одновременного просмотра содержимого виджетов. При перетаскивании разделителя происходит изменение размеров виджетов, размещенных в разделителе.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/06-510/** или с помощью следующего QR-кода (рис. 6.14):



Рис. 6.14. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 7

Элементы отображения

> Видеть — значит верить, но только чувства отражают истину.
>
> *Томас Фуллер*

Элементы отображения не принимают активного участия в действиях пользователя и используются для информирования его о происходящем. Эта информация может носить как текстовый, так и графический характер (картинки, графика).

Надписи

Виджет надписи служит для показа состояния приложения или поясняющего текста и представляет собой текстовое поле, текст которого не подлежит изменению со стороны пользователя. Информация, отображаемая этим виджетом, может изменяться только самим приложением. Таким образом, приложение может сообщить пользователю о своем изменившемся состоянии, но пользователь не может изменить эту информацию в самом виджете. Класс виджета надписи `QLabel` определен в заголовочном файле QLabel.

Виджет надписи унаследован от класса `QFrame` и может иметь рамку. Отображаемая им информация может быть текстового, графического или анимационного характера, для передачи ее используются слоты `setText()`, `setPixmap()` и `setMovie()`.

Расположением текста можно управлять при помощи метода `setAlignment()`. Метод использует большое количество флагов, некоторые из них приведены в табл. 7.1. Обратите внимание, что значения не пересекаются, и это позволяет комбинировать их друг с другом с помощью логической операции `|` (ИЛИ). Наглядным примером служит значение `AlignCenter`, составленное из значений `AlignVCenter` и `AlignHCenter`.

Таблица 7.1. *Значения флагов* `AlignmentFlag` *пространства имен Qt*

Константа	Значение	Описание
`AlignLeft`	`0x0001`	Расположение текста слева
`AlignRight`	`0x0002`	Расположение текста справа
`AlignHCenter`	`0x0004`	Центровка текста по горизонтали
`AlignJustify`	`0x0008`	Растягивание текста по всей ширине
`AlignTop`	`0x0010`	Расположение текста вверху

Таблица 7.1 *(окончание)*

Константа	Значение	Описание
AlignBottom	0x0020	Расположение текста внизу
AlignVCenter	0x0040	Центровка текста по вертикали
AlignCenter	AlignVCenter | AlignHCenter	Центровка текста по вертикали и горизонтали

Как видно из рис. 7.1, виджет надписи может отображать не только обычный текст, но и текстовую информацию в формате HTML (HyperText Markup Language, язык гипертекстовой разметки). В этом примере (листинг 7.1) использовался HTML для вывода текста, таблицы и растрового изображения.

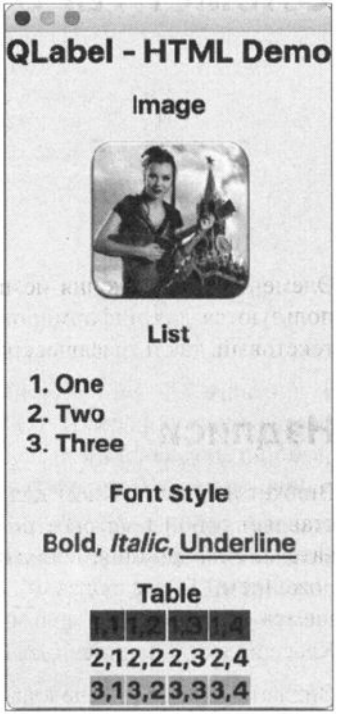


Рис. 7.1. Отображение виджетом надписи информации в формате HTML

Листинг 7.1. Создание виджета надписи с использованием формата HTML (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QLabel lbl("<H1><CENTER>QLabel - HTML Demo</CENTER></H1>"
               "<H2><CENTER>Image</CENTER><H2>"
               "<CENTER><IMG BORDER=\"0\" SRC=\":/Balalaika.png \"></CENTER>"
               "<H2><CENTER>List</CENTER><H2>"
               "<OL><LI>One</LI>"
               "    <LI>Two</LI>"
               "    <LI>Three</LI>"
               "</OL>"
```

```
                "<H2><CENTER>Font Style</CENTER><H2>"
                "<CENTER><FONT COLOR=RED>"
                "    <B>Bold</B>, <I>Italic</I>, <U>Underline</U>"
                "</FONT></CENTER>"
                "<H2><CENTER>Table</CENTER></H2>"
                "<CENTER> <TABLE>"
                "    <TR BGCOLOR=#ff00ff>"
                "        <TD>1,1</TD><TD>1,2</TD><TD>1,3</TD><TD>1,4</TD>"
                "    </TR>"
                "    <TR BGCOLOR=YELLOW>"
                "        <TD>2,1</TD><TD>2,2</TD><TD>2,3</TD><TD>2,4</TD>"
                "    </TR>"
                "    <TR BGCOLOR=#00f000>"
                "        <TD>3,1</TD><TD>3,2</TD><TD>3,3</TD><TD>3,4</TD>"
                "    </TR>"
                "</TABLE> </CENTER>"
            );
    lbl.show();

    return app.exec();
}
```

В листинге 7.1 при создании виджета надписи `lbl` первым параметром в конструктор передается текст в формате HTML. Его можно передать и после создания этого виджета при помощи метода-слота `setText()`. Второй параметр конструктора опущен, а так как по умолчанию он равен 0, то это делает его виджетом верхнего уровня.

Следующий пример (листинг 7.2), показанный на рис. 7.2, демонстрирует возможность отображения информации графического характера в виджете надписи без использования формата HTML.



Рис. 7.2. Отображение виджетом надписи графической информации

Листинг 7.2. Создание виджета надписи без использования формата HTML (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
```

```
    QPixmap pix;
    pix.load(":/mira.jpg");

    QLabel lbl;
    lbl.resize(pix.size());
    lbl.setPixmap(pix);

    lbl.show();

    return app.exec();
}
```

Как видно из листинга 7.2, сначала создается объект растрового изображения `QPixmap`. После этого вызовом метода `load()` в него загружается из ресурса файл mira.jpg. Следующим шагом является создание самого виджета надписи — объекта `lbl` класса `QLabel`. Затем вызовом метода `resize()` его размеры приводятся в соответствие с размерами растрового изображения. И, наконец, вызов метода `setPixmap()` устанавливает в виджете само растровое изображение.

При помощи метода `setBuddy()` виджет надписи может ассоциироваться с любым другим виджетом. Если текст надписи содержит знак `&` (амперсанд), то символ, перед которым он стоит, будет подчеркнутым. При нажатии клавиши этого символа совместно с клавишей <Alt> фокус перейдет к виджету, установленному методом `setBuddy()`.

Особенности для Mac OS X

По умолчанию в Mac OS X управление фокусом при помощи амперсанда не активно, и для его активации необходимо вызвать функцию `qt_set_sequence_auto_mnemonic()` со значением `true`.

На рис. 7.3 показаны такие виджеты, а в листинге 7.3 приведен текст соответствующей программы.

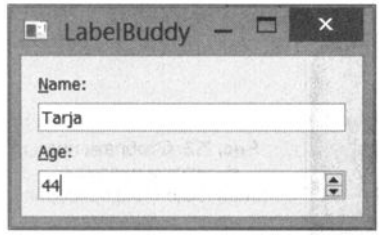


Рис. 7.3. Результат использования знака &: символы **N** и **A** — подчеркнуты

Листинг 7.3. Определение в тексте виджета символов быстрого доступа с помощью знака & (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget    wgt;
    QLabel*    plblName = new QLabel("&Name:");
```

```
    QLineEdit* ptxtName = new QLineEdit;
    plblName->setBuddy(ptxtName);

    QLabel*    plblAge = new QLabel("&Age:");
    QSpinBox*  pspbAge = new QSpinBox;
    plblAge->setBuddy(pspbAge);

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(plblName);
    pvbxLayout->addWidget(ptxtName);
    pvbxLayout->addWidget(plblAge);
    pvbxLayout->addWidget(pspbAge);
    wgt.setLayout(pvbxLayout);

    wgt.show();

    return app.exec();
}
```

В листинге 7.3 виджет `wgt` класса `QWidget` является виджетом верхнего уровня, так как по умолчанию его конструктор присваивает указателю на виджет-предок значение 0. Виджеты не обладают способностью самостоятельного размещения виджетов-потомков, поэтому позже в нем осуществляется установка компоновки для вертикального размещения `QVBoxLayout`. В виджете надписи **Name** (Имя) в тексте символ `N` определен как символ для быстрого доступа. Затем создается виджет однострочного текстового поля. Далее, вызов метода `setBuddy()` связывает виджет надписи с созданным текстовым полем, используя указатель на поле в качестве аргумента. Аналогично происходит создание виджета надписи **Age** (Возраст), поля для ввода возраста (класса `QSpinBox`) и их связывание.

Во всех виджетах есть возможность обработки событий клавиатуры и мыши. Этим можно воспользоваться, например, для создания гипертекстовой ссылки, которая при нажатии вызовет определенную HTML-страницу. Подробно события будут рассмотрены в *части III*.

Однако можно поступить и проще. Дело в том, что класс `QLabel` предоставляет поддержку для гипертекстовых ссылок и при нажатии на ссылку отправляет сигнал `linkActivated()`, который можно соединить со слотом, из которого произойдет вызов страницы.

Но есть и другой, еще более простой способ, он заключается в том, чтобы перевести виджет `QLabel` в состояние, когда он сам сможет открывать ссылки в веб-браузере (что будет достигаться неявным вызовом статического метода `QDesktopServices::openUrl()`). Для этого нужно просто вызвать метод `setOpenExternalLinks()` с параметром `true`. Например:

```
QLabel* plbl =
      new QLabel("<A HREF=\"http://qt-book.com\"> qt-book.com </A>");
plbl->setOpenExternalLinks(true);
```

Индикатор выполнения

Индикатор выполнения (progress bar) — это виджет, демонстрирующий ход процесса выполнения операции и заполняющийся слева направо. Полное заполнение индикатора информирует о завершении операции. Этот виджет необходим в том случае, когда программа

выполняет продолжительные действия, — виджет дает пользователю понять, что программа не зависла, а находится в работе. Он также показывает, сколько уже проделано и сколько еще предстоит сделать. Класс `QProgessBar` виджета индикатора выполнения определен в заголовочном файле QProgressBar. Обычно индикаторы выполнения располагаются в горизонтальном положении, но это можно изменить, передав в слот `setOrientation()` значение `Qt::Vertical`, — после этого он будет расположен вертикально.

Следующий пример демонстрирует использование индикатора выполнения. При нажатии кнопки **Step** (Шаг) выполняется увеличение значения индикатора на один шаг. Нажатие кнопки **Reset** (Сбросить) сбрасывает значение индикатора. В основной программе, приведенной в листинге 7.4, создается виджет, показанный на рис. 7.4.

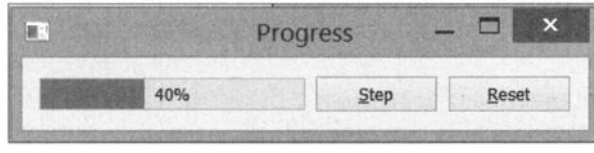


Рис. 7.4. Индикатор выполнения

Листинг 7.4. Создание индикатора выполнения (файл main.cpp)

```
#include <QApplication>
#include "Progress.h"

int main (int argc, char** argv)
{
    QApplication app(argc, argv);
    Progress     progress;

    progress.show();

    return app.exec();
}
```

В листинге 7.5 приведен файл Progress.h, который содержит определение класса `Progress`, унаследованного от `QWidget`. Класс содержит два атрибута: указатель на виджет индикатора выполнения и целое значение, представляющее номер шага. В классе определены два слота: `slotStep()` и `slotReset()`. Первый предназначен для наращивания шага на единицу, а второй — для установки индикатора выполнения в нулевое положение.

Листинг 7.5. Определение класса `Progress` (файл Progress.h)

```
#pragma once

#include <QWidget>

class QProgressBar;
```

```
// ============================================================================
class Progress : public QWidget {
    Q_OBJECT
private:
    QProgressBar* m_pprb;
    int           m_nStep;

public:
    Progress(QWidget* pobj = 0);

public slots:
    void slotStep ();
    void slotReset();
};
```

В конструкторе класса (листинг 7.6) атрибуту `m_nStep` присваивается значение 0. После создания объекта индикатора `m_pprb` вызовом метода `setRange()` задается количество шагов, равное 5, а метод `setMinimumWidth()` устанавливает минимальную длину виджета индикации выполнения, — в нашем случае мы запрещаем ему иметь длину менее двухсот пикселов. Вызов метода `setAlignment()` с параметром `Qt::AlignCenter` переводит индикатор в режим отображения процентов в центре (см. табл. 7.1). Затем создаются две кнопки: **Step** (Шаг) и **Reset** (Сбросить), которые соединяются со слотами `slotStep()` и `slotReset()`. В слоте `slotStep()` значение атрибута `m_nStep` увеличивается на 1 и передается в слот `QProgressBat::setValue()` объекта индикатора выполнения. Слот `slotReset()` устанавливает значение атрибута `m_nStep` равным 0 и, вызвав слот `QProgressBar::reset()`, возвращает индикатор в исходное состояние. Для размещения виджетов-потомков горизонтально и слева направо необходимо установить в виджете `Progress` объект класса компоновки `QVBoxLayout`, предварительно добавив в него, в нужной очередности, виджеты-потомки.

Листинг 7.6. Конструктор класса `Progress` (файл Progress.cpp)

```
#include <QtWidgets>
#include "Progress.h"

// ----------------------------------------------------------------------------
Progress::Progress(QWidget* pwgt/*= 0*/)
    : QWidget(pwgt)
    , m_nStep(0)
{
    m_pprb = new QProgressBar;
    m_pprb->setRange(0, 5);
    m_pprb->setMinimumWidth(200);
    m_pprb->setAlignment(Qt::AlignCenter);

    QPushButton* pcmdStep  = new QPushButton("&Step");
    QPushButton* pcmdReset = new QPushButton("&Reset");

    QObject::connect(pcmdStep, SIGNAL(clicked()), SLOT(slotStep()));
    QObject::connect(pcmdReset, SIGNAL(clicked()), SLOT(slotReset()));
```

```
    //Layout setup
    QHBoxLayout* phbxLayout = new QHBoxLayout;
    phbxLayout->addWidget(m_pprb);
    phbxLayout->addWidget(pcmdStep);
    phbxLayout->addWidget(pcmdReset);
    setLayout(phbxLayout);
}

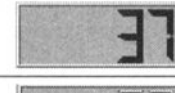
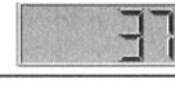
// -----------------------------------------------------------------------
void Progress::slotStep()
{
    m_pprb->setValue(++m_nStep);
}
// -----------------------------------------------------------------------
void Progress::slotReset()
{
    m_nStep = 0;
    m_pprb->reset();
}
```

Электронный индикатор

Класс `QLCDNumber` виджета электронного индикатора определен в заголовочном файле QLCDNumber. По внешнему виду этот виджет представляет собой набор сегментных указателей, как, например, на электронных часах. С помощью виджета электронного индикатора отображаются целые числа. Допускается использование точки, которую можно отображать между позициями сегментов или как отдельный символ, вызывая метод `setSmallDecimalPoint()` и передавая в него `true` или `false` соответственно. Количество отображаемых сегментов можно задать в конструкторе или с помощью метода `setNumDigits()`. В том случае, когда для отображения числа не хватает сегментов индикатора, отсылается сигнал `overflow()`.

По умолчанию стиль электронного индикатора соответствует стилю `Outline`, но его можно изменить, передав методу `setSegmentStyle()` одно из следующих значений: `QLCDNumber::Outline`, `QLCDNumber::Filled` или `QLCDNumber::Flat`. В табл. 7.2 показан внешний вид виджета для каждого из перечисленных стилей.

Таблица 7.2. *Стили электронного индикатора*

Константа	Внешний вид
`Outline`	
`Flat`	
`Filled`	

Электронный индикатор можно включать в режиме отображения двоичной, восьмеричной, десятеричной или шестнадцатеричной систем счисления. Режим отображения изменяется с помощью метода `setMode()`, в который передается одно из следующих значений: `QLCDNumber::Bin` (двоичная), `QLCDNumber::Oct` (восьмеричная), `QLCDNumber::Dec` (десятеричная) или `QLCDNumber::Hex` (шестнадцатеричная). Также для смены режима отображения можно воспользоваться слотами `setBinMode()`, `setOctMode()`, `setDecMode()` и `setHexMode()` соответственно.

Следующий пример (листинг 7.7) демонстрирует электронный индикатор, отображающий шестнадцатеричные значения (рис. 7.5).

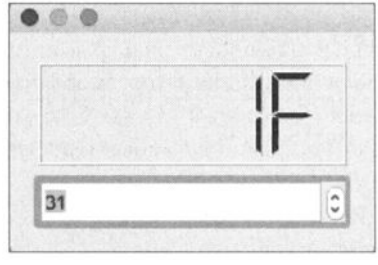


Рис. 7.5. Электронный индикатор

В листинге 7.7 создаются виджеты электронного индикатора (указатель `plcd`) и счетчика (указатель `pspb`). Затем вызовом метода `setSegmentStyle()` изменяется стиль сегментных указателей. Электронный индикатор вызовом метода `setMode()` с параметром `QLCDNumber::Hex` переключается в режим шестнадцатеричного отображения. У элемента счетчика методом `setFixedHeight()` устанавливается неизменная высота, равная 30. После этого сигнал `valueChanged()` виджета счетчика соединяется со слотом `display()` электронного индикатора

Листинг 7.7. Создание электронного индикатора (файл main.cpp)

```
#include <QtWidgets>

int main (int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget     wgt;
    QLCDNumber* plcd = new QLCDNumber;
    QSpinBox*   pspb = new QSpinBox;

    plcd->setSegmentStyle(QLCDNumber::Filled);
    plcd->setMode(QLCDNumber::Hex);

    pspb->setFixedHeight(30);

    QObject::connect(pspb, SIGNAL(valueChanged(int)),
                     plcd, SLOT(display(int))
                    );

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
```

```
    pvbxLayout->addWidget(plcd);
    pvbxLayout->addWidget(pspb);
    wgt.setLayout(pvbxLayout);

    wgt.resize(250, 150);
    wgt.show();

    return app.exec();
}
```

Резюме

В этой главе мы познакомились с элементами отображения. Виджет надписи содержит информацию, предназначенную только для просмотра и не подлежащую изменению со стороны пользователя. Этот виджет способен отображать не только простой текст, но и текст в формате HTML. Кроме текстовой может отображаться и графическая информация.

Виджет индикатора выполнения — прекрасный элемент для оповещения пользователя о режиме работы приложения. Он незаменим для иллюстрации процесса выполнения продолжительных операций.

Виджет электронного индикатора по внешнему виду представляет собой набор сегментных указателей, как на электронных часах. Стиль виджета можно изменять и использовать различные режимы отображения чисел: в двоичной, десятеричной, шестнадцатеричной и восьмеричной системах счисления.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/07-510/** или с помощью следующего QR-кода (рис. 7.6):



Рис. 7.6. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 8

Кнопки, флажки и переключатели

> Нажми на кнопку, получишь результат, и твоя мечта осуществится.
>
> *Группа «Технология»*

Кнопки — это один из важнейших и наиболее часто встречаемых элементов пользовательского интерфейса. Даже если программистом в коде программы не создается ни одной, все равно в правом верхнем углу окна приложения присутствуют кнопки, этим окном управляющие. Кнопка может быть нажата (on) или отжата (off).

С чего начинаются кнопки? Класс *QAbstractButton*

Класс `QAbstractButton` — базовый для всех кнопок. В приложениях применяются три основных вида кнопок: нажимающиеся кнопки (`QPushButton`), которые обычно называют просто кнопками, флажки (`QCheckBox`) и переключатели (`QRadioButton`). В классе `QAbstractButton` реализованы методы и возможности, присущие всем кнопкам. Сначала мы обсудим основные из этих возможностей, а потом поговорим о каждом виде в отдельности.

Установка текста и изображения

Все кнопки могут содержать текст, который можно передать как в конструкторе первым параметром, так и установить с помощью метода `setText()`. Для получения текста в классе `QAbstractButton` определен метод `text()`.

Растровое изображение устанавливается на кнопке при помощи метода `setIcon()`. После установки изображения вызовом метода `setIconSize()` можно изменить его максимальный размер, который занимает изображение на кнопке (изображения меньшего размера не растягиваются). Для получения текущего максимального размера изображения определен метод `iconSize()`. И наконец, для того чтобы кнопка возвратила установленное в ней изображение, нужно вызвать метод `icon()`.

Взаимодействие с пользователем

Для взаимодействия с пользователем класс `QAbstractButton` предоставляет следующие сигналы:

- `clicked()` — отправляется при щелчке кнопкой мыши;
- `pressed()` — отправляется при нажатии на кнопку мыши;

- `released()` — отправляется при отпускании кнопки мыши;
- `toggled()` — отправляется при изменении состояния кнопки, имеющей статус выключателя.

Опрос состояния

Для опроса текущего состояния кнопок в классе `QAbstractButton` определены три метода:

- `isDown()` — возвращает значение `true`, если кнопка находится в нажатом состоянии. Изменить текущее состояние может либо пользователь, нажав на кнопку, либо вызов метода `setDown()`;
- `isChecked()` — возвращает значение `true`, когда кнопка находится во включенном состоянии. Изменить текущее состояние может либо пользователь, нажав на кнопку, либо вызов метода `setChecked()`;
- кнопка доступна, то есть реагирует на действия пользователя, если метод `isEnabled()` возвращает значение `true`. Изменить текущее состояние можно вызовом метода `setEnabled()`.

Кнопки

Виджет кнопки можно встретить в любом приложении, — например, почти всегда там имеются кнопки **Ok** или **Cancel** (Отмена) — без них не обходится ни одно диалоговое окно. Иногда такой виджет называют «командной кнопкой». Он представляет собой прямоугольный элемент управления и используется, как правило, для выполнения определенной операции при нажатии на него. Класс `QPushButton` виджета нажимающейся кнопки определен в заголовочном файле QPushButton.

Создать нажимающуюся кнопку можно следующим образом:

```
QPushButton* pcmd = new QPushButton("My Button");
```

Первый параметр (типа «строка») задает надпись кнопки. По обыкновению, в конструктор можно передавать также и виджет предка, но так делать смысла нет, поскольку менеджер сделает это за нас.

Примеры, показанные на рис. 8.1, демонстрируют различные варианты нажимающихся кнопок:

- **Normal Button** (Обычная кнопка) — соответствует той самой кнопке, которую мы привыкли видеть в большинстве случаев. После отпускания кнопка всегда возвращается в свое исходное положение;
- **Toggle Button** (Выключатель) — может пребывать в двух состояниях: нажатом или не нажатом, которые соответствуют положениям «включено» или «выключено». Логика действия этой кнопки идентична, например, логике обычного комнатного электровыключателя;
- **Flat Button** (Плоская кнопка) — по своим функциональным особенностям идентична обычной кнопке. Разница лишь во внешнем виде. Например, благодаря тому, что контуры этой кнопки не видны, ею можно воспользоваться для размещения «секретной кнопки» диалогового окна;
- наконец, последняя кнопка **Pixmap Button** (Кнопка с изображением) — представляет собой кнопку, содержащую растровое изображение.

В листинге 8.1 приводится текст приложения, окно которого как раз и показано на рис. 8.1. Сначала создается виджет обычной кнопки (указатель `pcmdNormal`). Кнопка-выключатель (указатель `pcmdToggle`) создается так же, как и обычная кнопка, но для нее вызовом метода `setCheckable()` с параметром `true` устанавливается режим выключателя. Вызов метода `setChecked()` с параметром `true` приводит эту кнопку во включенное состояние.



Рис. 8.1. Нажимающиеся кнопки

Затем создается виджет плоской кнопки (указатель `pcmdFlat`) как обычной кнопки. Вызовом метода `setFlat()` с параметром `true` ей придается плоский вид.

Для создания кнопки с растровым изображением сначала создается объект растрового изображения `pix`, в который из ресурсов загружается файл ChordsMaestro.png, указанный в конструкторе. Затем создается кнопка (указатель `pcmdPix`), после чего объект растрового изображения передается в метод `setIcon()` для установки на кнопке. Слот `setIconSize()` задает размеры растрового изображения, в данном случае они соответствуют оригинальным размерам изображения, которые возвращает метод `size()` объекта `QPixmap`.

Листинг 8.1. Создание кнопок (файл main.cpp)

```
#include <QtWidgets>

int main (int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget      wgt;
    QPushButton* pcmdNormal = new QPushButton("&Normal Button");

    QPushButton* pcmdToggle = new QPushButton("&Toggle Button");
    pcmdToggle->setCheckable(true);
    pcmdToggle->setChecked(true);

    QPushButton* pcmdFlat = new QPushButton("&Flat Button");
    pcmdFlat->setFlat(true);

    QPixmap pix(":/ChordsMaestro.png");
    QPushButton* pcmdPix = new QPushButton("&Pixmap Button");
    pcmdPix->setIcon(pix);
    pcmdPix->setIconSize(pix.size());

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(pcmdNormal);
    pvbxLayout->addWidget(pcmdToggle);
    pvbxLayout->addWidget(pcmdFlat);
```

```
    pvbxLayout->addWidget(pcmdPix);
    wgt.setLayout(pvbxLayout);

    wgt.show();

    return app.exec();
}
```

Существует возможность использования в нажимающихся кнопках всплывающего меню (рис. 8.2). Подобные кнопки можно встретить, например, в обозревателе Microsoft Internet Explorer. Кнопка **Start** (Пуск) панели задач ОС Windows 7 также представляет собой такую кнопку. Добавить меню можно, вызвав метод `setMenu()` и передав указатель на объект всплывающего меню. Такие кнопки могут использоваться в качестве альтернативы для выпадающего списка (см. *главу 11*).

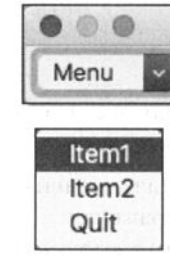


Рис. 8.2. Нажимающаяся кнопка со всплывающим меню

В листинге 8.2 создаются виджеты кнопки `cmd` и всплывающего меню `pmnu`. Вызовом метода `addAction()` добавляется элемент меню (см. *главу 31*). Последняя команда **Quit** (Выход) соединяется со слотом `quit()` объекта приложения, что позволяет пользователю завершить приложение, выбрав эту команду из меню. Установка созданного меню в нажимающейся кнопке осуществляется вызовом метода `setMenu()`.

Листинг 8.2. Создание кнопки со всплывающим меню (файл main.cpp)

```
#include <QtWidgets>

int main (int argc, char** argv)
{
    QApplication app(argc, argv);

    QPushButton  cmd("Menu");
    QMenu*       pmnu = new QMenu(&cmd);
    pmnu->addAction("Item1");
    pmnu->addAction("Item2");
    pmnu->addAction("&Quit", &app, SLOT(quit()));

    cmd.setMenu(pmnu);
    cmd.show();

    return app.exec();
}
```

Флажки

Большинство программ предоставляют наборы настроек, дающих возможность изменять поведение программы. Для этих целей может пригодиться виджет флажка, который позволяет пользователю выбирать сразу несколько опций. Класс `QCheckBox` виджета кнопки флажка определен в заголовочном файле QCheckBox.

Используйте виджет списка для большого количества опций

Если опций больше пяти, лучше использовать виджет списка `QListWidget` (см. *главы 11* и *12*).

Флажок представляет собой маленький прямоугольник и может содержать поясняющий текст или картинку. При щелчке на виджете в прямоугольнике появится отметка. Этого же можно добиться нажатием клавиши <Пробел>, когда виджет находится в фокусе. Виджет флажка устанавливается в положение «включено» или «выключено» и является, по логике действия, кнопкой-выключателем (toggle button). Но, в отличие от последней, флажок может иметь еще и третье состояние — неопределенное (рис. 8.3). Пример использования такого состояния можно увидеть в диалоговом окне **Properties** (Свойства) Проводника в ОС Windows при выборе нескольких файлов, имеющих разные атрибуты.

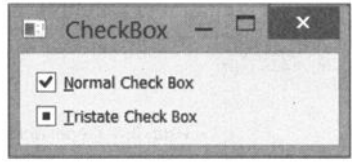


Рис. 8.3. Флажки

В листинге 8.3 создаются два флажка: указатели `pchkNormal` и `pchkTristate`. Флажок **Normal Check Box** (Обычный флажок) помечается вызовом метода `setChecked()` с параметром `true`. Флажок **Tristate Check Box** (Флажок с неопределенным состоянием) переводится в режим поддержки третьего, неопределенного состояния передачей значения `true` в метод `setTristate()`. Затем вызовом метода `setCheckState()` и передачей в него значения `Qt::PartiallyChecked` устанавливается третье состояние.

Листинг 8.3. Создание флажков (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget    wgt;
    QCheckBox* pchkNormal = new QCheckBox("&Normal Check Box");
    pchkNormal->setChecked(true);

    QCheckBox* pchkTristate = new QCheckBox("&Tristate Check Box");
    pchkTristate->setTristate(true);
    pchkTristate->setCheckState(Qt::PartiallyChecked);

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(pchkNormal);
    pvbxLayout->addWidget(pchkTristate);
    wgt.setLayout(pvbxLayout);

    wgt.show();

    return app.exec();
}
```

Переключатели

Свое английское название — radiobutton — виджет переключателя получил благодаря схожести с кнопками переключения диапазонов радиоприемника, на панели которого может быть нажата только одна из таких кнопок. Нажатие на кнопку другого диапазона приводит к тому, что автоматически отключается кнопка диапазона, нажатая до этого.

Переключатель представляет собой виджет (рис. 8.4), который должен находиться в одном из двух состояний: включено (on) или выключено (off). Эти состояния пользователь может устанавливать с помощью мыши или клавиши <Пробел>, когда кнопка находится в фокусе. Класс `QRadioButton` виджета переключателя определен в заголовочном файле QRadioButton.

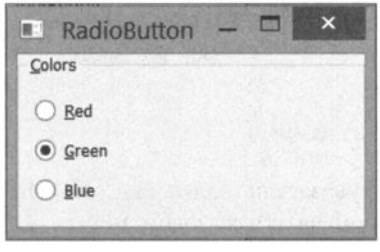


Рис. 8.4. Переключатели

Виджеты переключателей должны предоставлять пользователю, по меньшей мере, выбор одной из двух альтернатив, не могут использоваться в отдельности и должны быть сгруппированы вместе. Их группировку можно выполнить, например, при помощи класса `QGroupBox`.

Поясняющие надписи должны быть определены для каждого используемого в группе переключателя, желательно также задать и сочетания клавиш для быстрого доступа к каждому из переключателей. Это достигается включением в надпись символа `&` перед нужной буквой.

> ***Используйте виджет выпадающего списка для большого количества опций***
>
> Преимущество переключателей состоит в том, что все опции видны сразу, но они занимают много места. Поэтому если количество переключателей больше пяти, то лучше воспользоваться виджетом выпадающего списка `QComboBox` (см. *главу 11*).

В листинге 8.4 создается виджет для группы переключателей `gbx`. После создания переключателей — `pradRed`, `pradGreen` и `pradBlue` — один из них (`pradGreen`) устанавливается во включенное состояние вызовом метода `setChecked()` с параметром `true`. Переключатели размещаются на поверхности виджета группы `gbx`, а объект класса `QVBoxLayout` автоматически выстраивает их в вертикальном порядке (см. рис. 8.4).

Листинг 8.4. Создание виджета для группы переключателей (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
```

```
    QGroupBox gbx("&Colors");

    QRadioButton* pradRed   = new QRadioButton("&Red");
    QRadioButton* pradGreen = new QRadioButton("&Green");
    QRadioButton* pradBlue  = new QRadioButton("&Blue");
    pradGreen->setChecked(true);

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(pradRed);
    pvbxLayout->addWidget(pradGreen);
    pvbxLayout->addWidget(pradBlue);
    gbx.setLayout(pvbxLayout);

    gbx.show();

    return app.exec();
}
```

Группировка кнопок

Виджеты группировки кнопок, в основном, не предназначены для взаимодействия с пользователем. Их основная задача — повысить удобство использования и упростить понимание программы. Для этого важно, чтобы элементы интерфейса были объединены в отдельные логические группы. Кроме того, нужно помнить, что кнопки переключателей `QRadioButton` не могут использоваться отдельно друг от друга и должны быть объединены вместе.

Класс `QGroupBox` является классом для такой группировки и представляет собой контейнер, содержащий в себе различные элементы управления. Он может иметь поясняющую надпись в верхней области, и эта надпись может содержать клавишу быстрого доступа, при нажатии на которую фокус переводится на саму группу. Такой контейнер также можно снабдить дополнительным флажком, который будет управлять доступностью сгруппированных элементов. Класс `QGroupBox` определен в заголовочном файле QGroupBox.

Следующий пример создает группу переключателей, отвечающих за цвет фона (рис. 8.5). Например, выбор переключателя **Red** (Красный) приведет к тому, что фон виджета верхнего уровня станет красным. Флажок **Light** (Яркость) управляет яркостью цвета, а кнопка **Exit** (Выход) осуществляет выход из программы. Текст соответствующего файла main.cpp приведен в листинге 8.5.

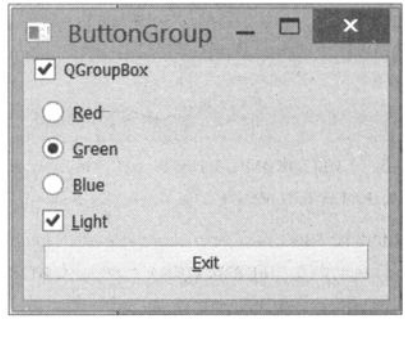


Рис. 8.5. Группировка переключателей

Листинг 8.5. Создание группы переключателей (файл main.cpp)

```
#include <QApplication>
#include "Buttons.h"

int main (int argc, char** argv)
{
    QApplication app(argc, argv);

    Buttons buttons;
    buttons.show();

    return app.exec();
}
```

Как видно из листинга 8.6, в котором определяется класс `Buttons`, он наследуется от класса `QGroupBox`. В классе определяются атрибуты `m_pchk` для флажка и `m_pradRed`, `m_pradGreen`, `m_pradBlue` для переключателей. Это необходимо, чтобы указанные атрибуты были доступны из слота `slotButtonClicked()`.

Листинг 8.6. Определение класса `Buttons` (файл Buttons.h)

```
#pragma once

#include <QGroupBox>

class QCheckBox;
class QRadioButton;

// ============================================================================
class Buttons : public QGroupBox {
    Q_OBJECT
private:
    QCheckBox*    m_pchk;
    QRadioButton* m_pradRed;
    QRadioButton* m_pradGreen;
    QRadioButton* m_pradBlue;

public:
    Buttons(QWidget* pwgt = 0);

public slots:
    void slotButtonClicked();
};
```

В конструкторе класса `Buttons` (листинг 8.7) вызовом метода `setCheckable()` с параметром `true` устанавливается флажок, который включается методом `setChecked()`.

После создания трех переключателей (указатели `pradRed`, `pradGreen` и `pradBlue`) первый из них выделяется вызовом метода `setChecked()` с параметром `true`. Сигнал `clicked()` для каждого переключателя соединяется со слотом `slotButtonClicked()`.

Флажок **Light** (Яркость) (указатель `m_pchk`) включается сразу после своего создания при помощи метода `setChecked()`.

Последней из кнопок создается кнопка **Exit** (Выход) (указатель `pcmd`). Для выхода из приложения при нажатии на эту кнопку сигнал `clicked()` соединяется со слотом `quit()` объекта приложения.

Затем созданные виджеты размещаются в вертикальном порядке при помощи объекта класса `QVBoxLayout`.

В последней строке вызывается слот `slotButtonClicked()` для инициализации.

Листинг 8.7. Конструктор класса Buttons (файл Buttons.cpp)

```
Buttons::Buttons(QWidget* pwgt/*= 0*/) : QGroupBox("QGroupBox", pwgt)
{
    resize(100, 150);
    setCheckable(true);
    setChecked(true);

    m_pradRed   = new QRadioButton("&Red");
    m_pradGreen = new QRadioButton("&Green");
    m_pradBlue  = new QRadioButton("&Blue");
    m_pradGreen->setChecked(true);
    connect(m_pradRed, SIGNAL(clicked()), SLOT(slotButtonClicked()));
    connect(m_pradGreen, SIGNAL(clicked()), SLOT(slotButtonClicked()));
    connect(m_pradBlue, SIGNAL(clicked()), SLOT(slotButtonClicked()));

    m_pchk = new QCheckBox("&Light");
    m_pchk->setChecked(true);
    connect(m_pchk, SIGNAL(clicked()), SLOT(slotButtonClicked()));

    QPushButton* pcmd = new QPushButton("&Exit");
    connect(pcmd, SIGNAL(clicked()), qApp, SLOT(quit()));

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(m_pradRed);
    pvbxLayout->addWidget(m_pradGreen);
    pvbxLayout->addWidget(m_pradBlue);
    pvbxLayout->addWidget(m_pchk);
    pvbxLayout->addWidget(pcmd);
    setLayout(pvbxLayout);

    slotButtonClicked();
}
```

В листинге 8.8 приводится реализация метода `slotButtonClicked()`, в котором при каждом вызове создается объект палитры и проверяется состояние флажка **Light** (Яркость) для установки переменной `nLight`, управляющей яркостью цвета. В операторах `if` выполняется анализ состояния кнопок-переключателей при помощи метода `isChecked()`. В зависимости от помеченной кнопки переключателя, цвет фона палитры `QWidget::backgroundRole()` из-

меняется вызовом метода `QPalette::setColor()` и устанавливается в виджете при помощи метода `QWidget::setPalette()`. Подробнее о палитре можно прочитать в *главе 13*.

Листинг 8.8. Метод `slotButtonClicked()` (файл Buttons.cpp)

```
void Buttons::slotButtonClicked()
{
    QPalette pal    = palette();
    int      nLight = m_pchk->isChecked() ? 150 : 80;
    if(m_pradRed->isChecked()) {
        pal.setColor(backgroundRole(), QColor(Qt::red).light(nLight));
    }
    else if(m_pradGreen->isChecked()) {
        pal.setColor(backgroundRole(), QColor(Qt::green).light(nLight));
    }
    else {
        pal.setColor(backgroundRole(), QColor(Qt::blue).light(nLight));
    }
    setPalette(pal);
}
```

Резюме

Существуют три основных виджета, унаследованных от класса `QAbstractButton`: кнопки, флажки и переключатели.

Кнопки служат для выполнения определенных действий. При нажатии на кнопку отправляется сигнал `pressed()`, после отпускания — сигнал `released()`. Чаще всего используется сигнал `clicked()`, который отправляется, если пользователь нажал и отпустил кнопку.

Флажки часто применяются в диалоговых окнах, содержащих опции. Группа флажков служит для одновременного выбора нескольких опций. Возможен вариант, когда не будет выбран ни один из них.

Переключатели используются только в группе, в которой одновременно можно выбрать лишь один из переключателей. Тем самым при помощи этой группы моделируется отношение «один-ко-многим».

Основная задача виджета группировки — облегчить восприятие и работу с программой. С его помощью элементы интерфейса объединяются по принадлежности в отдельные логические группы.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/08-510/** или с помощью следующего QR-кода (рис. 8.6):



Рис. 8.6. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 9

Элементы настройки

> Происходящим необходимо управлять — иначе оно будет управлять вами.
>
> *Бак Роджерс, вице-президент IBM*

Группа виджетов, относимых к элементам настройки, используется, как правило, для установки значений, не требующих большой точности, — например, настройки громкости звука, скорости движения курсора (указателя) мыши, скроллинга содержимого окна и других подобных действий.

Класс *QAbstractSlider*

Этот класс является базовым для всех виджетов настройки: ползунка (`QSlider`), полосы прокрутки (`QScrollBar`) и установщика (`QDial`) (см. рис. 5.1). Все описанные далее возможности также доступны и во всех унаследованных от него классах. Его определение содержится в заголовочном файле QAbstractSlider.

Если требуется создать свой собственный виджет, то можно унаследовать этот класс и реализовать метод `sliderChange()`, который вызывается всякий раз при изменении значения.

Изменение положения

Классы виджетов, унаследованные от класса `QAbstractSlider`, могут быть как горизонтальными, так и вертикальными. Для изменения расположения используется слот `setOrientation()`, в который для задания горизонтального расположения передается значение `Qt::Horizontal`, а для вертикального — `Qt::Vertical`.

Установка диапазона

Для установки диапазона значений используется метод `setRange()`. В этот метод первым параметром передается минимально возможное значение (его нижняя граница), а вторым — задается его максимально возможное значение (верхняя граница). Также можно воспользоваться методами `setMinimum()` и `setMaximum()` соответственно. Например, для того чтобы задать диапазон от 1 до 10, можно поступить следующим образом:

```
psld->setRange(1, 10);
```

или так:

```
psld->setMinimum(1);
psld->setMaximum(10);
```

Установка шага

При помощи метода `setSingleStep()` можно задать *шаг*, то есть значение, на которое, например, ползунок сдвинется при нажатии на стрелки полосы прокрутки или на клавиши управления курсором с клавиатуры.

Метод `setPageStep()` задает шаг для страницы. Перемещение страниц выполняется, например, для элемента ползунка при нажатии на область, находящуюся между стрелками и головкой ползунка, или клавишами <Page Up>, <Page Down>.

Установка и получение значений

Для того чтобы установить какое-либо значение, необходимо воспользоваться слотом `setValue()`. Для получения текущего значения можно вызвать метод `value()`.

Сигнал `sliderMoved(int)` передает актуальное значение положения и отправляется при изменении пользователем указателя текущего положения.

Сигнал `valueChanged()` посылается одновременно с сигналом `sliderMoved(int)` сразу после изменения положения ползунка и также передает измененное значение полосы прокрутки. Поведение сигнала изменяется вызовом метода `setTracking()`. Если передать ему значение `false`, это приведет к тому, что сигнал `valueChanged()` будет отправляться только при отпускании указателя текущего положения.

Чтобы узнать, отпустил ли пользователь указатель текущего положения ползунка или все еще удерживает его, можно присоединиться к сигналам `sliderPressed()` или `sliderReleased()`.

Ползунок

Ползунок позволяет довольно комфортно выполнять настройки некоторых параметров приложения. Класс `QSlider` ползунка определен в заголовочном файле QSlider.

Класс `QSlider` содержит метод, управляющий размещением рисок (шкалы) ползунка. Риски очень важны при отображении ползунка. Они дают пользователю визуально более четкое представление о его местонахождении и показывают шаг. Возможные значения, которые можно передать в метод `setTickPosition()`, приведены в табл. 9.1.

Таблица 9.1. *Значения перечисления* `TickPosition` *класса* `QSlider`

Константа	Описание	Вид
`NoTicks`	Ползунок без рисок	
`TicksAbove`	Отображение рисок на верхней стороне ползунка	
`TicksBelow`	Отображение рисок на нижней стороне ползунка	
`TicksBothSides`	Отображение рисок на верхней и нижней сторонах ползунка	

Метод `setTickInterval()` задает шаг рисования рисок. Не следует задавать большое количество рисок, так как это приведет к появлению сплошной серой линии и не принесет никакой пользы.

Следующий пример (листинг 9.1) демонстрирует создание ползунка. Окно приложения, изображенное на рис. 9.1, содержит виджеты ползунка и надписи, причем текст надписи изменяется в зависимости от положения ползунка.

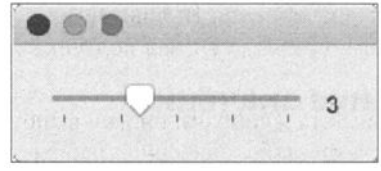


Рис. 9.1. Окно приложения, демонстрирующее работу ползунка

Листинг 9.1. Создание ползунка (файл main.cpp)

```
#include <QtWidgets>

int main (int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget  wgt;
    QSlider* psld = new QSlider(Qt::Horizontal);
    QLabel*  plbl = new QLabel("3");

    psld->setRange(0, 9);
    psld->setPageStep(2);
    psld->setValue(3);
    psld->setTickInterval(2);
    psld->setTickPosition(QSlider::TicksBelow);
    QObject::connect(psld, SIGNAL(valueChanged(int)),
                     plbl, SLOT(setNum(int))
                    );

    //Layout setup
    QHBoxLayout* phbxLayout = new QHBoxLayout;
    phbxLayout->addWidget(psld);
    phbxLayout->addWidget(plbl);
    wgt.setLayout(phbxLayout);

    wgt.show();

    return app.exec();
}
```

В листинге 9.1 создаются виджеты ползунка (указатель `psld`) и надписи (указатель `plbl`). После этого вызовом метода `setRange()` осуществляется установка диапазона значений ползунка от 0 до 9. Шаг страницы устанавливается равным 2 методом `setPageStep()`.

При помощи метода `setValue()` можно задавать стартовое значение, используемое для синхронизации с другими элементами управления, работающими вместе с ползунком. С его помощью можно сделать так, чтобы величины виджетов совпадали друг с другом, он также может служить просто для задания начального значения при первом показе элемента. Здесь в метод ползунка `setValue()` передается значение 3, синхронизирующее его со значением, отображаемым при создании надписи.

Шаг для рисования рисок устанавливается равным двум, для чего в метод ползунка `setTickInterval()` передается значение 2. Вызов метода `setTickmarks()` осуществляет установку рисок снизу. Методом `connect()` сигнал ползунка `valueChanged(int)` соединяется со слотом надписи `setNum(int)`.

В завершение при помощи горизонтальной компоновки выполняется размещение элементов на поверхности виджета `wgt`.

Полоса прокрутки

Полоса прокрутки — это важная составляющая практически любого пользовательского интерфейса. Она интуитивно воспринимается пользователем, и с ее помощью отображаются текстовые или графические данные, по размерам превышающие отведенную для них область. Используя указатель текущего положения полосы прокрутки, можно перемещать данные в видимую область. Этот указатель показывает относительную позицию видимой части объекта, благодаря которой можно получить представление о размере самих данных. Класс `QScrollBar` является реализацией виджета полосы прокрутки. Он определен в заголовочном файле QscrollBar и не содержит никаких дополнительных методов и сигналов, расширяющих определения класса `QAbstractSlider`.

Отдельно полосы прокрутки используются очень редко. Они встроены в виджет `QAbstractScrollArea`. Поэтому если вы намерены воспользоваться классом полосы прокрутки `QScrollBar`, то не исключено, что лучшим вариантом может оказаться использование одного из виджетов, наследующих базовый класс для видовой прокрутки `QAbstractScrollArea`.

Виджет полосы прокрутки имеет минимальное и максимальное значения, текущее значение и ориентацию. Перемещение указателя текущего положения осуществляется с помощью левой кнопки мыши. В качестве альтернативы можно просто нажать на кнопки стрелок, расположенных на концах полосы прокрутки.

Окно приложения (листинг 9.2), показанное на рис. 9.2, состоит из электронного индикатора и полосы прокрутки. Значение, отображаемое электронным индикатором, изменяется в зависимости от положения указателя текущего положения.



Рис. 9.2. Окно приложения, демонстрирующее работу полосы прокрутки

Листинг 9.2. Создание электронного индикатора и полосы прокрутки (файл main.cpp)

```
#include <QtWidgets>

int main (int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget     wgt;
    QLCDNumber* plcd = new QLCDNumber(4);
    QScrollBar* phsb = new QScrollBar(Qt::Horizontal);

    QObject::connect(phsb, SIGNAL(valueChanged(int)),
                     plcd, SLOT(display(int))
                    );

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(plcd);
    pvbxLayout->addWidget(phsb);
    wgt.setLayout(pvbxLayout);

    wgt.resize(250, 150);
    wgt.show();

    return app.exec();
}
```

В листинге 9.2 создаются виджеты электронного индикатора (указатель `plcd`) и полосы прокрутки (указатель `phsb`). После этого сигнал `valueChanged()` полосы прокрутки соединяется со слотом `display()` электронного индикатора, служащего для отображения значений целого типа, при помощи метода `connect()`. В завершение виджеты электронного индикатора и полосы прокрутки размещаются вертикально на поверхности виджета `wgt` при помощи объекта класса `QVBoxLayout`.

Установщик

Класс `QDial` виджета установщика определен в заголовочном файле QDial. Этот виджет очень похож на регулятор громкости радиоприемника, который можно вращать при помощи мыши или клавиш управления курсором. По своим функциональным возможностям он похож на ползунок. Разница в том, что круглая форма этого виджета позволяет пользователю после достижения максимального значения сразу перейти к минимальному, и наоборот. Для разрешения или запрета прокручивания служит слот `setWrapping()`.

За отображение рисок отвечают метод `setNotchTarget()`, который устанавливает их количество, и слот `setNotchesVisible()`, который управляет их видимостью.

Окно приложения (листинг 9.3), представленное на рис. 9.3, содержит виджеты установщика и индикатора выполнения. Состояние последнего зависит от местоположения стрелки установщика.

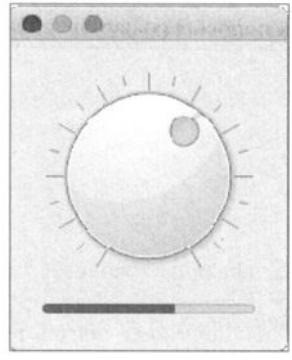


Рис. 9.3. Окно приложения, демонстрирующее работу установщика

Листинг 9.3. Создание установщика и индикатора выполнения (файл main.cpp)

```
#include <QtWidgets>

int main (int argc, char** argv)
{
    QApplication app(argc, argv);

    QWidget       wgt;
    QDial*        pdia = new QDial;
    QProgressBar* pprb = new QProgressBar;

    pdia->setRange(0, 100);
    pdia->setNotchTarget(5);
    pdia->setNotchesVisible(true);
    QObject::connect(pdia, SIGNAL(valueChanged(int)),
                     pprb, SLOT(setValue(int))
                    );

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(pdia);
    pvbxLayout->addWidget(pprb);
    wgt.setLayout(pvbxLayout);

    wgt.resize(180, 200);
    wgt.show();

    return app.exec();
}
```

В листинге 9.3 создаются виджеты установщика (указатель `pdia`) и индикатора выполнения (указатель `pprb`). Вызовом метода `setRange()` виджета установщика задается диапазон значений от 0 до 100. Метод `setNotchTarget()` устанавливает шаг рисок, равный 5, а слот `setNotchesVisible()` делает их видимыми, получив значение `true`.

Сигнал виджета установщика `valueChanged(int)` соединяется при помощи метода `connect()` со слотом индикатора выполнения `setProgress(int)`.

В завершение виджеты установщика и индикатора выполнения при помощи объекта класса `QVBoxLayout` размещаются вертикально.

Резюме

Виджеты настроек применяются в тех случаях, когда не требуется точная установка значений. Базовым классом для всех виджетов установки является класс `QAbstractSlider`, который содержит основные методы для определения диапазона, шага и текущего значения.

Ползунок позволяет осуществлять настройки параметров приложения — например, громкость звука, скорость движения указателя мыши и т. п. Этот виджет содержит риски, дающие пользователю визуально более четкое представление о шаге и месте нахождения ползунка.

Полоса прокрутки позволяет отображать текстовые или графические данные, превышающие по размерам отведенную для них область. При помощи указателя текущего положения можно перемещать данные из невидимой области в видимую.

Установщик очень похож на виджет ползунка, разница состоит во внешнем виде, в префиксах отправляемых сигналов и в возможности разрешения прокручивания при переходе от максимального значения к минимальному.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/09-510/** или с помощью следующего QR-кода (рис. 9.4):



Рис. 9.4. QR-код для доступа на страницу комментариев к этой главе

ГЛАВА 10

Элементы ввода

> Человек и компьютер — разные, и, увы, многие считают, что легче приспособить человека к компьютеру, чем наоборот.
>
> *Чарльз Петцольд*

Группа виджетов элементов ввода представляет собой основу для ввода и редактирования данных — текста и чисел — пользователем. Бóльшая часть элементов ввода может работать с буфером обмена и поддерживает технологию перетаскивания (drag & drop), что избавляет разработчика от дополнительной реализации. Текст можно выделять с помощью мыши, клавиатуры и контекстного меню.

Однострочное текстовое поле

Этот виджет является самым простым элементом ввода. Класс `QLineEdit` однострочного текстового поля определен в заголовочном файле QLineEdit.

Текстовое поле состоит из прямоугольной области для ввода строки текста, поэтому не следует использовать этот виджет в тех случаях, когда требуется вводить более одной строки. Для ввода многострочного текста имеется класс `QTextEdit`.

Текст, находящийся в виджете, возвращает метод `text()`. Если содержимое виджета изменилось, то отправляется сигнал `textChanged()`. В тех же случаях, когда нужно реагировать на изменения содержимого, которые были произведены не программно, а пользователем, то следует подсоединиться к сигналу `textEdited()`. Сигнал `returnPressed()` уведомляет о нажатии пользователем клавиши <Enter>. Вызов метода `setReadOnly()` с параметром `true` устанавливает режим «только для чтения», в котором пользователь может лишь просматривать текст, но не редактировать его. Текст для инициализации виджета можно передать в слот `setText()`.

Для однострочного текстового поля можно включить режим ввода пароля, который устанавливается вызовом метода `setEchoMode()` с флагом `Password`. В результате этого вводимые символы не отображаются, а заменяются символом *.

Окно программы (листинг 10.1), показанное на рис. 10.1, имеет два однострочных поля, одно из которых установлено в режим ввода пароля. Вводимый в это поле текст отображается в виджете надписи.



Рис. 10.1. Окно программы с однострочными полями ввода

Листинг 10.1. Создание однострочных полей (файл main.cpp)

```
#include <QtWidgets>

int main (int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QLabel* plblDisplay = new QLabel;
    plblDisplay->setFrameStyle(QFrame::Box | QFrame::Raised);
    plblDisplay->setLineWidth(2);
    plblDisplay->setFixedHeight(50);

    QLabel*    plblText = new QLabel("&Text:");
    QLineEdit* ptxt     = new QLineEdit;
    plblText->setBuddy(ptxt);
    QObject::connect(ptxt, SIGNAL(textChanged(const QString&)),
                     plblDisplay, SLOT(setText(const QString&))
                    );

    QLabel*    plblPassword = new QLabel("&Password:");
    QLineEdit* ptxtPassword = new QLineEdit;
    plblPassword->setBuddy(ptxtPassword);
    ptxtPassword->setEchoMode(QLineEdit::Password);
    QObject::connect(ptxtPassword, SIGNAL(textChanged(const QString&)),
                     plblDisplay, SLOT(setText(const QString&))
                    );

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(plblDisplay);
    pvbxLayout->addWidget(plblText);
    pvbxLayout->addWidget(ptxt);
    pvbxLayout->addWidget(plblPassword);
    pvbxLayout->addWidget(ptxtPassword);
    wgt.setLayout(pvbxLayout);

    wgt.show();

    return app.exec();
}
```

В листинге 10.1 создается виджет надписи (указатель `plblDisplay`). Метод `setFrame()` устанавливает стиль рамки, а `setLineWidth()` — ее толщину. Высота виджета надписи фиксируется с помощью метода `setFixedHeight()`. Еще две надписи, `plblText` и `plblPassword`, связываются с виджетами однострочного текстового поля методом `setBuddy()`. Затем сигналы `textChanged()` соединяются со слотами `setText()` виджета надписи `plblDisplay` для отображения вводимого текста. И в завершение виджеты размещаются вертикально при помощи объекта класса `QVBoxLayout`.

Количество вводимых символов можно ограничить методом `setMaxLength()`, передав в него целое значение, ограничивающее максимальную длину строки. Для получения текущего максимального значения длины существует метод `maxLength()`.

Класс `QLineEdit` предоставляет следующие слоты для работы с буфером обмена:

- `copy()` — копирует выделенный текст;
- `cut()` — копирует выделенный текст и удаляет его из поля ввода;
- `paste()` — вставляет текст (начиная с позиции курсора), стирая выделенный текст.

Метод `undo()` отменяет последнюю проделанную операцию, а метод `redo()` повторяет последнюю отмененную. Уточнить возможность исспользования операций отмены и повтора можно с помощью методов `isUndoAvailable()` и `isRedoAvailable()`, возвращающих булевы значения.

Правильность ввода гарантируется при помощи специального объекта-контроллера (validator), пример реализации которого рассмотрен далее в *разд. «Проверка ввода»*.

Редактор текста

Класс `QTextEdit` позволяет осуществлять просмотр и редактирование как простого текста, так и текста в формате HTML. Он унаследован от класса `QAbstractScrollArea`, что дает возможность автоматически отображать полосы прокрутки, если текст не может быть полностью отображен в отведенной для него области.

> ***Редактор для обычного текста***
>
> Если вам нужен редактор для обычного текста, то целесообразнее воспользоваться вместо класса `QTextEdit` классом `QPlainTextEdit`. Класс `QPlainTextEdit` не поддерживает формат RTF (Rich Text Format, формат «обогащенного» текста), в силу чего является более легковесным, простым и эффективным.

Класс `QTextEdit` содержит следующие методы:

- `setReadOnly()` — устанавливает или снимает режим блокировки изменения текста;
- `text()` — возвращает текущий текст.

А вот и некоторые его слоты:

- `setPlainText()` — установка обычного текста;
- `setHtml()` — установка текста в формате HTML;
- `copy()`, `cut()` и `paste()` — работа с буфером обмена (копировать, вырезать и вставить соответственно);
- `selectAll()` или `deselect()` — выделение или снятие выделения всего текста;
- `clear()` — очистка поля ввода.

И сигналы:

- `textChanged()` — отправляется при изменении текста;
- `selectionChanged()` — отправляется при изменениях выделения текста.

Для работы с выделенным текстом служит класс `QTextCursor`, и объект этого класса содержится в классе `QTextEdit`. Класс `QTextCursor` предоставляет методы для создания участков

выделения текста, получения содержимого выделенного текста и его удаления. Указатель на объект класса `QTextCursor` можно получить вызовом метода `QTextEdit::textCursor()`.

Виджеты класса `QTextEdit` также содержат в себе объект `QTextDocument`, указатель на который можно получить посредством метода `QTextEdit::document()`. Можно также присвоить ему другой документ при помощи метода `QTextEdit::setDocument()`. Класс `QTextDocument` предоставляет слот `undo()` (для отмены) или `redo()` (для повтора действий). При вызове слотов `undo()` и `redo()` посылаются сигналы `undoAvailable(bool)` и `redoAvailable(bool)`, сообщающие об успешном (или безуспешном) проведении операции. Эти сигналы отправляются как из класса `QTextDocument`, так и из класса `QTextEdit`. В большинстве случаев удобнее использовать сигналы класса `QTextEdit`.

Большинство методов класса `QTextEdit` являются делегирующими для класса `QTextDocument`. Например, как уже было сказано ранее, класс `QTextEdit` способен отображать файлы с кодом на языке HTML, содержащие таблицы и растровые изображения. Для его размещения и показа можно воспользоваться либо методом `setHtml()`, в который передается строка, содержащая в себе текст в формате HTML, либо слотом `insertHtml()`. Эти методы определены в обоих классах, и их вызов из объекта класса `QTextEdit` приведет к тому, что будет вызван аналогичный метод из объекта класса `QTextDocument`.

Для помещения обычного текста в область виджета можно воспользоваться методом `setPlainText()` или слотом `insertPlainText()`. При помощи слота `append()` осуществляется добавление текста, причем добавленный текст не вносится в список операций, действие которых можно вернуть с помощью слота `undo()`, что делает этот слот быстрым и не требующим дополнительных затрат памяти. Метод `find()` может быть использован для поиска и выделения заданной строки в тексте.

Подсветка синтаксиса

Класс `QTextEditor` можно использовать совместно с классом `QSyntaxHighlighter` для подсветки (расцветки) синтаксиса (см. далее).

Слоты `zoomIn()` и `zoomOut()` предназначены для увеличения или уменьшения размера шрифта, и их действие не распространяется на растровые изображения.

Отображение текста в формате HTML

Если вам требуется только отобразить текст в формате HTML, то, возможно, лучше воспользоваться классом `QLabel` (см. *главу 7*).

Следующий пример (листинг 10.2) отображает HTML-документ (рис. 10.2). Текст документа можно редактировать.

Листинг 10.2. Программа отображения HTML-документа (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QTextEdit    txt;

    txt.setHtml("<HTML>"
                "<BODY BGCOLOR=YELLOW>"
```

```
                "<CENTER><IMG SRC=\":/MetroGnome.png\"></CENTER>"
                "<H2><CENTER>Gnome Poem</CENTER></H2>"
                "<FONT COLOR=BLUE>"
                "<P ALIGN=\"center\">"
                    "<I>"
                    "Magic! Magic!<BR>"
                    "Magic! Magic!<BR>"
                    "Magic! Magic!<BR>"
                    "..."
                    "</I>"
                "</P>"
                "</FONT>"
                "</BODY>"
                "</HTML>"
               );
    txt.resize(320, 350);
    txt.show();

    return app.exec();
}
```

В листинге 10.2 создается виджет верхнего уровня `txt`. Метод `setHtml()` устанавливает в виджете `QTextEdit` текст в формате HTML.

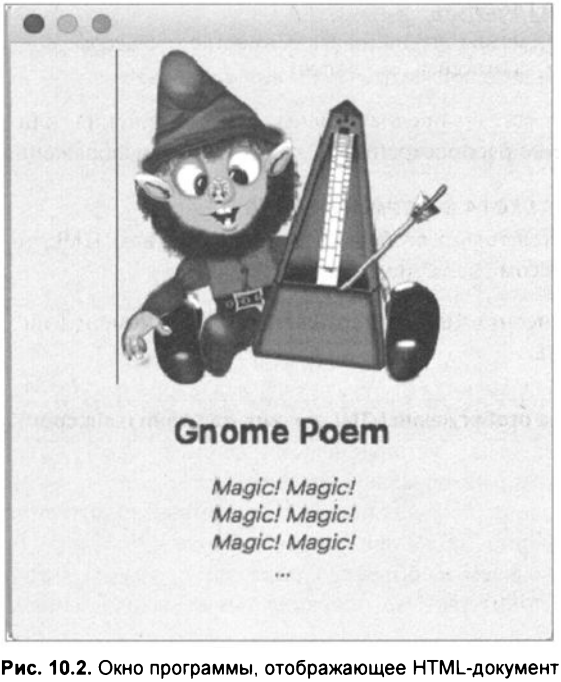


Рис. 10.2. Окно программы, отображающее HTML-документ

Запись в файл

Класс `QTextDocumentWriter` предоставляет три формата для записи содержимого объекта класса `QTextDocument`: в PlainText (простой текст), в HTML и в ODF (OpenDocument Format, открытый формат документов для офисных приложений). Последний формат используется многими приложениями, включая OpenOffice.org и LibreOffice. Для того чтобы записать файл в нужном формате, необходимо передать строку с форматом в метод `setFormat()`. Например, для записи в ODF-формат программный код может быть следующим (листинг 10.3).

Листинг 10.3. Запись файла в формате ODF

```
QTextEdit* ptxt = new QTextEdit("This is a <b>TEST</b>");
QTextDocumentWriter writer;
writer.setFormat("odf");
writer.setFileName("output.odf");
writer.write(ptxt->document());
```

В листинге 10.3 мы создали виджеты редактора текста (указатель `ptxt`) и объект поддержки записи (`writer`). Затем установили вызовом метода `setFormat()` нужный нам формат ODF и задали имя файла вызовом метода `setFileName()`. Вызов метода `write()` выполняет запись в файл, этот метод принимает в качестве параметра указатель на объект класса `QTextDocument`.

Запись в формат PDF классом `QTextDocumentWriter` не поддерживается, но ее легко осуществить путем рисования в контексте `QPrinter`. Вот небольшой пример, как это можно сделать (листинг 10.4).

> ***Использование класса QPrinter***
>
> Класс `QPrinter` находится в модуле `QtPrintSupport`, поэтому, для того чтобы использовать этот класс, необходимо добавить в проектный файл строку `QT += printsupport`.

Листинг 10.4. Запись файла в формате PDF

```
QTextEdit* ptxt = new QTextEdit("This is a <b>TEST</b>");
QPrinter printer(QPrinter::HighResolution);
printer.setOutputFormat(QPrinter::PdfFormat);
printer.setOutputFileName("output.pdf");
ptxt->document()->print(&printer);
```

В листинге 10.4 мы создаем виджет текстового редактора (указатель `ptxt`), а также объект принтера (`printer`), который устанавливаем при создании в режим высокого разрешения `HighResolution`. Далее мы устанавливаем формат для вывода вызовом метода `setOutputFormat()`, в который передаем значение `PdfFormat`, — возможны также варианты `NativeFormat` и `PostScriptFormat` для печати в системный принтер или для записи в формате PostScript соответственно. Затем при помощи метода `setOutputFileName()` мы задаем имя файла для записи и вызываем из объекта класса `QTextDocument` метод, в который передаем адрес на наш объект принтера. Эта операция осуществляет запись в назначенный нами файл.

> ***Вариант создания PDF-файлов***
>
> Для создания PDF-файлов можно использовать так же и класс `QPdfWriter`.

Расцветка синтаксиса (syntax highlighting)

Расцветка и форматирование способствуют более удобному восприятию структуры текста программы. Добавить расцветку синтаксиса в `QTextEdit` очень просто — для этого нужно унаследовать класс `QSyntaxHighlighter` и реализовать в унаследованном классе метод `highlightBlock()`.

Следующий простой пример выделяет цифры в тексте красным цветом:

```
/*virtual*/ void MyHighlighter::highlightBlock(const QString& str)
{
    for (int i = 0; i < str.length(); ++i) {
        if (str.at(i).isNumber()) {
            setFormat(i, 1, Qt::red);
        }
    }
}
```

Как правило, в метод `highlightBlock()` передается одна строка текста. В первом параметре метода `setFormat()` мы передаем стартовое значение, второй параметр задает количество символов (в нашем случае 1), а в последнем параметре мы передаем цвет для расцветки (в нашем случае `Qt::red`). Вместо цвета в последнем параметре можно также передавать и шрифт (`QFont`, см. *главу 20*).

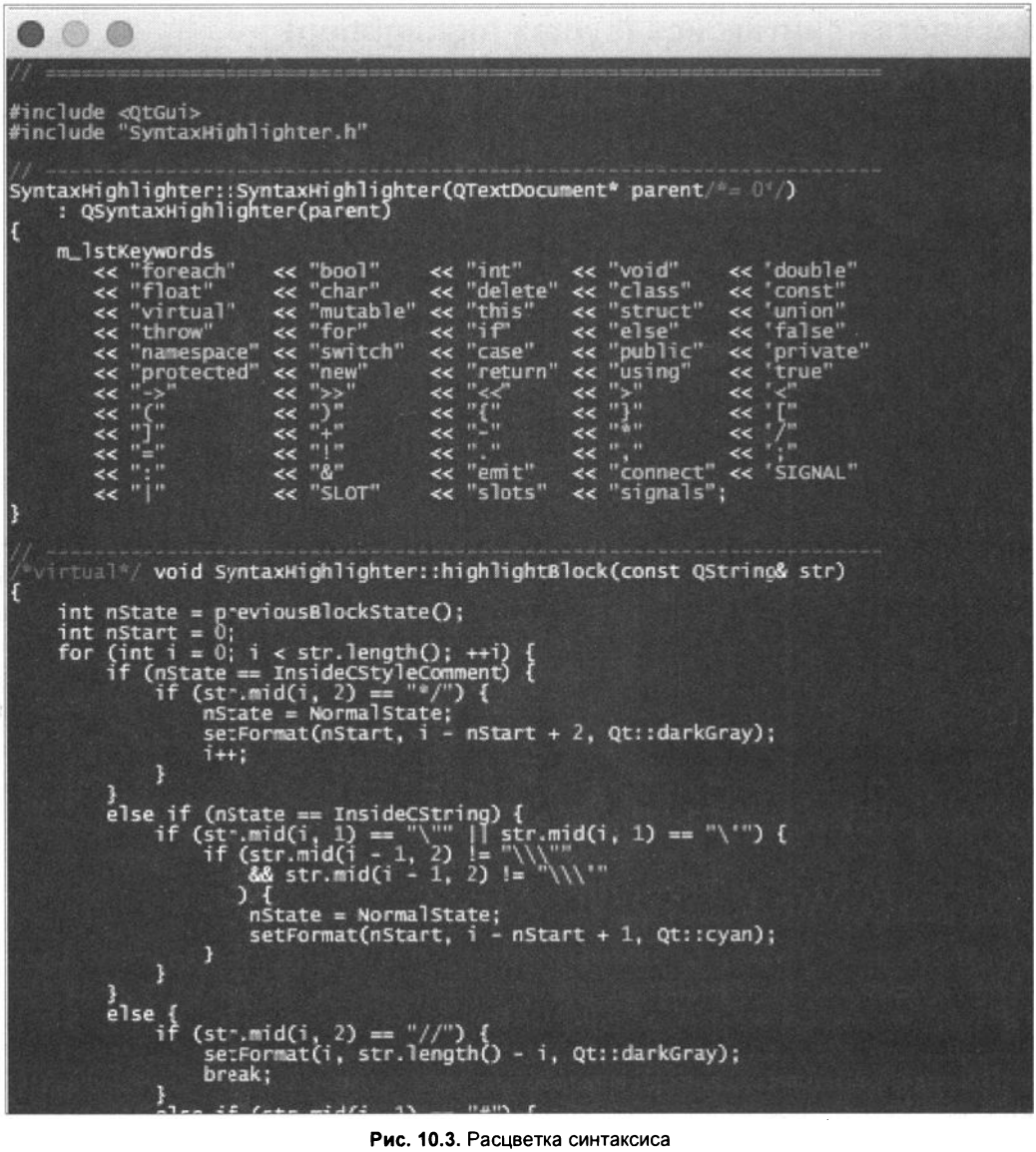
Для того чтобы применить объект класса расцветки к объекту `QTextEdit`, нужно при создании передать ему указатель на объект `QTextDocument`. Например:

```
MyHighlighter* pHighlighter = new MyHighlighter(ptxt->document());
```

Как видно из примера, для задания расцветки синтаксиса нужен указатель на объект класса `QTextDocument`, а это значит, что применение расцветки не ограничивается только классом `QTextEdit`, и ее можно применять ко всем классам, имеющим в своем распоряжении объект класса `QTextDocument`. В число таких классов входят, например, `QTextBrowser`, `QTextFrame`, `QTextTable`, класс элемента текста `QGraphicsTextItem` графического представления (см. *главу 21*) и другие классы.

Теперь, когда нам стал ясен принцип применения класса `QSyntaxHighlighter`, перейдем к более сложному примеру и реализуем виджет, который делает расцветку программ на языке C++ в стиле Borland (рис. 10.3).

В основной программе (листинг 10.5) мы создаем виджет `txt` класса `QTextEdit` — это и будет наш редактор. Затем мы создаем объект шрифта `fnt` и устанавливаем его в нашем редакторе вызовом метода `setDefaultFont()` из объекта документа редактора. Далее создаем объект расцветки синтаксиса созданного нами и описанного чуть позже класса `SyntaxHighlighter` и передаем ему в качестве предка указатель на объект документа нашего редактора — этот объект позаботится об его уничтожении при своем разрушении. Создаем объект палитры `pal` — он нам нужен, чтобы установить цвет шрифта (желтый) и фона (темно-синий) по умолчанию. Отображаем наш редактор вызовом метода `show()` и задаем его размеры методом `resize()`. Исходный файл SyntaxHighlighter.cpp включен нами в ресурс, поэтому при его открытии методом `open()` нам не нужно проверять успешность этой операции. Его текст считывается методом `readAll()` и устанавливается в нашем редакторе вызовом метода `setPlainText()`.



```
// ============================================================================

#include <QtGui>
#include "SyntaxHighlighter.h"

// ----------------------------------------------------------------------------
SyntaxHighlighter::SyntaxHighlighter(QTextDocument* parent/*= 0*/)
    : QSyntaxHighlighter(parent)
{
    m_lstKeywords
        << "foreach"   << "bool"    << "int"     << "void"    << "double"
        << "float"     << "char"    << "delete"  << "class"   << "const"
        << "virtual"   << "mutable" << "this"    << "struct"  << "union"
        << "throw"     << "for"     << "if"      << "else"    << "false"
        << "namespace" << "switch"  << "case"    << "public"  << "private"
        << "protected" << "new"     << "return"  << "using"   << "true"
        << "->"        << ">>"      << "<<"      << ">"       << "<"
        << "("         << ")"       << "{"       << "}"       << "["
        << "]"         << "+"       << "-"       << "*"       << "/"
        << "="         << "!"       << "."       << ","       << ";"
        << ":"         << "&"       << "emit"    << "connect" << "SIGNAL"
        << "|"         << "SLOT"    << "slots"   << "signals";
}

// ----------------------------------------------------------------------------
/*virtual*/ void SyntaxHighlighter::highlightBlock(const QString& str)
{
    int nState = previousBlockState();
    int nStart = 0;
    for (int i = 0; i < str.length(); ++i) {
        if (nState == InsideCStyleComment) {
            if (str.mid(i, 2) == "*/") {
                nState = NormalState;
                setFormat(nStart, i - nStart + 2, Qt::darkGray);
                i++;
            }
        }
        else if (nState == InsideCString) {
            if (str.mid(i, 1) == "\"" || str.mid(i, 1) == "\'") {
                if (str.mid(i - 1, 2) != "\\\""
                    && str.mid(i - 1, 2) != "\\\'"
                   ) {
                    nState = NormalState;
                    setFormat(nStart, i - nStart + 1, Qt::cyan);
                }
            }
        }
        else {
            if (str.mid(i, 2) == "//") {
                setFormat(i, str.length() - i, Qt::darkGray);
                break;
            }
            else if (str.mid(i, 1) == "#") {
```

Рис. 10.3. Расцветка синтаксиса

Листинг 10.5. Создание редактора, обеспечивающего расцветку синтаксиса (файл main.cpp)

```
#include <QtWidgets>
#include "SyntaxHighlighter.h"

int main (int argc, char** argv)
{
    QApplication app(argc, argv);
    QTextEdit    txt;

    QFont fnt("Lucida Console", 9, QFont::Normal);
    txt.document()->setDefaultFont(fnt);
```

```
    new SyntaxHighlighter(txt.document());

    QPalette pal = txt.palette();
    pal.setColor(QPalette::Base, Qt::darkBlue);
    pal.setColor(QPalette::Text, Qt::yellow);
    txt.setPalette(pal);

    txt.show();
    txt.resize(640, 480);

    QFile file(":/SyntaxHighlighter.cpp");
    file.open(QFile::ReadOnly);
    txt.setPlainText(QLatin1String(file.readAll()));

    return app.exec();
}
```

В заголовочном файле (листинг 10.6) наш класс `SyntaxHighlighter` наследуется от класса `QSyntaxHighlighter`. В нем определены перечисления `NormalState`, `InsideCStyleComment` и `InsideCString`, они понадобятся далее для определения текущего состояния фрагмента. Мы также перегружаем метод `highlightBlock()`, который необходим для реализации собственной расцветки синтаксиса. Метод `getKeyword()` — это наш эксперт, который будет давать ответ на вопрос, является ли строка на указанной позиции ключевым словом языка C++ или определением Qt.

Листинг 10.6. Определение класса `SyntaxHighlighter` (файл SyntaxHighlighter.h)

```
#pragma once

#include <QSyntaxHighlighter>

class QTextDocument;

// ============================================================================
class SyntaxHighlighter: public QSyntaxHighlighter {
Q_OBJECT
private:
    QStringList m_lstKeywords;
protected:
    enum { NormalState = -1, InsideCStyleComment, InsideCString };

    virtual void highlightBlock(const QString&);

    QString getKeyword(int i, const QString& str) const;

public:
    SyntaxHighlighter(QTextDocument* parent = 0);
};
```

В конструкторе мы инициализируем объект списка `m_lstKeywords` ключевыми словами (листинг 10.7).

Листинг 10.7. Конструктор `SyntaxHighlighter()` (файл SyntaxHighlighter.cpp)

```
SyntaxHighlighter::SyntaxHighlighter(QTextDocument* parent/*= 0*/)
    : QSyntaxHighlighter(parent)
{
    m_lstKeywords
       << "foreach"   << "bool"    << "int"    << "void"    << "double"
       << "float"     << "char"    << "delete" << "class"   << "const"
       << "virtual"   << "mutable" << "this"   << "struct"  << "union"
       << "throw"     << "for"     << "if"     << "else"    << "false"
       << "namespace" << "switch"  << "case"   << "public"  << "private"
       << "protected" << "new"     << "return" << "using"   << "true"
       << "->"        << ">>"      << "<<"     << ">"       << "<"
       << "("         << ")"       << "{"      << "}"       << "["
       << "]"         << "+"       << "-"      << "*"       << "/"
       << "="         << "!"       << "."      << ","       << ";"
       << ":"         << "&"       << "emit"   << "connect" << "SIGNAL"
       << "|"         << "SLOT"    << "slots"  << "signals";
}
```

С принципом реализации метода `highlightBlock()` мы уже успели познакомиться, поэтому остановимся только на основных моментах. В метод `highlightBlock()` передается только одна строка текста, но не все концепции синтаксиса ограничиваются одной строкой. Поэтому мы ввели для этих случаев три состояния:

- `NormalState` — нормальное состояние, при котором должна использоваться расцветка, задаваемая нашей палитрой (см. листинг 10.5);
- `InsideCString` — состояние, в котором текущая позиция находится внутри строки. В этом случае цвет текста должен быть бирюзовым: `Qt::cyan`;
- `InsideCStyleComment` — состояние, когда текущая позиция находится в комментарии вида `/*...*/`. В этом случае цвет текста должен быть темно-серым: `Qt::darkGray`.

Текущее состояние мы устанавливаем в переменной `nState` (листинг 10.8), а получаем его вызовом метода `previousBlockState()`. Самым первым в цикле мы контролируем моменты завершения состояний `InsideCStyleComment` и `InsideCString`. Для завершения первого мы проверяем на текущей позиции строку `*/` и, если находим ее, присваиваем переменной `nState` состояние `NormalState` и докрашиваем строку в темно-серый цвет, после чего увеличиваем переменную `i` на единицу, так как следующий символ мы уже обработали.

В отслеживании наступления состояния конца строки `InsideCString` мы сначала проверяем наличие символа кавычек (`"` либо `'`), а потом следим за тем, чтобы этому символу не предшествовал символ `\`, так как иначе, встретив строку вида `\"`, расцветка синтаксиса неправильно интерпретирует ситуацию и раскрасит косую черту как элемент строки, а кавычку поймет как конец строки. Если последнее контрольное условие выполняется, то строка завершилась, и мы изменяем статус на нормальный и докрашиваем ее последний символ в бирюзовый цвет.

В третьей секции основного `if` мы контролируем наступления состояний, сравнивая текущую позицию в строке с различными символами, которые должны менять форматирование.

Первые три сравнения, как и последнее, не проводят смену состояния, так как не могут выходить за пределы одной строки. Это однострочный комментарий вида `//`, директивы препроцессора, начинающиеся с символа `#`, цифры (определяются с помощью метода `QString::isNumber()`), а также ключевые слова языка C++ и определения Qt. Ключевые слова определяются при помощи реализованного нами (листинг 10.9) метода `getKeyword()`. Этот метод возвращает само ключевое слово, которое он нашел на заданной позиции. Найденное слово мы раскрашиваем в белый цвет и увеличиваем переменную `i` на его длину, так как обрабатываем все символы слова.

Для смены состояния на `InsideCStyleComment` или `InsideCString` достаточно нахождения на текущей позиции строки `/*` либо символа " соответственно.

В случае если цикл завершился, и мы находимся в текущем состоянии `InsideCStyleComment` либо `InsideCString`, мы закрашиваем наш блок с начала позиции смены состояния и до самого конца. В завершение мы устанавливаем текущее состояние методом `setCurrentState()`.

Листинг 10.8. Метод `highlightBlock()` (файл SyntaxHighlighter.cpp)

```
/*virtual*/ void SyntaxHighlighter::highlightBlock(const QString& str)
{
    int nState = previousBlockState();
    int nStart = 0;
    for (int i = 0; i < str.length(); ++i) {
        if (nState == InsideCStyleComment) {
            if (str.mid(i, 2) == "*/") {
                nState = NormalState;
                setFormat(nStart, i - nStart + 2, Qt::darkGray);
                i++;
            }
        }
        else if (nState == InsideCString) {
            if (str.mid(i, 1) == "\"" || str.mid(i, 1) == "\'") {
                if (str.mid(i - 1, 2) != "\\\""
                    && str.mid(i - 1, 2) != "\\\'"
                   ) {
                    nState = NormalState;
                    setFormat(nStart, i - nStart + 1, Qt::cyan);
                }
            }
        }
        else {
            if (str.mid(i, 2) == "//") {
                setFormat(i, str.length() - i, Qt::darkGray);
                break;
            }
            else if (str.mid(i, 1) == "#") {
                setFormat(i, str.length() - i, Qt::green);
                break;
            }
```

```
            else if (str.at(i).isNumber()) {
                setFormat(i, 1, Qt::cyan);
            }
            else if (str.mid(i, 2) == "/*") {
                nStart = i;
                nState = InsideCStyleComment;
            }
            else if (str.mid(i, 1) == "\"" || str.mid(i, 1) == "\'") {
                nStart = i;
                nState = InsideCString;
            }
            else {
                QString strKeyword = getKeyword(i, str);
                if (!strKeyword.isEmpty()) {
                    setFormat(i, strKeyword.length(), Qt::white);
                    i += strKeyword.length() - 1;
                }
            }
        }
    }
    if (nState == InsideCStyleComment) {
        setFormat(nStart, str.length() - nStart, Qt::darkGray);
    }
    if (nState == InsideCString) {
        setFormat(nStart, str.length() - nStart, Qt::cyan);
    }
    setCurrentBlockState(nState);
}
```

Назначение метода `getKeyword()` (листинг 10.9) заключается в нахождении ключевых слов языка C++ и определений Qt. Мы работаем с объектом списка ключевых слов `m_lstKeyword` и всякий раз при вызове метода пытаемся найти в цикле `foreach` совпадение с элементами списка согласно текущей позиции и размеру ключевого слова. При удаче ключевое слово присваивается промежуточной строковой переменной `strTemp`, цикл прерывается (`break`), и значение этой переменной возвращается в качестве результата.

Листинг 10.9. Метод getKeyword() (файл SyntaxHighlighter.cpp)

```
QString SyntaxHighlighter::getKeyword(int nPos, const QString& str) const
{
    QString strTemp = "";

    foreach (QString strKeyword, m_lstKeywords) {
        if (str.mid(nPos, strKeyword.length()) == strKeyword) {
            strTemp = strKeyword;
            break;
        }
    }

    return strTemp;
}
```

Надо отметить, что приведенный здесь алгоритм несовершенен и неправильно обрабатывает целый ряд ситуаций — например, не отличает цифр в идентификаторах от численных значений. В редакторах, поддерживающих подсветку синтаксиса, применяются, как правило, более сложные алгоритмы, рассматривать которые здесь не представляется возможным, — наш пример лишь иллюстрирует работу с классом `QSyntaxHighlighter`.

С чего начинаются виджеты счетчиков?

Виджет абстрактного счетчика — класс `QAbstractSpinBox` — предоставляет всем унаследованным от него классам небольшое текстовое поле и две стрелки для уменьшения или увеличения числовых значений. От этого виджета унаследованы следующие классы (см. рис. 5.1):

- `QSpinBox` — счетчик;
- `QDateTimeEdit` — элемент для ввода даты и времени;
- `QDoubleSpinBox` — элемент для ввода значений, имеющих тип `double`.

Можно установить циклический режим, когда за максимально возможным значением будет следовать минимально возможное, и наоборот. Этот режим устанавливается вызовом метода `setWrapping()` с параметром `true`.

Реализованы два метода пошагового изменения значений `stepUp()` и `stepDown()`, которые симулируют нажатие на кнопки стрелок.

С помощью метода `setSpecialValueText()` устанавливается текст, независимо от числового значения, например:

```
pspb->setSpecialValueText("default");
```

Счетчик

Виджет `QSpinBox` предоставляет пользователю доступ к ограниченному диапазону чисел. Для предотвращения выхода за пределы установленного диапазона, который устанавливается методом `setRange()`, все вводимые значения проверяются. Значения можно устанавливать с помощью метода `setValue()`, а получать — методом `value()`. При изменении значений посылаются сразу два сигнала `valueChanged()`: один с параметром типа `int`, а другой — с `const QString&`.

Можно изменить способ отображения с помощью методов `setPrefix()` и `setSuffix()`. Например, вызов следующих методов приведет к тому, что число будет отображаться в скобках:

```
pspb->setPrefix("(");
pspb->setSuffix(")");
```

Следующий пример (листинг 10.10) позволяет выбирать и устанавливать числа в диапазоне от 1 до 100 (рис. 10.4).

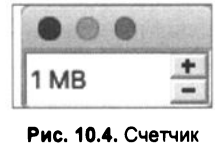


Рис. 10.4. Счетчик

В листинге 10.10 создается виджет счетчика spb. После его создания выполняется установка диапазона при помощи метода setRange(). Вызов метода setSuffix() добавляет строку " MB" после отображаемой счетчиком величины, а метод setButtonSymbols(), которому передается флаг PlusMinus, заменяет стрелки счетчика знаками +/-. Метод setWrapping() устанавливает циклический режим. Не все стили поддерживают отображение этого режима, поэтому в конце мы устанавливаем стиль Windows.

Листинг 10.10. Создание счетчика (файл main.cpp)

```
#include <QtWidgets>

int main (int argc, char** argv)
{
    QApplication app(argc, argv);
    QSpinBox     spb;

    spb.setRange(1, 100);
    spb.setSuffix(" MB");
    spb.setButtonSymbols(QSpinBox::PlusMinus);
    spb.setWrapping(true);
    spb.show();
    spb.resize(100, 30);

    QApplication::setStyle("Windows");
    return app.exec();
}
```

Элемент ввода даты и времени

Этот виджет состоит из нескольких секций, предназначенных для показа и изменения даты и времени.

При изменении даты или времени посылается сигнал dateTimeChanged(). Для класса QDateTimeEdit этот сигнал передает константную ссылку на объект типа QDateTime.

Следующий пример (листинг 10.11) отображает актуальную дату и время запуска программы, которые можно модифицировать (рис. 10.5).

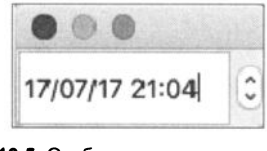


Рис. 10.5. Отображение даты и времени

Как видно из листинга 10.11, при создании виджета QDataTimeEdit в его конструктор передаются текущая дата и время, которые возвращаются вызовом статического метода QDateTime::currentDateTime(). Виджет отобразится на экране после вызова метода show().

Листинг 10.11. Создание виджета, отображающего дату и время (файл main.cpp)

```
#include <QtWidgets>

int main (int argc, char** argv)
{
    QApplication  app(argc, argv);
    QDateTimeEdit dateTimeEdit(QDateTime::currentDateTime());
    dateTimeEdit.show();
    return app.exec();
}
```

Проверка ввода

Объект класса `QValidator` (далее контролёр) гарантирует правильность ввода пользователя. Для установки объекта класса `QValidator` необходимо передать его в метод `setValidator()`, который содержится в классах `QComboBox` и `QLineEdit`. Для проверки ввода чисел пользуются готовыми классами `QIntValidator` и `QDoubleValidator`. Создавая свой класс проверки ввода, нужно унаследовать класс от `QValidator` и перезаписать метод `validate()`, в который передается вводимая строка и позиция курсора. Метод должен возвращать следующие значения:

- `QValidator::Invalid` — если строка не может быть принята;
- `QValidator::Intermediate` — строка не может быть принята в качестве конечного результата. Например, если строка должна представлять численное значение от 50 до 100, то введение числа 1 будет представлять собой промежуточное значение;
- `QValidator::Acceptable` — если строка может быть принята без изменений.

В следующем примере пользователю предлагается ввести свое имя, и цифры в этом случае недопустимы (рис. 10.6). Такое ограничение отслеживается классом контролёра, который не допускает, чтобы имя содержало цифры.

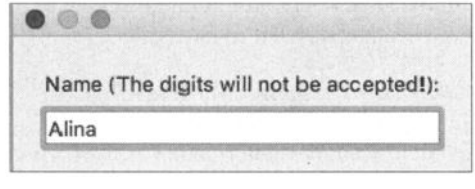


Рис. 10.6. Окно программы, контролирующей данные, вводимые пользователем

В основной программе (листинг 10.12) класс `NameValidator` унаследован от класса `QValidator`. В этом классе выполняется перезапись метода `validate()`, который получает вводимый текст и позицию курсора. Внутри метода создается объект регулярного выражения `rxp` (см. *главу 4*), в конструктор которого передается шаблон, представляющий собой диапазон цифр от 0 до 9. В операторе `if` вызовом метода `QString::contains()` осуществляется проверка строки на содержание в ней заданного шаблона. В случае, если совпадение найдено, метод возвращает значение `Invalid`, сообщая, что ввод не удовлетворяет заданным критериям, в противном случае возвращается значение `Acceptable`, и ввод принимается.

Листинг 10.12. Класс NameValidator (файл main.cpp)

```
class NameValidator : public QValidator {
public:
    NameValidator(QObject* parent) : QValidator(parent)
    {
    }

    virtual State validate(QString& str, int& pos) const
    {
        QRegExp rxp = QRegExp("[0-9]");
        if (str.contains(rxp)) {
            return Invalid;
        }
        return Acceptable;
    }
};
```

В листинге 10.13 создается виджет надписи (указатель `plblText`), виджет однострочного текстового поля (указатель `ptxt`) и объект контролёра (указатель `pnameValidator`). После создания вызовом метода `setValidator()` контролёр устанавливается в виджете однострочного текстового поля (указатель `ptxt`), а последний связывается с надписью вызовом метода `setBuddy()`.

Листинг 10.13. Создание контролёра (файл main.cpp)

```
int main (int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QLabel* plblText =
           new QLabel("&Name (The digits will not be accepted!):");

    QLineEdit* ptxt = new QLineEdit;

    NameValidator* pnameValidator = new NameValidator(ptxt);
    ptxt->setValidator(pnameValidator);
    plblText->setBuddy(ptxt);

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(plblText);
    pvbxLayout->addWidget(ptxt);
    wgt.setLayout(pvbxLayout);
    wgt.show();

    return app.exec();
}
```

Резюме

В этой главе мы познакомились с группой виджетов, позволяющих пользователю осуществлять ввод текста и числовых значений. Большая часть этих виджетов обладает всеми необходимыми методами для работы с буфером обмена, а также поддерживает технологию перетаскивания (drag & drop).

Класс `QLineEdit` предназначен для ввода одной текстовой строки. Для многострочного ввода следует использовать класс `QTextEdit`. Этот класс унаследован от класса `QAbstractScrollArea`, что позволяет ему отображать текст частями, если для отображения всего текста недостаточно места. При помощи класса `QTextDocumentWriter` можно записывать содержимое в форматы ODF, HTML, а также — с помощью объекта `QPrinter` — создавать PDF-файлы.

Виджет счетчика (`QSpinBox`) применяется для ввода числовых значений. Он осуществляет контроль, чтобы вводимые значения не выходили за пределы установленного диапазона.

Для проверки правильности ввода данных в виджетах классов `QComboBox` (см. *главу 11*) и `QLineEdit` можно устанавливать объект контролёра с помощью метода `setValidator()`.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/10-510/** или с помощью следующего QR-кода (рис. 10.7):



Рис. 10.7. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 11

Элементы выбора

> Если вам предстоит сделать выбор, а вы его не делаете — это тоже выбор.
>
> *У. Джеймс*

Элементы выбора представляют собой стандартные элементы графического интерфейса пользователя для отображения, модификации и выбора данных. Нужно сразу сказать, что если вы собираетесь использовать классы для просмотра списков и таблиц в простых ситуациях, то описанных здесь классов будет вполне достаточно. Надо также иметь в виду, что ряд классов этой главы: `QListWidget`, `QTreeWidget` и `QTableWidget` — базируется на архитектуре «модель-представление», рассмотренной в *главе 12*. Непосредственное же использование классов этой архитектуры дает, в сравнении с описанными здесь классами, некоторые преимущества, и поэтому в большинстве случаев является более предпочтительным. Для того чтобы ознакомиться с архитектурой «модель-представление», просто откройте следующую главу, но лучше прочтите сначала эту, тем более что ее материал не ограничивается только описанием таблиц и списков. Итак...

Простой список

Класс `QListWidget` — это виджет списка, предоставляющий пользователю возможность выбора одного или нескольких элементов. Элементы списка могут содержать текст и растровые изображения. Чтобы добавить элемент в список, нужно вызвать метод `addItem()`. В этом методе реализовано два его варианта: для текста и для объекта класса `QListWidgetItem`. Если необходимо удалить все элементы из списка, надо вызвать слот `clear()`.

Класс `QListWidgetItem` — это класс для элементов списка. Объекты этого класса могут создаваться неявно — например, при передаче текста в метод `QListWidget::addItem()`. Следует отметить, что класс `QListWidgetItem` предоставляет конструктор копирования, что позволяет создавать копии элементов. Также для этой цели можно воспользоваться методом `clone()`.

Вставка элементов

В список можно добавить сразу несколько текстовых элементов, передав объект класса `QStringList`, содержащий список строк (см. *главу 4*), в метод `insertItems()`. Допустимо воспользоваться методом `insertItem()` и для создания текстового элемента — для этого

в метод надо передать строку текста. С помощью метода `insertItem()` может быть вставлен в список также и объект `QListWidgetItem`. Отличие метода `insertItem()` от метода `addItem()` состоит в том, что он дает возможность явно указать позицию добавляемого элемента.

Созданному элементу можно присвоить растровое изображение, что выполняется с помощью метода `QListWidgetItem::setIcon()` объекта элемента списка.

Примечательно также и то, что в элементах списка можно устанавливать не только растровые изображения и текст, но и виджеты. Для этого в классе `QListWidget` определены методы `setItemWidget()` и `itemWidget()`. Первым параметром метода `setItemWidget()` нужно передать указатель на объект элемента списка, а вторым — указатель на виджет. Для того чтобы получить указатель на виджет, расположенный в элементе списка, необходимо передать в метод `itemWidget()` указатель на объект элемента списка.

> ***СНИЖЕНИЕ БЫСТРОДЕЙСТВИЯ СПИСКА НЕЖЕЛАТЕЛЬНО!***
>
> Использование виджетов в элементах списка существенно снижает быстродействие самого списка.

Следующий пример (листинг 11.1) демонстрирует использование простого списка, в котором перечисляются операционные системы (рис. 11.1).

В листинге 11.1 создается виджет простого списка `lwg`. Методом виджета списка `setIconSize()` задается размер для растровых изображений элементов. Затем список строк `lst` заполняется надписями для элементов. Пройдя по этому списку при помощи `foreach`, мы создаем и добавляем элементы в список. Вызов метода `setIcon()` устанавливает растровое изображение для каждого элемента.

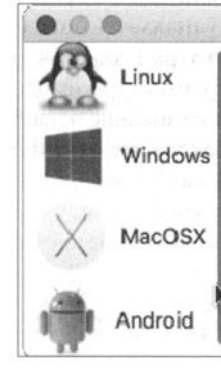


Рис. 11.1. Простой список

Листинг 11.1. Создание простого списка (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication     app(argc, argv);
    QStringList      lst;
    QListWidget      lwg;
    QListWidgetItem* pitem = 0;

    lwg.setIconSize(QSize(48, 48));
    lst << "Linux" << "Windows" << "MacOSX" << "Android";
    foreach(QString str, lst) {
        pitem = new QListWidgetItem(str, &lwg);
        pitem->setIcon(QPixmap(":/" + str + ".jpg"));
    }
```

```
    lwg.resize(125, 175);
    lwg.show();

    return app.exec();
}
```

Выбор элементов пользователем

Узнать, какой элемент выбрал пользователь, можно с помощью метода `QListWidget::currentItem()`, который возвращает указатель на выбранный элемент. Если же выбранных элементов несколько, то нужно вызвать метод `selectedItems()`, который вернет список выбранных элементов. Чтобы разрешить режим множественного выделения, необходимо установить при помощи метода `setSelectionMode()`, реализованного в базовом классе `QAbstractItemView`, значение `QAbstractItemView::MultiSelection`. В этот метод можно передавать и другие значения — например, для того чтобы запретить выделение вовсе, в него нужно передать `QAbstractItemView::NoSelection`, а для возможности выделения только одного из элементов — `QAbstractItem::SingleSelection`.

После щелчка на элементе списка отправляется сигнал `itemClicked()`. При двойном щелчке мыши на элементе отправляется сигнал `itemDoubleClicked()` с параметром `QListWidgetItem*`. После каждого изменения выделения элементов отсылается сигнал `itemSelectionChanged()`.

Изменение элементов пользователем

Чтобы предоставить пользователю возможность изменения текста элемента, необходимо вызвать из нужного объекта элемента метод `QListWidgetItem::setFlags()` и передать в него значение `Qt::ItemIsEditable` и другие требуемые значения. Например:

```
pitem->setFlags(Qt::ItemIsEditable | Qt::ItemIsEnabled);
```

Переименование осуществляется двойным щелчком мыши на элементе списка либо нажатием клавиши <F2>. По завершении переименования виджет `QListWidget` отправляет сигналы `itemChanged(QListWidgetItem*)` и `itemRenamed(QListWidgetItem*)`.

Режим пиктограмм

Виджет списка можно перевести в режим пиктограмм (режим представления в виде значков), который позволяет выбирать элементы, а также проводить над ними операции перемещения и перетаскивания (drag & drop).

Следующий пример (листинг 11.2) предоставляет пользователю выбор значка одной из четырех операционных систем (рис. 11.2).



Рис. 11.2. Режим пиктограмм

В листинге 11.2 создается виджет списка `lwg`. Вызов метода `setSelectionMode()` осуществляет установку режима выбора нескольких элементов. Далее вызовом метода `setViewMode()` с параметром `QListView::IconMode` устанавливается режим пиктограмм. Затем создается

список строк lst, который заполняется именами файлов значков. Сами элементы списка lwg создаются в цикле foreach. В метод setFlags() каждого элемента передается комбинация флагов, которые делают его доступным (Qt::ItemIsEnabled), разрешают выделение (Qt::ItemIsSelectable), делают его редактируемым (Qt::ItemIsEditable) и перетаскиваемым (Qt::ItemIsDragEnabled).

По умолчанию все помещаемые элементы будут заполнять виджет QListWidget слева направо и сверху вниз. Это можно изменить с помощью метода setFlow(), который определен в базовом классе QListView. Для этого нужно передать ему значение QListView::TopToBottom, что приведет к заполнению опций сверху вниз и слева направо.

Листинг 11.2. Создание списка в режиме пиктограмм (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication     app(argc, argv);
    QListWidget      lwg;
    QListWidgetItem* pitem = 0;
    QStringList      lst;

    lwg.setIconSize(QSize(48, 48));
    lwg.setSelectionMode(QAbstractItemView::MultiSelection);
    lwg.setViewMode(QListView::IconMode);
    lst << "Linux" << "Windows" << "MacOSX" << "Android";
    foreach(QString str, lst) {
        pitem = new QListWidgetItem(str, &lwg);
        pitem->setIcon(QPixmap(":/" + str + ".jpg"));
        pitem->setFlags(Qt::ItemIsEnabled | Qt::ItemIsSelectable |
                        Qt::ItemIsEditable | Qt::ItemIsDragEnabled);

    }
    lwg.resize(150, 150);
    lwg.show();

    return app.exec();
}
```

Сортировка элементов

Элементы списка можно упорядочить вызовом метода sortItems(). При передаче в этот метод значения Qt::AscendingOrder сортировка элементов будет выполнена в возрастающем порядке, а при значении Qt::DescendingOrder — в убывающем. Однако, если выполнить сортировку, а затем добавить новые элементы, они сортироваться не будут. Сортировка проводится в алфавитном порядке, а если нужно отсортировать по дате или по числовому значению, то необходимо унаследовать класс элемента QListWidgetItem и перезаписать в нем operator<().

Иерархические списки

Виджет `QTreeWidget` отображает элементы списка в иерархической форме и поддерживает возможность выбора пользователем одного или нескольких из них. Его часто применяют для показа содержимого дисков и каталогов. В случае, когда область отображения не в состоянии разместить все элементы, появляются полосы прокрутки. С помощью метода `setItemWidget()` в иерархическом списке можно размещать виджеты. Но при всем этом необходимо соблюдать осторожность, потому что использование виджетов в элементах может заметно снизить быстродействие самого списка. Уже, начиная с пары сотен элементов, это становится очень заметно.

Поэтому если вы хотите использовать списки с большим количеством элементов, то постарайтесь обойтись либо без виджетов, либо замените их на их эквивалентные представления. Например, для кнопок флажков есть возможность установить их представление в любом столбце. Так, после создания элемента иерархического списка (указатель `pwti`) мы устанавливаем в нем флаг `Qt::ItemIsUserCheckable` — это переводит его в режим использования кнопки флажка (листинг 11.3). Далее мы устанавливаем представление кнопки в первом столбце и снабжаем в последней строке этот столбец поясняющим текстом. На рис. 11.3 показан результат.

Листинг 11.3. Установка представления кнопки флажка

```
QTreeWidgetItem* ptwi = new QTreeWidgetItem(pTreeView);
ptwi->setFlags(ptwiTemp->flags() | Qt::ItemIsUserCheckable);
ptwi->setCheckState(0, Qt::Checked);
ptwi->setText(0, "Checkable Item");
```

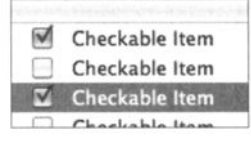


Рис. 11.3. Элементы с кнопками флажков

Для того чтобы узнать, какое состояние имеет кнопка флажка, можно осуществить перебор всех элементов списка, опросить статус по номеру столбца и сравнить его со значением `Qt::Checked`. Например:

```
if (ptwi->checkState(0) == Qt::Checked) {
    //This item is checked
}
```

Элементы списка являются объектами класса `QTreeWidgetItem` и предоставляют возможность отображать несколько столбцов с данными. Класс `QTreeWidgetItem` содержит конструктор копирования и метод `clone()` для создания копий элементов. При помощи методов `addChildren()` и `insertChildren()` можно добавлять и вставлять сразу несколько элементов.

Если необходимо снабдить элементы небольшими растровыми изображениями, то они устанавливаются с помощью метода `QTreeWidgetItem::setIcon()`, а текст элемента — с помощью `QTreeWidgetItem::setText()`. Первый параметр обоих методов соответствует номеру столбца.

Рассмотрим следующий пример. Список на рис. 11.4 содержит три элемента. Двойной щелчок мыши на элементе **Local Disk(C)** сворачивает и разворачивает вложенные элементы (папки).

Для отображения структуры каталогов диска (например, как в Проводнике ОС Windows) необходимо построить объекты `QTreeWidgetItem` в нужном иерархическом порядке. В листинге 11.4 создается виджет списка `twg`. Надписи для столбцов вносятся в список `lst` при помощи оператора <<. Метод `setHeaderLabels()` задает заголовки столбцов **Folders** (Папки) и **Used Space** (Занимаемый объем). В конструктор объекта первого элемента (указатель `ptwgItem`) передается адрес виджета списка `twg`. Этот элемент является предком для создаваемых в дальнейшем элементов и представляет собой вершину иерархии. Далее в цикле создаются 20 потомков для вершины списка. При помощи метода `setText()` задается текст столбцов, а метод `setIcon()` добавляет к первому столбцу растровое изображение. Метод `setExpanded()` разворачивает элемент вершины `ptwgItem` и показывает иерархию элементов.

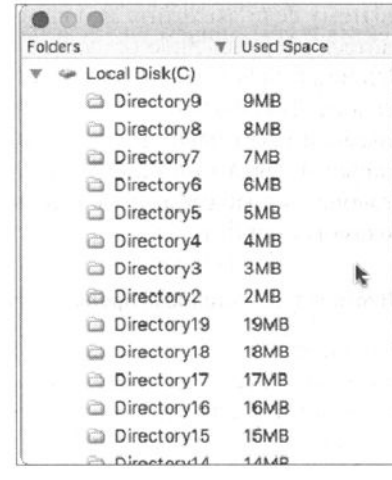


Рис. 11.4. Иерархический список

Листинг 11.4. Создание иерархического списка (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QTreeWidget  twg;
    QStringList  lst;

    lst << "Folders" << "Used Space";
    twg.setHeaderLabels(lst);
    twg.setSortingEnabled(true);

    QTreeWidgetItem* ptwgItem = new QTreeWidgetItem(&twg);
    ptwgItem->setText(0, "Local Disk(C)");
    ptwgItem->setIcon(0, QPixmap(":/drive.bmp"));

    QTreeWidgetItem* ptwgItemDir = 0;
    for (int i = 1; i < 20; ++i) {
        ptwgItemDir = new QTreeWidgetItem(ptwgItem);
        ptwgItemDir->setText(0, "Directory" + QString::number(i));
        ptwgItemDir->setText(1, QString::number(i) + "MB");
```

```
        ptwgItemDir->setIcon(0, QPixmap(":/folder.bmp"));
    }
    ptwgItem->setExpanded(true);
    twg.resize(350, 310);
    twg.show();

    return app.exec();
}
```

Обратите внимание на столбцы — они предоставляют намного больше возможностей, чем просто указание заголовка. При нажатии на заголовок столбца проводится сортировка элементов в убывающем или возрастающем порядке. Подобную сортировку можно разрешить или запретить, вызвав метод `setSortingEnabled()`. Для разрешения нужно передать в этот метод значение `true`, а для запрещения — `false`.

По умолчанию пользователь может одновременно выбирать из списка только один из элементов. Если этого недостаточно, то нужно вызвать метод `setSelectionMode()` базового класса `QAbstractItemView` с параметром `QAbstractItemView::MultiSelection`, который устанавливает режим множественного выделения. Для того чтобы «пройтись» по всем элементам виджета иерархического списка `QtreeWidget`, можно воспользоваться итератором, например:

```
QTreeWidgetItemIterator it(&twg, QTreeWidgetItemIterator::All);
while (*(++it)) {
    qDebug() << (*it)->text(0);
}
```

Обратите внимание, что в первом параметре в конструктор итератора был передан адрес самого виджета иерархического списка. Вторым параметром передается флаг, указывающий, какие виджеты должны приниматься во внимание при обходе. В нашем конкретном случае мы передали значение `QTreeWidgetItemIterator::All`, что осуществит обход всех элементов иерархического списка. Для обхода, например, только выделенных элементов в конструктор нужно передать значение `QTreeWidgetItemIterator::Selected`. Есть и другие флаги, их полный список можно посмотреть в документации.

Класс `QTreeWidget` содержит следующие сигналы:

- `itemSelectionChanged()` — сообщает об изменении выбранных элементов;
- `itemClicked(QTreeWidgetItem*, int)` — отправляется после щелчка на элементе;
- `itemDoubleClicked(QTreeWidgetItem*, int)` — отправляется при двойном щелчке мыши;
- `itemActivated(QTreeWidgetItem*, int)` — отправляется при двойном щелчке мыши, а также при нажатии клавиши <Enter> на элементе. В случаях, когда двойной щелчок мыши и нажатие клавиши <Enter> должны вызвать одно и то же действие, код можно реализовать в одном слоте и соединить его с этим сигналом.

Второй параметр последних трех сигналов содержит номер столбца, на котором произошел щелчок.

Класс `QTreeWidget` поддерживает технологию перетаскивания (drag & drop). Для ее реализации необходимо вызвать метод `setFlags()` для тех элементов, которым нужно включить поддержку перетаскивания с параметром `Qt::ItemIsDragEnabled`, возможно, скомбинированным с другими требуемыми значениями. Например:

```
pitem->setFlags(Qt::ItemIsDragEnabled | Qt::ItemIsEnabled);
```

По умолчанию режим переименования элементов отключен. Включить его можно, передав в метод `setFlags()` значение `Qt::ItemIsEditable` (но не забудьте указать и другие необходимые значения!).

Сортировка элементов

Так же как и в классе простого списка `QListWidget`, элементы списка можно упорядочить вызовом метода `sortItems()`, в который передаются значения для сортировки в убывающем или возрастающем порядке. Для сортировки по датам и числовым значениям необходимо унаследовать класс `QTreeWidgetItem` и перезаписать в нем `operator<()`. Его перезапись может выглядеть так, как это показано в листинге 11.5.

Листинге 11.5. Перезапись `QTreeWidgetItem::operator<()`

```
bool MyTreeWidgetItem::operator(const QTreeWidgetItem& ptwiOther)
{
    bool bRet    = false;
    int  nColumn = treeWidget()->sortColumn();
    if (nColumn == 0) {
        QString strFormat = "dd.MM.yyyy";
        bRet = QDate::fromString(text(nColumn))
               < QDate::fromString(ptwi.text(nColumn));
    }
    return bRet;
}
```

В листинге 11.5 мы исходим из того, что первым столбцом (столбец с индексом 0) будет столбец с датами. Вызовом метода `sortColumn()` мы запрашиваем индекс, по которому пользователь осуществляет сортировку, этот метод вызывается из виджета иерархического списка `QTreeWidget`. Если индекс равен 0, то мы переводим строку с датой к типу `QDate` и сравниваем значения дат текущего элемента с другим. После чего возвращаем результат (переменная `bRet`).

Таблицы

Класс `QTableWidget` представляет собой таблицу. Объект ячейки реализован в классе `QTableWidgetItem`. Созданную ячейку можно вставить в таблицу вызовом метода `QTableWidget::setItem()`. Первым параметром метода `setItem()` является номер строки, а вторым — номер столбца. Таким образом эти параметры задают местоположение ячейки в таблице. Установить текст в самой ячейке можно с помощью метода `QTableWidgetItem::setText()`, а для размещения растрового изображения — воспользоваться методом `QTableWidgetItem::setIcon()`. Если в ячейке установлены как текст, так и растровое изображение, то растровое изображение займет место слева от текста. Класс ячейки `QTableWidgetItem` предоставляет конструктор копирования, что позволяет создавать копии элементов. Также для этой цели можно воспользоваться методом `clone()`.

В таблицу допускается, помимо текста и значков, помещать и виджеты. Для этого служит метод `setCellWidget()`.

При двойном щелчке кнопкой мыши на поле ячейки она переходит в режим редактирования, используя при этом виджет `QLineEdit`.

Следующий пример (листинг 11.6) представляет таблицу, состоящую из трех столбцов и трех строк (рис. 11.5). Содержимое ячеек можно изменять.

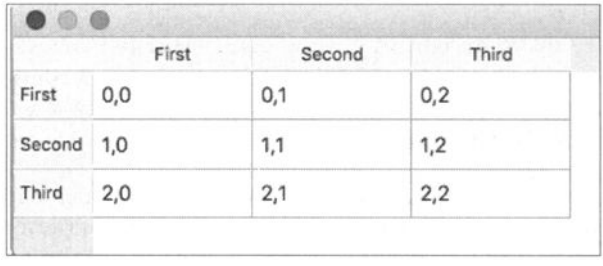


Рис. 11.5. Таблица

В листинге 11.6 создается виджет таблицы `tbl`. Первый и второй параметры, передаваемые в конструктор, задают количество строк и столбцов. Наша таблица будет иметь размерность 3 на 3. Чтобы задать заголовки строк и столбцов таблицы (которые у нас совпадают), мы сначала формируем их список, а затем передаем его в метод `setHorizontalHeaderLabel()` — для горизонтальных и в метод `setVerticalHeaderLabel()` — для вертикальных заголовков.

Создание объектов ячеек выполняется в цикле (указатель `ptwi`). В качестве текста в конструктор передаются номера строки и столбца ячейки. Вызовом метода `setItem()` созданная ячейка таблицы устанавливается в позицию, указанную в первом и втором параметрах.

Сортировка

Сортировка выполняется аналогичным способом — так же, как в классах `QListWidget` и `QTreeWidget`.

Листинг 11.6. Создание таблицы (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    const int n = 3;

    QApplication      app(argc, argv);
    QTableWidget      tbl(n, n);
    QTableWidgetItem* ptwi = 0;
    QStringList       lst;

    lst << "First" << "Second" << "Third";
    tbl.setHorizontalHeaderLabels(lst);
    tbl.setVerticalHeaderLabels(lst);

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {
            ptwi = new QTableWidgetItem(QString("%1,%2").arg(i).arg(j));
```

```
            tbl.setItem(i, j, ptwi);
        }
    }
    tbl.resize(370, 135);
    tbl.show();

    return app.exec();
}
```

Выпадающий список

Класс `QComboBox` предоставляет пользователю возможность выбора одного элемента из нескольких. Его функциональное назначение совпадает с виджетом простого списка `QListWidget`. Основное преимущество выпадающего списка состоит в отображении только одного (выбранного) элемента, благодаря чему для его размещения не требуется много места. Отображение всего списка (раскрытие) происходит только на некоторый промежуток времени, чтобы пользователь мог сделать выбор, а затем список возвращается в свое исходное состояние (сворачивается).

В качестве элемента можно добавить текст и/или картинку. Для этого служит метод `addItem()`. Можно добавить сразу несколько текстовых элементов, передав указатель на объект класса `QStringList` в метод `addItems()`. Вызвав метод `setDuplicatesEnabled(false)`, можно включить режим, исключающий повторяющиеся элементы из списка. Если необходимо удалить все элементы выпадающего списка, тогда вызывается слот `clear()`.

Чтобы узнать, какой из элементов является текущим, нужно вызвать метод `currentIndex()`, который возвратит его порядковый номер. Можно сделать так, чтобы пользователь мог сам добавлять элементы в список. Типичным примером этого является адресная строка Проводника ОС Windows, содержащая в себе список просмотренных адресов (ссылок). Для установки виджета в этот режим вызывается метод `setEditable()` с параметром `true`. После изменения пользователем текста выбранного элемента отправляется сигнал `editTextChanged(const QString&)`, и новый элемент добавляется в список.

После выбора элемента отправляются сразу два сигнала `activated()`: один с параметром типа `int` (индексом выбранного элемента), а другой — с параметром типа `const QString&` (его значением). Эти сигналы отправляются, даже если пользователь выбрал ранее выбранный элемент, — для информирования о реальном изменении служат два сигнала `currentIndexChanged()`, также отправляемые с параметрами `int` и `const QString&` каждый.

Следующий пример (листинг 11.7) демонстрирует виджет выпадающего списка (рис. 11.6). При изменении элемента список дополнится новым.

В программе, приведенной в листинге 11.7, создается виджет выпадающего списка `cbo`. Затем в список `lst` добавляются четыре строки. Эти четыре строки устанавливаются вызовом

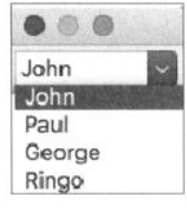


Рис. 11.6. Выпадающий список

метода `addItems()` в виджете выпадающего списка. Вызов метода `setEditable()` с параметром `true` переводит список в режим редактирования.

> ***При необходимости используйте контролёр***
>
> Если при редактировании текста требуется проверять правильность ввода информации, в виджете `QComboBox` следует установить объект класса `QValidator` (см. *главу 10*).

Листинг 11.7. Создание выпадающего списка (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QComboBox    cbo;
    QStringList  lst;

    lst << "John" << "Paul" << "George" << "Ringo";
    cbo.addItems(lst);
    cbo.setEditable(true);
    cbo.show();

    return app.exec();
}
```

Вкладки

При чтении книг вы, возможно, используете закладки, которые помогают вам быстро открыть книгу в нужном месте. Ситуация с компьютерными вкладками аналогична, с той лишь разницей, что они служат для быстрой смены диалоговых окон. При выборе вкладки в окне отображается закрепленный за ней виджет. Вкладки могут содержать как текст, так и растровое изображение.

Основное назначение вкладок — разгрузить сложное диалоговое окно, имеющее большое количество опций, разделив его на серию логически скомпонованных диалоговых подокон.

Вкладки можно делать доступными и недоступными. Чтобы сделать вкладку недоступной, нужно вызвать метод `setTabEnabled()` и передать ему значение `false`. Вызовом слота `setCurrentIndex()` можно сделать вкладку текущей.

Следующий пример (листинг 11.8) организует четыре вкладки: **Linux**, **Windows**, **MacOSX** и **Android** (рис. 11.7). При выборе каждой из вкладок происходит отображение виджета, закрепленного за ней.

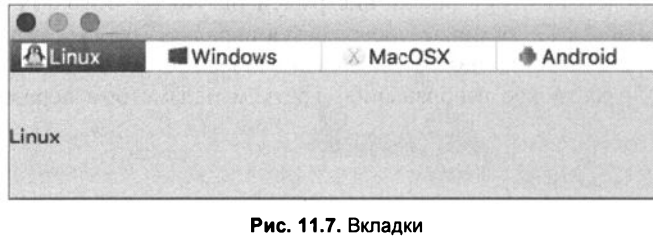


Рис. 11.7. Вкладки

В листинге 11.8 создается виджет вкладок `tab`. Вызовом метода `addTab()` в цикле добавляются новые вкладки. В первом параметре этого метода передается указатель на виджет, который должен отображаться при выборе вкладки, во втором — растровое изображение, а в третьем — текст вкладки.

Листинг 11.8. Организация вкладок (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QTabWidget   tab;
    QStringList  lst;

    lst << "Linux" << "Windows" << "MacOSX" << "Android";
    foreach(QString str, lst) {
        tab.addTab(new QLabel(str, &tab), QPixmap(":/" + str + ".jpg"), str);
    }
    tab.resize(420, 100);
    tab.show();

    return app.exec();
}
```

Виджет панели инструментов

Класс `QToolBox` представляет собой вкладки, расположенные вертикально. Связанные (с вкладками) виджеты отображаются непосредственно под ними. Текст вкладки добавляется вместе с виджетом при вызове метода `addItem()`. Если требуется вставить вкладку на определенную позицию, то вызывается метод `insertItem()`. Количество вкладок можно узнать, вызвав метод `count()`. Для удаления вкладок реализован метод `removeItem()`.

Вызвав метод `currentWidget()`, можно получить указатель на закрепленный за текущей вкладкой виджет.

Виджет панели инструментов располагает только одним сигналом `currentChanged(int)`, отсылаемым при выборе одной из вкладок.

В следующем примере (листинг 11.9) реализована панель инструментов, содержащая четыре вкладки с закрепленными за ними виджетами надписей (рис. 11.8).

В листинге 11.9 создается виджет панели инструментов. После этого в него с помощью метода `addItem()` в цикле добавляются вкладки. Первым параметром передается указатель на виджет, который отображается при выборе вкладки. Вторым — растровое изображение. Третьим параметром передается текст вкладки.

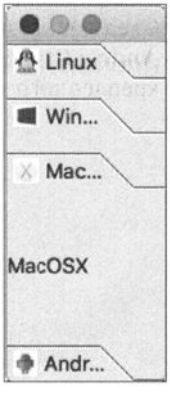


Рис. 11.8. Панель инструментов

Листинг 11.9. Создание панели инструментов (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QToolBox     tbx;
    QStringList  lst;

    lst << "Linux" << "Windows" << "MacOSX" << "Android";
    foreach(QString str, lst) {
        tbx.addItem(new QLabel(str, &tbx), QPixmap(":/" + str + ".jpg"), str);
    }
    tbx.resize(100, 80);
    tbx.show();

    return app.exec();
}
```

Резюме

Элементы выбора могут содержать как текст, так и растровые изображения. Сами элементы можно выбирать с помощью левой кнопки мыши или клавиш управления курсором.

Базой для классов `QListWidget`, `QTreeWidget` и `QTableWidget` являются классы архитектуры «модель-представление», с которыми мы познакомимся в следующей главе.

Вкратце классы, рассмотренные в этой главе, можно охарактеризовать следующим образом:

- класс `QListWidget` удобен для выбора от одного до нескольких элементов. Этот класс можно переключать в режим пиктограмм (представления в виде значков);
- класс `QTreeWidget` предназначен для отображения элементов в иерархическом виде;
- класс `QTableWidget` представляет собой таблицу, в ячейки которой можно помещать текст и растровые изображения;
- класс `QComboBox` позволяет выбрать только один элемент. Основное его преимущество состоит в том, что он не требует много места;
- класс `QTabWidget` позволяет разбить сложное диалоговое окно на серию простых диалоговых окон, благодаря чему приложение становится более понятным;
- класс `QToolBox` по сути очень похож на класс `QTabWidget`. Разница состоит в вертикальном расположении вкладок.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/11-510/** или с помощью следующего QR-кода (рис. 11.9):



Рис. 11.9. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 12

Интервью, или модель-представление

Вкусовых ощущений только пять, но вкусовых сочетаний так много, что никому не суждено познать их все.
Сунь Цзы

Использование элементно-ориентированного подхода, описанного в *главе 11*, не всегда представляется оптимальным. Такой подход идеален для простых ситуаций, когда нужно отобразить небольшой объем данных. Но в более сложных ситуациях, таких как, например, работа с базами данных, файловой системой и т. п., ориентироваться на этот подход не рекомендуется — из соображений эффективности и расхода памяти. Представьте себе, что для получения результатов SQL-запросов вам придется записывать их в элементы и тем самым дублировать данные. А при использовании трех виджетов, показывающих эти данные, объем дублирования утроится, и, кроме того, вам нужно будет при отображении этих данных решать проблему синхронизации.

Qt предоставляет технологию, называемую *«интервью»*, или, иначе, *«модель-представление»*. Наверняка, многие из читателей уже знакомы с шаблоном проектирования «модель-представление». Очень важно понимать, что архитектура «модель-представление», реализованная в Qt, не является прямой реализацией этого шаблона, а использует только основные его идеи, такие как, например, отделение данных от их представления. Применение технологии «интервью» дает следующие преимущества:

- *возможность показа данных в нескольких представлениях без дублирования.* Если вы работаете на основе элементно-ориентированного подхода и вам необходимо добавить новые элементы, то при синхронизации отображения с данными происходит дублирование самих данных. В подходе «модель-представление» мы работаем с моделью данных, поэтому дублирования не происходит;
- *возможность внесения изменений с минимумом временных затрат.* Например, представим себе, что в программе полностью изменился способ сохранения данных. Но это не станет помехой, так как связь с данными осуществляется с помощью интерфейса, и до тех пор, пока сам интерфейс останется нетронутым, это не повлечет за собой больших изменений в коде программы;
- *удобство программного кода.* Поскольку осуществляется разделение на данные и представление, то, если появится необходимость что-то дополнить или исправить, эти изменения коснутся лишь одной из частей кода. Остальные части вашего приложения останутся без изменений;
- *удобство тестирования кода.* Как только интерфейс задан, можно написать тест, который может быть использован для любой модели, реализующей этот интерфейс. Qt предоставляет специальную библиотеку для проведения тестов модулей (см. *главу 45*);

- *упрощение интеграции баз данных*. Эта же модель применяется Qt и для SQL, чтобы сделать интеграцию баз данных проще для программистов, не связанных с разработкой баз данных (см. *главу 41*).

Концепция

Все части технологии «интервью» могут взаимодействовать друг с другом в соответствии с направлениями стрелок, показанными на рис. 12.1.



Рис. 12.1. Взаимодействие компонентов «интервью»

Давайте сначала разберемся в назначениях частей этой технологии, а затем перейдем к их подробному рассмотрению. Вот ее основные составляющие:

- *модель* — отвечает за управление данными и предоставляет интерфейс для чтения и записи данных;
- *представление* — отвечает за представление данных пользователю и за их расположение;
- *выделение элемента* — специальная модель, отвечающая за централизованное использование выделений элементов;
- *делегат* — отвечает за рисование каждого элемента в отдельности, а также за его редактирование.

Классы представлений практически всегда используются как есть. Наследовать чаще всего приходится от классов моделей и иногда от классов делегатов.

Модель

Модель — это оболочка вокруг исходных данных, предоставляющая стандартный интерфейс для доступа к ним. Так как именно интерфейс модели является основной единицей, обеспечивающей связь между моделью и представлением, это дает дополнительные пре-

имущества, а именно: модели можно разрабатывать отдельно друг от друга и при необходимости заменять одну на другую. Интерфейс любой Qt-модели базируется на классе `QAbstractItemModel` (рис. 12.2). Для того чтобы создать свою собственную модель, вам придется унаследовать либо этот класс, либо один из его потомков.

Сам класс `QAbstractItemModel` представляет собой обобщенную таблицу, за каждой ячейкой которой может быть закреплена подтаблица. Благодаря этому свойству можно создавать модели для сложных структур данных. Например, для древовидной структуры, описывающей содержимое каталога, некоторая ячейка, находящаяся в строке и представляющая каталог, будет иметь подтаблицу, строки которой будут соответствовать файлам и подкаталогам. Подкаталоги, в свою очередь, могут также иметь подтаблицы со строками, в которых будут появляться файлы и подкаталоги и т. д.

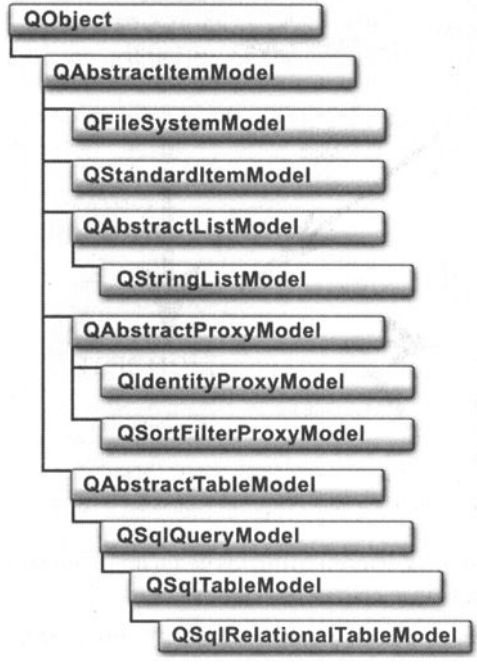


Рис. 12.2. Иерархия классов модели

Рассмотрим некоторые унаследованные от `QAbstractItemModel` классы, показанные на рис. 12.2:

- класс `QAbstractListModel` представляет собой одномерный список, а класс `QAbstractTableModel` — двумерную таблицу;
- класс `QStandardItemModel` позволяет напрямую сохранять данные в модели. Хоть это и немного противоречит основной идее «модель-представление», но в некоторых приложениях, которые манипулируют незначительным количеством данных, является довольно удобным и практичным компромиссом;
- класс `QStringListModel` — это реализация `QAbstractListModel`, которая предоставляет одномерную модель, предназначенную для работы со списком строк. Список строк (`QStringList`) — здесь источник данных. Эта модель предоставляет возможность редактирования, то есть если пользователь с помощью представления изменит одну из записей, то старая запись будет замещена новой. Каждая запись соответствует одной строке.

Например:

```
QStringListModel model;
model.setStringList(QStringList() << "Первая строка"
                                  << "Вторая строка"
                                  << "Третья строка"
                    );
```

- основная идея класса `QAbstractProxyModel` состоит в извлечении данных из модели, проведении некоторых манипуляций с ними и возвращении их в качестве новой модели. Таким образом можно осуществлять выборку и сортировку данных. Для проведения указанных операций можно воспользоваться унаследованным классом `QSortFilterProxyModel`. Этот подход будет подробно рассмотрен здесь далее;
- класс `QFileSystemModel` представляет собой готовый класс иерархии файловой системы.

Данные, предоставляемые моделями, могут посредством интерфейса (рис. 12.3) совместно использоваться различными представлениями (виджетами, унаследованными от `QAbstarctItemView`). Для того чтобы модель и представление могли понимать друг друга, модель информирована об основных свойствах представления: каждая запись занимает в ней одну строку и столбец, а также может иметь индекс, который играет важную роль во вложенных структурах.



Рис. 12.3. Связь модели с различными представлениями

Представление

Как можно видеть на рис. 12.4, базовым классом подавляющего большинства классов представлений является класс `QAbstactScrollArea`, что позволяет в тех случаях, когда отображаемая информация занимает больше места, чем область показа, воспользоваться полосами прокрутки.

> ***Класс QComboBox***
>
> Представлением может также являться и класс `QComboBox`, который напрямую унаследован от класса `QWidget`. Класс `QComboBox` предоставляет метод `setModel()` для установки моделей, как и все далее описанные классы представлений.

Классы представлений наследуются от класса `QAbstractItemView`, который дает для всех представлений такие базовые возможности, как, например, установка моделей в представлении, методы для прокрутки изображения и многие другие. Этот класс содержит метод `setEditTriggers()`, задающий параметры переименования элементов.

В этот метод можно передать следующие значения:

- `NoEditTriggers` — переименование невозможно;
- `DoubleClicked` — переименовать, если на элементе был осуществлен двойной щелчок мышью;

- `SelectedClicked` — переименовать, если произошел щелчок мышью на выбранном элементе.

Для представления данных в Qt используются, в основном, три класса:

- `QListView` — представляет собой одномерный список. Этот класс также располагает режимом пиктограмм (отображения значков);
- `QTreeView` — отображает иерархические списки. Этот класс также способен отображать столбцы;
- `QTableView` — отображает данные в виде таблицы.

Заметьте, что класс `QHeaderView` унаследован непосредственно от `QAbstractItemView`. Но он не предназначен для самостоятельного отображения данных, а используется совместно с классами `QTableView` и `QTreeView` для отображения заголовков столбцов и строк.

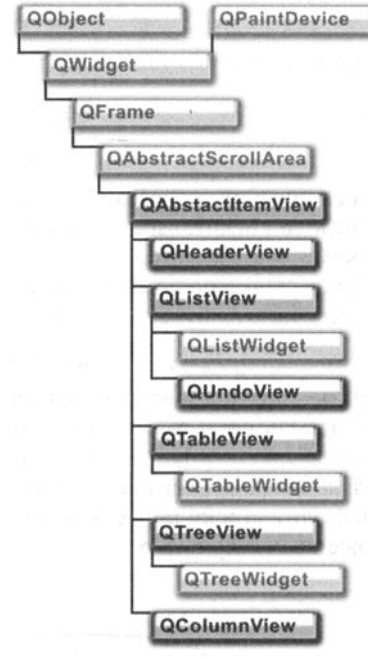


Рис. 12.4. Классы представлений (выделены полужирным шрифтом)

Выделение элемента

Обычно в представлениях имеется механизм, управляющий выделением элементов, — то есть каждое представление реализует свое собственное выделение элементов в отдельной части кода. Это неудобно, поскольку для большого количества представлений этот код может быть разбросан по разным частям программы. Описанный далее механизм позволяет реализовать выделение элемента централизованно, в одном месте. Таким образом, мы получаем возможность разделения между различными представлениями, работающими с одной моделью данных, не только собственно ее данных, но и механизма выделения.

Управление выделением осуществляется при помощи специальной модели, реализованной в классе `QItemSelectionModel` (рис. 12.5). Для получения модели выделения элементов, установленной в представлении, нужно вызвать метод `QAbstractItemView::selectionModel()`, а установить новую модель можно с помощью метода `QAbstractItemView::setSelectionModel()`.

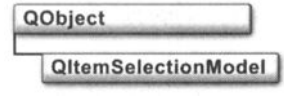


Рис. 12.5. Класс выделения `QItemSelectionModel`

Программа (листинг 12.1), окно которой показано на рис. 12.6, выполняет разделение выделения элементов между тремя представлениями. Выделение элемента в одном из представлений приведет к выделению этого же элемента и в остальных представлениях.

В листинге 12.1 мы создаем модель списка строк (объект `model`), которую инициализируем тремя элементами. Каждый элемент является строкой. Затем создаем три разных представ-

ления (указатели pTreeView, pListView и pTableView) и устанавливаем в них нашу модель, вызывая метод setModel(). Самый важный момент — создание модели выделения (объекта класса QItemSelectionModel). При создании этот объект инициализируется оригинальной моделью (объект model). После этого модель выделения устанавливается вызовом метода setSelectionModel() во всех трех объектах представления.

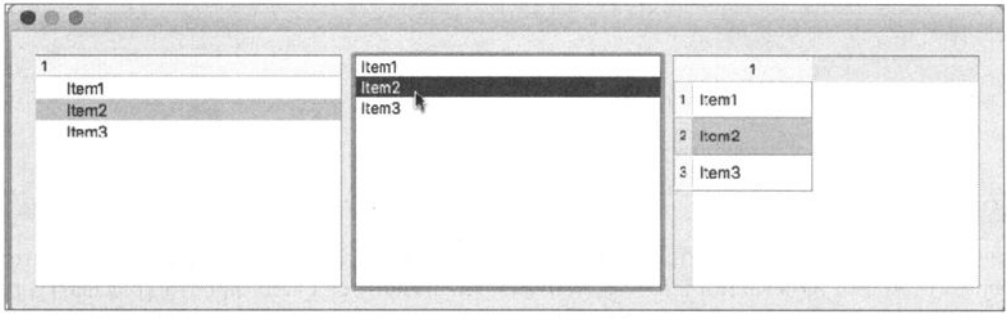


Рис. 12.6. Демонстрация разделения выделения элементов между представлениями

Индексы текущих выделенных позиций можно получить вызовом метода QItemSelectionModel::selectedIndexes(). А выделять элементы программно можно с помощью метода QItemSelectionModel::select(). При изменениях выделения модель выделений отсылает сигналы currentChanged(), selectionChanged(), currentColumnChanged() и currentRowChanged().

Листинг 12.1. Разделение выделения элементов между тремя представлениями (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QStringListModel model;
    model.setStringList(QStringList() << "Item1" << "Item2" << "Item3");

    QTreeView* pTreeView = new QTreeView;
    pTreeView->setModel(&model);

    QListView* pListView = new QListView;
    pListView->setModel(&model);

    QTableView* pTableView = new QTableView;
    pTableView->setModel(&model);

    QItemSelectionModel selection(&model);
    pTreeView->setSelectionModel(&selection);
    pListView->setSelectionModel(&selection);
    pTableView->setSelectionModel(&selection);
```

```
    //Layout setup
    QHBoxLayout* phbxLayout = new QHBoxLayout;
    phbxLayout->addWidget(pTreeView);
    phbxLayout->addWidget(pListView);
    phbxLayout->addWidget(pTableView);
    wgt.setLayout(phbxLayout);

    wgt.show();

    return app.exec();
}
```

Делегат

Для стандартных представлений списков и таблиц отображение элементов выполняется посредством так называемого *делегирования*. Это позволяет очень просто создавать представления для любых нужд без написания большого количества нового кода. Делегат отвечает за рисование каждого элемента и за его редактирование (изменение пользователем). В Qt есть готовый класс делегата `QStyledItemDelegate` (рис. 12.7), который предоставляет методы редактирования каждой записи при помощи элемента однострочного текстового поля, и для большинства случаев его вполне достаточно. Но если вам потребуется осуществлять особый контроль над отображением и редактированием данных, то понадобится создать свой собственный делегат. Для этого необходимо унаследовать свой класс либо от `QAbstractItemDelegate`, либо от `QStyledItemDelegate`.

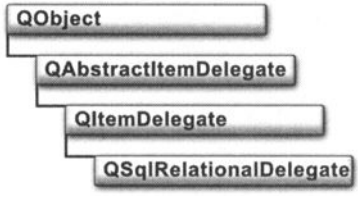


Рис. 12.7. Классы делегатов

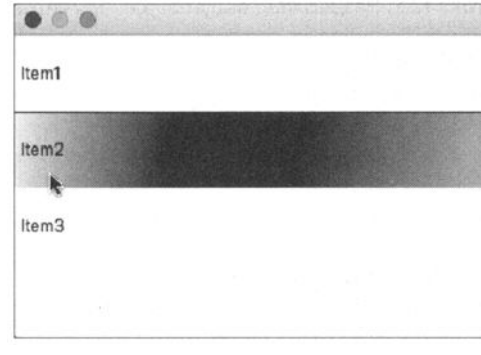


Рис. 12.8. Демонстрация делегата

Давайте реализуем простой пример делегата (листинг 12.2), который выделяет элемент, как только пользователь проведет над ним курсор (указатель) мыши (рис. 12.8).

В функции `main()`, приведенной в листинге 12.2, вызов метода `setItemDelegate()` устанавливает в представлении объект созданного нами делегата (класс `SimpleDelegate`).

Для того чтобы представление реагировало на перемещения курсора мыши над ним, необходимо в окне просмотра при помощи метода `setAttribute()` установить флаг `Qt::WA_Hover`.

В классе `SimpleDelegate` в методе для рисования `paint()` мы получаем три аргумента. Первый аргумент — это указатель на объект класса `QPainter`. Второй — это ссылка на структуру `QStyleOptionViewItem`, определенную в классе `QStyle`. Третий — модельный индекс,

который будет рассмотрен в следующем разделе. Для перерисовки элемента каждый раз, когда пользователь проведет над ним мышь, мы просто проверяем флаги состояния установленных битов объекта структуры `option`, чтобы определить, находится ли мышь на элементе или нет. Если биты флага `QStyle::State_MouseOver` установлены, это значит, что курсор мыши находится над элементом, и в этом случае его фон рисуется при помощи линейного градиента (рисование градиентом рассмотрено в *главе 18*). Перегрузив метод `sizeHint()`, мы можем управлять величиной области каждого элемента — в нашем примере мы делегируем ширину по умолчанию из объекта класса `QStyleOptionViewItem`, а высоту устанавливаем равной 55 пикселам.

Если бы мы захотели изменить стандартный способ редактирования, то нам пришлось бы реализовать в унаследованном классе методы `createEditor()`, `setEditorData()` и `setModelData()`. Метод `createEditor()` создает виджет для редактирования. Метод `setEditorData()` устанавливает данные в виджете редактирования. Метод `setModelData()` извлекает данные из виджета редактирования и передает их модели.

Листинг 12.2. Реализация простого делегата (файл main.cpp)

```
#include <QtWidgets>
// =====================================================================
class SimpleDelegate : public QStyledItemDelegate {
public:
    SimpleDelegate(QObject* pobj = 0) : QStyledItemDelegate(pobj)
    {
    }

    void paint(QPainter*                    pPainter,
               const QStyleOptionViewItem& option,
               const QModelIndex&          index
              ) const
    {
        if (option.state & QStyle::State_MouseOver) {
            QRect           rect = option.rect;
            QLinearGradient gradient(0, 0, rect.width(), rect.height());

            gradient.setColorAt(0, Qt::white);
            gradient.setColorAt(0.5, Qt::blue);
            gradient.setColorAt(1, Qt::green);
            pPainter->setBrush(gradient);
            pPainter->drawRect(rect);
        }
        QStyledItemDelegate::paint(pPainter, option, index);
    }

    QSize sizeHint(const QStyleOptionViewItem& option,
                   const QModelIndex& /*index*/
                  ) const
    {
        return QSize(option.rect.width(), 55);
    }
};
```

```
// ------------------------------------------------------------------------
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QStringListModel model;
    model.setStringList(QStringList() << "Item1" << "Item2" << "Item3");

    QListView listView;
    listView.setModel(&model);
    listView.setItemDelegate(new SimpleDelegate(&listView));
    listView.viewport()->setAttribute(Qt::WA_Hover);
    listView.show();

    app.exec();
}
```

Индексы модели

Итак, мы определили структуру данных, представляющую собой таблицу. Теперь нам нужен способ для получения доступа к каждому ее элементу. Например, для двумерной таблицы без иерархии это означает, что мы могли бы использовать строку и столбец, чего было бы достаточно. Но при использовании иерархии нам понадобится другое решение, поскольку в этом случае требуется иметь относительно строки и столбца некоторую дополнительную информацию. Эта информация и будет индексом модели.

Индекс модели — это небольшой список объектов, который служит для адресации ячейки в таблице, имеющей иерархию. Индекс модели представляет собой информацию, состоящую из трех частей: строки, столбца и внутреннего идентификатора. Внутренний идентификатор зависит от реализации — это может быть указатель или, например, целочисленный индекс.

Каждая ячейка таблицы имеет уникальный индекс, который представлен классом `QModelIndex`. Индексы класса `QModelIndex` запоминать в программе не имеет смысла, так как они могут измениться, например, после сортировки. Однако ими удобно пользоваться для получения текущих значений ячеек таблицы с помощью метода `QAbstractItemModel::data()`. Получить индекс модели для любой ячейки можно методом `QAbstractItemModel::index()`, который позволяет двигаться по всей структуре данных. Например, для того чтобы узнать значение ячейки с координатами (2, 5), нужно проделать следующее:

```
QModelIndex index = pModel->index(2, 5, QModelIndex());
QVariant value = pModel->data(index);
```

Может получиться так, что подтаблица не располагает элементом с заданными нами координатами (2, 5). В этом случае метод `index()` возвратит пустой индекс (invalid index). Является ли индекс действительно неверным или пустым, можно проверить вызовом метода `QModelIndex::isValid()`.

> **Метод data() класса QModelIndex**
>
> Класс `QModelIndex` имеет метод `data()` — это очень удобно, поскольку, чтобы получить доступ к данным, достаточно иметь только объект этого класса, и вовсе не обязательно обращаться к его модели.

Иерархические данные

Каждая ячейка в таблице может иметь дочерние таблицы. И думая об иерархиях, мы должны не забыть об иерархиях таблиц. Давайте воспользуемся интерфейсом модели `QStandardItemModel` для создания нашей иерархии и отобразим ее (рис. 12.9).

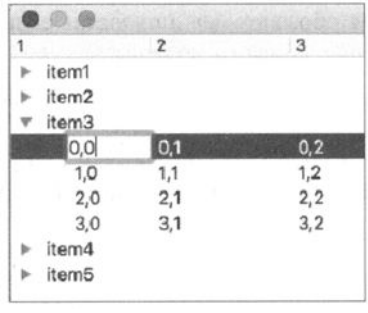


Рис. 12.9. Отображение иерархических данных

Схема использования класса `QStandardItemModel` очень проста (листинг 12.3). Сначала необходимо создать его объект, а потом методом `setData()` установить данные для каждого элемента. Впоследствии эти данные можно будет получать методом `data()`.

Листинг 12.3. Отображение иерархических данных (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication       app(argc, argv);
    QStandardItemModel model(5, 3);

    for (int nTopRow = 0; nTopRow < 5; ++nTopRow) {
        QModelIndex index = model.index(nTopRow, 0);
        model.setData(index, "item" + QString::number(nTopRow + 1));

        model.insertRows(0, 4, index);
        model.insertColumns(0, 3, index);
        for (int nRow = 0; nRow < 4; ++nRow) {
            for (int nCol = 0; nCol < 3; ++nCol) {
                QString strPos = QString("%1,%2").arg(nRow).arg(nCol);
                model.setData(model.index(nRow, nCol, index), strPos);
            }
        }
    }

    QTreeView treeView;
    treeView.setModel(&model);
    treeView.show();

    return app.exec();
}
```

В примере, приведенном в листинге 12.3, мы создаем модель, представляющую собой таблицу из пяти строк и трех столбцов. В цикле получаем текущий индекс с помощью метода `index()` и задаем данные для элемента. Затем вставляем по текущему индексу подтаблицу с четырьмя строками и тремя столбцами при помощи методов `insertRows()` и `insertColumns()`. Во вложенных циклах вызовом метода `setData()` ячейки подтаблицы одна за другой заполняются данными.

Приведенный пример не является хорошим образцом манипуляции с данными, но дает представление о том, как выполнять заполнение данными модели `QStandardItemModel`.

Теперь рассмотрим пример, успевший стать с момента появления Qt 4 настоящей классикой. Он как нельзя лучше показывает простоту и потенциал, связанные с использованием моделей. В программе (листинг 12.4), окно которой показано на рис. 12.10, благодаря готовой модели `QFileSystemModel` всего лишь при помощи нескольких строк реализуется обозреватель файловой системы.

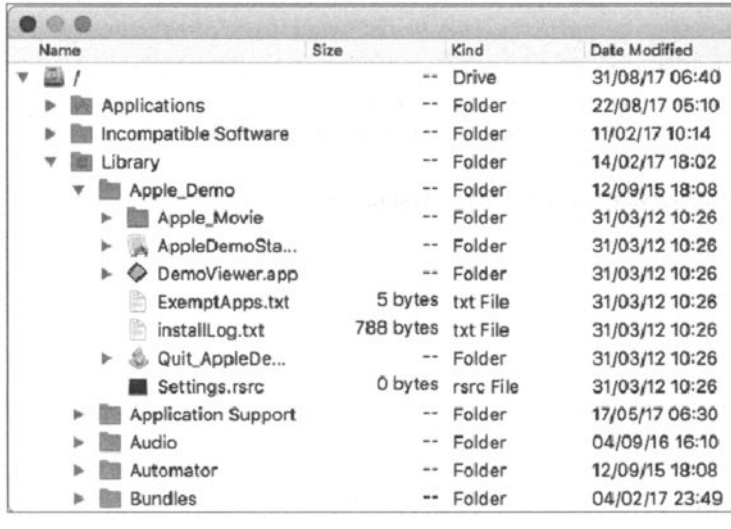


Рис. 12.10. Обозреватель файловой системы: показаны каталоги и файлы

В листинге 12.4 создается и устанавливается в представлении объект модели класса `QFileSystemMode`. И приложение готово! В принципе, эту модель можно устанавливать в любом представлении, но для отображения иерархических данных и навигации лучше всего подойдет `QTreeView`.

Листинг 12.4. Отображение каталогов и файлов (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication     app(argc, argv);
    QFileSystemModel model;
    QTreeView        treeView;
```

```
    model.setRootPath(QDir::rootPath());
    treeView.setModel(&model);
    treeView.show();

    return app.exec();
}
```

Для показа определенного пути можно воспользоваться методом `index()`. Обычно этот метод ожидает три параметра: столбец, строку и индекс предка, но можно обойтись и без них — достаточно передать в него строку с путем. Таким образом, добавив в нашу программу (см. листинг 12.4) следующие далее строки, мы увидим в нашем представлении содержимое только текущего каталога:

```
QModelIndex index = model.index(QDir::currentPath());
treeView.setRootIndex(index);
```

Воспользовавшись слотом `setRootIndex()` и еще несколькими слотами и сигналами, можно соединить иерархическое представление с табличным представлением и реализовать программу обозревателя (листинг 12.5), окно которого показано на рис. 12.11.

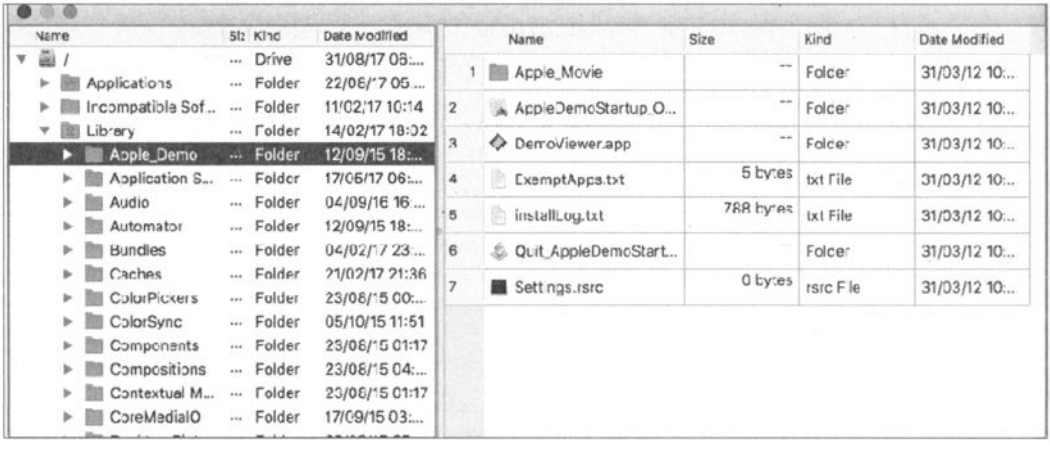


Рис. 12.11. Окно программы обозревателя из листинга 12.5: показано содержимое текущего каталога

Первые строки листинга 12.5 идентичны листингу 12.4 с той лишь разницей, что здесь дополнительно создается виджет разделителя (`spl`) и табличное представление (указатель `pTableView`). Оба представления разделяют одну и ту же модель (`model`). Основа реализации нашей программы заключается в сигнально-слотовых соединениях. Первое соединение в табличном представлении осуществляет установку в качестве узлового каталога, выбранного в иерархическом представлении. Второе соединение нам нужно для того, чтобы при выборе одного из каталогов табличного представления выполнялось выделение этого каталога в иерархическом представлении. Последнее соединение служит для показа содержимого каталога при работе в табличном представлении. Таким образом, двойной щелчок мыши или нажатие на клавишу <Enter> на каталоге вышлет из табличного представления сигнал `activated()`, который будет отловлен самим табличным представлением, и слот `setRootIndex()` установит этот каталог в качестве базового. А это значит, что табличное представление позволяет нам входить только внутрь каталогов, а не выходить из них. Но это не проблема — например, правая часть Проводника ОС Windows работает аналогичным

образом. Воспользовавшись же левой частью Проводника, мы можем выбрать любой интересующий нас каталог.

Листинг 12.5. Программа-обозреватель (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication     app(argc, argv);
    QSplitter        spl(Qt::Horizontal);
    QFileSystemModel model;

    model.setRootPath(QDir::rootPath());

    QTreeView* pTreeView = new QTreeView;
    pTreeView->setModel(&model);

    QTableView* pTableView = new QTableView;
    pTableView->setModel(&model);

    QObject::connect(pTreeView,  SIGNAL(clicked(const QModelIndex&)),
                     pTableView, SLOT(setRootIndex(const QModelIndex&))
                    );
    QObject::connect(pTableView, SIGNAL(activated(const QModelIndex&)),
                     pTreeView, SLOT(setCurrentIndex(const QModelIndex&))
                    );
    QObject::connect(pTableView, SIGNAL(activated(const QModelIndex&)),
                     pTableView, SLOT(setRootIndex(const QModelIndex&))
                    );

    spl.addWidget(pTreeView);
    spl.addWidget(pTableView);
    spl.show();

    return app.exec();
}
```

Роли элементов

Благодаря индексу (`QModelIndex`) модель может ссылаться на нужные данные и, тем самым, передать эти данные представлению. Для того чтобы данные были правильно показаны на экране, представление обращается посредством объекта индекса (`QModelIndex`) к так называемым *ролям*.

Каждый элемент в модели может содержать различные данные, которые привязаны к разным значениям ролей. Данные заданной роли можно получить с помощью метода `QAbstractItemModel::data()`, передав в него индекс и значение нужной роли, — например, `DisplayRole`. Если для заданной роли не будет найдено соответствующего значения, то метод `data()` возвратит объект класса `QVariant`, не содержащий никаких данных.

Элементы, помимо текста, могут иметь и растровое изображение, а также и дополнительный текст. Нам нужно каким-либо образом обращаться к этим данным. Разумеется, когда

мы рисуем элементы, то используем делегат и можем не пользоваться дополнительной информацией, которая заложена в элементе посредством ролей.

Существующие представления и делегаты понимают много ролей. Вот наиболее часто используемые из них:

- `DisplayRole` — текст для показа:
- `DecorationRole` — растровое изображение;
- `FontRole` — шрифт для текста;
- `ToolTipRole` — текст для подсказки (ToolTip);
- `WhatThisRole` — текст для подсказки «Что это?»;
- `TextColorRole` — цвет текста;
- `BackgroundColorRole` — цвет фона элемента.

Можно задать и свои собственные роли для своих классов моделей, например, так:

```
class GeoModel : public QAbstractListModel {
Q_OBJECT
public:
    enum GeoRoles
    {
        DistanceRole = Qt::UserRole + 1,
        LatitudeRole,
        LongitudeRole
    };
...
};
```

Следующий пример (листинг 12.6) демонстрирует применение ролей и показывает установку ролей `Qt::DisplayRole`, `Qt::ToolTipRole` и `Qt::DecorationRole` (рис. 12.12).

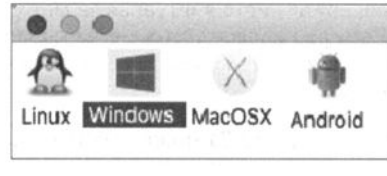


Рис. 12.12. Демонстрация ролей

В листинге 12.6 для отображения текста мы создаем список строк с использованием роли `Qt::DisplayRole`, а для отображения всплывающей подсказки применяем роль `Qt::ToolTipRole`. С ролью декорации `Qt::DecorationRole` добавляется растровое изображение, на которое мы ссылаемся посредством конкатенации строки с `".jpg"`. В результате получается элемент с текстом, растровым изображением и всплывающей подсказкой.

Листинг 12.6. Применение и установка ролей (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
```

```
    QStringList  lst;
    lst << "Linux" << "Windows" << "MacOS" << "OS2";

    QStandardItemModel model(lst.size(), 1);
    for (int i = 0; i < model.rowCount(); ++i) {
        QModelIndex index = model.index(i, 0);
        QString     str   = lst.at(i);
        model.setData(index, str, Qt::DisplayRole);
        model.setData(index, "ToolTip for " + str, Qt::ToolTipRole);
        model.setData(index, QIcon(str + ".jpg"), Qt::DecorationRole);
    }

    QListView listView;
    listView.setViewMode(QListView::IconMode);
    listView.setModel(&model);
    listView.show();

    return app.exec();
}
```

Создание собственных моделей данных

Как уже было сказано ранее, для создания своей собственной модели нужно унаследовать либо класс `QAbstractItemModel`, либо один из унаследованных от него классов. Обычно это могут быть `QAbstractListModel` или `QAbstractTableModel`. Первый класс представляет собой одномерный список, второй — двумерную таблицу.

> ***Наследование классов, содержащих чисто виртуальные методы***
>
> Наследовать классы `QAbstractItemModel`, `QAbstractListModel` и `QAbstractTableModel` нужно обязательно, так как они содержат чисто виртуальные методы, и произвести от них объект напрямую не получится. Есть два класса, которые не содержат чисто виртуальных методов, — это `QStandardItemModel` и `QStringListModel` — от них можно производить объекты, не наследуя их (см. листинг 12.3).

На диаграмме классов моделей (см. рис. 12.2) можно найти класс модели для списка строк, но модели для списка целых чисел там нет. Давайте устраним этот недостаток и реализуем такую модель. Программа (листинг 12.7), выполнение которой показано на рис. 12.13 и которую нам предстоит реализовать, осуществляет отображение данных модели целых чисел.

В листинге 12.7 при создании объекта нашей модели мы в конструктор передаем список из пяти чисел. Затем устанавливаем созданную модель в двух представлениях, вызывая метод `setModel()`.

Листинг 12.7. Отображение данных модели целых чисел (файл main.cpp)

```
#include <QtWidgets>
#include "IntListModel.h"

int main( int argc, char** argv ) {
    QApplication app( argc, argv );
    IntListModel model(QList<int>() << 123 << 2341 << 32 << 5342 << 723);
```

```
    QListView list;
    list.setModel(&model);
    list.show();

    QTableView table;
    table.setModel(&model);
    table.show();

    return app.exec();
}
```

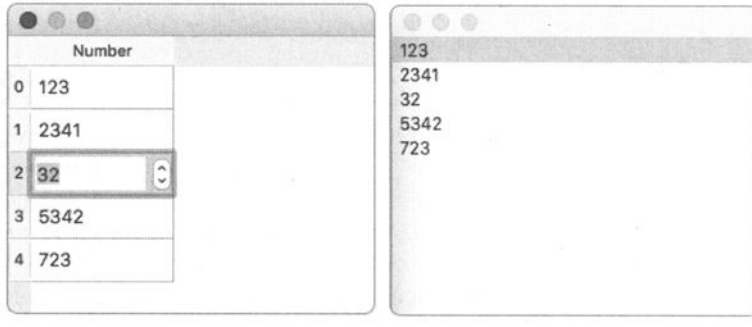


Рис. 12.13. Отображение данных модели списка целых чисел

Заголовочный файл самой модели приведен в листинге 12.8. Для реализации модели на базе класса `QAbstractListModel` необходимо реализовать методы `rowCount()` и `data()`. Метод `rowCount()` будет сообщать о количестве строк модели, а метод `data()` отвечает за доставку данных, которые он возвращает в объектах класса `QVariant`.

Мы также переопределяем здесь метод `headerData()`. Это нужно для того, чтобы модель работала с заголовками `QTableView` и `QTreeView`, хотя такое требование и не является обязательным. Наша модель допускает изменение данных, и именно поэтому мы обязаны переопределить методы `flags()` и `setData()`.

Создаваемая модель будет допускать операции вставки и удаления строк, и нам понадобится в нашем классе модели еще реализовать методы `insertRows()` и `removeRows()`.

> ***Еще о реализации методов `insertRows()` и `removeRows()`***
>
> Хотя в нашем конкретном случае, когда над моделью не производятся операций вставки и удаления строк, можно обойтись и без реализации этих двух методов. Тем не менее, будет интересно реализовать их для осознания более полной картины реализации класса модели.

Листинг 12.8. Определение класса `IntListModel` (файл IntListModel.h)

```
#pragma once

#include <QAbstractListModel>

// ============================================================================
class IntListModel : public QAbstractListModel {
Q_OBJECT
```

```
private:
    QList<int> m_list;

public:
    IntListModel(const QList<int>& list, QObject* pobj = 0);

    QVariant data(const QModelIndex& index, int nRole) const;

    bool setData(const QModelIndex& index,
                 const QVariant&    value,
                 int                nRole
                );

    int rowCount(const QModelIndex& parent = QModelIndex()) const;

    QVariant headerData(int             nSection,
                        Qt::Orientation orientation,
                        int             nRole = Qt::DisplayRole
                       ) const;

    Qt::ItemFlags flags(const QModelIndex &index) const;

    bool insertRows(int                nRow,
                    int                nCount,
                    const QModelIndex& parent = QModelIndex()
                   );

    bool removeRows(int                nRow,
                    int                nCount,
                    const QModelIndex& parent = QModelIndex()
                   );
};
```

Конструктор нашего класса модели списка целых чисел (листинг 12.9) служит для инициализации атрибута `m_list` списком чисел и передачи указателя на объект предка унаследованному классу.

Листинг 12.9. Конструктор `IntListModel()` (файл IntListModel.cpp)

```
IntListModel::IntListModel(const QList<int>& list, QObject* pobj/*=0*/)
        : QAbstractListModel(pobj)
        , m_list(list)
{
}
```

Посредством интерфейса модели, которую мы реализовали для связи со структурой данных, опрашиваемых представлением, метод `data()` (листинг 12.10) возвращает интересующую представление информацию об элементе в объекте класса `QVariant`. Помимо индекса, в этот метод передаются значения ролей. В нашем случае, если они предназначены для отображения (`Qt::DisplayRole`) или редактирования (`Qt::EditRole`), мы возвращаем значение, записанное в атрибуте, представляющем собой список целых чисел, на позиции строки. Если роль не предназначена для отображения или редактирования, или же представление запро-

сит о данных, которых мы не имеем, то возвратим пустой объект QVariant и тем самым укажем на то, что не располагаем этими данными.

Листинг 12.10. Метод data() (файл IntListModel.cpp)

```
QVariant IntListModel::data(const QModelIndex& index, int nRole) const
{
    if (!index.isValid()) {
        return QVariant();
    }
    if (index.row() < 0 || index.row() >= m_list.size()) {
        return QVariant();
    }
    return (nRole == Qt::DisplayRole || nRole == Qt::EditRole)
           ? m_list.at(index.row())
           : QVariant();
}
```

Для установки значения в методе setData() требуются три параметра: индекс, значение и роль (листинг 12.11). Прежде всего мы проверяем индекс вызовом метода isValid(). Если индекс не пуст, а роль предназначена для редактирования, то мы заменяем существующее значение в атрибуте списка (m_list) новым, используя метод replace(). Заметьте, что перед тем как провести замену, мы должны преобразовать значение к нужному нам типу. В нашем случае мы преобразуем атрибут value класса QVariant к целому типу при помощи шаблонного метода QVariant::value<T>(), параметризовав его типом int (см. *главу 4*). Эту конструкцию можно заменить методом QVariant::toInt(). После замены отправляем сигнал dataChanged(), что необходимо для того, чтобы все подсоединенные к модели представления могли незамедлительно обновить свое содержимое. Возвращаемое нами (из метода) значение true сообщает об успешно проведенной операции установки данных, а false сообщает представлению об ошибке.

Листинг 12.11. Метод setData() (файл IntListModel.cpp)

```
bool IntListModel::setData(const QModelIndex& index,
                           const QVariant&    value,
                           int                nRole
                          )
{
    if (index.isValid() && nRole == Qt::EditRole) {
        m_list.replace(index.row(), value.value<int>());
        emit dataChanged(index, index);
        return true;
    }
    return false;
}
```

Метод rowCount(), приведенный в листинге 12.12, должен сообщать о количестве строк. Количество строк модели соответствует количеству элементов, содержащихся в атрибуте списка целых чисел (m_list). Если индекс предка вдруг окажется действительным, то мы

должны вернуть значение 0, так как это модель списка, и в ней не должно быть ни предков, ни дочерних элементов.

Листинг 12.12. Метод rowCount() (файл IntListModel.cpp)

```
int IntListModel::rowCount(const QModelIndex& parent/*=QModelIndex()*/
                           ) const
{
    if (parent.isValid()) {
        return 0;
    }

    return m_list.size();
}
```

Метод headerData() из листинга 12.13 нужен для того, чтобы модель была в состоянии подписывать горизонтальные и вертикальные секции заголовков, которыми располагают иерархическое и табличное представления. Иерархическое представление имеет только горизонтальные заголовки, табличное представление располагает обоими типами заголовков (горизонтальными и вертикальными). Если представление запрашивает надпись для горизонтального заголовка, то мы возвращаем строку "Number", а для вертикальных секций заголовков возвращается номер переданной секции.

Листинг 12.13. Метод headerData() (файл IntListModel.cpp)

```
QVariant IntListModel::headerData(int             nSection,
                                  Qt::Orientation orientation,
                                  int             nRole/*=DisplayRole*/
                                 ) const
{
    if (nRole != Qt::DisplayRole) {
        return QVariant();
    }
    return (orientation == Qt::Horizontal) ? QString("Number")
                                           : QString::number(nSection);
}
```

Для того чтобы предоставить возможность редактирования элементов, метод flags() возвращает для каждого элемента унаследованное значение с добавлением флага Qt::ItemIsEditable. Если индекс пуст, то возвращается значение без добавлений (листинг 12.14).

Листинг 12.14. Метод flags() (файл IntListModel.cpp)

```
Qt::ItemFlags IntListModel::flags(const QModelIndex& index) const
{
    Qt::ItemFlags flags = QAbstractListModel::flags(index);
    return index.isValid() ? (flags | Qt::ItemIsEditable)
                           : flags;
}
```

В листинге 12.15 метод `insertRows()` принимает в качестве входных параметров: позицию для вставки `nRow`, количество элементов, которые с этой позиции необходимо вставить `nCount`, и индекс предка. Если индекс предка является действительным, то мы покидаем этот метод, так как недопустимо, чтобы модель списка содержала элементы предков. Теперь, прежде чем приступить к вставке элементов, мы обязаны вызвать метод `QAbstractListItem::beginInsertRows()` — это нужно для того, чтобы модель смогла уведомить об этом всех заинтересованных в ее данных — например, виджеты представлений, подсоединенные к ней. В этот метод нужно передать: индекс, начальную `nRow` и конечную позиции для вставки элементов. Конечную позицию мы вычисляем, прибавив к начальной позиции количество элементов и отняв единицу. В цикле вставка элементов производится вызовом метода `QList<T>::insert()` — каждый новый вставленный элемент получает значение `0`. Проделав изменение со списком `m_list`, мы, опять же, обязаны уведомить, что завершили процесс изменения данных, всех заинтересованных в этом. Для этого мы вызываем метод `QAbstractListItem::endInsertRows()`.

Листинг 12.15. Метод `insertRows()` (файл IntListModel.cpp)

```
bool IntListModel::insertRows(int                nRow,
                              int                nCount,
                              const QModelIndex& parent/*=QModelIndex()*/
                             )
{
    if (parent.isValid()) {
        return false;
    }

    beginInsertRows(QModelIndex(), nRow, nRow + nCount - 1);
    for (int i = 0; i < nCount; ++i) {
        m_list.insert(nRow, 0);
    }
    endInsertRows();

    return true;
}
```

Реализация метода `removeRows()` (листинг 12.16) очень похожа на реализацию метода `insertRows()` (см. листинг 12.15) с той лишь разницей, что происходит удаление элементов. Поэтому используется метод `QList<T>::removeAt()`. Для того чтобы модель могла уведомить все связанные с ней объекты о том, что начнут производиться изменения, вызываем метод `QAbstractListItem::beginRemoveRows()`, а по завершении изменений вызываем метод `QAbstractListItem::endRemoveRows()`.

Листинг 12.16. Метод `removeRows()` (файл IntListModel.cpp)

```
bool IntListModel::removeRows(int                nRow,
                              int                nCount,
                              const QModelIndex& parent/*=QModelIndex()*/
                             )
```

```
{
    if (parent.isValid()) {
        return false;
    }

    beginRemoveRows(QModelIndex(), nRow, nRow + nCount - 1);
    for (int i = 0; i < nCount; ++i) {
        m_list.removeAt(nRow);
    }
    endRemoveRows();

    return true;
}
```

Для создания табличной модели на базе класса `QAbstractTableModel` нужно поступить так же, как мы поступили в случае класса `QAbstractListModel`. Дополнительно к этому, в унаследованном от `QAbstractTableModel` классе необходимо реализовать метод `columnCount()`, предоставляющий информацию о количестве столбцов таблицы. И если модель предполагает возможность вставки и удаления столбцов, то необходимо еще дополнительно реализовать методы `insertColumns()` и `removeColumns()`.

В качестве примера создадим программу (листинг 12.17), отображающую данные созданной нами табличной модели (рис. 12.14).

	178	179	180	181	182	183	18
93	93,178	93,179	93,180	93,181	93,182	93,183	93,184
94	94,178	94,179	94,180	94,181	94,182	94,183	94,184
95	95,178	95,179	95,180	95,181	95,182	95,183	95,184
96	96,178	96,179	96,180	96,181	96,182	96,183	96,184
97	97,178	97,179	97,180	97,181	97,182	97,183	97,184
98	98,178	98,179	98,180	98,181	98,182	98,183	98,184
99	99,178	99,179	99,180	99,181	99,182	99,183	99,184
100	100,178	100,179	100,180	100,181	100,182	100,183	100,184
101	101,178	101,179	101,180	101,181	101,182	101,183	101,184
102	102,178	102,179	102,180	102,181	102,182	102,183	102,184
103	103,178	103,179	103,180	103,181	103,182	103,183	103,184

Рис. 12.14. Отображение табличной модели данных

Пример, показанный в листинге 12.17, в целом очень похож на предыдущий, приведенный в листингах 12.7–12.16. Разница в том, что в этом случае наши данные мы храним в хэше и дополнительно перезаписываем метод `columnCount()`. В конструкторе мы допускаем инициализацию данными. Данные ячеек устанавливаются автоматически и представляют собой строку, составленную из номеров строки и столбца ячейки. Но эти данные можно изменять.

Для этого мы проверяем роль `Qt::EditRole` в методе `setData()`, а в методе `flags()` возвращаем значение с добавленным флагом `Qt::ItemIsEditable`. В функции `main()` мы создаем нашу модель размером 200 на 200 и устанавливаем ее в табличное представление.

Листинг 12.17. Отображение табличной модели данных (файл main.cpp)

```
#include <QtWidgets>
// ====================================================================
class TableModel : public QAbstractTableModel {
private:
    int                          m_nRows;
    int                          m_nColumns;
    QHash<QModelIndex, QVariant> m_hash;

public:
     // ----------------------------------------------------------------
    TableModel(int nRows, int nColumns, QObject* pobj = 0)
        : QAbstractTableModel(pobj)
        , m_nRows(nRows)
        , m_nColumns(nColumns)
    {
    }

     // ----------------------------------------------------------------
    QVariant data(const QModelIndex& index, int nRole) const
    {
        if (!index.isValid()) {
            return QVariant();
        }
        QString str =
           QString("%1,%2").arg(index.row() + 1).arg(index.column() + 1);
        return (nRole == Qt::DisplayRole || nRole == Qt::EditRole)
               ? m_hash.value(index, QVariant(str))
               : QVariant();
    }

     // ----------------------------------------------------------------
    bool setData(const QModelIndex& index,
                 const QVariant&    value,
                 int                nRole
                )
    {
        if (index.isValid() && nRole == Qt::EditRole) {
            m_hash[index] = value;
            emit dataChanged(index, index);
            return true;
        }
        return false;
    }
```

```
    // ------------------------------------------------------------------
    int rowCount(const QModelIndex&) const
    {
        return m_nRows;
    }

    // ------------------------------------------------------------------
    int columnCount(const QModelIndex&) const
    {
        return m_nColumns;
    }

    // ------------------------------------------------------------------
    Qt::ItemFlags flags(const QModelIndex& index) const
    {
        Qt::ItemFlags flags = QAbstractTableModel::flags(index);
        return index.isValid() ? (flags | Qt::ItemIsEditable)
                               : flags;
    }
};

// ------------------------------------------------------------------------
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    TableModel model(200, 200);

    QTableView tableView;
    tableView.setModel(&model);
    tableView.show();

    return app.exec();
}
```

Если бы мы хотели построить класс нашей модели на классе `QAbstractItemModel`, то его реализация выглядела бы аналогично табличной модели `QAbstractTableModel`. Но дополнительно в унаследованном классе потребовалось бы реализовать методы `QAbstractItemModel::index()` и `QAbstractItemModel::parent()`. В этом случае для создания индексов использовался бы метод `QAbstractItemModel::createIndex()`, поэтому его тоже нужно было бы перегрузить.

Промежуточная модель данных (Proxy model)

В оригинальной модели может получиться так, что какой-либо из элементов находится первым, а нам нужно поместить его в конец или в середину. Изменение расположения данных в оригинальной модели данных вызовет изменения во всех присоединенных представлениях, что может быть нежелательно. В подобных случаях нам понадобится промежуточная модель. *Промежуточная модель* — это модель, находящаяся между моделью данных и представлением (рис. 12.15). Такая модель предоставляет возможность выполнять манипуляции с данными, при этом не изменяя данные оригинальной модели.



Рис. 12.15. Схема использования промежуточной модели

С помощью промежуточной модели можно выполнить сортировку или перестановку данных. Таким образом можно сделать два представления: одно из которых показывает измененные данные, а другое — оригинальные.

Еще одна полезная операция, которую можно выполнить с помощью промежуточной модели, — это отбор элементов данных, то есть их фильтрация. Для этого в промежуточной модели необходимо установить критерии, с помощью которых будет осуществляться отбор. Класс `QSortFilterProxyModel` является обобщенной реализацией промежуточной модели, позволяющей выполнять сортировку и отбор. Для задания критериев отбора может быть использован слот `QSortFilterProxyModel::setFilterRegExp()`, в который передается объект регулярного выражения класса `QRegExp` (см. *главу 4*).

При отборе модель возвращает индексы только тех строк, для которых текст в столбце соответствует указанному критерию. При сортировке порядок расположения осуществляется в соответствии со значениями элементов, расположенных в каждом столбце. Сортировку каждого столбца можно проводить по возрастанию и убыванию.

Программа (листинг 12.18), окно которой показано на рис. 12.16, осуществляет отбор тех элементов, имена которых начинаются на букву «E». В левой части окна программы представление отображает оригинальную модель, а в правой — промежуточную.

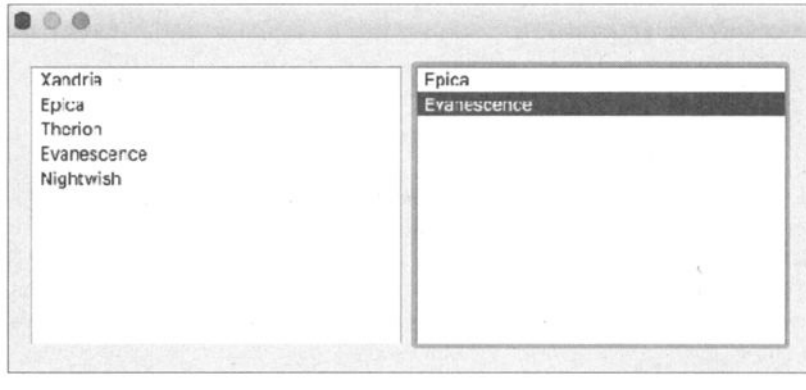


Рис. 12.16. Отбор элементов

В функции `main()`, используемой в листинге 12.18, мы создаем модель `QStringListModel`, которую инициализируем строковыми значениями. После этого создается промежуточная модель `QSortFilterProxyModel`, которая с помощью метода `setSourceMode()` связывается с оригинальной моделью. Для осуществления отбора элементов, начинающихся на букву «E», в слоте `setFilterWildcard()` устанавливается маска `"E*"`.

Вариант: создание виджета текстового поля

Можно было бы поступить и так: создать виджет текстового поля, с помощью которого пользователь сам бы устанавливал критерий для выборки. В этом случае сигнал

textChanged() нужно было бы соединить со слотом setFilterWildcard() модели SortFilterProxyModel, что позволило бы при изменении критерия отбора, находящегося в текстовом поле, сразу же его применить.

Создав два представления QListView, в одном из них мы устанавливаем оригинальную модель, а в другом — промежуточную.

Листинг 12.18. Отбор элементов (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QStringListModel model;
    model.setStringList(QStringList() << "Xandria"
                                      << "Epica"
                                      << "Therion"
                                      << "Evanescence"
                                      << "Nightwish"
                        );

    QSortFilterProxyModel proxyModel;
    proxyModel.setSourceModel(&model);
    proxyModel.setFilterWildcard("E*");

    QListView* pListView1 = new QListView;
    pListView1->setModel(&model);

    QListView* pListView2 = new QListView;
    pListView2->setModel(&proxyModel);

    //Layout setup
    QHBoxLayout* phbxLayout = new QHBoxLayout;
    phbxLayout->addWidget(pListView1);
    phbxLayout->addWidget(pListView2);
    wgt.setLayout(phbxLayout);

    wgt.show();

    return app.exec();
}
```

Модель элементно-ориентированных классов

В *главе 11* мы познакомились с классами элементно-ориентированного подхода, которые работают по принципу: создать элемент с данными и вставить его в представление. Заметьте, данные содержатся в самих элементах. Этих классов три (с постфиксом Widget): QListWidget, QTreeWidget и QTableWidget.

На самом деле эти три класса тоже основаны на архитектуре «модель-представление» и унаследованы от классов представлений `QListView`, `QTreeView` и `QTableView` (см. рис. 12.4). Но, в отличие от этих классов, внутри себя они имеют свою собственную, встроенную модель данных. А это значит, что данные элементно-ориентированных классов можно разделять с другими представлениями (рис. 12.17), для чего нужно лишь получить указатель на эту модель данных, который возвращает метод `QAbstractItemView::model()`.

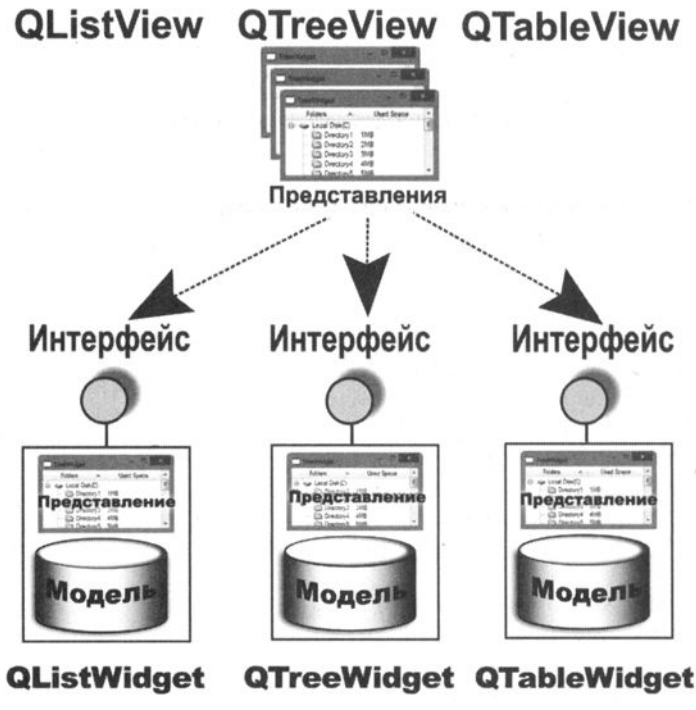


Рис. 12.17. Разделение моделей классов элементно-ориентированного подхода с представлениями

Подобный механизм разделения моделей данных виджетов элементно-ориентированного подхода с представлениями не рекомендуется с позиции «модель-представление», но для простых ситуаций он вполне приемлем и может помочь сэкономить время, если в программе уже имеется реализация элементно-ориентированных классов.

На рис. 12.18 показан пример разделения модели виджета класса `QListWidget` (*слева*) с представлением класса `QListView` (*справа*) — другими словами, оба виджета смотрят на одну и ту же модель данных. Программный код, реализующий этот пример, приведен в листинге 12.19.

За базу для программы из листинга 12.19 был взят код листинга 11.1, в котором создается и заполняется элементами данных виджет элементно-ориентированного класса `QListWidget`. Его мы дополнили созданием представления списка `QListView`.

Для того чтобы иметь возможность показать модель виджета (класса `QListWidget`), мы вызываем из него метод `model()` и передаем возвращенный им указатель в метод `setModel()` объекта `listView`. Кроме того, разделяем модель выделения при помощи методов `selectionModel()` и `setSelectionModel()`.

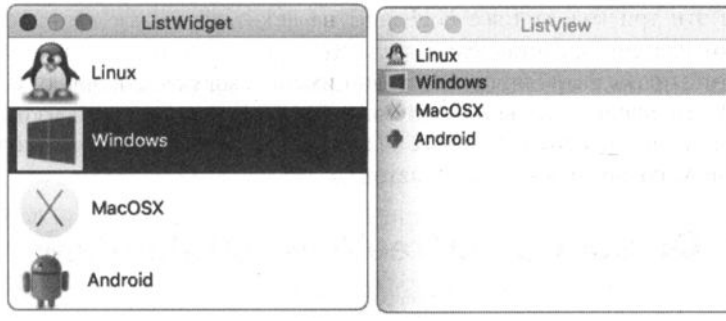


Рис. 12.18. Разделение модели данных

Листинг 12.19. Получение доступа к модели (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication     app(argc, argv);
    QStringList      lst;
    QListWidget      lwg;
    QListWidgetItem* pitem = 0;

    lwg.setIconSize(QSize(48, 48));
    lst << "Linux" << "Windows" << "MacOS" << "OS2";
    foreach(QString str, lst) {
        pitem = new QListWidgetItem(str, &lwg);
        pitem->setIcon(QPixmap(str + ".jpg"));
    }
    lwg.setWindowTitle("ListWidget");
    lwg.show();

    QListView listView;
    listView.setModel(lwg.model());
    listView.setSelectionModel(lwg.selectionModel());
    listView.setWindowTitle("ListView");
    listView.show();

    return app.exec();
}
```

Резюме

Концепция «интервью» является свободной вариацией шаблона разработки «модель-представление», адаптированного специально для элементов данных. Она отделяет данные от их представления, что делает возможным отображение одних и тех же данных в различных представлениях по-разному, без каких-либо изменений лежащей в основе структуры самих данных. Эта концепция также обеспечивает расширяемость и гибкость при использовании большого объема данных.

Благодаря такому подходу можно разделять между представлениями не только модель данных, но и выделение самих элементов в представлениях.

Модель — это оболочка вокруг данных, связь с которыми происходит с помощью интерфейса. Вы можете реализовать свой собственный интерфейс для своей собственной структуры данных. Для обеспечения связи с данными необходимо унаследовать один из классов моделей и реализовать интерфейс своей модели.

При помощи промежуточной модели можно манипулировать с данными, не воздействуя на данные оригинальной модели.

Классы элементно-ориентированного подхода основаны на архитектуре «модель-представление» и имеют встроенную модель данных, которую можно разделять с другими представлениями.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/12-510/** или с помощью следующего QR-кода (рис. 12.19):



Рис. 12.19. QR-код для доступа на страницу комментариев к этой главе

ГЛАВА 13

Цветовая палитра элементов управления

Просыпается программист, открывает окно, на улице стоит серый ненастный день. «Опять палитра слетела», — раздосадованно подумал он.

Цветовая палитра элементов управления — это таблица, в которой содержатся цвета, используемые виджетом при отображении на экране. Дело в том, что цвета виджетов не определены окончательно и в любой момент могут быть изменены передачей соответствующего цвета текста, цвета фона и т. п.

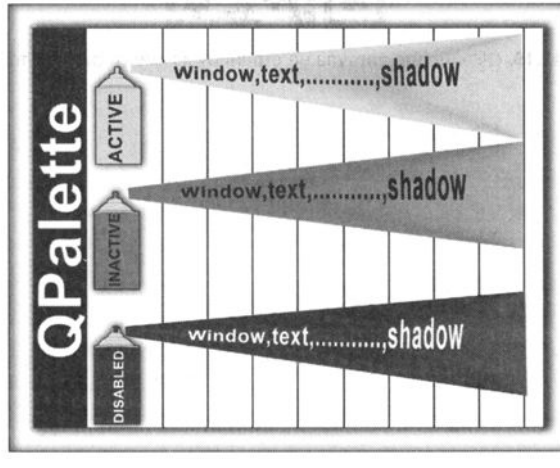


Рис. 13.1. Схема цветовой палитры виджета

Каждый из виджетов содержит в себе объект палитры, доступ к которому можно получить с помощью метода `palette()` класса `QWidget`. Сама палитра — это класс `QPalette`, который включает три основные группы объектов (рис. 13.1). Эти группы определяют три возможных состояния виджета: *активное* (Active), *неактивное* (Inactive) и *недоступное* (Disabled). Каждая из указанных групп состоит из различных цветовых ролей (color roles), описание которых приведено в табл. 13.1. Каждая роль имеет кисть (`QBrush`) и цвет (`QColor`).

Механизм неявных общих данных

Класс `QPalette` задействует механизм неявных общих данных, а это означает, что все виджеты используют ссылку на один и тот же объект палитры. Если палитра виджета подвергается изменению, виджет получает свой собственный объект данных палитры.

Таблица 13.1. Цветовые перечисления `ColorRole` *класса* `QPalette`

Флаг	Описание
`WindowText`	Цвет, который используется по умолчанию для рисования пером (см. *главу 18*). Этот цвет находится на переднем плане. По умолчанию это черный цвет
`Text`	Цвет текста. По умолчанию — черный
`BrightText`	Яркий цвет текста, отличающийся от цвета `WindowText`. Обычно совпадает с `Text`. По умолчанию — черный
`ButtonText`	Цвет текста для надписей на кнопках. По умолчанию — черный
`Highlight`	Цвет фона выделения элементов. По умолчанию — темно-голубой
`HighlightedText`	Цвет текста выделенных элементов. Контрастен к цвету, заданному значением `Hightlight`. По умолчанию — белый
`Window`	Основной цвет фона. По умолчанию — светло-серый
`Base`	Цвет для заднего фона виджета. По умолчанию — белый или другой цвет светлого оттенка
`Button`	Цвет кнопки, как правило, одного цвета с фоном. По умолчанию — светло-серый
`Link`	Цвет, используемый для непосещенной гипертекстовой ссылки. По умолчанию — голубой
`LinkVisited`	Цвет, используемый для обозначения посещенной гипертекстовой ссылки. По умолчанию — розовый
`Light`	Цвет эффекта объемности. Должен быть светлее цвета, заданного значением `Button`. По умолчанию — белый (рис. 13.2)
`Midlight`	Цвет эффекта объемности. По умолчанию — светло-серый (см. рис. 13.2)
`Dark`	Цвет эффекта объемности. Должен быть темнее цвета, заданного значением `Button`. По умолчанию — темно-серый (см. рис. 13.2)
`Mid`	Цвет эффекта объемности. По умолчанию — средне-серый
`Shadow`	Цвет эффекта объемности. По умолчанию — черный (см. рис. 13.2)

На рис. 13.2 показано, каким областям виджета счетчика соответствуют некоторые из значений, приведенных в табл. 13.1.

Пример изменения палитры приведен в листинге 13.1. На рис. 13.3 показан виджет счетчика с измененной палитрой.

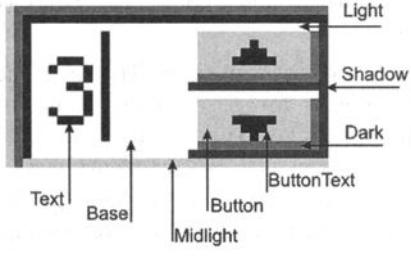


Рис. 13.2. Некоторые элементы палитры виджета счетчика

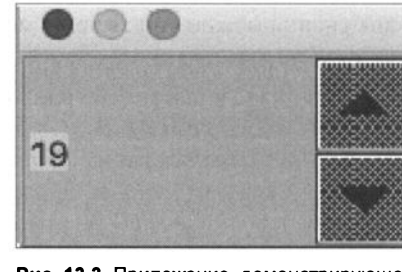


Рис. 13.3. Приложение, демонстрирующее измененную палитру виджета

Листинг 13.1. Изменение палитры (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QSpinBox     spb;

    QPalette pal = spb.palette();

    pal.setBrush(QPalette::Button, QBrush(Qt::red, Qt::Dense3Pattern));
    pal.setColor(QPalette::ButtonText, Qt::blue);
    pal.setColor(QPalette::Text, Qt::magenta);
    pal.setColor(QPalette::Active, QPalette::Base, Qt::green);

    spb.setPalette(pal);
    spb.resize(150, 74);
    spb.show();

    app.setStyle("Windows");

    return app.exec();
}
```

В листинге 13.1 создается виджет счетчика `spb` и объект палитры `pal`. При создании объекту палитры присваивается значение палитры виджета счетчика, которое извлекается вызовом метода `palette()`. После этого объект палитры подвергается изменениям с помощью метода `setBrush()` и серии вызовов метода `setColor()`. В эти методы передаются флаги цветовых ролей, которые нужно поменять (см. табл. 13.1). Обратите внимание на последний вызов, в котором указано, что значение роли цвета `QPalette::Base` предназначено только для активного состояния. Это значит, что если окно сделать неактивным, то базовый цвет будет взят из неактивной группы палитры (в нашем случае зеленый цвет поменяется на белый). Полученная палитра устанавливается в виджет методом `setPalette()`. Наконец, мы применяем стиль Windows для того, чтобы наш виджет не использовал стиль платформы и выглядел везде одинаково. Более подробная информация о стилях приведена в *главе 26*.

Как мы уже знаем из листинга 13.1, при помощи метода `setBrush()` можно задавать не только цвет, но и образец заполнения. Правда, это имеет смысл лишь для заполнения площадей, так как при рисовании линий образец будет проигнорирован. Задать образец заполнения для кнопки можно при помощи метода `setBrush()`.

Листинг 13.1 демонстрирует изменение палитры только одного виджета. На практике это нежелательно — представьте себе приложение, все элементы которого имеют разные цвета! Поэтому если необходимо изменить палитру для виджетов, то лучше делать это для всех виджетов сразу, централизованно. Для этого необходимо передать объект палитры в статический метод `QAppilcation::setPalette()`. Желательно создавать такую палитру, чтобы в приложении использовалось не менее трех и не более семи цветов. Мы устанавливаем стиль `Windows` для того, чтобы на всех операционных системах кнопки со стрелками имели прямоугольную форму.

Листинг 13.2 показывает, как установить палитру для всего приложения. При создании объекта палитры для задания цвета кнопок и фона в ее конструктор передаются два параметра: первый — цвет для кнопок, а второй — основной цвет. Все остальные цвета палитры автоматически вычисляются на основе этих двух, для чего используется несложный алгоритм.

Листинг 13.2. Установка палитры для всего приложения (файл main.cpp)

```
int main (int argc, char** argv)
{
    QApplication app(argc, argv);
    QPalette     pal(Qt::red, Qt::blue);
    QApplication::setPalette(pal);
    ...
}
```

Резюме

Из этой главы мы узнали, что цвета виджетов в любой момент могут быть изменены передачей палитры. Каждый из виджетов содержит свой собственный объект палитры, который может быть изменен. Палитра виджета состоит из трех групп, соответствующих активному, неактивному и недоступному состояниям. В палитре могут участвовать не только цвета, но и образцы заполнения. Палитру, используемую в приложении, можно изменить для всех виджетов сразу с помощью статического метода `QApplication::setPalette()`.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/13-510/** или с помощью следующего QR-кода (рис. 13.4):



Рис. 13.4. QR-код для доступа на страницу комментариев к этой главе



ЧАСТЬ III

События и взаимодействие с пользователем

За двумя зайцами погонишься... не поймаешь так согреешься.

Народная мудрость

Глава 14. События

Глава 15. Фильтры событий

Глава 16. Искусственное создание событий



ГЛАВА 14

События

— Что-то происходит, но ты не знаешь что.
— А вы знаете, мистер Джонс?
Боб Дилан, «Баллада худого человека»

Обработка событий лежит в основе работы каждого приложения, имеющего пользовательский интерфейс. Событие можно охарактеризовать как механизм оповещения приложения о каком-либо происшествии. Например, нажатие пользователем кнопки мыши или клавиши клавиатуры приведет к созданию события мыши или клавиатуры, событие будет создаваться и при необходимости перерисовки содержимого окна и т. п. Очевидно, что основная масса событий тесно связана с действиями, предпринимаемыми пользователем. Но есть и события, создаваемые самой операционной системой, — например, событие таймера. Все события помещаются в соответствующую очередь для их дальнейшей обработки.

Но ведь если «что-то происходит», то высылаются сигналы. Зачем же тогда нужны события?

Механизм сигналов и слотов, по сравнению с событиями, представляет собой механизм более высокого уровня, предназначенный для связи объектов. Хотя и то, и другое является уведомлением о происходящем. Например, нажатие кнопки приводит к оповещению о происходящем всех подключенных к сигналу объектов. События оповещают объекты о действиях пользователя общего и детального характера (например, о перемещении указателя мыши или нажатии какой-либо клавиши клавиатуры). Другими словами, воспользовавшись стандартными сигналами, вы можете сделать заключение о том, что кнопка была нажата, но узнать координаты указателя мыши в момент нажатия не представляется возможным. Для получения подобного рода информации понадобятся объекты событий, часто содержащие дополнительную информацию, которой может воспользоваться объект, получающий события. В частности, в объекте события мыши `QMouseEvent` передаются координаты и код нажатой кнопки.

Использование событий особенно интересно при создании собственных виджетов, поскольку часто сигналы высылаются из методов обработки событий. Например, при щелчке мыши на виджете можно из метода обработки события `mousePressEvent()` выслать сигнал `clicked()`.

Определение и переопределение методов обработки событий

Следует учитывать, что в Qt все методы обработки событий определены как `virtual protected`. Поэтому при переопределении этих методов в унаследованных классах желательно определять их как `protected`.

Есть еще одно отличие сигналов от событий — события обрабатываются лишь одним методом, а сигналы могут обрабатываться неограниченным количеством соединенных с ними слотов. Кроме того, сигналы могут базироваться на событиях, то есть высылаться из методов событий, но высылку событий из сигналов просто даже невозможно себе представить.

Qt предоставляет целый ряд классов для различного рода событий: клавиатуры, мыши, таймера и др. На рис. 14.1 представлена иерархия классов событий Qt.

Как видно из рис. 14.1, класс `QEvent` является базовым для всех категорий событий. Его объекты содержат информацию о типе произошедшего события. А для каждого типа собы-



Рис. 14.1. Иерархия классов событий

тия имеется целочисленный идентификатор, который устанавливается в конструкторе QEvent и может быть получен при помощи метода QEvent::type().

Класс события QEvent содержит методы accept() и ignore(), с помощью которых устанавливается или сбрасывается специальный флаг, регулирующий дальнейшую обработку события предком объекта. Если был вызван метод ignore(), то при возвращении из метода обработки события событие будет передано дальше на обработку объекту-предку. В случае, если был вызван метод accept(), событие считается полностью обработанным и дальше передаваться объекту-предку не будет.

Класс QInputEvent является базовым для событий, связанных с пользовательским вводом (см. рис. 14.1). Этот класс реализует всего лишь один метод: modifiers(). С его помощью все унаследованные от него классы способны получать состояние клавиш-модификаторов <Ctrl>, <Shift> и <Alt>, которые могут быть нажаты в момент наступления события. Некоторые их значения указаны в табл. 14.1.

Таблица 14.1. *Некоторые значения модификаторов пространства имен Qt*

Константа	Значение (HEX)	Описание
NoModifier	0	Клавиши модификаторов не нажаты
ShiftModifier	2000000	Нажата клавиша <Shift>
ControlModifier	4000000	Нажата клавиша <Ctrl>
AltModifier	8000000	Нажата клавиша <Alt>

Обработка событий начинается с момента вызова в вашей основной программе метода QCoreApplication::exec(). События доставляются всем объектам, созданным от классов, которые унаследованы от класса QObject. Некоторые события могут быть доставлены сразу, а некоторые попадают в очередь и могут быть обработаны только при возвращении управления циклу обработки события QCoreApplication::exec().

Qt использует специальные механизмы для оптимизации некоторых типов событий. Так, например, серия событий перерисовки QPaintEvent в целях увеличения производительности может быть «упакована» (объединена) в одно событие с регионом рисования, составленным из регионов всех находящихся в очереди событий рисования. Тем самым метод обработки события рисования (paintEvent()) будет вызван только один раз.

Переопределение специализированных методов обработки событий

Переопределение специализированных методов является самым распространенным способом их обработки. Для того чтобы обработать определенное событие, нужно унаследовать необходимый класс и переопределить нужный метод обработки события. В этот метод передается указатель на объект события, который содержит информацию о нем. Каждый метод получает объект соответствующего типа — например, методы keyPressEvent() и keyReleaseEvent() получают указатель на объект класса QKeyEvent.

События клавиатуры

Класс *QKeyEvent*

Класс `QKeyEvent` содержит данные о событиях клавиатуры. С его помощью можно получить информацию о клавише, вызвавшей событие, а также ASCII-код отображенного символа (American Standard Code for Information Interchange, американский стандартный код для обмена информацией). Объект события передается в методы `QWidget::keyPressEvent()` и `QWidget::keyReleaseEvent()`, определенные в классе `QWidget`. Событие может вызываться нажатием любой клавиши на клавиатуре, включая <Shift>, <Ctrl>, <Alt>, <Esc> и <F1>–<F12>. Исключение составляют клавиша табулятора <Tab> и ее совместное нажатие с клавишей <Shift>, которые используются методом обработки `QWidget::event()` для передачи фокуса следующему виджету.

Метод `keyPressEvent()` вызывается каждый раз при нажатии одной из клавиш на клавиатуре, а метод `keyReleaseEvent()` — при ее отпускании.

В методе обработки события с помощью метода `QKeyEvent::key()` можно определить, какая из клавиш его инициировала. Этот метод возвращает значение целого типа, которое можно сравнить с константами клавиш, определенными в классе Qt (табл. 14.2).

Давайте кратко рассмотрим некоторые из кодов клавиш, приведенных в табл. 14.2:

- коды от `20` до `3F` полностью совпадают со значениями ASCII-кодов (см. *приложение 3*);
- от `30` до `39` — соответствуют цифровым клавишам, расположенным на клавиатуре в виде горизонтального ряда прямо над клавишами букв;
- от `41` до `5A` — являются идентификаторами букв. Обратите внимание на то, что они совпадают со значениями ASCII-кодов заглавных букв (см. *приложение 3*);
- от `1000030` до `1000052` — являются функциональными. В общей сложности их 35, но в табл. 14.2 указано только 12, что соответствует обычной клавиатуре;
- от `1000006` до `100009`, а также `1000025` и `1000026` — являются кодами клавиш цифровой клавиатуры;
- от `1000010` до `1000017` — являются кодами клавиш управления курсором;
- от `1000020` до `1000023` — соответствуют клавишам модификаторов;
- `1000000`, `1000001`, `1000004` и `1000005` — можно объединить в отдельную группу, так как они также генерируют коды символов.

Если необходимо узнать, были ли в момент наступления события совместно с клавишей нажаты клавиши модификаторов, например <Shift>, <Ctrl> или <Alt>, то это можно проверить с помощью метода `modifiers()`.

С помощью метода `text()` можно узнать Unicode-текст, полученный вследствие нажатия клавиши. Этот метод может оказаться полезным в том случае, если в виджете потребуется обеспечить ввод с клавиатуры. Для клавиш модификаторов метод вернет пустую строку. В таком случае нужно воспользоваться методом `key()`, который будет содержать код клавиши (см. табл. 14.1).

Метод для обработки событий клавиатуры класса, унаследованного от класса `QWidget`, может выглядеть следующим образом:

```
void MyWidget::keyPressEvent(QKeyEvent* pe)
{
    switch (pe->key()) {
    case Qt::Key_Z:
```

Таблица 14.2. *Некоторые значения перечислений* `Key` *пространства имен Qt*

Константа	Значение (HEX)	Константа	Значение (HEX)	Константа	Значение (HEX)
Key_Space	20	Key_B	42	Key_Insert	1000006
Key_NumberSign	23	Key_C	43	Key_Delete	1000007
Key_Dollar	24	Key_D	44	Key_Pause	1000008
Key_Percent	25	Key_E	45	Key_Print	1000009
Key_Ampersand	26	Key_F	46	Key_Home	1000010
Key_Apostrophe	27	Key_G	47	Key_End	1000011
Key_ParenLeft	28	Key_H	48	Key_Left	1000012
Key_ParenRight	29	Key_I	49	Key_Up	1000013
Key_Asterisk	2A	Key_J	4A	Key_Right	1000014
Key_Plus	2B	Key_K	4B	Key_Down	1000015
Key_Comma	2C	Key_L	4C	Key_PageUp	1000016
Key_Minus	2D	Key_M	4D	Key_PageDown	1000017
Key_Period	2E	Key_N	4E	Key_Shift	1000020
Key_Slash	2F	Key_O	4F	Key_Control	1000021
Key_0	30	Key_P	50	Key_Meta	1000022
Key_1	31	Key_Q	51	Key_Alt	1000023
Key_2	32	Key_R	52	Key_CapsLock	1000024
Key_3	33	Key_S	53	Key_NumLock	1000025
Key_4	34	Key_T	54	Key_ScrollLock	1000026
Key_5	35	Key_U	55	Key_F1	1000030
Key_6	36	Key_V	56	Key_F2	1000031
Key_7	37	Key_W	57	Key_F3	1000032
Key_8	38	Key_X	58	Key_F4	1000033
Key_9	39	Key_Y	59	Key_F5	1000034
Key_Colon	3A	Key_Z	5A	Key_F6	1000035
Key_Semicolon	3B	Key_Backslash	5C	Key_F7	1000036
Key_Less	3C	Key_Escape	1000000	Key_F8	1000037
Key_Equal	3D	Key_Tab	1000001	Key_F9	1000038
Key_Greater	3E	Key_Backspace	1000003	Key_F10	1000039
Key_Question	3F	Key_Return	1000004	Key_F11	100003A
Key_A	41	Key_Enter	1000005	Key_F12	100003B

```
        if (pe->modifiers() & Qt::ShiftModifier) {
            // Выполнить какие-либо действия
        }
        else {
            // Выполнить какие-либо действия
        }
        break;
    default:
        QWidget::keyPressEvent(pe); // Передать событие дальше
    }
}
```

В этом примере проверяется, не нажаты ли совместно клавиши <Z> и <Shift>. Для проверки статуса значения, возвращаемого методом `modifiers()`, используются значения, указанные в табл. 14.1.

Класс *QFocusEvent*

Когда пользователь набирает что-нибудь на клавиатуре, информацию о нажатых клавишах может принимать только один виджет. Если виджет в этот момент выбран для ввода с клавиатуры, то говорят, что он *находится в фокусе*. Объект события фокуса `QFocusEvent` передается в методы обработки сообщений `focusInEvent()` и `focusOutEvent()`. Этот объект не содержит значимой информации. Основное назначение класса `QFocusEvent` — сообщить о получении или потере виджетом фокуса, для того чтобы можно было, например, изменить его внешний вид. Эти методы вызываются в том случае, когда виджет получает (`focusInEvent()`) или теряет (`focusOutEvent()`) фокус.

Событие обновления контекста рисования. Класс *QPaintEvent*

Qt поддерживает *двойную буферизацию* (double buffering). Ее можно отключить вызовом метода `QWidget::setAttribute(Qt::WA_PaintOnScreen)`. Вполне возможно, последствия вас удивят: дело в том, что некогда выведенная в окно графическая информация вдруг исчезнет при изменении размеров окна приложения или после перекрытия его окном другого приложения. Чтобы этого не произошло, необходимо получать и обрабатывать событие `QPaintEvent`. В объекте класса `QPaintEvent` передается информация для перерисовки всего изображения или его части. Событие возникает тогда, когда виджет впервые отображается на экране явным или неявным вызовом метода `show()`, а также в результате вызова методов `repaint()` и `update()`. Объект события передается в метод `paintEvent()`, в котором реализуется отображение самого виджета. В большинстве случаев этот метод используется для полной перерисовки виджета. Для маленьких виджетов такой подход вполне приемлем, но для виджетов больших размеров рациональнее перерисовывать только отдельную область, действительно нуждающуюся в этом. Для получения координат и размеров такого участка вызывается метод `region()`. Вызовом метода `contains()` можно проверить, находится ли объект в заданной области. Например:

```
MyClass::paintEvent(QPaintEvent* pe)
{
    QPainter painter(this);
    QRect    r(40, 40, 100, 100);
```

```
    if (pe->region().contains(r)) {
        painter.drawRect(r);
    }
}
```

О графике мы еще поговорим в *части IV* этой книги.

События мыши

Мышь дает возможность пользователю указывать на объекты, находящиеся на экране компьютера, и с ее помощью можно проводить различные манипуляции над объектами, которые невозможно или неудобно выполнять с помощью клавиатуры.

Самое большое преимущество мыши перед клавиатурой состоит в том, что указание на предметы реального мира — это естественное действие для человека, заложенное в нем с раннего детства, чего не скажешь о работе с клавиатурой. Мышь можно охарактеризовать как располагающееся в виртуальном пространстве продолжение руки человека, с помощью которого можно выполнять разного рода операции. Например, указывать на объекты, выбирать их, перемещать с одного места на другое. Реализация событий мыши сложнее других, поскольку программа должна определять, какая кнопка нажата, удерживается она или нет, был ли выполнен двойной щелчок, и какие клавиши клавиатуры были нажаты в момент возникновения события.

Класс *QMouseEvent*

Объект этого класса содержит информацию о событии, вызванном действием мыши, и хранит в себе информацию о позиции указателя мыши в момент вызова события, статус кнопок мыши и даже некоторых клавиш клавиатуры. Этот объект передается в методы `mousePressEvent()`, `mouseMoveEvent()`, `mouseReleaseEvent()` и `mouseDoubleClickEvent()`.

Метод `mousePressEvent()` вызывается тогда, когда произошло нажатие на одну из кнопок мыши в области виджета. Если нажать кнопку мыши и, не отпуская ее, переместить указатель мыши за пределы виджета, то он будет получать события мыши, пока кнопку не отпустят. При движении мыши станет вызываться метод `mouseMoveEvent()`, а при отпускании кнопки произойдет вызов метода `mouseReleaseEvent()`.

По умолчанию метод `mouseMoveEvent()` вызывается при перемещении указателя мыши, только если одна из ее кнопок нажата. Это позволяет не создавать лишних событий во время простого перемещения указателя. Если же необходимо отслеживать все перемещения указателя мыши, то нужно воспользоваться методом `setMouseTracking()` класса `QWidget`, передав ему параметр `true`.

Метод `mouseDoubleClickEvent()` вызывается при двойном щелчке кнопкой мыши в области виджета.

Для определения местоположения указателя мыши в момент возникновения события можно воспользоваться методами `globalX()`, `globalY()`, `x()` и `y()`, которые возвращают целые значения, а также методами `pos()` или `globalPos()`. Метод `pos()` класса `QMouseEvent` возвращает позицию указателя мыши в момент наступления события относительно левого верхнего угла виджета. Если нужна абсолютная позиция (относительно левого верхнего угла экрана), то ее получают с помощью метода `globalPos()`.

Вызвав метод `button()`, можно узнать, какая из кнопок мыши была нажата в момент наступления события, — этот метод возвращает битовую комбинацию из приведенных в табл. 14.3 значений. Как можно видеть, эти значения не пересекаются, поэтому можно применять операцию | (ИЛИ) для их объединения.

Если необходимо узнать, были ли в момент возникновения события мыши нажаты клавиши-модификаторы <Ctrl>, <Shift> и/или <Alt>, то это можно проверить с помощью метода `modifiers()`, реализованного в базовом классе `QInputEvent` (см. рис. 14.1 и табл. 14.1).

Таблица 14.3. *Некоторые значения перечисления* `MouseButton` *пространства имен Qt*

Константа	Значение	Описание
`NoButton`	0	Кнопки мыши не нажаты
`LeftButton`	1	Нажата левая кнопка мыши
`RightButton`	2	Нажата правая кнопка мыши
`MidButton`	4	Нажата средняя кнопка мыши

При вызове метода `mouseDoubleClickEvent()` метод `mousePressEvent()` вызывается дважды, поскольку двойной щелчок обрабатывается как два простых нажатия. По умолчанию интервал двойного щелчка составляет 400 мс, а для изменения этого интервала нужно вызвать метод `setDoubleClickInterval()` класса `QApplication`.

Следующая программа демонстрирует обработку событий мыши. Ее реализация приведена в листингах 14.1–14.3. На рис. 14.2 показан момент перемещения указателя мыши с нажатой левой кнопкой и с нажатой клавишей <Ctrl>.

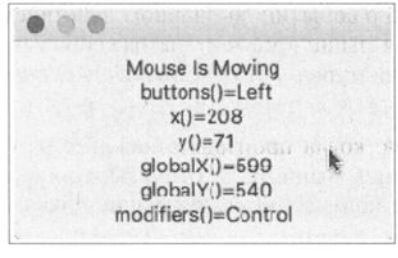


Рис. 14.2. Виджет, получающий события мыши

В функции `main()` (см. листинг 14.1) выполняется создание объекта реализованного нами класса `MouseObserver` и с помощью метода `resize()` устанавливаются размеры окна виджета. Вызов метода `show()` отображает виджет на экране.

Листинг 14.1. Обработка событий мыши (файл main.cpp)

```
#include <QtWidgets>
#include "MouseObserver.h"

int main(int argc, char** argv)
{
    QApplication  app(argc, argv);
    MouseObserver wgt;

    wgt.resize(250, 130);
    wgt.show();

    return app.exec();
}
```

В классе `MouseObserver` (листинг 14.2) определены три метода для обработки событий мыши:

- `mousePressEvent()` — для нажатия на кнопку мыши;
- `mouseReleaseEvent()` — для отпускания кнопки мыши;
- `mouseMoveEvent()` — для перемещения мыши.

Метод `dumpEvent()` выводит информацию о состоянии события мыши. Для предоставления информации (в виде строки) о клавишах-модификаторах и кнопках мыши в классе определены методы `modifiersInfo()` и `buttonsInfo()`.

Листинг 14.2. Определение класса `MouseObserver` (файл MouseObserver.h)

```
#pragma once

#include <QtWidgets>

// ============================================================================
class MouseObserver : public QLabel {
public:
    MouseObserver(QWidget* pwgt = 0);

protected:
    virtual void mousePressEvent  (QMouseEvent* pe);
    virtual void mouseReleaseEvent(QMouseEvent* pe);
    virtual void mouseMoveEvent   (QMouseEvent* pe);

    void    dumpEvent     (QMouseEvent* pe, const QString& strMessage);
    QString modifiersInfo (QMouseEvent* pe                            );
    QString buttonsInfo   (QMouseEvent* pe                            );
};
```

В конструкторе класса (листинг 14.3) вызов метода `setAlignment()` с параметром `AlignCenter` выполняет центровку всей выводимой нами информации. В методах `mousePressEvent()`, `mouseReleaseEvent()` и `mouseMoveEvent()`, отслеживающих события мыши, вызывается один и тот же метод — `dumpEvent()`, в который передаются указатель на объект события и строка, информирующая о методе обработки этого события. Метод `modifiersInfo()` предоставляет в виде строки информацию о клавишах-модификаторах, нажатие которых проверяется вызовом метода `modifiers()`. Информацию о нажатых кнопках мыши предоставляет метод `buttonsInfo()`. Вся информация собирается в методе `dumpEvent()` в одну строку и выводится при помощи метода `setText()`.

Листинг 14.3. Конструктор `MouseObserver()` (файл MouseObserver.cpp)

```
#include "MouseObserver.h"

// ----------------------------------------------------------------------
MouseObserver::MouseObserver(QWidget* pwgt /*= 0*/) : QLabel(pwgt)
{
    setAlignment(Qt::AlignCenter);
```

```
    setText("Mouse interactions\n(Press a mouse button)");
}

// ------------------------------------------------------------------------
/*virtual*/void MouseObserver::mousePressEvent(QMouseEvent* pe)
{
    dumpEvent(pe, "Mouse Pressed");
}

// ------------------------------------------------------------------------
/*virtual*/void MouseObserver::mouseReleaseEvent(QMouseEvent* pe)
{
    dumpEvent(pe, "Mouse Released");
}
// ------------------------------------------------------------------------
/*virtual*/ void MouseObserver::mouseMoveEvent(QMouseEvent* pe)
{
    dumpEvent(pe, "Mouse Is Moving");
}

// ------------------------------------------------------------------------
void MouseObserver::dumpEvent(QMouseEvent* pe, const QString& strMsg)
{
    setText(strMsg
            + "\n buttons()=" + buttonsInfo(pe)
            + "\n x()=" + QString::number(pe->x())
            + "\n y()=" + QString::number(pe->y())
            + "\n globalX()=" + QString::number(pe->globalX())
            + "\n globalY()=" + QString::number(pe->globalY())
            + "\n modifiers()=" + modifiersInfo(pe)
           );
}

// ------------------------------------------------------------------------
QString MouseObserver::modifiersInfo(QMouseEvent* pe)
{
    QString strModifiers;

    if(pe->modifiers() & Qt::ShiftModifier) {
        strModifiers += "Shift ";
    }
    if(pe->modifiers() & Qt::ControlModifier) {
        strModifiers += "Control ";
    }
    if(pe->modifiers() & Qt::AltModifier) {
        strModifiers += "Alt";
    }
    return strModifiers;
}
```

```
// ------------------------------------------------------------------------
QString MouseObserver::buttonsInfo(QMouseEvent* pe)
{
    QString strButtons;

    if(pe->buttons() & Qt::LeftButton) {
        strButtons += "Left ";
    }
    if(pe->buttons() & Qt::RightButton) {
        strButtons += "Right ";
    }
    if(pe->buttons() & Qt::MidButton) {
        strButtons += "Middle";
    }
    return strButtons;
}
```

Класс *QWheelEvent*

Учитывая, что в последнее время часто используются мыши, оснащенные колесиком, рекомендуется реализовывать метод для обработки события прокрутки колеса — `QWheelEvent`, который унаследован от `QInputEvent` (см. рис. 14.1).

Объект класса `QWheelEvent` содержит информацию о событии, вызванном колесиком мыши. Объект события передается в метод `wheelEvent()` и содержит информацию об угле и направлении, в котором было повернуто колесико, а также о позиции указателя мыши, статусе кнопок мыши и некоторых клавиш клавиатуры. Наряду с методами `buttons()`, `pos()` и `globalPos()`, которые полностью идентичны методам класса события `QMouseEvent`, в классе `QWheelEvent` имеется метод `angleDelta()`, с помощью которого можно узнать угол поворота колесика мыши. Положительное значение говорит о том, что колесико было повернуто от себя, а отрицательное значение — на себя.

Не во всех случаях на компьютере имеется мышь с колесиком — например, на компьютерах Mac чаще всего ее роль выполняет *тачпад* (touchpad). Поэтому есть метод `pixelDelta()`, который возвращает значение в пикселах, предназначенных для прокручивания области. В программе важно использовать оба метода — чтобы поддержать оба устройства. Например:

```
void MyWidget::wheelEvent(QWheelEvent* pe)
{
    QPoint nAngle = pe->angleDelta();
    QPoint nPixels = pe->pixelDelta();
    if (!nAngle.isNull()) {
        // используем значение nAngle
    }
    else if (!nPixels.isNull()) {
        // используем значение nPixels
    }
}
```

Методы *enterEvent()* и *leaveEvent()*

Эти методы вызываются в том случае, когда указатель мыши попадает или покидает область виджета. Их можно переопределить, например, в том случае, если требуется изменить внешний вид виджета. Метод `enterEvent()` получает объект события типа `QEvent` и вызывается каждый раз, когда указатель мыши входит в область виджета. Метод `leaveEvent()` получает объект события типа `QEvent` и вызывается, когда указатель мыши выходит за пределы области виджета.

Событие таймера. Класс *QTimerEvent*

Объект класса `QTimerEvent` содержит информацию о событии, инициированном таймером. Этот объект передается в метод обработки события `timerEvent()`. Объект события содержит идентификационный номер таймера. Например, для класса, унаследованного от класса `QWidget`, метод обработки этого события может выглядеть следующим образом:

```
void MyClass::timerEvent(QTimerEvent* e)
{
    if (event->timerId() == myTimerId) {
        // Выполнить какие-либо действия
    }
     else {
        QWidget::timerEvent(e); // Передать событие дальше
    }
}
```

Более подробную информацию о таймерах вы найдете в *главе 37*.

События перетаскивания (drag & drop)

Этой теме посвящена *глава 29*. Поэтому здесь мы ограничимся кратким описанием классов событий.

Класс *QDragEnterEvent*

Класс события `QDragEnterEvent` унаследован от класса `QDragMoveEvent` (см. рис. 14.1). Объект класса содержит данные события перетаскивания. Если пользователь, перетаскивая объект, попадает в область виджета, то вызывается метод `dragEnterEvent()`.

Класс *QDragLeaveEvent*

Объект класса `QDragLeaveEvent` содержит данные события перетаскивания в том случае, если пользователь, перетаскивая объект, выходит за область виджета. При этом вызывается метод `dragLeaveEvent()`.

Класс *QDragMoveEvent*

Этот класс служит для представления данных события перетаскивания в тот момент, когда данные находятся в области виджета. Возникновение этого события приводит к вызову метода `dragMoveEvent()`.

Класс *QDropEvent*

Объект класса `QDropEvent` передается в метод `dropEvent()` при отпускании объекта в принимающей области виджета.

Остальные классы событий

Класс *QChildEvent*

Это событие происходит в момент создания или удаления объекта-потомка. Объект события передается в метод `childEvent()`, который определен в классе `QObject`. Вызовом метода `QChildEvent::child()` можно получить указатель на этот объект. При помощи методов `QChildEvent::added()` и `QChildEvent::removed()` можно узнать о создании и удалении объекта-потомка.

Класс *QCloseEvent*

Событие класса `QCloseEvent` создается при закрытии окна виджета. Оно может быть вызвано пользователем или методом `QWidget::close()`. Объект класса `QCloseEvent` передается в метод `closeEvent()`, в котором можно спросить пользователя, действительно ли он хочет закрыть приложение. Это имеет смысл в тех случаях, когда пользователь не сохранил свои данные.

При помощи методов `accept()` и `ignore()` устанавливается флаг, сообщающий о согласии получателя события закрыть окно. Вызов `accept()` приведет к тому, что после возвращения из этого метода окно будет спрятано методом `hide()`. Вызов `ignore()` оставит окно без изменений.

Класс *QHideEvent*

Это событие создается при нажатии пользователем кнопки свертывания приложения. Оно также может быть вызвано методом `hide()`, делающим виджет невидимым. Объект события класса `QHideEvent` передается в метод `hideEvent()`.

Класс *QMoveEvent*

Событие класса `QMoveEvent` возникает при перемещении виджета. Для виджетов верхнего уровня это соответствует перемещению его окна. Объект события класса `QMoveEvent` передается в метод `moveEvent()` и содержит информацию о старых и новых координатах виджета, которые можно получить вызовом методов `pos()` и `oldPos()`.

Класс *QShowEvent*

Событие генерируется при создании виджета и при вызове метода `show()`. Объект события `QShowEvent` передается в метод `showEvent()`.

Класс *QResizeEvent*

Пользователь может изменять размеры окна при помощи мыши. При этом создается объект события `QResizeEvent`. Объект передается в метод `resizeEvent()` и содержит информацию о старых и новых размерах виджета, которые можно получить вызовом методов `size()` и `oldSize()`.

Реакции на изменение размеров виджета могут быть следующими:

- перерисовка содержимого окна;
- изменение размеров виджетов-потомков.

Следующий пример (листинг 14.4) демонстрирует перезапись метода `resizeEvent()`. В окне отображается информация о текущей ширине и высоте окна, которая обновляется при изменении его размера (рис. 14.3).

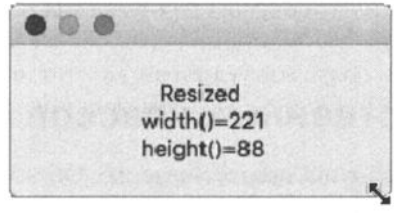


Рис. 14.3. Изменение размеров окна при помощи мыши

В листинге 14.4 в функции `main()` создается виджет (определенного нами класса `ResizeObserver`), размеры которого изменяются вызовом метода `resize()`. Метод `resizeEvent()` вызывается всякий раз, когда пользователь или программа изменяют размеры окна. Метод `size()` объекта события возвращает объект класса `QSize`, при помощи которого можно узнать текущую ширину и высоту виджета (методы `width()` и `height()`). Значения ширины и высоты передаются в статический метод `QString::number()`. Таким образом они преобразуются в строки и присоединяются к основному сообщению, которое выводится при помощи слота `QLabel::setText()`.

Листинг 14.4. Перезапись метода `resizeEvent()` (файл main.cpp)

```
#include <QtWidgets>

// ========================================================================
class ResizeObserver : public QLabel {
public:
    ResizeObserver(QWidget* pwgt = 0) : QLabel(pwgt)
    {
        setAlignment(Qt::AlignCenter);
    }

protected:
    virtual void resizeEvent(QResizeEvent* pe)
    {
        setText(QString("Resized")
                + "\n width()=" + QString::number(pe->size().width())
                + "\n height()=" + QString::number(pe->size().height())
               );
    }
};
```

```
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication   app(argc, argv);
    ResizeObserver wgt;

    wgt.resize(250, 130);
    wgt.show();

    return app.exec();
}
```

Реализация собственных классов событий

Если вам не будет хватать событий, предоставляемых Qt, и понадобится определить свое собственное, то необходимо поступить так, как это делается для всех классов событий Qt, а именно — унаследовать базовый класс для всех событий `QEvent`. В конструкторе `QEvent` нужно передать идентификационный номер для типа события, который должен быть больше, чем значение `QEvent::User` (равное 1000 — см. далее табл. 14.4), — чтобы не создать конфликт с уже определенными типами. В созданном событии можно реализовать все необходимые вам методы для передачи дополнительной информации. Например:

```
class MyEvent : public QEvent {
public:
    MyEvent() : QEvent((Type)(QEvent::User + 200))
    {
    }

    QString info()
    {
        return "CustomEvent";
    }
};
```

Свои собственные события можно высылать с помощью методов `QCoreApplication::sendEvent()` или `QCoreApplication::postEvent()` (см. *главу 16*), а получать методами `QObject::event()` или `QObject::customEvent()`.

Переопределение метода *event()*

Все возникающие в системе события направляются в очередь, из которой они извлекаются циклом событий, находящимся в методе `exec()` объекта приложения `QCoreApplication` или `QApplication`. Объект класса `QApplication` посылает события тем виджетам, которым они предназначены. Все объекты событий поступают в метод `event()`, в котором определяется их тип и осуществляется вызов специализированных методов обработки, предназначенных для этих событий, — например, вызов метода `mousePressEvent()` при нажатии кнопки мыши (рис. 14.4).

Метод `event()`, как и все остальные специализированные методы обработки событий, — виртуальный. Его можно переопределить, но делать так следует лишь в тех случаях, когда в этом есть острая необходимость, поскольку такое переопределение может изрядно услож-



Рис. 14.4. Схема доставки и обработки событий

нить исходный код всей программы. Если все события будут обрабатываться только одним методом `event()`, это может привести к появлению громоздкого метода, содержащего несколько тысяч строк для обработки всех возможных событий. Поэтому, если есть возможность, лучше всего перезаписывать соответствующие специализированные методы для обработки событий. А обработку событий в методе `event()` применять для тех типов событий, для которых не существует специализированных методов обработки.

В метод `event()` передается указатель на объект типа `QEvent`. Все события унаследованы от класса `QEvent`, который содержит атрибут, представляющий собой целочисленный идентификатор типа события, и с его помощью можно всегда привести указатель на объект класса `QEvent` к нужному типу. В методе `event()` программа должна определить, какое событие произошло. Для облегчения этой задачи существует метод `type()`. Возвращаемое им значение можно сравнить с предопределенной константой (табл. 14.4), что даст возможность привести это событие к правильному типу.

Таблица 14.4. *Некоторые типы событий*

Константа	Значение	Константа	Значение
None	0	Leave	11
Timer	1	Paint	12
MouseButtonPress	2	Move	13
MouseButtonRelease	3	Resize	14
MouseButtonDblClick	4	Destroy	16
MouseMove	5	Show	17
KeyPress	6	Hide	18
KeyRelease	7	Close	19
FocusIn	8	ParentChange	21
FocusOut	9	ThreadChange	22
Enter	10	WindowActivate	24

Таблица 14.4 (окончание)

Константа	Значение	Константа	Значение
WindowDeactivate	25	FontChange	97
ShowToParent	26	EnabledChange	98
HideToParent	27	ActivationChange	99
Wheel	31	StyleChange	100
WindowTitleChange	33	IconTextChange	101
WindowIconChange	34	ModifiedChange	102
ApplicationWindowIconChange	35	WindowBlocked	103
ApplicationFontChange	36	WindowUnblocked	104
ApplicationLayoutDirectionChange	37	WindowStateChange	105
ApplicationPaletteChange	38	MouseTrackingChange	109
PaletteChange	39	ToolTip	110
Clipboard	40	WhatsThis	111
SockAct	50	StatusTip	112
ShortcutOverride	51	ActionChanged	113
DeferredDelete	52	ActionAdded	114
DragEnter	60	ActionRemoved	115
DragMove	61	FileOpen	116
DragLeave	62	Shortcut	117
Drop	63	WhatsThisClicked	118
ChildAdded	68	ToolBarChange	120
ChildPolished	69	ApplicationActivated	121
ChildRemoved	71	ApplicationDeactivate	122
PolishRequest	74	QueryWhatsThis	123
Polish	75	EnterWhatsThisMode	124
LayoutRequest	76	LeaveWhatsThisMode	125
UpdateRequest	77	ZOrderChange	126
UpdateLater	78	HoverEnter	127
ContextMenu	82	HoverLeave	128
InputMethod	83	HoverMove	129
TabletMove	87	ParentAboutToChange	131
LocaleChange	88	WinEventAct	132
LanguageChange	89	TouchBegin	194
LayoutDirectionChange	90	TouchUpdate	195
StyleChange	91	TouchEnd	196
TabletPress	92	TouchCancel	209
TabletRelease	93	User	1000
IconDrag	96		

Перезапись метода event() для класса, унаследованного, например, от класса QWidget, может выглядеть следующим образом:

```
bool MyClass::event(QEvent* pe)
{
    if (pe->type() == QEvent::KeyPress) {
        QKeyEvent* pKeyEvent = static_cast<QKeyEvent*>(pe);
        if (pKeyEvent->key() == Qt::Key_Tab) {
            // Выполнить какие-либо действия
            return true;
        }
    }
    if (pe->type() == QEvent::Hide) {
        // Выполнить какие-либо действия
        return true;
    }
    return QWidget::event(pe);
}
```

Метод возвращает true в том случае, если событие было обработано и не требует передачи дальше. После этого событие удаляется из очереди событий. При возвращении false событие будет передано дальше — виджету-предку. Если ни один из предков не сможет обработать событие, то оно будет просто проигнорировано и удалено из очереди событий.

Мультитач

Термин *мультитач* (Multi-touch) переводится с английского языка как множественное касание, то есть одновременное прикосновение к двум и более точкам. Если вы являетесь обладателем какого-нибудь мобильного устройства, например смартфона или планшета, то уже догадываетесь, что подразумевается под этим термином. С сенсорным экраном мобильного устройства пользователи взаимодействуют при помощи пальцев, но пальцы — это не курсор мыши, который существует в единственном экземпляре, пальцев на обеих руках аж 10. Самые распространенные действия — это увеличение какого-нибудь изображения на экране для того, чтобы лучше его разглядеть, либо повернуть его в удобное для просмотра положение. Для этих операций используются два пальца. Но есть операции, в которых используется и больше двух пальцев, — например, игра для двух игроков, которые взаимодействуют с одним экраном устройства одновременно или нажатие на виртуальном пианино от трех и более клавиш сразу и т. д. Подобного рода действия и называются «мультитач».

Виджеты Qt способны получать мультитач-события, хотя по умолчанию и игнорируют их. Поэтому, для того чтобы активизировать в виджете получение мультитач-событий, необходимо вызвать метод setAttribute() и произвести в виджете установку атрибута Qt::WA_AcceptsTouchEvents.

Получаемые мультитач-события содержатся в объектах класса QTouchEvent. А обрабатываются они в центральном методе обработки событий event(), который мы только что рассмотрели.

Активизация получения мультитач-событий

Для активизации получения мультитач-событий у виджета видовой прокрутки QAbstractScrollArea атрибут Qt::WA_AcceptsTouchEvents нужно устанавливать в виджете,

указатель на который возвращает метод `QAbstactScrollArea::viewPort()`, а для отслеживания и обработки этого события необходимо перезаписать метод `QAbstractScrollArea::viewportEvent()`.

Еще про обработку мультитач-событий

Объекты класса `QGraphicsItem` так же, как и виджеты, способны получать и обрабатывать мультитач-события.

Мультитач-события в методе обработки можно «отловить» с помощью трех основных идентификаторов:

- идентификатор `QEvent::TouchBegin` приходит первым в метод обработки и открывает цепочку мультитач-событий. Этот идентификатор не может следовать один за другим, потому что вслед за ним должны следовать `QEvent::TouchUpdate`;
- идентификатор `QEvent::TouchUpdate` сигнализирует об мультитач-активности пользователя/пользователей и подразумевает, что следом будут идти дальнейшие мультитач-события;
- идентификатор `QEvent::TouchEnd` сообщает о том, что пришло последнее событие цепочки мультитач-событий.

Теперь рассмотрим класс `QTouchEvent`. Этот класс мультитач-события содержит список мест прикосновения пользователем/пользователями. С помощью объектов этого класса можно также узнать, с каким типом устройства взаимодействуют пользователи: с сенсорным экраном: `QTouchDevice::TouchScreen` или с сенсорной панелью-тачпадом: `QTouchDevice::TouchPad`. Для этого нужно вызвать метод `QTouchEvent::device()`, который вернет указатель на объект класса `QTouchDevice`. Это разделение важно потому, что в случае с сенсорным экраном мы можем узнать абсолютные координаты мест прикосновения, а в случае с сенсорной панелью — только лишь позиции относительно курсора экрана. Подробная информация о каждом из мест прикосновения хранится в списке объектов вложенного класса `QTouchEvent::TouchPoint`. Она включает в себя:

- целочисленный идентификатор точки прикосновения, который может быть полезен для ее идентификации в программе. Для того чтобы получить идентификатор точки прикосновения, нужно вызвать метод `TouchPoint::id()`;
- статус: нажато — `Qt::TouchPointPressed`, перемещено — `Qt::TouchPointMoved`, неподвижно — `Qt::TouchPointStationary` и отпущено — `Qt::TouchPointReleased`. Получить информацию о статусе можно вызовом метода `TouchPoint::state()`;
- позиции: текущая — `TouchPoint::pos()`, предыдущая — `TouchPoint::lastPos()` и начальная — `TouchPoint::startPos()`. Эта информация может быть очень полезна, если вы захотите создать свои собственные *жесты множественных касаний* (gestrures);
- сила нажатия — в том случае, если устройство предоставляет эту информацию. Получить силу нажатия можно вызовом метода `TouchPoint::pressure()`.

Для того чтобы подвести итог сказанному, реализуем небольшое приложение, отображающее множественные касания в своем окне. При этом отображаться касания будут в виде их начальной и текущей точек, соединенных между собой линиями различных цветов (рис. 14.5).

В основной программе листинга 14.5 мы создаем виджет нашего класса `MultiTouchWidget`, который обладает поддержкой для множественных касаний.

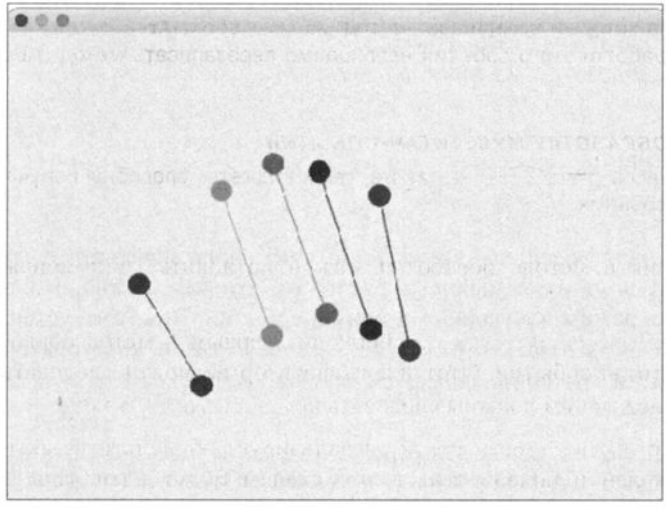


Рис. 14.5. Демонстрация мультитача с использованием пяти пальцев

Листинг 14.5. Отображение множественных касаний (файл main.cpp)

```
#include <QtWidgets>
#include "MultiTouchWidget.h"

int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    MultiTouchWidget wgt;

    wgt.resize(640, 480);
    wgt.show();

    return app.exec();
}
```

В листинге 14.6 определения класса `MultiTouchWidget` мы определяем атрибут класса `m_lstCols` — список различных цветов и атрибут класса `m_lstTps` — список точек касания. Далее переопределяем два виртуальных метода обработки событий `paintEvent()` и `event()`.

Листинг 14.6. Определение класса `MultiTouchWidget` (файл MultiTouchWidget.h)

```
#pragma once

#include <QWidget>

class MultiTouchWidget : public QWidget {
private:
    QList<QColor>                  m_lstCols;
    QList<QTouchEvent::TouchPoint> m_lstTps;
```

```
protected:
    virtual void paintEvent(QPaintEvent*);
    virtual bool event     (QEvent*     );

public:
    MultiTouchWidget(QWidget* pwgt = 0);
};
```

Как уже было сказано ранее, все виджеты по умолчанию игнорируют мультитач-события и преобразуют их в события мыши, а это не то, что нам нужно. Поэтому нам необходимо активировать режим получения мультитач-событий. Для этого устанавливаем атрибут `Qt::WA_AcceptTouchEvents` вызовом метода `setAttribute()` в конструкторе нашего класса (листинг 14.7). Затем мы инициализируем список цветов `m_lstCols` десятью различными цветовыми значениями.

Листинг 14.7. Конструктор `MultiTouchWidget()` (файл MultiTouchWidget.cpp)

```
MultiTouchWidget::MultiTouchWidget(QWidget* pwgt/*= 0*/) : QWidget(pwgt)
{
    setAttribute(Qt::WA_AcceptTouchEvents);

    m_lstCols << Qt::cyan << Qt::green << Qt::blue << Qt::black
              << Qt::red << Qt::magenta << Qt::darkYellow
              << Qt::gray << Qt::darkCyan << Qt::darkBlue;
}
```

В центральном методе получения и обработки событий `event()` (листинг 14.18) мы «отлавливаем» события `QTouchEvent`, используя идентификаторы `QEvent::TouchBegin`, `QEvent::TouchUpdate` и `QEvent::TouchEnd`, и записываем списки объектов точек касания в атрибут класса `m_lstTps`, после чего вызываем метод `update()`, который приводит к неявному вызову события рисования `paintEvent()`.

Листинг 14.8. Центральный метод получения событий (файл MultiTouchWidget.cpp)

```
/*virtual*/ bool MultiTouchWidget::event(QEvent* pe)
{
    switch (pe->type()) {
    case QEvent::TouchBegin:
    case QEvent::TouchUpdate:
    case QEvent::TouchEnd:
        {
        QTouchEvent* pte = static_cast<QTouchEvent*>(pe);
        m_lstTps = pte->touchPoints();
        update();
        }
    default:
        return QWidget::event(pe);
    }
    return true;
}
```

В методе обработки события рисования (листинг 14.9) при помощи цикла `foreach` мы итерируем по списку точек касания и проверяем информацию о статусе (метод `TouchPoint::state()`) каждого из элементов. Если статус `Qt::TouchPointStationary` (то есть неподвижно), то мы игнорируем точку касания и переходим с помощью оператора `continue` к следующему витку итерации. В противном случае мы отображаем ее начальную и текущую позиции вызовом двух методов `QPainter::drawEllipse()` и отображаем линию между ними вызовом `QPainter::drawLine()`. Цвета для отображения мы задаем для каждой точки индивидуально, используя список цветовых значений `m_lstCols` и ее идентификационный номер, который получаем вызовом метода `TouchPoint::id()`.

Листинг 14.9. Метод обработки события рисования (файл MultiTouchWidget.cpp)

```
/*virtual*/ void MultiTouchWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);

    int nColsCount = m_lstCols.count();
    foreach (QTouchEvent::TouchPoint tp, m_lstTps) {
        switch (tp.state()) {
        case Qt::TouchPointStationary:
            continue;
        default:
            QColor c(m_lstCols.at(tp.id() % nColsCount));
            painter.setPen(c);
            painter.setBrush(c);

            QRectF r1(tp.pos(), QSize(20, 20));
            QRectF r2(tp.startPos(), QSize(20, 20));
            painter.drawEllipse(r1.translated(-10, -10));
            painter.drawEllipse(r2.translated(-10, -10));

            painter.drawLine(tp.pos(), tp.startPos());
        }
    }
}
```

Для того чтобы создать простой жест поворота, в методе `event()` листинга 14.8 можно было бы поступить следующим образом: после присвоения списка точек касания `m_lstTps = pte->touchPoints()` проверить, состоит ли этот список строго из двух точек, так как этот жест выполняется при помощи двух пальцев. Если да, то вычислить угол поворота, используя начальные и текущие значения точек касания. И в завершение использовать значение полученного угла поворота, чтобы произвести операцию поворота. Вот как все сказанное может выглядеть в программном коде:

```
...
m_lstTps = pte->touchPoints();
if (lstTps.count() == 2) {
    QTouchEvent::TouchPoint pt1 = touchPoints.first();
    QTouchEvent::TouchPoint pt2 = touchPoints.last();
```

```
    QLineF line1(pt1.lastPos(), pt2.lastPos());
    QLineF line2(pt1.pos(), pt2.pos());
    qreal fAngle = line2.angleTo(line1);
    doRotation(fAngle);
}
...
```

Аналогичным образом можно создать также и жест масштабирования, который выполняется при помощи раздвигания и сдвигания двух пальцев.

Сохранение работоспособности приложения

В некоторых ситуациях при интенсивных действиях в вашей программе может случиться так, что графический интерфейс программы «замрет» и станет неспособным обрабатывать события, связанные с интерактивными действиями пользователя (нажатие кнопки мыши или клавиши клавиатуры). Возьмем следующий пример:

```
for (int i = 0; i < 1000; ++i) {
    // Выполнить трудоемкие вычисления
}
```

Этот код, если он исполняется в основном потоке, заблокирует на определенное время обработку событий, и, значит, пользовательские действия с интерфейсом все это время обрабатываться не будут, а также, если окно этой программы будет перекрыто другим, то оно не будет перерисовываться.

Один из лучших вариантов решения этой проблемы — исполнение подобного кода в отдельном потоке (см. *главу 38*). Более простой способ — вызов метода `QCoreApplication::processEvents()`, который позаботится о том, чтобы все накопившиеся в очереди события были обработаны. Для этого наш пример должен быть изменен следующим образом:

```
for (int i = 0; i < 1000; ++i) {
    // Выполнить трудоемкие вычисления
    qApp->processEvents(); // Доставить накопившиеся события
}
```

Теперь в каждой итерации цикла после выполнения действий осуществляется обработка событий, что дает программе возможность перед выполнением очередных действий «вдохнуть воздух» и отреагировать на накопившиеся события.

Резюме

Наступление события вызывается каким-либо действием со стороны пользователя, например щелчком кнопкой мыши, изменением размеров окна и т. п. Некоторые события вызываются самой программой. В этой главе вы узнали, что события являются механизмом оповещения более низкого уровня по сравнению с сигналами и слотами. Для получения доступа к информации о событии предусмотрен объект события, который передается в метод обработчика события. Все методы обработчиков событий, за исключением метода `event()`, относятся к группе специализированных обработчиков. Метод `event()` является центральным методом обработки событий. Сначала все события попадают в него, а он осуществляет вызов специализированных методов для обработки соответствующих событий, — напри-

мер, при нажатии клавиши на клавиатуре вызывается обработчик `keyPressEvent()`. В метод `event()` передается в качестве аргумента указатель на объект класса `QEvent`, который содержит информацию о событиях. Метод `QEvent::type()` возвращает целочисленный идентификатор типа события.

При написании собственных виджетов не следует обрабатывать все события в методе `event()`, так как это может изрядно усложнить код. Более правильным подходом является перезапись специализированных методов обработки событий.

Событие перерисовки окна возникает тогда, когда виджет был частично или полностью перекрыт другим окном. Объект события `QPaintEvent` содержит информацию об участке, который должен быть перерисован. Для обработки события необходимо переопределить метод `paintEvent()`.

События мыши обрабатываются методами `mousePressEvent()`, `mouseMoveEvent()`, `mouseReleaseEvent()` и `mouseDoubleClickEvent()`. Для определения местоположения указателя мыши можно воспользоваться методами `globalX()`, `globalY()`, `x()`, `y()`, `pos()` или `globalPos()`.

Для обработки событий клавиатуры предназначены методы `keyPressEvent()` и `keyReleaseEvent()`: метод `keyPressEvent()` вызывается каждый раз, когда пользователь нажимает на клавиатуре одну из клавиш, а метод `keyReleaseEvent()` вызывается при отпускании клавиши.

Для определения своего собственного события необходимо задать число (идентификатор), которое будет определять тип события и не должно совпадать с уже существующими идентификаторами типов событий.

Виджеты Qt способны получать и обрабатывать мультитач-события. Для этого необходимо активировать в них режим их получения и отслеживать мультитач-события при помощи идентификаторов в центральном методе обработки событий виджета.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/14-510/** или с помощью следующего QR-кода (рис. 14.6):



Рис. 14.6. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 15

Фильтры событий

Тот, кто спрашивает, всегда получит ответ.
Притчи Камеруна

Как правило, событие передается тому объекту, над которым было осуществлено действие, но иногда требуется его обработка в другом объекте. В библиотеке Qt предусмотрен очень мощный механизм перехвата событий, который позволяет объекту фильтра просматривать события раньше объекта, которому они предназначены, и принимать решение по своему усмотрению — обрабатывать их и/или передавать дальше. Такой мониторинг осуществляется с помощью метода `QObject::installEventFilter()`, в который передается указатель на объект, осуществляющий фильтрацию событий.

Важно то, что установка фильтров событий происходит не на уровне классов, а на уровне самих объектов. Это дает возможность вместо того, чтобы наследовать класс или изменять уже имеющийся (что не всегда возможно), просто воспользоваться объектом фильтра. Для настройки на определенные события необходимо создать класс фильтра, установив его в нужном объекте. Все получаемые события и их обработка будут касаться только тех объектов, в которых установлены фильтры.

Фильтры событий можно использовать, например, в тех случаях, когда нужно добавить функциональность к каким-либо уже реализованным классам, не наследуя при этом каждый из них. После реализации класса фильтра его объекты можно будет устанавливать в любых объектах, созданных от наследующих `QObject` классов. Это позволит сэкономить время на реализацию, так как потребуется написать меньше кода, и вместе с тем значительно сократит временны́е затраты на отладку программы. Разработчику больше не придется заботиться о методах обработки событий для каждого из классов в отдельности, потому что это будет выполняться централизованно, одним классом фильтра, имеющим силу для всех объектов, в которых он был установлен.

Реализация фильтров событий

Чтобы реализовать класс фильтра, нужно унаследовать класс `QObject` и переопределить метод `eventFilter()`. Этот метод будет вызываться при каждом событии, предназначенном для объекта, в котором установлен фильтр событий до его получения. Метод имеет два параметра: первый — это указатель на объект, для которого предназначено событие, а второй — указатель на сам объект события.

Если метод `eventFilter()` возвращает значение `true`, то это означает, что событие не должно передаваться дальше, а возвращение `false` говорит о том, что событие должно быть передано объекту, для которого оно и было предназначено.

Приведенный далее пример (листинги 15.1–15.3) демонстрирует работу фильтра, который устанавливается в трех виджетах (рис. 15.1). Щелчок мыши на любом из них приводит к появлению окна сообщения, информирующего об имени класса виджета.

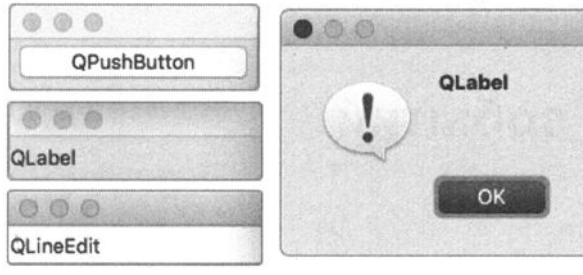


Рис. 15.1. Программа, демонстрирующая перехват события

В функции `main()`, приведенной в листинге 15.1, создаются три виджета: `QLineEdit`, `QLabel` и `QPushButton`. В каждом из них с помощью метода `installEventFilter()` устанавливается объект, созданный от одного и того же класса фильтра событий. В конструктор создаваемого фильтра в качестве предка передается виджет, в котором устанавливается фильтр. Это позволит при уничтожении виджета автоматически уничтожить и объект установленного в нем фильтра.

Листинг 15.1. Программа, демонстрирующая перехват события (файл main.cpp)

```
#include <QtWidgets>
#include "MouseFilter.h"

int main (int argc, char** argv)
{
    QApplication app(argc, argv);

    QLineEdit txt("QLineEdit");
    txt.installEventFilter(new MouseFilter(&txt));
    txt.show();

    QLabel lbl("QLabel");
    lbl.installEventFilter(new MouseFilter(&lbl));
    lbl.show();

    QPushButton cmd("QPushButton");
    cmd.installEventFilter(new MouseFilter(&cmd));
    cmd.show();

    return app.exec();
}
```

В листинге 15.2 обратите внимание на прототип метода `eventFilter()`, который получает не только указатель на объект события, но и указатель на сам объект, для которого это событие предназначено. Объект фильтра может делать с этим объектом все, что ему будет угодно, вплоть до удаления.

Листинг 15.2. Определение класса MouseFilter (файл MouseFilter.h)

```
#pragma once

#include <QObject>

// ============================================================================
class MouseFilter : public QObject {
protected:
    virtual bool eventFilter(QObject*, QEvent*);

public:
    MouseFilter(QObject* pobj = 0);

};
```

В листинге 15.3 метод `eventFilter()` отслеживает событие типа `QEvent::MouseButtonPress`, соответствующее нажатию одной из кнопок мыши. Если событие относится к этому типу, то выполняется преобразование указателя на объект события к указателю типа `QMouseEvent`. Затем вызывается метод `button()` класса `QMouseEvent`, чтобы узнать, какая из кнопок мыши была нажата. Если была нажата левая кнопка, то в информационном окне сообщения выводится имя класса виджета, возвращается значение `true` и событие дальше не передается. В остальных случаях возвращается значение `false` и событие передается дальше.

Листинг 15.3. Конструктор MouseFilter() (файл MouseFilter.cpp)

```
#include <QtWidgets>
#include "MouseFilter.h"

// ----------------------------------------------------------------------------
MouseFilter::MouseFilter(QObject* pobj/*= 0*/)
    : QObject(pobj)
{
}

// ----------------------------------------------------------------------------
/*virtual*/bool MouseFilter::eventFilter(QObject* pobj, QEvent* pe)
{
    if (pe->type() == QEvent::MouseButtonPress) {
        if (static_cast<QMouseEvent*>(pe)->button() == Qt::LeftButton) {
            QString strClassName = pobj->metaObject()->className();
            QMessageBox::information(0, "Class Name", strClassName);
            return true;
        }
    }
    return false;
}
```

В объектах возможна установка сразу нескольких фильтров, при этом последний установленный фильтр будет применяться первым.

Существует возможность глобальной установки фильтра событий, то есть фильтра, который будет действовать на все объекты приложения. Для этого нужно вызвать метод `installFilter()` объекта `QCoreApplication` или метод `QApplication()`. Этот фильтр будет получать и обрабатывать события раньше всех объектов приложения, то есть прежде, чем его получают сами объекты или их фильтры событий.

Такой метод может пригодиться при отладке приложения, но брать его за основу не рекомендуется, так как при этом снижается скорость доставки каждого отдельного события. Также возможно перезаписать метод диспетчеризации событий `QCoreApplication::notify()` для полного контроля доставки событий.

Резюме

Иногда возникает необходимость обработки события в другом объекте. В Qt предусмотрен очень мощный механизм перехвата событий, который позволяет без наследования классов изменять реакцию объектов на события. Тем самым экономится время на написание и отладку программы. Чтобы реализовать класс фильтра, нужно унаследовать класс `QObject` и переопределить метод `eventFilter()`.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/15-510/** или с помощью следующего QR-кода (рис. 15.2):



Рис. 15.2. QR-код для доступа на страницу комментариев к этой главе

ГЛАВА 16

Искусственное создание событий

> Компьютеры бесполезны. Они могут только давать вам ответы.
>
> *Пабло Пикассо*

Иногда возникает необходимость в событиях, созданных искусственно. Например, это полезно при отладке своей программы, чтобы имитировать действия пользователя.

Для генерации события можно воспользоваться одним из двух статических методов класса `QCoreApplication`: `sendEvent()` или `postEvent()`. Оба метода получают в качестве параметров указатель на объект, которому посылается событие, и адрес объекта события. Разница между ними состоит в том, что метод `sendEvent()` отправляет событие без задержек, то есть его вызов приводит к немедленному вызову метода события, в то время как метод `postEvent()` помещает его в очередь для дальнейшей обработки.

Рассмотрим это на примере приложения (листинги 16.1 и 16.2), имитирующего нажатие пользователем клавиш от <A> до <Z> (рис. 16.1).



Рис. 16.1. Программа, имитирующая нажатие пользователем клавиш клавиатуры

В исходном коде, приведенном в листинге 16.1, создается объект `txt` класса `QLineEdit`, который будет выступать в качестве поля ввода. В цикле происходит создание событий типа `QKeyEvent`. Первый параметр, передаваемый конструктору, задает тип события клавиатуры (здесь он соответствует событию нажатия клавиши клавиатуры `QEvent::KeyPress`). Если мы хотим имитировать клавиатуру, то после каждого события `KeyPress` должно следовать событие `KeyRelease` (событие отпускания клавиши клавиатуры), — в противном случае много виджетов, которым будет послано только одно событие нажатия, поведут себя неправильно, — например, в `QLineEdit` перестанет мигать курсор ввода. Второй параметр задает саму нажатую клавишу. Третий — указывает на клавиши-модификаторы, которые могли быть совместно нажаты, в нашем случае это значение равно `Qt::NoModifier` и означает, что никаких клавиш-модификаторов нажато не было (см. табл. 14.1). Последний, четвертый параметр указывает на представление клавиши в ASCII-коде (в примере это число начинается с 65, что соответствует заглавной букве «A», и циклически увеличивается на единицу). Как было сказано ранее, вызов метода `sendEvent()` не помещает объект события в системную очередь, а исполняет его сразу же после вызова, поэтому, чтобы не произошло утечки памяти, мы создаем объекты событий не динамически, при помощи оператора `new`, а как локаль-

ные объекты keyPress и keyRelease, которые будут автоматически разрушаться при завершении итераций цикла.

Листинг 16.1. Программа, имитирующая нажатие пользователем клавиш клавиатуры (файл main.cpp)

```
#include <QtWidgets>

int main (int argc, char** argv)
{
    QApplication app (argc, argv);

    QLineEdit txt("User input: ");
    txt.show();
    txt.resize(300, 20);

    int i;
    for (i = 0; i < Qt::Key_Z - Qt::Key_A + 1; ++i) {
        QChar ch      = 65 + i;
        int   nKey    = Qt::Key_A + i;
        QKeyEvent keyPress(QEvent::KeyPress, nKey, Qt::NoModifier, ch);
        QApplication::sendEvent(&txt, &keyPress);

        QKeyEvent keyRelease(QEvent::KeyRelease, nKey, Qt::NoModifier, ch);
        QApplication::sendEvent(&txt, &keyRelease);
    }

    return app.exec();
}
```

Если бы нам понадобилось симулировать нажатие мыши на какой-либо виджет, то реализация функций для этого могла бы выглядеть, как это показано в листинге 16.2.

Листинг 16.2. Симуляция нажатия мыши

```
void mousePress(QWidget*         pwgt,
                int              x,
                int              y,
                Qt::MouseButton  bt  = Qt::LeftButton,
                Qt::MouseButtons bts = Qt::LeftButton
               )
{
    if (pwgt) {
        QMouseEvent* pePress =
          new QMouseEvent(QEvent::MouseButtonPress,
                          QPoint(x, y),
                          bt,
                          bts,
                          Qt::NoModifier
                         );
        QApplication::postEvent(pwgt, pePress);
    }
}
```

Здесь в функции `mousePress()` мы первым параметром принимаем указатель на виджет, с которым хотим провести симуляцию нажатия, а вторым и третьим идут координаты позиции, в которой было выполнено нажатие. Четвертый и пятый параметры не обязательные и по умолчанию инициализируются нажатием на левую кнопку, что является самым частым действием при нажатии. Внутри функции мы проверяем действительность указателя на объект виджета, после чего создаем объект события `QMouseEvent`, инициализируем переданными в функцию параметрами и в завершение пересылаем событие виджету (указатель `pwgt`) вызовом метода `postEvent()`.

Модифицировать объекты событий возможно не всегда. При совместном создании искусственных событий с фильтрами можно осуществить подмену самих объектов событий. Более подробно фильтры событий рассмотрены в *главе 15*. В качестве показательного примера использования подобного перехвата события с целью его подмены можно назвать изменение назначения клавиш клавиатуры. Так как класс события клавиатуры не обладает методами, позволяющими его модифицировать, то каждое сообщение клавиатуры можно получить в объекте фильтра и перед передачей дальше подменить его другим.

В следующем примере (листинги 16.3 и 16.4) происходит подмена клавиши <Z> на клавишу <A>. При этом нажатие пользователем клавиши <Z> повлечет за собой отображение буквы **A** в поле ввода (рис. 16.2). Таким образом можно, например, имитировать измененную раскладку клавиатуры.

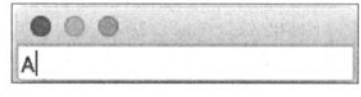


Рис. 16.2. Программа, демонстрирующая подмену события клавиатуры

В листинге 16.3 после создания объекта класса `QLineEdit` и вызова метода `show()` создается объект фильтра событий клавиатуры `pFilter`, в конструктор которого в качестве объекта-предка передается адрес на однострочное текстовое поле `txt`. После этого созданный фильтр привязывается к текстовому полю при помощи метода `installEventFilter()`.

Листинг 16.3. Программа, демонстрирующая подмену события клавиатуры (файл main.cpp)

```
#include <QtWidgets>
#include "KeyFilter.h"

int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QLineEdit txt;
    txt.show();

    KeyFilter* pFilter = new KeyFilter(&txt);
    txt.installEventFilter(pFilter);

    return app.exec();
}
```

В листинге 16.4 в методе `eventFilter()` отслеживается идентификатор события `QEvent::KeyPress`, который соответствует событию нажатия клавиши клавиатуры. При обнаружении этого события его объект преобразовывается к типу `QKeyEvent`, чтобы иметь возможность вызова метода `key()`, который определен в этом классе и позволяет получить код нажатой клавиши. Затем создается и высылается новое событие нажатия клавиши <A>. После этого возвращается значение `true`, и это означает, что событие не должно передаваться дальше. Если событие, переданное в параметрах метода `eventFilter()`, не удовлетворяет двум поставленным условиям, то этот метод вернет значение `false`, и, тем самым, событие будет передано дальше.

Листинг 16.4. Определение класса `KeyFilter` (файл KeyFilter.h)

```
#pragma once

#include <QtWidgets>

// ============================================================================
class KeyFilter : public QObject {
protected:
    bool eventFilter(QObject* pobj, QEvent* pe)
    {
        if (pe->type() == QEvent::KeyPress) {
            if (((QKeyEvent*)pe)->key() == Qt::Key_Z) {
                QKeyEvent keyEvent (QEvent::KeyPress,
                                    Qt::Key_A,
                                    Qt::NoModifier,
                                    "A"
                                   );
                QApplication::sendEvent(pobj, &keyEvent);
                return true;
            }
        }
        return false;
    }

public:
    KeyFilter(QObject* pobj = 0)
        : QObject(pobj)
    {
    }
};
```

Резюме

В этой главе мы узнали о возможности искусственного создания событий из самой программы. Для этого можно воспользоваться методами `QApplication::sendEvent()` или `QApplication:: postEvent()`. При совместном использовании указанных методов с механизмом фильтров событий искусственное создание событий позволяет осуществлять подмену объектов событий.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/16-510/** или с помощью следующего QR-кода (рис. 16.3):



Рис. 16.3. QR-код для доступа на страницу комментариев к этой главе



ЧАСТЬ IV

Графика и звук

Не стыдно не знать, стыдно не учиться.
Народная мудрость

Глава 17. Введение в компьютерную графику

Глава 18. Легенда о короле Артуре и контекст рисования

Глава 19. Растровые изображения

Глава 20. Работа со шрифтами

Глава 21. Графическое представление

Глава 22. Анимация

Глава 23. Работа с OpenGL

Глава 24. Вывод на печать

Глава 25. Разработка собственных элементов управления

Глава 26. Элементы со стилем

Глава 27. Мультимедиа

ГЛАВА 17

Введение в компьютерную графику

...как будто волшебный фонарик освещает изнутри образы на экране...
Т. С. Элиот. «Песнь любви Альфреда Пруфрока»

Графика — одна из наиболее быстро развивающихся отраслей компьютерной индустрии. Известно, что 70% информации человек воспринимает визуально, поэтому графика — это важнейший компонент для взаимодействия человека с компьютером.

Для программирования компьютерной графики часто используются такие классы геометрии, как точки, двумерные размеры, прямоугольники, а также специальные классы для хранения цветовых значений.

Классы геометрии

Группа классов геометрии ничего не отображает на экране. Основное их назначение состоит в задании расположения, размеров и в описании формы объектов.

Точка

Для задания точек в двумерной системе координат служат два класса: `QPoint` и `QPointF`. При этом точка обозначается парой чисел X и Y, где X — горизонтальная, а Y — вертикальная координаты. В отличие от обычного расположения координатных осей, при задании координат точки в Qt обычно подразумевается, что ось Y смотрит вниз (рис. 17.1).



Рис. 17.1. Создание и сложение точек

Класс QPoint описывает точку с целочисленными координатами, а QPointF — с вещественными. Интерфейс обоих классов одинаков, в него входят методы, позволяющие проводить различные операции с координатами, например сложение и вычитание с координатами другой точки. При сложении/вычитании точек выполняется попарное сложение/вычитание их координат X и Y. Следующий пример складывает две точки: pt1 и pt2 (см. рис. 17.1):

```
QPoint pt1(10, 20);
QPoint pt2(20, 10);
QPoint pt3; //(0, 0)
pt3 = pt1 + pt2;
```

Объекты точек можно умножать и делить на числа. Например:

```
QPoint pt(10, 20);
pt *= 2; // pt = (20, 40)
```

Для получения координат точки (X, Y) реализованы методы x() и y() соответственно. Изменяются координаты точки с помощью методов setX() и setY().

Можно получать ссылки на координаты точки, чтобы изменять их значения. Например:

```
QPoint pt(10, 20);
pt.rx() += 10; // pt = (20, 20)
```

Объекты точек можно сравнивать друг с другом при помощи операторов == (равно) и != (не равно). Например:

```
QPoint pt1(10, 20);
QPoint pt2(10, 20);
bool b = (pt1 == pt2); // b = true
```

Если необходимо проверить, равны ли координаты X и Y нулю, то вызывается метод isNull(). Например:

```
QPoint pt; // (0, 0)
bool b = pt.isNull(); // b = true
```

Метод manhattanLength() возвращает сумму абсолютных значений координат X и Y. Например:

```
QPoint pt(10, 20);
int n = pt.manhattanLength(); // n = 10 + 20 = 30
```

Этот метод был назван в честь улиц Манхэттена, расположенных перпендикулярно друг к другу. Возвращаемое значение является грубым приближением к $\sqrt{X^2+Y^2}$.

Двумерный размер

Классы QSize и QSizeF служат для хранения соответственно целочисленных и вещественных размеров. Оба класса обладают одинаковыми интерфейсами. Структура их очень похожа на QPoint, так как хранит две величины, над которыми можно проводить операции сложения/вычитания и умножения/деления.

Классы QSize и QSizeF, как и классы QPoint, QPointF, предоставляют операторы сравнения ==, != и метод isNull(), возвращающий значение true в том случае, если высота и ширина равны нулю.

Для получения ширины и высоты вызываются методы `width()` и `height()`. Изменить эти параметры можно с помощью методов `setWidth()` и `setHeight()`. При помощи методов `rwidth()` и `rheight()` получают ссылки на значения ширины и высоты. Например:

```
QSize size(10, 20);
int n = size.rwidth()++; // n = 11; size = (11, 20)
```

Помимо них, класс предоставляет метод `scale()`, позволяющий изменять размеры оригинала согласно переданному в первом параметре размеру. Второй параметр этого метода управляет способом изменения размера (рис. 17.2), а именно:

- `Qt::IgnoreAspectRatio` — изменяет размер оригинала на переданный в него размер;
- `Qt::KeepAspectRatio` — новый размер заполняет заданную площадь, насколько это будет возможно с сохранением пропорций оригинала;
- `Qt::KeepAspectRatioByExpanding` — новый размер может находиться за пределами переданного в `scale()`, заполняя всю его площадь.

Согласно рис. 17.2, изменение размеров `size1`, `size2` и `size3` может выглядеть следующим образом:

```
QSize size1(320, 240);
size1.scale(400, 600, Qt::IgnoreAspectRatio); // => (400, 600)

QSize size2(320, 240);
size2.scale(400, 600, Qt::KeepAspectRatio); // => (400, 300)

QSize size3(320, 240);
size3.scale(400, 600, Qt::KeepAspectRatioByExpanding); // => (800, 600)
```

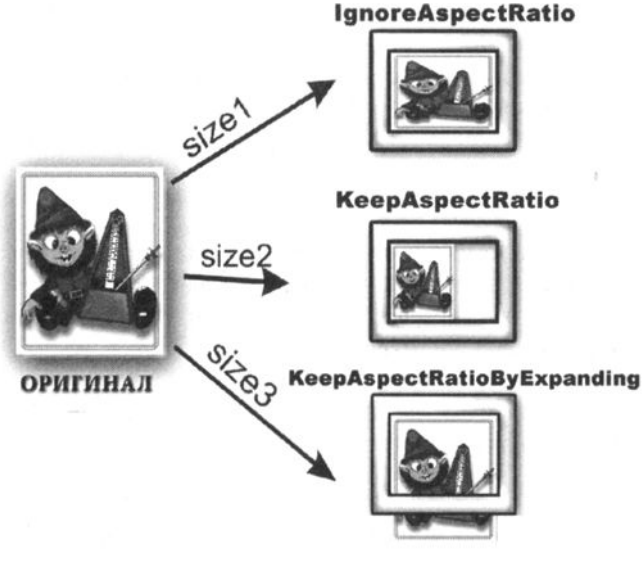


Рис. 17.2. Изменение размеров оригинала

Прямоугольник

Классы `QRect` и `QRectF` служат для хранения целочисленных и вещественных координат прямоугольных областей (точка и размер) соответственно. Задать прямоугольную область можно, например, передав в конструктор точку (верхний левый угол) и размер. Область, приведенная на рис. 17.3, создается при помощи следующих строк:

```
QPoint pt(10, 10);
QSize  size(20, 10);
QRect  r(pt, size);
```

Получить координаты *X* левой грани прямоугольника или *Y* верхней можно при помощи методов `x()` или `y()` соответственно. Для изменения этих координат нужно воспользоваться методами `setX()` и `setY()`.

Размер получают с помощью метода `size()`, который возвращает объект класса `QSize`. Можно просто вызвать методы, возвращающие составляющие размера: ширину `width()` и высоту `height()`. Изменить размер можно методом `setSize()`, а каждую его составляющую — методами `setWidth()` и `setHeight()`.

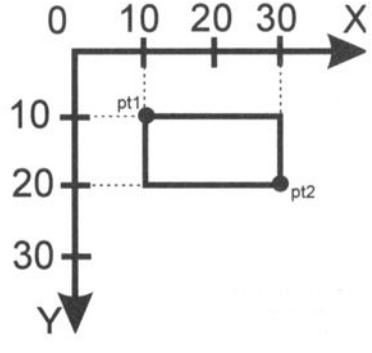


Рис. 17.3. Задание прямоугольной области точкой и размером

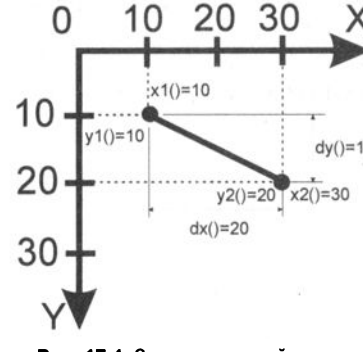


Рис. 17.4. Задание прямой линии двумя точками

Прямая линия

Классы `QLine` и `QLineF` описывают прямую линию или, правильней сказать, отрезок на плоскости в целочисленных и вещественных координатах соответственно. Позиции начальной точки можно получить при помощи методов `x1()` и `y1()`, а конечной — `x2()` и `y2()`. Аналогичного результата можно добиться вызовами `p1()` и `p2()`, которые возвращают объекты класса `QPoint`/`QPointF`, описанные ранее. Методы `dx()` и `dy()` возвращают величины горизонтальной и вертикальной проекций прямой на оси *X* и *Y* соответственно. Прямую, показанную на рис. 17.4, можно создать при помощи одной строки кода:

```
QLine line(10, 10, 30, 20);
```

Оба класса: `QLine` и `QLineF` — предоставляют операторы сравнения `==`, `!=` и метод `isNull()`, возвращающий логическое значение `true` в том случае, когда начальная и конечная точки не установлены.

Многоугольник

Многоугольник (или полигон) — это фигура, представляющая собой замкнутый контур, образованный ломаной линией. В Qt эту фигуру реализуют классы `QPolygon` и `QPolygonF`, в целочисленном и вещественном представлении соответственно. По своей сути эти классы являются массивами точек `QVector<QPoint>` и `QVector<QPointF>`. Самый простой способ инициализации объектов класса полигона — это использование оператора потока вывода `<<`. Треугольник представляет собой самую простую форму многоугольника (рис. 17.5), а его создание выглядит следующим образом:

```
QPolygon polygon;
polygon << QPoint(10, 20) << QPoint(20, 10) << QPoint(30, 30);
```

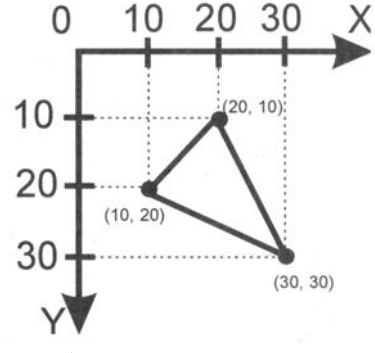


Рис. 17.5. Задание треугольника тремя точками

Цвет

Цвета, которые вы видите, есть не что иное, как свойство объектов материального мира, воспринимаемое нами как зрительное ощущение от воздействия света, имеющего различные электромагнитные частоты. На самом деле, человеческий глаз в состоянии воспринимать очень малый диапазон этих частот. Цвет с наибольшей частотой, которую в состоянии воспринять глаз, — фиолетовый, а с наименьшей — красный. Но даже в таком небольшом диапазоне находятся миллионы цветов. Одновременно человеческий глаз может воспринимать около 10 тысяч различных цветовых оттенков.

Цветовая модель — это спецификация в трехмерной или четырехмерной системе координат, которая задает все видимые цвета. В Qt поддерживаются цветовые модели: *RGB* (Red, Green, Blue — красный, зеленый, голубой), *RGBA* (Red, Green, Blue, Alpha — красный, зеленый, голубой, прозрачный), *CMYK* (Cyan, Magenta, Yellow и Key color — голубой, пурпурный, желтый и «ключевой» черный цвет), *HSV* (Hue, Saturation, Lightness — оттенок, насыщенность, светлота) и *HSV* (Hue, Saturation, Value — оттенок, насыщенность, значение).

Класс *QColor*

С помощью класса `QColor` можно сохранять цвета во всех упомянутых цветовых моделях. Его определение находится в заголовочном файле QColor. Объекты класса `QColor` можно сравнивать при помощи операторов `==` и `!=`, присваивать и создавать копии.

Цветовая модель RGB

Наиболее чувствителен глаз к зеленому цвету, потом следует красный, а затем — синий. На этих трех цветах и построена модель RGB (Red, Green, Blue — красный, зеленый, синий). Пространство цветов задает куб, длина ребер которого равна 255 (рис. 17.6) в целочисленном числовом представлении (либо единице в вещественном представлении).

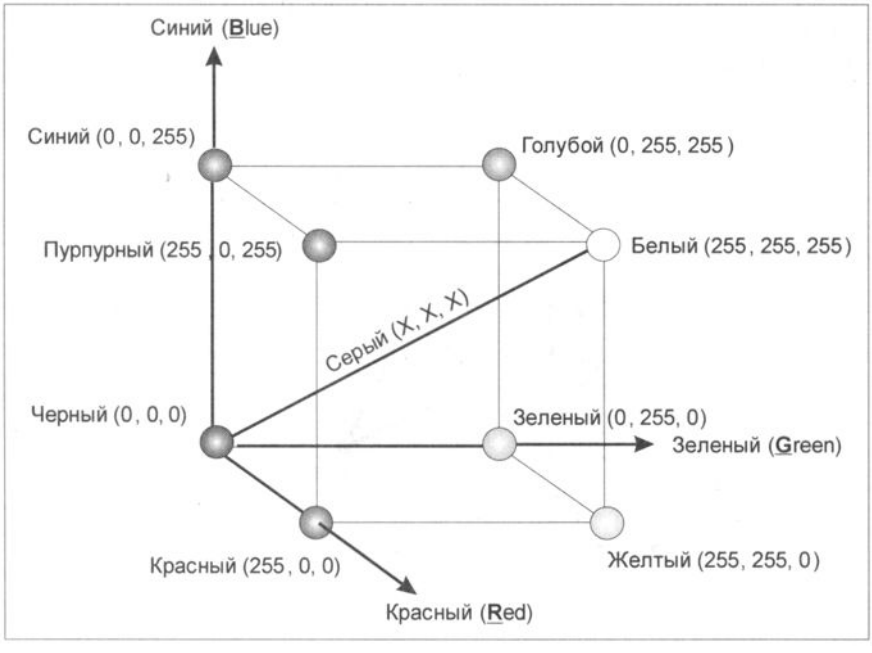


Рис. 17.6. Цветовая модель RGB

Как видно из рис. 17.6, цвет задается сразу тремя параметрами. Первый параметр задает оттенок красного, второй — зеленого, а третий — оттенок синего цвета. Диагональ куба, идущая от черного цвета к белому, — это оттенки серого цвета. Диапазон каждого из трех значений может изменяться в пределах от 0 до 255 (либо от 0 до 1 в вещественном представлении), где 0 означает полное отсутствие оттенка цвета, а 255 — его максимальную насыщенность.

Эта модель является «аддитивной», то есть посредством смешивания базовых цветов в различном процентном соотношении можно создать любой нужный цвет. Смешав, например, синий и зеленый, мы получим голубой цвет.

Для создания цветового значения RGB нужно просто передать в конструктор класса `QColor` три значения. Каналы цвета класса `QColor` могут содержать необязательный уровень прозрачности, значение для которого можно передавать в конструктор четвертым параметром. В первый параметр передается значение красного цвета, во второй — зеленого, в третий — синего, а в четвертый — уровень прозрачности. Например:

```
QColor colorBlue(0, 0, 255, 128);
```

Получить из объекта `QColor` каждый компонент цвета возможно с помощью методов `red()`, `green()`, `blue()` и `alpha()`. Эти же значения можно получить и в вещественном представлении, для чего нужно вызвать: `redF()`, `greenF()`, `blueF()` и `alphaF()`. Можно вообще обой-

тись одним методом `getRgb()`, в который передаются указатели на переменные для значений цветов, например:

```
QColor color(100, 200, 0);
int r, g, b;
color.getRgb(&r, &g, &b);
```

Для записи значений RGB можно, по аналогии, воспользоваться методами, похожими на описанные, но имеющими префикс `set` (вместо префикса `get`, если он есть), после которого идет заглавная буква. Также для этой цели можно прибегнуть к структуре данных `QRgb`, которая состоит из четырех байтов и полностью совместима с 32-битным значением. Эту структуру можно создать с помощью функции `qRgb()` или `qRgba()`, передав в нее параметры красного, зеленого и синего цветов. Но можно присваивать переменным структуры `QRgb` и 32-битное значение цвета. Например, синий цвет устанавливается сразу несколькими способами:

```
QRgb rgbBlue1 = qRgba(0, 0, 255, 255); // С информацией о прозрачности
QRgb rgbBlue2 = qRgb(0, 0, 255);
QRgb rgbBlue3 = 0x000000FF;
```

При помощи функций `qRed()`, `qGreen()`, `qBlue()` и `qAlpha()` можно получить значения цветов и информацию о прозрачности соответственно.

Значения типа `QRgb` можно передавать в конструктор класса `QColor` или в метод `setRgb()`:

```
QRgb rgbBlue = 0x000000FF;
QColor colorBlue1(rgbBlue);
QColor colorBlue2;
colorBlue2.setRgb(rgbBlue);
```

Также можно получать значения структуры `QRgb` от объектов класса `QColor` вызовом метода `rgb()`.

Цвет можно установить, передав значения в символьном формате, например:

```
QColor colorBlue1("#0000FF");
QColor colorBlue2;
colorBlue2.setNameColor("#0000FF");
```

Цветовая модель HSV

Модель HSV (Hue, Saturation, Value — оттенок, насыщенность, значение) не смешивает основные цвета при моделировании нового цвета, как в случае с RGB, а просто изменяет их свойства. Это очень напоминает принцип, используемый художниками для получения новых цветов, — подмешивая к чистым цветам белую, черную или серую краски.

Пространство цветов этой модели задается пирамидой с шестиконечным основанием, так называемым *Hexcone* (рис. 17.7). Координаты в этой модели имеют следующий смысл:

- оттенок (Hue) — это «цвет» в общеупотребительном смысле этого слова, например: красный, оранжевый, синий и т. д., который задается углом в цветовом круге, изменяющемся от 0 до 360 градусов;
- насыщенность (Saturation) обозначает наличие белого цвета в оттенке. Значение насыщенности может изменяться в диапазоне от 0 до 255. Значение, равное 255 в целочисленном числовом представлении либо единице в вещественном представлении, соответ-

ствует полностью насыщенному цвету, который не содержит оттенков белого. Частично насыщенный оттенок светлее — например, оттенок красного с насыщенностью, равной 128, либо 0,5 в вещественном представлении, соответствует розовому;

- значение (Value) или яркость — определяет интенсивность цвета. Цвет с высокой интенсивностью — яркий, а с низкой — темный. Значение этого параметра может изменяться в диапазоне от 0 до 255 в целочисленном числовом представлении (либо от 0 до 1 в вещественном представлении).

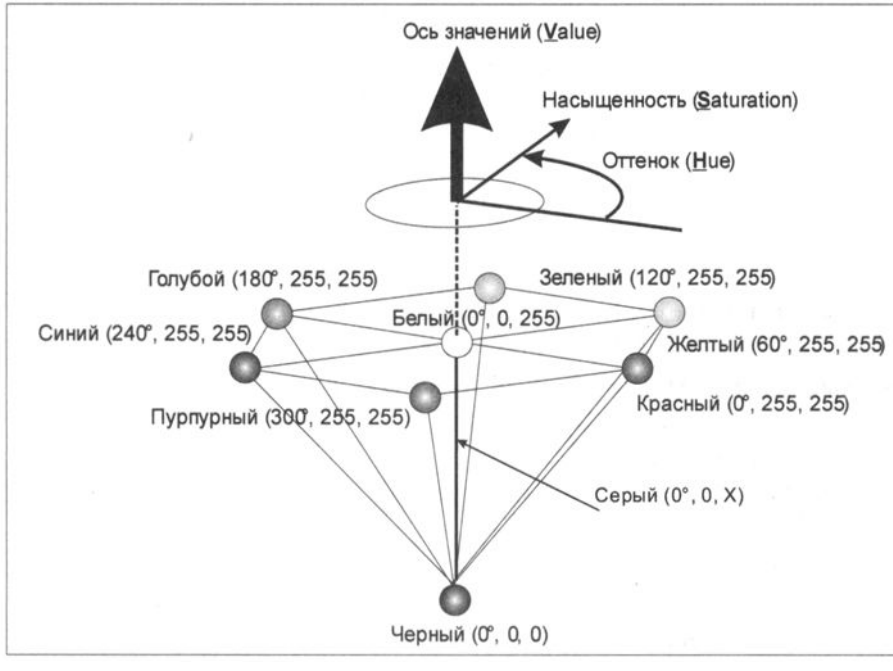


Рис. 17.7. Цветовая модель HSV

Установку значения цвета в координатах HSV можно выполнить с помощью метода `QColor::setHsv()` или `QColor::setHsvF()`:

```
color.setHsv(233, 100, 50);
```

Для того чтобы получить цветовое значение в цветовой модели HSV, нужно передать в метод `getHsv()` адреса трех целочисленных значений (или вещественных, если это `getHsvF()`). Следующий пример устанавливает RGB-значение и получает в трех переменных его HSV-эквивалент:

```
QColor color(100, 200, 0);
int h, s, v;
color.getHsv(&h, &s, &v);
```

Цветовая модель CMYK

Применение модели CMYK (Cyan, Magenta, Yellow, Key color — голубой, пурпурный, желтый, «ключевой» черный цвет) чаще всего связано с печатью. Пространство цветов этой модели так же, как и в случае с RGB, представляет собой куб с длиной сторон равной 255

(рис. 17.8) в целочисленном числовом представлении или единице в вещественном представлении. В отличие от RGB, эта модель является *«субтрактивной»*, то есть вычитаемой. В субтрактивной цветовой модели любой цвет представляется в виде трех величин, каждая из которых указывает, какое количество определенного цвета подлежит исключению из белого.

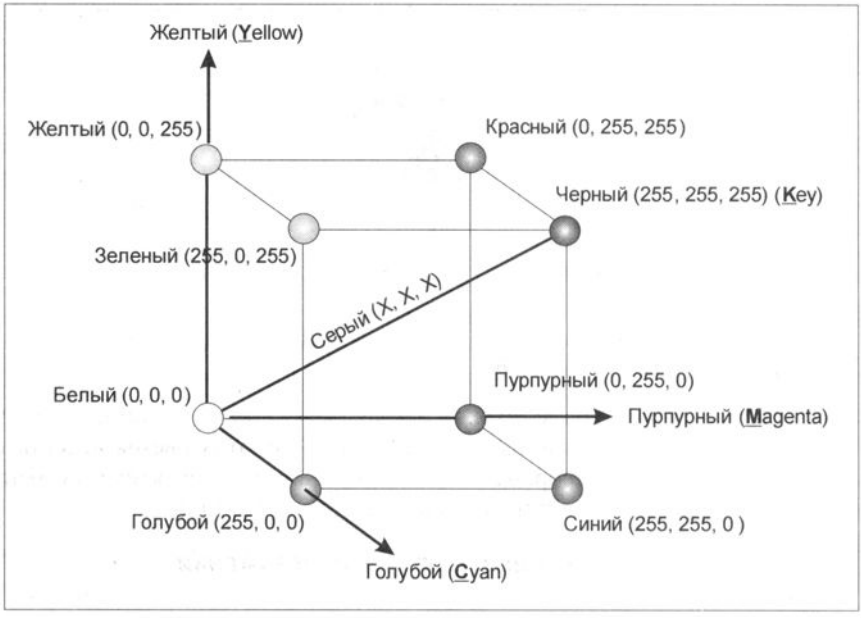


Рис. 17.8. Цветовая модель CMYK

Таким образом, для получения черного цвета на печатающем устройстве необходимо задействовать все три основные составляющие: желтую, голубую и пурпурную. Такой подход не отличается экономичностью и на практике не всегда дает чисто черный цвет, в связи с чем в печатающих устройствах дополнительно используется черная краска. Вот именно поэтому черный цвет и представляет собой четвертую составляющую (Key color) этой модели.

> ***KEY COLOR, А НЕ BLACK!***
>
> Конечно, логичнее было бы назвать последнюю компоненту не Key color, а Black. Но Black начинается с буквы «B», и в этом случае могла бы возникнуть путаница с синим цветом (Blue), чья первая буква уже задействована в модели RGB.

Установка цвета для цветовой модели CMYK осуществляется таким же образом, как и в случаях с RGB и HSV. Класс `QColor` предоставляет методы `getCmyk()` и `getCmykF()` для получения значений, а методы `setCmyk()` и `setCmykF()` предназначены для их установки.

Палитра

Палитра предоставляет ограниченное (в большинстве случаев числом 256) количество цветовых значений. Цветовые значения адресуются при помощи индексов. Сами индексируемые цветовые значения можно задавать свободно. На рис. 17.9 отображается пиксел, имеющий значение цвета RGB(200, 75, 13), адресуемое индексом 3.

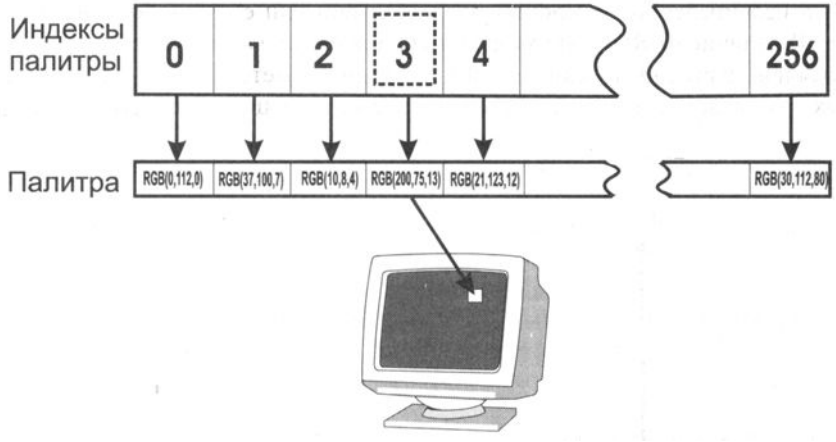


Рис. 17.9. Отображение пиксела, имеющего указанный в палитре цвет

Предопределенные цвета

В табл. 17.1 приведены константы именованных цветов, предопределенных в Qt. Они представляют собой палитру, состоящую из 17 цветов. Конечно, этих цветов недостаточно для получения фотореалистичных изображений, но они удобны на практике, особенно в тех ситуациях, когда требуется отображать основные цветовые значения.

Константы для рисования двухцветных изображений

В общей сложности именованных цветов 19. Две константы, не приведенные в таблице: `Qt::color0` и `Qt::color1` — используются для рисования двухцветных изображений.

Таблица 17.1. *Цвета перечисления* `GlobalColor` *пространства имен Qt*

Константа	RGB-значение	Описание
`black`	`(0, 0, 0)`	Черный
`white`	`(255, 255, 255)`	Белый
`darkGray`	`(128, 128, 128)`	Темно-серый
`gray`	`(160, 160, 164)`	Серый
`lightGray`	`(192, 192, 192)`	Светло-серый
`red`	`(255, 0, 0)`	Красный
`green`	`(0, 255, 0)`	Зеленый
`blue`	`(0, 0, 255)`	Синий
`cyan`	`(0, 255, 255)`	Голубой
`magenta`	`(255, 0, 255)`	Пурпурный
`yellow`	`(255, 255, 0)`	Желтый
`darkRed`	`(128, 0, 0)`	Темно-красный
`darkGreen`	`(0, 128, 0)`	Темно-зеленый
`darkBlue`	`(0, 0, 128)`	Темно-синий
`darkCyan`	`(0, 128, 128)`	Темно-голубой

***Таблица 17.1** (окончание)*

Константа	RGB-значение	Описание
darkMagenta	(128, 0, 128)	Темно-пурпурный
darkYellow	(128, 128, 0)	Темно-желтый

Класс `QColor` предоставляет методы `lighter()` и `darker()`, с помощью которых можно получать значения цвета, делая основное значение светлее или темнее. Методы `lighter()` и `darker()` не изменяют исходный объект цвета, а создают новый. Для этого текущий цвет в модели RGB преобразуется в цвет модели HSV и ее компонент «Значение» (Value) умножается (для `darker()` — делится) на множитель (выраженный в процентах), переданный в этот метод, а затем полученное значение преобразуется обратно в модель RGB. Сделать красный цвет немного темнее можно следующим образом:

```
QColor color = QColor(Qt::red).darker(160);
```

Резюме

Графика играет очень важную роль, ее использование можно встретить практически во всех серьезных программных продуктах.

Qt предоставляет ряд классов геометрии, необходимых при создании программ с графикой. Объекты классов `QPoint`/`QPointF` хранят в себе координаты *X* и *Y*, описывающие расположение точки на плоскости. Классы размера `QSize`/`QSizeF` предназначены для хранения значений ширины и высоты. Классы `QRect`/`QRectF` объединяют в себе величины, хранящиеся в объектах классов `QPoint`/`QPointF` и `QSize`/`QSizeF`. Классы `QLine`/`QLineF` и `QPolygon`/`QPolygonF` предоставляют возможность описания линий и многоугольников.

Qt поддерживает три цветовые модели: RGB, CMYK и HSV. RGB — это очень распространенная цветовая модель, в которой любой цвет получается в результате смешения трех цветов: красного, зеленого и синего. Цветовая модель CMYK получила большое распространение в полиграфии. В модели HSV цвет задается тремя параметрами: оттенком (Hue), насыщенностью (Saturation) и значением (Value), или, иначе, яркостью.

Представление цвета TrueColor дает возможность получить любой нужный цвет. В него входят все представления, имеющие более 8 битов.

Палитра — это массив, в котором каждому возможному значению пиксела в соответствие ставится значение цвета.

Класс `QColor` предназначен для хранения цветовых значений и предоставляет множество полезных методов, с помощью которых можно конвертировать цветовые значения из RGB, CMYK в HSV (и наоборот), сравнивать их, делать светлее или темнее.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/17-510/** или с помощью следующего QR-кода (рис. 17.10):



Рис. 17.10. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 18

Легенда о короле Артуре и контекст рисования

Сэр Артур осмотрел свой меч и остался им доволен.
— Что тебе больше нравится, — сказал Мерлин, — меч или ножны?
— Мне больше нравится меч, — сказал Артур.
— Не мудр твой ответ, — сказал Мерлин, — ибо эти ножны в десять раз драгоценней меча.

Марк Твен,
«Янки из Коннектикута при дворе короля Артура»

Под именем «Артур» (Arthur) подразумевается новая архитектура для рисования, создание которой началось с желания использовать методы рисования `QPainter` в OpenGL. Затем появилось множество других идей: от точного представления в вещественных координатах до отображения векторной графики, которые получили свое воплощение в этой архитектуре.

Три краеугольных камня этой технологии составляют классы `QPainter`, `QPaintEngine` и `QPaintDevice` (рис. 18.1).

Класс `QPaintEngine` используется классами `QPainter` и `QPaintDevice` неявно и для разработчиков не интересен, если нет необходимости создавать свой собственный контекст рисования. Если же такая необходимость возникла, то вам придется унаследовать этот класс и реализовать некоторые его методы.

Основной класс для программирования графики, с которым нам придется иметь дело, — это `QPainter`. С его помощью можно рисовать точки, линии, эллипсы, многоугольники (полигоны), кривые Безье, растровые изображения, текст и многое другое. Более того, класс `QPainter` поддерживает режим *сглаживания* (antialiasing), прозрачность и градиенты, о которых будет рассказано в этой главе.

Контекст рисования `QPaintDevice` можно представить себе как поверхность для вывода графики. `QPaintDevice` — это основной абстрактный класс для всех классов объектов, которые можно рисовать. От него унаследована целая серия классов, показанных на рис. 18.2.

В основном, рисование выполняется из метода обработки события `QPaintEvent` (см. *главу 14*). Если пользователь запустит программу и выполнит некоторые действия, вследствие которых перекроется, частично или полностью, окно программы, то после его открытия будет сгенерировано событие перерисовки и вызван метод `QWidget::paintEvent()` для тех виджетов, которые должны быть перерисованы. Сам объект события `QPaintEvent` содержит метод `region()`, возвращающий область для перерисовки. Метод `QPaintEvent::rect()` возвращает прямоугольник, который охватывает эту область.

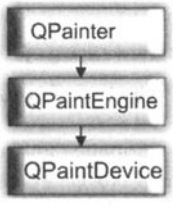


Рис. 18.1. Взаимосвязь классов архитектуры «Артур»

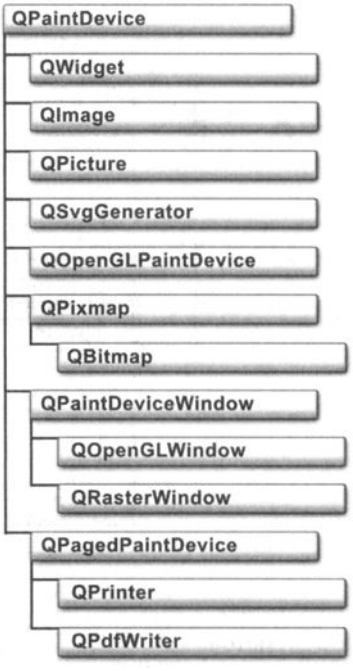


Рис. 18.2. Иерархия классов контекста рисования

Для подавления эффекта мерцания Qt использует технику *двойной буферизации* (double buffering). Двойная буферизация представляет собой очень распространенное и простое решение этой проблемы. Суть заключается в формировании изображения в невидимой области (буфере). Сформированное изображение помещается в видимую область (буфер) за один раз. Это выполняется автоматически, и вам не нужно реализовывать для этого код в `paintEvent()`.

Эффекта прозрачности можно добиться, установив для рисования цвет, содержащий значение прозрачности альфа-канала (см. *главу 17*). Это значение может изменяться от 0 до 255. Значение 0 говорит том, что цвет полностью прозрачен, а 255 означает полную непрозрачность.

Класс *QPainter*

Класс `QPainter`, определенный в заголовочном файле QPainter, является исполнителем команд рисования. Он содержит массу методов для отображения линий, прямоугольников, окружностей и др. Рисование осуществляется на всех объектах классов, унаследованных от класса `QPaintDevice` (рис. 18.3). Это означает, что то, что отображается контекстом рисования одного объекта, может быть точно так же отображено контекстом и другого объекта.

Чтобы использовать объект `QPainter`, необходимо передать ему адрес объекта контекста, на котором должно осуществляться рисование. Этот адрес можно передать как в конструкторе, так и с помощью метода `QPainter::begin()`. Смысл метода `begin()` состоит в том, что он позволяет рисовать на одном контексте несколькими объектами класса `QPainter`. При использовании метода `begin()` нужно по окончании работы с контекстом вызвать метод `QPainter::end()`, чтобы отсоединить установленную этим методом связь с контекстом рисования, давая возможность для рисования другому объекту (листинг 18.1).

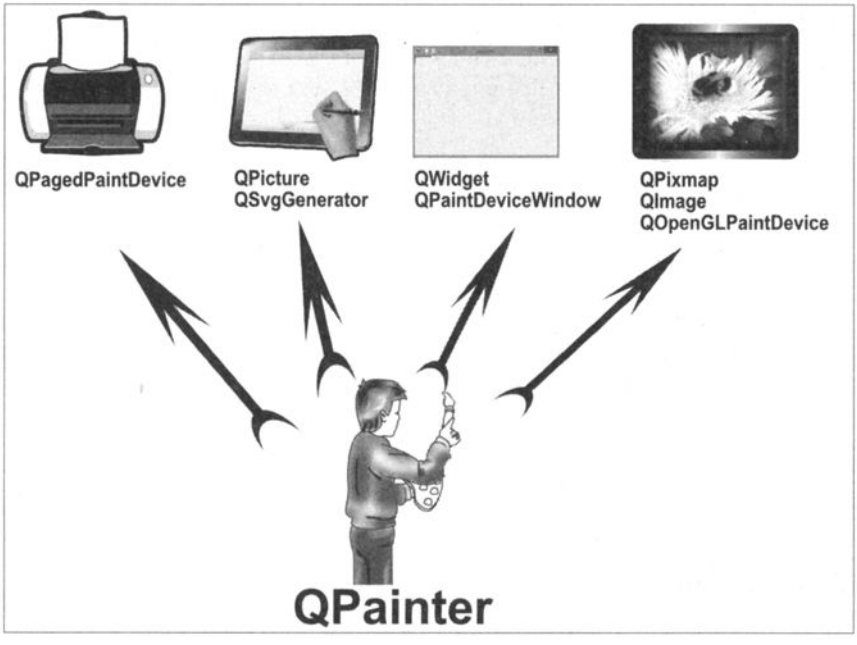


Рис. 18.3. `QPainter` и контексты рисования

Листинг 18.1. Рисование двумя объектами `QPainter` в одном контексте

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter1;
    QPainter painter2;

    painter1.begin(this);
    // Команды рисования
    painter1.end();

    painter2.begin(this);
    // Команды рисования
    painter2.end();
}
...
```

Но чаще всего используется рисование одним объектом `QPainter` в разных контекстах (листинг 18.2).

Листинг 18.2. Рисование одним объектом `QPainter` в двух разных контекстах

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter;
```

```
    painter.begin(this); //контекст виджета
    // Команды рисования
    painter.end();

    QPixmap pix(rect());
    painter.begin(&pix); //контекст растрового изображения
    // Команды рисования
    painter.end();
}
...
```

Объекты `QPainter` содержат установки, влияющие на их рисование. Это могут быть трансформация координат, модификация кисти и пера, установка шрифта, установка режима сглаживания и т. д. Поэтому, для того чтобы оставить старые настройки без изменений, рекомендуется перед началом изменения сохранить их с помощью метода `QPainter::save()`, а по окончании работы — восстановить с помощью метода `QPainter::restore()`.

Перья и кисти

Перья и кисти — это основа для программирования графики с использованием библиотеки Qt. Без них не получится вывести на экран даже точку.

Перо

Перо служит для рисования контурных линий фигуры. Атрибуты пера: цвет, толщина и стиль. Установить новое перо можно с помощью метода `QPainter::setPen()`, передав в него объект класса `QPen`. Можно передавать и предопределенные стили пера, указанные в табл. 18.1.

Таблица 18.1. *Некоторые значения из перечисления* `PenStyle` *пространства имен Qt*

Константа	Значение	Вид (толщина = 4)
`NoPen`	0	
`SolidLine`	1	
`DashLine`	2	
`DotLine`	3	
`DashDotLine`	4	
`DashDotDotLine`	5	

Толщина линии является значением целого типа, которое передается в метод `QPen::setWidth()`. Если значение равно нулю, то это не означает, что линия будет невидима, а говорит лишь о том, что она должна быть изображена как можно тоньше.

Если необходимо, чтобы линия не отображалась вообще, то тогда устанавливается стиль `NoPen`. Зачем же нужно перо, которое не рисует? Бывают и такие случаи, когда и пустое перо пригодится, — например, когда нужно вывести четырехугольник определенного цвета без контурной линии.

Цвет пера задается с помощью метода `QPen::setColor()`, в который передается объект класса `QColor`. Следующий пример создает перо красного цвета, толщиной в три пиксела и со стилем «штрих». Объект пера устанавливается в объекте `QPainter` вызовом метода `setPen()`:

```
QPainter painter(this);
painter.setPen(QPen(Qt::red, 3, Qt::DashLine));
```

Стили для концов линий пера устанавливаются методом `setCapStyle()`, в который передается один из флагов: `Qt::FlatCap` (край линии квадратный и проходит через граничную точку), `Qt::SquareCap` (край квадратный и перекрывает граничную точку на половину ширины линии) или `Qt::RoundCap` (край закругленный и также покрывает граничную точку линии).

Можно устанавливать стили и для переходов одной линии в другую — методом `setJoinStyle()`, передав в него флаги: `Qt::MiterJoin` (линии продлеваются и соединяются под острым углом), `Qt::BevelJoin` (пространство между линиями заполняется) или `Qt::RoundJoin` (угол закругляется). Но эти переходы будут видны только на толстых линиях.

Кисть

Кисть служит для заполнения непрерывных контуров, таких как прямоугольники, эллипсы и многоугольники. Класс кисти `QBrush` определен в заголовочном файле QBrush. Кисть задается двумя параметрами: цветом и образцом заливки.

Установить кисть можно методом `QPainter::setBrush()`, передав в него объект класса `QBrush` или один из предопределенных шаблонов, указанных в табл. 18.2. Если заполнение не нужно, то в метод `QPainter::setBrush()` следует передать значение `NoBrush`.

Таблица 18.2. *Перечисление* `BrushStyle` *пространства имен Qt (выборочно)*

Константа	Значение	Вид	Константа	Значение	Вид
NoBrush	0		VerPattern	10	
SolidPattern	1		CrossPattern	11	
Dense1Pattern	2		BDiagPattern	12	
Dense2Pattern	3		FDiagPattern	13	
Dense3Pattern	4		DiagCrossPattern	14	
Dense4Pattern	5		LinearGradientPattern	15	
Dense5Pattern	6		RadialGradientPattern	16	
Dense6Pattern	7		ConicalGradientPattern	17	
Dense7Pattern	8		TexturePattern	24	
HorPattern	9				

Следующие строки устанавливают красную кисть с горизонтальной штриховкой:

```
QPainter painter(this);
painter.setBrush(QBrush(Qt::red, Qt::HorPattern));
```

Если в табл. 18.2 не нашлось подходящей кисти, то можно создать свою собственную с помощью стиля `TexturePattern`. Чтобы применить этот стиль, нужно передать в метод `setTexture()` растровое изображение. Растровое изображение можно также использовать и при создании кисти (рис. 18.4):

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    QPixmap pix(":/Alina.jpg");
    painter.setBrush(QBrush(Qt::black, pix));
    painter.drawEllipse(0, 0, 300, 150);
}
...
```



Рис. 18.4. Заполнение эллипса растровым изображением

Градиенты

Градиент — это плавный переход от одного цвета к другому. В настоящее время применение градиентов стало очень популярно, ведь с их помощью можно придать элементам изображений в ваших приложениях эффект объемности. В основе градиентов лежит гладкая интерполяция между двумя и более цветовыми значениями. Qt предоставляет три основных типа градиентов: линейный (linear), конический (conical) и радиальный (radial).

Линейные (linear) *градиенты* реализует класс `QLinearGradient`. Они задаются двумя цветовыми точками контроля и несколькими точками останова (color stops) на линии, соединяющей цвета этих точек. Листинг 18.3 иллюстрирует эту возможность.

Листинг 18.3. Линейный градиент

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter        painter(this);
    QLinearGradient gradient(0, 0, width(), height());
```

```
    gradient.setColorAt(0, Qt::red);
    gradient.setColorAt(0.5, Qt::green);
    gradient.setColorAt(1, Qt::blue);
    painter.setBrush(gradient);
    painter.drawRect(rect());
}
...
```

В листинге 18.3 мы задали три цвета на трех различных позициях между двумя точками контроля. Позиции точек задаются вещественными значениями от 0 до 1, где 0 представляет собой первую контрольную точку, а 1 — вторую. Цвета между этими точками будут интерполированы (рис. 18.5).

Конический (conical) *градиент* реализуется классом `QConicalGradient` и задается центральной точкой и углом. Распространение цветов вокруг центральной точки соответствует повороту часовой стрелки. В листинге 18.4 приведена реализация конического градиента (рис. 18.6).

Листинг 18.4. Конический градиент

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter         painter(this);
    QConicalGradient gradient(width() / 2, height() / 2, 0);
    gradient.setColorAt(0, Qt::red);
    gradient.setColorAt(0.4, Qt::green);
    gradient.setColorAt(0.8, Qt::blue);
    gradient.setColorAt(1, Qt::red);
    painter.setBrush(gradient);
    painter.drawRect(rect());
}
...
```

Радиальный (radial) *градиент* реализует класс `QRadialGradient` и задается центральной точкой, радиусом и точкой фокуса. Центральная точка и радиус задают окружность. На рас-

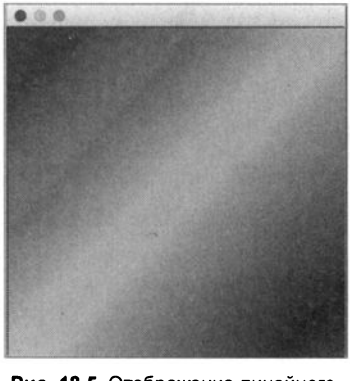


Рис. 18.5. Отображение линейного градиента

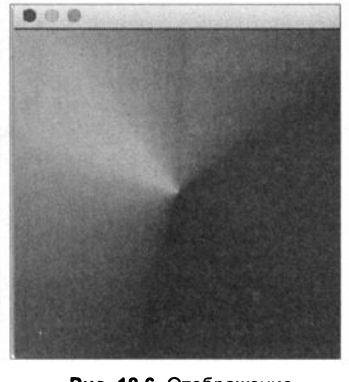


Рис. 18.6. Отображение конического градиента

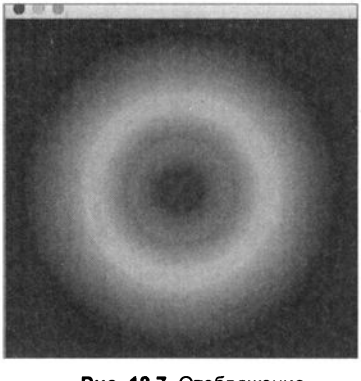


Рис. 18.7. Отображение радиального градиента

пространение цветов за пределами точки фокуса влияет центральная точка или точка, находящаяся внутри окружности. Пример реализации такого градиента приведен в листинге 18.5, а результат показан на рис. 18.7.

Листинг 18.5. Радиальный (лучевой) градиент

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter        painter(this);
    QPointF         ptCenter(rect().center());
    QRadialGradient gradient(ptCenter, width() / 2, ptCenter);
    gradient.setColorAt(0, Qt::red);
    gradient.setColorAt(0.5, Qt::green);
    gradient.setColorAt(1, Qt::blue);
    painter.setBrush(gradient);
    painter.drawRect(rect());
}
...
```

Техника сглаживания (Anti-aliasing)

Одним из побочных эффектов, возникающих при рисовании геометрических фигур, является ступенчатость, хорошо заметная на контурах (рис. 18.8). Это и понятно, поскольку такие ступени есть не что иное, как пикселы, через которые проходит контур. Для подавления этого нежелательного эффекта используется техника *сглаживания* (Anti-aliasing). С ее помощью границы кривых можно сделать более гладкими, убирая ступени, образующиеся на краях объектов, что достигается добавлением промежуточных цветов. Это снижает скорость рисования, но улучшает визуальный эффект.

Режим сглаживания распространяется на отображение текста и геометрических фигур. Его можно включить в объекте класса `QPainter` при помощи метода `setRenderHint()`:

```
painter.setRenderHint(QPainter::Antialiasing, true);
```

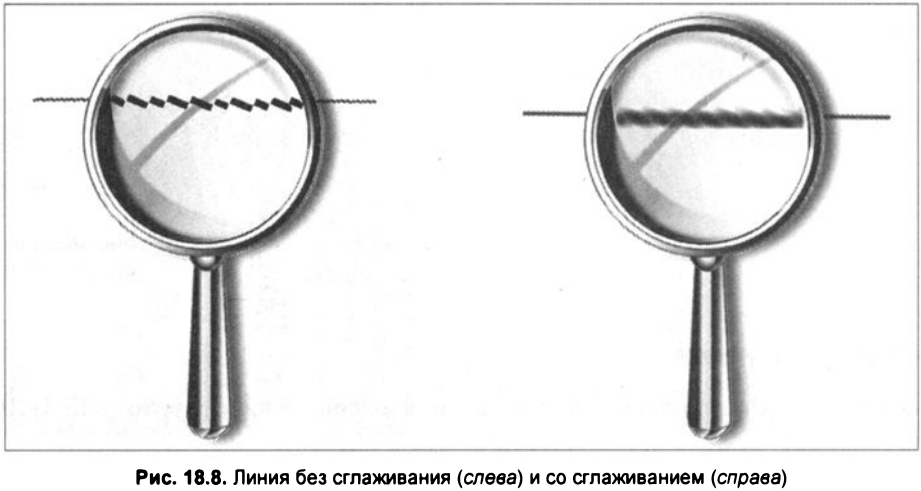


Рис. 18.8. Линия без сглаживания (*слева*) и со сглаживанием (*справа*)

Рисование

Отображение фигур — задача несложная, ведь для этого не требуется вычислять расположение каждого выводимого пиксела, так как уже имеется целый ряд методов для вывода практически всех геометрических фигур. Например, вызовом метода `drawRect()` можно нарисовать прямоугольник.

Перед тем как приступить к рисованию, важно понять философию координат пиксела. Она заключается в том, что центр пиксела лежит в его середине. То есть пиксел, находящийся в самом верхнем левом углу, будет иметь координаты (0,5; 0,5). Если мы попытаемся нарисовать пиксел с координатами (0; 0), то `QPainter` автоматически прибавит к ним 0,5, и результатом все равно станет (0,5; 0,5). Этот сдвиг выполняется при выключенном режиме сглаживания. Если этот режим включен, и мы попытаемся нарисовать черный пиксел с координатами (50; 50), то в результате сглаживания будут нарисованы сразу четыре серых пиксела с координатами (49,5; 49,5), (49,5; 50,5), (50,5; 49,5) и (50,5; 50,5). Но если мы попытаемся нарисовать черный пиксел с координатами (49,5; 49,5), то в результате увидим только один пиксел.

Рисование точек

Для отображения точек применяется только перо. В листинге 18.6 на экране рисуются восемь точек (рис. 18.9).

Листинг 18.6. Рисование точек методом `drawPoint()`

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    painter.setPen(QPen(Qt::black, 3));

    int n = 8;
    for (int i = 0; i < n; ++i) {
        qreal fAngle = ::qDegreesToRadians(360.0 * i / n);
        qreal x      = 50 + cos(fAngle) * 40;
        qreal y      = 50 + sin(fAngle) * 40;
        painter.drawPoint(QPointF(x, y));
    }
}
...
```

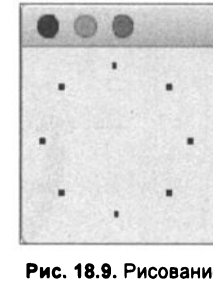


Рис. 18.9. Рисование точек

Рисование линий

Для рисования отрезка прямой линии, идущего из одной точки в другую (рис. 18.10), используется метод `drawLine()`, в который передаются координаты начальной (x1, y1) и конечной (x2, y2) точек `QPointF` (листинг 18.7).

Листинг 18.7. Рисование линий методом drawLine()

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.drawLine(QPointF(10, 10), QPointF(90, 90));
}
...
```

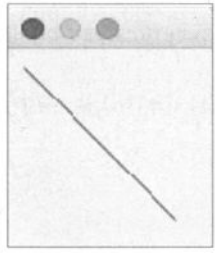


Рис. 18.10. Линия

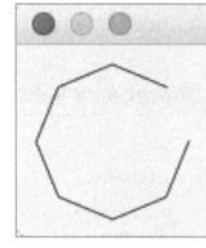


Рис. 18.11. Соединение точек линиями

Метод `drawPolyLine()` проводит линию, которая соединяет точки, передаваемые в первом параметре. Второй параметр задает количество точек, которые должны быть соединены (т. е. число элементов массива). Первая и последняя точки не соединяются. В листинге 18.8 рисуется фигура, показанная на рис. 18.11.

Листинг 18.8. Рисование фигуры методом drawPolyline()

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);

    const int n = 8;
    QPointF  a[n];
    for (int i = 0; i < n; ++i) {
        qreal fAngle = ::qDegreesToRadians(360.0 * i / n);
        qreal x      = 50 + cos(fAngle) * 40;
        qreal y      = 50 + sin(fAngle) * 40;
        a[i] = QPointF(x, y);
    }
    painter.drawPolyline(a, n);
}
...
```

Рисование сплошных прямоугольников

Прямоугольник — это очень распространенная геометрическая фигура, посмотрите вокруг — прямоугольные предметы окружают нас везде. Qt содержит два метода для рисования прямоугольников без контурных линий: `fillRect()` и `eraseRect()`. Их внешний вид задается только кистью. В метод `fillRect()` передаются пять параметров. Первые четыре параметра задают координаты (x, y) и размеры (ширина, высота) прямоугольника. Пятый параметр задает кисть.

В метод `eraseRect()` передаются только четыре параметра, задающие позицию и размеры прямоугольной области. Для ее заполнения используется фон, установленный в виджете. Таким образом, вызов этого метода эквивалентен вызову `fillRect()` с пятым параметром — значением, возвращаемым методом `paletteBackgroundColor()`. В листинге 18.9 приведен пример использования методов `fillRect()` и `eraseRect()`, а результат показан на рис. 18.12.

Листинг 18.9. Рисование сплошных прямоугольников методами `fillRect()` и `eraseRect()`

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    QBrush brush(Qt::red, Qt::Dense4Pattern);
    painter.fillRect(10, 10, 100, 100, brush);
    painter.eraseRect(20, 20, 80, 80);
}
...
```

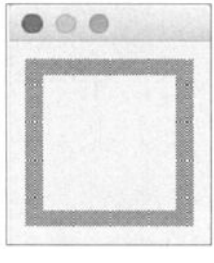


Рис. 18.12. Сплошные прямоугольники

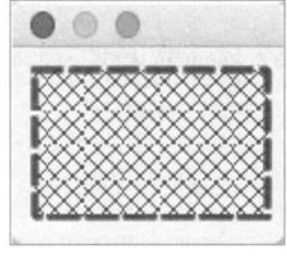


Рис. 18.13. Заполненный прямоугольник

Рисование заполненных фигур

Для рисования фигур также применяются методы, использующие перо `QPen` и кисть `QBrush`. Если требуется нарисовать только контур фигуры, без заполнения, то для этого методом `QPainter::setBrush()` нужно установить значение стиля кисти `QBrush::NoBrush`. А для рисования фигуры без контурной линии можно методом `QPainter::setPen()` установить стиль пера `QPen::NoPen`.

Метод `drawRect()` рисует прямоугольник (рис. 18.13). В него передаются следующие параметры: координаты (x, y) верхнего левого угла, ширина и высота. В этот метод можно передать также объект класса `QRect` (листинг 18.10).

Листинг 18.10. Рисование заполненного прямоугольника методом drawRect()

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setBrush(QBrush(Qt::red, Qt::DiagCrossPattern));
    painter.setPen(QPen(Qt::blue, 3, Qt::DashLine));
    painter.drawRect(QRect(10, 10, 110, 70));
}
...
```

Метод `drawRoundRect()` рисует прямоугольник с закругленными углами (рис. 18.14). Закругленность углов достигается с помощью четвертинок эллипса. Последние два параметра метода задают, насколько сильно должны быть закруглены углы в направлениях осей координат X и Y соответственно. При передаче нулевых значений углы не будут закруглены, а при задании им значения 100 прямоугольник превратится в эллипс. Прямоугольную область можно задавать объектом класса `QRect` (листинг 18.11).

Листинг 18.11. Рисование прямоугольника с закругленными углами методом drawRoundRect()

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setBrush(QBrush(Qt::green));
    painter.setPen(QPen(Qt::black));
    painter.drawRoundRect(QRect(10, 10, 110, 70), 30, 30);
}
...
```

Метод `drawEllipse()` рисует заполненный эллипс (рис. 18.15), размеры и расположение которого задаются прямоугольной областью (листинг 18.12).



Рис. 18.14. Прямоугольник с закругленными углами

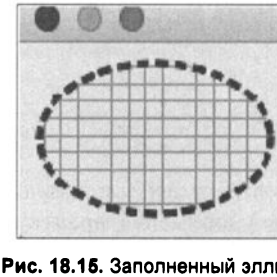


Рис. 18.15. Заполненный эллипс

Листинг 18.12. Рисование заполненного эллипса методом drawEllipse()

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setBrush(QBrush(Qt::green, Qt::CrossPattern));
    painter.setPen(QPen(Qt::red, 3, Qt::DotLine));
    painter.drawEllipse(QRect(10, 10, 110, 70));
}
...
```

Метод `drawChord()` рисует хорду, отсекающую часть эллипса (рис. 18.16). Размеры и расположение эллипса задаются прямоугольной областью, а отображаемая часть — двумя последними параметрами, представляющими собой значения углов. Углы задаются в единицах, равных одной шестнадцатой градуса. Начальная и конечная точки соединяются прямой линией. При положительных значениях двух последних параметров (углов) начальная точка перемещается вдоль кривой эллипса против часовой стрелки. Предпоследний параметр задает начальный угол. Последний параметр задает угол, под которым кривые должны пересекаться (листинг 18.13).

Листинг 18.13. Рисование хорды, отсекающей часть эллипса, методом drawChord()

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setBrush(QBrush(Qt::yellow));
    painter.setPen(QPen(Qt::blue));
    painter.drawChord(QRect(10, 10, 110, 70), 45 * 16, 180 * 16);
}
...
```

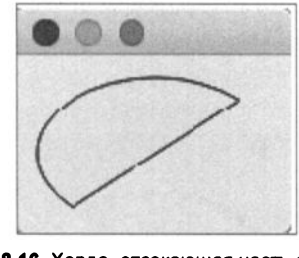


Рис. 18.16. Хорда, отсекающая часть эллипса

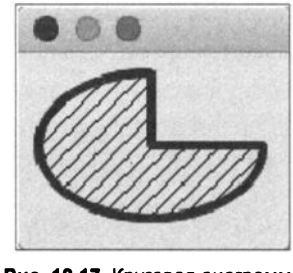


Рис. 18.17. Круговая диаграмма

В мире деловой графики пользуются спросом круговые диаграммы (рис. 18.17). Такие рисунки очень удобны для представления статистических данных. Метод `drawPie()` рисует часть эллипса. Начальная и конечная точки соединяются с центром эллипса. Последние два параметра метода задаются в шестнадцатых долях градуса (листинг 18.14).

Листинг 18.14. Рисование круговой диаграммы методом `drawPie()`

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setBrush(QBrush(Qt::black, Qt::BDiagPattern));
    painter.setPen(QPen(Qt::blue, 4));
    painter.drawPie(QRect(10, 10, 110, 70), 90 * 16, 270 * 16);
}
...
```

Метод `drawPolygon()` рисует заполненный многоугольник (рис. 18.18), последняя из заданных вершин которого соединена с первой (листинг 18.15).

Листинг 18.15. Рисование заполненного многоугольника методом `drawPolygon()`

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setBrush(QBrush(Qt::lightGray));
    painter.setPen(QPen(Qt::black));

    int       n = 8;
    QPolygonF polygon;
    for (int i = 0; i < n; ++i) {
        qreal fAngle = ::qDegreesToRadians(360.0 * i / n);
        qreal x      = 50 + cos(fAngle) * 40;
        qreal y      = 50 + sin(fAngle) * 40;
        polygon << QPointF(x, y);
    }
    painter.drawPolygon(polygon);
}
...
```

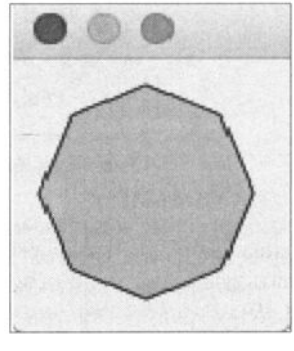


Рис. 18.18. Заполненный многоугольник

Запись команд рисования

Класс `QPicture` — это контекст рисования, который предоставляет возможность протоколирования команд класса `QPainter`. С его помощью команды можно даже записывать в отдельные файлы (называемые *метафайлами*), а потом загружать их снова, чтобы повторить ранее проделанные действия. Эти действия можно перенаправлять и на другие контексты рисования, например на принтер или экран. В листинге 18.16 выполняется запись одной команды рисования в файл myline.dat.

Листинг 18.16. Протоколирование команд рисования

```
QPicture pic;
QPainter painter;

painter.begin(&pic);
painter.drawLine(20, 20, 50, 50);
painter.end();

if (!pic.save("myline.dat")) {
    qDebug() << "can not save the file";
}
```

Листинг 18.17 демонстрирует загрузку команд из файла и их исполнение в другом контексте. Для отображения в другом контексте используется метод `drawPicture()`. Первый параметр этого метода устанавливает позицию, с которой начнется рисование, а во втором параметре передается объект класса `QPicture`.

Листинг 18.17. Загрузка и выполнение команды в другом контексте рисования

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPicture pic;
    if (!pic.load("myline.dat")) {
        qDebug() << "can not load the file";
    }

    QPainter painter;
    painter.begin(this);
    painter.drawPicture(QPoint(0, 0), pic);
    painter.end();
}
...
```

Трансформация систем координат

Класс `QPainter` предоставляет очень мощный механизм трансформации координат для отображения объектов. Это позволяет показывать изображения в повернутом, масштабированном, смещенном и скошенном виде (рис. 18.19).

Каждая точка в двумерной системе координат описывается, соответственно, двумя координатами. Трансформация одинаково действует на все точки графического объекта. Для трансформации в классе `QPainter` определены следующие методы: `translate()`, `scale()`, `rotate()` и `shear()`. Трансформации можно комбинировать, но порядок их следования отражается на конечном результате. Например, если сначала провести операцию скоса, а затем операции поворота и снова скоса, то результат будет отличаться от итога, полученного после двух операций скоса и поворота.

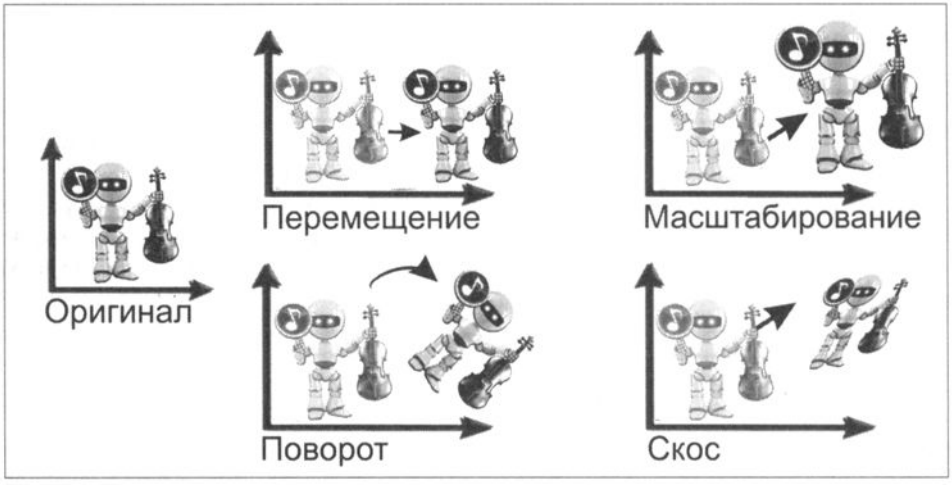


Рис. 18.19. Различные виды трансформации объектов изображений

Для сохранения и восстановления состояний объектов класса `QPainter` Qt предоставляет методы `save()` и `restore()`. Это очень удобно при получении извне указателя на объект класса `QPainter`. Можно сохранить его состояние с помощью метода `save()`, а затем проводить с ним разного рода трансформации. По окончании трансформации вызов метода `restore()` вернет объект класса `QPainter` в исходное состояние. Например:

```
pPainter->save();
pPainter->translate(20, 40);
pPainter->restore();
```

Перемещение

Часто требуется переместить изображение на экране. Класс `QPainter` предоставляет для этого метод `translate()`, в который передаются два целочисленных параметра. В первом параметре передается значение перемещения по оси *X*, во втором — по оси *Y*. Положительные значения первого параметра перемещают объект вправо, а отрицательные — влево. Положительные значения второго параметра смещают объект вниз, а отрицательные — вверх. Например, следующий вызов осуществляет перемещение всех рисуемых объектов вправо на 20 и вниз на 10 пикселов:

```
QPainter painter;
...
painter.translate(20, 10);
```

Масштабирование

Метод `scale()` изменяет размер изображения в соответствии с передаваемыми в него двумя множителями: для ширины и высоты. Значения меньше единицы выполняют уменьшение, больше единицы — увеличение объекта. Например, после следующего вызова ширина всех рисуемых объектов увеличится в полтора раза, а их высота уменьшится наполовину:

```
QPainter painter;
...
painter.scale(1.5, 0.5);
```

Поворот

Одной из основных операций в графике является поворот изображения на определенный угол. Для этого класс `QPainter` содержит метод `rotate()`, в который передается значение типа `double`, обозначающее угол в градусах. При положительных значениях поворот осуществляется по часовой стрелке, а при отрицательных — против нее. Следующий вызов приведет к изображению рисуемых объектов, повернутых (по часовой стрелке) на 30 градусов:

```
QPainter painter;
...
painter.rotate(30.0);
```

Скос

Этот вид трансформации также важен в компьютерной графике. Он реализуется в классе `QPainter` методом `shear()`. Первый параметр задает сдвиг по вертикали, а второй — по горизонтали. Следующий пример осуществляет скос вниз по вертикали:

```
QPainter painter;
...
painter.shear(0.3, 0.0);
```

Трансформационные матрицы

В каждом объекте класса `QPainter` хранится трансформационная матрица. Ее можно считать из объекта, а можно установить созданную матрицу трансформации с помощью метода `QPainter::setTransform()`.

Если для того чтобы получить нужный результат, вам необходимо вызывать несколько методов трансформации, то эффективнее записать их в объект трансформационной матрицы и устанавливать ее в объекте `QPainter` всякий раз, когда необходима трансформация. Например:

```
QTransform mat;
mat.scale(0.7, 0.5);
mat.shear(0.2, 0.5);
mat.rotate(15);
painter.setTransform(mat);
painter.drawText(rect(), Qt::AlignCenter, "Transformed Text");
```

Любая двумерная трансформация может быть описана матрицей размерностью 3×3:

```
M11 M12 0
M21 M22 0
Dx  Dy  0
```

Если стандартных трансформаций недостаточно, то можно определить свою собственную при помощи значений `M11`, `M12`, `M21`, `M22`, `Dx` и `Dy`, которые задаются в конструкторе класса `QTransform` и представляют собой действительные числа. В табл. 18.3 сведены вместе основные формы трансформации в матричном представлении. Например, установка следующей матрицы в объекте `QPainter` будет соответствовать вызову его метода `translate(20, 10)`:

```
QTransform mat(1, 0, 0, 1, 20, 10);
painter.setTransform(mat);
```

Таблица 18.3. *Трансформации*

Элемент матрицы	Перемещение	Поворот	Скос	Масштабирование
M11	1	cos(угол)	1	Горизонтальный компонент
M12	0	sin(угол)	Горизонтальный компонент	0
M21	0	-sin(угол)	Вертикальный компонент	0
M22	1	cos(угол)	1	Вертикальный компонент
Dx	Горизонтальный компонент	0	0	0
Dy	Вертикальный компонент	0	0	0

Графическая траектория (painter path)

Графическая траектория может состоять из различных геометрических объектов: прямоугольников, эллипсов и других фигур различной сложности. Ее основное преимущество заключается в том, что, единожды создав траекторию, ее можно отображать сколько угодно раз, одним лишь вызовом метода `QPainter::drawPath()`. Листинг 18.18 реализует траекторию, созданную из трех геометрических объектов: прямоугольника, эллипса и кривой Безье (рис. 18.20).

Листинг 18.18. Создание графической траектории

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainterPath path;
    QPointF      pt1(width(), height() / 2);
    QPointF      pt2(width() / 2, -height());
```

```
    QPointF      pt3(width() / 2, 2 * height());
    QPointF      pt4(0, height() / 2);
    path.moveTo(pt1);
    path.cubicTo(pt2, pt3, pt4);

    QRect rect(width() / 4, height() / 4, width() / 2, height() / 2);
    path.addRect(rect);
    path.addEllipse(rect);

    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setPen(QPen(Qt::blue, 6));
    painter.drawPath(path);
}
...
```

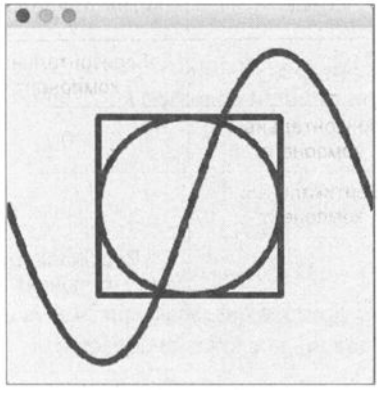


Рис. 18.20. Отображение графической траектории

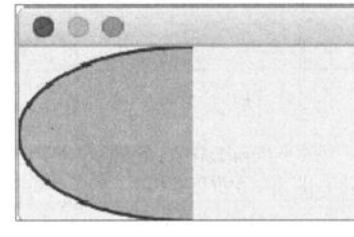


Рис. 18.21. Отсечение фигуры эллипса прямоугольной областью

Отсечения

Отсечения ограничивают вывод графики определенной областью (многоугольником или эллипсом). Если осуществляется попытка рисования за этими пределами, то оно будет невидимым. Установка прямоугольной области отсечения выполняется с помощью метода `setClipRect()`. Листинг 18.19 демонстрирует отсечение фигуры эллипса прямоугольной областью (рис. 18.21).

Листинг 18.19. Отсечение фигуры эллипса прямоугольной областью

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setClipRect(0, 0, 100, 100);
    painter.setBrush(QBrush(Qt::green));
```

```
    painter.setPen(QPen(Qt::black, 2));
    painter.drawEllipse(0, 0, 200, 100);
}
...
```

Более сложные области отсечения устанавливаются методами `QPainter::setClipRegion()` и `QPainter::setClipPath()`.

В метод `setClipRegion()` передается объект класса `QRegion`. В конструкторе класса можно задать область в виде прямоугольника или эллипса. Например, следующий вызов создаст прямоугольную область с координатами левого верхнего угла (10, 10), а также шириной и высотой, равными 100:

```
QRegion region(10, 10, 100, 100);
```

Область отсечения в виде эллипса, вписанного в такой прямоугольник, создается следующим образом:

```
QRegion region (10, 10, 100, 100, QRegion::Ellipse);
```

В качестве области отсечения можно использовать и многоугольник, передав его в конструктор. Точки в многоугольнике можно установить при помощи оператора <<. Например:

```
QRegion region(QPolygon() << QPoint(0, 100)
                          << QPoint(100, 100)
                          << QPoint(100, 0)
                          << QPoint(0, 0)
               );
```

Объекты класса `QRegion` можно комбинировать друг с другом, создавая при помощи методов `united()`, `intersected()`, `subtracted()` и `xored()` довольно сложные области:

- метод `united()` возвращает область, полученную в результате объединения двух областей;
- метод `intersected()` возвращает область, полученную в результате пересечения двух областей;
- метод `subtracted()` возвращает область, полученную в результате вычитания аргумента из исходной области;
- метод `xored()` возвращает область, содержащую точки из каждой области, но не из обеих областей.

Например:

```
QRegion region1(10, 10, 100, 100);
QRegion region2(10, 10, 100, 100, QRegion::Ellipse);
QRegion region3 = region1.subtract(region2);
painter.setClipRegion(region3);
```

Режим совмещения (composition mode)

Под *режимом совмещения* понимается задание способа совмещения пиксела источника с целевым пикселом при рисовании. Пикселы эти могут иметь различный уровень прозрачности.

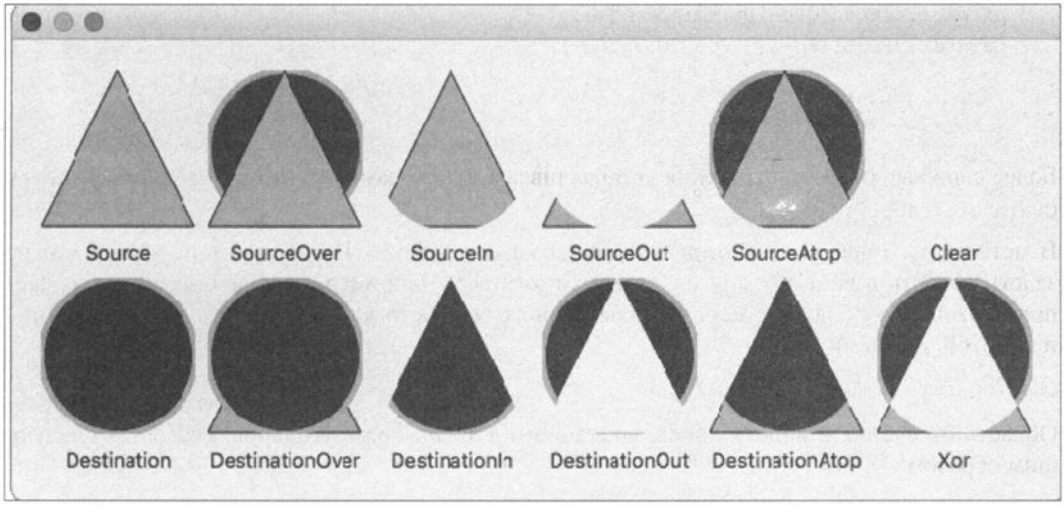


Рис. 18.22. Режимы совмещения

Такой механизм применяется для всех операций рисования, включая кисть, перо, градиенты и растровые изображения. Возможные операции проиллюстрированы на рис. 18.22.

Режим совмещения устанавливается с помощью метода `QPainter::setCompositionMode()`. По умолчанию режимом совмещения является «источник сверху»: `QPainter::CompositionMode_SourceOver`.

Если вам понадобится прохождение по пикселам растрового изображения, чтобы манипулировать альфа-каналом или цветовыми значениями, то прежде всего проверьте, нет ли для решения вашей задачи подходящего режима совмещения.

Функция `lbl()`, приведенная в листинге 18.20, предназначена для создания виджетов надписей с установленными растровыми изображениями, иллюстрирующими результат определенной операции совмещения. В виджете надписи методом `setFixedSize()` устанавливается неизменяемый размер, который впоследствии служит для инициализации объекта прямоугольной области `rect`. Этот объект используется для задания размеров исходного (`sourceImage`) и результирующего (`resultImage`) объекта растрового изображения. В качестве исходного изображения мы рисуем, вызывая метод `QPainter::drawPolygon()`, треугольник, заполненный зеленым цветом. В результирующем объекте растрового изображения вызовом метода `QPainter::drawEllipse()` рисуется окружность, заполненная красным цветом. Затем методом `QPainter::setCompositeMode()` устанавливается режим совмещения, переданный в функцию `lbl()` в переменной `mode`, и растровое (`sourceImage`) изображение рисуется в результирующем изображении методом `QPainter::drawImage()`. Метод `QLabel::setPixmap()` устанавливает результирующее растровое изображение в виджете надписи. В конце возвращается указатель на созданный виджет надписи.

Листинг 18.20. Функция `lbl()` — создание виджетов надписей с установленными растровыми изображениями (файл main.cpp)

```
QLabel* lbl(const QPainter::CompositionMode& mode)
{
    QLabel* plbl = new QLabel;
    plbl->setFixedSize(100, 100);
```

```
    QRect    rect(plbl->contentsRect());
    QPainter painter;

    QImage sourceImage(rect.size(), QImage::Format_ARGB32_Premultiplied);
    sourceImage.fill(QColor(0, 0, 0, 0));
    painter.begin(&sourceImage);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setBrush(QBrush(QColor(0, 255, 0)));
    painter.drawPolygon(QPolygon() << rect.bottomLeft()
                                   << QPoint(rect.center().x(), 0)
                                   << rect.bottomRight()
                        );
    painter.end();

    QImage resultImage(rect.size(), QImage::Format_ARGB32_Premultiplied);
    painter.begin(&resultImage);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setCompositionMode(QPainter::CompositionMode_SourceOver);
    painter.setPen(QPen(QColor(0, 255, 0), 4));
    painter.setBrush(QBrush(QColor(255, 0, 0)));
    painter.drawEllipse(rect);
    painter.setCompositionMode(mode);
    painter.drawImage(rect, sourceImage);
    painter.end();

    plbl->setPixmap(QPixmap::fromImage(resultImage));
    return plbl;
}
```

В основной функции, приведенной в листинге 18.21, создаются виджеты надписей с изображениями (вызовы функций `lbl()`) и поясняющими надписями, описывающими примененный режим совмещения. Все элементы размещаются при помощи менеджеров компоновки (layout) табличного размещения (`pgrd`) на поверхности виджета (`wgt`).

Листинг 18.21. Создание виджетов надписей с изображениями — основная функция (файл main.cpp)

```
int main(int argc, char** argv)
{
   QApplication app(argc, argv);
   QWidget      wgt;

   //Layout Setup
   QGridLayout* pgrd = new QGridLayout;
   pgrd->addWidget(lbl(QPainter::CompositionMode_Source), 0, 0);
   pgrd->addWidget(new QLabel("<CENTER>Source</CENTER>"), 1, 0);
   pgrd->addWidget(lbl(QPainter::CompositionMode_SourceOver), 0, 1);
   pgrd->addWidget(new QLabel("<CENTER>SourceOver</CENTER>"), 1, 1);
   pgrd->addWidget(lbl(QPainter::CompositionMode_SourceIn), 0, 2);
   pgrd->addWidget(new QLabel("<CENTER>SourceIn</CENTER>"), 1, 2);
   pgrd->addWidget(lbl(QPainter::CompositionMode_SourceOut), 0, 3);
```

```
    pgrd->addWidget(new QLabel("<CENTER>SourceOut</CENTER>"), 1, 3);
    pgrd->addWidget(lbl(QPainter::CompositionMode_SourceAtop), 0, 4);
    pgrd->addWidget(new QLabel("<CENTER>SourceAtop</CENTER>"), 1, 4);
    pgrd->addWidget(lbl(QPainter::CompositionMode_Clear), 0, 5);
    pgrd->addWidget(new QLabel("<CENTER>Clear</CENTER>"), 1, 5);
    pgrd->addWidget(lbl(QPainter::CompositionMode_Destination), 2, 0);
    pgrd->addWidget(new QLabel("<CENTER>Destination</CENTER>"), 3, 0);
    pgrd->addWidget(lbl(QPainter::CompositionMode_DestinationOver), 2, 1);
    pgrd->addWidget(new QLabel("<CENTER>DestinationOver</CENTER>"), 3, 1);
    pgrd->addWidget(lbl(QPainter::CompositionMode_DestinationIn), 2, 2);
    pgrd->addWidget(new QLabel("<CENTER>DestinationIn</CENTER>"), 3, 2);
    pgrd->addWidget(lbl(QPainter::CompositionMode_DestinationOut), 2, 3);
    pgrd->addWidget(new QLabel("<CENTER>DestinationOut</CENTER>"), 3, 3);
    pgrd->addWidget(lbl(QPainter::CompositionMode_DestinationAtop), 2, 4);
    pgrd->addWidget(new QLabel("<CENTER>DestinationAtop</CENTER>"), 3, 4);
    pgrd->addWidget(lbl(QPainter::CompositionMode_Xor), 2, 5);
    pgrd->addWidget(new QLabel("<CENTER>Xor</CENTER>"), 3, 5);
    wgt.setLayout(pgrd);

    wgt.show();

    return app.exec();
}
```

Графические эффекты

Графические эффекты можно применять к любым виджетам, а также и к объектам класса `QGraphicsItem` (см. *главу 21*). Устанавливаются эффекты при помощи метода `setGraphicsEffect()`. Основным классом в иерархии графических эффектов является класс `QGraphicsEffect`. Всего доступно четыре эффекта: размытие (blur), расцвечивание (colorization), тень (drop shadow) и непрозрачность (opacity). Каждый из этих эффектов реализован в отдельном классе, каждый такой класс унаследован от класса `QGraphicsEffect` (рис. 18.23).

Если четырех эффектов для вас недостаточно, вы можете реализовать класс собственного эффекта, — для этого необходимо унаследовать класс `QGraphicsEffect` и перезаписать метод `draw()`, потому что именно этот метод вызывается всякий раз, когда эффект нуждается в перерисовке. В методе `draw()` в вашем распоряжении будет указатель на объект `QPainter`, с помощью которого можно выполнить все необходимые графические операции.

Кроме того, графические эффекты можно очень удачно комбинировать с анимацией (см. *главу 22*).

Покажем использование трех эффектов на примере, приведенном в листингах 18.22 и 18.23 (рис. 18.24).

Функция `lbl()`, приведенная в листинге 18.22, создает виджеты надписей сразу с растровыми изображениями. В нее передается объект эффекта, который применяется вызовом метода `setGraphicsEffect()`. В случае, если указатель на объект эффекта равен нулевому значению, то это означает, что объект эффекта отсутствует, и метод применения эффекта вызываться не должен. В завершение из метода возвращается указатель на созданный виджет надписи.

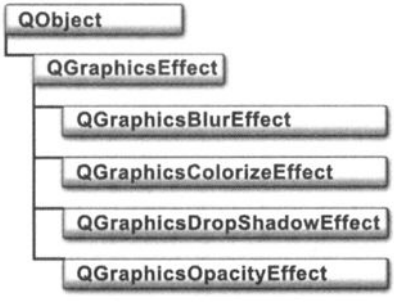


Рис. 18.23. Классы графических эффектов

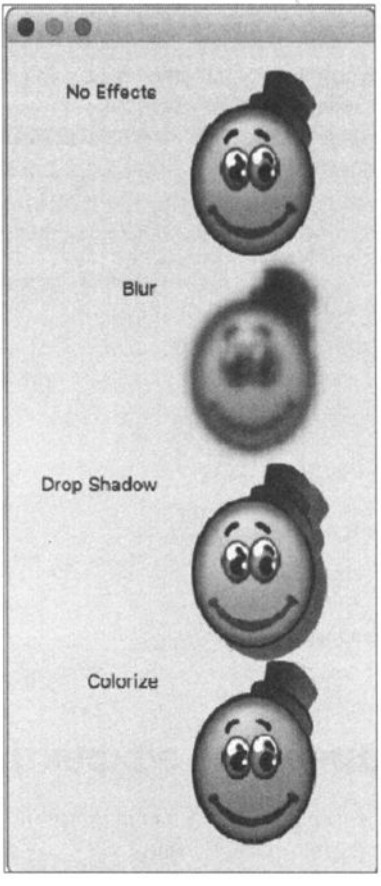


Рис. 18.24. Демонстрация трех графических эффектов: размытие (Blur), тень (Drop Shadow) и расцвечивание (Colorization)

Листинг 18.22. Функция `lbl()` — создание виджетов надписей с установленными растровыми изображениями (файл main.cpp)

```
QLabel* lbl(QGraphicsEffect* pge)
{
    QLabel* plbl = new QLabel;
    plbl->setPixmap(QPixmap(":/happyos.png"));

    if (pge) {
        plbl->setGraphicsEffect(pge);
    }
    return plbl;
}
```

В основной функции программы, приведенной в листинге 18.23, создаются объекты трех эффектов (указатели `pBlur`, `pShadow` и `pColorize`). При помощи табличного размещения `QFormLayout` мы размещаем надписи с виджетами, к которым были применены эффекты.

Листинг 18.23. Создание объектов эффектов — основная функция (файл main.cpp)

```
int main(int argc, char** argv)
{
    QApplication                app(argc, argv);
    QWidget                     wgt;
    QGraphicsBlurEffect*        pBlur     = new QGraphicsBlurEffect;
    QGraphicsDropShadowEffect*  pShadow   = new QGraphicsDropShadowEffect;
    QGraphicsColorizeEffect*    pColorize = new QGraphicsColorizeEffect;

    //Layout Setup
    QFormLayout* pform = new QFormLayout;
    pform->addRow("No Effects", lbl(0));
    pform->addRow("Blur", lbl(pBlur));
    pform->addRow("Drop Shadow", lbl(pShadow));
    pform->addRow("Colorize", lbl(pColorize));
    wgt.setLayout(pform);

    wgt.show();

    return app.exec();
}
```

Резюме

Объект класса `QPainter` выполняет рисование на объектах классов, унаследованных от класса `QPaintDevice`. Класс `QPainter` предоставляет методы для рисования точек, линий, эллипсов, растровых изображений, текста (см. *главу 20*) и др. Для рисования класс `QPainter` предоставляет перо и кисть. Перо служит для рисования дуг, линий и контуров замкнутых фигур. Кисти используются для заполнения внутренней части фигуры. Все команды рисования можно сохранять в объектах класса `QPicture`, который, в свою очередь, позволяет сохранять команды в файле и считывать их оттуда.

Эффект мерцания возникает в том случае, когда пикселы за короткие промежутки времени перерисовываются разными цветами. Использование механизма двойной буферизации, встроенного в Qt, автоматически подавляет этот нежелательный эффект.

Qt поддерживает три типа градиентов: линейный, конический и радиальный. С их помощью можно придать объемность некоторым элементам вашей программы.

Режим сглаживания позволяет улучшить визуальный эффект, убирая ступенчатость на контурах выводимых геометрических фигур.

Класс `QPainter` предоставляет возможность проведения геометрических преобразований, таких как перемещение, поворот, масштабирование и скос.

Отсечения используются для ограничения вывода графики заданной областью. Класс `QRegion` служит для задания областей, которые могут иметь очень сложные формы.

Графическая траектория позволяет задавать произвольные формы геометрических фигур, соединяя геометрические фигуры вместе.

Режим совмещения задает способ объединения пиксела источника и целевого пиксела.

При помощи объектов, унаследованных от класса `QGraphicsEffect`, можно легко применять к виджетам различные графические эффекты.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/18-510/** или с помощью следующего QR-кода (рис. 18.25):



Рис. 18.25. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 19

Растровые изображения

Рисунок расскажет больше чем тысячи слов.
Древняя китайская мудрость

Растровые изображения представляют собой набор цветовых значений, именуемых пикселами. *Пикселы* — это «клеточки», формирующие графический образ на устройстве вывода. Глаз человека не способен различать отдельные такие клеточки, поэтому мозг синтезирует общую картину, соединяя их в одно целое.

Форматы графических файлов

Растровые изображения можно как записывать в файлы разных форматов, так и загружать из них. Qt поддерживает следующие растровые форматы: PNG, BMP, ICO, TGA, TIFF, XBM, XPM, PNM, JPEG, MNG, GIF, PNM, PBM, PGM и PPM. Приведем описание некоторых наиболее распространенных форматов.

> ***ПЕРЕДАВАЙТЕ КЛИЕНТАМ ФАЙЛЫ РАСШИРЕНИЙ ВМЕСТЕ С САМИМ ПРИЛОЖЕНИЕМ!***
>
> Если вы написали приложение с поддержкой графических файлов и собираетесь передать его клиентам, то не забудьте позаботиться о том, чтобы система расширений для нужных форматов была передана вместе с вашим приложением, — иначе файлы этих форматов отображаться у них не будут. Например, для ОС Windows сами файлы системы расширений *qgif.dll*, *qico.dll*, *qjpeg.dll*, *qtiff.dll*, *qtga.dll*, *qw.bmp* должны находиться в каталоге *<MyApplication>/imageformats/*. Для Linux используется тот же подкаталог, а для Mac OS X воспользуйтесь утилитой macdeployqt, которая сделает эту работу за вас. Для Windows подобная утилита называется windeployqt.

Формат BMP

Формат BMP (сокр. от Bit map) — растровый формат для ОС Windows. Он используется для хранения практически всех типов растровых данных, а также поддерживает любые разрешения экрана. Данные в этом формате почти всегда хранятся в несжатом виде и поэтому занимают сравнительно много места. Структура формата BMP тесно связана с интерфейсом прикладного программирования (API) для ОС Windows. Формат BMP никогда не рассматривался как переносимый и не использовался для обмена растровыми изображениями между операционными системами, но с поддержкой этого формата в Qt все изменилось — он стал платформонезависимым.

Формат GIF

Формат GIF (Graphics Interchange Format, формат обмена графическими данными) произносится как «джиф» — один из самых популярных растровых форматов в Интернете. Основное его преимущество состоит в высокой степени сжатия без потерь, что достигается применением алгоритма сжатия LZW (Lempel-Ziv-Welch, по фамилиям разработчиков: Лемпель, Зив и Велч). GIF поддерживает и анимацию. Недостатками этого формата являются поддержка только 8-битной глубины цвета и требование лицензионных отчислений за каждую программу, использующую LZW-алгоритм.

Формат PNG

Формат PNG (Portable Network Graphics, переносимая сетевая графика) произносится как «пинг» — разработан как альтернатива GIF в пику юридическим сложностям, связанным с требованиями его оплаты при использовании. Неофициальная трактовка названия PNG — «PNG's Not GIF» («PNG — это не GIF»). Подавляющее большинство веб-браузеров поддерживают этот формат, который не сложен в реализации, а по своим функциональным возможностям даже превосходит GIF. PNG распространяется бесплатно, что позволяет избежать бремени лицензионных платежей и патентных сборов. Как и в формате GIF, в нем обеспечивается сжатие данных без потерь и поддерживается прозрачность. Кроме того, он обеспечивает поддержку глубины цвета до 48 битов.

> ***Альтернатива для анимированных файлов формата GIF***
>
> В качестве альтернативы для анимированных файлов формата GIF можно использовать формат MNG, хранящий в себе серии изображений формата PNG.

Формат JPEG

Формат JPEG получил свое название от Joint Photographic Experts Group, объединенной экспертной группы по фотографии (организации, разработавшей стандарт и метод сжатия) и произносится как «джейпег». Этот формат разрабатывался с 1991 по 1993 г., после чего был стандартизирован. Его отличительная особенность — очень высокая степень сжатия, но с потерей информации, поэтому файлы такого формата используются, в основном, для хранения фотографических изображений, поскольку именно на них наименее всего заметны погрешности сжатия.

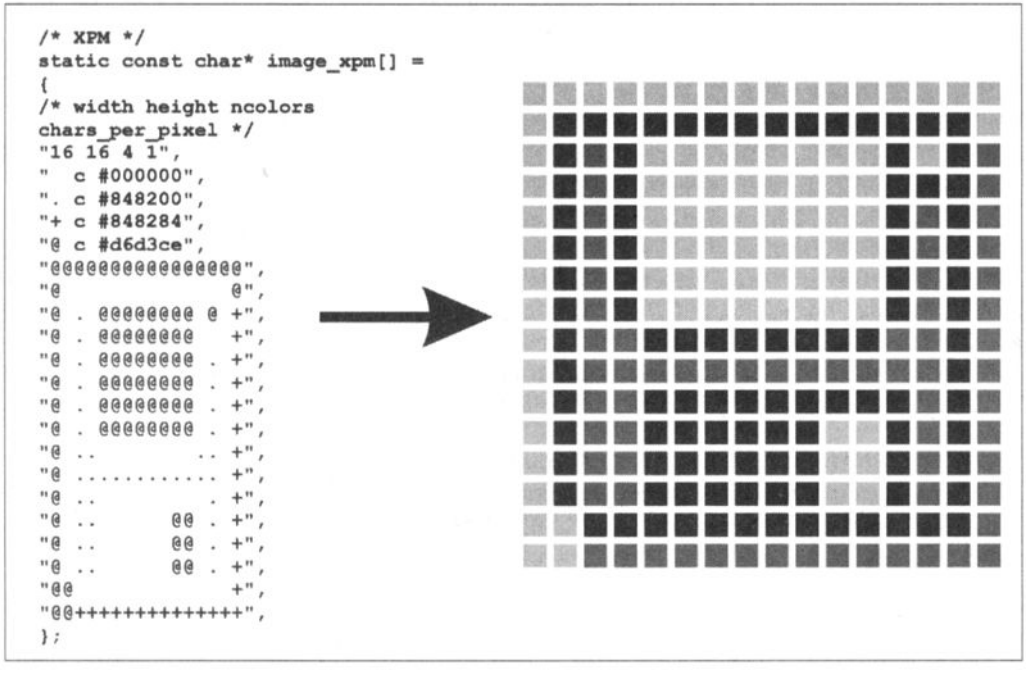
Формат XPM

XPM (XPixMap) — формат, распространенный в системе X11 (UNIX). Мы привыкли к тому, что изображения хранятся в виде двоичной информации, но в XMP-формате данные хранятся в виде исходного кода на языке C, который можно вставлять в свои программы. Это позволяет превратить обычный текстовый редактор в инструмент для создания и изменения растровых изображений. Формат можно использовать для любых разрешений экрана и 24-битной глубины цвета.

> ***Учтите, что формат XPM неэкономично расходует дисковое пространство!***
>
> Ввиду того, что формат XPM неэкономично расходует дисковое пространство, его лучше использовать для небольших по размеру растровых изображений. Этот формат очень интенсивно использовался до третьей версии Qt, но с появлением в четвертой версии систе-

мы ресурсов (см. *главу 3*) обращение к нему потеряло былую актуальность, так как стало возможным включать в исполняемый код программы и библиотеки более экономичные форматы, такие, например, как PNG и JPEG.

На рис. 19.1 показано содержимое файла в формате XPM, представляющее собой код на языке C, в котором определен массив, хранящий растровые данные (*слева*), и само растровое изображение (*справа*).



```
/* XPM */
static const char* image_xpm[] =
{
/* width height ncolors
chars_per_pixel */
"16 16 4 1",
"  c #000000",
". c #848200",
"+ c #848284",
"@ c #d6d3ce",
"@@@@@@@@@@@@@@@@",
"@              @",
"@ . @@@@@@@@ @ +",
"@ . @@@@@@@@   +",
"@ . @@@@@@@@ . +",
"@ . @@@@@@@@ . +",
"@ . @@@@@@@@ . +",
"@ . @@@@@@@@ . +",
"@ ..         .. +",
"@ ............ +",
"@ ..         . +",
"@ ..      @@ . +",
"@ ..      @@ . +",
"@ ..      @@ . +",
"@@            +",
"@@++++++++++++++",
};
```

Рис. 19.1. Файл в формате XPM

В первой строке файла стоит комментарий `/*XPM*/`, сообщающий о формате файла. За ним следует константный указатель на массив `image_xpm`, задающий растровое изображение. Первые два целых числа в строке устанавливают ширину и высоту растрового изображения в пикселах (в нашем случае размер 16×16 пикселов). Третье целое число указывает количество цветов (в примере оно равно 4). Четвертое число определяет количество знаков на пиксел — в нашем случае это один знак. Следующие четыре значения задают цветовую палитру из четырех цветов, которые кодируются символом `#`, за которым следует шестнадцатеричная запись цвета в формате RGB (такое же обозначение цветов принято в формате HTML). Символ отделяется от цветового значения буквой `c`, которая является сокращением для слова color (цвет).

Символические имена для цветов

Можно вместо буквы `c` использовать букву `s`, что дает возможность использования символических имен для цветов. Так, например, указав после этого символа `None`, можно задать символ для прозрачного цвета.

В примере на рис. 19.1 для черного цвета используется символ пробела, для коричневого — точка, для темно-серого — плюс, а для светло-серого — `@`. Затем следует расположение пикселов в матрице растрового изображения.

Контекстно-независимое представление

Контекстно-независимое представление не связано с возможностями графической карты компьютера и, следовательно, с графическим режимом, установленным в операционной системе. Данные растрового изображения помещаются в обычный массив, что дает возможность эффективного обращения к каждому из пикселов в отдельности, а также позволяет эффективно выполнять операции записи и считывания файлов растровых изображений.

Класс *QImage*

Класс `QImage` является основным классом для контекстно-независимого представления растровых изображений. Этот класс унаследован от класса контекста рисования `QPaintDevice`, что позволяет использовать все методы рисования, определенные в классе `QPainter`.

Класс `QImage` предоставляет поддержку для форматов, указанных в табл. 19.1.

Таблица 19.1. *Перечисление языка Си* `Format` *класса* `QImage`

Константа	Описание
`Format_Invalid`	Растровое изображение недействительно
`Format_Mono`	Каждый пиксел представлен одним битом. Биты укомплектованы в байте таким образом, что первый является старшим разрядом
`Format_MonoLSB`	Каждый пиксел представлен одним битом. Биты укомплектованы в байте таким образом, что первый является младшим разрядом
`Format_Index8`	Данные растрового изображения представляют собой 8-битные индексы цветовой палитры (см. *главу 17*)
`Format_RGB32`	Каждый пиксел представлен тридцатью двумя битами. Однако значение для альфа-канала всегда равно значению 0xFF, то есть непрозрачно
`Format_ARGB32`	Каждый пиксел представлен 32 битами
`Format_ARGB32_Premultiplied`	Практически идентичен формату `Format_ARGB32`, но код оптимизирован для использования объекта `QImage` в качестве контекста рисования

Значение формата можно узнать с помощью метода `format()`. Для конвертирования растрового изображения из одного формата в другой предусмотрен метод `convertToFormat()`, который возвращает новый объект `QImage`.

Чтобы создать объект этого класса, необходимо в конструктор передать ширину, высоту и формат растрового изображения. Например, следующая строка создает растровое изображение шириной 320 и высотой 240 пикселов и глубиной цвета 32 бита.

```
QImage img(320, 240, QImage::Format_RGB32);
```

Содержимое для объекта `QImage` также можно считать из файла, хранящего растровое изображение, передав имя файла в конструктор при создании объекта класса `QImage`. Если графический формат файла поддерживается Qt, то данные будут загружены и автоматически установятся ширина, высота и формат. Например:

```
QImage img("lisa.jpg");
```

В конструктор можно передавать и указатель на массив данных XPM. Например, загрузка растровых данных, приведенных на рис. 19.1, будет выглядеть следующим образом:

```
#include "image_xpm.h"
...
QImage img(image_xpm);
```

В качестве альтернативы для считывания файла можно воспользоваться методом `load()`. Несомненное достоинство этого метода в том, что так можно загружать растровые изображения в любой момент. Первым параметром передается имя файла, а вторым — формат. Формат файла обозначается строкой типа `unsigned char*`, принимающей одно из следующих строковых значений: GIF, BMP, JPG, XPM, XBM или PNG. Если во втором параметре вообще ничего не передавать, то класс `QImage` попытается распознать графический формат самостоятельно. Например:

```
QImage img;
img.load("lisa.jpg");
```

При помощи метода `save()` можно сохранять растровое изображение из объекта класса `QImage` в файл. Первым параметром передается имя файла, а вторым — формат, в котором он должен быть сохранен, третьем параметром можно передать значение качества. Значение качества варьируется в диапазоне от 0 до 100. Чем меньше значение параметра качества, тем меньше размер файла, но хуже качество изображения, и наоборот — чем больше значение этого параметра, тем больше размер файла, но лучше качество изображения. Например:

```
QImage img(320, 240, 32, QColor::blue);
img.save("blue.jpg", "JPG", 85);
```

Использование формата GIF требует лицензионных отчислений!

Ввиду того, что фирма Unisys имеет патент на метод сжатия LZW, используемый форматом GIF, в некоторых странах создание файлов в формате GIF без соответствующей лицензии является незаконным. По этой причине Qt не предоставляет возможность сохранения в формате GIF.

Класс `QImage` просто незаменим по эффективности, когда нужно изменить или получить цвета пикселов растрового изображения. RGB-значение пиксела с координатами (*X*, *Y*) можно получить с помощью метода `pixel(x, y)`. Для записи RGB-значения используется структура данных `QRgb` (см. *главу 17*). Например:

```
QRgb rgb = img.pixel(250, 100);
```

Особенность использования метода `pixelIndex()`

Метод `pixelIndex()` возвращает индекс палитры пиксела (см. *главу 17*) с координатами (*X*, *Y*). Этот метод работает только для растровых изображений, имеющих глубину цвета 8 битов.

Установить пикселу с координатами (*X*, *Y*) новое RGB-значение можно методом `setPixel(x, y, rgb)`.

Особенность растровых изображений, имеющих глубину цвета 8 битов

Для растровых изображений, имеющих глубину цвета 8 битов, значение `rgb` будет соответствовать индексу палитры (см. *главу 17*), которое может быть установлено вызовом метода `setColor()`.

Например:

```
QRgb rgb = qRgb(200, 100, 0);
img.setPixel(20, 50, rgb);
```

Данные растрового изображения хранятся в объектах класса `QImage` построчно, и в каждой строке пикселы расположены слева направо. `QImage` также содержит массив указателей, указывающих на начало каждой строки изображения.

> ***О ПОСЛЕДНЕМ БАЙТЕ СТРОКИ ИЗОБРАЖЕНИЯ***
>
> При использовании глубины цвета в один бит, каждый байт хранит данные восьми последовательно идущих по горизонтали пикселов (см. табл. 19.1). Если количество битов ширины изображения не делится на восемь, то последний байт строки будет содержать биты, значения которых можно проигнорировать, а следующая строка начнется с нового байта.

С помощью метода `scanLine()` можно получить адрес строки, номер которой соответствует значению, переданному в этот метод. Имеет смысл передавать в него индексы строк, лежащие в диапазоне от 0 до высоты растрового изображения, уменьшенного на единицу. Проиллюстрируем использование этого метода на небольшом примере (листинги 19.1–19.3), где реализуем функцию `brightness()`, которая будет уменьшать либо увеличивать яркость растровых изображений. Результаты действий этой функции показаны на рис. 19.2.

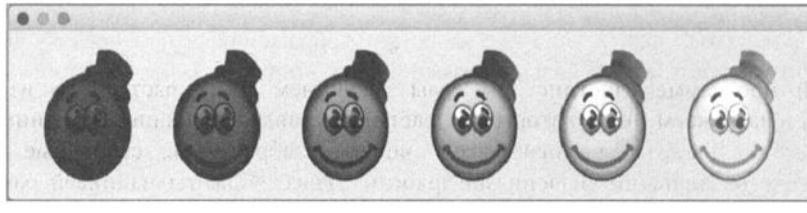


Рис. 19.2. Функция увеличения/уменьшения яркости

Функция `brightness()` (листинг 19.1) принимает два параметра: первый — растровое изображение, второй — значение яркости. Мы не изменяем переданный объект растрового изображения и создаем поэтому для дальнейших изменений его копию (объект `imgTemp`). Далее работаем только с объектом копии. Мы получаем его размеры вызовом методов `width()` и `height()`. Запускаем цикл перебора строк от 0 до `height()`. Для каждой новой строки вызываем метод `scanLine()` и устанавливаем на нее указатель `tempLine`. В следующем цикле мы идем вдоль строки до значения `width()`. Исходя из текущего значения указателя строки, вызовом методов `qRed()`, `qGreen()`, `qBlue()` опрашиваем значение компонентов RGB и вычисляем новые значения, прибавляя к ним значения яркости (переменная `n`). Значения альфа-канала оставляем неизменным. Все эти значения у нас теперь хранятся в промежуточных переменных `r`, `g`, `b` и `a`. Далее мы присваиваем значению, на которое указывает указатель строки (`tempLine`), новое значение `QRgb`, для этого вызываем функцию `qRgba()` и устанавливаем в ней вычисленные нами значения с учетом того, чтобы они не выходили за диапазон от 0 до 255. После чего увеличиваем наш указатель строки на единицу и тем самым перемещаемся вдоль нее на следующий пиксел. В конце работы цикла мы возвращаем из функции получившийся объект растрового изображения `imgTemp`.

Листинг 19.1. Функция `brightness()`

```
QImage brightness(const QImage& imgOrig, int n)
{
    QImage imgTemp = imgOrig;
```

```
    qint32 nHeight = imgTemp.height();
    qint32 nWidth  = imgTemp.width();

    for (qint32 y = 0; y < nHeight; ++y) {
        QRgb* tempLine = reinterpret_cast<QRgb*>(imgTemp.scanLine(y));

        for (qint32 x = 0; x < nWidth; ++x) {
            int r = qRed(*tempLine) + n;
            int g = qGreen(*tempLine) + n;
            int b = qBlue(*tempLine) + n;
            int a = qAlpha(*tempLine);

            *tempLine++ = qRgba(r > 255 ? 255 : r < 0 ? 0 : r,
                                g > 255 ? 255 : g < 0 ? 0 : g,
                                b > 255 ? 255 : b < 0 ? 0 : b,
                                a
                               );
        }
    }

    return imgTemp;
}
```

В основной программе (листинг 19.2) мы загружаем файл растрового изображения happyos.png и запускаем цикл, который создает новые виджеты надписей. В них мы вызовом метода `setPixmap()` устанавливаем новые изображения, созданные функцией `brightness()` с различными значениями яркости. Далее виджеты надписей размещаются в горизонтальном порядке вызовом метода `addWidget()`.

Листинг 19.2. Функция `main()`

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QImage       img(":/happyos.png");
    QWidget      wgt;

    QHBoxLayout* phbx = new QHBoxLayout;
    phbx->setSpacing(0);

    for (int i = -150; i < 150; i += 50) {
        QLabel* plbl = new QLabel;
        plbl->setFixedSize(img.size());
        plbl->setPixmap(QPixmap::fromImage(brightness(img, i)));
        phbx->addWidget(plbl);
    }

    wgt.setLayout(phbx);
    wgt.show();

    return app.exec();
}
```

Объект класса `QImage` можно отобразить в контексте рисования методом `QPainter::drawImage()`. Перед тем как отобразить объект `QImage` на экране, метод `drawImage()` преобразует его в контекстно-зависимое представление (объект класса `QPixmap`). В листинге 19.3 приведен вывод изображения с позиции (0, 0) (рис. 19.3).

Листинг 19.3. Вывод растрового изображения

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    QImage   img(":/lisa.jpg");
    painter.drawImage(0, 0, img);
}
...
```

Если необходимо вывести только часть растрового изображения, то следует указать эту часть в дополнительных параметрах метода `drawImage()`. В листинге 19.4 приведен пример отображения участка растрового изображения, задаваемого координатой (30, 30) и имеющей ширину равной 110, а высоту — 100 пикселам (рис. 19.4).

Листинг 19.4. Вывод части растрового изображения

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    QImage   img(":/lisa.jpg");
    painter.drawImage(0, 0, img, 30, 30, 110, 100);
}
...
```



Рис. 19.3. Вывод объекта `QImage` в контексте рисования



Рис. 19.4. Вывод части объекта `QImage` в контексте рисования

Вызвав метод `invertPixels()` и передав в него значение `QImage::InvertRgb`, можно управлять инвертированием пикселов. А передача значения `QImage::InvertRgba` позволяет инвертировать не только пикселы, но и альфа-канал. В листинге 19.5 показана реализация этой возможности, а результат отображен на рис. 19.5.

Листинг 19.5. Инвертирование пикселов

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    QImage img(":/lisa.jpg");
    painter.drawImage(0, 0, img);
    img.invertPixels(QImage::InvertRgb);
    painter.drawImage(img.width(), 0, img);
}
...
```

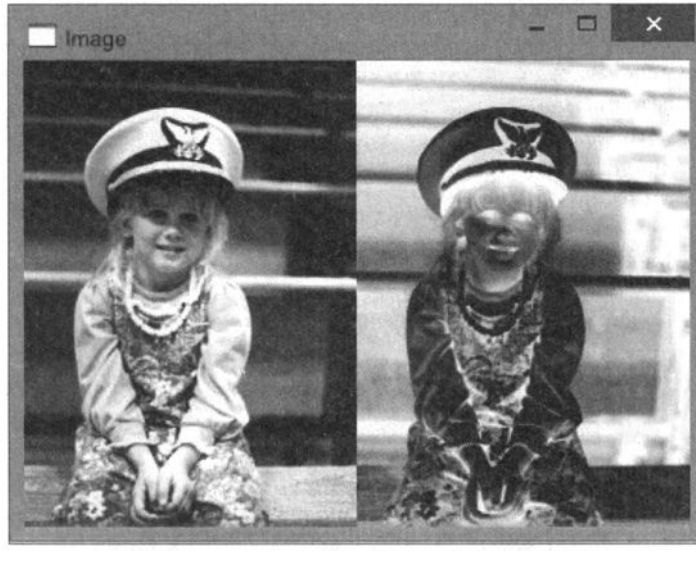


Рис. 19.5. Инвертирование значений пикселов

При помощи метода `scaled()` можно получить новое растровое изображение с измененными размерами. Действие флагов `Qt::IgnoreAspectRatio` и `Qt::KeepAspectRatio`, управляющих изменением размеров, рассмотрено в *главе 17*. Листинг 19.6 демонстрирует возможность изменения размеров растрового изображения, а результаты такого изменения показаны на рис. 19.6.

Листинг 19.6. Изменение размеров растрового изображения

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
```

```
    QImage img1(":/lisa.jpg");
    painter.drawImage(0, 0, img1);

    QImage img2 =
     img1.scaled(img1.width() / 2, img1.height(), Qt::IgnoreAspectRatio);
    painter.drawImage(img1.width(), 0, img2);

    QImage img3 =
     img1.scaled(img1.width(), img1.height() / 2, Qt::IgnoreAspectRatio);
    painter.drawImage(0, img1.height(), img3);

    QImage img4 =
     img1.scaled(img1.width() / 2, img1.height(), Qt::KeepAspectRatio);
    painter.drawImage(img1.width(), img1.height(), img4);
}
...
```



Рис. 19.6. Изменение размеров растрового изображения

Класс `QImage` предоставляет возможность горизонтального или вертикального отражения растрового изображения. Для этого в метод `mirrored()` необходимо передать два булевых значения, управляющих горизонтальным и вертикальным отражениями соответственно. Метод `mirrored()` не изменяет растровое изображение объекта, из которого он был вызван,

а создает новое. В листинге 19.7 реализуются и вертикальное, и горизонтальное отражения. Результат показан на рис. 19.7.

Листинг 19.7. Отражение изображения

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    QImage img(":/lisa.jpg");
    painter.drawImage(0, 0, img);
    painter.drawImage(img.width(), 0, img.mirrored(true, true));
}
...
```



Рис. 19.7. Отражение изображения

Класс *QImage* как контекст рисования

Как уже упоминалось, класс `QImage` является контекстом рисования. Его целесообразно использовать в тех случаях, когда соображения качества и точности отображения преобладают над скоростью. Одно из необходимых условий для использования объекта класса `QImage` в качестве контекста рисования — формат изображения должен быть 32-битным, т. е. следующим: `QImage::Format_ARGB32` или `QImage::FormatARGB32_Premultiplied`. Формат `QImage::FormatARGB32_Premultiplied` является более предпочтительным, поскольку он оптимизирован для операций рисования. Операция рисования продемонстрирована в листинге 19.8. Результат показан на рис. 19.8 — окно программы динамически изменяет размеры эллипса в соответствии с изменениями своих размеров.

В листинге 19.8 в методе обработки события рисования `paintEvent()` создается объект класса `QImage` с размерами виджета, которые возвращаются методом `size()`. После создания

объекта рисования класса `QPainter` в его метод `begin()` передается адрес объекта `QImage` в качестве контекста рисования. Метод `setRenderHint()` устанавливает режим сглаживания `QPainter::Antialiasing`, второй параметр `true` указывает, что этот режим надо включить (`false` означало бы, что его надо отключить). Метод `eraseRect()` очищает прямоугольную область, указанную в его параметре. В нашем случае эта область соответствует текущей области виджета, которую возвращает метод `rect()`. Рисование эллипса осуществляется методом `drawEllipse()`. Вызов метода `end()` закрывает блок рисования на объекте `QImage` — чтобы мы могли использовать объект рисования с контекстом виджета. Для этого его указатель передается в метод `begin()`. Затем, методом `drawImage()` содержимое объекта `QImage` отображается в области виджета.

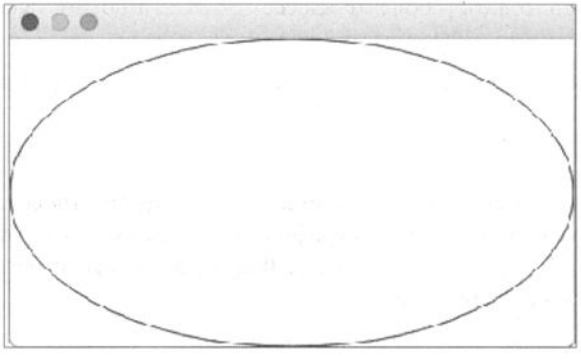


Рис. 19.8. Динамическое изменение размеров эллипса в соответствии с изменениями размеров окна

Листинг 19.8. Рисование на объекте QImage

```
void ImageDraw::paintEvent(QPaintEvent* pe)
{
    QImage   img(size(), QImage::Format_ARGB32_Premultiplied);
    QPainter painter;

    painter.begin(&img);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.eraseRect(rect());
    painter.drawEllipse(0, 0, size().width(), size().height());
    painter.end();

    painter.begin(this);
    painter.drawImage(0, 0, img);
    painter.end();
}
```

Контекстно-зависимое представление

Контекстно-зависимое представление позволяет отображать растровые изображения на экране гораздо быстрее, чем контекстно-независимое, поскольку оно не требует проведения дополнительных преобразований. К объектам контекстно-зависимого представления можно

применять все методы класса QPainter. Большинство процессоров графических карт обладают способностью отображать графические примитивы, например линии, многоугольники (полигоны) и поверхности, не задействуя при этом основного процессора, благодаря чему очень сильно ускоряются графический вывод и работа самой программы.

Класс *QPixmap*

Этот класс унаследован от класса контекста рисования QPaintDevice. Его определение находится в заголовочном файле QPixmap. Объекты класса содержат растровые изображения, не отображая их на экране. При необходимости этот класс можно использовать в качестве промежуточного буфера для рисования — то есть, если требуется нарисовать изображение, его можно сначала нарисовать в объекте класса QPixmap, а потом, используя объект класса QPainter, скопировать в видимую область.

Для создания объекта этого класса в его конструктор нужно передать ширину и высоту. Например:

```
QPixmap pix(320, 240);
```

Глубина цвета в создаваемом объекте класса QPixmap автоматически будет установлена в соответствии с актуальным значением графического режима. Это значение можно узнать с помощью метода QPixmap::defaultDepth(). Файл растрового изображения можно передать прямо в конструктор. Например:

```
QPixmap pix(":/forest.jpg");
```

Класс QPixmap, как и QImage, предоставляет возможность загрузки данных в формате XPM прямо в конструктор:

```
#include "image_xpm.h"
...
QPixmap pix(image_xpm);
```

Объекты класса QPixmap содержат не сами данные, а их идентификаторы, с помощью которых они могут обратиться к системе. Поэтому прямой доступ к каждому пикселу в отдельности будет очень медленным. В этом случае разумнее будет воспользоваться классом QImage.

В распоряжении класса QPainter имеются методы для сохранения и записи графических изображений: load() и save(). При проведении этих операций будет осуществляться промежуточное конвертирование из объекта класса QImage (или в него).

Объект класса QPixmap отображается в видимой области с помощью метода QPainter::drawPixmap(). Листинг 19.9 демонстрирует два разных варианта вызова метода drawPixmap(). В первом варианте указана только позиция, с которой нужно осуществить вывод. Во втором — вывод задается прямоугольной областью, в которой должно отображаться растровое изображение. Результат вывода показан на рис. 19.9.

Листинг 19.9. Вывод растрового изображения

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
```

```
    QPixmap pix(":/forest.jpg");
    painter.drawPixmap(0, 0, pix);

    QRect r(pix.width(), 0, pix.width() / 2, pix.height());
    painter.drawPixmap(r, pix);
}
...
```

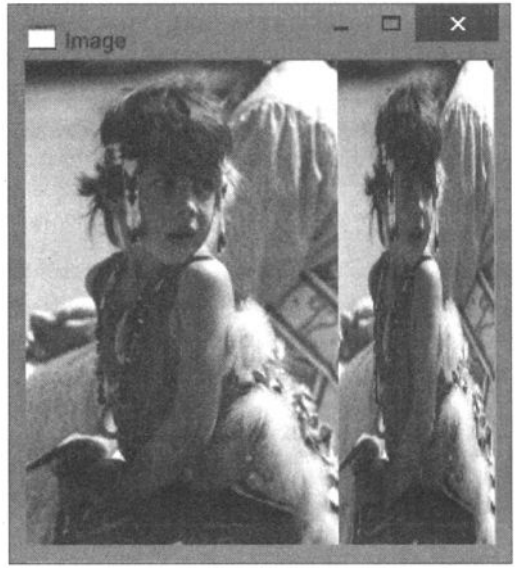


Рис. 19.9. Разные способы отображения объекта класса `QPixmap`

Класс *QPixmapCache*

Этот класс реализует кэш для объектов `QPixmap`. Все операции выполняются с глобальным объектом кэша, поэтому все методы этого класса определены как статические. С помощью метода `insert()` можно поместить объект класса `QPixmap` с ключом строкового типа в кэш.

Если растровое изображение было загружено из файла, то в качестве строки удобно использовать имя файла. Передавая строку-ключ в метод `find()`, можно получить указатель на растровое изображение, внесенное в кэш. В кэш имеет смысл помещать растровые изображения, часто используемые в программе, — чтобы избежать загрузки из файла при каждом обращении к ним.

Класс *QBitmap*

Класс `QBitmap` унаследован от класса `QPixmap` и определен в заголовочном файле QBitmap. Объекты класса предназначены для хранения растровых изображений, обладающих глубиной цвета, равной одному биту. Это позволяет хранить изображения, имеющие только два цвета: `Qt::color0` и `Qt::color1`. Такое изображение называется *двухуровневым* (bi-level). Класс используется в основном для хранения указателей мыши и масок прозрачности.

Создание нестандартного окна виджета

Для того чтобы сделать окно виджета не квадратным, а любой другой формы, нужно просто сделать сам фон виджета прозрачным. Это достигается вызовом метода `setAttribute()` и установкой в нем флага `Qt::WA_TranslucentBackground`.

Продемонстрируем использование этого флага небольшой программой (листинг 19.10), результат исполнения которой показан на рис. 19.10. На нем мы видим картинку с небольшой кнопкой в виде крестика слева. Нажатие на эту кнопку завершает нашу программу. Благодаря тому, что некоторые графические форматы, например GIF, PNG и XMP, могут содержать прозрачность, в нашем примере форму окну будет задавать уровень прозрачности из PNG-файла.

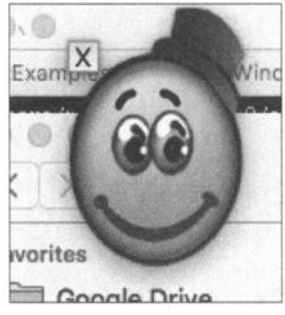


Рис. 19.10. Пример нестандартного окна программы с прозрачным фоном

Листинг 19.10. Использование флага `Qt::WA_TranslucentBackground` (файл main.cpp)

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    Window       wnd;

    wnd.setAttribute(Qt::WA_TranslucentBackground);
    wnd.setPixmap(QPixmap(":/happyos.png"));

    QPushButton* pcmdQuit = new QPushButton("X");
    pcmdQuit->setFixedSize(16, 16);
    QObject::connect(pcmdQuit, SIGNAL(clicked()), &app, SLOT(quit()));

    //setup layout
    QVBoxLayout* pvbx = new QVBoxLayout;
    pvbx->addWidget(pcmdQuit);
    pvbx->addStretch(1);
    wnd.setLayout(pvbx);

    wnd.show();

    return app.exec();
}
```

В листинге 19.10 мы создаем виджет нашего окна (`wnd`) и устанавливаем в нем атрибут прозрачности виджета фона `Qt::WA_TranslucentBackground`. Вот и все, теперь осталось только

позаботиться о том, чтобы пользователь смог завершать само приложение. Для этого мы используем виджет кнопки нажатия, задаем ей неизменяемый размер 16×16 пикселов, и, чтобы наше приложение завершалось при нажатии этой кнопки, соединяем ее сигнал `clicked()` со слотом `QCoreApplicaion::quit()` объекта приложения `app`. Саму кнопку размещаем на нашем виджете надписи при помощи вертикального размещения (указатель `pvbx`).

Для того чтобы убрать заголовок окна, мы передаем в конструктор `QLabel` значение модификации свойства окна `Qt::FramelessHint`. Но для окна без заголовка нужно позаботиться о возможности его перемещения. Для этого в классе `Window` (листинг 19.11) переопределяются методы событий мыши `mousePressEvent()` и `mouseMoveEvent()`, реализуя тем самым код, необходимый для изменения расположения окна на экране. Атрибут `m_ptPosition` нужен для хранения координат указателя мыши относительно начала окна виджета.

Листинг 19.11. Код, необходимый для изменения расположения окна на экране (файл main.cpp)

```
class Window : public QLabel {
private:
    QPoint m_ptPosition;

protected:
    virtual void mousePressEvent(QMouseEvent* pe)
    {
        m_ptPosition = pe->pos();
    }

    virtual void mouseMoveEvent(QMouseEvent* pe)
    {
        move(pe->globalPos() - m_ptPosition);
    }

public:
    Window(QWidget* pwgt = 0)
        : QLabel(pwgt,  Qt::FramelessWindowHint | Qt::Window)
    {
    }
};
```

Резюме

Qt поддерживает много форматов файлов растровых изображений. В их число входят такие популярные форматы, как BMP, GIF, PNG, JPEG, XPM и XBM. Формат XPM применяется в основном для отображения значков. Основное отличие форматов XPM и XBM от других состоит в том, что они содержат исходный код на языке C.

Классы для хранения растровых данных подразделяются в Qt на контекстно-зависимые и контекстно-независимые.

Для контекстно-независимого представления основным является класс `QImage`. Объекты класса `QImage` способны содержать и изменять данные, находящиеся в графическом режиме,

который не поддерживается графической картой. Независимость от контекста позволяет эффективно получать и изменять цвета отдельных пикселов, а также проводить операции загрузки и сохранения файлов. Класс `QImage` содержит ряд методов, позволяющих проводить различные операции с пикселами, такие как: отражение растровых изображений, изменение их размеров, инвертирование и др. При помощи класса `QPainter` на объектах класса `QImage` можно рисовать линии, эллипсы, прямоугольники и т. п.

Основным представителем контекстно-зависимого представления является класс `QPixmap`. Объекты этого класса отображаются гораздо быстрее объектов класса `QImage`. На объектах класса `QPixmap` также можно рисовать при помощи класса `QPainter`. Эти объекты имеют глубину цвета, равную глубине цвета текущего режима графической карты. Унаследованный от `QPixmap` класс `QBitmap` предоставляет возможность для хранения двухцветных растровых изображений.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/19-510/** или с помощью следующего QR-кода (рис. 19.11):

Рис. 19.11. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 20

Работа со шрифтами

> Размышляющий найдет, что черта делает буквы, буквы — слог, а слоги — слово, следовательно, слог был прежде слова, буква прежде слога и черта прежде буквы.
>
> *Карл Эккартсгаузен, «Ключ к таинствам натуры»*

Шрифт имеет очень древнюю историю. Согласно историческим данным, первый буквенный знак появился более восьми тысяч лет назад. На протяжении тысячелетий буквы создавались вручную. Процесс печати был основан в Китае примерно две с половиной тысячи лет назад. И вот — произошла вторая революция, и теперь мы видим шрифты на экранах наших мониторов. Эта глава посвящена масштабируемым шрифтам (гарнитурам). Гарнитура шрифта — это его внешний вид (рисунок шрифта). Масштабируемая гарнитура — это идеальное математическое описание шрифта. Она позволяет отображать его на экране без искажений и выводить на печать в различных размерах. Проведение соответствующих преобразований берут на себя специальные функции растеризации, которые преобразуют математическое представление шрифта для его последующего отображения в растровую матрицу. Эти функции вызываются неявно и не накладывают на разработчика дополнительных временны́х затрат при разработке.

В Qt класс `QFont` является основным для работы со шрифтом. Объект этого класса задается целым рядом параметров:

- семейство шрифта;
- размер;
- толщина — нормальное или полужирное начертание;
- отображаемые знаки;
- стиль — нормальный или наклонный.

При передаче объекта класса `QFont` в метод `QWidget::setFont()` в виджете устанавливается шрифт, который будет использоваться при его отображении. Если требуется установить один шрифт для всего приложения, то объект класса `QFont` нужно передать в статический метод `QApplication::setFont()`.

Qt содержит дополнительные классы для работы со шрифтами: `QFontDataBase`, `QFontInfo` и `QFontMetrics`:

- класс `QFontDatabase` предоставляет информацию обо всех установленных в системе шрифтах. Для их получения вызывается метод `families()`, который возвращает список шрифтов в объекте класса `QStringList`. Класс `QFontDatabase` содержит метод `styleString()` для получения стиля шрифта от класса `QFontInfo`;

- класс `QFontInfo` служит для получения информации о конкретном шрифте. С помощью метода `family()` можно узнать семейство шрифта. Методы `italic()` и `bold()` возвращают значения булевого типа, информирующие о стиле (наклонности и жирности) шрифта;
- класс `QFontMetrics` предоставляет информацию о характеристиках шрифта, показанных на рис. 20.1.

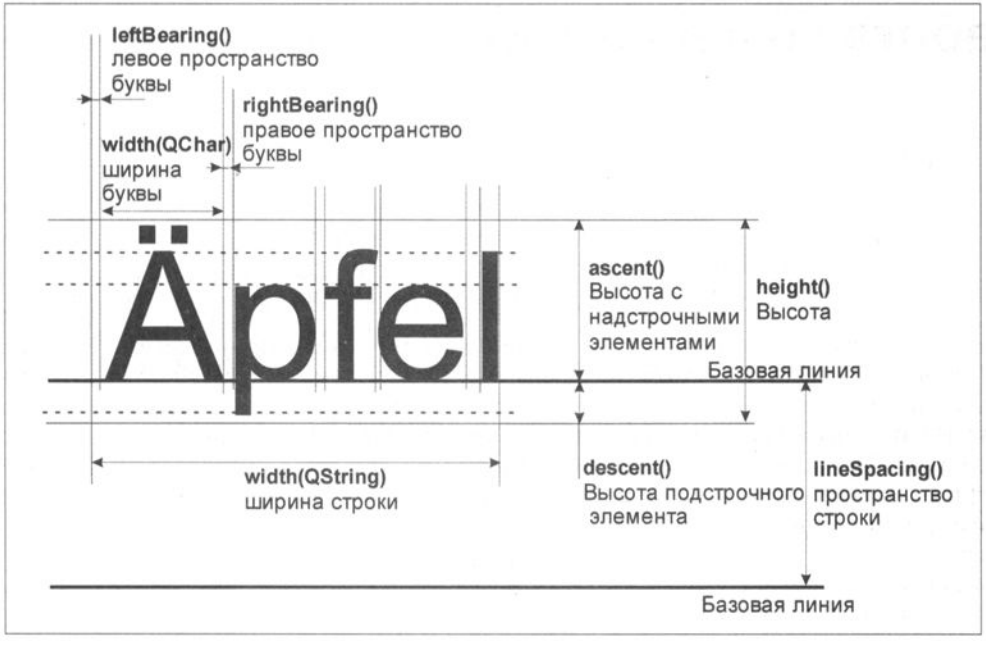


Рис. 20.1. Характеристики шрифта

Передавая в конструктор класса `QFontMetrics` объект класса `QFont`, можно получать его характеристики. Методы `leftBearing()` и `rightBearing()` возвращают в пикселах левое и правое пространство буквы соответственно. Метод `lineSpacing()` возвращает расстояние между базовыми линиями. Передав в метод `width(const QString&, int len)` строку и количество символов, узнают его ширину, — если количество символов не передано, то берется вся длина строки. Чтобы узнать размер всей строки — ее нужно передать в метод `width()`. Высота возвращается методом `height()`. Например:

```
QFontMetrics fm(QFont("Courier", 18, QFont::Bold));
QString      str = "String";
qDebug() << "Widht:"  << fm.width(str)
         << "Height:" << fm.height();
```

Для получения высоты надстрочного и подстрочного элементов шрифта необходимо вызвать методы `ascent()` и `descent()` соответственно. Высота надстрочного элемента — это максимальная высота символа над базовой линией шрифта (включая диакритические знаки), а высота подстрочного элемента — это максимальное значение, на которое символ может уходить ниже базовой линии шрифта.

Вызвав метод `boundingRect()` и передав в него строку, можно получить объект класса `QRect`, соответствующий прямоугольной области, необходимой для отображения текста строки.

Этот метод удобно использовать для определения геометрии текста до начала его отображения.

Отображение строки

В объектах класса `QPainter` методом `QPainter::setFont()` устанавливаются объекты класса `QFont`. В классе `QPainter` имеются несколько различных вариантов метода `drawText()` для отображения текста установленным шрифтом, наиболее часто используются следующие два:

- вариант `drawText(int x, int y, const QString& str)` — отображает текст `str`. Координату левого края текста задает параметр `x`, а параметр `y` указывает координату базовой линии;
- вариант `drawText(const QPoint& pt, const QString& str, int nLen = -1)` — отличается от предыдущего варианта тем, что в первом параметре вместо `x` и `y` передается объект точки `QPoint`. Параметр `nLen` задает количество символов, которые должны быть отображены. По умолчанию параметр равен −1 и означает, что должны отображаться все символы.

В листинге 20.1 приведена реализация вывода на экране строки текста **Draw Text** (рис. 20.2).

Листинг 20.1. Вывод на экране строки текста методами `setFont()` и `drawText()`

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    painter.setFont(QFont("Times", 25, QFont::Normal));
    painter.drawText(10, 40, "Draw Text");
}
...
```

Для более тонкой настройки отображаемого текста класс `QPainter` предоставляет другие варианты метода `drawText()`: `drawText(const QRect& r, int flags, const QString& str)` и отображает текст `str` в заданной параметром `r` прямоугольной области. С помощью параметра `flags` можно повлиять на размещение и отображение текста. Значение этого параметра получается комбинацией значений, указанных в табл. 7.1 и 20.1, с помощью логической операции `|` (ИЛИ).

В листинге 20.2 строка текста выводится по центру (рис. 20.3). Выводимая строка не помещается полностью в прямоугольной области, задаваемой переменной `r`, поэтому с помощью

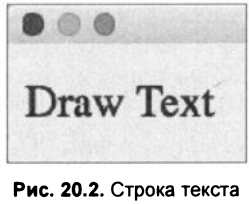


Рис. 20.2. Строка текста



Рис. 20.3. Вывод строки по центру

Таблица 20.1. *Перечисления языка Си* `TextFlag` *пространства имен Qt*

Константа	Значение	Описание
`TextSingleLine`	0x0100	Игнорирует знаки новой строки (знак `\n`)
`TextDontClip`	0x0200	Гарантирует, что в том случае, если текст будет выступать за пределы, он не будет обрезан
`TextExpandTabs`	0x0400	Замещает знаки табуляции `\t` равносильным пространством
`TextShowMnemonic`	0x0800	Знак `&` не будет отображаться, а следующий за ним символ будет подчеркнут и получит клавишу быстрого доступа
`TextWordWrap`	0x1000	Если строка выходит за пределы заданного прямоугольника, она будет перенесена

флага `TextWordWrap` осуществляется переход на новую строку. Метод `drawRect()` вызывается для отображения границ прямоугольной области.

Листинг 20.2. Вывод строки по центру методами `setFont()`, `drawRect()` и `drawText()`

```
...
void MyWidget::paintEvent(QPaintEvent*)
{
    QPainter painter(this);
    QRect r(0, 0, 120, 200);
    painter.setFont(QFont("Times", 25, QFont::Normal));
    painter.drawRect(r);
    painter.drawText(r, Qt::AlignCenter | Qt::TextWordWrap, "Draw Text");
}
...
```



Рис. 20.4. Текст, заполненный градиентом

Текст можно залить градиентом (см. *главу 18*), как это показано на рис. 20.4. Для этого надо создать объект градиента (в нашем случае это будет линейный градиент) и при помощи метода `QGradient::setColorAt()` осуществить переход цвета, например, от красного к зеленому и от зеленого к синему:

```
QLinearGradient gradient(0, 0, 500, 0);
gradient.setColorAt(0, Qt::red);
gradient.setColorAt(0.5, Qt::green);
gradient.setColorAt(1, Qt::blue);
```

Полученный градиент необходимо передать в конструктор для создания объекта пера `QPen`. Затем установить в объекте `QPainter` перо вызовом метода `setPen()` и шрифта методом `setFont()`, после чего можно отобразить текст на экране с помощью метода `drawText()`:

```
QPainter painter(this);
painter.setPen(QPen(gradient, 0));
painter.setFont(QFont("Times", 50, QFont::Normal));
painter.drawText(60, 60, "Gradient Text");
```

Бывает, что область, предназначенная для показа текста, не может отобразить весь текст целиком. Подобные ситуации часто наблюдаются, например, при отображении путей каталогов. В таком случае можно сделать разрыв в тексте, заполнив его точками, и тем самым показать, что отображенный текст не является полным. Реализуется эта возможность на базе метода `elidedText()` класса `QFontMetrics` (листинг 20.3). На рис. 20.5 показано окно с текстом, при изменении размеров которого, в случае невозможности размещения текста целиком, в середине текста будет показан разрыв, заполненный точками.

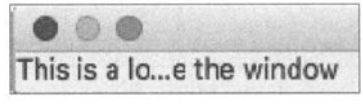


Рис. 20.5. Текст с разрывом

Листинг 20.3. Усеченное отображение строк с показом разрыва

```
#include <QtWidgets>

// ============================================================================
class ElidedText : public QWidget {
protected:
    virtual void paintEvent(QPaintEvent*)
    {
        QString  str = "This is a long text. Please, resize the window";
        QString  strElided =
            fontMetrics().elidedText(str, Qt::ElideMiddle, width());
        QPainter painter(this);
        painter.drawText(rect(), strElided);
    }

public:
    ElidedText(QWidget* pwgt = 0) : QWidget(pwgt)
    {
    }
};

// ----------------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    ElidedText   et;
    et.resize(200, 20);
    et.show();

    return app.exec();
}
```

В листинге 20.3 в методе `paintEvent()` создаем строку с длинным текстом (объект `str`). Эту строку вместе с двумя другими параметрами передаем в метод `elideText()`, то есть передаем режим показа разрыва в середине `Qt::ElideMiddle` и текущую ширину виджета (другие возможные значения приведены в табл. 20.2). Этот метод возвращает текст новой строки, который мы отображаем при помощи метода `drawText()`.

Таблица 20.2. *Перечисления языка Си* `TextElideMode` *пространства имен Qt*

Константа	Значение	Описание
`ElideLeft`	`0x0000`	Разрыв должен быть показан в тексте в начале
`ElideRight`	`0x0001`	Разрыв должен быть показан в тексте справа
`ElideMiddle`	`0x0002`	Разрыв должен быть показан в тексте в середине
`ElideNone`	`0x0003`	Разрыв появляться в тексте не должен

Резюме

Шрифты используются для вывода текста на контексте рисования. Они задаются рядом параметров, а именно высотой, шириной и названием (семейством).

Класс `QFont` является основным классом шрифта.

При помощи класса `QFontInfo` можно получить информацию о семействе шрифта.

Класс `QFontDataBase` предоставляет информацию о шрифтах, установленных в системе.

Класс `QFontMetrics` дает информацию о целом ряде характеристик шрифта, например высоте, ширине букв и др.

Класс `QPainter` содержит метод для установки шрифта, а также ряд методов, позволяющих отображать текст различным образом. Текст, кроме того, может быть заполнен градиентом.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/20-510/** или с помощью следующего QR-кода (рис. 20.6):



Рис. 20.6. QR-код для доступа на страницу комментариев к этой главе

ГЛАВА 21

Графическое представление

А вот и тропинка. Она приведет меня прямо наверх...
Но как она кружит! Прямо штопор, а не тропинка.
Льюис Кэрролл, «Алиса в Зазеркалье»

При программировании графики мы зачастую имеем дело с целой массой объектов, двигающихся и перекрывающих друг друга. Все они должны быть отображены в режиме реального времени без побочных эффектов. И это представляет собой нелегкую задачу для разработчика. Графическое представление — это инструмент для управления и взаимодействия с большим количеством элементов двумерных изображений, включая их визуальное увеличение/уменьшение и поворот. Кроме того, оно берет на себя также и обнаружение столкновений (collision detection). Классы графического представления, подобно классам `QTableWidget`, `QTreeWidget` и `QListWidget` (см. *главу 11*), являют собой элементный подход, опирающийся на концепцию «модель-представление» (Model-View) (см. *главу 12*).

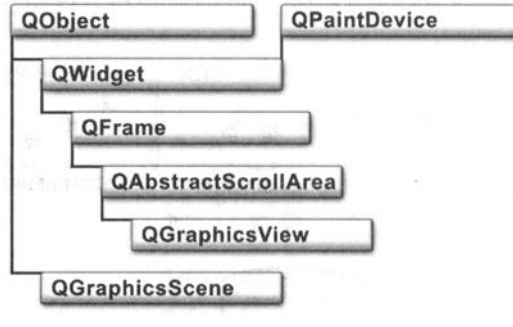


Рис. 21.1. Иерархия классов графического представления

Графическое представление базируется на трех понятиях: *сцена* — `QGraphicsScene` и *представление* — `QGraphicsView` (рис. 21.1), а также *элемент* — `QGraphicsItem` (рис. 21.2).

Взаимодействие классов вкратце можно описать следующим образом: класс `QGraphicsScene` является моделью для графических элементов, которые реализуются унаследованными от класса `QGraphicsItem`, а класс `QGraphicsView` — это представление, которое унаследовано от класса `QAbstractScrollArea`. Представления служат для показа элементов модели (объекта класса `QGraphicsScene`), и с одной моделью может быть связано сразу несколько виджетов представления (рис. 21.3).

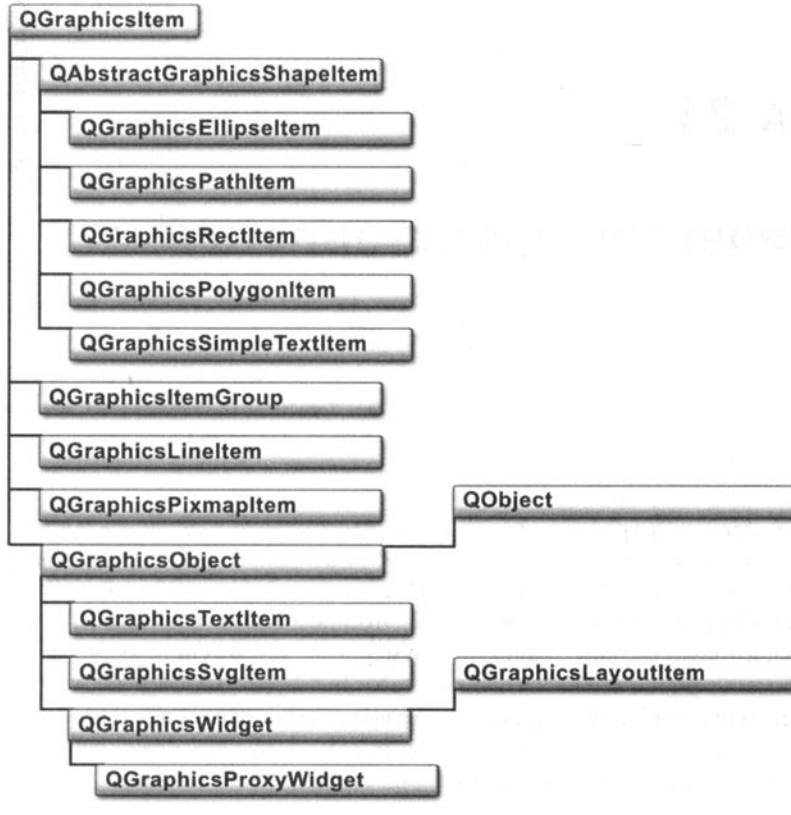


Рис. 21.2. Иерархия классов элементов графического представления



Рис. 21.3. Взаимодействие представлений со сценой

Сцена

Класс сцены (`QGraphicsScene`) является классом для управления элементами без их отображения. Как только какой-либо элемент сцены подвергся изменениям, объект класса `QGraphicsScene` запоминает его состояние. Перед тем как сообщить о необходимости пере-

рисовки, вся область разделяется на подобласти и осуществляется анализ того, в какой из них были выполнены изменения. Эта операция реализуется очень быстро и способна выполняться в режиме реального времени для миллионов элементов. Отправкой сигнала `changed()` объект сцены сообщает представлениям о необходимости отображения измененного содержимого, после чего представления отображают найденные области.

Объект сцены (`QGraphicsScene`) представляет собой контейнер, содержащий в себе объекты, которые созданы от классов, наследующих `QGraphicsItem` (см. рис. 21.2). Эти объекты являются данными без графического представления. Элементы добавляются в сцену при помощи метода `QGraphicsScene::addItem()`.

Для добавления элементов в сцену можно воспользоваться методами `addEllipse()`, `addLine()`, `addPath()`, `addPixmap()`, `addPolygon()`, `addRect()` и `addText()`, которые неявно создадут соответствующий элемент, добавят его и вернут его указатель.

Для того чтобы получить указатели на все элементы сцены, можно воспользоваться методом `QGraphicScene::items()`. Если вас интересует один элемент с определенными координатами, то получить указатель на него можно с помощью метода `QGraphicScene::itemAt()`, который возвращает самый верхний элемент, находящийся на заданных координатах.

Для создания объекта сцены в конструктор класса `QGraphicsScene` нужно передать объект прямоугольной области с вещественными параметрами `QRectF`. Координаты области могут содержать и отрицательные значения.

Представление

Класс `QGraphicsView` является виджетом, предназначенным для визуализации содержимого сцены (`QGraphicsScene`). Подобное отделение данных от их графического представления позволяет отображать одну и ту же сцену (`QGraphicsScene`) в различных виджетах `QGraphicsView`. Благодаря тому, что класс `QGraphicsView` унаследован от класса `QAbstractScrollArea`, при отображении появляются полосы прокрутки в случаях, когда пространства для показа сцены недостаточно. Все элементы, хранящиеся в `QGraphicsScene`, автоматически отображаются в окне представления. То есть в окне представления мы не рисуем изображение, а просто отображаем элементы модели и управляем ими. Виджеты класса `QGraphicsView` связаны с моделью и получают уведомления всякий раз, когда им необходимо обновить свое изображение. В связи с этим отпадает необходимость заботиться о перерисовке изображения содержимого представления, так как она происходит автоматически.

Для того чтобы отцентрировать представление относительно определенной точки, можно вызвать метод `QGraphicsView::centerOn()`, передав в него координаты этой точки.

Более того, одно из самых примечательных свойств, которое получил класс `QGraphicsView` в наследство от класса `QAbstractScrollArea`, — это способность в качестве области просмотра (viewport) использовать любой виджет. Она позволяет заменять `QWidget` на `QGLWidget` (см. *главу 23*) и дает возможность выбора наиболее подходящего варианта для визуализации в процессе работы программы. Установка виджета области просмотра осуществляется вызовом метода `QAbstractScrollArea::setViewPort()`. Например, для того чтобы изменить область просмотра на виджет, поддерживающий OpenGL, нужно сделать следующее:

```
pView->setViewport(new QGLWidget);
```

Использование матрицы трансформации, устанавливаемой методом `setMatrix()`, позволяет увеличивать, уменьшать и поворачивать отображаемую в представлении сцену.

Элемент

Класс `QGraphicsItem` является основным для элементов (см. рис. 21.2). Этот класс предоставляет поддержку для событий мыши и клавиатуры, перетаскивания (drag & drop), группировки элементов и *определения столкновений* (collision detection). Также возможны следующие операции над элементами:

- установка местоположения методом `setPos()`;
- скрытие и показ: методы `hide()` и `show()`;
- установка доступного/недоступного состояния с помощью метода `setEnable()`;
- трансформация: производится при помощи метода `setTransformation()`, в который передается объект класса, ответственного за трансформации (`QTransform`). Класс `QTransform` будет использован в некоторых примерах этой главы;
- перерисовка — `paint()`.

Класс `QGraphicsItem` не позволяет создавать объекты, так как является абстрактным. Для создания объектов нужно воспользоваться готовыми унаследованными от него классами (см. рис. 21.2) или создать свой класс, унаследовав его от `QGraphicsItem` и реализовав в нем методы `paint()` и `boundingRect()`.

Классы, унаследованные от класса `QAbstractGraphicsShapeItem`, представляют собой различные геометрические фигуры: эллипс (`QGraphicsEllipseItem`), многоугольник (`QGraphicsPolygonItem`), прямоугольник (`QGraphicsRectItem`) и текст (`QgraphicsSimpleTextItem`). Если нужно создать класс для поддержки какой-либо другой формы, то, в большинстве случаев, лучше унаследовать именно класс `QAbstractGraphicsShapeItem`.

Наверняка вы уже заметили, что в схеме на рис. 21.2 присутствуют два класса для текстовых элементов. Это `QGraphicsSimpleTextItem` и `QGraphicsTextItem`. Класс `QGraphicsSimpleTextItem` предназначен для быстрого отображения простого текста при малом расходе памяти. Если же вам потребуется отобразить форматированный текст, то для этого нужно воспользоваться классом `QGraphicsTextItem`, который обладает массой возможностей, позволяющих управлять текстовым документом вплоть до его редактирования.

К готовым классам элементов также относятся линии (`GraphicsLineItem`), растровые изображения (`QGraphicsPixmapItem`) и векторная графика (`QGraphicsSvgItem`). При помощи класса `QGraphicsItemGroup` можно объединять элементы в группы.

В приведенном далее листинге 21.1 организуется размещение четырех элементов на сцене (рис. 21.4). Здесь создаются объект приложения `app` и объекты классов `QGraphicsScene` и `QGraphicsView`. Виджет `view` при создании получает адрес объекта `scene`, но в качестве альтернативы можно воспользоваться методом `QGraphicsView::setScene()`.

Объекту элемента (указатель `pRectItem`) при создании передается ссылка на объект класса `QGraphicsScene`, что приводит к добавлению элемента в сцену. Аналогичный результат дал бы вызов метода `QGraphicsScene::addItem()`. Вызов метода `setPen()` устанавливает черный цвет пера для контурной линии элемента. Кисть, предназначенная для заливки фона элемента, получает зеленый цвет с помощью метода `setBrush()`. Метод `setRect()` задает расположение и размеры прямоугольной области. Вызовами методов `QGraphicsScene::`

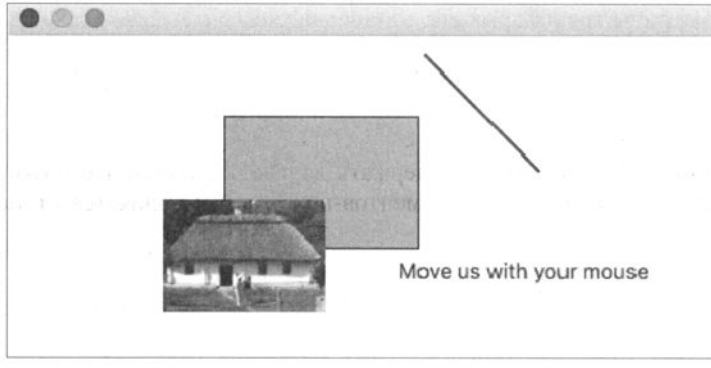


Рис. 21.4. Отображение элементов, положение которых можно изменять при помощи мыши

`addPixmap()`, `QGraphicsScene::addText()` и `QGraphicsScene::addLine()` в сцену добавляются элементы растрового изображения, текста и линии.

Для того чтобы все добавленные элементы можно было перемещать мышью, эта возможность активизируется передачей в метод `setFlags()` значения `QGraphicsItem::ItemIsMoveable`.

В завершение, вызовом метода `show()` сцена отображается в представлении, и ее элементы становятся видимыми на экране.

Листинг 21.1. Размещение на сцене четырех элементов (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication   app(argc, argv);
    QGraphicsScene scene(QRectF(-100, -100, 300, 300));
    QGraphicsView  view(&scene);

    QGraphicsRectItem* pRectItem =
        scene.addRect(QRectF(-30, -30, 120, 80),
                      QPen(Qt::black),
                      QBrush(Qt::green)
                     );
    pRectItem->setFlags(QGraphicsItem::ItemIsMovable);

    QGraphicsPixmapItem* pPixmapItem =
        scene.addPixmap(QPixmap(":/haus.jpg"));
    pPixmapItem->setFlags(QGraphicsItem::ItemIsMovable);

    QGraphicsTextItem* pTextItem =
        scene.addText("Move us with your mouse");
    pTextItem->setFlags(QGraphicsItem::ItemIsMovable);

    QGraphicsLineItem* pLineItem =
        scene.addLine(QLineF(-10, -10, -80, -80), QPen(Qt::red, 2));
    pLineItem->setFlags(QGraphicsItem::ItemIsMovable);
```

```
    view.show();

    return app.exec();
}
```

Элементы, подобно виджетам, могут содержать другие элементы, что позволяет осуществлять их группировку. Расположение элементов-потомков выполняется относительно предка, например:

```
QGraphicsLineItem* pLineItem =
    scene.addLine(QLineF(-10, -10, -80, -80), QPen(Qt::red, 2));
QGraphicsTextItem* pTextItem = scene.addText("Child");
pTextItem->setParentItem(pLineItem);
```

Каждый элемент имеет свою собственную локальную систему координат, которая может использоваться для выполнения геометрических преобразований, то есть, проще говоря, может быть подвержена трансформации, например:

```
QGraphicsTextItem* pTextItem = scene.addText("Shear");
pTextItem->setTransform(QTransform().shear(-0.5, 0.0), true);
```

Если предок подвергнется трансформации, то вместе с ним будут трансформированы и его потомки.

Класс `QGraphicsItem` предоставляет возможность определения столкновений элементов. Это задача выполняется с помощью методов `QGraphics::shape()` и `QGraphicsItem::collidesWith()`. Если вы создаете свой класс элемента, то для определения столкновений необходимо перезаписать метод `QGraphicsItem::shape()`. Этот метод должен возвращать форму элемента в локальных координатах. Благодаря этому класс `QGraphicsItem` будет самостоятельно распознавать столкновения.

События

События модели могут передаваться отдельным элементам, находящимся в модели. Класс `QGraphicsView` является наследником класса `QAbstractScrollArea`, поэтому можно переопределить и воспользоваться любым из методов обработки событий этого класса. Можно, но это не самый удобный подход. Предположим, что пользователь нажал мышью на один из элементов сцены. Для того чтобы выяснить, что это за элемент, нужно получить указатель на объект сцены вызовом метода `QGraphicsView::scene()`, затем определить, какие элементы сцены находятся на координатах указателя мыши, и выбрать из них самый верхний. И если элемент должен как-то отреагировать на это событие, то нужно будет вызвать соответствующий метод. Согласитесь, это не совсем удобно.

Самая интересная возможность при обработке событий — это их обработка из самих элементов. Внутренне это работает следующим образом: представление получает события мыши и клавиатуры, затем оно переводит их в события для сцены, изменяя координаты в соответствии с координатами сцены, и передает событие нужному элементу. Если элемент получает событие мыши, то сцена сама позаботится, чтобы координаты были приведены к локальным координатам элемента. И все это происходит без вашего участия. На рис. 21.5 показана иерархия событий, которые способен получать элемент.

Элементы могут обрабатывать события клавиатуры, мыши, а также события, возникающие при попадании указателя мыши в их область (`QGraphicsHoverEvent`) и при вызове контекстного меню (`QGraphicsContextMenuEvent`).

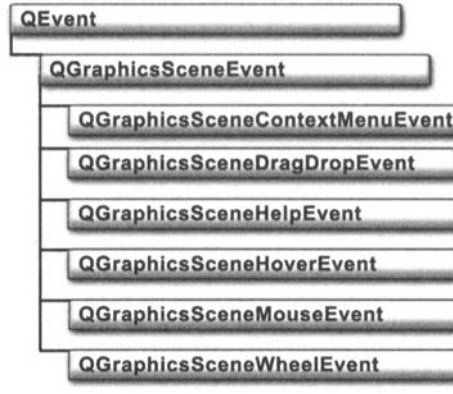


Рис. 21.5. Иерархия классов событий графического представления

Существует возможность создания и обработки событий перетаскивания (drag & drop). Элементы могут как разрешать, так и запрещать поддержку перетаскивания, вызывая метод `setAcceptDrops()`. Для управления принятием сбрасываемых объектов необходимо переопределить методы `dragEnterEvent()`, `dragMoveEvent()`, `dragLeaveEvent()` и `dropEvent()`.

Для того чтобы начать перетаскивание из элемента, надо создать объект класса `QDrag`, передав ему виджет представления, из которого происходит перетаскивание. Элементы могут отображаться в нескольких представлениях, но перетаскивание происходит только из какого-либо одного. Для того чтобы получить указатель виджета этого представления, нужно вызвать метод `QGraphicsSceneEvent::widget()`. Для реализации перетаскивания из элементов надо переопределить методы обработки событий мыши.

Теперь подведем итог изложенного материала и создадим программу (листинги 21.2–21.4), окно которой показано на рис. 21.6. В ней используются:

- собственный класс представления;
- собственный класс элемента;
- обработка событий;
- группировка элементов.

В листинге 21.2 приведен собственный класс представления `MyView`, который наследуется от класса `QGraphicsView`. Класс `MyView` предоставляет слоты для уменьшения, увеличения и поворота, которые мы впоследствии соединим с сигналами соответствующих кнопок (см. листинг 21.4).

Листинг 21.2. Собственный класс представления (файл MyView.h)

```
#pragma once

#include <QGraphicsView>

// ============================================================================
class MyView: public QGraphicsView {
Q_OBJECT
public:
    MyView(QGraphicsScene* pScene, QWidget* pwgt = 0)
        : QGraphicsView(pScene, pwgt)
```

```
    {
    }

public slots:
    void slotZoomIn()
    {
        scale(1.1, 1.1);
    }

    void slotZoomOut()
    {
        scale(1 / 1.1, 1 / 1.1);
    }

    void slotRotateLeft()
    {
        rotate(-5);
    }

    void slotRotateRight()
    {
        rotate(5);
    }
};
```

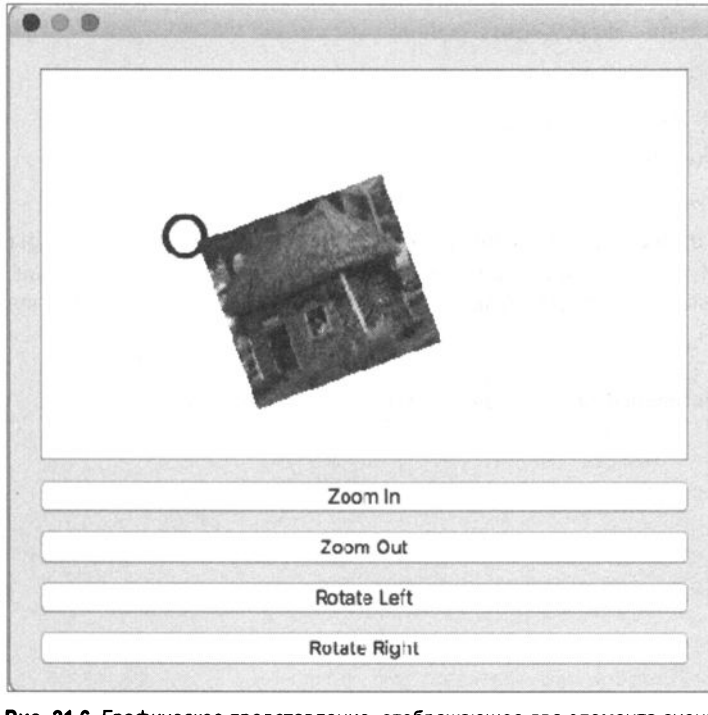


Рис. 21.6. Графическое представление, отображающее два элемента сцены

Класс `SimpleItem`, приведенный в листинге 21.3, является реализацией собственного элемента. Метод `boundingRect()` необходим представлению для определения невидимых элементов и неперекрытых областей, которые должны быть нарисованы. В нашем примере этот метод возвращает прямоугольную область, в которую вписывается окружность, с учетом толщины линии (`penWidth`).

Метод `paint()` отвечает за отображение элемента. В нашем примере это окружность, которая рисуется синим пером толщиной 3 пиксела. Поскольку мы изменяем настройки объекта `QPainter` с помощью метода `setPen()`, то его состояние предварительно сохраняется методом `save()`, а в конце рисования восстанавливается в первоначальное состояние методом `restore()`.

При нажатии на кнопку мыши вызывается метод `mousePressEvent()`, который изменяет указатель мыши на изображение, сигнализирующее о том, что элемент может быть перемещен, а затем передает указатель на объект события в метод `mousePressEvent()` унаследованного класса.

Метод `mouseReleaseEvent()` вызывается сразу после отпускания кнопки мыши. Он восстанавливает исходное изображение курсора мыши и передает объект события дальше на обработку методу `mouseReleseEvent()` унаследованного класса.

Листинг 21.3. Класс `SimpleItem` — реализация собственного элемента (файл main.cpp)

```
class SimpleItem : public QGraphicsItem {
private:
    enum {nPenWidth = 3};

public:
    virtual QRectF boundingRect() const
    {
        QPointF ptPosition(-10 - nPenWidth, -10 - nPenWidth);
        QSizeF  size(20 + nPenWidth * 2, 20 + nPenWidth * 2);
        return QRectF(ptPosition, size);
    }

    virtual void paint(QPainter* ppainter,
                       const QStyleOptionGraphicsItem*,
                       QWidget*
                      )
    {
        ppainter->save();
        ppainter->setPen(QPen(Qt::blue, nPenWidth));
        ppainter->drawEllipse(-10, -10, 20, 20);
        ppainter->restore();
    }

    virtual void mousePressEvent(QGraphicsSceneMouseEvent* pe)
    {
        QApplication::setOverrideCursor(Qt::PointingHandCursor);
        QGraphicsItem::mousePressEvent(pe);
    }
```

```
    virtual void mouseReleaseEvent(QGraphicsSceneMouseEvent* pe)
    {
        QApplication::restoreOverrideCursor();
        QGraphicsItem::mouseReleaseEvent(pe);
    }
};
```

В функции `main()`, приведенной в листинге 21.4, создаются объекты сцены (`scene`), представления (указатель `pView`), элементы (указатели `pSimpleItem` и `pPixmapItem`) и кнопки, предназначенные для поворота (указатели `pcmdRotateLeft` и `pcmdRotateRight`), увеличения и уменьшения (указатели `pcmdZoomIn` и `pcmdZoomOut` соответственно).

Вызов метода `setRenderHint()` из объекта представления устанавливает в нем режим сглаживания, что необходимо для более мягкого отображения контуров элементов (на растровые изображения это не распространяется).

Элемент определенного нами класса `SimpleItem` добавляется в сцену при помощи метода `QGraphicsScene::addItem()`, а метод `setPos()` устанавливает его положение на сцене. Мы даем возможность изменения местоположения элемента на сцене, для чего в метод `setFlags()` передается значение `QGraphicsItem::ItemIsMovable`.

Созданному элементу растрового изображения (указатель `pPixmapItem`) с помощью метода `setParent()` присваивается предок, которым является элемент, произведенный от созданного нами класса `SimpleItem`. Вызов метода `setFlags()` разрешает перемещение элемента при помощи мыши. Это мы сделали умышленно, для того, чтобы продемонстрировать взаимосвязь группировки элементов отношением «предок-потомок». Итак, обратите внимание, что когда мы перемещаем растровое изображение, то перемещается только оно, но если мы попытаемся переместить элемент окружности, то растровое изображение будет перемещаться вместе с ним, так как оно является его потомком.

В завершение кнопки соединяются с соответствующими слотами созданного нами класса `MyView`, а элементы при помощи вертикальной компоновки `QVBoxLayout` размещаются на поверхности виджета `wgt`.

Листинг 21.4. Функция `main()`: создание объектов сцены и пр. (файл main.cpp)

```
int main(int argc, char** argv)
{
    QApplication   app(argc, argv);
    QWidget        wgt;
    QGraphicsScene scene(QRectF(-100, -100, 640, 480));

    MyView*      pView           = new MyView(&scene);
    QPushButton* pcmdZoomIn      = new QPushButton("&Zoom In");
    QPushButton* pcmdZoomOut     = new QPushButton("Z&oom Out");
    QPushButton* pcmdRotateLeft  = new QPushButton("&Rotate Left");
    QPushButton* pcmdRotateRight = new QPushButton("Ro&tate Right");

    pView->setRenderHint(QPainter::Antialiasing, true);

    SimpleItem*  pSimpleItem = new SimpleItem;
    scene.addItem(pSimpleItem);
```

```
    pSimpleItem->setPos(0, 0);
    pSimpleItem->setFlags(QGraphicsItem::ItemIsMovable);

    QGraphicsPixmapItem* pPixmapItem =
        scene.addPixmap(QPixmap(":/haus2.jpg"));
    pPixmapItem->setParentItem(pSimpleItem);
    pPixmapItem->setFlags(QGraphicsItem::ItemIsMovable);

    QObject::connect(pcmdZoomIn, SIGNAL(clicked()),
                     pView,      SLOT(slotZoomIn())
                    );
    QObject::connect(pcmdZoomOut, SIGNAL(clicked()),
                     pView,       SLOT(slotZoomOut())
                    );
    QObject::connect(pcmdRotateLeft, SIGNAL(clicked()),
                     pView,          SLOT(slotRotateLeft())
                    );
    QObject::connect(pcmdRotateRight, SIGNAL(clicked()),
                     pView,           SLOT(slotRotateRight())
                    );

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(pView);
    pvbxLayout->addWidget(pcmdZoomIn);
    pvbxLayout->addWidget(pcmdZoomOut);
    pvbxLayout->addWidget(pcmdRotateLeft);
    pvbxLayout->addWidget(pcmdRotateRight);
    wgt.setLayout(pvbxLayout);

    wgt.show();

    return app.exec();
}
```

Виджеты в графическом представлении

Класс `QGraphicsScene` предоставляет возможность размещения не только графических объектов, но и виджетов, причем, благодаря механизму событий, помещенные в качестве элементов виджеты не теряют своей функциональности и реагируют на действия пользователя, как обычные виджеты. Однако самое интересное свойство состоит в том, что, как только виджет становится элементом сцены, с ним можно выполнять те же геометрические преобразования, что и с обычными графическими элементами, а также использовать механизм для определения столкновений. Это открывает большой простор для фантазии — ведь вы сможете теперь реализовывать пользовательские интерфейсы очень необычного вида. Продемонстрируем эту возможность на конкретном примере и разместим в сцене три виджета (рис. 21.7).

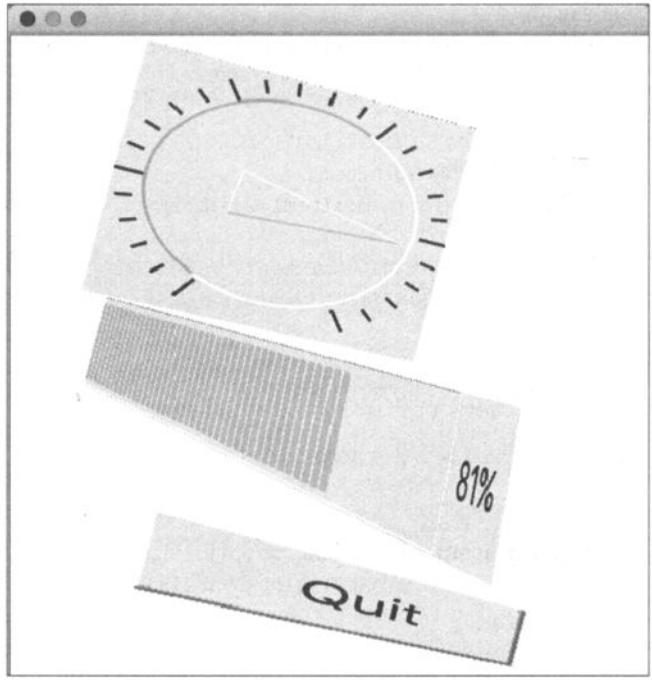


Рис. 21.7. Виджеты в графическом представлении

В начале программы (листинг 21.5) мы создаем объект сцены (`scene`) и виджет представления (`view`). Затем создаем виджет кнопки **Quit** и добавляем ее при помощи метода сцены `addWidget()`, который возвращает объект элемента сцены `QGraphicsProxyWidget`. С этим элементом можно осуществлять геометрические преобразования, для чего мы используем специальный класс `QTransform`. Этот класс предоставляет методы сдвига `translate()`, поворота `rotate()` и масштабирования `scale()`. Обратите внимание на вызов метода `rotate()` — в нем, помимо угла поворота, мы также указываем и ось, вокруг которой нужно осуществить поворот, — в примере это ось *Y* (`Qt::YAxis`). Геометрическое преобразование применяется к элементу (указатель `pproxyWidget`) вызовом метода `setTransform()`. Аналогично мы поступаем с виджетами `QDial` и `QProgressBar`. Затем мы выполняем сигнально-слотовые соединения виджетов, — так, например, кнопку **Quit** мы связываем со слотом приложения `quit()`, а сигнал `valueChanged()` виджета `QDial` — со слотом `setValue()` виджета `QProgressBar`. В завершение осуществляем трансформацию сцены и поворачиваем ее на 15 градусов по оси *Z*. Обратите внимание на тот факт, что виджеты, расположенные в графическом представлении, полностью сохраняют свои функциональные возможности.

Листинг 21.5. Виджеты в графическом представлении (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication   app(argc, argv);
    QGraphicsScene scene(QRectF(0, 0, 400, 400));
    QGraphicsView  view(&scene);
```

```
    QPushButton cmd("Quit");
    QGraphicsProxyWidget* pproxyWidget = scene.addWidget(&cmd);
    QTransform            transform    = pproxyWidget->transform();

    transform.translate(100, 350);
    transform.rotate(-45, Qt::YAxis);
    transform.scale(8, 2);
    pproxyWidget->setTransform(transform);
    QObject::connect(&cmd, SIGNAL(clicked()), &app, SLOT(quit()));

    QDial dia;
    dia.setNotchesVisible(true);
    pproxyWidget = scene.addWidget(&dia);
    transform    = pproxyWidget->transform();

    transform.scale(4, 2);
    transform.rotate(-45, Qt::YAxis);
    pproxyWidget->setTransform(transform);

    QProgressBar prb;
    prb.setFixedSize(500, 40);
    pproxyWidget = scene.addWidget(&prb);
    transform    = pproxyWidget->transform();

    transform.translate(20, 200);
    transform.scale(2, 2);
    transform.rotate(80, Qt::YAxis);
    transform.rotate(30, Qt::XAxis);
    pproxyWidget->setTransform(transform);

    QObject::connect(&dia, SIGNAL(valueChanged(int)),
                     &prb, SLOT(setValue(int))
                    );

    view.rotate(15);
    view.resize(500, 500);
    view.show();

    app.setStyle("Windows");

    return app.exec();
}
```

Резюме

Применение технологии графического представления идеально подходит для приложений, в которых должно содержаться большое количество графических элементов, которыми управляет пользователь. Классы `QGraphicsScene`, `QGraphicsView`, а также `QGraphicsItem` и унаследованные от него, представляют собой очень мощное средство для работы с двумер-

ной графикой. Взаимодействие этих классов друг с другом базируется на шаблоне «модель-представление». Это позволяет показывать один и тот же объект класса `QGraphicsScene` в нескольких разных виджетах представления `QGraphicsView`.

Класс `QGraphicsScene` можно представить как плоскость, на которой можно размещать элементы. Хотя того же эффекта можно добиться и с виджетами — так как они тоже обладают способностью отображать своих потомков, работа с графическим представлением позволяет более эффективно использовать процессор и память компьютера.

Классы элементов, унаследованные от класса `QGraphicsItem`, служат для представления различных геометрических форм, а также текста, растровых, векторных и анимированных изображений. Элементы способны получать и обрабатывать события. Для создания своих собственных классов элементов нужно унаследовать этот класс. Элементы (`QGraphicsItem`) должны помещаться в объект класса `QGraphicsScene`.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/21-510/** или с помощью следующего QR-кода (рис. 21.8):



Рис. 21.8. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 22

Анимация

Зри в корень.
Козьма Прутков

Понятие анимации пришло к нам из ранних лет кино. Само слово переводится с латыни как «оживление неподвижных предметов». Принцип анимации — тот же самый, что используется в играх с «движущимися» картинками. Если быстро отображать одно изображение за другим, то создается иллюзия движения. Обычные мультипликационные фильмы тоже состоят из целой серии рисованных картинок, в которых позиции объектов последовательно и незначительно изменяются относительно друг друга.

Класс *QMovie*

Для создания анимации можно воспользоваться классом `QPixmap`, показывая изображения одно за другим. Но лучше обратиться к уже готовому классу `QMovie`, который выполнит эту работу за вас. Объекты класса хранят в себе анимацию и могут возвращать отдельные изображения в объектах класса `QPixmap` или `QImage`. Этот класс поддерживает форматы файлов MNG и GIF. Если же вы ищете возможность воспроизведения видеофайлов, то, скорее всего, *глава 27*, описывающая модуль `QtMultimedia`, и есть то, что вам нужно.

В конструктор класса `QMovie` передается имя анимационного файла. Проигрывание анимации начинается сразу после создания объекта. Также в этом классе определены конструктор копирования и оператор присваивания.

На проигрывание анимации можно влиять следующими слотами: `setPaused(bool)`, `setSpeed()`, `stop()` и `start()`.

Передача в метод `setPaused()` значения `true` приостанавливает проигрывание анимации, а значения `false` — возобновляет проигрывание. В обоих случаях осуществляется пересылка сигнала `stateChanged()` со статусом состояния проигрывания. Вызов метода `QMovie::start()` запускает воспроизведение анимации, а `stop()` его останавливает.

Информацию о статусе проигрывания можно получить при помощи метода `state()`, который возвращает одно из трех значений:

- `QMovie::Paused` — сообщает, что проигрывание было приостановлено;
- `QMovie::Running` — означает, что анимация находится в состоянии проигрывания;
- `QMovie::NoRunning` — сигнализирует о завершении проигрывания анимации.

Для того чтобы узнать количество кадров анимационного файла, нужно вызвать метод `frameCount()`. Если необходимо получить растровое изображение актуального кадра, то надо вызвать метод `currentPixmap()` или `currentImage()`, которые возвращают ссылки на объекты `QPixmap` или `QImage`.

Самый простой способ показа анимации — это использование класса `QLabel`. Класс `QLabel` содержит метод `setMovie()`, с помощью которого можно устанавливать объекты анимации. Пример, показанный в листинге 22.1, иллюстрирует эту возможность (рис. 22.1).

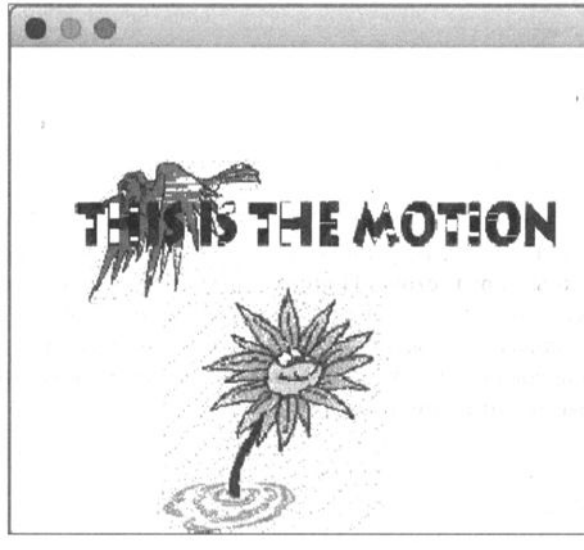


Рис. 22.1. Показ анимации

Как видно из листинга 22.1, в программе создаются всего 3 объекта: приложение (`app`), надпись (`lbl`) и анимационный объект (`mov`), причем последний инициализируется файлом motion.gif. Вызовом метода `setMovie()` анимационный объект устанавливается в виджете надписи. После показа надписи (метод `show()`) вызовом метода `start()` и запускается анимация. Размеры окна мы будем приводить в соответствие с размером кадра анимации, который получаем при помощи метода `frameRect()` и метода `size()`. Метод `resize()` устанавливает размеры окна, то есть виджета надписи.

Листинг 22.1. Создание анимации (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QLabel       lbl;
    QMovie       mov(":/motion.gif");

    mov.start();

    lbl.setMovie(&mov);
```

```
    lbl.resize(mov.frameRect().size());
    lbl.show();

    return app.exec();
}
```

SVG-графика

SVG — это формат *масштабируемой векторной графики* (Scalable Vector Graphics). Он был рекомендован в 2001 году Консорциумом Всемирной паутины W3C (World Wide Web Consortium). Этот формат описывает двумерную векторную графику и анимацию в формате XML — следовательно, содержимое файлов можно изменять в обычном текстовом редакторе. К настоящему моменту формат SVG получил большое распространение и поддерживается почти всеми веб-браузерами.

Qt для поддержки этого формата предоставляет отдельный модуль `QtSvg`. А это значит, что в проектных файлах (файлах с расширением pro) нужно не забывать прикреплять этот модуль, для чего достаточно просто добавить строку `QT += svg`. Класс для показа SVG-файлов называется `QSvgWidget`. Загрузить SVG-файл можно либо передав его в конструктор `QSvgWidget`, либо воспользовавшись его методом `load()`. Метод `load()` примечателен тем, что в него можно передавать не только путь к файлу, но и объекты класса `QByteArray`. Само изображение создается вспомогательным классом `QSvgRenderer`, который используется неявно, и если вы не отображаете анимацию самостоятельно, то о его существовании вы, скорее всего, даже и не вспомните. Класс `QSvgRenderer` можно также использовать для помещения созданных им изображений в объекты `QImage` и `QGlWidget`.

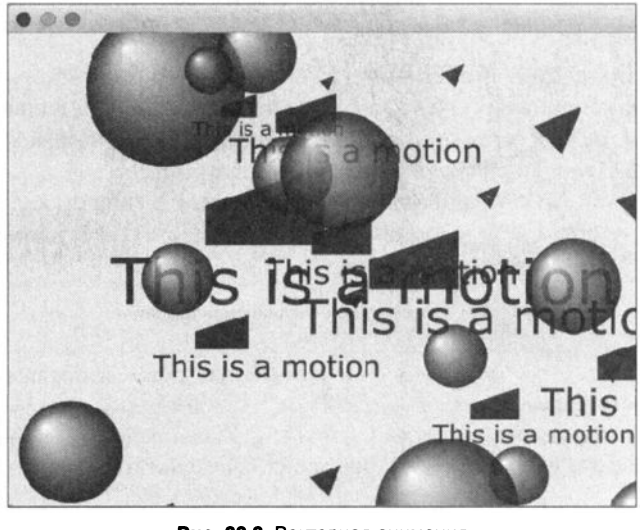


Рис. 22.2. Векторная анимация

Показанный в листинге 22.2 пример иллюстрирует всю простоту использования класса `QSvgWidget` (рис. 22.2). Здесь мы создаем объект класса `QSvgWidget` и передаем в его конструктор SVG-файл motion.svg, который находится в ресурсах. Затем мы просто вызываем метод `show()`, чтобы отобразить сам виджет. Соединение сигнала `repaintNeeded()` объекта

класса QSvgRenderer со слотом repaint() класса QSvgWidget нужно для того, чтобы после создания каждого нового изображения оно отображалось виджетом QSvgWidget.

Листинг 22.2. Создание векторной анимации (файл main.cpp)

```
#include <QtWidgets>
#include <QtSvg>

int main(int argc, char **argv)
{
    QApplication app(argc, argv);

    QSvgWidget svg(":/motion.svg");
    svg.show();

    QObject::connect(svg.renderer(), SIGNAL(repaintNeeded()),
                     &svg,           SLOT(repaint())
                    );

    return app.exec();
}
```

Анимационный движок и машина состояний

Qt также предоставляет возможности анимации пользовательского интерфейса, которые базируются на изменении свойств объектов, — таких как, например, размер, позиция, цвет и т. д. На самом деле некоторые элементы анимации уже были в Qt и до появления этого модуля — вспомните, например, о процессе перетаскивания и помещения окон из и в области доков основного окна приложения (об этом мы предметно поговорим в *главе 34*). Но тогда не было специализированного отдельного механизма для реализации анимации, и ее воспроизводили обычно с помощью классов QTimer, QTimeLine и QGraphicsItemAnimation. Такой подход с появлением нового механизма теперь считается устаревшим, и желательно его более не использовать.

Новый механизм анимации позволяет задействовать математические функции, называемые *смягчающими линиями*, которые дают возможность более реалистично проводить изменения свойств объектов в заданном промежутке времени. Анимации могут группироваться друг с другом и совместно работать с машиной состояний. Их также можно использовать для всех виджетов и элементов GraphicsView.

Новая реализация анимации базируется на классах, показанных в схеме, приведенной на рис. 22.3. Класс QAbstractAnimation — это базовый класс, он абстрагирует таймер и его события и имеет все основные слоты для управления анимацией, — такие как start(), stop() и pause() для запуска, приостановки и остановки анимации. В сигнале stateChanged() этот класс предоставляет всю информацию, когда анимация началась и когда закончилась. Класс QAbstractAnimation наследуют три класса: первый — это QVariantAnimation, второй — QAnimationGroup и третий — QPauseAnimation. Фрагмент Variant класса QVariantAnimation означает, что этот класс может анимировать различные типы, цвет, целые значения, а также и любой другой тип, если к нему есть интерполятор. Мы в основном будем использовать унаследованный от QVariantAnimation класс QPropertyAnimation, потому что это конкретный класс, работающий со свойствами объекта.

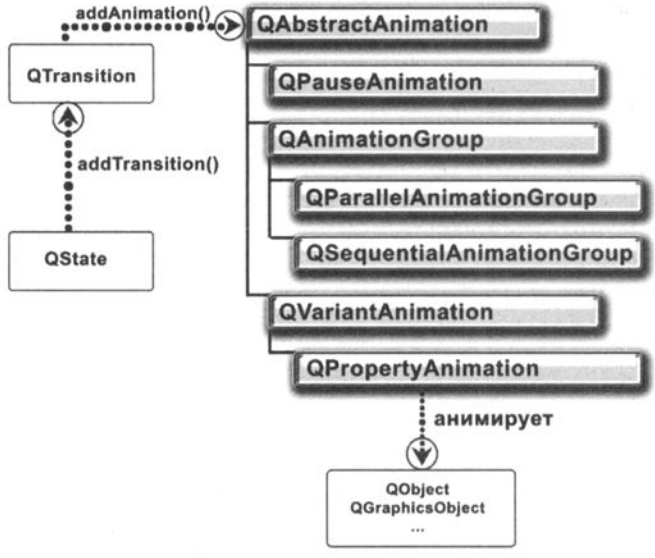


Рис. 22.3. Взаимодействие классов анимационного движка

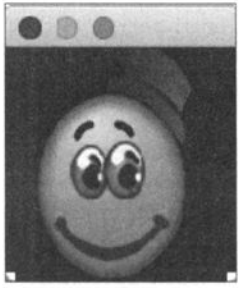


Рис. 22.4. Цветовая анимация

Теперь перейдем к практической стороне дела и создадим анимацию для виджета (рис. 22.4), которая будет изменять свойство цвета (листинг 22.3).

В листинге 22.3 мы создаем виджет класса `QLabel` и устанавливаем в нем вызовом метода `setPixmap()` растровое изображение. После чего создаем объект эффекта цвета (см. *главу 18*) и устанавливаем его в виджете надписи методом `setGraphicsEffect()`. При создании объекта анимации свойств (`anim`) мы передаем в конструкторе адрес на объект эффекта `effect`, а во втором параметре имя свойства `"color"`, которое мы хотим анимировать. Метод `setStartValue()` задает начальное цветовое значение, метод `setKeyValueAt()` — ключевые значения на промежутке от 0 до 1, а вызов метода `setEndValue()` — конечное цветовое значение.

> ***О ЗАДАНИИ СТАРТОВОГО ЗНАЧЕНИЯ***
>
> **Задавать стартовое значение совсем не обязательно, так как оно в этом случае может быть автоматически взято из самого свойства. Задавать стартовое значение необходимо в тех случаях, когда стартовое значение должно отличаться от значения свойства, которое оно имеет по умолчанию.**

Метод `setDuration()` задает время продолжительности изменения значений от стартового до конечного в миллисекундах. В нашем случае оно составляет 3000 мс или, иначе, 3 сек. Метод `setLoopCount()` устанавливает количество раз исполнения цикла анимации, а передача в качестве аргумента отрицательного значения, как в нашем случае, устанавливает бесконечное количество раз. И наконец, вызов метода `start()` выполняет запуск анимации.

Листинг 22.3. Анимация цвета (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QLabel       lbl;
```

```
    lbl.setPixmap(QPixmap(":/happyos.png"));

    QGraphicsColorizeEffect effect;
    lbl.setGraphicsEffect(&effect);

    QPropertyAnimation anim(&effect, "color");
    anim.setStartValue(QColor(Qt::gray));
    anim.setKeyValueAt(0.25f, QColor(Qt::green));
    anim.setKeyValueAt(0.5f, QColor(Qt::blue));
    anim.setKeyValueAt(0.75f, QColor(Qt::red));
    anim.setEndValue(QColor(Qt::black));
    anim.setDuration(3000);
    anim.setLoopCount(-1);
    anim.start();

    lbl.show();

    return app.exec();
}
```

Теперь пришло время немного рассказать о классе `QAnimationGroup`. Этот класс является контейнером для анимаций. Тут так же используется механизм объектной иерархии, то есть модель предков и потомков, реализуемая классом `QObject`. Класс группы отвечает за анимацию всех его потомков. Сама группа для анимаций может быть либо последовательной, либо параллельной. Таким образом, если вы хотите создать анимации, которые работают одновременно друг с другом, то это должна быть параллельная группа. А вот анимации, которые исполняются в порядке очереди, должны находиться в последовательной группе. Кроме того, анимации могут быть добавлены в различные типы групп, как это показано в листинге 22.4.

Листинг 22.4. Группы анимации

```
QParallelAnimationGroup* pgroup1 = new QParallelAnimationGroup;
pgroup1->addAnimation(panim1);
pgroup1->addAnimation(panim2);
QSequentialAnimationGroup* pgroup2 = new QSequentialAnimationGroup;
pgroup2->addAnimation(panim3);
pgroup2->addAnimation(panim4);
pgroup2->addAnimation(panim5);
pgroup2->addAnimation(group1);
...
pgroup2->start();
```

Здесь мы создаем объект параллельной группы (класс `QParallelAnimationGroup`) и вызовом метода `addAnimation()` добавляем в нее два указателя объектов анимации: `panim1` и `panim2`. С объектом последовательной группы (класс `QSequentialAnimationGroup`) мы поступаем аналогичным образом, но в последнем методе `addAnimation()` добавляем не просто анима-

цию, а объект анимационной группы. После чего методом `start()` запускаем исполнение всех анимаций в порядке созданной нами иерархии.

Смягчающие линии

До сих пор наши анимации исполнялись по линейному закону, но окружающий нас мир более разнообразен и сложен. Смягчающие линии (Easing Curves) придают особую реалистичность осуществляемой анимации и создают у пользователей чувство того, что в программе реализован серьезный математический движок для симуляции динамики изменения объектов. Продемонстрируем применение смягчающих линий на простом примере (листинг 22.5), в котором будут анимированы два окна виджетов (рис. 22.5).

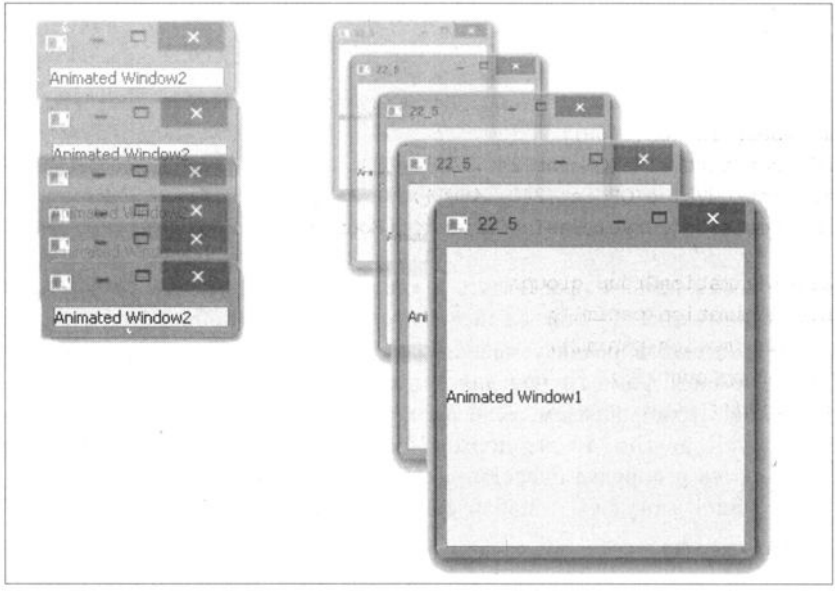


Рис. 22.5. Использование смягчающих линий

В листинге 22.5 мы создаем два виджета надписи (`lbl1` и `lbl2`) и две анимации свойств (указатели `panim1` и `panim2`). Первая анимация будет применяться к свойству виджета `"geometry"` для изменения местоположения и размеров, а вторая — к свойству `"pos"` для изменения только местоположения. Обеим анимациям присваиваем одинаковую продолжительность, равную трем секундам (метод `setDuration()`). Затем в первой анимации мы устанавливаем при помощи методов `setStartValue()` и `setEndValue()` начальные и конечные значения для изменений. В первом случае это объекты класса `QRect`, а во втором — объекты `QPoint`. Метод `setEasingCurve()` устанавливает в первой и второй анимации смягчающие линии. Для того чтобы увидеть различия, используем две разные смягчающие линии: `InOutExpo` и `OutBounce`. Мы хотим, чтобы оба наших окна были анимированы одновременно, поэтому задаем параллельную группу (объект `group`). Добавляем в нее анимации вызовом методов `addAnimation()` и устанавливаем количество ее исполнений, равное трем (метод `setLoopCount()`). И наконец, вызовом метода `start()` из группы анимаций запускаем ее на исполнение.

Листинг 22.5. Смягчающие линии

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QLabel       lbl1("Animated Window1");
    QLabel       lbl2("Animated Window2");

    QPropertyAnimation* panim1 =
        new QPropertyAnimation(&lbl1, "geometry");
    panim1->setDuration(3000);
    panim1->setStartValue(QRect(120, 0, 100, 100));
    panim1->setEndValue(QRect(480, 380, 200, 200));
    panim1->setEasingCurve(QEasingCurve::InOutExpo);

    QPropertyAnimation* panim2 = new QPropertyAnimation(&lbl2, "pos");
    panim2->setDuration(3000);
    panim2->setStartValue(QPoint(240, 0));
    panim2->setEndValue(QPoint(240, 480));
    panim2->setEasingCurve(QEasingCurve::OutBounce);

    QParallelAnimationGroup group;
    group.addAnimation(panim1);
    group.addAnimation(panim2);
    group.setLoopCount(3);
    group.start();

    lbl1.show();
    lbl2.show();

    return app.exec();
}
```

В табл. 22.1 собраны готовые динамики смягчающих линий, которые вы можете использовать в своих программах, передавая в метод `setEasingCurve()` указанные в таблице значения. Если среди приведенных в ней динамик вы не найдете подходящей, то при помощи класса `QEasingCurve` можете создать свою собственную динамику.

Таблица 22.1. *Линии смягчения*

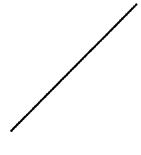
Значение	График динамики	Значение	График динамики
Linear		OutExpo	

Таблица 22.1 *(продолжение)*

Значение	График динамики	Значение	График динамики
`InQuad`		`InOutExpo`	
`OutQuad`		`OutInExpo`	
`InOutQuad`		`InCirc`	
`OutInQuad`		`OutCirc`	
`InCubic`		`InOutCirc`	
`OutCubic`		`OutInCirc`	

Таблица 22.1 (продолжение)

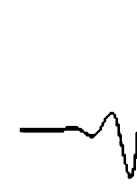
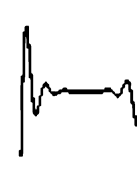
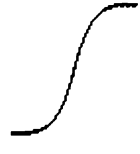
Значение	График динамики	Значение	График динамики
InOutCubic		InElastic	
OutInCubic		OutElastic	
InQuart		InOutElastic	
OutQuart		OutInElastic	
InOutQuart		InBack	
OutInQuart		OutBack	

Таблица 22.1 *(продолжение)*

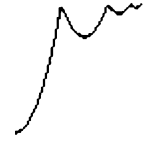
Значение	График динамики	Значение	График динамики
`InQuint`		`InOutBack`	
`OutQuint`		`OutInBack`	
`InOutQuint`		`InBounce`	
`OutInQuint`		`OutBounce`	
`InSine`		`InOutBounce`	
`OutSine`		`OutInBounce`	

Таблица 22.1 *(окончание)*

Значение	График динамики	Значение	График динамики
`InOutSine`		`InExpo`	
`OutInSine`			

Машина состояний и переходы

Цель *состояний* (States) заключается в создании различных аспектов приложения. С их помощью можно присвоить те или иные значения свойствам объектов, которые будут храниться в определенном объекте состояния. Таким образом можно создать много состояний и поместить их в группу состояний. Логика функционирования этой группы подобна использованию кнопок переключения, то есть активным может быть только одно состояние, и активация одного из состояний деактивирует все остальные. Если мы станем показывать одно состояние, затем другое, третье и т. д., то просто одна картинка будет резко сменяться другой, и получится что-то похожее на показ слайдов, а это не так уж и привлекательно. Реальный же мир намного интереснее и разнообразнее. Поэтому между отдельными состояниями нужны более плавные *переходы* (Transitions). Переходы являются инструментом смены состояний и соединения состояний с анимациями. В качестве демонстрации работы состояний и переходов реализуем пример элемента, базирующегося на двух состояниях: `Off` и `On` (листинг 22.6). Первое состояние — `Off` — является начальным. Нажатие на кнопку **Push** будет перемещать ее в противоположный конец и изменять текущее состояние, как это показано на рис. 22.6.

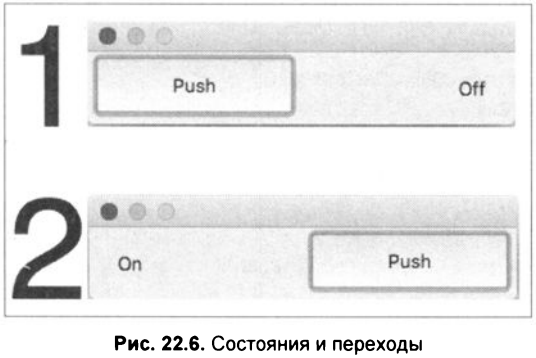


Рис. 22.6. Состояния и переходы

В самом начале листинга 22.6 мы создаем виджет, на поверхности которого будем размещать наши элементы (wgt), и вызовом метода setFixedSize() задаем ему постоянные размеры. Затем создаем виджеты надписей (указатели plblOff и plblOn) и размещаем их в менеджере компоновки (указатель phbx). Созданный виджет кнопки нажатия не будем размещать в менеджере компоновки, потому что хотим быть в состоянии самостоятельно изменять его местоположение и размеры. Далее начинается ключевая часть программы — создаем объект машины состояний (класс QStateMachine), которая будет управлять нашими состояниями. Создаем наше первое состояние, объект класса QState, для состояния Off (указатель pStateOff). Методами assignProperty() устанавливаем значение геометрии для кнопки и состояния видимости для текстовых надписей. В состоянии Off текстовая надпись **Off** должна быть видима, а **On** нет. Вызов метода setInitialState() с передачей указателя pStateOff из объекта машины состояний делает это состояние начальным. Для состояния On мы также вызываем серию методов assignProperty(), только присваиваем другие значения, соответствующие этому состоянию.

Вызов метода addTransition() добавляет переход из того состояния, из которого вызван этот метод, в другое, указанное в этом методе третьим параметром состояния. В нашем примере мы хотим перейти из состояния Off в состояние On — поэтому первым и вторым параметром мы указываем на то, что этот переход должен происходить при нажатии на кнопку **Push**. Аналогично мы поступаем и с состоянием On, но только в обратном порядке. Метод addTransction() возвращает указатель на объект класса QSignalTransition, который мы используем дальше для присвоения переходу нужной анимации. Создаем две анимации (указатели panim1 и panim2) для перехода в состояние On и Off и устанавливаем их вызовом методов addAnimation() в объектах переходов (указатели ptrans1 и ptrans2). В завершение выполним запуск созданной нами машины состояний вызовом слота start().

Листинг 22.6. Состояния и переходы (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;
    wgt.setFixedSize(300, 50);
    wgt.show();

    QLabel* plblOff = new QLabel("Off");
    QLabel* plblOn  = new QLabel("On");

    QHBoxLayout* phbx = new QHBoxLayout;
    phbx->addWidget(plblOn);
    phbx->addStretch(1);
    phbx->addWidget(plblOff);
    wgt.setLayout(phbx);

    QPushButton* pcmd = new QPushButton("Push", &wgt);
    pcmd->setAutoFillBackground(true);
    pcmd->show();
```

```
    int nButtonWidth = wgt.width() / 2;

    QStateMachine* psm = new QStateMachine;

    QState* pStateOff = new QState(psm);
    QRect   rect1(0, 0, nButtonWidth, wgt.height());
    pStateOff->assignProperty(pcmd, "geometry", rect1);
    pStateOff->assignProperty(plblOff, "visible", true);
    pStateOff->assignProperty(plblOn, "visible", false);
    psm->setInitialState(pStateOff);

    QState* pStateOn = new QState(psm);
    QRect   rect2(nButtonWidth, 0, nButtonWidth, wgt.height());
    pStateOn->assignProperty(pcmd, "geometry", rect2);
    pStateOn->assignProperty(plblOff, "visible", false);
    pStateOn->assignProperty(plblOn, "visible", true);

    QSignalTransition* ptrans1 =
        pStateOff->addTransition(pcmd, SIGNAL(clicked()), pStateOn);

    QSignalTransition* ptrans2 =
        pStateOn->addTransition(pcmd, SIGNAL(clicked()), pStateOff);

    QPropertyAnimation* panim1 =
        new QPropertyAnimation(pcmd, "geometry");
    ptrans1->addAnimation(panim1);

    QPropertyAnimation* panim2 =
        new QPropertyAnimation(pcmd, "geometry");
    ptrans2->addAnimation(panim2);

    psm->start();

    return app.exec();
}
```

Резюме

Анимация широкое распространение получила сравнительно недавно. Анимационные файлы состоят из некоторого количества неподвижных изображений, которые при последовательном отображении с определенной скоростью создают иллюзию движения. Основным классом для простой анимации является класс `QMovie`, который обладает рядом методов для управления анимацией. Проще всего установить объект анимации в виджете надписи с помощью метода `setMovie()`.

Для отображения векторной графики или анимации в формате SVG существует модуль `QtSvg`, а центральным классом для ее показа является виджет `QSvgWidget`.

Qt предоставляет возможности применения анимаций для интерфейса пользователя. Анимация придает приложению более естественный вид и осуществляет изменения таких

свойств виджета, как позиции, прозрачность, размеры и т. п. в заданном промежутке времени. На этом промежутке времени можно использовать смягчающие линии, которые осуществят указанные изменения по нелинейным законам, что придаст происходящему еще более естественный вид.

Для анимации можно использовать и отдельные состояния. С их помощью удается задать различные позиции сразу для многих виджетов. Переключение одного состояния в другое не очень красиво, потому что это просто перескакивание из одного состояния в другое. Поэтому есть еще одна возможность, которую можно использовать совместно, — это переходы с анимациями, необходимые для того, чтобы сделать изменения между состояниями более привлекательными и естественными.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/22-510/** или с помощью следующего QR-кода (рис. 22.7):



Рис. 22.7. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 23

Работа с OpenGL

Реальность воображаема, а воображаемое — реально.
В. Соло

Трехмерная графика, несомненно, одна из самых захватывающих тем в программировании, которая существует с ранних стадий развития компьютерной техники. На протяжении продолжительного времени она применялась исключительно в рамках правительственных проектов и в проектах разработки симуляторов полета. С недавнего времени трехмерная графика вышла за рамки научной деятельности и превратилась в целую индустрию, включающую в себя такие сферы, как анимация (от спецэффектов до компьютерных игр), кино, виртуальная реальность, медицина и многое другое. С каждым годом все больше людей задействованы в этой области.

Графическая библиотека OpenGL — это стандарт для двумерной и трехмерной графики, впервые введенный компанией Silicon Graphics в 1992 году. Это уже устоявшийся стандарт, и все вносимые в него изменения делаются с учетом гарантий нормальной работы ранее написанного кода. Сама же библиотека может быть создана кем угодно, главное, чтобы она отвечала спецификации, установленной стандартом. С точки зрения программиста, библиотека OpenGL представляет собой множество команд для создания объектов и выполнения сложных операций — от сглаживания (Anti-aliasing) до наложения текстур. Для ее использования достаточно усвоить несколько простых правил, которые обеспечат возможность реализации замечательных программ.

Хотя OpenGL и является платформонезависимой библиотекой, но все равно, чтобы использовать OpenGL-программу на разных платформах, требуется провести ряд преобразований кода программы для осуществления привязки *контекста воспроизведения* (rendering context) к оконной системе платформы. Использование же OpenGL «в оправе» Qt освобождает разработчиков от каких бы то ни было изменений текста исходного кода, что обеспечивает для OpenGL-программ полную платформонезависимость. Так как OpenGL интегрирован в Qt, его можно использовать в Qt-приложениях сразу. Благодаря продуманности системы рисования Arthur, все операции `QPainter` (см. *главу 18*) также могут быть применены и для библиотеки OpenGL. Преимущество использования OpenGL состоит в возможности работы с двумерной и трехмерной графикой и ее быстрого отображения.

Основные положения OpenGL

Библиотека OpenGL не является объектно-ориентированной. При работе с библиотекой разработчик имеет дело только с функциями, переменными и константами. Имена всех

функций OpenGL начинаются с букв `gl`, а констант — с `GL_`. В имена функций входят суффиксы, говорящие о количестве и типе передаваемых параметров. Например, прототип функции `glColor3f()` говорит о том, что в нее должны передаваться три значения с плавающей точкой (рис. 23.1). Поэтому при описании функций в OpenGL, чтобы не повторяться, принято вместо числа передаваемых аргументов и их типа ставить символ `*`. Итак, общий вид для упомянутой ранее функции будет выглядеть следующим образом: `glColor*()`. При этом подразумевается, что речь идет не об одной функции, а о целой серии функций, начинающихся с `glColor`.

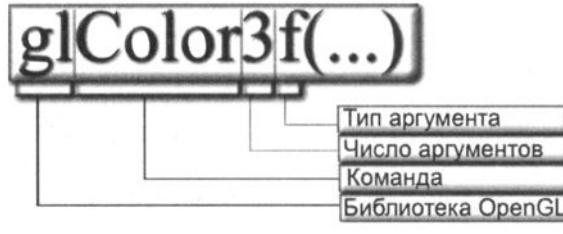


Рис. 23.1. Формат команд OpenGL

В табл. 23.1 приведены типы OpenGL и символы суффиксов, используемые в ней.

Таблица 23.1. *Суффиксы и типы OpenGL*

Суффикс	Тип OpenGL	C++ Эквивалент	Описание
b	GLbyte	Char	Байт
s	GLshort	Short	Короткое целое
i	GLint	Int	Целое
f	GLfloat	float	С плавающей точкой
d	GLdouble	double	С плавающей точкой двойной точности
ub	GLubyte	unsigned byte	Байт без знака
us	GLushort	usnigned short	Короткое целое без знака
ui	GLuint	unsigned int	Целое без знака
GL_	GLenum	Enum	Перечисление
v			Массив из n параметров

Суффикс `v` говорит о том, что функция принимает массив.

Например, массив из трех значений с плавающей точкой в функцию `glColor3fv()` передается следующим образом:

```
GLfloat a[] = {1.0f, 0.0f, 0.0f}
glColor3fv(a);
```

Реализация OpenGL-программы

Чтобы воспользоваться OpenGL, необходимо унаследовать класс `QOpenGLWidget`. Как видно из названия, этот класс является виджетом и организует соединение с функциями библиотеки OpenGL.

> ***Использование объектов класса QOpenGLWidget в качестве контекста рисования для QPainter***
>
> Объекты класса `QOpenGLWidget` могут также использоваться в качестве контекста рисования для `QPainter` (см. *главу 18*).

В унаследованном от `QOpenGLWidget` классе необходимо, по меньшей мере, переопределить три виртуальных метода: `initializeGL()`, `resizeGL()` и `paintGL()`. Эти методы определены в классе `QOpenGLWidget` как `virtual protected`.

Метод `initializeGL()` вызывается сразу после создания объекта. Это требуется для проведения инициализаций, связанных с OpenGL. Метод вызывается, если объекту, унаследованному от класса `QOpenGLWidget`, присваивается контекст OpenGL.

Назначение метода `resizeGL(int width, int height)` схоже с назначением метода обработки события изменения размера `resizeEvent()`. Этот метод вызывается при изменении размеров объекта, созданного от класса, наследующего `QOpenGLWidget`. В параметрах метода передаются актуальные размеры виджета.

Назначение метода `paintGL()` схоже с назначением метода обработки события рисования `paintEvent()`. Метод вызывается в тех случаях, когда требуется заново перерисовать содержимое виджета. Это происходит, например, после вызова метода `resizeGL()`.

Далее рассмотрено приложение (листинги 23.1–23.6), которое отображает четырехугольник с вершинами разного цвета со сглаживанием (рис. 23.2).

Программа, приведенная в листинге 23.1, лишь создает объект класса `OGLQuad`, унаследованный от класса `QOpenGLWidget`.



Рис. 23.2. Сглаживание цветов вершин четырехугольника

Листинг 23.1. Создание объекта класса OGLQuad (файл main.cpp)

```
#include <QApplication>
#include "OGLQuad.h"

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    OGLQuad      oglQuad;

    oglQuad.resize(200, 200);
    oglQuad.show();

    return app.exec();
}
```

В листинге 23.2 класс OGLQuad определяется как класс, наследующий QOpenGLWidget, в котором переопределены три метода: initializeGL(), resizeGL() и paintGL().

Листинг 23.2. Определение класса OGLQuad (файл OGLQuad.h)

```
#pragma once

#include <QOpenGLWidget>

// ========================================================================
class OGLQuad : public QOpenGLWidget {
protected:
    virtual void initializeGL(                      );
    virtual void resizeGL    (int nWidth, int nHeight);
    virtual void paintGL     (                      );

public:
    OGLQuad(QWidget* pwgt = 0);
};
```

Конструктор, приведенный в листинге 23.3, не имеет реализации, он просто передает указатель на объект предка pwgt конструктору наследуемого класса.

Листинг 23.3. Конструктор OGLQuad (файл OGLQuad.cpp)

```
OGLQuad::OGLQuad(QWidget* pwgt/*= 0*/) : QOpenGLWidget(pwgt)
{
}
```

В листинге 23.4 мы получаем при помощи статического метода currentContext() класса QOpenGLContext() доступ к текущему контексту OpenGL и методом functions() — к его функциям.

> ***Контекст OpenGL***
>
> Контекст OpenGL — это набор переменных состояния, и каждый объект класса QOpenGLWidget создает его автоматически.

Затем методом glClearColor() устанавливается цвет для очистки буфера изображения в формате OpenGL.

Листинг 23.4. Метод initializeGL() (файл OGLQuad.cpp)

```
/*virtual*/void OGLQuad::initializeGL()
{
   QOpenGLFunctions* pFunc =
        QOpenGLContext::currentContext()->functions();
   pFunc->glClearColor(0.0f, 0.0f, 0.0f, 1.0f);
}
```

В листинге 23.5 функция `glMatrixMode()` устанавливает матрицу проектирования текущей матрицей. Это означает, что все последующие преобразования будут влиять только на нее. Вызовом функции `glLoadIdentity()` текущая матрица устанавливается в единичную. В OpenGL существуют две матрицы, применяющиеся для преобразования координат. Первая — *матрица моделирования* (modelview matrix) — служит для задания положения объекта и его ориентации, а вторая — *матрица проектирования* (projection matrix) — отвечает за выбранный способ проектирования. Способ проектирования может быть либо ортогональным, либо перспективным.

Метод `resizeGL()` — это самое удобное место, чтобы установить *видовое окно* (viewport). Видовое окно устанавливается вызовом функции `glViewPort()` и представляет собой прямоугольную область в пределах окна виджета (окно в окне). В нашем случае видовое окно совпадает со всей областью виджета. Соотношение сторон видового окна задается параметрами функции `glOrtho()`. Первый и второй параметры задают положения левой и правой отсекающих плоскостей, третий и четвертый — верхней и нижней отсекающих плоскостей, пятый и шестой — передней и задней отсекающих плоскостей, соответственно.

Листинг 23.5. Метод `resizeGL()` (файл OGLQuad.cpp)

```
/*virtual*/void OGLQuad::resizeGL(int nWidth, int nHeight)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glViewport(0, 0, (GLint)nWidth, (GLint)nHeight);
    glOrtho(0, 100, 100, 0, -1, 1);
}
```

Перед формированием нового изображения нужно функцией `glClear()` очистить буферы изображения (`GL_COLOR_BUFFER_BIT`) и глубины (`GL_DEPTH_BUFFER_BIT`). Последний служит для удаления невидимых поверхностей. Для очистки буфера изображения будет использован цвет, установленный методом `glClearColor()`.

Единица информации OpenGL — *вершина*. Из вершин строятся сложные объекты, при создании которых нужно указать, каким образом они должны быть соединены друг с другом. Способом соединения вершин управляет функция `glBegin()`. В нашем примере флаг `GL_QUADS` говорит о том, что на создаваемых вершинах должен быть построен четырехугольник (листинг 23.6). При помощи функции `glColor*()` задается текущий цвет, который распространяется на последующие вызовы функций `glVertex*()`, задающих расположение вершин. Функции задания вершин должны всегда находиться между `glBegin()` и `glEnd()`. В нашем примере каждая вершина имеет свой цвет, и, поскольку по умолчанию в OpenGL включен режим сглаживания цветов, это приводит к созданию радужной окраски области прямоугольника. Режимом сглаживания цветов управляет функция `glShadeModel()`. Ее вызов с флагом `GL_FLAT` отключает режим сглаживания, а передача флага `GL_SMOOTH` включает его.

Листинг 23.6. Метод `paintGL()` (файл OGLQuad.cpp)

```
/*virtual*/void OGLQuad::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
```

```
    glBegin(GL_QUADS);
        glColor3f(1, 0, 0);
        glVertex2f(0, 100);

        glColor3f(0, 1, 0);
        glVertex2f(100, 100);

        glColor3f(0, 0, 1);
        glVertex2f(100, 0);

        glColor3f(1, 1, 1);
        glVertex2f(0, 0);
    glEnd();
}
```

Напоминание

Электронный архив с примерами к этой книге можно скачать с FTP-сервера издательства «БХВ-Петербург» по ссылке **ftp://ftp.bhv.ru/9785977536783.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение 4*).

Разворачивание OpenGL-программ во весь экран

Поскольку класс `QOpenGLWidget` унаследован от `QWidget`, он обладает всеми свойствами, присущими этому классу. Развернуть программу на полный экран очень просто — для этого нужно заменить вызов метода `show()` на вызов метода `showFullScreen()`. В результате такой замены виджет верхнего уровня перекроет своим окном все остальные и займет весь экран. Это очень удобно, так как дает возможность отлаживать программу в маленьком окне, а когда она будет готова, просто поменять метод `show()` на `showFullScreen()`.

Производительность OpenGL-программы в полноэкранном режиме полностью зависит от возможностей видеокарты.

Графические примитивы OpenGL

OpenGL предоставляет средства для рисования графических примитивов, таких как точки, линии, ломаные и многоугольники, которые задаются одной или несколькими вершинами. Для этого необходимо передать список вершин.

Следующий пример (листинги 23.7–23.13) формирует различные фигуры, построенные на одних и тех же вершинах (рис. 23.3).

В листинге 23.7 в основной программе создается виджет `oglDraw` класса `OGLDraw`.

Листинг 23.7. Сздание виджета `oglDraw` класса `OGLDraw` (файл main.cpp)

```
#include <QApplication>
#include "OGLDraw.h"
```

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    OGLDraw      oglDraw;

    oglDraw.resize(400, 200);
    oglDraw.show();

    return app.exec();
}
```

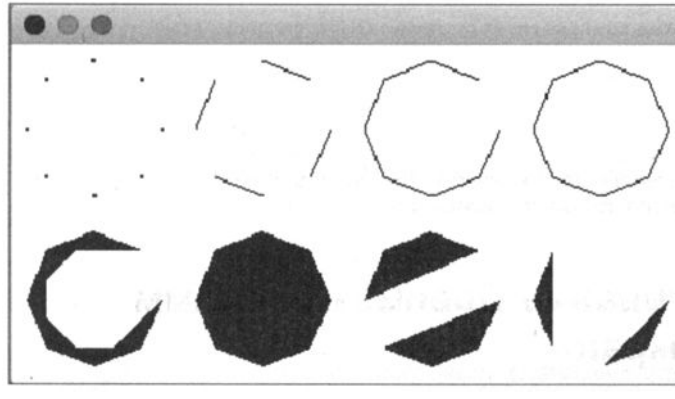


Рис. 23.3. Отображение фигур, построенных на одних и тех же вершинах

Класс `OGLDraw` наследуется от класса `QOpenGLWidget` и перезаписывает три его виртуальных метода: `initializeGL()`, `resizeGL()` и `paintGL()` (листинг 23.8). В классе определен метод `draw()`, получающий координаты `x` и `y`, с которых нужно начинать строить фигуру. Тип фигуры задается третьим параметром.

Листинг 23.8. Определение класса `OGLDraw` (файл OGLDraw.h)

```
#pragma once

#include <QOpenGLWidget>

// ======================================================================
class OGLDraw : public QOpenGLWidget {
protected:
    virtual void initializeGL(                      );
    virtual void resizeGL    (int nWidth, int nHeight);
    virtual void paintGL     (                      );

public:
    OGLDraw(QWidget* pwgt = 0);

    void draw(int xOffset, int yOffset, GLenum type);
};
```

В листинге 23.9 приведен конструктор класса `OGLDraw`, который не выполняет никаких действий, а просто передает указатель на объект-предок в конструктор класса `QOpenGLWidget`.

Листинг 23.9. Конструктор `OGLDraw` (файл OGLDraw.cpp)

```
OGLDraw::OGLDraw(QWidget* pwgt/*= 0*/) : QOpenGLWidget(pwgt)
{
}
```

В листинге 23.10 в методе `initializeGL()` устанавливается белый цвет для очистки буфера изображения.

Листинг 23.10. Метод `initializeGL()` (файл OGLDraw.cpp)

```
/*virtual*/void OGLDraw::initializeGL()
{
    QOpenGLFunctions* pFunc =
        QOpenGLContext::currentContext()->functions();
    pFunc->glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
}
```

В листинге 23.11 действия метода `resizeGL()` аналогичны действиям листинга 23.6.

Листинг 23.11. Метод `resizeGL()` (файл OGLDraw.cpp)

```
/*virtual*/void OGLDraw::resizeGL(int nWidth, int nHeight)
{
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glViewport(0, 0, (GLint)nWidth, (GLint)nHeight);
    glOrtho(0, 400, 200, 0, -1, 1);
}
```

В листинге 23.12 приводится метод `paintGL()`, в котором после очистки буферов изображения и глубины осуществляется серия вызовов метода `draw()` (см. листинг 23.13). В этот метод передаются координаты `x` и `y`, с которых будет начинаться рисование фигуры. Тип фигуры, как уже отмечалось ранее, передается третьим параметром:

- тип `GL_POINTS` говорит о том, что отображаться должны только точки;
- при построении фигуры типа `GL_LINES` каждая пара вершин задает отрезки, которые, как правило, не соединяются друг с другом;
- тип `GL_LINE_STRIP` задает ломаную линию и используется, в основном, для аппроксимации кривых. Если требуется получить замкнутый контур, то нужно указать одну и ту же вершину в начале и в конце серии вершин;
- тип `GL_LINE_LOOP` также задает ломаную линию, но последняя ее точка соединяется с первой;
- тип `GL_TRIANGLE_STRIP` задает треугольники с общей стороной. Каждая вершина, начиная с третьей, комбинируется с двумя предыдущими и определяет очередную ячейку;

- тип GL_POLYGON задает многоугольник;
- при построении фигуры типа GL_QUADS каждые четыре вершины задают четырехугольник;
- при построении фигуры типа GL_TRIANGLES каждые три вершины задают треугольник.

Листинг 23.12. Метод paintGL() (файл OGLDraw.cpp)

```
/*virtual*/void OGLDraw::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    draw(0, 0, GL_POINTS);
    draw(100, 0, GL_LINES);
    draw(200, 0, GL_LINE_STRIP);
    draw(300, 0, GL_LINE_LOOP);

    draw(0, 100, GL_TRIANGLE_STRIP);
    draw(100, 100, GL_POLYGON);
    draw(200, 100, GL_QUADS);
    draw(300, 100, GL_TRIANGLES);
}
```

Листинг 23.13. Метод draw() (файл OGLDraw.cpp)

```
void OGLDraw::draw(int xOffset, int yOffset, GLenum type)
{
    int n = 8;

    glPointSize(2);
    glBegin(type);
        glColor3f(0, 0, 0);
        for (int i = 0; i < n; ++i) {
            float fAngle = 2 * 3.14 * i / n;
            int   x      = (int)(50 + cos(fAngle) * 40 + xOffset);
            int   y      = (int)(50 + sin(fAngle) * 40 + yOffset);
            glVertex2f(x, y);
        }
    glEnd();
}
```

В листинге 23.13 приведен метод draw(), который осуществляет построение фигуры заданного типа type с позиции, определяемой переданными координатами. Для задания размеров точки служит функция glPointSize(). В нашем примере ее размер устанавливается равным 2. Функция glColor*() устанавливает черный цвет для вершин. Вызов функции glVertex*() задает расположение вершин.

Трехмерная графика

Следующий пример (листинги 23.14–23.22) представляет программу, отображающую пирамиду в трехмерном пространстве, поворот которой относительно осей X и Y осуществляется при помощи мыши (рис. 23.4).

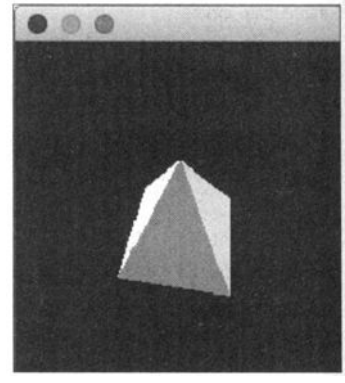


Рис. 23.4. Пирамида

В основной программе (листинг 23.14) создается OpenGL-виджет класса `OGLPyramid`.

Листинг 23.14. Создание OpenGL-виджета класса `OGLPyramid` (файл main.cpp)

```
#include <QApplication>
#include "OGLPyramid.h"

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    OGLPyramid   oglPyramid;

    oglPyramid.resize(200, 200);
    oglPyramid.show();

    return app.exec();
}
```

В классе `OGLPyramid` (листинг 23.15) определена переменная `m_nPyramid`, которая хранит номер дисплейного списка объекта пирамиды.

> ***Дисплейные списки***
>
> Дисплейные списки позволяют выделить конкретный набор команд, запомнить его и вызывать всякий раз, когда в нем возникает необходимость. Этот механизм очень похож на вызов обычных функций или на механизм записи графических команд, предоставляемый классом `QPicture`. Для запуска команд дисплейного списка необходимо знать присвоенный ему уникальный номер.

Переменные `m_xRotate`, `m_yRotate` нужны для хранения углов поворота по осям X и Y. Переменная `m_ptPosition` хранит координату указателя мыши в момент нажатия. Кроме трех методов, унаследованных от `QOpenGLWidget`, переопределяются методы обработки события мыши `mousePressEvent()` и `mouseMoveEvent()` для осуществления поворота пирамиды. Метод `createPyramid()` создает дисплейный список для отображения объекта пирамиды.

Листинг 23.15. Определение класса OGLPyramid (файл OGLPyramid.h)

```
#pragma once

#include <QOpenGLWidget>

// ===================================================================
class OGLPyramid : public QOpenGLWidget {
private:
    GLuint  m_nPyramid;
    GLfloat m_xRotate;
    GLfloat m_yRotate;
    QPoint  m_ptPosition;

protected:
    virtual void   initializeGL   (                        );
    virtual void   resizeGL       (int nWidth, int nHeight);
    virtual void   paintGL        (                        );
    virtual void   mousePressEvent(QMouseEvent* pe         );
    virtual void   mouseMoveEvent (QMouseEvent* pe         );
            GLuint createPyramid  (GLfloat fSize = 1.0f    );

public:
    OGLPyramid(QWidget* pwgt = 0);
};
```

Задача конструктора класса `OGLPyramid` (листинг 23.16) состоит в инициализации переменных-членов для поворота и передачи указателя на виджет предка конструктору наследуемого класса `QOpenGLWidget`.

Листинг 23.16. Конструктор OGLPyramid (файл OGLPyramid.cpp)

```
OGLPyramid::OGLPyramid(QWidget* pwgt/*= 0*/) : QOpenGLWidget(pwgt)
                                             , m_xRotate(0)
                                             , m_yRotate(0)
{
}
```

При инициализации с помощью метода `glClearColor()` устанавливается черный цвет очистки буфера изображения (листинг 23.17). Функция `glEnable()` устанавливает режим разрешения проверки глубины фрагментов. Режим сглаживания цветов по умолчанию разрешен, поэтому его необходимо отключить, передав в функцию `glShadeMode()` флаг `GL_FLAT`, иначе боковые грани пирамиды будут иметь радужную окраску. Вызов метода `createPyramid()` (см. листинг 23.22) создает дисплейный список для пирамиды и возвращает его номер, который присваивается переменной `m_nPyramid`. Параметр, передаваемый в этот метод, задает размеры самой пирамиды.

Листинг 23.17. Метод initializeGL() (файл OGLPyramid.cpp)

```
/*virtual*/void OGLPyramid::initializeGL()
{
    QOpenGLFunctions* pFunc =
        QOpenGLContext::currentContext()->functions();
    pFunc->glClearColor(0.0f, 0.0f, 0.0f, 1.0f);

    pFunc->glEnable(GL_DEPTH_TEST);
    glShadeModel(GL_FLAT);
    m_nPyramid = createPyramid(1.2f);
}
```

В листинге 23.18 функция `glViewPort()` устанавливает размеры видового окна равными размерам окна виджета. Функция `glMatrixMode()` делает текущей матрицу проектирования. Вызов функции `glLoadIdentity()` присваивает матрице проектирования единичную матрицу. Функция `glFrustum()` задает так называемую *пирамиду видимости*. Ее параметры задают положения левой, правой, верхней, нижней, передней и задней отсекающих плоскостей. Последние два значения должны быть положительными и отсчитываться от центра проецирования вдоль оси Z — по ним устанавливается значение перспективы.

Листинг 23.18. Метод resizeGL() (файл OGLPyramid.cpp)

```
/*virtual*/void OGLPyramid::resizeGL(int nWidth, int nHeight)
{
    glViewport(0, 0, (GLint)nWidth, (GLint)nHeight);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glFrustum(-1.0, 1.0, -1.0, 1.0, 1.0, 10.0);
}
```

При рисовании после очистки буфера изображения текущей устанавливается матрица моделирования, служащая для задания положения объекта и его ориентации (листинг 23.19). Функция `glLoadIdentity()` присваивает матрице моделирования единичную матрицу. Функция `glTranslate()` сдвигает начало системы координат по оси Z на 3 единицы. Функция `glRotate()` поворачивает систему координат вокруг осей X и Y на угол, задаваемый переменными `m_xRotate` и `m_yRotate`. Передача в функцию `glCallList()` номера дисплейного списка пирамиды отобразит эту пирамиду.

Листинг 23.19. Метод paintGL() (файл OGLPyramid.cpp)

```
/*virtual*/void OGLPyramid::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -3.0);
```

```
    glRotatef(m_xRotate, 1.0, 0.0, 0.0);
    glRotatef(m_yRotate, 0.0, 1.0, 0.0);

    glCallList(m_nPyramid);
}
```

При нажатии пользователем кнопки мыши переменной `m_ptPosition` присваиваются координаты указателя мыши (листинг 23.20).

Листинг 23.20. Метод `mousePressEvent()` (файл OGLPyramid.cpp)

```
/*virtual*/void OGLPyramid::mousePressEvent(QMouseEvent* pe)
{
    m_ptPosition = pe->pos();
}
```

В методе обработки события перемещения мыши (листинг 23.21) вычисляются углы поворота для осей *X* и *Y*. Вызов метода `updateGL()` обновляет изображение на экране, используя новые углы поворота. Переменной `m_ptPosition` присваивается актуальная координата указателя мыши.

Листинг 23.21. Метод `mouseMoveEvent()` (файл OGLPyramid.cpp)

```
/*virtual*/void OGLPyramid::mouseMoveEvent(QMouseEvent* pe)
{
    m_xRotate += 180 * (GLfloat)(pe->y() - m_ptPosition.y()) / height();
    m_yRotate += 180 * (GLfloat)(pe->x() - m_ptPosition.x()) / width();
    updateGL();

    m_ptPosition = pe->pos();
}
```

Метод `createPyramid()` создает дисплейный список для отображения пирамиды и возвращает его номер (листинг 23.22). Функция `glGenLists()` возвращает первый свободный номер для идентификации дисплейного списка. Этот номер передается в функцию `glNewList()`. Второй параметр, `GL_COMPILE`, говорит о том, что команды нужно лишь запомнить. Все команды, находящиеся между функциями `glNewList()` и `glEndList()`, помещаются в соответствующий дисплейный список. Тип `GL_TRIANGLE_FAN` задает треугольники с общей вершиной, которая идет первой в списке. Следующие две вершины задают треугольник. Затем каждая последующая вершина, совместно с предыдущей, задает следующий треугольник. А для типа фигуры `GL_QUADS` каждые четыре вершины задают четырехугольник.

Листинг 23.22. Метод `createPyramid()` (файл OGLPyramid.cpp)

```
GLuint OGLPyramid::createPyramid(GLfloat fSize/*=1.0f*/)
{
    GLuint n = glGenLists(1);

    GLuint n = glGenLists(1);
```

```
    glNewList(n, GL_COMPILE);
        glBegin(GL_TRIANGLE_FAN);
            glColor4f(0.0f, 1.0f, 0.0f, 1.0f);
            glVertex3f(0.0, fSize, 0.0);
            glVertex3f(-fSize, -fSize, fSize);
            glVertex3f(fSize, -fSize, fSize);
            glColor4f(1.0f, 1.0f, 0.0f, 1.0f);
            glVertex3f(fSize, -fSize, -fSize);
            glColor4f(0.0f, 0.0f, 1.0f, 1.0f);
            glVertex3f(-fSize, -fSize, -fSize);
            glColor4f(1.0f, 1.0f, 1.0f, 1.0f);
            glVertex3f(-fSize, -fSize, fSize);
        glEnd();

        glBegin(GL_QUADS);
            glColor4f(1.0f, 0.0f, 0.0f, 1.0f);
            glVertex3f(-fSize, -fSize, fSize);
            glVertex3f(fSize, -fSize, fSize);
            glVertex3f(fSize, -fSize, -fSize);
            glVertex3f(-fSize, -fSize, -fSize);
        glEnd();
    glEndList();

    return n;
}
```

Резюме

OpenGL прост в изучении и давно используется в качестве стандартного API. Это вполне устоявшийся стандарт, действующий уже на протяжении десятилетий. Библиотека Qt предоставляет модуль для поддержки OpenGL. Для Qt-программ с использованием OpenGL следует в унаследованном от `QOpenGLWidget` классе перезаписать три метода: `initializeGL()`, `resizeGL()` и `paintGL()`.

Преимущество OpenGL заключается в возможности работы с трехмерной графикой. Единица информации OpenGL — вершина. Перечисляя вершины, можно создавать довольно сложные объекты.

Для того чтобы запустить программу в полноэкранном режиме, нужно заменить в основной программе вызов метода `show()` на вызов метода `showFullScreen()`.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/23-510/** или с помощью следующего QR-кода (рис. 23.5):



Рис. 23.5. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 24

Вывод на печать

> Начать пользоваться новым методом на практике легче, чем выверить его; причем, чем точнее метод, тем с большей осторожностью его надо использовать.
>
> *Р. Ф. Бейлс*

В большинстве случаев приложения должны предоставлять пользователю возможность вывода на печать. Это обстоятельство может испугать многих разработчиков, и причиной тому является ряд проблем: различные возможности принтеров, разница в отображении шрифтов на экране и в напечатанном образце, платформозависимые различия в программировании принтеров и т. д. Qt берет на себя решение большинства задач печати, существенно облегчая задачу разработчиков.

Класс *QPrinter*

Класс `QPrinter` является основным для вывода на печать. Благодаря тому, что класс унаследован от `QPaintDevice`, вывод на печать аналогичен выводу на экран. Для вывода на принтер могут применяться те же методы класса `QPainter`, что и для всех остальных контекстов рисования. Класс `QPrinter`, а также и все остальные классы, связанные с выводом на печать, содержится в отдельном модуле `QtPrintSupport`, поэтому для их использования нужно обязательно в pro-файле добавить этот модуль:

```
QT += printsupport
```

Класс принтера `QPrinter` обладает бóльшим числом настроек, чем все остальные классы, унаследованные от класса `QPaintDevice`. Можно, например, установить размер листа, количество копий и т. д. Qt предоставляет диалоговое окно печати, в котором пользователь сам выполняет необходимые настройки. Это окно реализовано в классе `QPrintDialog` (см. *главу 32*). Такие настройки можно проводить и программно с помощью методов класса `QPrinter`:

- при помощи метода `setPageOrientation()` можно задать ориентацию страницы, передав флаги: `QPageLayout::Portrait` — для горизонтального или `QPageLayout::Landscape` — для вертикального расположения страницы;
- метод `setCopyCount()` устанавливает количество выводимых на печать копий;
- с помощью метода `setFromTo()` можно задать диапазон страниц для печати;
- метод `setColorMode()` управляет цветным и черно-белым режимами печати. Для цветного режима нужно передать в метод флаг `QPrinter::Color`, а для черно-белого — флаг `QPrinter::Grayscale`;

- вызовом метода `setPageSize()` изменяется размер листа. В метод нужно передать одно из значений класса `QpageSize`, приведенных в табл. 24.1.

Таблица 24.1. *Перечисления* `PageSizeId` *класса* `QPageSize`

Константа	Размер (мм)	Константа	Размер (мм)	Константа	Размер (мм)
A0	841×1189	B0	1030×1456	B10	32×45
A1	594×841	B1	728×1030	C5E	163×229
A2	420×594	B2	515×728	Comm10E	105×241
A3	297×420	B3	364×515	DLE	110×220
A4	210×297	B4	257×364	Executive	191×254
A5	148×210	B5	182×257	Folio	210×330
A6	105×148	B6	128×182	Ledger	432×279
A7	74×105	B7	91×128	Legal	216×356
A8	52×74	B8	64×91	Letter	216×279
A9	37×52	B9	45×64	Tabloid	279×432

Вместо печати на принтер можно перенаправить вывод в файл. Для этого необходимо передать полное имя файла методу `setOutputFileName()`. Если в метод передать пустую строку, то перенаправление вывода в файл система проигнорирует, и вывод будет осуществлен на печатающее устройство. Имя печатаемого файла можно установить методом `setDocName()` — отметим, что это не имя файла, в который перенаправлена печать, а имя задания на печать.

Кроме перенаправления печати в файл, который содержит команды принтера, можно генерировать файлы в формате PDF (Portable Document Format, платформонезависимый формат электронных документов), и это так же просто, как и вывод на печать, — нужно лишь передать в метод `setOutputFormat()` значение `QPrinter::PdfFormat`.

Класс `QPrinter` содержит метод `setFontEmdeddingEnabled()` для внедрения шрифтов в документ-носитель — чтобы быть уверенным, что полученный файл содержит все необходимые шрифты. Это полезно в том случае, если вы хотите использовать файлы не только на той платформе, на которой они были созданы.

Чтобы отменить операцию печати, нужно вызвать метод `abort()`, который вернет значение булевого типа, сигнализирующее о результате выполнения операции отмены.

Следующий пример (листинги 24.1–24.7) организует вывод на печать картинки, показанной на рис. 24.1. При нажатии на кнопку **Print** (Печать) откроется диалоговое окно настроек принтера и после подтверждения вывод картинки на печать будет осуществлен.

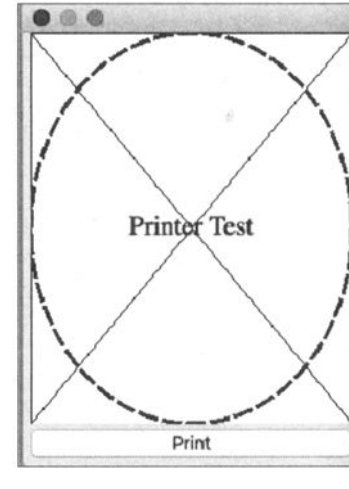


Рис. 24.1. Вывод на печать

В листинге 24.1 обратите внимание на опцию `QT` проектного файла — в ней мы указываем параметр `printsupport`. Это необходимо, чтобы во время компоновки программы подсоединить модуль `QtPrintSupport`.

Листинг 24.1. Проектный файл печати (файл Printer.pro)

```
TEMPLATE = app
QT          += widgets printsupport
HEADERS      = Printer.h
SOURCES      = Printer.cpp \
               main.cpp
TARGET = ../Printer
```

В программе, приведенной в листинге 24.2, создаются виджеты принтера (указатель `pprinter` на объект класса `Printer`, описанный в листингах 24.3–24.7) и кнопки (указатель `pcmd`). Сигнал `clicked()` виджета кнопки соединяется со слотом `slotPrint()` виджета принтера.

Листинг 24.2. Создание виджетов принтера и кнопки (файл main.cpp)

```
#include <QtWidgets>
#include "Printer.h"

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    Printer*     pprinter = new Printer;
    QPushButton* pcmd     = new QPushButton("&Print");

    QObject::connect(pcmd, SIGNAL(clicked()),
                     pprinter, SLOT(slotPrint())
                    );

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->setContentsMargins(0, 0, 0, 0);
    pvbxLayout->setSpacing(0);
    pvbxLayout->addWidget(pprinter);
    pvbxLayout->addWidget(pcmd);
    wgt.setLayout(pvbxLayout);

    wgt.resize(250, 320);
    wgt.show();

    return app.exec();
}
```

Определение класса `Printer`, приведенное в листинге 24.3, содержит указатель на объект принтера `QPrinter`. Метод `draw()` получает указатель на контекст рисования. В классе определен деструктор, в котором освобождается динамически выделяемая память для объекта принтера. Слот `slotPrint()` вызывается для выполнения вывода на печатающее устройство.

Листинг 24.3. Определение класса `Printer` (файл Printer.h)

```
#pragma once

#include <QWidget>

class QPrinter;
class QPaintDevice;

// ============================================================================
class Printer : public QWidget {
    Q_OBJECT

private:
    QPrinter* m_pprinter;

protected:
    virtual void paintEvent(QPaintEvent* pe  );
            void draw      (QPaintDevice* ppd);

public:
    Printer(QWidget* pwgt = 0);
    virtual ~Printer();

public slots:
    void slotPrint();
};
```

Как можно видеть из листинга 24.4, в конструкторе динамически создается объект принтера — указатель `m_pprinter`. Этот объект необходимо по завершении работы программы удалить, и лучше всего сделать это в деструкторе.

Листинг 24.4. Конструктор и деструктор (файл Printer.cpp)

```
// ----------------------------------------------------------------------
Printer::Printer(QWidget* pwgt/*=0*/) : QWidget(pwgt)
{
    m_pprinter = new QPrinter;
}

// ----------------------------------------------------------------------
/*virtual*/Printer::~Printer()
{
    delete m_pprinter;
}
```

В методе обработки события перерисовки, приведенном в листинге 24.5, вызывается метод `draw()` и в него, в качестве контекста рисования, передается указатель `this`.

Листинг 24.5. Метод `paintEvent()` (файл Printer.cpp)

```
/*virtual*/ void Printer::paintEvent(QPaintEvent* pe)
{
    draw(this);
}
```

В листинге 24.6 при создании диалогового окна (переменная `dlg`) в его конструктор передаются указатели на объект принтера (`m_pprinter`) и на виджет предка. Второй параметр нужен для того, чтобы диалоговое окно было отцентрировано относительно основного окна.

Методом `setMinMax()` устанавливается диапазон страниц, которые можно печатать. В нашем случае это всего одна страница, поэтому этот диапазон устанавливается от 1 до 1.

Вызов метода `exec()` откроет диалоговое окно, в котором пользователь может выбрать принтер и настроить опции печати. При нажатии на кнопку **OK** метод возвращает `QDialog::Accepted`. После этого объект `QPrinter` полностью готов к использованию, и его указатель `m_pprinter` передается методу `draw()`.

Вычисление количества страниц для печати

После того как в диалоговом окне будут выполнены все необходимые изменения, можно вычислить количество страниц для печати при помощи значений, возвращаемых методами `QAbstractPrintDialog::toPage()` и `QAbstractPrintDialog::fromPage()`, отняв от первого второе и прибавив единицу. Для нашего примера это выглядит следующим образом:

```
int nPages = dlg.toPage() - dlg.fromPage() + 1;
```

Получение следующего листа

Чтобы получить следующий лист для рисования и напечатать на нем, нужно вызвать метод `QPrinter::newPage()` и передать указатель `m_pprinter` в метод `draw()`.

Листинг 24.6. Метод `slotPrint()` (файл Printer.cpp)

```
void Printer::slotPrint()
{
    QPrintDialog dlg(m_pprinter, this);

    dlg.setMinMax(1, 1);
    if (dlg.exec() == QDialog::Accepted) {
        draw(m_pprinter);
    }
}
```

В метод `draw()`, приведенный в листинге 24.7, передаются указатели на контекст рисования. Подавляющее большинство принтеров не могут использовать всю площадь листа для печати. Поэтому, чтобы избежать вывода на недоступные для принтера места, нужно опросить прямоугольную область вывода, используя метод `QPainter::viewport()`. Исходя из полу-

ченных размеров, в контексте рисования с помощью методов `drawRect()`, `drawLine()`, `drawEllipse()` и `drawText()` отображаются соответственно прямоугольник, линии, эллипс и текст. Методами `setPen()` и `setBrush()` устанавливаются перья и кисти, имеющие различные цвета и образцы. Метод `setFont()` устанавливает шрифт для выводимого текста.

Листинг 24.7. Метод draw() (файл Printer.cpp)

```
void Printer::draw(QPaintDevice* ppd)
{
    QPainter painter(ppd);
    QRect    r(painter.viewport());

    painter.setBrush(Qt::white);
    painter.drawRect(r);
    painter.drawLine(0, 0, r.width(), r.height());
    painter.drawLine(r.width(), 0, 0, r.height());

    painter.setBrush(Qt::NoBrush);
    painter.setPen(QPen(Qt::red, 3, Qt::DashLine));
    painter.drawEllipse(r);

    painter.setPen(Qt::blue);
    painter.setFont(QFont("Times", 20, QFont::Normal));
    painter.drawText(r, Qt::AlignCenter, "Printer Test");
}
```

Резюме

Благодаря тому, что класс `QPrinter` унаследован от класса `QPaintDevice`, выводить информацию на печатающее устройство так же просто, как рисовать на экране. Объекты класса принтера `QPrinter` могут быть настроены пользователем при помощи специального диалогового окна или программно вызовом целого ряда методов. Класс принтера предоставляет также возможность создания файлов в формате PDF.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/24-510/** или с помощью следующего QR-кода (рис. 24.2):

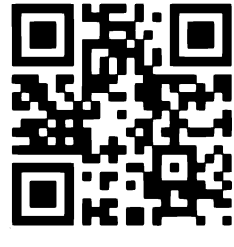


Рис. 24.2. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 25

Разработка собственных элементов управления

Карта — это не территория; имя — это не сам объект.
Альфред Коржибски

Создание собственных виджетов — задача несложная. Прежде всего нужно хорошо подумать над тем, какой из виджетов классовой иерархии Qt обладает наибольшим количеством необходимых вам свойств, чтобы использовать его в качестве базового. Это поможет существенно сэкономить время при разработке нового виджета.

Примеры создания виджетов

Допустим, нам нужен виджет текстового поля, способный принимать только числа в шестнадцатеричной системе счисления. Реализация такого виджета может ограничиться наследованием класса `QLineEdit` и определением конструктора нового виджета (листинг 25.1), в котором вызовом метода `setValidator()` создается и устанавливается объект контроллера (validator).

Листинг 25.1. Класс `HexLineEdit`

```
class HexLineEdit : public QLineEdit {
public:
    HexLineEdit(QWidget* pwgt = 0) : QLineEdit(pwgt)
    {
        QRegExp rxp("[0-9A-Fa-f]+");
        setValidator(new QRegExpValidator(rxp, this));
    }
};
```

Пример, приведенный в листинге 25.1, представляет собой частный случай. Рассмотрим список рекомендаций для общего случая — его пункты могут быть проигнорированы, если в них нет необходимости:

- переопределите методы обработки событий, на которые должен реагировать новый виджет. Это могут быть: `paintEvent()`, `resizeEvent()`, `mousePressEvent()`, `mouseReleaseEvent()` и др.;
- подумайте, будет ли создаваемый класс виджета наследоваться дальше, если да, то, вполне возможно, понадобится объявить некоторые из его атрибутов не как `private`, а как `protected`;

- подумайте и примите решение, какие сигналы будет отправлять виджет;
- решите, какие слоты будут определены в классе виджета;
- перезапишите методы `sizeHint()` и `sizePolicy()` — чтобы новый виджет без проблем мог использоваться компоновками.

> **Метод `setSizePolicy()`**
>
> Можно обойтись и без перезаписи метода `sizePolicy()`, вызвав метод `setSizePolicy()` и передав ему нужные значения из конструктора создаваемого класса.

Последний пункт списка нуждается в отдельном пояснении. Как вы уже знаете (см. *главу 6*), классы компоновки отвечают не только за расположение виджетов, но и управляют их размерами. Виджеты гарантированно будут иметь приемлемые размеры, если при их размещении используются значения, возвращаемые методами `sizeHint()` и `sizePolicy()`.

Метод `sizeHint()` возвращает объект класса `QSize`, который информирует о том, какие размеры необходимы виджету. Это зависит, прежде всего, от содержимого виджета. В листинге 25.2 создаются два виджета кнопок, имеющих надписи различной длины. Вызовы метода `sizeHint()` возвращают для каждой из созданных кнопок разные значения: вызов метода `pcmd1->sizeHint()` возвращает значение (75, 23), а `pcmd2->sizeHint()` — значение (145, 23).

Листинг 25.2. Вызов методов `sizeHint()`

```
QString str = "Button";
QPushButton* pcmd1 = new QPushButton(str);
QSize size = pcmd1->sizeHint(); // size = (75, 23)

str = "This is very long button label";
QPushButton* pcmd2 = new QPushButton(str);
size = pcmd2->sizeHint(); // size = (145, 23)
```

Метод `QWidget::sizePolicy()` возвращает объекты класса `QSizePolicy`, в которых содержится информация, влияющая на интерпретацию значения, возвращаемого методом `sizeHint()` при изменении размеров окна. Объект создается передачей в конструктор класса `QSizePolicy` двух флагов (для вертикального и горизонтального направлений), приведенных в табл. 25.1.

***Таблица 25.1.** Перечисление языка Си `SizeType` класса `QSizePolicy`*

Константа	Описание
`Fixed`	Должно учитываться только значение, возвращаемое методом `sizeHint()`, — другими словами, размер виджета изменять нельзя
`Minimum`	Виджет не должен быть меньше значения, возвращаемого методом `sizeHint()`
`Maximum`	Виджет не должен быть больше значения, возвращаемого методом `sizeHint()`
`Preferred`	Виджет может быть больше или меньше значения, возвращаемого методом `sizeHint()`, то есть виджет может как растягиваться, так и сжиматься
`MinimumExpanding`	Виджет не должен быть меньше чем значение, возвращаемое методом `sizeHint()`. Класс компоновки постарается предоставлять виджету как можно больше места

Таблица 25.1 *(окончание)*

Константа	Описание
`Expanding`	Виджет может быть больше или меньше значения, возвращаемого методом `sizeHint()`, но класс компоновки постарается предоставлять виджету как можно больше места. Другими словами, виджет может как растягиваться, так и сжиматься, но предпочтительнее его растягивать
`Ignored`	Значение, возвращаемое методом `sizeHint()`, не принимается во внимание. Однако класс компоновки постарается предоставлять виджету как можно больше места

Возьмем, например, виджет класса `QScrollView`. Невозможно заранее определить его размер. Благодаря полосам прокрутки с этим виджетом можно работать, даже если его размер будет меньше размера, возвращаемого методом `sizeHint()`. Поэтому значение, возвращаемое методом `sizeHint()`, должно приниматься во внимание как рекомендуемый размер. Но чем больше будет размер виджета `QScrollView`, тем удобнее будет им пользоваться. Для обеспечения подобного поведения в конструкторе этого класса вызывается метод `setSizePolicy()`, в который передаются два флага `Expanding` — для горизонтали и вертикали:

```
setSizePolicy(QSizePolicy(QSizePolicy::Expanding,QSizePolicy::Expanding);
```

В следующем примере (листинги 25.3–25.8) приведена программа, демонстрирующая создание собственного виджета индикатора выполнения (рис. 25.1). Само приложение состоит из индикатора выполнения и полосы прокрутки. Значение, отображаемое электронным индикатором, изменяется в зависимости от указателя текущего положения полосы прокрутки.

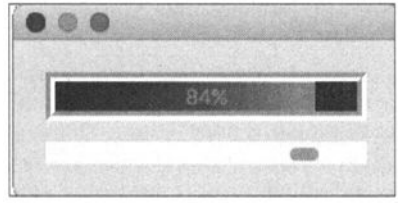


Рис. 25.1. Демонстрация собственного индикатора выполнения

В функции `main()` создаются виджеты разработанного нами индикатора выполнения — указатель `pcw`, и полосы прокрутки — указатель `phsb` (листинг 25.3). После этого сигнал `valueChanged(int)` полосы прокрутки при помощи метода `connect()` соединяется со слотом `slotSetProgress(int)`, служащим для отображения значений целого типа индикатора выполнения.

Листинг 25.3. Создание собственного виджета индикатора выполнения (файл main.cpp)

```
#include <QtWidgets>
#include "CustomWidget.h"

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;
```

```
    CustomWidget* pcw  = new CustomWidget;
    QScrollBar*   phsb = new QScrollBar(Qt::Horizontal);

    phsb->setRange(0, 100);

    QObject::connect(phsb, SIGNAL(valueChanged(int)),
                     pcw, SLOT(slotSetProgress(int))
                    );

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(pcw);
    pvbxLayout->addWidget(phsb);
    wgt.setLayout(pvbxLayout);

    wgt.show();

    return app.exec();
}
```

Определяемый в листинге 25.4 класс `CustomWidget` содержит целочисленный атрибут `m_nProgress`, необходимый для хранения текущего значения индикатора выполнения. Переопределяются методы `paintEvent()` и `sizeHint()`. Последний метод информирует компоновку о желаемом размере. В классе определяется сигнал `progressChanged(int)`, который будет отправляться каждый раз при изменении значения индикатора выполнения. Слот `slotSetProgress()` управляет установкой значения этого индикатора.

Листинг 25.4. Определение класса `CustomWidget` (файл CustomWidget.h)

```
#pragma once

#include <QFrame>

// ============================================================================
class CustomWidget : public QFrame {
    Q_OBJECT
protected:
    int m_nProgress;

    virtual void paintEvent(QPaintEvent*);

public:
    CustomWidget(QWidget* pwgt = 0);

    virtual QSize sizeHint() const;

signals:
    void progressChanged(int);
```

```
public slots:
    void slotSetProgress(int n);
};
```

В листинге 25.5 приведен конструктор класса. Методы `setLineWidth()` и `setFrameStyle()` устанавливают толщину и стиль рамки. Первый параметр: `QSizePolicy::Minimum`, передаваемый в метод `setSizePolicy()`, говорит классу компоновки о том, что ширина виджета не должна быть меньше значения, возвращаемого методом `sizeHint()`. Второй параметр: `QSizePolicy::Fixed` сообщает, что высота виджета должна быть равна значению, возвращаемому методом `sizeHint()`.

Листинг 25.5. Конструктор `CustomWidget` (файл CustomWidget.cpp)

```
CustomWidget::CustomWidget(QWidget* pwgt/*= 0*/) : QFrame(pwgt)
                                                 , m_nProgress(0)
{
    setLineWidth(3);
    setFrameStyle(Box | Sunken);

    setSizePolicy(QSizePolicy::Minimum, QSizePolicy::Fixed);
}
```

Метод `paintEvent()` осуществляет отображение индикатора выполнения (листинг 25.6). Объект `painter` инициализируется контекстом виджета. Вызов метода `fillRect()` закрашивает виджет в черный цвет. Линейный градиент и ширина прямоугольной области для заливки градиентом вычисляются в соответствии с текущим значением процесса (переменная `f`). По окончании рисования прямоугольной области устанавливается перо зеленого цвета. Строковой переменной `str` присваивается приведенное к строковому типу значение индикатора выполнения, после чего оно отображается с помощью метода `drawText()` в центре индикатора (флаг `AlignCenter`). Адрес `&painter` передается в метод `QFrame::drawFrame()` для того, чтобы класс `QFrame` смог отобразить свою рамку.

Листинг 25.6. Метод обработки события `paintEvent()` (файл CustomWidget.cpp)

```
/*virtual*/ void CustomWidget::paintEvent(QPaintEvent*)
{
    QPainter        painter(this);
    QLinearGradient gradient(0, 0, width(), height());
    float           f = m_nProgress / 100.0f;

    gradient.setColorAt(0, Qt::black);
    gradient.setColorAt(f, Qt::green);

    painter.fillRect(rect(), Qt::black);
    painter.setBrush(gradient);
    painter.drawRect(0, 0, (int)(width() * f), height());

    painter.setPen(QPen(Qt::green));
    QString str = QString().setNum(m_nProgress) + "%";
    painter.drawText(rect(), Qt::AlignCenter, str);

    drawFrame(&painter);
}
```

Приведенный в листинге 25.7 слот `slotSetProgress()` управляет установкой значения индикатора выполнения (атрибут `m_nProgress`) и следит за тем, чтобы значение индикатора не выходило за пределы диапазона: от 0 до 100. После присвоения значения методом `repaint()` осуществляется перерисовка и отправляется сигнал `progressChanged(int)` с его актуальным значением.

Листинг 25.7. Слот-метод `slotSetProgress()` (файл CustomWidget.cpp)

```
void CustomWidget::slotSetProgress(int n)
{
    m_nProgress = n > 100 ? 100 : n < 0 ? 0 : n;
    repaint();
    emit progressChanged(m_nProgress);
}
```

В листинге 25.8 метод `sizeHint()` информирует класс компоновки о размере, который желателен для виджета.

Листинг 25.8. Метод `sizeHint()` (файл CustomWidget.cpp)

```
/*virtual*/ QSize CustomWidget::sizeHint() const
{
    return QSize(200, 30);
}
```

Резюме

Унаследовав класс `QWidget` или его наследника, можно создавать свои элементы управления. Скорость реализации нового виджета во многом зависит от удачного подбора базового класса. При создании нового класса виджета важно помнить, какие методы событий необходимо перезаписать, а также какие сигналы и слоты будет предоставлять виджет.

Классы, базирующиеся на классе `QWidget`, предоставляют интерфейс оповещения о размере. На вершине этого интерфейса находится класс `QSizePolicy`, с помощью которого виджет может сообщить, желательно ли подвергать его уменьшению/увеличению, отдельно — по вертикали и горизонтали. С помощью метода `sizeHint()` класс компоновки узнает о размерах, необходимых виджету, и интерпретирует их в соответствии со значением объекта `QSizePolicy`.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/25-510/** или с помощью следующего QR-кода (рис. 25.2):



Рис. 25.2. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 26

Элементы со стилем

Встречают по одежке, а провожают по уму.
Народная мудрость

Один из немаловажных аспектов при программировании с использованием библиотеки Qt — это управление *видом* и *поведением* приложения (look & feel). Ввиду того, что Qt-приложения создаются под большое число платформ, для изменения их вида и поведения необходим соответствующий механизм. Его наличие позволяет добиться, чтобы внешний вид приложения «не выбивался из колеи» при запуске в какой-либо операционной системе, и не создавалось впечатление, что программа на этой платформе «чужая».

В Qt внешний вид компонентов может быть свободно изменен, и это ее свойство можно использовать, чтобы разнообразить внешний вид приложения, если вы относитесь к группе программистов, считающих стандартный вид неподходящим для своих программ. Например, при написании компьютерной игры вы наверняка захотите большей дизайнерской свободы и попробуете придать элементам управления особый вид. К изменению стиля можно прибегнуть и в том случае, когда нужно добиться, чтобы ваши программы узнавали по их внешнему виду. И это очень важно — потому что в первую очередь продается не сама программа, а ее внешний вид. К сожалению, многие программисты склонны недооценивать этот момент, хотя он и очевиден. Им кажется, что если программа обладает гениальным функциональным содержанием, то и продаваться она будет «на ура», а ее внешний вид — вопрос не первой значимости. Это не правильно, и, прежде всего, по той причине, что для того чтобы оценить гениальность функций вашей программы, у пользователя должно возникнуть желание познакомиться с ней, а если сама программа внешне не привлекательна, то у пользователя может и не возникнуть такого желания. Хоть народная мудрость «не все то золото, что блестит» и верна, но все-таки то, что блестит, безусловно, привлекает внимание окружающих, а это и есть первичная цель, — поэтому ваша программа должна выделяться из всех прочих своим красивым внешним видом. Те кто разделяет точку зрения, что внешний вид программы не важен, прежде всего должны спросить себя, какие машины им больше нравятся, — стильные и красивые или обычные. А с какой девушкой вы захотите познакомиться? С красивой? Или неказистой? Да, первая девушка может оказаться просто пустышкой, а вторая будет наполнена богатым внутренним содержанием. Но первое ваше устремление все равно обратится на красавицу...

В программной индустрии выбор программы пользователем осуществляется точно так же. Внешний вид очень важен, чтобы обратить на себя внимание, притянуть пользователя и вызвать в нем желание попробовать и саму программу. Функциональная сторона и содержание программы оцениваются пользователем уже на втором этапе знакомства, и вот тут-то

и будут востребованы гениальные возможности ее функциональных качеств, и если их не окажется, то пользователь, скорее всего, с такой программой расстанется. На этом месте хочется немного перефразировать слова Антона Павловича Чехова и заменить в его известном высказывании слово «человек» на слово «программа». Тогда получится «В программе должно быть все прекрасно: и лицо, и одежда, и душа, и мысли». Заметьте, что лицо и одежда, то есть внешний вид, стоят у классика на первом месте, и это неспроста.

В Qt можно создать классы виджетов, которые будут иметь свой собственный облик, отличный от стандартного. Но это неудобно, поскольку каждый раз придется реализовывать новые классы и изменять исходный код. Однако Qt предоставляет и специальные *классы стилей*, позволяющие изменять внешний вид и поведение для любых классов виджетов. Стили программы можно изменять даже в процессе ее работы, а это значит, что пользователю можно предоставить в меню целый список стилей, чтобы он смог выбрать оптимальный для себя. Стили можно создавать самому или использовать уже готовые, встроенные в библиотеку Qt.

Возможность реализации и использования классов стилей, не зависящих от кода программы, дает большую свободу, позволяющую разделить разработку проекта, как минимум, на две команды, которые могут работать независимо друг от друга. Одна команда будет работать над кодом самой программы, а другая — над ее дизайном. Созданные второй командой классы, или CSS-файлы, стилей могут использоваться и в других Qt-проектах.

В контексте Qt стиль — это класс, унаследованный от класса `QStyle` и реализующий возможности для рисования рамок, кнопок, растровых изображений и т. п. Qt делает каждый виджет ответственным за свое отображение, что соответственно повышает скорость отображения и его гибкость.

На рис. 26.1 показана иерархия классов стилей. Класс `QStyle` является базовым для всех стилей.

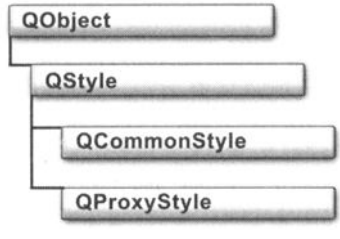


Рис. 26.1. Иерархия классов стилей

Поддержка языков с написанием справа налево

Класс `QStyle` обладает поддержкой языков с написанием справа налево: пользователю необходимо запустить приложение, указав в командной строке опцию `-reverse`.

Этот класс определяет целый ряд методов для изменения внешнего вида и поведения всех виджетов приложения. Для создания своего собственного стиля можно переопределить эти методы. Класс `QStyle` имеет только одного потомка — класс `QCommonStyle`, который является общим для всех классов стилей, определяя часто используемые методы. Поэтому предпочтительнее наследовать именно этот класс, а не `QStyle`.

Непосредственно от класса `QCommonStyle` наследуется классы, относящиеся к *встроенным стилям* Qt.

Встроенные стили

По умолчанию вид и поведение Qt-приложения определяется операционной системой, в которой оно запущено. Но это можно изменить и использовать — например, в ОС Windows Qt предоставляет следующие классы встроенных стилей (рис. 26.1): `QWindowsStyle` и `QFusionStyle`. Эти стили эмулируются и доступны на любых платформах — кроме стилей `QWindowsXPStyle`, `QWindowsVistaStyle`, `QWindowsMobileStyle`, `QGtkStyle`, `QAndroidStyle`, `QWindowsCEStyle` и `QMacStyle`, которые доступны только на «родных» платформах.

Стиль можно установить, вызвав любое Qt-приложение с командной опцией `-style`, или же программно. Для того чтобы установить стиль в приложении, нужно вызвать метод `setStyle()` и передать ему объект стиля или строку с его названием:

```
QApplication::setStyle("QFusionStyle");
```

или

```
QApplication::setStyle(QStyleFactory::create("QFusionStyle"));
```

В последнем случае вызывается статический метод `QApplication::setStyle()`, которому вызовом статического метода `QStyleFactory::create()` передается динамически созданный объект стиля. Классы стилей наследуются от класса `QObject`, не используя механизм объектной иерархии, и ответственность за уничтожение созданных объектов несет класс `QApplication`. Передача объекта стиля в метод `setStyle()` приводит к тому, что этот метод сначала с помощью оператора `delete` уничтожает старый объект стиля. Поэтому можно создавать объекты стиля непосредственно в самом методе `setStyle()`, как это показано в листинге 26.1, и не создавать дополнительные указатели на эти объекты.

В Qt можно смешивать различные стили, то есть задавать каждому из виджетов в отдельности свой собственный стиль. Этого делать не рекомендуется, но, тем не менее, в целях демонстрации, чтобы показать различные стили, мы это правило проигнорируем (рис. 26.2).

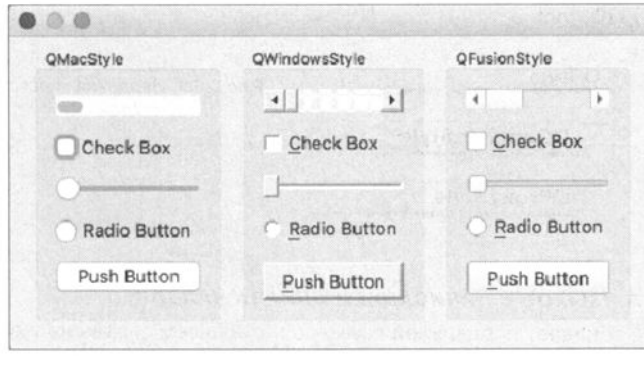


Рис. 26.2. Пример отображения виджетов с различными стилями

В функции `main()` листинга 26.1 создается виджет `wgt` для горизонтального размещения виджетов. Мы по списку всех доступных нам стилей организуем цикл `foreach`, который возвращает статический метод `QStyleFactory::keys()`. В функцию `styledWidget()` передается указатель на объект стиля, который создает виджет и возвращает указатель этого виджета. В самой функции создается виджет группы (указатель `pgr`), в конструктор которого при помощи метода `className()` передается имя класса стиля, что возможно, так как базовый класс стилей унаследован от класса `QObject`. Затем в созданном виджете группы (`pgr`)

посредством вертикальной компоновки (pvbxLayout) размещаются остальные создаваемые виджеты. Наконец, вызов метода findChildren<T>() формирует список всех потомков виджета pgr, и в цикле foreach устанавливается стиль для каждого виджета с помощью метода setStyle(), в который передается указатель на объект стиля.

Листинг 26.1. Создание объектов стиля (файл main.cpp)

```
#include <QtWidgets>

// ----------------------------------------------------------------------
QWidget* styledWidget(QStyle* pstyle)
{
    QGroupBox*    pgr = new QGroupBox(pstyle->metaObject()->className());
    QScrollBar*   psbr = new QScrollBar(Qt::Horizontal);
    QCheckBox*    pchk = new QCheckBox("&Check Box");
    QSlider*      psld = new QSlider(Qt::Horizontal);
    QRadioButton* prad = new QRadioButton("&Radio Button");
    QPushButton*  pcmd = new QPushButton("&Push Button");
    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(psbr);
    pvbxLayout->addWidget(pchk);
    pvbxLayout->addWidget(psld);
    pvbxLayout->addWidget(prad);
    pvbxLayout->addWidget(pcmd);
    pgr->setLayout(pvbxLayout);

    QList<QWidget*> pwgtList = pgr->findChildren<QWidget*>();
    foreach(QWidget* pwgt, pwgtList) {
        pwgt->setStyle(pstyle);
    }
    return pgr;
}

// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    //Layout setup
    QHBoxLayout* phbxLayout = new QHBoxLayout;
    foreach (QString str, QStyleFactory::keys()) {
        phbxLayout->addWidget(styledWidget(QStyleFactory::create(str)));
    }
    wgt.setLayout(phbxLayout);

    wgt.show();

    return app.exec();
}
```

В следующем примере (листинги 26.2–26.4) стиль изменяется для всего приложения. Выбрать нужный стиль можно при помощи выпадающего списка (рис. 26.3).



Рис. 26.3. Изменение стиля для всего приложения

В листинге 26.2 создается виджет класса `MyApplication`. Определение этого класса приведено в листинге 26.3, а реализация — в листинге 26.4.

Листинг 26.2. Изменение стиля для всего приложения (файл main.cpp)

```
#include <QApplication>
#include "MyApplication.h"

int main (int argc, char** argv)
{
    QApplication  app(argc, argv);
    MyApplication myApplication;

    myApplication.show();

    return app.exec();
}
```

Класс `MyApplication` (листинг 26.3) наследуется от класса `QWidget`. Помимо конструктора, в классе реализуется слот `slotChangeStyle(const QString&)`, который будет соединен с виджетом выпадающего списка.

Листинг 26.3. Определение класса `MyApplication` (файл MyApplication.h)

```
#pragma once

#include <QWidget>

// ============================================================================
class MyApplication : public QWidget {
    Q_OBJECT
public:
    MyApplication(QWidget* pwgt = 0);

public slots:
    void slotChangeStyle(const QString& str);
};
```

В листинге 26.4 в конструкторе класса `MyApplication` создается виджет выпадающего списка, в который методом `addItems()` добавляются имена всех доступных нам стилей из списка, возвращаемого статическим методом `QStyleFactory::keys()`. После этого метод `connect()` соединяет сигнал `activated(const QString&)` со слотом `slotChangeStyle(const QString&)`. При выборе нового элемента списка слот получает строку со стилем, выбранную пользователем. Она передается в статический метод `QStyleFactory::create()`, который создает соответствующий объект стиля. Далее указатель на объект стиля передается в статический метод `setStyle()`, который осуществляет установку стиля для всего приложения.

Листинг 26.4. Конструктор `MyApplication()` (файл MyApplication.cpp)

```
#include <QtGui>
#include "MyApplication.h"

// -----------------------------------------------------------------------
MyApplication::MyApplication(QWidget* pwgt/*= 0*/) : QWidget(pwgt)
{
    QComboBox*    pcbo = new QComboBox;
    QSpinBox*     pspb = new QSpinBox;
    QSlider*      psld = new QSlider(Qt::Horizontal);
    QRadioButton* prad = new QRadioButton("&Radio Button");
    QCheckBox*    pchk = new QCheckBox("&Check Box");
    QPushButton*  pcmd = new QPushButton("&Push Button");

    pcbo->addItems(QStyleFactory::keys());

    connect(pcbo,
            SIGNAL(activated(const QString&)),
            SLOT(slotChangeStyle(const QString&))
           );

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(pcbo);
    pvbxLayout->addWidget(pspb);
    pvbxLayout->addWidget(psld);
    pvbxLayout->addWidget(prad);
    pvbxLayout->addWidget(pchk);
    pvbxLayout->addWidget(pcmd);
    setLayout(pvbxLayout);
}

// -----------------------------------------------------------------------
void MyApplication::slotChangeStyle(const QString& str)
{
    QStyle* pstyle = QStyleFactory::create(str);
    QApplication::setStyle(pstyle);
}
```

Создание собственных стилей

Каждый виджет имеет указатель на объект стиля `QStyle`. Его можно получить вызовом метода `QWidget::style()`. Виджеты вызывают методы `QStyle` в различных ситуациях: при наступлении событий мыши, при событиях перерисовки, при вызове метода `sizeHint()` менеджером компоновки и т. д.

Собственные стили можно создать наследованием встроенного стиля Qt и перезаписью некоторых методов либо при помощи CSS (каскадного стиля документа), который описан в этой главе далее. Если вы собираетесь реализовать свой собственный стиль первым способом, то, прежде чем это сделать, нужно решить, какой из классов стилей унаследовать. Можно унаследовать и сам класс `QStyle`, переопределив необходимые методы. Но это потребует гораздо больше усилий, чем при наследовании класса, уже обладающего необходимыми свойствами. Воспользовавшись подобным классом, вам потребуется перезаписать лишь несколько методов, в которых будут реализованы все необходимые различия.

Класс `QStyle` — это абстрактный класс, который является интерфейсом для реализации стилей. Для рисования элементов управления вам, в основном, придется иметь дело со следующими методами, определяемыми классом `QStyle`:

- `drawPrimitive()` — предназначен для рисования простых элементов управления;
- `drawControl()` — служит для рисования элементов управления;
- `drawComplexControl()` — обрабатывает рисование составных элементов управления.

Каждый из указанных ранее методов для рисования принимает специальный идентификатор, который сообщает о том, что должно быть сделано. Для передачи дополнительной информации предусмотрен параметр `QStyleOption`. Все данные этого класса определены как `public`, то есть можно напрямую обращаться к атрибутам:

- `version` (версия) — номер версии. Если вы наследуете класс `QStyleOption` или унаследованный от него класс, номер следует увеличить на единицу;
- `type` (тип) — целочисленный идентификатор типа;
- `state` (статус) — содержит информацию о состоянии виджета, например: доступен (`enabled`), активен (`active`), в нажатом состоянии (`pressed`) и т. п.;
- `direction` (расположение) — расположение текста, по умолчанию это значение равно `Qt::LeftToRight`, но может принимать и значение `Qt::RightToLeft`;
- `rect` (сокр. от слова прямоугольник) — прямоугольная область, необходимая для рисования элемента;
- `fontMetrics` (шрифт) — информация о шрифте;
- `palette` (палитра) — информация о палитре.

В большинстве случаев вам не понадобится больше информации, чем предоставляется классом `QStyleOption`. Но если вы создаете не только свой собственный стиль, но и свои собственные виджеты, то, унаследовав класс `QStyleOption`, сможете передавать любую дополнительную информацию, необходимую для ваших виджетов. Добавьте в унаследованный класс необходимые атрибуты и не забудьте позаботиться о типе и версии. Например:

```
class MyStyleOptionProgress : public QStyleOption {
    enum {Type = SO_ProgressBar};
    enum {Version = 1};
```

```
    int nMaximum;
    int nMinimum;
    int nProgress;
    QString str;
    Qt::Alignment textAlignment;
};
```

Теперь, когда вы в общих чертах познакомились с параметрами, самое время перейти к рассмотрению самих методов для рисования элементов управления.

Метод рисования простых элементов управления

В группу простых элементов управления входят индикаторы, флажки, рамки и другие подобные им элементы.

Для изменения их внешнего вида нужно переопределить метод `drawPrimitive()`:

```
void drawPrimitive(PrimitiveElement    elem,
                   const QStyleOption* popt,
                   QPainter*           ppainter,
                   const QWidget*      pwgt = 0
                  )
```

Первый аргумент — это значение, которое сообщает, с каким простым элементом мы будем иметь дело. Второй — это объект класса `QStyleOption`, содержащий в себе дополнительную информацию для стиля. Третий параметр — это указатель на объект `QPainter`, необходимый для рисования элемента. Четвертый параметр является необязательным, по умолчанию он равен нулю, но иногда содержит указатель на виджет, который может оказаться полезным при рисовании.

Метод рисования элементов управления

В группу элементов управления входят такие виджеты, как, например, кнопка и индикатор выполнения.

Для изменения их внешнего вида необходимо перегрузить метод `drawControl()`:

```
void drawControl(ControlElement      elem,
                 const QStyleOption* popt,
                 QPainter*           ppainter,
                 const QWidget*      pwgt = 0
                )
```

Первый аргумент — это значение, сообщающее о том, какой элемент управления должен быть нарисован. Три последних аргумента аналогичны аргументам метода `drawPrimitive()`.

Метод рисования составных элементов управления

В эту группу входят элементы управления, состоящие из нескольких частей, — например, полоса прокрутки, выпадающий список, виджет счетчика и т. п.

Для отображения их внешнего вида необходимо реализовать метод `drawComplexControl()`:

```
void drawComplexControl(ComplexControl             control,
                        const QStyleOptionComplex* popt,
                        QPainter*                  ppainter,
                        const QWidget*             pwgt = 0
                       )
```

Первый аргумент — это значение составного элемента. Вторым параметром метод получает указатель не на `QStyleOption`, как предыдущие два рассмотренных метода, а на `QStyleOptionComplex`. Этот класс унаследован от `QStyleOption` и дополняется информацией о подэлементах, которые должны рисоваться, и об активных подэлементах, то есть о тех, которые находятся непосредственно под указателем мыши или на которых был выполнен щелчок мышью.

Реализация стиля простого элемента управления

Очень распространенным элементом управления, который может присутствовать во многих составных элементах управления, — таких как, например, виджет выпадающего списка `QComboBox`, — является кнопка. Поэтому прямоугольную область для кнопки целесообразно рисовать в методе рисования простых элементов — чтобы любой элемент управления, нуждающийся в рисовании кнопки, мог ею воспользоваться. Пример, рассмотренный в листингах 26.5–26.9, демонстрирует стиль, изменяющий вид кнопки (рис. 26.4).

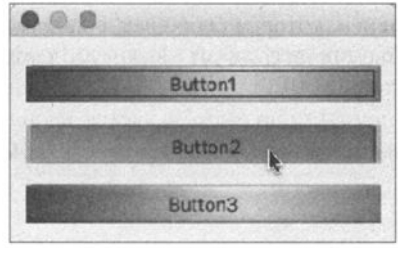


Рис. 26.4. Реализация стиля кнопки: первая кнопка находится в текущем, вторая в нажатом и третья в обычном состоянии

В функции `main()`, приведенной в листинге 26.5, реализуются три виджета кнопок, созданные от класса `QPushButton`. Вызов статического метода `setStyle()` устанавливает стиль `CustomStyle`.

Листинг 26.5. Стиль, изменяющий вид кнопки (файл main.cpp)

```
#include <QtWidgets>
#include "CustomStyle.h"

int main (int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QPushButton* pcmdA = new QPushButton ("Button1");
    QPushButton* pcmdB = new QPushButton ("Button2");
    QPushButton* pcmdC = new QPushButton ("Button3");
```

```
    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(pcmdA);
    pvbxLayout->addWidget(pcmdB);
    pvbxLayout->addWidget(pcmdC);
    wgt.setLayout(pvbxLayout);

    app.setStyle(new CustomStyle);
    wgt.show();

    return app.exec();
}
```

В листинге 26.6 класс стиля `CustomStyle` наследуется от класса `QCommonStyle`. При этом переопределяются методы `polish()` и `unpolish()`, которые присутствуют в классе `QStyle`. Не все стили нуждаются в переопределении указанных методов. В нашем случае это необходимо для того, чтобы изменять внешний вид кнопки при каждом появлении указателя мыши в ее области.

Также в унаследованном классе мы переписываем метод `drawPrimitive()`, предназначенный для отображения простых элементов управления (primitive control element). Первый параметр: `elem` — содержит в себе идентификатор элемента, который должен быть нарисован. Второй аргумент — это указатель на объект класса `QStyleOption`. Третий параметр — это объект `QPainter` (указатель `ppainter`), который будет использоваться для рисования элемента. Параметр `pwgt` представляет собой указатель на виджет, который может быть преобразован к нужному типу виджета.

Листинг 26.6. Определение класса `CustomStyle` (файл CustomStyle.h)

```
#pragma once

#include <QtWidgets>

class QPainter;

// ============================================================================
class CustomStyle : public QCommonStyle {
public:
    virtual void polish  (QWidget* pwgt);
    virtual void unpolish(QWidget* pwgt);

    virtual void drawPrimitive(      PrimitiveElement elem,
                               const QStyleOption*    popt,
                                     QPainter*        ppainter,
                               const QWidget*         pwgt = 0
                              ) const;
};
```

Метод `polish()` вызывается только один раз при каждом рисовании виджета (листинг 26.7). Мы переопределили его для того, чтобы присвоить атрибуту `Qt::WA_Hover` кнопки

QPushButton значение true. Это нужно, чтобы при каждом попадании указателя мыши в область кнопки Qt создавала событие перерисовки. В таком случае мы можем придать кнопке немного иной вид.

Листинг 26.7. Метод polish() (файл CustomStyle.cpp)

```
void CustomStyle::polish(QWidget* pwgt)
{
    if (qobject_cast<QPushButton*>(pwgt)) {
        pwgt->setAttribute(Qt::WA_Hover, true);
    }
}
```

Метод unpolish() должен осуществлять отмену всех модификаций, сделанных методом polish(). В нашем случае для простоты мы ограничились присвоением атрибуту Qt::WA_Hover значения false (листинг 26.8).

Листинг 26.8. Метод unpolish() (файл CustomStyle.cpp)

```
void CustomStyle::unpolish(QWidget* pwgt)
{
    if (qobject_cast<QPushButton*>(pwgt)) {
        pwgt->setAttribute(Qt::WA_Hover, false);
    }
}
```

В самом методе (листинг 26.9) используется переключатель switch. В нем задается секция case, в которой выясняется элемент, который нам предстоит отобразить. В нашем примере это кнопка.

Класс QStyleOption содержит все, что нужно знать о виджете для того, чтобы нарисовать его. Для получения специфической информации, присущей кнопкам, мы приводим его к типу QStyleOptionButton. Приведенный указатель объекта задействуем для получения информации о состоянии и размере прямоугольной области виджета, чтобы правильно его отобразить.

В зависимости от переменной bDown мы используем различные растровые изображения: для отображения нажатой кнопки — btnprs.bmp, а для кнопки в нормальном состоянии — btn.bmp. Изображение кнопки рисуется при помощи метода QPainter::drawPixmap().

Переменная bHover принимает значение true, когда указатель попадает в область кнопки. В этом случае с помощью метода QPainter::fillRect() мы рисуем поверх кнопки заполненный прямоугольник с таким уровнем прозрачности, который затемняет кнопку.

Последний параметр pwgt, передаваемый в метод, — это указатель на виджет, с которым мы работаем. Как вы уже успели заметить, мы не использовали его вообще, так как практически вся необходимая нам информация содержится в объекте QStyleOption. Но если вам когда-нибудь придется воспользоваться указателем на виджет pwgt, то не забывайте, что его передача не является обязательной, а это значит, необходимо проконтролировать, что этот указатель не равен нулю.

Обычно, когда создается стиль, изменяются только некоторые элементы. Совсем необязательно обрабатывать все случаи и изменять абсолютно все. Поэтому в секции `default` для обработки всех остальных случаев вызывается метод унаследованного класса.

Листинг 26.9. Метод drawPrimitive() (файл CustomStyle.cpp)

```
void CustomStyle::drawPrimitive(      PrimitiveElement elem,
                                const QStyleOption*    popt,
                                      QPainter*        ppainter,
                                const QWidget*         pwgt
                               ) const
{
    switch (elem) {
    case PE_PanelButtonCommand:
    {
        const QStyleOptionButton* pOptionButton =
            qstyleoption_cast<const QStyleOptionButton*>(popt);
        if (pOptionButton) {
            bool bDown = (pOptionButton->state & State_Sunken)
                       || (pOptionButton->state & State_On);

            QPixmap pix = bDown ? QPixmap(":/images/btnprs.bmp")
                                : QPixmap(":/images/btn.bmp");

            ppainter->drawPixmap(pOptionButton->rect, pix);

            bool bHover = (pOptionButton->state & State_Enabled)
                        && (pOptionButton->state & State_MouseOver);
            if (bHover) {
                ppainter->fillRect(pOptionButton->rect,
                                   QColor(25, 97, 45, 83)
                                  );
            }
        }
        break;
    }

    default:
        QCommonStyle::drawPrimitive(elem, popt, ppainter, pwgt);
        break;
    }
    return;
}
```

Использование каскадных стилей документа

Реализация собственного стиля может оказаться очень сложным, кропотливым и утомительным занятием. Кроме того, многие графические дизайнеры не знают язык C++. Интересно также отметить, что подавляющее большинство программистов на C++ не занимают-

ся графическим дизайном, а следовательно, C++ — это не тот язык, на котором должен реализовываться стиль программы. Очень важно еще и то обстоятельство, что внесение изменений в стиль, реализованный с помощью языка C++, требует перекомпиляции программы, а это заметно снижает скорость разработки программных проектов. Подобных трудностей можно было бы избежать, если бы стиль являлся отдельным файлом и загружался в процессе исполнения самой программы. Упомянутые здесь причины и послужили толчком для внедрения в Qt нового способа создания стиля. За основу был взят язык CSS (Cascading Style Sheets, каскадные таблицы стилей), используемый в HTML. В Qt этот язык адаптировали для виджетов, но концепция и синтаксис языка остались теми же, что позволило полностью отделить создание стиля от приложения. Программа Qt Designer (см. *главу 44*) предоставляет возможность интеграции CSS-стилей, что упрощает их просмотр на примерах различных виджетов.

По своей сути, каскадный стиль есть не что иное, как текстовое описание стиля. Это описание может находиться в отдельном файле, обычно имеющим расширение **qss**. Его можно установить в приложении, используя метод `QApplication::setStyleSheet()`, или в отдельном виджете при помощи аналогичного метода `QWidget::setStyleSheet()`. А можно подключить каскадный стиль в командной строке, указав после ключа `-stylesheet` имя файла CSS-стиля:

```
MyApp -stylesheet MyStyle.qss
```

Это очень удобно, так как можно очень быстро опробовать стиль в действии на любой Qt-программе.

Основные положения

Типичный синтаксис параметров и целевых элементов CSS выглядит следующим образом:

```
селектор {свойство: значение}
```

Целевой элемент называется *селектором*. Содержимое фигурных скобок, идущих следом за названием виджета, называется *определением селектора*. Селектор указывает, для какого из виджетов будет использоваться определение. Например, следующая строка устанавливает красный цвет текста для виджета однострочного текстового ввода:

```
QLineEdit {color: red}
```

Определение состоит из свойств и значений, а если их несколько, то они отделяются друг от друга точкой с запятой, например:

```
QLabel {
    color: red;
    background-color: white;
}
```

Цвет можно задавать названием в формате RGB или в HTML-стиле:

```
QLabel {
    color: rgb(255, 0, 0);
    background-color: #FFFFFF;
}
```

При помощи свойства `background-image` мы можем задать растровое изображение для заднего фона виджета. Например:

```
QLabel {
    background-image: url(pic.png);
}
```

Если в разных селекторах используются одинаковые определения, то их можно сгруппировать в единую директиву. Для этого элементы перечисляются через запятую, после чего указывается определение:

```
QPushButton, QLineEdit, QLabel {color: red}
```

Указание имени виджета означает, что определения будут применены также и к унаследованным от них классам. Для того чтобы исключить унаследованные классы, нужно поставить перед его именем точку. Например:

```
.QPushButton {color: red}
```

Для применения определений сразу ко всем виджетам приложения нужно поставить вместо имен звездочку (*). Например: `*{color: black};`

Можно ограничить применение стиля к виджетам с определенным именем объекта, установленным вызовом метода `setObjectName()`, — так, например, `QLabel#MyLabel` предписывает задействовать описанный далее стиль только у виджетов `QLabel` с именем объекта `MyLabel`. Из своего опыта скажу, что в этом свойстве необходимость появляется очень часто.

Если нужно, чтобы стиль оказал воздействие только на дочерние виджеты какого-либо виджета родителя, то сначала нужно указать класс виджета родителя, а затем через пробел класс дочернего виджета. Например, сделаем так, чтобы кнопки имели красный фон только у диалоговых окон:

```
QDialog QPushButton {
    background: red;
}
```

Если вы хотите применить стиль к собственному классу, который находится в пространстве имен, например:

```
namespace MyNameSpace {
    class MyClass : public QWidget {
    ...
    };
}
```

то к нему в CSS можно обратиться и изменить его внешний вид следующим образом:

```
MyNameSpace—MyClass {
    background-colode: black;
    color: white;
}
```

Каскадный стиль не чувствителен к регистру, например: `COLOR` = `color` = `CoLoR` и т. д. Исключение составляют лишь имена классов.

Комментарии в стандарте CSS помещаются, как и в языке C/C++, внутрь последовательности символов `/* */`, например:

```
/* комментарий */
```

Интересно еще то, что свойства виджетов можно изменять прямо из самого стиля. Так, например, вместо того чтобы в коде C++ программы вызвать из виджета класса `QListView` метод `setSelectionRectVisible(true)` или метод `setProperty("selectionRectVisible", true)`, можно это сделать непосредственно в самом стиле следующим образом:

```
QListView{qproperty-selectionRectVisible: true;}
```

Причем этот вызов окажет воздействие сразу на все виджеты класса `QListView` программы.

Изменение подэлементов

Многие виджеты формируются из составных элементов, так называемых *подэлементов*. Для задания стиля таких виджетов необходимо получить доступ к подэлементам. Делается это добавлением классификатора подэлемента после имени класса. Например, для изменения кнопки со стрелкой элемента выпадающего списка `QComboBox` нужно поступить следующим образом:

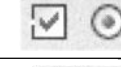
```
QComboBox::drop-down {image: url(pic.png)}
```

или, например, кнопка может иметь меню:

```
QPushButton::menu-indicator {image: url(downarrow.png)}
```

В табл. 26.1 сведены некоторые из самых распространенных подэлементов.

Таблица 26.1. *Некоторые подэлементы*

Подэлемент	Описание	Возможные виды
`::down-arrow`	Стрелка вниз. Имеется, например, у виджета выпадающего списка и у счетчика	
`::down-button`	Кнопка вниз. Имеется у виджета счетчика	
`::drop-down`	Стрелка виджета выпадающего списка	
`::indicator`	Индикатор кнопки флажка или переключателя, а также группировки кнопок	
`::item`	Элемент меню, строки состояния	
`::menu-indicator`	Индикатор меню кнопки нажатия, обычно это стрелка	
`::title`	Надпись группы	
`::up-arrow`	Стрелка вверх. Имеется у виджета счетчика	
`::up-button`	Кнопка вверх. Имеется у виджета счетчика	

Стили подэлементов управляются так же, как и стили элементов. Например:

```
QPushButton::menu-indicator:hover{image: url(hovereddownarrow.png)}
```

Размещение подэлементов выполняется при помощи subcontrol-position. Например, для того чтобы разместить подэлемент по центру слева, нужно сделать следующее:

```
QPushButton::menu-indicator {
    subconrol-position: left center;
    image: url(left_arrow.png);
}
```

Управление состояниями

Помимо подэлементов можно указывать состояния (табл. 26.2). При этом определение применяется к виджету только в том случае, если он находится в определенном (указанном) состоянии. Например, следующее правило будет применяться в том случае, если указатель мыши будет находиться над кнопкой:

```
QPushButton:hover {color: red}
```

Таблица 26.2. *Некоторые состояния*

Обозначение	Описание
:checked	Активировано
:closed	Виджет находится в закрытом либо свернутом состоянии
:disabled	Виджет недоступен
:enabled	Виджет доступен
:focus	Виджет находится в фокусе ввода
:hover	Указатель мыши находится над виджетом
:indeterminate	Кнопка находится в промежуточном неопределенном состоянии
:off	Выключено (для виджетов, которые могут быть в фиксированном состоянии нажато/не нажато)
:on	Включено (для виджетов, которые могут быть в фиксированном состоянии нажато/не нажато)
:open	Виджет находится в открытом или развернутом состоянии
:pressed	Виджет был нажат мышью
:unchecked	Деактивировано

Состояния можно объединять. Следующий пример говорит о том, что правило должно применяться тогда, когда указатель мыши располагается поверх виджета, находящегося в активированном состоянии:

```
QCheckBox:hover:checked {color: white}
```

Если необходимо сделать так, чтобы правило срабатывало в тех случаях, когда виджет находится в одном из нескольких состояний, то можно поступить следующим образом:

```
QCheckBox:hover, QCheckBox:checked {color: white}
```

Также можно применять правило, когда состояние не имеет места, для этого нужно использовать перед состоянием знак отрицания:

```
QCheckBox:!hover {color: white}
```

Пример

Как принято говорить, все познается в сравнении. Поэтому, для того чтобы нам было с чем сравнивать, давайте создадим стиль, идентичный созданному нами ранее (см. рис. 26.4).

Реализация нашего стиля приведена в листинге 26.10. Первый селектор устанавливает минимальную ширину кнопки, равную 80 пикселам. Второй селектор описывает кнопку в обычном состоянии, третий — при наведенной на него мыши, четвертый — в нажатом. В этих селекторах при помощи свойств `border-image` и `border-width` устанавливаются толщины границы элемента и растрового изображения равными пяти пикселам, а также задаются сами изображения, находящиеся в ресурсе.

Листинг 26.10. Реализация стиля (файл simple.qss)

```
/* Simple Style */
QPushButton {
    min-width: 80px;
}

QPushButton {
    border-image: url(:/style/btn.bmp) 5px;
    border-width: 5px;
}

QPushButton:hover {
    border-image: url(:/style/btnhvd.bmp) 5px;
    border-width: 5px;
}

QPushButton:pressed {
    border-image: url(:/style/btnprs.bmp) 5px;
    border-width: 5px;
}
```

В листинге 26.11, как и в листинге 26.5, мы создаем три виджета кнопок, которые при помощи класса компоновки размещаются вертикально. Основные различия заключаются в том, что мы загружаем стиль из файла, записываем его содержимое в строковую переменную `strCSS` и вызовом `QApplicaiton::setStyleSheet()` применяем его в приложении.

Для более простых случаев — например, чтобы сменить цвет фона кнопок при попадании на них указателя мыши, мы могли бы ограничиться только одной строкой кода:

```
qApp->setStyleSheet("QPushButton::hover {background-color: blue}");
```

При реализации стилей возможно так же применять различные градиенты — возьмем, например, линейный градиент и изменим внешний вид все той же кнопки (рис. 26.5).

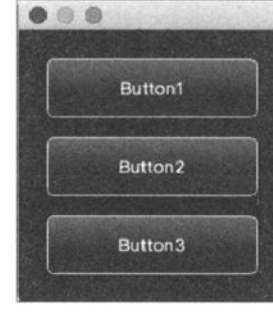


Рис. 26.5. Кнопки с градиентом

Листинг 26.11. Создание виджетов кнопок (файл main.cpp)

```
#include <QtWidgets>

int main (int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget      wgt;

    QPushButton* pcmdA = new QPushButton ("Button1");
    QPushButton* pcmdB = new QPushButton ("Button2");
    QPushButton* pcmdC = new QPushButton ("Button3");

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(pcmdA);
    pvbxLayout->addWidget(pcmdB);
    pvbxLayout->addWidget(pcmdC);
    wgt.setLayout(pvbxLayout);

    QFile file(":/style/simple.qss");
    file.open(QFile::ReadOnly);
    QString strCSS = QLatin1String(file.readAll());

    qApp->setStyleSheet(strCSS);
    wgt.show();

    return app.exec();
}
```

В первых трех строках листинга 26.12 мы изменяем цвет фона для всех виджетов приложения на черный. Затем задаем минимальную ширину кнопки, равную 150 пикселам (свойство `min-width`), а минимальную высоту — 40 пикселам. Затем, используя свойство `background`, задаем линейный градиент. Для этого устанавливаем первую (`x1`, `y1`) и вторую (`x2`, `y2`) точки контроля, а также две точки остановки: `1` и `0.4`. В результате получим вертикальный градиент, идущий от светло-серого к темно-серому цвету. Для того чтобы текст кнопки читался, задаем белый цвет для текста (свойство `color`). А чтобы кнопка смотрелась красивее, зададим ей наружный контур толщиной в один пиксел белого цвета (свойство `border`) и закруглим его углы (свойство `border-radius`).

Листинг 26.12. Изменение основного вида

```
QWidget {
    background: black;
}

QPushButton {
    min-width: 150px;
    min-height: 40px;
```

```
    background: qlineargradient(x1:0, y1:1, x2:0, y2:0,
                                stop:1 rgb(133,133,135),
                                stop:0.4 rgb(31, 31, 33) );
    color: white;
    border: 1px solid white;
    border-radius:5px;
}
```

В листинге 26.13 для состояния, когда мышь находится поверх кнопки, задаем градиент при помощи трех точек остановки. Заметьте, что цвета первых двух остались идентичными листингу 26.13, а последняя точка задает синий цвет. Это создает эффект голубого подсвечивания.

Листинг 26.13. Изменение состояния при подведении указателя

```
QPushButton:hover {
    background: qlineargradient(x1:0, y1:1, x2:0, y2:0,
                                stop:1 rgb(133,133,135),
                                stop:0.4 rgb(31, 31, 33),
                                stop:0.2 rgb(0, 0, 150) );
}
```

В случае, когда кнопка нажата (листинг 26.14), мы создаем горизонтальный градиент при помощи трех точек остановки. Это делается для того, чтобы пользователь сразу заметил разницу в изменении состояния кнопки.

Листинг 26.14. Изменение состояния нажатия

```
QPushButton:pressed {
    background: qlineargradient(x1:0, y1:0, x2:1, y2:0,
                                stop:0 rgba(1, 1, 5, 80),
                                stop:0.6 rgba(18, 18, 212, 80),
                                stop:0.5 rgba(142, 142, 245,80) );
    border: 1px solid rgb(18, 18, 212);
}
```

Кнопка после нажатия может также находиться в активированном состоянии (`toggle`), и для того чтобы пользователь сразу же распознал, что кнопка находится в этом состоянии, мы просто перевернем градиент в обратную сторону и отобразим его от темно-серого к светло-серому цвету (листинг 26.15).

Листинг 26.15. Изменение состояния активации

```
QPushButton:checked {
    background: qlineargradient(x1:0, y1:0, x2:0, y2:1,
                                stop:1 rgb(133,133,135),
                                stop:0.4 rgb(31, 31, 33) );
    border: 1px solid rgb(18, 18, 212);
}
```

В активированном состоянии кнопки пользователь может подвести указатель мыши к самой кнопке, поэтому мы также и в этом состоянии при помощи трех точек останова сделаем эффект подсвечивания (листинг 26.16).

Листинг 26.16. Изменение состояния активации с подведенным указателем

```
QPushButton:checked:hover {
    background: qlineargradient(x1:0, y1:1, x2:0, y2:0,
                                stop:1 rgb(31, 31, 33),
                                stop:0.4 rgb(133,133,135),
                                stop:0.1 rgb(0, 0, 150) );
}
```

В рассмотренном примере дизайн наших кнопок предусматривал закругленные углы, и этого мы добились при помощи свойства `border-radius`. Но очень часто бывает так, что вам уже дали готовое растровое изображение для кнопки с закругленными углами. Как же быть тогда — ведь при изменении ее размеров могут неправдоподобно деформироваться и пострадать закругления? Решить эту задачу можно с помощью свойства `border-image`. Оно позволяет задать угловые части, которые не должны быть подвержены деформации. При этом боковые части могут быть подвержены горизонтальной либо вертикальной деформации, а средняя часть может быть подвержена сразу обоим направлениям деформации. С помощью свойства `border-image` разобьем растровое изображение на 9 частей (рис. 26.6). В этом случае мы ограничиваем области закруглений кнопки, а также зоны светового блика и его закруглений.

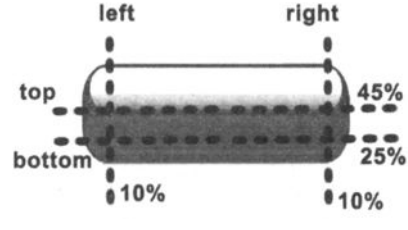


Рис. 26.6. Угловые части, не подверженные деформации

Порядок параметров свойства `border-image` задается следующим образом:

- `top` — верх;
- `right` — право;
- `bottom` — низ;
- `left` — лево.

Для рис. 26.6 эти значения с учетом разрешения растрового изображения кнопки могли бы выглядеть так:

```
border-top-width: 9px;
border-right-width: 2px;
border-bottom-width: 5px;
border-left-width: 2px;
border-image: url(:/style/tst.png) 9 2 5 2 stretch stretch;
```

Если задан всего один числовой параметр, то это означает, что параметрам `top`, `right`, `bottom` и `left` должно быть присвоено одинаковое значение. Например:

```
border-width: 5px;
border-image: url(button.png) 5;
```

Резюме

Библиотека Qt предоставляет возможность использования в программах различных стилей (look & feel) и их смены в процессе работы программы. Стиль может устанавливаться как для всего приложения, так и для каждого виджета в отдельности. Как правило, один стиль устанавливается сразу всем виджетам приложения, для чего используется статический метод `QApplication::setStyle()`.

Каждый стиль имеет различия во внешнем виде и поведении. В распоряжении программиста имеются готовые стили, которые можно установить в приложении. На основе классов унаследованных от `QStyle` можно создавать свои собственные стили. Нужно лишь унаследовать такой класс и переписать методы, необходимые для реализации различий. В каждый из методов передается ряд параметров, предоставляющих всю необходимую информацию для отображения виджетов. Дополнительная информация доступна в объектах класса `QStyleOption`.

Qt предоставляет и более простой способ реализации стилей, который базируется на каскадных стилях языка CSS. Для создания CSS-стиля знание C++ совсем не обязательно, что позволяет привлечь к работе над приложением дизайнеров, не имеющих навыков программирования. Каскадные стили состоят из последовательности правил стиля. Правило стиля задается указанием селектора, определяющего подходящие элементы, и собственно определением стиля.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/26-510/** или с помощью следующего QR-кода (рис. 26.7):



Рис. 26.7. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 27

Мультимедиа

Я думаю, поэтому отображаю.
Джерри Зайденберг

Когда-то, в незапамятные времена, элементы мультимедиа использовались только в играх и специализированных программах, теперь же мультимедиа являются важнейшей составляющей множества приложений, в том числе и веб-приложений, и их применение расширяется с астрономической скоростью. На сегодняшний день мультимедиа играют огромную роль в образовании, а в перспективе, несомненно, будут играть еще бóльшую. Мультимедиа задействуют множество каналов восприятия человека, что упрощает усваивание, понимание и запоминание информации. Таким образом, самый дешевый компьютер с мультимедийной программой способен превратить вашу комнату в эффективный центр обучения.

Мультимедиа можно смело назвать одной из наиболее популярных технологий за всю историю существования компьютерной техники. В настоящее время о них говорят все. Так что же такое мультимедиа? Существует мудрое народное высказывание: «Новое — это хорошо забытое старое», к данному случаю это выражение подходит как нельзя лучше, так как само понятие «мультимедиа» уходит в далекие 50-е годы прошлого века. Именно тогда использование на лекциях и в презентациях нескольких проекторов и звукового сопровождения заложило основу самому этому понятию, но по-настоящему слово приобрело популярность и вошло в обиход только с началом использования этой технологии в компьютерах. Слово «мультимедиа» происходит от латинских слов: *multum* — «много» и *medium* — «средство передачи информации». Таким образом, понятие мультимедиа — это не что иное, как одновременное использование различных типов информации. Под это определение подходят также и телевидение, фильмы, видеоконференции, караоке, компьютерные игры, компьютерные тренинги, веб-браузеры и т. п. Словом, осознаем мы это или нет, но мы уже давно живем в мире мультимедиа. В этой главе мы познакомимся с модулем `QtMultimedia`, обеспечивающим в Qt воспроизведение и запись видео и звука.

Звук

По мнению психологов, человек воспринимает посредством слуха около 30% информации. Поэтому не стоит исключать возможность использования в своих приложениях информации звукового характера. Программа должна быть привлекательна для пользователя не только с точки зрения зрительного эффекта, но также и слухового восприятия. При этом нельзя забывать, что звук в компьютере часто отключен, так что полностью полагаться на него нельзя.

Вы, наверняка, замечали, что в критических ситуациях некоторые программы для привлечения вашего внимания используют короткий звуковой сигнал. Чтобы воспользоваться подобным приемом в своих программах, можно просто вызвать статический метод `QApplication::beep()`. Нужно учесть, что издаваемый звук — громкий, поэтому пользоваться им лучше только в случаях крайней необходимости.

Для серьезного программирования вам этого, естественно, будет недостаточно и, скорее всего, потребуется нечто большее, чем выдача простого предупреждающего звукового сигнала, — как минимум, понадобится воспроизвести какой-нибудь звуковой файл.

Воспроизведение WAV-файлов: класс *QSound*

Класс `QSound` предоставляет только примитивные возможности воспроизведения звуковых файлов формата WAV, и если вам необходимы более продвинутые возможности, такие как, например, управление позицией воспроизведения, громкостью, внедрением эффектов, а также и воспроизведение других звуковых форматов, отличных от WAV, то переходите сразу к изучению класса `QMediaPlayer`, который описан в следующем разделе этой главы.

В классе `QSound` для воспроизведения звука имеется статический метод `play()`, которому нужно передать полное имя звукового файла. Это самый простой способ проигрывания звукового файла и вместе с тем самый экономичный с точки зрения использования ресурсов памяти. Например:

```
QSound::play(":/LoveSong.wav");
```

Если вашей программе необходимо проиграть звуковой файл несколько раз подряд, то этим статическим методом уже не обойтись, и потребуется создать объект класса `QSound`. Он предоставляет метод `setLoops()`, который устанавливает количество повторов при воспроизведении звукового файла. Передача значения `-1` создаст бесконечный цикл повторений. Этот метод должен вызваться до запуска метода `play()`:

```
QSound sound(":/LoveSong.wav");
sound.setLoops(3); // 3 раза
sound.play();
```

Если вам необходимо узнать количество оставшихся повторений, то надо вызвать метод `QSound::loopsRemaining()`, который вернет интересующее вас значение.

Чтобы остановить проигрывание звукового файла, нужно вызвать метод `QSound::stop()`. Для рассматриваемого примера операция остановки воспроизведения выглядит следующим образом:

```
sound.stop();
```

Но этот метод применяется, разумеется, в том случае, когда создан объект класса `QSound` и вызван его метод `play()`.

Если вам вдруг потребуется узнать, закончено ли проигрывание до конца или оно было прервано, то можно вызвать метод `QSound::isFinished()`, который вернет значение `true` в том случае, если проигрывание выполнено до конца, или `false`, если оно было остановлено. Если же воспроизведение продолжается, то метод `isFinished()` вернет значение `false`.

Более продвинутые возможности воспроизведения звуковых файлов: класс *QMediaPlayer*

Несмотря на внутреннюю сложность и многофункциональность класса `QMediaPlayer()`, он очень прост в использовании. Это класс высокого уровня, и он позволяет воспроизводить не только звуковые файлы, но и видеофайлы, и интернет-радио. Всего лишь нескольких строчек в программе будет вполне достаточно, чтобы заставить зазвучать звуковой файл. Например:

```
QMediaPlayer* pPlayer = new QMediaPlayer;
pPlayer->setMedia(QUrl::fromLocalFile(":/LoveSong.mp3"));
pPlayer->play();
```

В этом примере мы создали объект класса `QMediaPlayer`, установили вызовом метода `setMedia()` MP3-файл для воспроизведения и вызвали метод `play()`, для того чтобы файл зазвучал. Вот и все.

Заметьте, что метод `setMedia()` принимает в качестве аргумента только объекты `QUrl`, тем самым класс `QMediaPlayer` обеспечивает возможность проигрывать не только локальные файлы на компьютере или мобильном устройстве, но так же и файлы, находящиеся в Паутине на удаленных серверах. В нашем конкретном случае нам необходимо проиграть локальный файл из ресурса программы, и мы воспользовались статическим методом `QUrl::fromLocalFile()`, для того чтобы преобразовать локальный путь файла в формат `Url`.

Важно отметить, что возможности воспроизведения различных форматов звуковых файлов ограничиваются кодеками, которые предоставляет операционная система, и кодеками, установленными в ней пользователем.

Настало время перейти от простого примера к более сложному. Мы реализуем музыкальный плеер, показанный на рис. 27.1, который обладает графическим интерфейсом и предоставляет возможности выбора файлов для воспроизведения, изменения позиции и громкости, а также имеет кнопки останова, воспроизведения, паузы и индикаторы для отображения актуальной позиции времени воспроизведения и оставшегося времени.

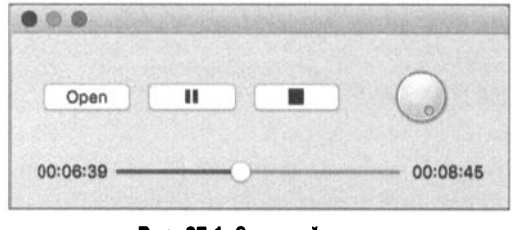


Рис. 27.1. Звуковой плеер

Прежде всего, в проектный файл необходимо включить модуль мультимедиа следующей строкой:

```
QT += multimedia
```

Наша основная программа (листинг 27.1) создает объект класса `SoundPlayer`, задает размеры методом `resize()` и, вызвав метод `show()`, делает этот объект видимым.

Листинг 27.1. Основная программа плеера (файл main.cpp)

```
#include <QApplication>
#include "SoundPlayer.h"
```

```
// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    SoundPlayer  soundPlayer;

    soundPlayer.resize(320, 75);
    soundPlayer.show();

    return app.exec();
}
```

Определение класса `SoundPlayer`, приведенное в листинге 27.2, содержит указатель на объект класса `QMediaPlayer`, а также указатели на компоненты пользовательского интерфейса плеера, такие как кнопки, ползунок и надписи. Сам же класс `SoundPlayer` мы наследуем от самого элементарного класса для виджетов: `QWidget`.

Метод `msecToString()`, который находится в секции `private`, будет нам служить для перевода миллисекунд в строки, — это необходимо нашему плееру для правильного показа времени воспроизведения.

Далее идут слоты:

- `slotOpen()` — отвечает за открытие файлов;
- `slotPlay()` — управляет воспроизведением;
- `slotSetSliderPostion()` — устанавливает актуальную позицию ползунка и обновляет значения времени воспроизведения;
- `slotSetMediaPosition()` — устанавливает позицию для воспроизведения в объекте класса `QMediaPlayer`;
- `slotSetDuration()` — получает общее время воспроизведения файла;
- `slotStatusChanged()` — получает актуальный статус объекта `QMediaPlayer`.

Листинг 27.2. Определение класса `SoundPlayer` (файл SoundPlayer.h)

```
#pragma once
#include <QWidget>
#include <QMediaPlayer>

class QPushButton;
class QSlider;
class QLabel;
class QVBoxLayout;

// ======================================================================
class SoundPlayer : public QWidget {
    Q_OBJECT
protected:
    QMediaPlayer* m_pmp;
    QVBoxLayout*  m_pvbxMainLayout;
```

```
private:
    QPushButton*  m_pcmdPlay;
    QPushButton*  m_pcmdStop;
    QSlider*      m_psldPosition;
    QLabel*       m_plblCurrent;
    QLabel*       m_plblRemain;

    QString msecsToString(qint64 n);

public:
    SoundPlayer(QWidget* pwgt = 0);

private slots:
    void slotOpen             (                    );
    void slotPlay             (                    );
    void slotSetSliderPosition(qint64              );
    void slotSetMediaPosition (int                 );
    void slotSetDuration      (qint64              );
    void slotStatusChanged    (QMediaPlayer::State);
};
```

В конструкторе (листинг 27.3) мы создаем и инициализируем объекты для элементов управления плеером. Для кнопок `m_pcmdPlay` и `m_pcmdStop` мы устанавливаем значки и делаем их недоступными вызовами методов `setEnabled()`. Это делается с тем, чтобы натолкнуть пользователя на мысль, что для воспроизведения должен быть выбран файл. Из этих же соображений диапазон ползунка для позиционирования воспроизведения мы методом `setRange()` устанавливаем нулевым и вызовом метода `setOrientaion()` с параметром `Qt::Horizontal` задаем ему горизонтальную ориентацию. По умолчанию ориентация элемента ползунка вертикальная, поэтому для ползунка, предназначенного для регулирования громкости, мы метод `setOrientation()` не вызываем. Для этого элемента мы вызовом метода `setRange()` устанавливаем диапазон регулировки звука от 0 до 100 и задаем начальную громкость для синхронности сразу в двух объектах: объекте установщика (указатель `pdiaVolume`) и в объекте класса `QMediaPlayer` (указатель `m_pmp`).

После создания и инициализации элементов мы производим сигнально-слотовые соединения. Так, кнопку для открытия файлов (указатель `pcmdOpen`) мы соединяем со слотом нашего класса `SoundPlayer::slotOpen()`. Кнопка **Play** (указатель `m_pcmdPlay`) соединяется со слотом `SoundPlayer::slotPlay()`. Кнопку **Stop** (указатель `m_pcmdStop`) мы соединяем с одноименным слотом объекта (указатель `m_pmp`) класса `QMediaPlayer::stop()` напрямую. Установщик регулировки громкости (указатель `pdiaVolume`) тоже напрямую соединяется со слотом объекта класса `QMediaPlayer::setVolume()`. Прямое соединение здесь возможно потому, что диапазон, установленный в элементе ползунка, идентичен диапазону регулировки громкости объекта класса `QMediaPlayer`.

Для того чтобы при перемещении пользователем слайдера, отвечающего за позицию воспроизведения, всякий раз устанавливалась новая позиция в объекте плеера (указатель `m_pmp`), мы соединяем сигнал слайдера `QSlider::sliderMoved()` со слотом `slotSetMediaPosition()`. В свою очередь, при проигрывании файла слайдер должен всегда отображать текущую позицию, кроме того, элементы для отображения времени воспроизведения должны тоже показывать актуальные величины. Для этой цели в классе

SoundPlayer предусмотрен слот slotSetPosition(), который соединяется с сигналом QMediaPlayer::positionChanged().

Теперь очень важный момент! Продолжительность звучания файла мы должны получать асинхронно. Это сделано в Qt в целях повышения эффективности — чтобы программа не блокировалась во время загрузки файлов, а также с учетом тех редких случаев, когда продолжительность звучания файлов может изменяться в процессе их воспроизведения. Поэтому ни в коем случае не вызывайте метод QMediaPlayer::duration() сразу после вызова метода QMediaPlayer::setMedia() — он вернет неправильные результаты. Для того чтобы все работало корректно, необходимо произвести соединение с сигналом QMediaPlayer::durationChanged(). Так мы и поступили в листинге 27.3, соединив этот сигнал со слотом нашего класса SoundPlayer::slotSetDuration().

Для отслеживания изменения состояния воспроизведения плеера мы используем сигнал QMediaPlayer::stateChanged() и соединяем его со слотом slotStatusChanged() нашего класса SoundPlayer.

В завершение мы располагаем элементы управления нашего плеера на поверхности виджета SoundPlayer при помощи объектов размещения.

Листинг 27.3. Конструктор класса SoundPlayer (файл SoundPlayer.cpp)

```
SoundPlayer::SoundPlayer(QWidget* pwgt/*=0*/) : QWidget(pwgt)
{
    QPushButton* pcmdOpen   = new QPushButton("&Open");
    QDial*       pdiaVolume = new QDial;

    m_pcmdPlay     = new QPushButton;
    m_pcmdStop     = new QPushButton;
    m_psldPosition = new QSlider;
    m_plblCurrent  = new QLabel(msecsToString(0));
    m_plblRemain   = new QLabel(msecsToString(0));
    m_pmp          = new QMediaPlayer;

    m_pcmdPlay->setEnabled(false);
    m_pcmdPlay->setIcon(style()->standardIcon(QStyle::SP_MediaPlay));

    m_pcmdStop->setEnabled(false);
    m_pcmdStop->setIcon(style()->standardIcon(QStyle::SP_MediaStop));

    m_psldPosition->setRange(0, 0);
    m_psldPosition->setOrientation(Qt::Horizontal);

    pdiaVolume->setRange(0, 100);
    int nDefaultVolume = 50;
    m_pmp->setVolume(nDefaultVolume);
    pdiaVolume->setValue(nDefaultVolume);

    connect(pcmdOpen, SIGNAL(clicked()), SLOT(slotOpen()));
    connect(m_pcmdPlay, SIGNAL(clicked()), SLOT(slotPlay()));
    connect(m_pcmdStop, SIGNAL(clicked()), m_pmp, SLOT(stop()));
```

```
    connect(pdiaVolume, SIGNAL(valueChanged(int)),
            m_pmp,      SLOT(setVolume(int))
           );

    connect(m_psldPosition, SIGNAL(sliderMoved(int)),
            SLOT(slotSetMediaPosition(int))
           );

    connect(m_pmp, SIGNAL(positionChanged(qint64)),
            SLOT(slotSetSliderPosition(qint64))
           );
    connect(m_pmp, SIGNAL(durationChanged(qint64)),
            SLOT(slotSetDuration(qint64))
           );
    connect(m_pmp, SIGNAL(stateChanged(QMediaPlayer::State)),
           SLOT(slotStatusChanged(QMediaPlayer::State))
           );

    //Layout setup
    QHBoxLayout* phbxPlayControls = new QHBoxLayout;
    phbxPlayControls->addWidget(pcmdOpen);
    phbxPlayControls->addWidget(m_pcmdPlay);
    phbxPlayControls->addWidget(m_pcmdStop);
    phbxPlayControls->addWidget(pdiaVolume);

    QHBoxLayout* phbxTimeControls = new QHBoxLayout;
    phbxTimeControls->addWidget(m_plblCurrent);
    phbxTimeControls->addWidget(m_psldPosition);
    phbxTimeControls->addWidget(m_plblRemain);

    m_pvbxMainLayout = new QVBoxLayout;
    m_pvbxMainLayout->addLayout(phbxPlayControls);
    m_pvbxMainLayout->addLayout(phbxTimeControls);

    setLayout(m_pvbxMainLayout);
}
```

Слот `slotOpen()` (листинг 27.4) производит вызов диалогового окна открытия файлов, и после его закрытия, в случае, если пользователь выбрал файл, этот файл будет передан как объект `QUrl` в метод `QMediaPlayer::setMedia()`. Теперь, чтобы пользователь мог воспроизвести загруженный файл, мы делаем кнопки **Play** и **Stop** доступными.

Листинг 27.4. Слот `slotOpen()` (файл SoundPlayer.cpp)

```
void SoundPlayer::slotOpen()
{
    QString strFile = QFileDialog::getOpenFileName(this,
                                                    "Open File"
                                                   );
```

```
    if (!strFile.isEmpty()) {
        m_pmp->setMedia(QUrl::fromLocalFile(strFile));
        m_pcmdPlay->setEnabled(true);
        m_pcmdStop->setEnabled(true);
    }
}
```

Слот `slotPlay()` (листинг 27.5) вызывается при каждом нажатии на кнопку **Play**, и, в зависимости от текущего состояния, которое определяется вызовом метода `QMediaPlayer::state()` объекта плеера (указатель `m_pmp`), вызывает либо слот метода `play()`, либо слот метода `pause()`. Например, в случае, если объект плеера находится в состоянии воспроизведения: `QMediaPlayer::PlayingState`, — то вызывается метод `QMediaPlayer::pause()` для его приостановки. Во всех других случаях вызывается метод `QMediaPlayer::play()` для воспроизведения файла.

Листинг 27.5. Слот `slotPlay()` (файл SoundPlayer.cpp)

```
void SoundPlayer::slotPlay()
{
    switch(m_pmp->state()) {
    case QMediaPlayer::PlayingState:
        m_pmp->pause();
        break;
    default:
        m_pmp->play();
        break;
    }
}
```

Слот `slotStatusChanged()` (листинг 27.6) вызывается при каждой смене состояния плеера. Всего их три: `QMediaPlayer::StoppedState` (плеер остановлен), `QMediaPlayer::PlayingState` (плеер находится в режиме воспроизведения) и `QMediaPlayer::PausedState` (воспроизведение поставлено на паузу). В листинге 27.6 мы проверяем только одно состояние: `QMediaPlayer::PlayingState`, чтобы убедиться, что плеер находится в режиме воспроизведения, и отобразить значок паузы. Во всех остальных случаях мы отображаем значок готовности воспроизведения.

Листинг 27.6. Слот `slotStatusChanged()` (файл SoundPlayer.cpp)

```
void SoundPlayer::slotStatusChanged(QMediaPlayer::State state)
{
    switch(state) {
    case QMediaPlayer::PlayingState:
       m_pcmdPlay->setIcon(style()->standardIcon(QStyle::SP_MediaPause));
       break;
    default:
       m_pcmdPlay->setIcon(style()->standardIcon(QStyle::SP_MediaPlay));
       break;
    }
}
```

Слот `slotSetMediaPosition()` (листинг 27.7) вызывает слот `QMediaPlayer::setPosition()` объекта плеера, который принимает в качестве аргумента переменные типа `qint64`. Наш ползунок при перемещении высылает в сигнале `sliderMoved()` переменную типа `int`. Поэтому нам пришлось реализовать этот слот для совместимости типов `qint64` и `int`.

Листинг 27.7. Слот `slotSetMediaPosition()` (файл SoundPlayer.cpp)

```
void SoundPlayer::slotSetMediaPosition(int n)
{
    m_pmp->setPosition(n);
}
```

Метод `msecsToString()`, приведенный в листинге 27.8, осуществляет перевод значения миллисекунд в объект `QTime`, который в конце переводится в строку с заданным форматом для отображения времени.

Листинг 27.8. Метод `msecsToString()` (файл SoundPlayer.cpp)

```
QString SoundPlayer::msecsToString(qint64 n)
{
    int nHours   = (n / (60 * 60 * 1000));
    int nMinutes = ((n  % (60 * 60 * 1000)) / (60 * 1000));
    int nSeconds = ((n % (60 * 1000)) / 1000);

    return QTime(nHours, nMinutes, nSeconds).toString("hh:mm:ss");
}
```

Слот `slotSetDuration()` (листинг 27.9) получает значение продолжительности звучания файла в миллисекундах. В слоте устанавливается максимальное значение ползунка (указатель `m_psldPosition`) в соответствии со значением, переданным во втором аргументе метода `setRange()`. Значения для отображения текущего времени проигрывания и конечного вычисляются при помощи метода `msecsToString()`, рассмотренного в листинге 27.8, и устанавливаются в виджетах надписей (соответственно указатели `m_plblCurrent` и `m_plblRemain`).

Листинг 27.9. Слот `slotSetDuration()` (файл SoundPlayer.cpp)

```
void SoundPlayer::slotSetDuration(qint64 n)
{
    m_psldPosition->setRange(0, n);
    m_plblCurrent->setText(msecsToString(0));
    m_plblRemain->setText(msecsToString(n));
}
```

Слот `slotSetSliderPosition()` (листинг 27.10) вызывается плеером всякий раз при изменении позиции воспроизведения. Актуальную позицию `n` мы устанавливаем вызовом метода `setValue()` в объекте ползунка (указатель `m_psldPosition`). Эту же позицию мы переводим в строку и отображаем вызовом метода `setText()` в виджете надписи для текущего времени воспроизведения (указатель `m_plblCurrent`).

Далее, руководствуясь максимальным значением диапазона ползунка, мы получаем общую продолжительность звучания `nDuration`. Можно, конечно, было бы получить это значение и

прямым вызовом `QMediaPlayer::duration()`, что дало бы тот же результат. Значение общей продолжительности звучания нам нужно для того, чтобы вычислить оставшееся время для воспроизведения и установить его в виджете надписи (указатель `m_plblRemain`) вызовом метода `setText()`.

Листинг 27.10. Слот `slotSetSliderPosition()` (файл SoundPlayer.cpp)

```
void SoundPlayer::slotSetSliderPosition(qint64 n)
{
    m_psldPosition->setValue(n);

    m_plblCurrent->setText(msecsToString(n));
    int nDuration = m_psldPosition->maximum();
    m_plblRemain->setText(msecsToString(nDuration - n));
}
```

Видео и класс *QMediaPlayer*

Как уже было отмечено в самом начале, класс `QMediaPlayer` — это класс высокого уровня, который может воспроизводить не только звуковые, но и видеофайлы. Этим мы воспользуемся для быстрой реализации следующего примера и создадим видеоплеер, показанный на рис. 27.2. Как можно видеть, управляющие элементы видеоплеера полностью совпадают с нашим звуковым плеером (см. рис. 27.1). Единственное различие заключается только в добавленном видеодисплее, который нам нужен собственно для показа видео. Поэтому наш новый класс `VideoPlayer` мы унаследуем от уже имеющегося класса `SoundPlayer`

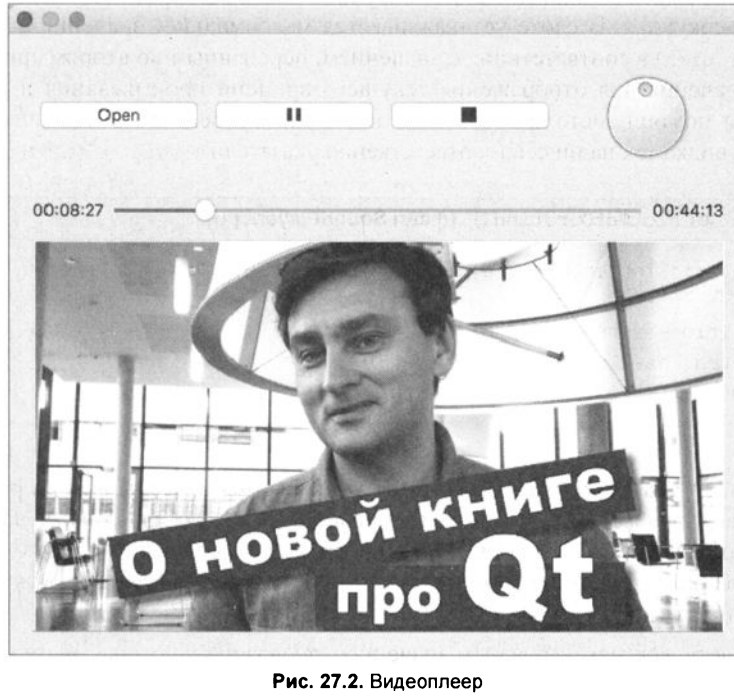


Рис. 27.2. Видеоплеер

(см. листинги 27.1–27.10) и расширим его возможностью отображать видео. Созданный нами плеер будет в состоянии воспроизводить все форматы видеофайлов, кодеки которых установлены в операционной системе, где он будет запущен.

Для отображения видео нам понадобится виджет `QVideoWidget`, который находится в отдельном модуле `QtMultimediaWidget`, поэтому нужно в проектном файле, помимо модуля `QtMultimedia`, добавить и этот модуль:

```
QT += multimedia multimediawidgets
```

В основной программе (листинг 27.11) мы создаем виджет нашего видеоплеера от класса `VideoPlayer`, задаем ему начальные размеры методом `resize()` и отображаем его вызовом метода `show()`.

Листинг 27.11. Основная программа плеера (файл main.cpp)

```
#include <QApplication>
#include "VideoPlayer.h"

// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    VideoPlayer  videoPlayer;

    videoPlayer.resize(400, 450);
    videoPlayer.show();

    return app.exec();
}
```

В листинге 27.12 класс `VideoPlayer` просто наследует ранее реализованный нами класс `SoundPlayer` (см. листинги 27.1–27.10).

Листинг 27.12. Определение класса `VideoPlayer` (файл VideoPlayer.h)

```
#pragma once
#include "../SoundPlayer/SoundPlayer.h"

// ======================================================================
class VideoPlayer : public SoundPlayer {
    Q_OBJECT

public:
    VideoPlayer(QWidget* pwgt = 0);
};
```

Далее в конструкторе (листинг 27.13) мы создаем виджет от класса `QVideoWidget` — это и есть наш видеодисплей. Присваиваем ему минимальный размер 300×300 пикселов, чтобы он всегда был видим (метод `setMinimumSize()`). Добавляем его в объекте размещения

(m_pvbxMainLayout) методом addWidget(). И устанавливаем вызовом метода setVideoOutput() в объекте нашего плеера (указатель m_pmp). Вот и все — видеоплеер готов.

Листинг 27.13. Конструктор класса VideoPlayer (файл VideoPlayer.cpp)

```
#include <QtWidgets>
#include <QVideoWidget>

#include "VideoPlayer.h"

// ----------------------------------------------------------------------
VideoPlayer::VideoPlayer(QWidget* pwgt/*=0*/) : SoundPlayer(pwgt)
{

    QVideoWidget* pvw = new QVideoWidget;
    pvw->setMinimumSize(300, 300);

    m_pvbxMainLayout->addWidget(pvw);

    m_pmp->setVideoOutput(pvw);
}
```

Ошибка в Mac OS X

Ввиду того, что на момент подготовки книги в Qt 5.10 класс QVideoWidget некорректно отображал в Mac OS X разрешение видеопотока, приведенные в этой главе листинги 27.12 и 27.13 отличаются от исходного кода примеров, включенных в электронный архив, сопровождающий книгу. Исходные файлы примеров содержат обходное решение этой проблемы (workaround).

Электронный архив

Напомню, что электронный архив с примерами к этой книге можно скачать с FTP-сервера издательства «БХВ-Петербург» по ссылке **ftp://ftp.bhv.ru/9785977536783.zip** или со страницы книги на сайте **www.bhv.ru** (см. *приложение 4*). Файлы, упомянутые в названиях листингов, находятся в папках с номерами соответствующих глав.

Резюме

В этой главе мы познакомились с простейшей возможностью воспроизведения звуковых WAV-файлов при помощи класса QSound. Мы рассмотрели методы этого класса для воспроизведения, остановки и задания количества повторений. Класс QSound годится лишь для простых задач воспроизведения звука, так как не предоставляет возможностей позиционирования, регулировки громкости и проигрывания файлов в формате, ином, чем WAV.

Если вам необходимы указанные возможности, используйте класс QMediaPlayer — этот класс способен воспроизводить не только звуковые файлы, но и видео. Методам для установки медиафайлов в этом классе необходимо передавать объекты QUrl, благодаря чему в качестве медиаисточника могут выступать также и файлы, находящиеся на удаленном сервере.

При помощи этого класса можно осуществлять следующие операции:

- запускать и останавливать воспроизведение, в том числе: делать паузу, перематывать вперед и назад;
- изменять уровень громкости или полностью отключать звук;
- реагировать на изменение состояния воспроизведения;
- воспроизводить видеофайлы.

Для воспроизведения видео нужен виджет класса `QVideoWidget`, который после создания должен быть установлен методом `setVideoOutput()` в объекте класса `QMediaPlayer`.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/27-510/** или с помощью следующего QR-кода (рис. 27.3):



Рис. 27.3. QR-код для доступа на страницу комментариев к этой главе



ЧАСТЬ V

Создание приложений

Ты можешь победить только тогда, когда веришь в победу.
А. Грам

Глава 28. Сохранение настроек приложения

Глава 29. Буфер обмена и перетаскивание

Глава 30. Интернационализация приложения

Глава 31. Создание меню

Глава 32. Диалоговые окна

Глава 33. Предоставление помощи

Глава 34. Главное окно, создание SDI- и MDI-приложений

Глава 35. Рабочий стол (Desktop)



ГЛАВА 28

Сохранение настроек приложения

> Изменение может стать вашим союзником ...
> Изменение станет вашим врагом, если оно застанет вас врасплох.
>
> *Бак Роджерс, вице-президент IBM*

Возможность изменения и сохранения настроек приложения очень удобна для «приспособления» интерфейса программы под конкретного пользователя. На самом деле, пользователю будет очень приятно, если при запуске приложения будут восстановлены настройки, сделанные им во время предыдущего сеанса работы: приложение будет находиться на том же месте, иметь те же размеры и выглядеть так, как удобно этому пользователю. Необходимо также сохранять названия и пути последних документов, чтобы пользователь мог быстро их выбрать.

Данные настроек приложения организованы в совокупность ключей и значений. *Ключ* (key) представляет собой строковое значение — имя, с помощью которого можно программно получать и устанавливать адресуемое ключом *значение* (key value).

Место, где хранятся эти данные, зависит от конкретной платформы. Так, приложение, запущенное в ОС Windows, сохраняет данные в системном реестре, например в ветках реестра HKEY_LOCAL_MACHINE\Software или HKEY_CURRENT_USER\Software.

> ***Системный реестр ОС Windows***
>
> Системный реестр ОС Windows — это центральная база данных для хранения установок приложений и системы.

В Linux для хранения данных задействованы каталоги $HOME/.qt или $QTDIR/etc.

В Qt для работы с настройками служит класс `QSettings`. При его создании в конструктор можно передать имя компании и название программы. Например:

```
QSettings settings("BHV","MyProgram");
```

Если настройки должны использоваться в нескольких местах, то лучше централизованно установить имя компании и название программы, для чего в классе `QCoreApplication` определены статические методы `setOrganisationName()` и `setApplicationName()`. Например:

```
QCoreApplication::setOrganizationName("BHV");
QCoreApplication::setApplicationName("MyProgram");
```

Для записи настроек приложения служит метод `setValue()`. Первым параметром в метод передается ключ, а вторым — значение. Если такого ключа не существует, то он будет соз-

дан. Настройки базируются на использовании класса `QVariant` (см. *главу 4*), что позволяет записывать значения следующих типов: `bool`, `double`, `int`, `QString`, `QRect`, `QImage` и т. д. Сохранить настройки приложения можно одним из следующих способов:

```
settings.setValue("/Settings/StringKey", "String Value");
settings.setValue("/Settings/IntegerKey", 213);
settings.setValue("/Settings/BooleanKey", true);
```

Для получения данных нужно воспользоваться методом `value()`, который возвращает значения типа `QVariant`. Благодаря этому класс `QSettings` предоставляет методы для чтения разных типов — полученные значения типа `QVariant` необходимо лишь привести к нужному вам типу. Это существенно облегчает задачу. В метод `value()` можно передавать два параметра: первый параметр — это сам ключ, а второй — значение ключа, которое будет возвращаться том в случае, если ключ не найден. Получить настройки можно следующим образом:

```
QString str = settings.value("/Settings/StringKey", "").toString();
int     n   = settings.value("/Settings/IntegerKey", 0).toInt();
bool    b   = settings.value("/Settings/BooleanKey", false).toBool();
```

Для удаления ключей и их значений нужно передать имя ключа в метод `remove()` класса `QSettings`:

```
settings.remove("/Settings/StringKey");
```

Чтобы в методы ключ не передавать полностью, можно задать его префикс методом `beginGroup()`. После использования ключей, связанных с этим префиксом, необходимо вызвать метод `endGroup()`. Методы префиксов разрешается вкладывать друг в друга, получая тем самым длинные ключи. Продемонстрируем вложение методов `beginGroup()` и `endGroup()` друг в друга:

```
settings.beginGroup("/Settings");
    settings.beginGroup("/Colors");
        int nRed = settings.value("/red");
    settings.endGroup();

    settings.beginGroup("/Geometry");
        int nWidth = settings.value("/width");
    settings.endGroup();
settings.endGroup();
```

Программа, приведенная в листингах 28.1–28.8, создает окно, показанное на рис. 28.1. Надпись информирует о количестве запусков приложения. Виджет выпадающего списка управляет изменением расцветки текстового поля и предоставляет два режима: **Classic** (Классический), когда текст — черный, а фон — белый, и **Borland** (стиль расцветки фирмы Borland), когда текст — желтый, а фон — синий. Флажок **Disable edit** (Отключить редактирование) включает/выключает режим редактирования текстового поля. Все упомянутые здесь настройки и размеры окна сохраняются при его закрытии и восстанавливаются при следующем запуске программы.

В листинге 28.1 создается виджет класса `MyProgram`.

Листинг 28.1. Создание виджета класса `MyProgram` (файл main.cpp)

```
#include <QApplication>
#include "MyProgram.h"
```

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    MyProgram    myProgram;

    myProgram.show();

    return app.exec();
}
```

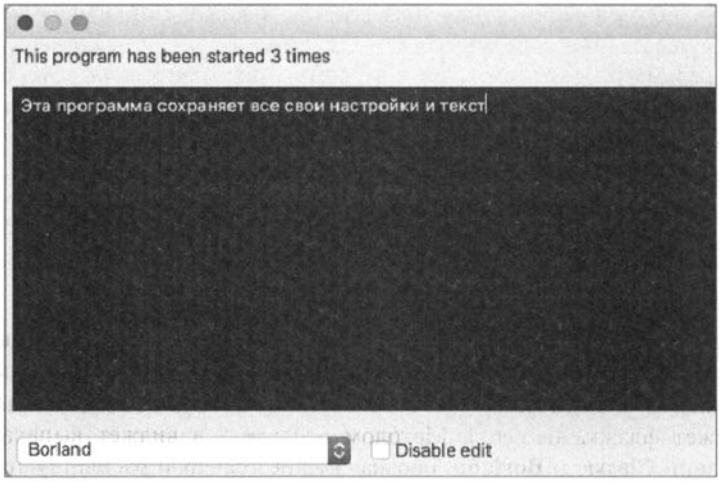


Рис. 28.1. Программа, сохраняющая и восстанавливающая настройки

Определение класса `MyProgram`, приведенное в листинге 28.2, содержит атрибут `m_settings` класса `QSettings`. Класс также содержит указатели на виджет выпадающего списка (`m_pcbo`), на флажок (`m_pchk`), на виджет текстового поля (`m_ptxt`), на виджет надписи (`m_plbl`) и счетчик количества запусков программы (`m_nCounter`). В классе `MyProgram` определены методы для записи и чтения данных настроек программы: `writeSettings()` и `readSettings()`. В нем также определены два слота: `slotCheckedBoxClicked()` и `slotComboBoxActivated()` — для выполнения действий, вызываемых изменением состояний флага и выпадающего списка.

Листинг 28.2. Определение класса `MyProgram` (файл MyProgram.h)

```
#pragma once

#include <QWidget>
#include <QSettings>

class QComboBox;
class QCheckBox;
class QTextEdit;
class QLabel;

// ============================================================================
class MyProgram : public QWidget {
Q_OBJECT
```

```
private:
    QSettings  m_settings;
    QComboBox* m_pcbo;
    QCheckBox* m_pchk;
    QTextEdit* m_ptxt;
    QLabel*    m_plbl;
    int        m_nCounter;

public:
    MyProgram(QWidget* pwgt = 0);
    virtual ~MyProgram();

    void writeSettings();
    void readSettings ();

public slots:
    void slotCheckBoxClicked  (   );
    void slotComboBoxActivated(int);
};
```

В листинге 28.3 выполняется инициализация объекта настроек `m_settings` двумя строками: названием фирмы `"BHV"` и названием продукта `"MyProgram"`. В конструкторе класса создается виджет надписи (`m_plbl`), виджет текстового поля (`m_ptxt`), виджет выпадающего списка (`m_pcbo`) и виджет флажка (`m_pchk`). Методом `addItem()` в виджет выпадающего списка добавляются опции **Classic** и **Borland**, обозначающие текущую расцветку текстового поля. Сигнал `clicked()` виджета флажка (указатель `m_pchk`) соединяется со слотом `slotCheckBoxClicked()`, а сигнал `activated(int)` виджета выпадающего списка — со слотом `slotComboBoxActivated(int)`. Затем вызывается метод `readSettings()`. В завершение созданные виджеты размещаются при помощи компоновок (см. *главу 6*).

Листинг 28.3. Конструктор `MyProgram` (файл MyProgram.cpp)

```
MyProgram::MyProgram(QWidget* pwgt/*=0*/)
    : QWidget(pwgt)
    , m_settings("BHV","MyProgram")
{
    m_plbl = new QLabel;
    m_ptxt = new QTextEdit;
    m_pcbo = new QComboBox;
    m_pchk = new QCheckBox("Disable edit");

    m_pcbo->addItem("Classic");
    m_pcbo->addItem("Borland");

    connect(m_pchk, SIGNAL(clicked()), SLOT(slotCheckBoxClicked()));
    connect(m_pcbo,
            SIGNAL(activated(int)),
            SLOT(slotComboBoxActivated(int))
           );
```

```
    readSettings();

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    QHBoxLayout* phbxLayout = new QHBoxLayout;

    pvbxLayout->setContentsMargins(5, 5, 5, 5);
    phbxLayout->setSpacing(15);
    pvbxLayout->setSpacing(15);
    pvbxLayout->addWidget(m_plbl);
    pvbxLayout->addWidget(m_ptxt);
    phbxLayout->addWidget(m_pcbo);
    phbxLayout->addWidget(m_pchk);

    pvbxLayout->addLayout(phbxLayout);
    setLayout(pvbxLayout);
}
```

В листинге 28.4 приведен метод чтения настроек `readSettings()`, в котором прежде всего методом `beginGroup()` объекту класса `QSettings` передается префикс ключа. Это делается для того, чтобы не передавать в методы чтения настроек полный ключ. Строковой переменной `strText` присваивается значение ключа `/text`, прочитанное методом `readEntry()`. В том случае, если ключа с именем `/text` не существует, метод вернет значение, указанное во втором параметре, то есть пустую строку. Аналогично выполняется инициализация переменных: `nWidth` и `nHeight`, предназначенных для хранения размеров окна, `nComboItem`, хранящей индекс опции виджета выпадающего списка, `bEdit`, хранящей значения разрешения редактирования, и `m_nCounter`, ведущей подсчет количества запусков программы. После прочтения виджеты изменяются в соответствии с полученными значениями. Значение счетчика `m_nCounter` увеличивается на единицу. В завершение, для снятия установленного префикса ключа, осуществляется вызов метода `endGroup()`.

Листинг 28.4. Метод readSettings() (файл MyProgram.cpp)

```
void MyProgram::readSettings()
{
    m_settings.beginGroup("/Settings");

    QString strText    = m_settings.value("/text", "").toString();
    int     nWidth     = m_settings.value("/width", width()).toInt();
    int     nHeight    = m_settings.value("/height", height()).toInt();
    int     nComboItem = m_settings.value("/highlight", 0).toInt();
    bool    bEdit      = m_settings.value("/edit", false).toBool();

    m_nCounter = m_settings.value("/counter", 1).toInt();

    QString str = QString().setNum(m_nCounter++);
    m_plbl->setText("This program has been started " + str + " times");

    m_ptxt->setPlainText(strText);
```

```
    resize(nWidth, nHeight);

    m_pchk->setChecked(bEdit);
    slotCheckBoxClicked();

    m_pcbo->setCurrentIndex(nComboItem);
    slotComboBoxActivated(nComboItem);

    m_settings.endGroup();
}
```

Деструктор, приведенный в листинге 28.5, — это самое удобное место, где можно выполнить запись настроек приложения, так как он вызывается при уничтожении виджета. Для записи настроек используется метод `writeSettings()`.

Листинг 28.5. Деструктор `~MyProgram()` (файл MyProgram.cpp)

```
/*virtual*/MyProgram::~MyProgram()
{
    writeSettings();
}
```

> ***АДЕКВАТНАЯ ЗАМЕНА ДЕСТРУКТОРА***
>
> Вместо деструктора можно воспользоваться методом обработки события `closeEvent()`, который вызывается при закрытии окна виджета.

В методе `writeSettings()` после установки префикса ключа с помощью метода `beginGroup()` записываются настройки приложения. Выполняется серия вызовов метода `setValue()`, где первым параметром указывается имя ключа, а вторым — его значение (листинг 28.6).

Листинг 28.6. Метод `writeSettings()` (файл MyProgram.cpp)

```
void MyProgram::writeSettings()
{
    m_settings.beginGroup("/Settings");
    m_settings.setValue("/counter", m_nCounter);
    m_settings.setValue("/text", m_ptxt->toPlainText());
    m_settings.setValue("/width", width());
    m_settings.setValue("/height", height());
    m_settings.setValue("/highlight", m_pcbo->currentIndex());
    m_settings.setValue("/edit", m_pchk->isChecked());
    m_settings.endGroup();
}
```

Метод `slotCheckBoxClicked()` устанавливает виджет текстового поля в активное или неактивное состояние в зависимости от состояния виджета флажка, возвращаемого методом `isChecked()` (листинг 28.7).

Листинг 28.7. Метод slotCheckBoxClicked() (файл MyProgram.cpp)

```
void MyProgram::slotCheckBoxClicked()
{
    m_ptxt->setEnabled(!m_pchk->isChecked());
}
```

Метод `slotComboBoxActivated()`, приведенный в листинге 28.8, устанавливает цвет фона и шрифта виджета текстового поля в зависимости от индекса элемента выпадающего списка `n`. Изменение палитры осуществляется только для активного состояния виджета `QPalette::Active` (см. *главу 13*).

Листинг 28.8. Метод slotComboBoxActivated() (файл MyProgram.cpp)

```
void MyProgram::slotComboBoxActivated(int n)
{
    QPalette pal = m_ptxt->palette();
    pal.setColor(QPalette::Active,
                 QPalette::Base,
                 n ? Qt::darkBlue : Qt::white
                );
    pal.setColor(QPalette::Active,
                 QPalette::Text,
                 n ? Qt::yellow : Qt::black
                );
    m_ptxt->setPalette(pal);
}
```

В больших проектах целесообразно обращаться к объекту настроек приложения централизованно. Для этого можно унаследовать класс `QApplication` и инкапсулировать объект настроек в нем. Этот подход продемонстрирован в примере (листинг 28.9).

Листинг 28.9. Пример для централизованного использования объекта настроек

```
class App : public QApplication {
Q_OBJECT
private:
    QSettings* m_pSettings;

public:
    App(int& argc,
        char** argv,
        const QString& strOrg,
        const QString& strAppName
       ) : QApplication(argc, argv)
         , m_pSettings(0)
    {
        setOrganizationName(strOrg);
        setApplicationName(strAppName);

        m_pSettings = new QSettings(strOrg, strAppName, this);
    }
```

```
    static App* theApp()
    {
        return (App*)qApp;
    }

    QSettings* settings()
    {
        return m_pSettings;
    }
};
```

В конструкторе мы делегируем переменные `argc`, `argv` унаследованному классу `QApplication`. Третий и четвертый параметры конструктора — это имя организации `strOrg` и название приложения `strAppName`. Строковые значения этих переменных устанавливаем вызовом методов `QCoreApplication::setOrganizationName()` и `QCoreApplication::setApplicationName()` в объекте приложения и используем их также при создании объекта настроек (указатель `m_pSettings`). Статический метод `theApp()` нам нужен для того, чтобы получать доступ к объекту нашего приложения из любой библиотеки проекта. Метод `settings()` возвращает указатель на объект настроек. Основная программа с использованием нашего класса приложения может выглядеть следующим образом:

```
int main(int argc, char** argv)
{
    App app(argc, argv, "MyCompany", "MyApp");
    ...
    ...
    return app.exec();
}
```

А обращение к объекту настроек из любой библиотеки проекта будет таким:

```
QSettings* pst = App::theApp()->settings();
pst->setValue("Language", "en");
```

Резюме

Qt предоставляет возможность хранения информации о конфигурации приложений, необходимой для того, чтобы дать пользователю возможность настраивать приложение под себя. Данные настроек приложения — это совокупность ключей и их значений. Ключи — это значения строкового типа, состоящие из подстрок, разделенных символом /. Значения могут иметь тип, поддерживаемый классом `QVariant`. Все значения можно читать методом `value()` и записывать методом `setValue()`.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/28-510/** или с помощью следующего QR-кода (рис. 28.2):



Рис. 28.2. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 29

Буфер обмена и перетаскивание

— У вас есть сильные программисты?
— Есть, а зачем они вам?
— Нужно несколько компьютеров перетащить.

Работая на компьютере, пользователи часто открывают несколько приложений и нуждаются в обмене данными между ними. Для поддержки такой возможности существуют две распространенные технологии: использование буфера обмена и перетаскивание объектов. Надо заметить, что часто эти технологии используются и в работе внутри одного приложения, а не только для обмена данными между разными приложениями.

Буфер обмена

Буфер обмена (Clipboard) обеспечивает возможность обмена данными как между разными приложениями, так и между открытыми в них окнами (или разными областями одного и того же окна). Это один из самых популярных способов копирования данных из одного места в другое. Он представляет собой область памяти, к которой могут иметь доступ все запущенные в системе приложения. Любое из них может записывать или считывать информацию из буфера обмена. Программы, работающие с буфером обмена, должны предоставлять стандартные команды: вырезать (cut), скопировать (copy) и вставить (paste), и эти команды необходимо снабдить определенными комбинациями «горячих» клавиш, ускоряющих работу пользователя: <Ctrl>+<X>, <Ctrl>+<C> и <Ctrl>+<V> соответственно.

Для работы с буфером обмена в Qt используется класс `QClipboard`. Не стоит пытаться создавать объект этого класса самостоятельно, так как он создается при запуске приложения автоматически и может существовать в приложении только в единственном числе. Объект класса `QClipboard` отправляет сигнал `dataChanged()` каждый раз, когда одно из приложений помещает в буфер обмена новые данные. Если необходимо контролировать данные, размещенные в буфере обмена, то этот сигнал соединяют с соответствующим слотом. Например:

```
connect(qApp->clipboard(), SIGNAL(dataChanged()),
        pwgt, SLOT(slotDataControl())
       );
```

Данные можно помещать в буфер обмена при помощи методов `setText()`, `setPixmap()`, `setImage()` или `setMimeData()`. Например:

```
QClipboard* pcb = QApplication::clipboard();
pcb->setText("My Text");
```

Метод setMimeData(): помещение в буфер обмена данных любого типа

При помощи метода `setMimeData()` можно помещать в буфер обмена данные любого типа. Метод принимает указатель на объект класса, унаследованного от класса `QMimeSource`. `QMimeSource` — это абстрактный класс, являющийся основой для типов данных, которые могут быть преобразованы в другие форматы.

Получение данных из буфера обмена осуществляется с помощью методов `text()`, `image()`, `pixmap()` и `mimeData()`. Например:

```
QClipboard* pcb = QApplication::clipboard();
QString     str = pcb->text();
if (!str.isNull()) {
    qDebug() <<  "Clipboard Text: " << str;
}
```

Перетаскивание

Перетаскивание (drag & drop) — это, наряду с буфером обмена, столь же мощная технология обмена данными как между разными приложениями, так и между открытыми в них окнами (или разными областями одного и того же окна). Она предоставляет пользователю более интуитивный, чем буфер обмена, механизм обмена данными. В настоящее время поддержка перетаскивания является неотъемлемой частью практически любого приложения. Процесс перетаскивания выглядит следующим образом: пользователь нажимает левую кнопку мыши, когда указатель мыши находится на объекте, и, удерживая кнопку, перетаскивает объект из одного окна (места в окне) в другое. Это позволяет обращаться с виртуальными объектами как с объектами реального мира, перетаскивая их с места на место. Один из ярких примеров возможности перетаскивания демонстрирует **Recycle Bin** (Корзина) на рабочем столе ОС Windows, в которую сбрасывают все удаляемые ненужные файлы.

Класс `QWidget` обладает всеми необходимыми методами для поддержки технологии перетаскивания, а некоторые из классов иерархии виджетов содержат ее полную реализацию. Поэтому, прежде чем приступить к реализации перетаскивания, необходимо убедиться в том, что оно не реализовано в виджете. Например, класс `QTextEdit` (см. *главу 10*) предоставляет возможность перетаскивания выделенного текста.

Для перетаскивания Qt предоставляет класс `QDrag`, а для размещения данных различных типов при перетаскивании — класс `QMimeData`. Обозначение «MIME» означает Multipurpose Internet Mail Extension (многоцелевые расширения почты Интернета). Он предусматривает пересылку текстовых сообщений на различных языках, а также изображений, аудио- и видеоинформации и некоторых других типов данных. К примеру, MIME-тип text/plain означает, что данные представляют собой обычный ASCII-текст, а text/html означает, что данные — это текст, форматированный с помощью языка HTML. Для растровых изображений используется тип вида image/*. Например, для файлов с расширением jpg MIME-типом является image/jpg. Если вы используете данные собственного типа, которые могут интерпретироваться только лишь вашим приложением, то тип должен иметь вид application/*. В табл. 29.1 сведены наиболее часто используемые типы.

Как мы уже упоминали ранее, реализация MIME в Qt представлена классом `QMimeData`. В этом классе определены методы для записи данных различных типов:

- цветовых значений — `setColorData()`;
- растровых изображений — `setImageData()`;

- текстовой информации — `setText()`;
- гипертекстовой информации в формате HTML — `setHtml()`;
- списков (ссылок) URL (Uniform Resource Locator, единообразный определитель ресурса) — `setUrls()`. Этот метод часто применяется для перетаскивания файлов.

Таблица 29.1. *MIME-типы*

MIME-тип	Описание
application/*	Данные собственного приложения, которые не могут интерпретироваться другими программами
audio/*	Звуковые данные, например: audio/wav
image/*	Растровое изображение, например: image/png
model/*	Данные моделей, зачастую трехмерные, например: model/vrml
text/*	Текст, например: text/plain
video/*	Видеоданные, например: video/mpeg

На все случаи перечисленных методов, естественно, не хватит, так как может понадобиться перетаскивать и принимать свои собственные типы данных (например, звуковые данные). Как поступать в подобных ситуациях? Для этих случаев в классе `QMimeData` определен метод `setData()`, в который первым параметром нужно передать строку, характеризующую тип данных, а вторым — сами данные в объекте класса `QByteArray` (см. *главу 4*). Можно поступить и иначе — унаследовать класс `QMimeData` и перезаписать методы `formats()` и `retrieveData()`.

Программирование поддержки перетаскивания можно условно разделить на две части: первая часть включает в себя код для перетаскивания объекта (drag), а вторая реализует область приема для сбрасываемых в нее объектов (drop). Также вторая часть должна распознавать, в состоянии она принять перетаскиваемый объект или нет. На рис. 29.1 показан процесс перетаскивания с соответствующими методами возникающих событий.

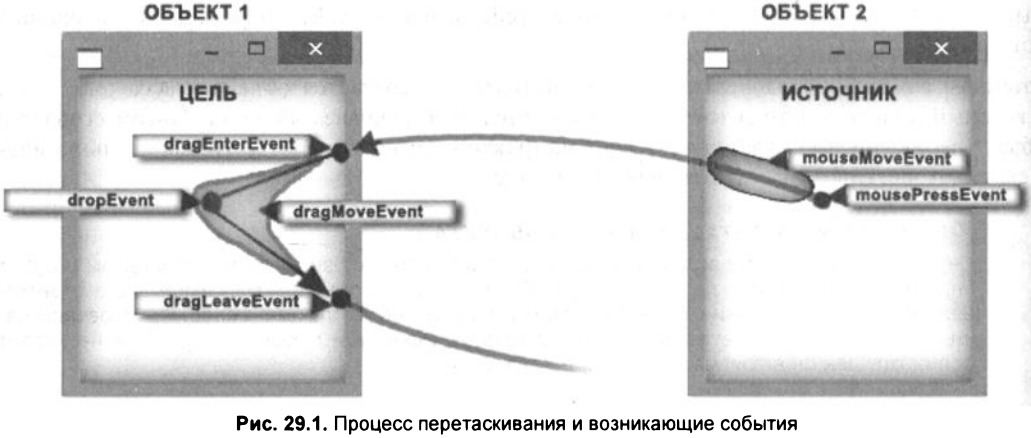


Рис. 29.1. Процесс перетаскивания и возникающие события

Реализация drag

Реализация drag — первой части перетаскивания — начинается с перезаписи методов `mousePressEvent()` и `mouseMoveEvent()`. В первом из этих методов сохраняется позиция указателя мыши, в которой была нажата кнопка. Эта позиция пригодится в методе `mouseMoveEvent()` для определения момента старта операции перетаскивания.

Следующий пример (листинг 29.1) демонстрирует метод перетаскивания текстовой информации из окна виджета в окно редактора (рис. 29.2).

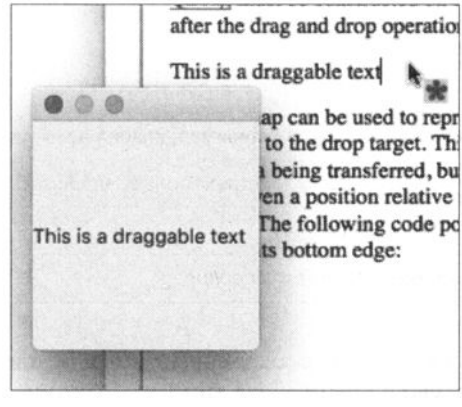


Рис. 29.2. Перетаскивание текста в окно текстового редактора

Определение класса `Drag`, приведенное в листинге 29.1, содержит атрибут `m_ptDragPos`, предназначенный для сохранения положения курсора мыши в момент нажатия левой кнопки. Инициализация этого атрибута осуществляется в методе `mousePressEvent()` только в том случае, если была нажата левая кнопка мыши.

Метод `mouseMoveEvent()` нужен для распознавания начала перетаскивания. Дело в том, что нажатие левой кнопки мыши и последующее перемещение указателя не всегда говорит о желании пользователя перетащить объект, — так, у пользователя могла просто дрогнуть рука, случайно переместив указатель мыши. Для большей уверенности необходимо вычислить расстояние между текущей позицией и той позицией, в которой была нажата левая кнопка мыши. Если это расстояние превышает величину, возвращаемую статическим методом `startDragDistance()` (обычно 4 пиксела), то можно считать, что перемещение указателя мыши было неслучайным, и пользователь действительно хочет перетащить выбранный объект.

Затем вызывается метод `startDrag()`. В этом методе создается объект класса `QMimeData`, в который вызовом метода `setText()` передается перетаскиваемый текст. Потом создается объект перетаскивания класса `QDrag`, в конструктор которого передается указатель на виджет, из которого осуществляется перетаскивание.

> ***Уничтожение перетаскиваемого объекта***
>
> Передача указателя на виджет вовсе не означает, что этот виджет станет предком и будет нести ответственность за уничтожение объекта перетаскивания. На самом деле ответственность за уничтожение объектов перетаскивания несет только менеджер перетаскивания. Уничтожение перетаскиваемого объекта выполняется в любом случае, и не играет роли, отпущен он в принимающей зоне или нет.

Вызов метода `setPixmap()` устанавливает небольшое растровое изображение, перемещаемое вместе с указателем мыши при перетаскивании. Метод `exec()` запускает операцию перетас-

кивания. В этот метод можно также передавать значения, влияющие на внешний вид значка, находящегося рядом с курсором мыши и поясняющего смысл действия перетаскивания. Например, для копирования — это значение `Qt::CopyAction`, для перемещения — `Qt::MoveAction`, для создания ссылки — `Qt::LinkAction`. По умолчанию это значение устанавливается равным `Qt::MoveAction`.

Листинг 29.1. Перетаскивание текстовой информации из окна виджета в окно редактора (файл Drag.h)

```
#pragma once

#include <QtWidgets>

// ============================================================================
class Drag : public QLabel {
Q_OBJECT
private:
    QPoint m_ptDragPos;

    void startDrag()
    {
        QMimeData* pMimeData = new QMimeData;
        pMimeData->setText(text());

        QDrag* pDrag = new QDrag(this);
        pDrag->setMimeData(pMimeData);
        pDrag->setPixmap(QPixmap(":/img1.png"));
        pDrag->exec();
    }

protected:
    virtual void mousePressEvent(QMouseEvent* pe)
    {
        if (pe->button() == Qt::LeftButton) {
            m_ptDragPos = pe->pos();
        }
        QWidget::mousePressEvent(pe);
    }

    virtual void mouseMoveEvent (QMouseEvent* pe)
    {
        if (pe->buttons() & Qt::LeftButton) {
            int distance = (pe->pos() - m_ptDragPos).manhattanLength();
            if (distance > QApplication::startDragDistance()) {
                startDrag();
            }
        }
        QWidget::mouseMoveEvent(pe);
    }
```

```
public:
    Drag(QWidget* pwgt = 0) : QLabel("This is a draggable text", pwgt)
    {
    }
};
```

Реализация drop

В следующем примере (листинг 29.2) реализован виджет, способный принимать сбрасываемые в него объекты (в данном случае — файлы). После сбрасывания виджет отображает полное имя файла (рис. 29.3).

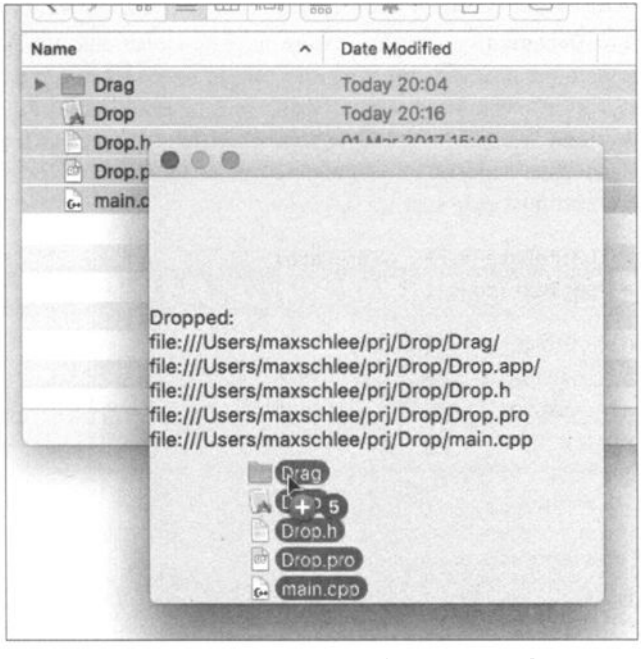


Рис. 29.3. Виджет, принимающий сбрасываемые объекты

Для того чтобы виджет был в состоянии принимать сбрасываемые объекты, в конструкторе класса `Drop` осуществляется вызов метода `setAcceptDrops()`, в который передается значение `true`. Кроме того, для получения сбрасываемых объектов необходимо переопределить методы `dragEnterEvent()` и `dropEvent()`.

Метод `dragEnterEvent()` вызывается каждый раз, когда перетаскиваемые объекты пересекают границу виджета (см. рис. 29.1). В этом методе виджет сообщает о готовности или неготовности принимать перетаскиваемые объекты, при этом указатель мыши из перечеркнутого круга превращается в стрелку с прямоугольником (возможно, со знаком плюс, если пользователь нажал при перетаскивании клавишу <Ctrl>), в противном случае внешний вид указателя остается без изменений.

Обычно виджет не способен принимать данные всех типов, поэтому необходимо проверять на совместимость тип данных перетаскиваемых объектов с помощью метода `hasFormat()`.

Полный MIME-тип

Полный MIME-тип состоит из типа и подтипа, разделенных косой чертой. В нашем случае он имеет вид `"text/uri-list"`.

Вызов метода `acceptProposedAction()` объекта события `QDragEnterEvent` сообщает о готовности виджета принять перетаскиваемый объект.

Методы accept() и ignore() класса QDragEnterEvent

Класс `QDragEnterEvent` унаследован от класса `QDragMoveEvent` (см. рис. 14.1). Этот класс предоставляет методы `accept()` и `ignore()`, которыми можно воспользоваться для разрешения (или запрета) приема перетаскиваемых объектов. В эти методы можно передавать объекты класса `QRect`. С их помощью можно ограничить размеры принимающей области в самом виджете. Чтобы разрешить, например, сбрасывание объектов во всей области виджета, можно сделать следующий вызов: `accept(rect())`.

Метод `dropEvent()` вызывается при сбрасывании перетаскиваемых объектов в пределах окна виджета, что происходит в момент отпускания левой кнопки мыши. Вызов метода `urls()` объекта класса `QMimeData` возвращает список файлов в переменную `urlList`. Далее в цикле `foreach` вызовом метода `QUrl::toString()` извлекаются строки, которые объединяются в одну с символом возврата каретки (`"\n"`) в качестве разделителя. После этого полученная строка отображается методом `setText()`.

Листинг 29.2. Реализация виджета, способного принимать сбрасываемые в него объекты (файл Drop.h)

```
#pragma once

#include <QtWidgets>

// ============================================================================
class Drop : public QLabel {
Q_OBJECT

protected:
    virtual void dragEnterEvent(QDragEnterEvent* pe)
    {
        if (pe->mimeData()->hasFormat("text/uri-list")) {
            pe->acceptProposedAction();
        }
    }

    virtual void dropEvent(QDropEvent* pe)
    {
        QList<QUrl> urlList = pe->mimeData()->urls();
        QString     str;
        foreach(QUrl url, urlList) {
            str += url.toString() + "\n";
        }
        setText("Dropped:\n" + str);
    }
```

```
public:
    Drop(QWidget* pwgt = 0) : QLabel("Drop Area", pwgt)
    {
        setAcceptDrops(true);
    }

};
```

Создание собственных типов перетаскивания

Возможности Qt не ограничены перетаскиванием данных определенных типов, таких как, например, текст или растровые изображения. Перетаскиваться может информация любого типа. Однако предварительно необходимо определиться с идентификацией типа перетаскиваемых данных, чтобы принимающая сторона могла решить — допускает она их или нет. Идентификация достигается включением строки `mimeType`, которая представляет перетаскиваемые данные, эта же строка используется принимающей стороной для получения доступа к данным. Таким образом, основной метод для создания объекта перетаскивания мог бы выглядеть примерно так, как показано в листинге 29.3.

Листинг 29.3. Собственный тип перетаскивания

```
MyDragClass::startDrag()
{
    QImage     img("mira.jpg");
    QByteArray data;
    QBuffer    buf(&data);
    QMimeData* pMimeData = new QMimeData;

    buf.open(QIODevice::WriteOnly);
    img.save(&buf, "JPG");
    pMimeData->setData("image/jpg", data);

    QDrag* pDrag = new QDrag(this);
    pDrag->setMimeData(pMimeData);
    pDrag->exec(Qt::MoveAction);
}
```

В листинге 29.3 создается объект растрового изображения, данные которого будут подвергнуты перетаскиванию. Конечно, мы могли бы просто воспользоваться методом `QMimeData::setImageData()`, но ради демонстрации создания собственного типа для перетаскивания будем исходить из того, что этого метода нет. Итак, чтобы поместить данные в бинарном виде в объект класса `QMimeData`, мы создаем объекты классов `QByteArray` и `QBuffer`. При помощи метода `QImage::save()` мы записываем данные в объект класса `QBuffer`, который управляет бинарным массивом (`data`). После чего передаем этот массив в метод `QMimeData::setData()`. Обратите внимание на первый параметр этого метода — строка `image/jpg` является идентификатором для перетаскиваемого типа данных.

Если перетаскивание должно работать только в пределах вашего приложения, то можно поступить еще проще и тем повысить его эффективность — избежать промежуточного копирования данных в `QByteArray`, а использовать их напрямую. Этот способ учитывает, что

приложение работает в одном адресном пространстве, и поэтому можно напрямую передавать адреса и указатели на объекты, записывая их в MIME-объектах. Разумеется, сами данные не должны быть локальными. Продемонстрируем этот подход на примере, где в качестве данных передаются указатели на виджеты, с помощью которых принимающая сторона может получить доступ к сброшенному виджету. В окнах программы (листинги 29.4–29.13), показанных на рис. 29.4, мы видим два виджета, которые можно передавать друг другу, а также и самим себе, посредством перетаскивания.

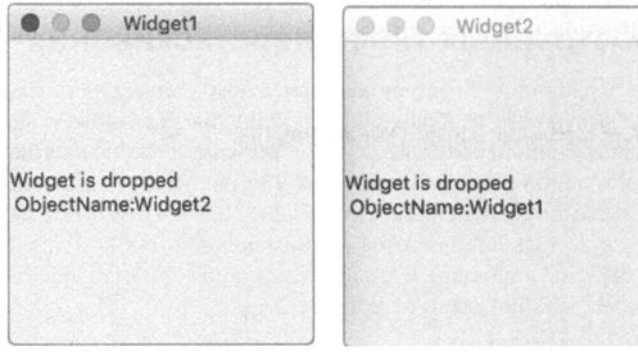


Рис. 29.4. Виджеты, принимающие сбрасываемые виджеты

В основной программе (листинг 29.4) мы создаем два виджета класса `Widget`, реализация которого приведена в листингах 29.7–29.13. Для того чтобы визуально отличать один виджет от другого, мы устанавливаем различные заголовки при помощи метода `setWindowTitle()`. А для программного различия мы присваиваем этим объектам имена `"Widget1"` и `"Widget2"` вызовом метода `setObjectName()`.

Листинг 29.4. Создание двух виджетов (файл main.cpp)

```
#include <QApplication>
#include "Widget.h"

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);
    Widget       wgt1;
    Widget       wgt2;

    wgt1.setWindowTitle("Widget1");
    wgt1.setObjectName("Widget1");
    wgt1.resize(200, 200);
    wgt1.show();

    wgt2.setWindowTitle("Widget2");
    wgt2.setObjectName("Widget2");
    wgt2.resize(200, 200);
    wgt2.show();

    return app.exec();
}
```

Наш MIME-класс (листинг 29.5) в качестве данных должен предоставлять указатель на виджет, поэтому мы определяем в классе `WidgetMimeData` атрибут `m_pwgt`. Статический метод `mimeType()` возвращает MIME-тип, в нашем случае это `"application/widget"`. Первая часть этой строки: `application` — говорит о том, что определяемый тип может использоваться только внутри этого приложения и более нигде. Вторая часть: `widget` — сообщает, что перетаскиваемые данные — это виджет. Указатель на сам виджет устанавливается методом `setWidget()`. В этом методе мы не просто присваиваем нашему указателю `m_pwgt` новый адрес, но и устанавливаем тип вызовом метода `setData()`, для чего передаем строковый идентификатор MIME-типа и пустой объект `QByteArray`. Указатель на виджет возвращает метод `widget()`.

Листинг 29.5. Класс `WidgetMimeData` (файл WidgetDrag.h)

```
class WidgetMimeData : public QMimeData {
private:
    QWidget* m_pwgt;

public:
    WidgetMimeData() : QMimeData()
    {
    }

    virtual ~WidgetMimeData()
    {
    }

    static QString mimeType()
    {
        return "application/widget";
    }

    void setWidget(QWidget* pwgt)
    {
        m_pwgt = pwgt;
        setData(mimeType(), QByteArray());
    }

    QWidget* widget() const
    {
        return m_pwgt;
    }
};
```

Класс `WidgetDrag` (листинг 29.6) — это класс для объекта перетаскивания, он наследуется от класса `QDrag`. В нем мы реализуем метод `setWidget()`, в котором создается объект нашего MIME-класса `WidgetMimeData`. В этом объекте вызовом метода `setWidget()` устанавливается переданный в метод указатель (`pwgt`). В завершение метод `setMimeData()` устанавливает в нашем объекте перетаскивания MIME-объект (указатель `pmd`).

Листинг 29.6. Класс `WidgetDrag` (файл WidgetDrag.h)

```
class WidgetDrag : public QDrag {
public:
    WidgetDrag(QWidget* pwgtDragSource = 0) : QDrag(pwgtDragSource)
    {
    }

    void setWidget(QWidget* pwgt)
    {
        WidgetMimeData* pmd = new WidgetMimeData;
        pmd->setWidget(pwgt);
        setMimeData(pmd);
    }
};
```

Теперь реализуем класс виджета, который будет поддерживать наш новый MIME-тип (листинг 29.7). Для того чтобы отображать текстовую информацию, унаследуем его от класса `QLabel`. Этот виджет в состоянии не только принимать, но и генерировать объекты перетаскивания. Поэтому в нем перезаписаны все четыре необходимых для этого метода события, которые уже знакомы нам из предыдущих примеров этой главы.

Листинг 29.7. Определение класса `Widget` (файл Widget.h)

```
#pragma once

#include <QPoint>
#include <QLabel>

// ============================================================================
class Widget : public QLabel {
Q_OBJECT
private:
    QPoint m_ptDragPos;

    void startDrag();

protected:
    virtual void mousePressEvent(QMouseEvent*     );
    virtual void mouseMoveEvent (QMouseEvent*     );
    virtual void dragEnterEvent (QDragEnterEvent*);
    virtual void dropEvent       (QDropEvent*     );

public:
    Widget(QWidget* pwgt = 0);
};
```

В конструкторе, приведенном в листинге 29.8, для того чтобы наш виджет мог принимать перетаскиваемые объекты, мы значение `true` передаем в метод `setAcceptDrops()`.

Листинг 29.8. Конструктор `Widget()` (файл Widget.cpp)

```
Widget::Widget(QWidget* pwgt/*=0*/) : QLabel(pwgt)
{
    setAcceptDrops(true);
}
```

В методе `startDrag()`(листинг 29.9) мы создаем объект перетаскивания и в качестве источника передаем в его конструктор указатель `this`. В качестве виджета, который будет перетаскиваться, мы передаем указатель `this` в метод `setWidget()`. Для начала процесса перетаскивания вызывается метод `exec()`.

Листинг 29.9. Метод `startDrag()` (файл Widget.cpp)

```
void Widget::startDrag()
{
    WidgetDrag* pDrag = new WidgetDrag(this);
    pDrag->setWidget(this);
    pDrag->exec(Qt::CopyAction);
}
```

В методе `mousePressEvent()` (листинг 29.10) мы запоминаем в атрибуте `m_ptDragPos` возможное начало позиции перетаскивания.

Листинг 29.10. Метод `mousePressEvent()` (файл Widget.cpp)

```
/*virtual*/ void Widget::mousePressEvent(QMouseEvent* pe)
{
    if (pe->button() == Qt::LeftButton) {
        m_ptDragPos = pe->pos();
    }
    QWidget::mousePressEvent(pe);
}
```

Метод `mouseMoveEvent()` (листинг 29.11) нужен для распознавания начала перетаскивания. Он полностью аналогичен одноименному методу, приведенному в листинге 29.1, в описании которого можно найти более подробное его пояснение.

Листинг 29.11. Метод `mouseMoveEvent()` (файл Widget.cpp)

```
/*virtual*/ void Widget::mouseMoveEvent(QMouseEvent* pe)
{
    if (pe->buttons() & Qt::LeftButton) {
        int distance = (pe->pos() - m_ptDragPos).manhattanLength();
        if (distance > QApplication::startDragDistance()) {
            startDrag();
        }
    }
    QWidget::mouseMoveEvent(pe);
}
```

Метод `dragEnterEvent()` (листинг 29.12) разрешает прием перетаскиваемого объекта только в том случае, если его тип совпадает с `"application/widget"`, для проверки чего вызывается метод `hasFormat()`. Строку идентификации типа возвращает статический метод `WidgetMimeData::mimeType()`.

Листинг 29.12. Метод `dragEnterEvent()` (файл Widget.cpp)

```
/*virtual*/ void Widget::dragEnterEvent(QDragEnterEvent* pe)
{
    if (pe->mimeData()->hasFormat(WidgetMimeData::mimeType())) {
        pe->acceptProposedAction();
    }
}
```

После сбрасывания мы вызываем метод `mimeData()` объекта события для получения указателя на объект перетаскивания (листинг 29.13). Еще раз, при помощи динамического преобразования типа, убеждаемся в том, что объект создан от класса `WidgetMimeData`. Если все прошло удачно, то получаем указатель сброшенного виджета вызовом метода `widget()`. И в завершение выводим информацию о его имени при помощи метода `setText()`.

Листинг 29.13. Метод `dropEvent()` (файл Widget.cpp)

```
/*virtual*/ void Widget::dropEvent(QDropEvent* pe)
{
    const WidgetMimeData* pmmd =
        dynamic_cast<const WidgetMimeData*>(pe->mimeData());
    if (pmmd) {
        QWidget* pwgt = pmmd->widget();
        QString str("Widget is dropped\n ObjectName:%1");
        setText(str.arg(pwgt->objectName()));
    }
}
```

Резюме

Основной задачей буфера обмена является поддержка простейшей формы обмена информацией как между разными приложениями, так и между открытыми в них окнами (или разными областями одного и того же окна). При этом программы могут записывать данные в буфер обмена и читать их из него.

Хорошее приложение характеризует интуитивность в обращении. Использование технологии перетаскивания (drag & drop) — верный шаг к достижению этой цели. Drag & drop — это процедура обмена данными как между разными приложениями, так и между открытыми в них окнами (или разными областями одного и того же окна). В отличие от буфера обмена, ее принцип основывается на перетаскивании объектов из одного места в другое. Объектами могут быть файлы, текст или иные типы данных. Перетаскиваемые данные могут быть сразу автоматически обработаны принимающим приложением, если оно поддерживает типы перетаскиваемых объектов. Например, Проводник ОС Windows часто является источником для перетаскивания списка файлов. И чтобы перетащить объект, нужно лишь просто нажать

на него левой кнопкой мыши, не отпуская кнопки, переместить его в нужное место и отпустить кнопку.

Qt предоставляет класс `QDrag` для транспортировки данных посредством drag & drop и класс `QMimeData` для размещения этих данных.

Чтобы из виджета можно было перетаскивать объекты, необходимо переопределить методы `mousePressEvent()` и `mouseMoveEvent()`. Для получения перетаскиваемых объектов необходимо переопределить методы `dragEnterEvent()` и `dropEvent()`, а также вызвать (обычно это делается в конструкторе) метод `setAcceptDrops()`, передав в него значение `true`.

Qt позволяет создавать свои собственные типы данных для перетаскивания.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/29-510/** или с помощью следующего QR-кода (рис. 29.5):



Рис. 29.5. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 30

Интернационализация приложения

> Штирлиц просматривал электронную почту. Незаметно входит Мюллер. У Штирлица на экране бессмысленный набор символов. «Шифровка!!!» — подумал Мюллер. «КОИ-8» — подумал Штирлиц.

Современные коммуникационные технологии до предела сокращают не только время, но и расстояния. Наш огромный мир плотно опутан средствами коммуникации. Но именно эти современные технологии накладывают дополнительные требования на реализацию программ. Уже не обойтись без поддержки *интернационализации*, то есть возможности выбора языка интерфейса, чтобы каждый пользователь мог работать с приложением на своем родном языке. Учитывая, что население нашей планеты Земля в 2017 году превысило 7,5 миллиарда человек, программа, интерфейс которой поддерживает только один язык, даже если на этом языке говорят 100 миллионов человек, упускает до 98% возможных ее пользователей во всем мире. Подумайте, ведь вы проделали большую работу, создавая свою программу, инвестировали в нее время, силы и средства, и теперь, когда дело дошло до ее «публики», вы остаетесь довольствоваться всего лишь двумя процентами от ее возможного количества? Итак, интернационализация чрезвычайно важна!

Для поддержки интернационализации в ваших Qt-приложениях требуется проделать следующие шаги:

1. Подготовить приложение к интернационализации (если оно еще не готово).
2. Запустить утилиту lupdate.
3. Перевести команды созданной программы (в этом может помочь приложение `Qt Linguist`).
4. Запустить утилиту lrelease для генерации двоичных файлов переводов, которые впоследствии будут загружаться в объект класса `QTranslator`.

Рассмотрим эти шаги подробнее.

Подготовка приложения к интернационализации

Работая над приложением, надо стремиться к тому, чтобы ваши приложения были готовы к интернационализации. На практике это значит, что при программировании все строки, предназначенные для вывода на экран, нужно заключать в статический метод `tr()`, определенный в классе `QObject`. Текст, переданный в этот метод, является источником для перевода.

> ***Комментарии и текстовые строки программы пишите на английском языке!***
>
> Настоятельно рекомендуется в качестве источника для перевода использовать текст на английском языке. Во-первых, символы этого языка входят в набор нерасширенной кодировки ASCII (так называемой latin1), что позволит вам и/или вашей команде использовать любые средства для редактирования кода. Во-вторых, английский язык является негласно принятым языком для общения в интернациональных командах. Даже если вы не работаете в такой команде, то, быть может, Вам придется со временем обратиться за вопросом или консультацией к кому-нибудь, и совсем не обязательно, что этот человек будет знать русский язык. В том же случае, когда комментарии и текстовые строки программы написаны на английском языке, подобных проблем можно будет избежать.

Первым параметром в метод `tr()` передается текстовая строка. Второй параметр — комментарий, он предоставляет дополнительную информацию переводчику, не является обязательным и может быть проигнорирован. Например:

```
setText(tr("Yes"));
```

Метод `tr()` также можно использовать в подстановках с методом `QString::arg()`:

```
setText(tr("User Name: %1").arg(strName));
```

Как было отмечено чуть ранее, во второй параметр метода `tr()` можно вставлять уточняющие комментарии для переводчика — это может быть, например, очень полезно, когда слово, в зависимости от контекста, имеет разные значения. Например, русское слово «замок» в зависимости от ударения может означать либо приспособление для закрытия дверей, либо архитектурное сооружение. Тогда комментарий для переводчика гарантирует, что перевод будет сделан правильно. Например:

```
QLabel lbl(tr("Location", "On the map"));
```

Можно комментарий для переводчика оформить и при помощи комментария C++. Например:

```
//: On the map
QLabel lbl(tr("Location"));
```

Важно помещать в функцию `tr()` не только текст, предназначенный для отображения на экране, но и обозначения валюты, комбинации «горячих» клавиш и другие отличительные особенности, присущие конкретной стране. Например:

```
QKeySequence keyseq(tr("CTRL+L"));
```

Одна из частых трудностей перевода заключается в определении множественного и единственного числа. Их образование в разных языках происходит по-разному — для этой цели в методе `tr()` предусмотрен третий параметр, предназначенный для передачи числа. Его использование может выглядеть следующим образом:

```
int n = getDays();
QLabel lbl(tr("day(s):", "Plural or singular", n));
```

В программе желательно избегать конкатенации строк для перевода. То есть предложения для перевода должны быть по возможности всегда целиком. Представьте себе, что в программе соединяются две строки, первая из которых «Уважаемый», и к ней должны присоединиться строки с именами и фамилиями. В этом случае все будет в порядке только до тех пор, пока среди всех имен и фамилий не попадется парочка женских.

Также избегайте надписей в картинках — картинки должны быть нейтральны к языку. Согласитесь, будет очень странно, если в программе, полностью переведенной на русский

язык, будут всплывать окна с картинками, надписи в которых представлены на английском или китайском. Но если есть необходимость использования картинок с языковыми надписями, и этого никак избежать нельзя, то тогда будет необходимо для всех поддерживаемых языков предоставить отдельную картинку и включить ее в ресурс программы в отдельной языковой секции. Например, для русского языка это секция "ru":

```
<qresource>
    <file alias="myimg.png">myimg.png</file>
</qresource>
<qresource lang="ru">
    <file alias="myimg.png">myimg_ru.png</file>
</qresource>
```

Как мы видим, интернационализация приложения, на самом деле, это гораздо больше, чем просто перевод текстов с одного языка на другой, она включает в себя также сложный процесс локализации.

> ***Интернационализация и локализация***
>
> Интернационализация обозначается так: i18n — это сделано, чтобы каждый раз не писать длинное слово «internationalization», — просто указываются начальная и конечная буквы, а между ними помещается число 18, обозначающее количество пропущенных в слове букв. Аналогичным приемом пользуются для обозначения слова «localization» (локализация): l10n.

Под локализацию попадают, например, адаптация разнообразных форматов дат и цифр. Например, представление числа 2876,56 в английском варианте — 2,876.56, а в немецком — 2.876,56. Строки преобразуются в числа и наоборот в соответствии с языком и географическим расположением. Например:

```
QLocale english(QLocale::English, QLocale::UnitedKingdom);
QString str = english.toString(2876.56); // str = 2,876.56
QLocale german(QLocale::German, QLocale::Germany);
str = german.toString(2876.56); //str = 2.876,56
```

Важно также учитывать, что в некоторых языках, например в арабском, написание слов осуществляется справа налево. А это значит, что после установки перевода надо будет еще перевернуть размещение виджетов в обратную сторону. Это можно сделать вызовом метода из глобального объекта приложения следующим образом:

```
qApp->setLayoutDirection(Qt::RightToLeft);
```

Утилита lupdate

После передачи в метод `tr()` всех нужных строк текста можно приступать к созданию файлов перевода. Для этого необходимо воспользоваться специальной утилитой lupdate. При этом назначение метода `tr()` сводится к указанию утилите в программном коде текста, который нуждается в переводе. Из строк, переданных в метод `tr()`, будут созданы отдельные TS-файлы (Translation Source, источник перевода) — файлы переводов. Чтобы создать для программы, приведенной в листинге 30.1 и показанной на рис. 30.1, файлы русского и немецкого переводов и *локализовать* программу, нужно внести в проектный файл Hello.pro строку:

```
TRANSLATIONS = main_ru.ts main_de.ts
```

Затем вызвать утилиту lupdate и передать ей в параметре проектный файл:

```
lupdate Hello.pro
```

> ***Утилита lupdate***
>
> Утилита lupdate ничего не удаляет и не стирает, все сделанные переводы остаются без изменений.

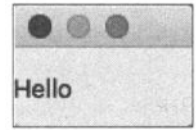


Рис. 30.1. Сообщение на английском языке

Листинг 30.1. Файл для перевода (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QLabel       lbl(QObject::tr("Hello"));

    lbl.show();
    return app.exec();
}
```

В результате будут созданы файлы переводов в формате XML, содержащие следующий код:

```
<!DOCTYPE TS><TS>
<context>
    <name>QObject</name>
    <message>
        <source>Hello</source>
        <translation type="unfinished"></translation>
    </message>
</context>
</TS>
```

В этот файл можно «от руки» внести перевод. Для этого в теге `translation` надо удалить атрибут `type` вместе с его значением и добавить перевод в текстовую зону тега. Например, для русского варианта тег `translation` будет выглядеть так:

```
<translation>Здравствуй</translation>
```

Программа Qt Linguist

Одна из самых важных задач в интернационализации — это переводы. Нужно предоставить целую серию переводов для всех строк вашего приложения и для каждого поддерживаемого языка. Можно, конечно, вносить переводы и вручную, как мы только что показали, но го-

раздо удобнее воспользоваться специально предназначенной для этого программой. Программа Qt Linguist входит в пакет Qt и предоставляет более удобный способ для редактирования файлов переводов. Применение этой программы незаменимо в больших проектах. В программу можно загружать и переводить с ее помощью сразу несколько файлов одновременно, и это очень удобно. Для начала редактирования требуется запустить программу, передав ей в качестве параметра файл перевода или сразу несколько файлов. Следующая команда запускает программу Qt Linguist и загружает в нее файл русского перевода (рис. 30.2):

```
linguist main_ru.ts
```

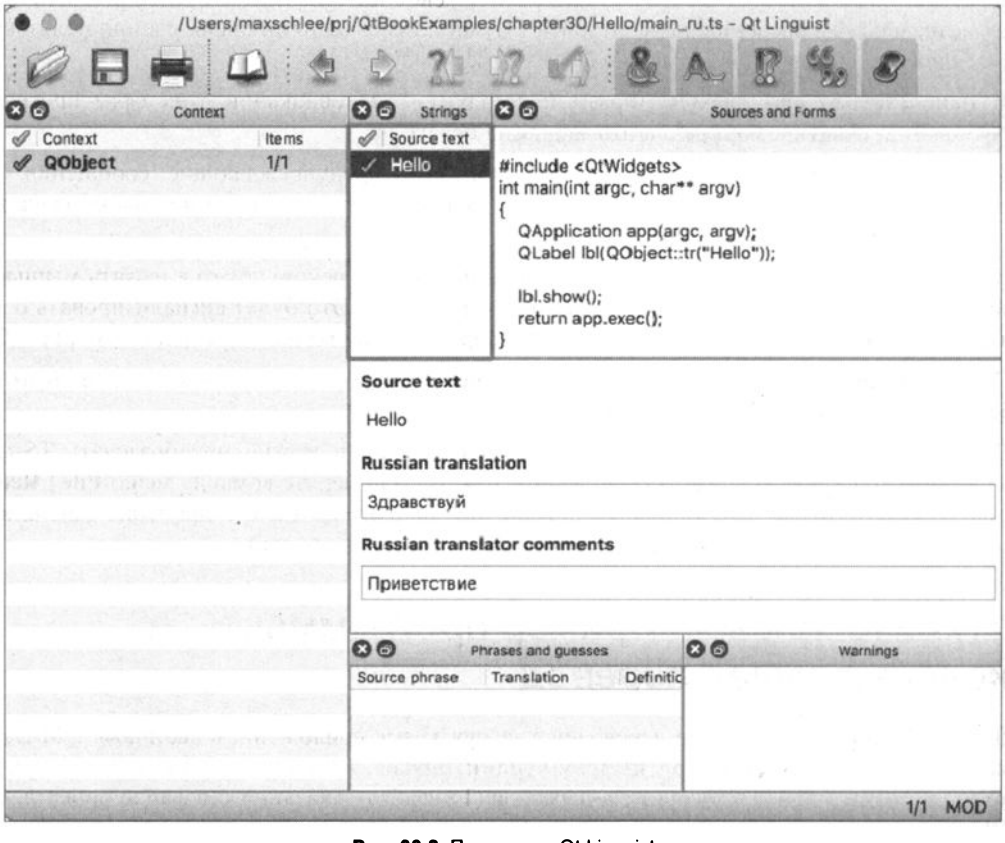


Рис. 30.2. Программа Qt Linguist

Для загрузки можно также воспользоваться диалоговым окном открытия файлов из самой программы. Для одновременного открытия и для перевода сразу нескольких файлов их нужно пометить в этом диалоговом окне, придерживая клавишу <Ctrl> (для выборочной пометки) или <Shift> (для последовательной группы файлов).

Основное окно приложения Qt Linguist содержит ряд окон: **Context**, **Strings**, **Sources and Forms**, **Source text**, **Phrases and guesses** и **Warnings**. Немного остановимся на их назначениях:

- окно **Context** показывает группу элементов для перевода, то есть содержимое нашего TS-файла. Окно имеет табличную структуру. Первая колонка символизирует степень завершенности перевода для всей группы. Во второй колонке (**Context**) отображается имя

класса, который представляет группу для перевода. А третья колонка (**Item**) показывает общее количество элементов, входящих в группу (на втором месте), и сколько из них переведено (на первом месте);

- окно **Strings** (Строки), как и предыдущее окно, имеет табличную структуру и представляет сами элементы для перевода. Первая колонка символизирует степень завершенности перевода. Во второй колонке показаны все элементы группы для перевода;
- окно **Source and Forms** (Исходный код и формы) показывает исходный файл кода, в котором расположена текущая фраза для перевода;
- окно **Source text** (Исходный текст) отображает исходный текст для перевода. Окно имеет два поля, в первое из которых вносится перевод на нужный язык, а второе поле предназначено для введения комментариев. Этими комментариями очень удобно пользоваться, чтобы уточнить смысл слов и фраз для перевода;
- окно **Phrases and guesses** (Фразы и догадки) — на основании этих фраз программа может автоматически предлагать варианты перевода;
- окно **Warnings** (Предупреждения) выводит все предупреждающие сообщения. Эти сообщения появляются не просто так, и поэтому очень важно, чтобы вы обращали на них внимание.

После того как вы снабдили переводом слово или фразу, рекомендуется нажать мышью на знак вопроса, после чего он сменится символом галочки. Это будет сигнализировать о том, что перевод сделан, и повысит визуальное восприятие работы над переводом (см. рис. 30.2).

После завершения работы над переводом его необходимо сохранить в файл. Для этого выберите команду меню **File | Save** (Файл | Сохранить).

Для дальнейшего использования перевода в приложении можно преобразовать TS-файл в QM-файл (Messages file, файл сообщений). Для этого выберите команду меню **File | Release** (Файл | Релиз), после чего откроется стандартное диалоговое окно сохранения файла, в котором нужно указать имя файла и нажать кнопку **Save** (Сохранить).

Утилита lrelease. Пример программы, использующей перевод

Для конвертирования файлов переводов в загружаемые приложением двоичные QM-файлы можно воспользоваться одной из трех утилит: lupdate, Qt Linguist и lrelease. Следующая команда создаст для каждого файла перевода (TS-файла), указанного в файле проекта, свой QM-файл:

```
lrelease Hello.pro
```

Полученные QM-файлы передаются в объект класса `QTranslator`. Объект класса `QTranslator` отвечает за перевод текстов с одного языка на другой. Этот перевод выполняется с помощью QM-файла, загруженного вызовом метода `load()` в объект класса `QTranslator`. Загрузку других QM-файлов можно осуществить в любой момент исполнения программы. Программа, приведенная в листинге 30.2, отображает на экране на русском языке сообщение, соответствующее английскому «Hello» (рис. 30.3).

В листинге 30.2 создается объект класса `QTranslator`. Вызовом метода `load()` в него загружается файл перевода main_ru.qm, указанный в первом параметре этого метода. Второй параметр задает каталог, содержащий файлы переводов. В нашем случае, вторым параметром

передается строка, содержащая точку, — это говорит о том, что QM-файлы находятся в том же каталоге, что и само приложение. Вызов метода `installTranslator()` из объекта класса `QApplication` применяет созданный объект переводчика ко всему приложению.



Рис. 30.3. Переведенное на русский язык сообщение

Листинг 30.2. Программа, отображающая на экране сообщение на русском языке (файл main.cpp)

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QTranslator translator;
    translator.load("main_ru.qm", ".");
    app.installTranslator(&translator);

    QLabel lbl(QObject::tr("Hello"));
    lbl.show();
    return app.exec();
}
```

В этом примере мы загрузили только файл собственного перевода, но в более сложных проектах используются компоненты Qt, такие как, например, стандартные диалоговые окна, контекстные меню для навигации в WebEngine и т. п. Все эти компоненты тоже должны быть в приложении переведены. Для этого QM-файлы переводов Qt тоже необходимо загрузить дополнительно до или после загрузки собственных переводов. Эти файлы находятся в каталоге <Qt>/translations. Программно этот путь можно получить следующим вызовом: `QLibraryInfo::location(QLibraryInfo::TranslationsPath)`. Если эти файлы указать в файле ресурсов, то загрузка файлов перевода в приложении может выглядеть следующим образом: сначала загружаем переводы для Qt, а потом — для приложения (листинг 30.3).

Листинг 30.3. Загрузка переводов для Qt и собственного перевода

```
QTranslator qtTranslator;
qtTranslator.load(QString(":/translations/qt_") +
                  QLocale::system().name());
qApp->installTranslation(&qtTranslator);

QTranslator appTranslator;
appTranslator.load(QString(":/translations/app_") +
                   QLocale::system().name());
appTanslator.load(QString(":/translations/main_") +
                   QLocale::system().name());
qApp->installTranslator(&appTranslator);
```

Всегда помещайте и загружайте QM-файлы переводов из ресурсов, как это показано в листинге 30.3. Встраивание QM-файлов в исполняемый модуль приложения не только снижает

количество файлов в поставке приложения, но и исключает риск их случайной потери или удаления.

Смена перевода в процессе работы программы

Хорошим тоном считается предоставление пользователю возможности изменять язык интерфейса с помощью выбора пункта меню или в диалоговом окне настройки приложения. И тут появляются три варианта того, как это можно сделать. Первый — самый простой. После того как пользователь выбрал необходимый ему язык, нужно записать его в объекте установок `QSettings` и попросить пользователя перезапустить программу. При новом старте программы следует просто считать новый язык из объекта установок и загрузить нужный перевод.

Второй способ немного сложнее, и он не требует завершения работы приложения с явным его перезапуском пользователем. Способ заключается в том, что само приложение после смены языка загружает его, выполняет уничтожение основного окна виджета программы и создает новое. Для подобной операции нужен дополнительный объект, который будет получать от виджета основного окна сигнал о смене языка и в ответ на это уничтожать его и создавать новое окно. Самый сложный момент заключается в том, что пользователь не должен заметить подмены, а значит, вновь созданное окно должно полностью соответствовать уничтоженному окну и содержать те же данные даже в том случае, если пользователь не успел их сохранить. Этого можно достичь с использованием объекта настроек класса `QSettings`.

В первом и втором способах перезагрузка и новое создание виджетов производились потому, что для перевода строкам должны были быть присвоены новые значения, которые в основном выполняются из конструкторов виджетов. Это и был самый уязвимый момент, так как конструктор мы не можем перезапускать как обычный метод. Вот почему мы перезапускали приложение или создавали вызовом конструктора виджет заново.

Идея третьего способа заключается в том, чтобы собрать присвоение строк в одном отдельном методе. Такой способ наиболее предпочтителен, поскольку в этом случае приложение после загрузки файлов перевода просто заново присваивает объектам строк новые текстовые значения. Как только происходит загрузка нового переводчика, методом `QCoreApplication::installTranslator()` генерируется событие `QEvent::LanguageChange`, которое получают все объекты классов, унаследованных от класса `QWidget`. После этого из метода события нужных виджетов можно вызвать метод, предназначенный для присвоения строкам новых значений. Принцип этого подхода демонстрирует листинг 30.4.

Листинг 30.4. Присвоение строкам актуально выбранного перевода

```
void MyLabel::changeEvent(QEvent* pe)
{
    if (pe->type() == QEvent::LanguageChange) {
        retranslateUi();
    }
    QWidget::changeEvent(pe);
}

void MyLabel::retranslateUi()
{
```

```
    setWindowTitle(tr("Current Language"));
    setText(tr("Hello"));
}
```

В листинге 30.4 при загрузке нового объекта переводчика вызывается метод события `QWidget::changeEvent()`, из которого будет вызван метод `retranslateUi()`. В этом методе произойдет присвоение новых строк перевода для заголовка окна и для текста надписи.

Для переключения языков можно в специально предназначенном для этого виджете создать отдельный метод, который может выглядеть следующим образом:

```
void MyProgram::switchLanguage(int n)
{
    QTranslator translator;

    switch (n) {
    case RUSSIAN:
        translator.load("myprogram_ru.qm",".");
        break;
    case GERMAN:
        translator.load("myprogram_de.qm",".");
        break;
    }
    qApp->installTranslator(&translator);
}
```

В случае, если пользователь еще не успел определиться с выбором языка (например, при первом запуске программы), также нелишне будет установить текущий язык в соответствии с выбранной на платформе локализацией. Выбранную локализацию можно узнать при помощи статического метода `QLocate::system()`. Он возвращает объект класса `QLocate`, а вызов метода `name()` этого объекта возвращает нам строку вида `"язык_СТРАНА"`. Например, для США эта строка будет выглядеть следующим образом: `"en_US"`.

> ***Код языка и код страны***
>
> Именование кода языка соответствует стандарту ISO 639-1, а код страны — ISO 3166-1.

В соответствии со строкой `"язык_СТРАНА"` и можно будет загрузить необходимый файл перевода. Приведем для наглядности небольшой отрывок, позволяющий это сделать:

```
QTranslator translator;
QString str = QLocale::system().name();
if (str == "en_US") {
   translator.load("myprogram_us.qm","."); // загружаем английский перевод
                                           // для США
}
else if (str == "de_CH") {
   translator.load("myprogram_de.qm","."); // загружаем немецкий перевод
                                           // для Швейцарии
}
else if (str == "ru") {
   translator.load("myprogram_ru.qm","."); // загружаем русский перевод
}
qApp->installTranslator(&translator);
```

Завершающие размышления

Интернационализация программ — это достаточно дорогое удовольствие, и нельзя недооценивать те временные и финансовые затраты, которые могут у вас возникнуть с решением интернационализировать свои программы. Большую долю этих затрат, конечно же, будут занимать переводы. С одной стороны, существуют автоматические средства перевода, такие как Google Translate (**http://translate.google.com**) или Яндекс.Переводчик (**http://translate.yandex.ru**), и хотя качество их перевода с каждым годом становится все лучше и лучше, но до перевода, сделанного человеком, им еще пока далеко. Тем не менее, для инициализирующего (предварительного, подстрочного) перевода их все же использовать можно. Таким путем вы можете облегчить процесс предстоящего перевода для профессионала или для сообщества любителей вашей программы. Вам нужно только сделать так, чтобы они могли принимать участие в самом процессе перевода. Для этого вовсе не обязательно давать им для работы программу Qt Linguist. Вы можете выложить файлы подготовленных инициализирующих переводов в Интернет, например, с помощью специально разработанного для этой цели проекта Pootle (**http://translate.sourceforge.net**). Окно, представленное на рис. 30.4, показывает пример его использования.



Рис. 30.4. Использование Pootle как средства для перевода (взято с sourceforge.net)

Файлы переводов могут быть разных форматов, то есть не только с расширением ts, но также и po, и xlf. Проводить конвертирование из одного формата в другой поможет утилита lconvert, которая поставляется вместе с Qt. Помимо всего прочего, как бы это странно ни звучало, в идеале нужно также делать и перевод с английского языка на английский. Это необходимо в тех случаях, когда этот язык используется в исходном коде программы, и тексты пишутся программистами, которые могут не настолько тонко чувствовать язык, как профессиональный переводчик. Поэтому у профессионального переводчика всегда должна быть возможность, не имея доступа к исходному коду программы, исправить все неточности и «шероховатости» исходного языка.

Резюме

Всего четыре шага отделяют обычное приложение от приложения с поддержкой интернационализации. Статический метод `tr()` класса `QObject` играет сразу две роли. Во-первых, он помогает утилите lupdate находить в программе текст, нуждающийся в переводе. Во-вторых, он является методом для перевода текста. Этот метод очень эффективен в поиске строк перевода, поэтому не нужно бояться снижения производительности программ. Процесс интернационализации приложения — это не только перевод текстов, но и еще локализация, в которую входит правильное отображение и управление: датой и временем, валютой, форматом чисел, размером бумаги, измерительными величинами, направлением написания текста и т. д.

Чтобы подготовить перевод, нужно запустить программу lupdate, которая создаст файл перевода с расширением ts (TS-файл) на основании исходного кода в файлах на языке C++. Этот файл необходимо перевести «от руки» или с помощью программы Qt Linguist. Для использования перевода в программе нужно переработать файлы перевода в QM-файлы. Такое преобразование выполняется при помощи утилиты lrelease.

Загрузка QM-файлов в объект класса `QTranslator` осуществляется методом `load()`. После этого объект перевода устанавливается в приложение с помощью метода `installTranslator()` класса `QCoreApplication`.

Изменение языка в процессе работы программы можно выполнить тремя разными способами, один из которых требует перезапуска приложения, а два других — нет.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/30-510/** или с помощью следующего QR-кода (рис. 30.5):



Рис. 30.5. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 31

Создание меню

— А что у нас сегодня в меню (Menu)?
— Филе! (File)

Виктор Свитковский

Меню является важной и неотъемлемой частью практически любого приложения. Главное меню, как правило, находится в верхней части главного окна приложения и представляет собой секцию для расположения большого количества команд, из которых пользователь может выбирать нужную ему. В приложениях используются меню четырех основных типов:

- меню верхнего уровня;
- всплывающее меню;
- отрывное меню;
- контекстное меню.

В библиотеке Qt меню реализуется классом `QMenu`. Этот класс определен в заголовочном файле QMenu. Основное назначение класса — это размещение команд в меню. Каждая команда представляет объект действия — класс `QAction` (см. *главу 34*). Все действия или сами команды меню могут быть соединены со слотами для исполнения соответствующего кода при выборе команды пользователем. Например, если пользователь выделил команду меню, то и меню, и объекты действий отправляют сигнал `hovered()`, и если вам потребуется в этот момент выполнить какие-либо действия, то их нужно соединить с соответствующим слотом.

«Анатомия» меню

Пользователю будет легче привыкнуть к работе с новой программой, если ее меню будет похоже на меню других программ. На рис. 31.1 показаны составляющие типичного меню.

- Основной отправной точкой меню является *меню верхнего уровня*. Оно представляет собой постоянно видимый набор команд, которые, в свою очередь, могут быть выбраны при помощи указателя мыши или клавиш клавиатуры (клавиши <Alt> и клавиш управления курсором). Команды меню верхнего уровня предназначены для отображения *выпадающих меню*, поэтому их не следует использовать для других целей, так как это может изрядно озадачить пользователя. Старайтесь логически группировать команды и объединять их в одном выпадающем меню, которое, в свою очередь, будет вызываться при выборе соответствующей команды меню верхнего уровня. Класс `QMenuBar` отвечает за меню верхнего уровня и определен в заголовочном файле QMenuBar.

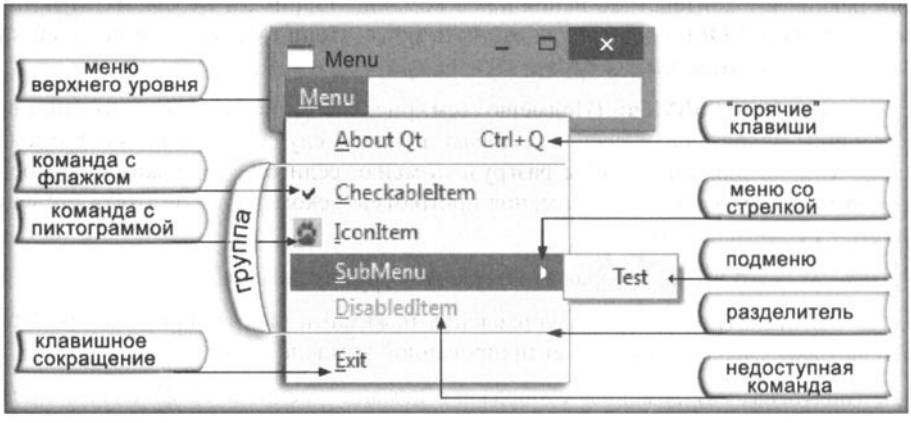


Рис. 31.1. «Анатомия» меню

- «Горячие» клавиши — это, по сути, определенные комбинации клавиш, с помощью которых выполняется то же самое действие, что и при выборе соответствующей команды меню. Например, для отображения окна **About Qt** (О Qt) в примере, показанном на рис. 31.1, используется комбинация клавиш <Ctrl>+<Q>. Старайтесь использовать для «горячих» клавиш стандартные комбинации. Некоторые из них приведены в табл. 31.1.

Таблица 31.1. *Некоторые стандартные комбинации для «горячих» клавиш*

Клавиши	Описание	Клавиши	Описание
<Esc>	Отменить текущую операцию	<Ctrl>+<Z>	Отменить предыдущее действие
<F1>	Вызвать файл помощи	<Ctrl>+<X>	Вырезать
<Shift>+<F1>	Вызвать контекстную помощь	<Ctrl>+<C>	Копировать
<Ctrl>+<N>	Создать	<Ctrl>+<V>	Вставить
<Ctrl>+<O>	Открыть	<Ctrl>+<F4>	Закрыть активный документ MDI-приложения
<Ctrl>+<P>	Печать	<Ctrl>+<F6>	Активировать окно просмотра следующего документа MDI-приложения
<Ctrl>+<S>	Сохранить	<Shift>+<Ctrl>+<F6>	Активировать окно просмотра предыдущего документа MDI-приложения

- По возможности, для всех пунктов меню должны быть определены *клавиши быстрого вызова*. Это позволит пользователю выбирать команды не только при помощи мыши, но и при помощи клавиатуры, нажав подчеркнутую букву (в названии команды) совместно с клавишей <Alt>. Например, для выбора команды **Exit** (Выход) нужно нажать <Alt>+<E> (см. рис. 31.1). Основные отличия подобного рода комбинаций для быстрого вызова от «горячих» клавиш состоят в следующем:
  - такие комбинации состоят из клавиши <Alt> и буквенной клавиши;
  - они встречаются не только в меню, но и в диалоговых окнах;

- они реализуют контекстное исполнение команд. Например, чтобы вызвать диалоговое окно **About Qt** (О Qt), нужно открыть меню **Menu** (Меню) комбинацией клавиш <Alt>+<M>, а затем нажать <Alt>+<A>.

- Стрелка у команды **SubMenu** (Подменю) (см. рис. 31.1) говорит о том, что при выборе этой команды появится *вложенное подменю*, в нашем случае — **Test** (Тест). Вложенное подменю удобно для того, чтобы разгрузить меню, если оно содержит большое количество команд. Для удобства понимания программы рекомендуется, чтобы степень вложенности не превышала двух.
- *Разделитель* — это черта, которая отделяет одну группу команд от другой.
- Команда с *флажком* служит для управления режимами работы приложения. Установленный флажок сигнализирует об активированной команде меню.
- *Значок* (пиктограмма) команды отображает команду меню в графическом виде. Это очень хороший прием для дополнительной иллюстрации действий самой команды.
- Иногда встречаются команды, которые нельзя исполнить в определенный момент времени. В таких случаях приложение должно делать такие команды *недоступными*, то есть сделать их выбор невозможным. Недоступные команды меню отображаются другим цветом — как правило, серым.

Также помните, что в случаях, когда команда меню вызывает диалоговое окно, в конец ее названия принято добавлять троеточие. Это правило, правда, не распространяется на вызов простых окон сообщений.

В листинге 31.1 реализуется меню, показанное на рис. 31.1.

Листинг 31.1. Пример реализации меню (файл main.cpp)

```
#include <QtWidgets>


int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QMenuBar mnuBar;
    QMenu*   pmnu   = new QMenu("&Menu");

    pmnu->addAction("&About Qt",
                    &app,
                    SLOT(aboutQt()),
                    Qt::CTRL + Qt::Key_Q
                   );

    pmnu->addSeparator();

    QAction* pCheckableAction = pmnu->addAction("&CheckableItem");
    pCheckableAction->setCheckable(true);
    pCheckableAction->setChecked(true);

    pmnu->addAction(QPixmap(":/img4.png"), "&IconItem");
```

```
    QMenu* pmnuSubMenu = new QMenu("&SubMenu", pmnu);
    pmnu->addMenu(pmnuSubMenu);
    pmnuSubMenu->addAction("&Test");

    QAction* pDisabledAction = pmnu->addAction("&DisabledItem");
    pDisabledAction->setEnabled(false);

    pmnu->addSeparator();

    pmnu->addAction("&Exit", &app, SLOT(quit()));

    mnuBar.addMenu(pmnu);
    mnuBar.show();

    return app.exec();
}
```

Чтобы создать полноценное меню, требуется к каждой команде меню верхнего уровня присоединить соответствующее *всплывающее меню*. За всплывающие меню отвечает класс `QMenu`. Итак, для создания меню необходимо иметь виджет класса `QMenuBar` — указатель `pmnuBar` и, по меньшей мере, один виджет класса `QMenu` — указатель `pmnu` (см. листинг 31.1).

Для добавления всплывающего меню к меню верхнего уровня нужно передать в метод `addAction()` название команды. Каждый метод `addAction()` возвращает указатель на объект действия `QAction`. Пользуясь этим указателем, можно получить доступ к команде меню. Вызов метода `setCheckable()` объекта действия (в нашем случае `pCheckableAction`) предоставляет возможность установки флажка. Дальнейший вызов метода `setChecked()` устанавливает состояние флажка. В нашем примере в этот метод передается значение `true`, а это значит, что флажок будет находиться во «включенном» состоянии.

Метод `addAction()` принимает четыре параметра. Первый задает название команды меню, в котором можно указать букву для быстрого вызова, поставив перед ней символ `&`. Не упускайте из виду, что разные команды меню должны использовать разные буквы для быстрого вызова, в противном случае одна из них окажется недоступной. Вторым параметром передается указатель на объект, содержащий слот, который вызывается при выборе этой команды. Сам слот передается третьим параметром. Последний параметр задает комбинацию для «горячей» клавиши. В нашем примере для отображения диалогового окна **About Qt** (О Qt) используется комбинация клавиш <Ctrl>+<Q>. Нажатие этой комбинации клавиш приведет к тому же действию, что и выбор соответствующей команды меню, а именно — будет вызван слот объекта приложения `aboutQt()`. Для составления комбинаций «горячих» клавиш можно воспользоваться табл. 14.2.

Вызов метода `addSeparator()` добавляет разделитель в меню.

Указателем на объект действия (`pDisableAction`) можно воспользоваться также и для того, чтобы сделать некоторые из команд меню недоступными — с помощью метода `setEnabled()`.

В метод `addAction()` первым параметром можно передавать объекты растровых изображений для установки значка команды.

Контекстные меню

Визитной карточкой профессионального приложения является наличие контекстного меню. *Контекстное меню* — это меню, которое открывается при нажатии правой кнопки мыши. Для его реализации, как и в случае всплывающего меню, используется класс `QMenu`. Различие состоит лишь в том, что это меню не присоединяется к виджету `QMenuBar`. На рис. 31.2 показано окно программы (листинг 31.2) и контекстное меню, отображаемое при нажатии правой кнопки мыши. В этом меню пользователь может выбрать одну из трех команд: **Red** (Красный), **Green** (Зеленый) или **Blue** (Синий), которые задают соответствующий цвет фона окна.

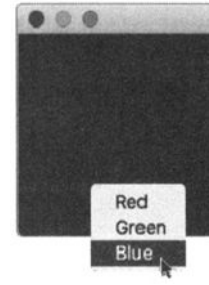


Рис. 31.2. Контекстное меню для выбора цвета окна

В конструкторе класса `ContextMenu` листинга 31.2 создается виджет контекстного меню — указатель `m_pmnu`. С помощью метода `addAction()` добавляются команды меню. Метод `connect()` соединяет сигнал меню `triggered(QAction*)` со слотом `slotActivated(QAction*)`. Сигнал отправляется каждый раз при выборе пользователем одной из команд меню. Этот слот получает указатель на объект действия. Благодаря тому, что наш класс `ContextMenu` унаследован от класса `QTextEdit`, мы можем устанавливать цвет фона при помощи строки в формате HTML — нужно только вызвать метод `QTextEdit::setHtml()`. Сам цвет устанавливается в соответствии с именем выбранной команды, из которого удаляется символ `&`, для чего вызывается метод `QString::remove()`. Строка с цветом записывается в переменную `strColor`.

Показ контекстного меню выполняется из метода обработки события контекстного меню `QWidget::contextMenuEvent()` и должен осуществляться на месте (в координатах) указателя мыши при нажатии ее правой кнопки. Для этого нужно передать в метод `exec()` значение, возвращаемое методом `globalPos()` объекта события контекстного меню. Этот метод возвращает объекты класса `QPoint`, содержащие координаты указателя мыши относительно верхнего левого угла экрана.

Листинг 31.2. Контекстное меню (файл main.cpp)

```
#pragma once

#include <QtWidgets>

// ============================================================================
class ContextMenu : public QTextEdit {
Q_OBJECT
private:
    QMenu* m_pmnu;

protected:
    virtual void contextMenuEvent(QContextMenuEvent* pe)
    {
        m_pmnu->exec(pe->globalPos());
    }
```

```
public:
    ContextMenu(QWidget* pwgt = 0)
       : QTextEdit(pwgt)
    {
        setReadOnly(true);
        m_pmnu = new QMenu(this);
        m_pmnu->addAction("Red");
        m_pmnu->addAction("Green");
        m_pmnu->addAction("Blue");
        connect(m_pmnu,
                SIGNAL(triggered(QAction*)),
                SLOT(slotActivated(QAction*))
               );
    }

public slots:
    void slotActivated(QAction* pAction)
    {
        QString strColor = pAction->text().remove("&");

        setHtml(QString("<BODY BGCOLOR=%1></BODY>").arg(strColor));
    }
};
```

Резюме

Большинство программ поддерживают меню, которые предоставляют пользователю разнообразные возможности выбора команд, управляющих различного рода действиями. Меню можно разделить на четыре основных типа: меню верхнего уровня, всплывающие, отрывные и контекстные меню.

Для всех пунктов меню должны быть определены клавиши для быстрого вызова, символы которых обозначаются знаком подчеркивания в названии команды меню, а для наиболее важных или часто используемых команд — «горячие» клавиши, представляющие собой комбинации клавиш, которые интерпретируются программой как команды меню. Если вызов команды отображает диалоговое окно, то рекомендуется добавлять в конце имени команды три точки. Недоступные команды меню отображаются, как правило, серым цветом, сообщая пользователю о том, что такая команда в данный момент или, вернее, в данной ситуации не может быть выполнена.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/31-510/** или с помощью следующего QR-кода (рис. 31.3):



Рис. 31.3. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 32

Диалоговые окна

Диалог с новой версией Windows, оснащенной искусственным интеллектом:
Windows: Вы действительно хотите удалить этот файл?
Пользователь: Да!
Windows: А почему?

Диалоговое окно — это центральный элемент, обеспечивающий взаимодействие между пользователем и приложением. Этот виджет может содержать ряд опций, изменение которых в ходе работы влечет за собой изменения в работе самой программы. Диалоговые окна всегда являются виджетами верхнего уровня и имеют свой заголовок. Их можно разбить на три основные категории:

- собственные;
- стандартные;
- окна сообщений.

Правила создания диалоговых окон

Диалоговые окна важны в любом приложении, и их создание — это рутина, которую часто приходится выполнять разработчику. Создание диалогового окна, на самом деле, включает в себя гораздо больше, чем просто размещение нужных элементов. Важно обеспечить пользователю возможность интуитивной работы с диалоговым окном, чтобы ему не пришлось тратить время на его изучение, а можно было бы сразу начать действовать. Для обеспечения интуитивной работы необходимо учитывать следующие правила:

- стремитесь к тому, чтобы диалоговое окно не содержало ничего лишнего и было как можно проще. В диалоговом окне настроек программы желательны только основные кнопки, например: **Ok**, **Cancel** (Отмена) и **Apply** (Применить);
- объединяйте виджеты в логические группы, снабжая их прямоугольной рамкой и подписью (см. *главу 8*). Используйте горизонтальные и вертикальные линии для разделения;
- никогда не делайте содержимое диалогового окна прокручивающимся. Если окно содержит много элементов, то постарайтесь разбить их на группы и разместить их с помощью вкладок (см. *главу 11*);
- нежелательно, чтобы вкладки в диалоговом окне занимали более одного ряда — это усложняет поиск;

- избегайте создания диалоговых окон с неизменяемыми размерами. Пользователь всегда должен иметь возможность увеличить или уменьшить размеры окна по своему усмотрению;
- сложные диалоговые окна лучше снабжать дополнительной кнопкой **Help** (Помощь), при нажатии на которую должно открываться окно контекстной помощи;
- команды меню, вызывающие диалоговые окна, должны оканчиваться многоточием, — например: **Open...** (Открыть...). Это делается для того, чтобы пользователь заранее знал, что нажатие команды меню приведет к открытию диалогового окна;
- старайтесь не добавлять меню в диалоговые окна. Меню должны использоваться в окне основной программы;
- по возможности используйте стандартные виджеты, хорошо знакомые пользователям. Не забывайте, что для освоения новых элементов управления может понадобиться дополнительное время;
- для показа настроек избегайте использования цвета. В большинстве случаев текст — лучшая альтернатива. Ведь один и тот же цвет может иметь в разных странах различные смысловые значения. Кроме того, не следует исключать пользователей, неспособных различать цветовые оттенки;
- не забывайте, что пользователь должен работать с диалоговым окном не только с помощью мыши, но и с помощью клавиатуры. Для этого необходимо снабдить все элементы окна клавишами быстрого вызова, которые позволят, нажав букву совместно с клавишей <Alt>, установить фокус на нужном элементе.

Класс *QDialog*

Класс `QDialog` является базовым для всех диалоговых окон, представленных в классовой иерархии Qt (см. рис. 5.1). Хотя диалоговое окно можно создавать при помощи любого виджета, сделав его виджетом верхнего уровня, тем не менее удобнее воспользоваться классом `QDialog`, который предоставляет ряд возможностей, необходимых всем диалоговым окнам. Диалоговые окна подразделяются на две группы:

- модальные;
- немодальные.

Режим модальности и немодальности можно установить, а также определить при помощи методов `QDialog::setModal()` и `QDialog::isModal()` соответственно. Значение `true` означает модальный режим, а `false` — немодальный.

Модальные диалоговые окна

Модальные диалоговые окна обычно используются для вывода важных сообщений. Например, иногда возникают ошибки, на которые пользователь должен отреагировать, прежде чем продолжить работать с приложением. Модальные окна прерывают работу приложения, и для продолжения его работы такое окно должно быть закрыто. В этих случаях модальное диалоговое окно — идеальное средство для привлечения внимания пользователя.

Для блокировки приложения запускается цикл события только для диалогового окна, а остальные события клавиатуры, мыши и других элементов приложения просто игнорируются. Этот цикл запускается вызовом слота `exec()`, который возвращает после закрытия диалого-

вого окна значение целого типа, которое информирует о нажатой кнопке и может равняться QDialog::Accepted или QDialog::Rejected, что соответствует кнопкам **Ok** и **Cancel** (Отмена). Типичным примером модального окна является напоминание пользователю о необходимости сохранения документа перед закрытием приложения. В момент отображения этого окна возможность работы с самим приложением должна быть заблокирована. Принцип вызова модального диалогового окна примерно следующий:

```
MyDialog* pdlg = new MyDialog(&data);
if (pdlg->exec() == QDialog::Accepted) {
    // Пользователь выбрал Accepted
    // Получить данные для дальнейшего анализа и обработки
    Data data = pdlg->getData();
    ...
}
delete pdlg;
```

Немодальные диалоговые окна

Немодальные диалоговые окна ведут себя как нормальные виджеты, не прерывая при своем появлении работу приложения. На первый взгляд может показаться, что применение немодальных диалоговых окон имеет больше смысла, чем модальных, так как в этом случае пользователь обладает большей свободой в своих действиях. Но на самом деле большинство приложений часто нуждается в приостановке до принятия решения пользователем, перед тем как возобновить дальнейшие действия. Тем не менее, немодальные диалоговые окна используются не реже, чем модальные.

Немодальное окно может быть отображено с помощью метода show(), как и обычный виджет. Метод show() не возвращает никаких значений и не останавливает выполнение всей программы. Метод hide() позволяет сделать окно невидимым. Этой возможностью можно воспользоваться, чтобы не создавать каждый раз объект диалогового окна и не удалять его из памяти при закрытии. То есть, ограничение вызовов методами show() и hide() дает возможность отображать диалоговое окно на том же месте, на котором оно было скрыто. Немодальные диалоговые окна необходимо снабжать кнопкой **Close** (Закрыть), чтобы дать возможность пользователю его закрыть.

> ***Немодальные диалоговые окна в Mac OS X***
>
> В Mac OS X используется совсем другой подход при отображении немодальных диалоговых окон. Для того чтобы немодальное диалоговое окно стало видно, одного вызова метода show() недостаточно, поскольку оно будет скрыто окном основного приложения. Поэтому после вызова метода show() должны всегда следовать вызовы raise() и activateWindow().

Создание собственного диалогового окна

Окна программы (листинги 32.1–32.6), показанные на рис. 32.1, иллюстрируют создание собственного диалогового окна. При запуске программы на экране открывается окно с кнопкой **Press Me** (Нажми меня) (см. рис. 32.1, *слева*), нажатие на которую отображает диалоговое окно ввода имени **First Name** (Имя) и фамилии **Last Name** (Фамилия) (см. рис. 32.1, *справа*).

В листинге 32.1 создается виджет класса StartDialog, предназначенный для запуска диалогового окна.

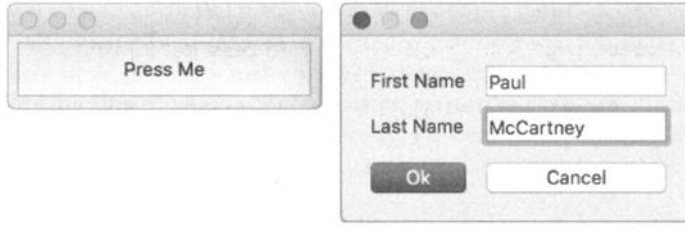


Рис. 32.1. Собственное диалоговое окно

Листинг 32.1. Создание собственного диалогового окна (файл main.cpp)

```
#include <QApplication>
#include "StartDialog.h"

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    StartDialog  startDialog;

    startDialog.show();

    return app.exec();
}
```

Класс `StartDialog` (листинг 32.2) унаследован от класса кнопки `QPushButton`. Сигнал `clicked()` методом `connect()` соединяется со слотом `slotButtonClicked()`. В этом слоте создается объект диалогового окна `InputDialog`, который не имеет предка.

> ***Центрирование диалоговых окон***
>
> Диалоговые окна, не имеющие предка, будут центрироваться на экране. Окна с предками будут отцентрированы относительно предка.

В условии оператора `if` выполняется запуск диалогового окна. После же его закрытия управление передается основной программе, и метод `exec()` возвращает значение нажатой пользователем кнопки. В том случае, если пользователем была нажата кнопка **Ok**, будет отображено информационное окно с введенными в диалоговом окне данными. По завершении метода диалоговое окно нужно удалить самому, так как у него нет предка, который позаботится об этом.

Листинг 32.2. Определение класса `StartDialog` (файл StartDialog.h)

```
#pragma once

#include <QtWidgets>
#include "InputDialog.h"

// ============================================================================
class StartDialog : public QPushButton {
Q_OBJECT
```

```
public:
    StartDialog(QWidget* pwgt = 0) : QPushButton("Press Me", pwgt)
    {
        connect(this, SIGNAL(clicked()), SLOT(slotButtonClicked()));
    }

public slots:
    void slotButtonClicked()
    {
        InputDialog* pInputDialog = new InputDialog;
        if (pInputDialog->exec() == QDialog::Accepted) {
            QMessageBox::information(0,
                                     "Information",
                                     "First Name: "
                                     + pInputDialog->firstName()
                                     + "\nLast Name: "
                                     + pInputDialog->lastName()
                                    );
        }
        delete pInputDialog;
    }
};
```

Для создания своего собственного диалогового окна нужно унаследовать класс `QDialog`, как это и сделано в листинге 32.3. Класс `InputDialog` содержит два атрибута: указатели `m_ptxtFirstName` и `m_ptxtLastName` на виджеты однострочных текстовых полей и два метода, возвращающие содержимое этих полей: `firstName()` и `lastName()`.

Листинг 32.3. Определение класса `InputDialog` (файл InputDialog.h)

```
#pragma once

#include <QDialog>

class QLineEdit;
// =============================================================================
class InputDialog : public QDialog {
    Q_OBJECT
private:
    QLineEdit* m_ptxtFirstName;
    QLineEdit* m_ptxtLastName;

public:
    InputDialog(QWidget* pwgt = 0);

    QString firstName() const;
    QString lastName () const;
};
```

По умолчанию область заголовка диалогового окна содержит кнопку ?, предназначенную для получения подробной информации о назначении виджетов. В нашем примере я решил

пренебречь ею — для этого необходимо установить флаги окна, поэтому вторым параметром передаются флаги `Qt::WindowTitleHint` и `Qt::WindowSystemMenuHint`. Первый устанавливает заголовок окна, а второй добавляет системное меню с возможностью закрытия окна (листинг 32.4).

Модальное диалоговое окно всегда должно содержать кнопку **Cancel** (Отмена). Сигналы `clicked()` кнопок **Ok** и **Cancel** (Отмена) соединяются со слотами `accept()` и `rejected()` соответственно. Это делается для того, чтобы метод `exec()` возвращал при нажатии кнопки **Ok** значение `QDialog::Accepted`, а при нажатии на кнопку **Cancel** (Отмена) — значение `QDialog::Rejected`.

Листинг 32.4. Конструктор `InputDialog()` (файл InputDialog.cpp)

```
InputDialog::InputDialog(QWidget* pwgt/*= 0*/)
     : QDialog(pwgt, Qt::WindowTitleHint | Qt::WindowSystemMenuHint)
{
    m_ptxtFirstName = new QLineEdit;
    m_ptxtLastName  = new QLineEdit;

    QLabel* plblFirstName    = new QLabel("&First Name");
    QLabel* plblLastName     = new QLabel("&Last Name");

    plblFirstName->setBuddy(m_ptxtFirstName);
    plblLastName->setBuddy(m_ptxtLastName);

    QPushButton* pcmdOk     = new QPushButton("&Ok");
    QPushButton* pcmdCancel = new QPushButton("&Cancel");

    connect(pcmdOk, SIGNAL(clicked()), SLOT(accept()));
    connect(pcmdCancel, SIGNAL(clicked()), SLOT(reject()));

    //Layout setup
    QGridLayout* ptopLayout = new QGridLayout;
    ptopLayout->addWidget(plblFirstName, 0, 0);
    ptopLayout->addWidget(plblLastName, 1, 0);
    ptopLayout->addWidget(m_ptxtFirstName, 0, 1);
    ptopLayout->addWidget(m_ptxtLastName, 1, 1);
    ptopLayout->addWidget(pcmdOk, 2,0);
    ptopLayout->addWidget(pcmdCancel, 2, 1);
    setLayout(ptopLayout);
}
```

Метод `firstName()` возвращает введенное пользователем имя (листинг 32.5).

Листинг 32.5. Метод `firstName()` (файл InputDialog.cpp)

```
QString InputDialog::firstName() const
{
    return m_ptxtFirstName->text();
}
```

Метод `firstName()` возвращает введенную пользователем фамилию (листинг 32.6).

Листинг 32.6. Метод `lastName()` (файл InputDialog.cpp)

```
QString InputDialog::lastName() const
{
    return m_ptxtLastName->text();
}
```

Стандартные диалоговые окна

Использование стандартных окон значительно ускоряет разработку тех приложений, в которых необходимо использовать диалоговые окна выбора файлов, шрифта, цвета и т. д. Вместо того чтобы тратить время на разработку своих собственных классов, можно воспользоваться готовыми классами библиотеки Qt. К достоинствам стандартных диалоговых окон можно отнести и целостность пользовательского интерфейса, так как вид окон во всех приложениях, их использующих, будет один и тот же.

Диалоговое окно выбора файлов

Диалоговое окно выбора файлов предназначено для выбора одного или нескольких файлов, а также файлов, находящихся на удаленном компьютере, и поддерживает возможность переименования файлов и создания каталогов. Класс `QFileDialog` предоставляет реализацию диалогового окна выбора файлов (рис. 32.2, 32.3) и отвечает за создание и работоспособность сразу трех диалоговых окон. Одно из них позволяет осуществлять выбор файла для открытия, второе предназначено для выбора пути и имени файла для его сохранения, а третье — для выбора каталога. Класс `QFileDialog` унаследован от класса `QDialog`. Его определение находится в файле QFileDialog.



Рис. 32.2. Диалоговое окно выбора файлов в Mac OS X

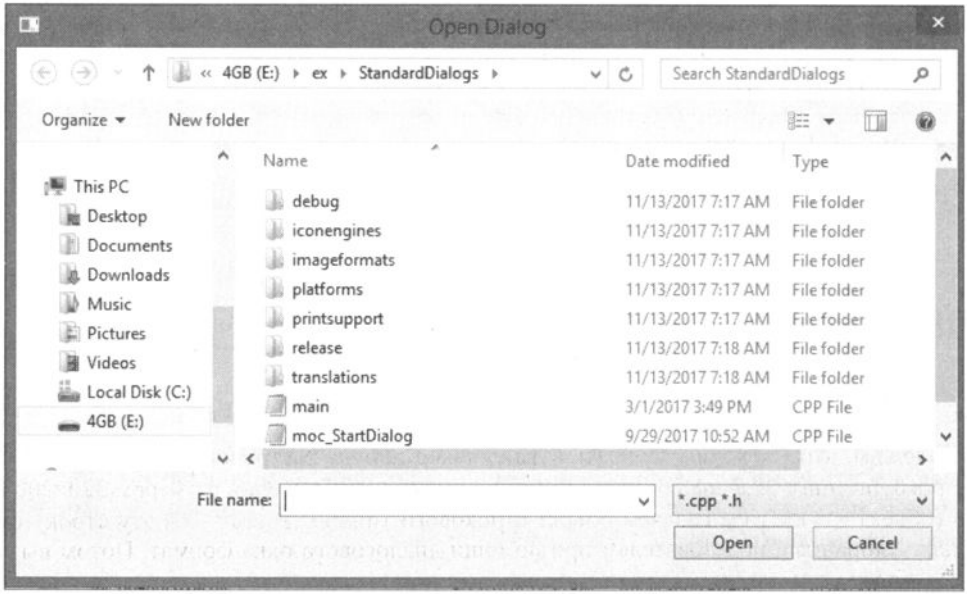


Рис. 32.3. Диалоговое окно выбора файлов в Windows

Класс `QFileDialog` предоставляет следующие статические методы:

- `getOpenFileName()` — создает диалоговое окно выбора одного файла. Этот метод возвращает значение типа `QString`, содержащее имя и путь выбранного файла (см. рис. 32.2);
- `getOpenFileNames()` — создает диалоговое окно выбора нескольких файлов. Возвращает список строк типа `QStringList`, содержащих пути и имена файлов;
- `getSaveFileName()` — создает диалоговое окно сохранения файла. Возвращает имя и путь файла в строковой переменной типа `QString`;
- `getExistingDirectory()` — создает окно выбора каталога. Возвращает значение типа `QString`, содержащее имя и путь выбранного каталога.

Первым параметром этих методов является указатель на объект-предок, вторым передается текст заголовка окна, третьим — строка, представляющая собой рабочий каталог.

Вызов метода `getOpenFileName()` запустит диалоговое окно открытия файла (см. рис. 32.2). Четвертый параметр, передаваемый в этот метод, представляет собой фильтр (или *маску*), задающий расширение файлов. Например:

```
QString str = QFileDialog::getOpenFileName(0, "Open Dialog", "", "*.cpp *.h");
```

Листинг 32.7 показывает, как можно использовать статический метод `getSaveFileName()`, предназначенный для диалогового окна записи файла.

Листинг 32.7. Использование диалогового окна для записи файла

```
QPixmap pix(320, 200);
QString strFilter;
QString str =
        QFileDialog::getSaveFileName(0,
                                     tr("Save Pixmap"),
                                     "Pixmap",
```

```
                                        "*.png ;; *.jpg ;; *.bmp",
                                        &strFilter
                                       );
if (!str.isEmpty()) {
    if (strFilter.contains("jpg")) {
        pix.save(str, "JPG");
    }
    else if (strFilter.contains("bmp")) {
        pix.save(str, "BMP");
    }
    else {
        pix.save(str, "PNG");
    }
}
```

Предположим, что мы хотим записать в файл какое-нибудь растровое изображение. В первой строке листинга 32.7 мы создаем объект растрового изображения размером 320×200 пикселов (объект `pix`). Затем создаем объект строкового типа `strFormat` — в эту строку будет помещен выбранный пользователем при помощи диалогового окна формат. Потом вызываем диалоговое окно при помощи статического метода `getSaveFileName()`. В этот метод мы передаем: нулевой указатель на объект предка, надпись самого окна `"Save Pixmap"`, имя для файла `"Pixmap"`, строку с тремя графическими форматами, разделенными между собой двумя символами точки с запятой: `;;` — чтобы каждый из них был представлен отдельным элементом. Последним передается адрес нашей строки, то есть место, куда будет помещен выбранный пользователем формат (объект `strFilter`). После закрытия диалогового окна мы проверяем строку `str` на содержимое, и если оно есть, то далее при помощи метода `QString::contains()` мы проверяем строку `strFilter` на содержание одного из обозначений графического формата. Растровое изображение записывается вызовом метода `QPixmap::save()`. Если совпадений для JPG или BMP не найдено, то растровое изображение будет записано в формате PNG.

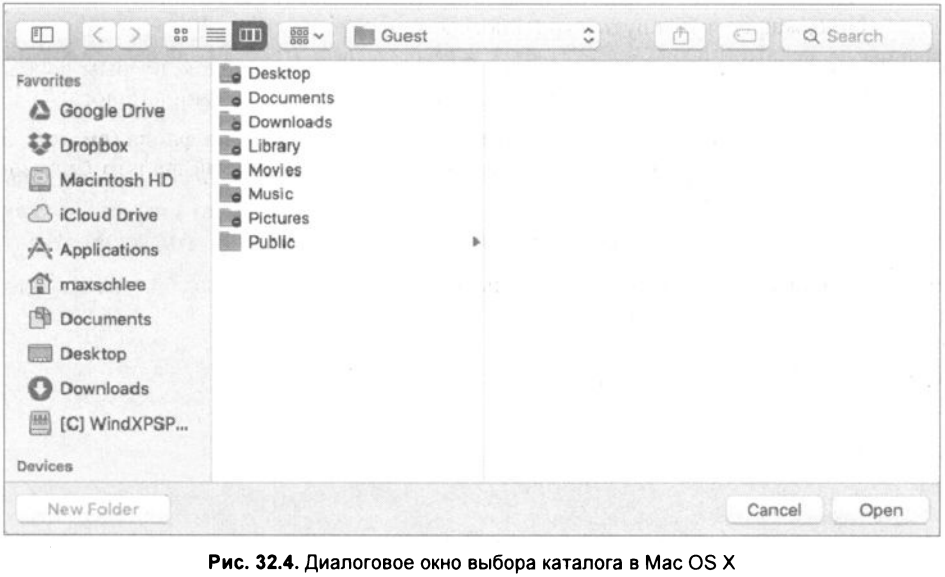


Рис. 32.4. Диалоговое окно выбора каталога в Mac OS X

При помощи метода `getExistingDirectory()` можно предоставить пользователю возможность выбора каталога (рис. 32.4, 32.5). Например:

```
QString str = QFileDialog::getExistingDirectory(0, "Directory Dialog", "");
```

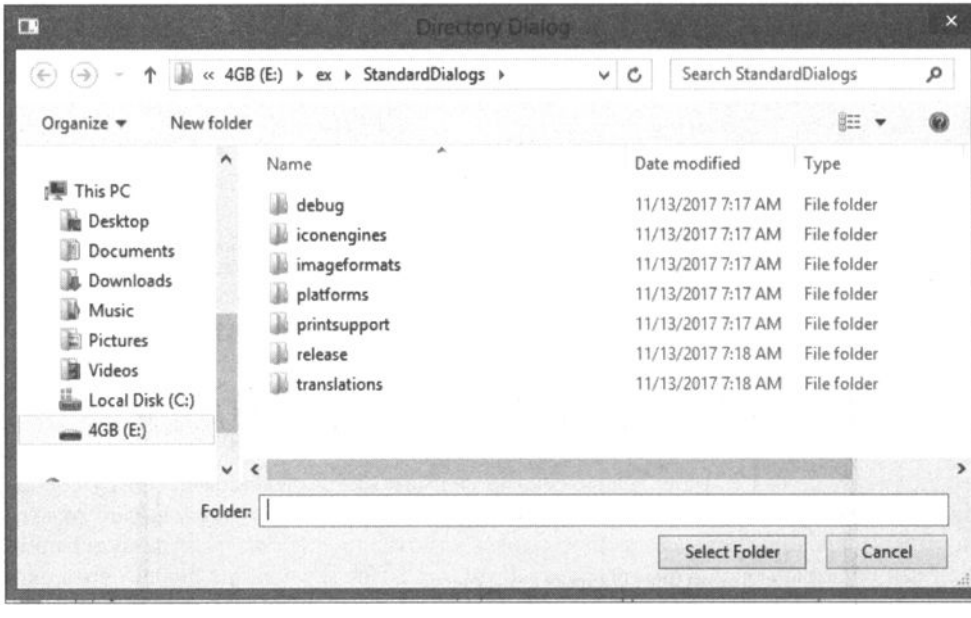


Рис. 32.5. Диалоговое окно выбора каталога в Windows

Диалоговое окно настройки принтера

Это окно позволяет выбрать принтер, изменить его параметры и задать диапазон страниц для вывода на печать (рис. 32.6, 32.7). Для использования этого диалогового окна необходимо включить в pro-файл модуль `QtPrintSupport`. Это делается следующей строкой:

```
QT += printsupport
```

Диалоговое окно настройки принтера реализовано в классе `QPrintDialog`, но вызывать его в отрыве от объекта принтера класса `QPrinter` (см. *главу 24*) не имеет смысла, так как главная наша задача состоит в настройке этого объекта для вывода на печать. Например:

```
QPrinter printer;
QPrintDialog* pPrintDialog = new QPrintDialog(&printer);
if (pPrintDialog->exec() == QDialog::Accepted) {
    // Печать
}
delete pPrintDialog;
```

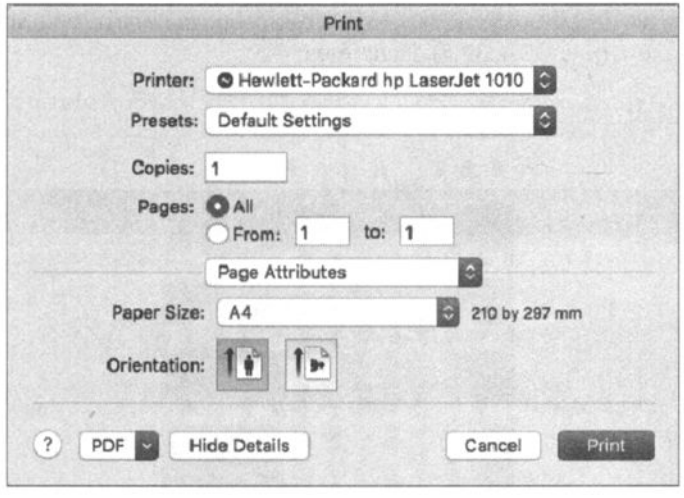


Рис. 32.6. Диалоговое окно настройки принтера в Mac OS X

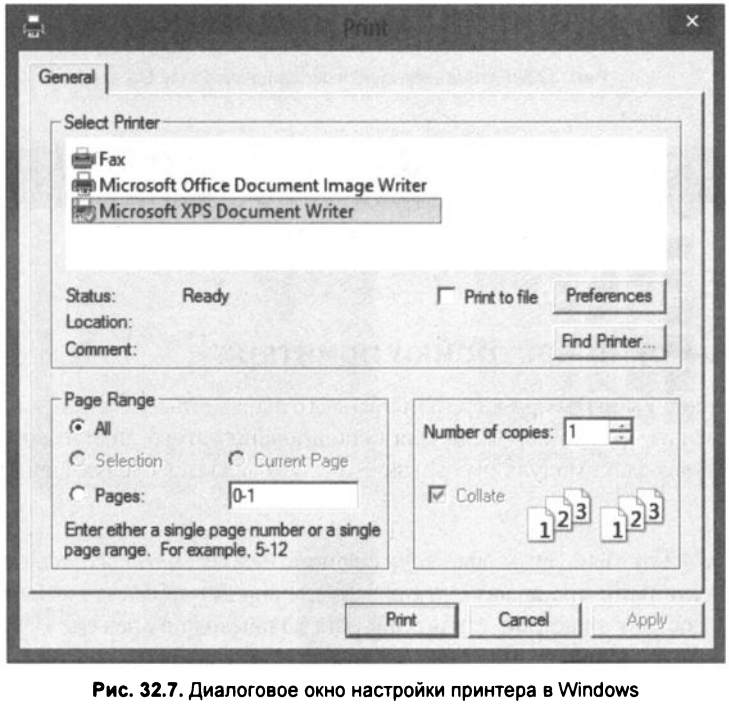


Рис. 32.7. Диалоговое окно настройки принтера в Windows

Диалоговое окно выбора цвета

Класс `QColorDialog` реализует диалоговое окно выбора цвета (рис. 32.8, 32.9). Для того чтобы показать это окно, вызывается статический метод `getColor()`. Первым параметром в метод можно передать цветовое значение для инициализации. Вторым параметром является указатель на виджет предка. После закрытия диалогового окна метод `getColor()` возвраща-

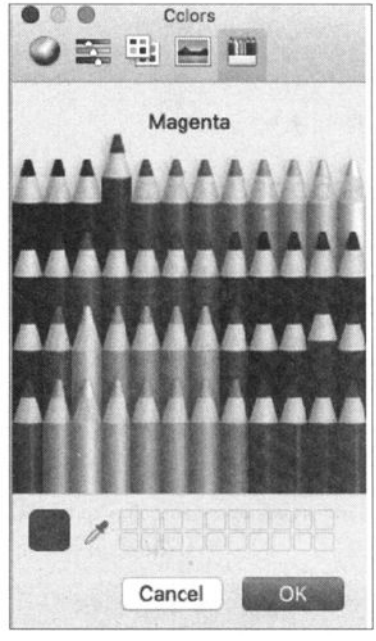


Рис. 32.8. Диалоговое окно выбора цвета в Mac OS X

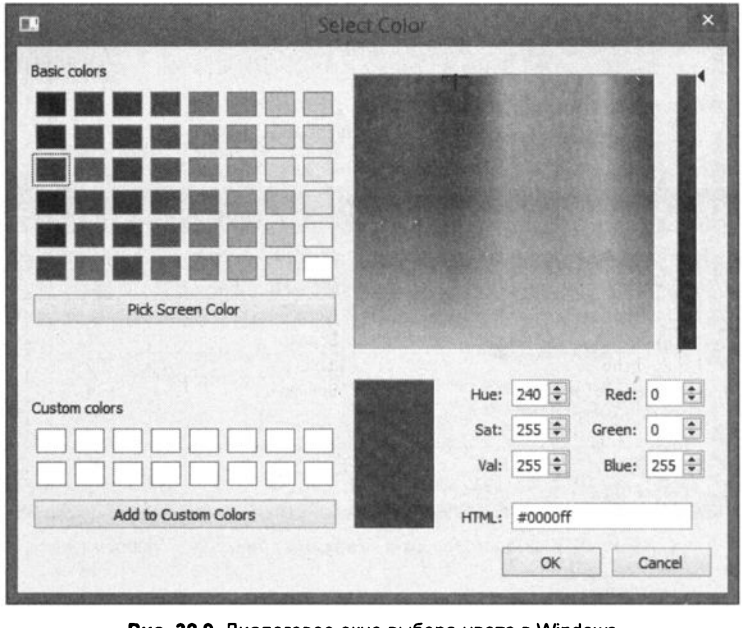


Рис. 32.9. Диалоговое окно выбора цвета в Windows

ет объект класса `QColor`. Чтобы узнать, какой кнопкой было закрыто окно: **OK** или **Cancel** (Отмена), необходимо вызвать метод `isValid()` из возвращенного объекта `QColor`. Значение `true` означает, что была нажата кнопка **OK**, в противном случае — **Cancel** (Отмена). Например:

```
QColor color = QColorDialog::getColor(blue);
if (!color.isValid()) {
    // Cancel
}
```

Диалоговое окно выбора шрифта

Окно выбора шрифта предназначено для выбора одного из зарегистрированных в системе шрифтов, а также для задания его стиля и размера (рис. 32.10, 32.11). Реализация этого диалогового окна содержится в классе `QFontDialog`, определенном в заголовочном файле QFontDialog.



Рис. 32.10. Диалоговое окно выбора шрифта в Mac OS X

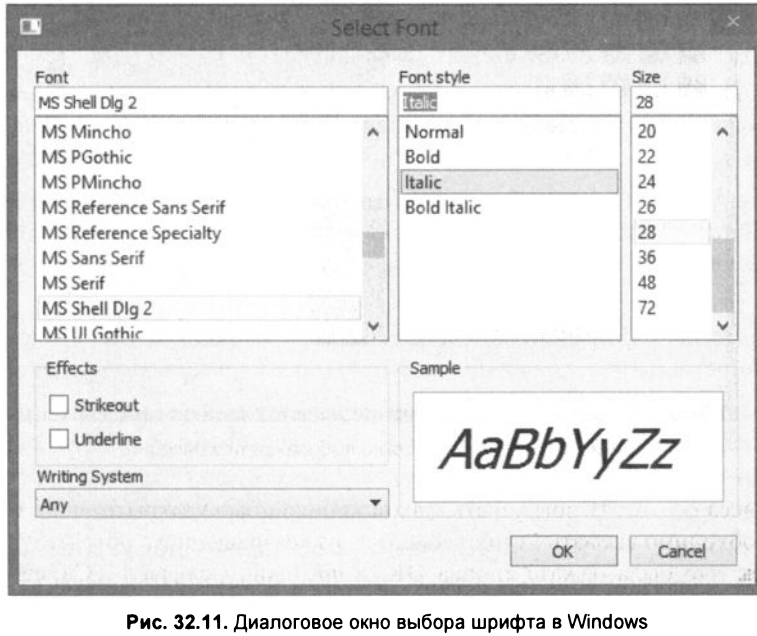


Рис. 32.11. Диалоговое окно выбора шрифта в Windows

Для того чтобы показать диалоговое окно, в большинстве случаев можно обойтись методом `QFontDialog::getFont()`. Первый параметр этого метода является указателем на переменную булевого типа. Метод записывает в эту переменную значение `true` в том случае, если диалоговое окно было закрыто нажатием на кнопку **OK**, в противном случае — значение `false`. Во втором параметре можно передать объект класса `QFont`, который будет использоваться для инициализации диалогового окна. После завершения выбора шрифта и закрытия окна статический метод `getFont()` возвращает шрифт, выбранный пользователем. Например:

```
bool bOk;
QFont fnt = QFontDialog::getFont(&bOk);
if (!bOk) {
    // Была нажата кнопка Cancel
}
```

Диалоговое окно ввода

Диалоговое окно ввода данных можно использовать для предоставления пользователю возможности ввода строки или числа. Это окно реализовано в классе `QInputDialog`. Конечно, можно и самому написать нечто подобное, разместив в диалоговом окне виджет класса `QLineEdit`, но зачем это делать, когда есть готовый класс? А вот для более сложных диалоговых окон, имеющих более одного поля ввода, этот класс уже не подойдет, и придется реализовывать свой собственный.

Для отображения диалогового окна ввода класс `QInputDialog` предоставляет четыре статических метода:

- `getText()` — для текста или пароля;
- `getInt ()` — для ввода целых чисел;
- `getDouble()` — для ввода чисел с плавающей точкой двойной точности;
- `getItem()` — для выбора элемента из списка строк.

В эти методы первым параметром передается указатель на виджет предка, вторым — заголовок диалогового окна, третьим — поясняющий текст, который будет отображен рядом с полем ввода. Начиная с четвертого, параметры этих методов отличаются друг от друга:

- в методах `getInt()` и `getDouble()` четвертым параметром передается значение для инициализации (по умолчанию равное нулю), а пятым и шестым — минимально возможное (по умолчанию −2 147 483 647) и максимально возможное (по умолчанию 2 147 483 647) значения;
- в методе `getText()` четвертый параметр управляет режимом ввода паролей, а пятым параметром передается текст для инициализации (по умолчанию это пустая строка);
- в методе `getItem()` четвертым параметром передается список строк (`QStringList`), пятый параметр делает одну из переданных строк текущей (по умолчанию — первую строку), шестой параметр управляет режимом возможности редактирования строк (по умолчанию включен).

Два последних параметра у всех этих методов аналогичные — это указатель на переменную булевого типа, информирующую о кнопке, нажатой при закрытии окна: **OK** или **Cancel** (Отмена), и флаги окна (см. *главу 5*).

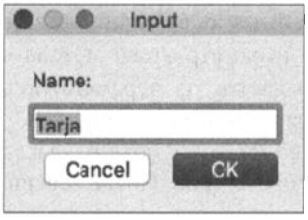


Рис. 32.12. Диалоговое окно ввода текста

Например, отобразить диалоговое окно ввода текста (рис. 32.12) можно следующим образом:

```
bool bOk;
QString str = QInputDialog::getText(0,
                                    "Input",
                                    "Name:",
                                    QLineEdit::Normal,
                                    "Tarja",
                                    &bOk
                                   );
if (!bOk) {
    // Была нажата кнопка Cancel
}
```

Изменив четвертый параметр с `QLineEdit::Normal` на `QLineEdit::Password`, мы переведем виджет однострочного текстового поля в режим ввода пароля.

Диалоговое окно процесса

Для отображения в диалоговом окне процесса выполнения какой-либо операции Qt предоставляет класс `QProgressDialog`, унаследованный от класса `QDialog`. Это окно информирует пользователя о начале продолжительной операции и дает ему возможность визуально оценить время работы. Окно может содержать кнопку **Cancel** (Отмена) для прерывания начатой операции. При нажатии на нее отправляется сигнал `canceled()`, который следует соединить со слотом, ответственным за прекращение проводимой операции.

Диалоговое окно процесса открывается в том случае, если длительность всей операции будет составлять более трех секунд, с гарантией, что оно не станет появляться на короткий промежуток времени и вводить пользователя в заблуждение. Время, впрочем, можно изменить, передав в метод `setMinimumDuration()` целочисленное значение в миллисекундах. Задать количество шагов от начала до конца операции можно в третьем параметре конструктора при его создании или же с помощью метода `setTotalSteps()`. В процессе выполнения операции должен вызываться метод `setProgress()`.

Диалоговое окно процесса, показанное на рис. 32.13, можно создать следующим образом:

```
int              n    = 100000;
QProgressDialog* pprd =
    new QProgressDialog("Processing the data...", "&Cancel", 0, n);

pprd->setMinimumDuration(0);
pprd->setWindowTitle("Please Wait");
```

```
for (int i = 0; i < n; ++i) {
    pprd->setValue(i);
    qApp->processEvents();
    if (pprd->wasCanceled()) {
        break;
    }
}
pprd->setValue(n);
delete pprd;
```

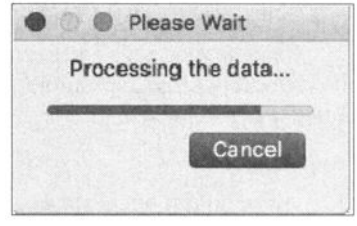


Рис. 32.13. Диалоговое окно процесса выполнения операции

Первым параметром в конструктор класса `QProgressDialog` передается текст поясняющей надписи проводимой операции. Второй параметр представляет собой надпись на кнопке **Cancel** (Отмена). Третий и четвертый параметры задают минимальное и максимальное значения для индикатора выполнения, определяя тем самым количество шагов для проводимой операции. Пятый, опциональный, параметр является указателем на виджет предка. Шестой параметр — также опциональный, он задает флаги окна (см. *главу 5*). В метод `setMinimumDuration()` передается значение 0, говорящее о том, что диалоговое окно будет показано без задержек. Вызов метода `setWindowTitle()` устанавливает текст в заголовке диалогового окна. В цикле, осуществляющем длительную обработку (в нашем случае в цикле `for`), вызывается метод `setValue()`, обновляющий состояние индикатора выполнения. Вызов метода `processEvents()` из объекта приложения заставляет перед проведением каждой итерации обрабатывать события, что позволяет разблокировать графический интерфейс программы в момент проведения интенсивных операций (см. *главу 14*). В операторе `if` вызовом метода `wasCanceled()` проверяется, не была ли нажата кнопка **Cancel** (Отмена), и если она была нажата, то выполнение цикла прерывается.

По окончании операции можно вызвать слот `reset()`, чтобы установить индикатор выполнения в начало. Можно сделать так, чтобы операция установки в начало проводилась автоматически, для чего в метод `setAutoReset()` передается значение `true`. Для автоматического закрытия окна по окончании операции вызывается метод `setAutoClose()`, в который передается значение `true`.

Диалоговые окна мастера

Диалоговые окна мастера были придуманы для сопровождения пользователя при выполнении им операций, которые требуют непосредственного его участия. Для навигации по страницам диалогового окна мастера служат две кнопки: **Next** (Вперед) и **Back** (Назад). Пользователь не имеет возможности сразу отобразить интересующую его страницу окна, не пройдя все предшествующие страницы, что гарантирует выполнение всех пунктов, содержащихся в этих диалоговых окнах.

Для создания класса мастера нужно унаследовать класс `QWizard` и добавить каждую новую страницу диалогового окна вызовом метода `addPage()`. В этот метод надо передать указа-

тель на виджет QWizardPage, в котором вызовом метода setTitle() можно установить строку заголовка, а при помощи класса компоновки — разместить все необходимые виджеты. О кнопках для продвижения вперед и назад беспокоиться не придется, поскольку они уже сконфигурированы для смены страниц. В листинге 32.8 реализован класс мастера с тремя страницами, содержащими виджеты надписи (рис. 32.14).

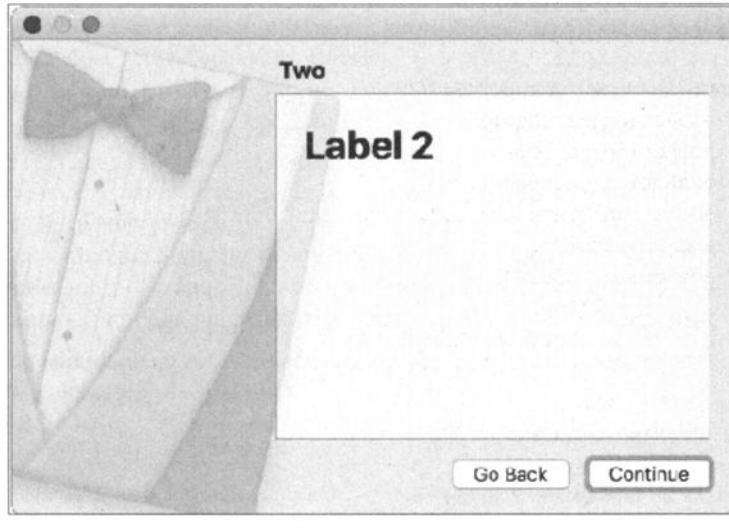


Рис. 32.14. Диалоговое окно мастера

Листинг 32.8. Диалоговое окно мастера

```
class Wizard : public QWizard {
private:
    QWizardPage* createPage(QWidget* pwgt, QString strTitle)
    {
        QWizardPage* ppage = new QWizardPage;
        ppage->setTitle(strTitle);

        QVBoxLayout* playout = new QVBoxLayout;
        playout->addWidget(pwgt);
        ppage->setLayout(playout);

        return ppage;
    }

public:
    Wizard::Wizard(QWidget* pwgt = 0) : QWizard(pwgt)
    {
        addPage(createPage(new QLabel("<H1>Label 1</H1>"), "One"));
        addPage(createPage(new QLabel("<H1>Label 2</H1>"), "Two"));
        addPage(createPage(new QLabel("<H1>Label 3</H1>"), "Three"));
    }
};
```

Диалоговые окна сообщений

Собственные диалоговые окна для вывода на экран сообщений создавать не имеет смысла — ведь для этого можно воспользоваться уже готовыми окнами, предоставляемыми классом QMessageBox. *Диалоговое окно сообщения* — это самое простое диалоговое окно, которое отображает текстовое сообщение и ожидает реакции со стороны пользователя. Основное назначение такого окна состоит в информировании пользователя об определенном событии. Все окна, предоставляемые классом QMessageBox, — модальные. Они могут содержать кнопки, заголовок и текст сообщения.

Класс QMessageBox предоставляет целую серию статических методов, с помощью которых можно создавать окна сообщений. Эти методы предоставляют поддержку сообщений трех уровней важности: информационного, предупреждающего и критического, которые выбираются в зависимости от обстоятельств. Окна могут содержать до трех кнопок. Все это очень удобно, поскольку отпадает необходимость в написании дополнительного кода для реализации вывода сообщения. Можно применять такие окна и для отладочных целей — вывести необходимую информацию и приостановить выполнение программы.

Окно сообщения, показанное на рис. 32.15, можно реализовать следующим образом:

```
QMessageBox* pmbx =
    new QMessageBox(QMessageBox::Information,
                    "MessageBox",
                    "<b>A</b> <i>Simple</i> <u>Message</u>",
                    QMessageBox::Yes | QMessageBox::No |
 QMessageBox::Cancel
                   );
int n = pmbx->exec();
delete pmbx;

if (n == QMessageBox::Yes) {
    //Нажата кнопка Yes
}
```

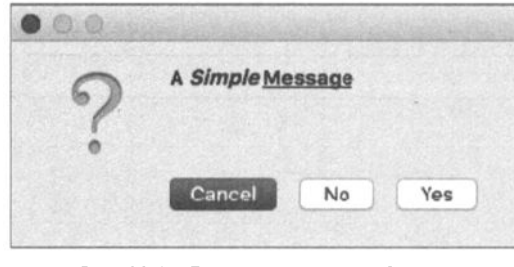


Рис. 32.15. Диалоговое окно сообщения

Это окно создается динамически. Первый параметр задает предопределенное растровое изображение, которое будет отображено слева (табл. 32.1). Во втором параметре конструктора передается текст заголовка окна, в третьем — текст сообщения, в котором можно использовать теги HTML. Четвертый параметр задает кнопки, которые будут размещены в диалоговом окне (табл. 32.2), — этот параметр необязателен, и если его не указать, то

диалоговое окно будет содержать только лишь одну кнопку **OK**. В нашем примере мы определили три кнопки **Yes**, **No** и **Cancel**. Следует обратить ваше внимание на то, что в подобных случаях всегда нужно обязательно определять кнопку **Cancel** (Отмена). Надо также учесть, что для отмены действия пользователями часто нажимается клавиша <Escape>, и диалоговое окно в этом случае должно возвратить результат нажатия кнопки **Cancel**. Четвертый и пятый параметры не обязательны, но в них можно передать указатель на виджет предка и флаги, управляющие внешним видом диалогового окна. Вызов метода `exec()` объекта класса `QMessageBox` приводит к остановке приема событий основной программы и ожиданию нажатия на одну из кнопок окна сообщения. После нажатия метод `exec()` возвращает идентификатор кнопки, который сохраняется в переменной `n`. Оператор `delete` удаляет из памяти виджет окна сообщения. В операторе `if` проверяется значение нажатой кнопки на равенство значению кнопки **Yes** (Да).

Параметры, передаваемые в конструктор, можно установить и с помощью методов, определенных в классе `QMessageBox`. Текст сообщения устанавливается методом `setText()`, а кнопки добавляются методом `addButton()`. Первым параметром в метод `addButton()` необходимо передать либо текст кнопки, либо указатель на объект самой кнопки. Вторым параметром в метод `addButton()` необходимо передать один из целочисленных идентификаторов роли, приведенных в табл. 32.2. Существует и более простой вариант метода `addButton()`, в который можно передать всего лишь одно значение перечисления `StandardButton` из указанных в табл. 32.3.

> ***Кнопка Escape (Отмена)***
>
> Если необходима кнопка **Escape** (Отмена), то ее можно установить отдельным вызовом метода `setEscapeButton()`, в который необходимо передать значение перечисления `StandardButton` из табл. 32.3.

При помощи метода `setWindowTitle()` устанавливается текст заголовка окна. Метод `setIcon()` устанавливает растровое изображение, для этого нужно передать в него одну из констант, указанных в табл. 32.1. Если приведенных в этой таблице растровых изображений недостаточно, то можно создать и установить свое собственное, передав объект класса `QPixmap` в метод `setIconPixmap()`.

***Таблица 32.1.** Растровые изображения*

Константа	Значение	Вид
`NoIcon`	0	—
`Information`	1	
`Warning`	2	
`Critical`	3	
`Question`	4	

Таблица 32.2. *Перечисление констант ButtonRole класса* `QMessageBox`

Константа	Константа	Константа
NoRole	YesRole	HelpRole
AcceptRole	DestructiveRole	ApplyRole
RejectRole	ActionRole	ResetRole

Таблица 32.3. *Некоторые перечисления констант StandardButton класса* `QMessageBox`

Константа	Значение	Константа	Значение	Константа	Значение
NoButton	0x00000000	No	0x00010000	YesToAll	0x00008000
Ok	0x00000400	Abort	0x00040000	NoToAll	0x00020000
Cancel	0x00400000	Retry	0x00080000	SaveAll	0x00001000
Yes	0x00004000	Ignore	0x00100000	Help	0x01000000

Окно информационного сообщения

Это окно служит для отображения сообщений после успешного исполнения операции. Вызов статического метода `information()` отображает на экране окно информационного сообщения, показанное на рис. 32.16:

```
QMessageBox::information(0, "Information", "Operation Complete");
```

Как только окно будет закрыто, метод вернет значение нажатой кнопки (см. табл. 32.2).

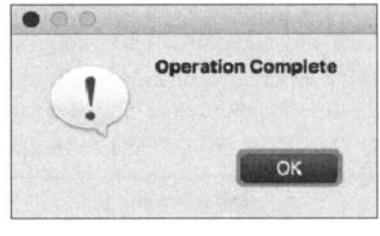


Рис. 32.16. Окно информационного сообщения

Окно предупреждающего сообщения

Для отображения предупреждающего сообщения (рис. 32.17) вызывается статический метод `warning()` класса `QMessageBox`.

Вывод окна предупреждающего сообщения может выглядеть так:

```
int n = QMessageBox::warning(0,
                             "Warning",
                             "The text in the file has changed"
                             "\n Do you want to save the changes?",
```

```
                                QMessageBox::Yes | QMessageBox::No,
                                QMessageBox::Yes
                               );
if (n == QMessageBox::Yes) {
    // Saving the changes!
}
```

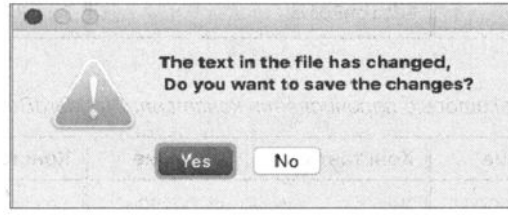


Рис. 32.17. Окно предупреждающего сообщения

В метод `warning()` первым параметром передается указатель на предка (в нашем случае нулевой), вторым — строка заголовка окна, третьим — текст сообщения. Четвертый параметр задает две кнопки (в нашем примере **Yes** и **No**). Пятый указывает на то, какая из кнопок будет кнопкой по умолчанию (в нашем случае это кнопка **Yes**). Если пользователь нажмет клавишу <Escape>, то метод `warning()` возвратит значение `QMessageBox::Escape`. В нашем примере мы проверяем нажатие на кнопку **Yes** (Да).

Окно критического сообщения

Это диалоговое окно следует показывать только в тех случаях, когда произошло что-то очень серьезное (рис. 32.18). Для его отображения нужно вызвать статический метод `critical()`, передав ему в первом параметре указатель на виджет предка, во втором — заголовок, а в третьем — само сообщение. В четвертом задаются кнопки:

```
int n = QMessageBox::critical(0,
                              "Attention",
                              "This operation will make your "
                              "computer unusable, continue?",
                              QMessageBox::Yes |  QMessageBox::No |
                              QMessageBox::Cancel
                             );
if (n == QMessageBox::Yes) {
    // Do it!
}
```

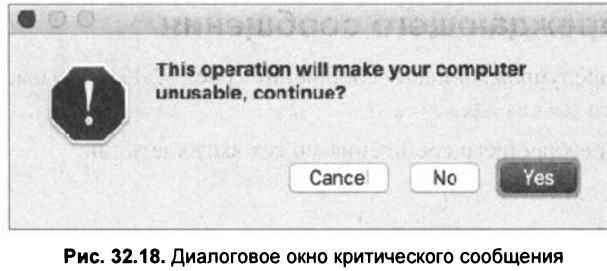


Рис. 32.18. Диалоговое окно критического сообщения

Окно сообщения о программе

Пожалуй, это самое простое диалоговое окно, которое отображается при вызове статического метода `about()` класса `QMessageBox`. Например, его можно использовать для предоставления пользователю общей информации о программе: версии, контактной информации, информации об авторских правах и т. д. (рис. 32.19). В этот метод передаются три параметра. Первый параметр — это указатель на виджет предка, второй — на заголовок окна, третий — представляет собой само сообщение:

```
QMessageBox::about(0, "About", "My Program Ver. 1.0");
```

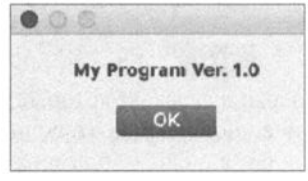


Рис. 32.19. Информация о программе

Окно сообщения *About Qt*

Для отображения диалогового окна **About Qt** (О Qt) класс `QMessageBox` предоставляет статический метод `aboutQt()`. В этот метод передаются два параметра. Первый служит для установки предка, а второй — заголовка окна. За отображаемое сообщение отвечает сама

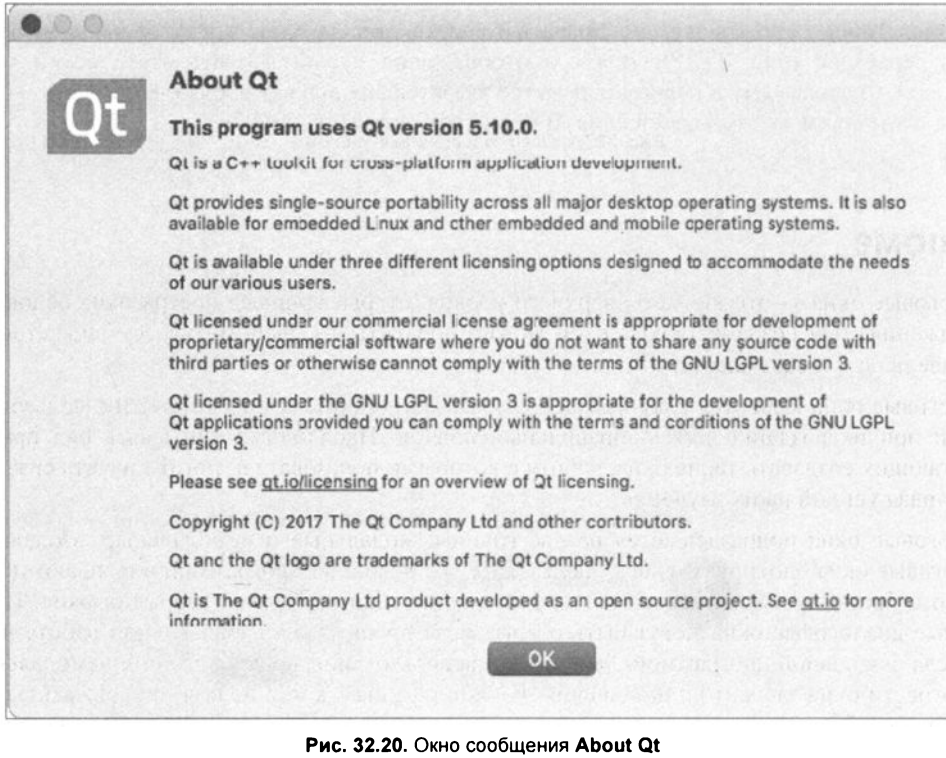


Рис. 32.20. Окно сообщения **About Qt**

библиотека Qt. Как видно из рис. 32.20, в этом окне содержится информация о библиотеке. Вызов окна выполняется следующим образом:

```
QMessageBox::aboutQt(0);
```

Окно сообщения об ошибке

Диалоговое окно сообщения об ошибке реализуется классом `QErrorMessage`, а не классом `QMessageBox`, как все остальные окна сообщений. Оно представляет собой немодальное диалоговое окно. Для отображения окна сообщения об ошибке создается объект этого класса и вызывается метод `showMessage()`, в который передается текст сообщения. Например:

```
(new QErrorMessage(this))->showMessage("Write Error");
```

Как видно из рис. 32.21, окно содержит флажок, который может быть снят пользователем, чтобы не показывать это сообщение снова при повторении указанной ошибки. Не следует злоупотреблять окнами сообщения об ошибке — применять их следует только в тех случаях, когда это не повредит пользователю. Этим окном имеет смысл сообщать о критических ошибках, так как после перезапуска программы флажок будет сброшен.

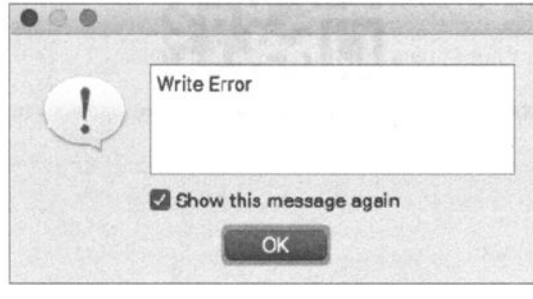


Рис. 32.21. Окно сообщения об ошибке

Резюме

Диалоговые окна — это виджеты верхнего уровня, открывающиеся поверх окна основного приложения. Для создания таких окон можно обратиться к любому классу виджетов, но удобнее использовать класс `QDialog`.

Диалоговые окна являются ключевыми элементами для обмена информацией с пользователем, и при их создании нужен правильный подход. Необходимо учитывать ряд правил, помогающих создавать такие окна, работу с которыми пользователь мог бы начать сразу, не затрачивая усилий на их изучение.

Диалоговые окна подразделяются на две группы: модальные и немодальные. Модальные диалоговые окна блокируют работу пользователя с основной программой и ожидают действий со стороны пользователя. Разблокировка происходит в момент закрытия окна. Немодальные диалоговые окна могут быть открыты, не препятствуя параллельной работе пользователя с основной программой. Решение об использовании модального или немодального диалогового окна зависит от назначения. В тех ситуациях, когда нельзя продолжать работу приложения без решения пользователя, нужно использовать модальные диалоговые окна. Стандартное диалоговое окно выбора файлов является типичным примером использования

модального диалогового окна. А, скажем, диалоговое окно для поиска лучше делать немодальным, чтобы обеспечить пользователю возможность выделения текста, не закрывая само окно поиска.

Применение стандартных диалоговых окон позволяет сэкономить время на разработку, так как при этом можно воспользоваться уже готовыми окнами для открытия файлов, выбора цвета, настройки принтера и т. п.

Наиболее распространенные в приложениях диалоговые окна — это окна сообщений, которые служат для оповещения пользователя о важном событии или для того, чтобы задать ему вопрос, требующий ответа «Да» или «Нет», — например, «Вы действительно хотите удалить этот файл?»

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/32-510/** или с помощью следующего QR-кода (рис. 32.22):



Рис. 32.22. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 33

Предоставление помощи

Сова приложила ухо к груди Буратино.
— Пациент скорее мертв, чем жив, — прошептала она.
Жаба прошлепала большим ртом:
— Пациент скорее жив, чем мертв...
— Одно из двух, — прошелестел Народный лекарь Богомол, — или пациент жив, или он умер. Если он жив — он останется жив, или он не останется жив. Если он мертв — его можно оживить или нельзя оживить.

Алексей Толстой, «Приключения Буратино»

Главная задача помощи состоит в обеспечении пользователя всей необходимой информацией о приложении и его элементах, для того чтобы сделать его работу более удобной. Различают три типа помощи:

- всплывающая подсказка;
- подсказка «Что это»;
- система помощи (Online Help).

Всплывающая подсказка

Работая с различными программами, вы, наверное, заметили, что при задержке указателя мыши над кнопками панелей инструментов рядом с ним автоматически появляется небольшое текстовое окошко, поясняющее назначение кнопки (рис. 33.1). Такое окно называется *всплывающей подсказкой* (tooltip) и, как правило, содержит только одну строку текста. Можно использовать подсказки, имеющие и более одной строки, но все же их следует делать как можно короче.

Присутствие всплывающей подсказки в приложении не является обязательным, но лучше все-таки ее предоставлять, поскольку она помогает пользователям быстрее сориентироваться среди множества кнопок. Несомненный плюс этого типа подсказки в том, что пользователь, не останавливая своей работы, получает информацию об элементах приложения.

Чаще всего такие подсказки «всплывают» у кнопок панелей инструментов, но их можно с успехом использовать и для любых виджетов. При этом не следует забывать, что применение всплывающих подсказок теряет смысл в тех случаях, когда объяснение излишне. Например, вряд ли логично повторить для кнопки **Cancel** (Отмена) ее надпись во всплывающей подсказке.

Чтобы установить подсказку в виджете, нужно вызвать метод `setToolTip()`:

```
QPushButton* pcmd = new QPushButton("&Ok");
pcmd->setToolTip("Button");
```

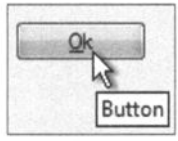


Рис. 33.1. Всплывающая подсказка



Рис. 33.2. Замена окна всплывающей подсказки

Чтобы удалить всплывающую подсказку, просто передайте в метод `setToolTip()` пустую строку.

Если нужно вместо окна всплывающей подсказки показать свое собственное окно (рис. 33.2), то можно поступить следующим образом (листинг 33.1). В этой программе при задержке курсора мыши на виджете окна мы отображаем виджет надписи `QLabel` с текстом, установленным методом `setToolTip()`.

Для реализации класса виджета со своим собственным окном отображения всплывающей подсказки нам нужно перезаписать метод `event()`, чтобы можно было отловить событие типа `QEvent::ToolTip`. Вся необходимая информация передается в объекте события `QHelpEvent`, поэтому мы приводим указатель события к этому типу и устанавливаем позицию окна виджета надписи нашей всплывающей подсказки на позицию указателя мыши. Текстовое сообщение подсказки мы получаем вызовом метода `toolTip()`. Для того чтобы сделать окно подсказки видимым, вызываем метод `show()`. Окно подсказки через какой-то промежуток времени должно обязательно исчезать, поэтому мы соединяем его слот `hide()` с таймером, установленным на 3 сек. В конструкторе класса `MyWidget` создаем виджет надписи для отображения подсказок и устанавливаем у него тип окна без обрамления. В нашем примере это `Qt::ToolTip`. В основной программе (функция `main()`) создаем экземпляр нашего класса `MyWidget` и устанавливаем в нем текст всплывающей подсказки методом `setToolTip()`.

Листинг 33.1. Создание своего окна всплывающей подсказки (файл main.cpp)

```
class MyWidget : public QWidget {
private:
    QLabel* m_plblToolTip;

protected:
    virtual bool event(QEvent* pe)
    {
        if (pe->type() == QEvent::ToolTip) {
            QHelpEvent* peHelp = static_cast<QHelpEvent*>(pe);
            m_plblToolTip->move(peHelp->globalPos());
            m_plblToolTip->setText(toolTip());
            m_plblToolTip->show();
            QTimer::singleShot(3000, m_plblToolTip, SLOT(hide()));

            return true;
        }

        return QWidget::event(pe);
    }
```

```
public:
    MyWidget(QWidget* pwgt = 0) : QWidget(pwgt)
    {
        m_plblToolTip = new QLabel;
        m_plblToolTip->setWindowFlags(Qt::ToolTip);
    }
};

// ----------------------------------------------------------------------
int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    MyWidget mw;
    mw.setFixedSize(70, 70);
    mw.setToolTip("<H1>My Tool Tip</H1>");
    mw.show();

    return app.exec();
}
```

Система помощи (Online Help)

Большие приложения нуждаются в объемной системе помощи, подробно описывающей все функциональные возможности программы. Самый простой вариант — это предоставление пользователю специального навигатора, открывающегося при выборе пункта меню **Help** (Справка) или при нажатии на клавишу <F1>. Текст помощи может быть представлен в формате HTML, который, помимо текстовой, может содержать и графическую информацию, ссылки на другие документы, а также дает возможность форматирования шрифтов. Большой плюс такого решения состоит в том, что для создания файлов в формате HTML существует множество редакторов. При острой необходимости и наличии навыков можно написать или подправить такой файл и «от руки» в простом текстовом редакторе.

Класс `QTextBrowser` располагает всем нужным для реализации навигатора (рис. 33.3), способного показывать текст в формате HTML. Следующий пример (листинги 33.2 и 33.3) демонстрирует применение этого класса.

В основной программе, показанной в листинге 33.2, создается виджет навигатора помощи — `helpBrowser`. В первом параметре конструктора передается путь на корневой каталог ресурса, а во втором — имя файла.

Листинг 33.2. Создание навигатора системы помощи (файл main.cpp)

```
#include <QApplication>
#include "HelpBrowser.h"

// ----------------------------------------------------------------------
int main (int argc, char** argv)
{
    QApplication app(argc, argv);
    HelpBrowser  helpBrowser(":/", "index.htm");
```

```
    helpBrowser.resize(450, 350);
    helpBrowser.show();

    return app.exec();
}
```



Рис. 33.3. Система помощи

В конструкторе класса `HelpBrowser` (листинг 33.3) создаются виджеты кнопок для дополнительной навигации: указатели `pcmdBack`, `pcmdHome` и `pcmdForward`. После этого с помощью метода `connect()` сигналы `clicked()` кнопок подсоединяются к соответствующим слотам виджета класса `QTextBrowser`: `backward()`, `home()` и `forward()`. Это необходимо для перемещения по документам справочной системы. Последние два метода `connect()` соединяют сигналы `backwardAvailable(bool)` и `forwardAvailable(bool)` виджета класса `QTextBrowser` со слотами `setEnabled(bool)` соответствующих кнопок (указатели `pcmdBack` и `pcmdForward`). Это позволяет делать кнопки, в зависимости от ситуации, активными или неактивными. Метод `setSearchPath()` устанавливает список путей для поиска документов, в нашем случае путь только один. Метод `setSource()` считывает переданный документ. Далее все виджеты размещаются при помощи вертикальной (`pvbxLayout`) и горизонтальной (`phbxLayout`) компоновок.

Листинг 33.3. Определение класса `HelpBrowser` (файл HelpBrowser.h)

```
#include <QtWidgets>

// ============================================================================
class HelpBrowser : public QWidget {
    Q_OBJECT
```

```
public:
    HelpBrowser(const QString& strPath,
                const QString& strFileName,
                      QWidget* pwgt       = 0
               ) : QWidget(pwgt)
    {
        QPushButton*  pcmdBack    = new QPushButton("<<");
        QPushButton*  pcmdHome    = new QPushButton("Home");
        QPushButton*  pcmdForward = new QPushButton(">>");
        QTextBrowser* ptxtBrowser = new QTextBrowser;

        connect(pcmdBack, SIGNAL(clicked()),
                ptxtBrowser, SLOT(backward())
               );
        connect(pcmdHome, SIGNAL(clicked()),
                ptxtBrowser, SLOT(home())
               );
        connect(pcmdForward, SIGNAL(clicked()),
                ptxtBrowser, SLOT(forward())
               );
        connect(ptxtBrowser, SIGNAL(backwardAvailable(bool)),
                pcmdBack, SLOT(setEnabled(bool))
               );
        connect(ptxtBrowser, SIGNAL(forwardAvailable(bool)),
                pcmdForward, SLOT(setEnabled(bool))
               );

        ptxtBrowser->setSearchPaths(QStringList() << strPath);
        ptxtBrowser->setSource(QString(strFileName));

        //Layout setup
        QVBoxLayout* pvbxLayout = new QVBoxLayout;
        QHBoxLayout* phbxLayout = new QHBoxLayout;
        phbxLayout->addWidget(pcmdBack);
        phbxLayout->addWidget(pcmdHome);
        phbxLayout->addWidget(pcmdForward);
        pvbxLayout->addLayout(phbxLayout);
        pvbxLayout->addWidget(ptxtBrowser);
        setLayout(pvbxLayout);
    }
};
```

Резюме

Использование помощи — верный шаг для повышения интуитивности работы пользователя с приложением. Помощь необходима для снабжения пользователя интересующей его информацией и подразделяется на три категории: всплывающая подсказка, подсказка типа «Что это» и система помощи (Online Help).

Всплывающая подсказка представляет собой небольшую по объему, часто ограниченную одной строкой, информацию. Она появляется возле указателя мыши, если его удержать на виджете.

Подсказка типа «Что это» очень похожа на всплывающую подсказку, но она предоставляет большую по объему информацию и отображается только после запуска специального режима.

Система помощи снабжает пользователя исчерпывающей информацией о функциях приложения. Чтобы воспользоваться ею, совсем не обязательно запускать само приложение, так как эта информация создается в формате HTML, и ее можно прочитать при помощи любого доступного в системе браузера. Для реализации своего собственного навигатора можно воспользоваться классом `QTextBrowser`.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/33-510/** или с помощью следующего QR-кода (рис. 33.4):



Рис. 33.4. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 34

Главное окно, создание SDI- и MDI-приложений

Приступай к решению любой задачи, имея в виду решить ее наилучшим образом.

Одна из трех заповедей IBM

Многие приложения имеют сходный интерфейс — окно включает меню, рабочую область, строку состояния и т. д. Неудивительно, что библиотека Qt содержит классы, помогающие создавать такие приложения.

Класс главного окна *QMainWindow*

Класс `QMainWindow` — это очень важный класс, который реализует главное окно (рис. 34.1), содержащее в себе типовые виджеты, необходимые большинству приложений, — такие как меню (см. *главу 31*), секции для панелей инструментов, рабочую область, строки состояния и т. п. В этом классе внешний вид окна уже подготовлен, и его виджеты не нуждаются в дополнительном размещении, поскольку уже находятся в нужных местах.

Окно приложения, изображенное на рис. 34.1, имеет рамку, область заголовка для отображения имени и три кнопки, управляющие окном. Оно также содержит меню, которое располагается ниже области заголовка окна. Панель инструментов находится под меню. Под рабочей областью размещена строка состояния.

Указатель на виджет меню можно получить вызовом метода `QMainWindow::menuBar()` и установить в нем нужные всплывающие меню:

```
QMenu* pmnuFile = new QMenu("&File");
pmnuFile->addAction("&Save");
...
menuBar()->addMenu(pmnuFile);
```

Как правило, устанавливаются следующие всплывающие меню:

- **File** (Файл) — содержит основные операции для работы с файлами: **New** (Создать), **Open** (Открыть), **Save** (Сохранить), **Print** (Печать) и **Quit** (Выход);
- **Edit** (Правка) — включает в себя команды общего редактирования: **Cut** (Вырезать), **Copy** (Копировать), **Paste** (Вставить), **Undo** (Отменить), **Redo** (Повторить), **Find** (Найти), **Replace** (Заменить) и **Delete** (Очистить);
- **View** (Вид) — содержит команды, изменяющие представление данных в рабочей области. Например, команда **Zoom** (Масштаб) масштабирует отображение документа. В это

меню можно включать и те команды, которые управляют отображением элементов интерфейса приложения, например панелей инструментов и строки состояния;

- **Help** (Справка) — необходима для предоставления помощи пользователю при освоении приложения (см. *главу 33*). Кроме того, используя команды этого меню, можно отобразить информацию об авторских правах на приложение. Например, при выборе команды **About** (О программе) обычно появляется окно, отображающее имя приложения, его версию и информацию об авторских правах.

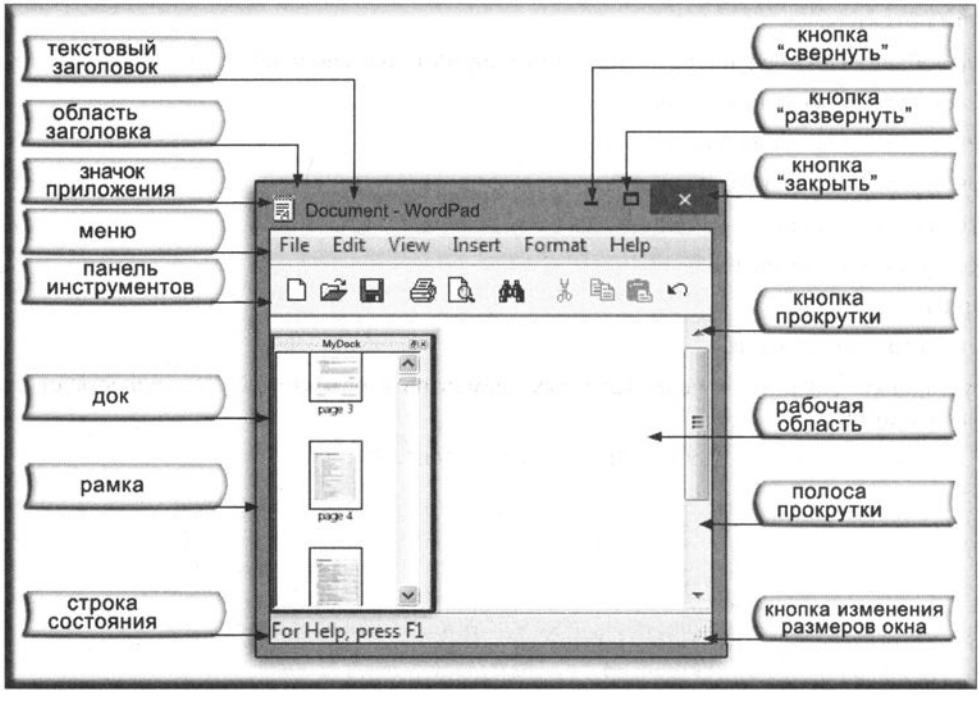


Рис. 34.1. Внешний вид главного окна приложения

Чтобы получить указатель на рабочую область, следует вызвать метод `QMainWindow::centralWidget()`, который вернет указатель на `QWidget`. Для установки виджета рабочей области потребуется вызвать метод `QMainWindows::setCentralWidget()` и передать в него указатель на этот виджет.

Метод `QMainWindow::statusBar()` возвращает указатель на виджет строки состояния. Кнопка изменения размеров окна, расположенная в нижнем правом углу строки состояния (см. рис. 34.1), является всего лишь подсказкой для пользователя, сообщающей ему о том, что размеры главного окна могут быть изменены. Этот виджет реализован в классе `QSizeGrip`. Получить указатель на него из класса главного окна (`QMainWindow`) невозможно, так как он находится под контролем виджета строки состояния.

Класс действия *QAction*

Класс действия `QAction` предоставляет очень мощный механизм, ускоряющий разработку приложений. Если бы его не было, то вам пришлось бы создавать команды меню, соединять их со слотами, затем создавать панели инструментов и соединять их с теми же слотами

и т. д. Это приводило бы к дублированию программного кода и вызывало бы проблемы при синхронизации элементов пользовательского интерфейса. Например, при необходимости сделать одну из команд меню недоступной, вам нужно было бы сделать ее недоступной в меню, а потом проделать то же самое и с кнопкой панели инструментов этой команды.

Объекты класса `QAction` предоставляют решение этой проблемы и значительно упрощают программирование. Например, если команда меню **File | New** (Файл | Создать) дублируется кнопкой на панели инструментов, для них можно создать один *объект действия*. Тем самым, если вдруг команду потребуется сделать недоступной, мы будем иметь дело только с одним объектом.

`QAction` объединяет следующие элементы интерфейса пользователя:

- текст для всплывающего меню;
- текст для всплывающей подсказки;
- текст подсказки «Что это»;
- «горячие» клавиши;
- ассоциированные значки;
- шрифт;
- текст строки состояния.

Для установки каждого из перечисленных элементов в объекте `QAction` существует свой метод. Например:

```
QAction* pactSave = new QAction("file save action", 0);
pactSave->setText("&Save");
pactSave->setShortcut(QKeySequence("CTRL+S"));
pactSave->setToolTip("Save Document");
pactSave->setStatusTip("Save the file to disk");
pactSave->setWhatsThis("Save the file to disk");
pactSave->setIcon(QPixmap(":/img4.png"));
connect(pactSave, SIGNAL(triggered()), SLOT(slotSave()));
QMenu* pmnuFile = new QMenu("&File");
pmnu->addAction(pactSave);
QToolBar* ptb = new QToolBar("Linker ToolBar");
ptb->addAction(pactSave);
```

Метод `addAction()` позволяет внести объект действия в нужный виджет. В нашем примере это виджет всплывающего меню `QMenu` (указатель `pmnuFile`) и панель инструментов `QToolBar` (указатель `ptb`).

Панель инструментов

Основная цель панели инструментов (Tool Bar) — предоставить пользователю быстрый доступ к командам программы одним нажатием кнопки мыши. Это делает панель инструментов более удобной, чем меню, в котором нужно сделать, по меньшей мере, два нажатия. Еще одно достоинство панели инструментов состоит в том, что она всегда видима, а это освобождает от необходимости тратить время на поиски в меню необходимой команды или вспоминать комбинацию клавиш ускорителя.

Панель инструментов представляет собой область, в которой расположены кнопки, дублирующие часто используемые команды меню. Для панелей инструментов библиотека Qt

предоставляет класс `QToolBar`, который определен в заголовочном файле QToolBar. Процесс создания панели инструментов несложен, и со временем вы сможете формировать панели, имеющие довольно сложную структуру и удовлетворяющие любым требованиям.

Для того чтобы поместить кнопку на панель инструментов, необходимо вызвать метод `addAction()`, который неявно создаст объект действия. В этот метод можно передать растровое изображение, поясняющий текст и соединение со слотом.

> ***Поясняющий текст***
>
> Старайтесь не игнорировать поясняющий текст и всегда его передавать. При отсутствии текста на кнопках инструментов дополнительные небольшие текстовые пояснения (см. *главу 33*) играют важную информационную роль. Эти пояснения всплывают в тот момент, когда пользователь задержит указатель мыши в области кнопки на небольшой промежуток времени.

Наряду с кнопками, в панели инструментов могут быть размещены и любые другие виджеты. Для этого в классе `QToolBar` определен метод `addWidget()`.

Программа (листинг 34.1), окно которой показано на рис. 34.2, демонстрирует использование панелей инструментов. Панели инструментов могут перемещаться с одного места на другое. В нашем случае был использован класс `QMainWindow`.

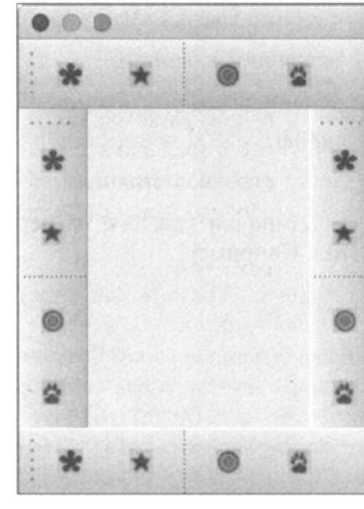


Рис. 34.2. Пример окна с панелями инструментов

Класс `MainWindow`, приведенный в листинге 34.1, унаследован от класса `QMainWindow`. С помощью метода `addToolBar()` в конструкторе класса к главному окну добавляются четыре панели инструментов, создаваемые методом `createToolBar()` класса `MainWindow`. Первый параметр метода `addToolBar()` задает расположение панели инструментов в главном окне приложения. В самом методе `createToolBar()` осуществляется неявное создание четырех объектов действий метода `addAction()`, в который передаются:

- растровое изображение — массив растровых данных, хранящихся в файле формата XPM (см. *главу 19*). Из этих данных создается объект класса `QPixmap`;
- поясняющая надпись (в нашем случае это номер);
- два последних параметра определяют соединение со слотом `slotNoImpl()`, который при вызове отображает диалоговое окно с надписью **Not implemented** (не реализовано).

Разделитель между кнопками вставляется с помощью метода `addSeparator()`.

Листинг 34.1. Использование панелей инструментов (файл MainWindow.h)

```
#pragma once

#include <QtWidgets>
```

```
// ========================================================================
class MainWindow : public QMainWindow {
Q_OBJECT
public:
    MainWindow(QWidget* pwgt = 0) : QMainWindow(pwgt)
    {
        addToolBar(Qt::TopToolBarArea, createToolBar());
        addToolBar(Qt::BottomToolBarArea, createToolBar());
        addToolBar(Qt::LeftToolBarArea, createToolBar());
        addToolBar(Qt::RightToolBarArea, createToolBar());
    }

    QToolBar* createToolBar()
    {
        QToolBar* ptb = new QToolBar("Linker ToolBar");

        ptb->addAction(QPixmap(":/img1.png"), "1", this, SLOT(slotNoImpl()));
        ptb->addAction(QPixmap(":/img2.png"), "2", this, SLOT(slotNoImpl()));
        ptb->addSeparator();
        ptb->addAction(QPixmap(":/img3.png"), "3", this, SLOT(slotNoImpl()));
        ptb->addAction(QPixmap(":/img4.png"), "4", this, SLOT(slotNoImpl()));

        return ptb;
    }

public slots:
    void slotNoImpl()
    {
        QMessageBox::information(0, "Message", "Not implemented");
    }
};
```

Доки

Панель инструментов не предназначена для работы со сложными виджетами. Для этой цели существует класс `QDockWidget`. Представьте себе, что вы создали виджет, составленный из целой серии виджетов, и вам бы хотелось использовать его подобно панели инструментов, то есть позволить пользователю перетаскиванием изменять его местоположение — например, помещать его с нужной стороны окна, а также отделять его от основного окна приложения. На рис. 34.1 этот виджет (док) показан у левой стороны окна, но на самом деле он может располагаться у любой из четырех его сторон. Создание и использование дока в унаследованном от `QMainWindow` классе может выглядеть следующим образом:

```
QDockWidget* pdock = new QDockWidget("MyDock", this);
QLabel*      plbl  = new QLabel("Label in Dock", pdoc);
pdock->setWidget(plbl);
addDockWidget(Qt::LeftDockWidgetArea, pdock);
```

В приведенном примере мы сначала создаем виджет дока (указатель `pdock`) и при создании указываем его название, которое будет отображаться в его верхней части: **MyDock**. Затем

создаем виджет надписи (указатель plbl) и устанавливаем его в виджете дока при помощи метода setWidget(). Далее док добавляется в основное окно приложения вызовом метода QMainWindow::addDockWidget(). В этом методе первым параметром указывается место, где должен быть расположен док, — в нашем случае это Qt::LeftDockWidgetArea, то есть слева. Если нам нужно ограничить возможные места расположения окна дока, то можно воспользоваться методом setAllowedAreas(). Например, если мы хотим запретить размещение док-виджета у правой и левой сторон окна приложения, то есть сделать так, чтобы его можно было размещать только внизу и вверху, то вызов этого метода будет выглядеть следующим образом:

```
pdock->setAllowedAreas(Qt::BottomDockWidgetArea
                       | Qt::TopDockWidgetArea
                      );
```

По умолчанию док-виджеты можно «отрывать» от окна основного приложения. Эта операция делает их самостоятельными окнами, которые возможно перемещать и вставлять в другие области приложения. Все док-виджеты, также по умолчанию, содержат в своем заголовке кнопку закрытия окна. Все это можно изменить вызовом метода setFeatures(). Например, для того чтобы убрать из заголовка док-виджета кнопку закрытия, мы просто не указываем этот флаг вместе с другими флагами:

```
pdock->setFeatures(QDockWidget::DockWidgetMovable
                   | QDockWidget::DockWidgetFloatable
                  );
```

Строка состояния

Этот виджет располагается в нижней части главного окна и отображает, как правило, текстовые сообщения, содержащие информацию о состоянии приложения или короткую справку о командах меню или кнопках панелей инструментов. Строку состояния реализует класс QStatusBar, определенный в заголовочном файле QStatusBar. Различают следующие типы сообщений строки состояния:

- промежуточный — вызывается методом showMessage(). Для очистки строки состояния следует вызвать метод clearMessage(). Если во втором параметре метода showMessage() задан временной интервал, то строка состояния будет очищаться автоматически по его истечении. Примером промежуточного отображения является вывод поясняющего текста для команд меню;
- нормальный — служит для отображения часто изменяющейся информации (например, для отображения позиции указателя мыши). Для этого рекомендуется поместить в строку состояния отдельный виджет. Например, если приложение должно отображать процесс выполнения какой-либо операции, то лучше разместить в строке состояния индикатор выполнения. Чтобы разместить виджет в строке состояния, нужно передать его указатель в метод addWidget(). Виджеты можно также удалять из строки состояния с помощью метода removeWidget();
- постоянный — отображает информацию, необходимую для работы с приложением. Например, для отображения состояния клавиатуры в строку состояния могут быть внесены виджеты надписей, отображающие состояние клавиш <Caps Lock>, <Num Lock>, <Insert> и т. д. Это достигается посредством вызова метода addPermanentWidget(), принимающего указатель на виджет в качестве аргумента.

Следующий пример (листинг 34.2) формирует окно, в котором в строку состояния помещено два виджета надписи для отображения актуальных координат указателя мыши (рис. 34.3).

В листинге 34.2 класс `MainWindow` наследуется от класса `QMainWindow`. Этот класс содержит атрибуты, хранящие указатели на виджеты надписи `m_plblX` и `m_plblY`. Сами виджеты надписей создаются в конструкторе и помещаются в строку состояния методом `addWidget()`.

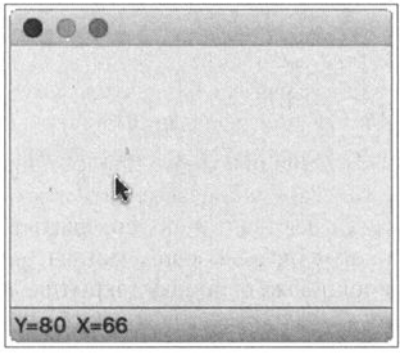


Рис. 34.3. Строка состояния, отображающая актуальную позицию указателя мыши

Для отображения актуальных координат указателя мыши текст виджетам надписи присваивается в методе обработки события `mouseMoveEvent()`. Значения координат указателя мыши возвращаются методами `x()` и `y()` объекта события (указатель `pe`).

Листинг 34.2. Формирование строки состояния, отображающей актуальную позицию указателя мыши (файл MainWindow.h)

```
#pragma once

#include <QtWidgets>

// ============================================================================
class MainWindow : public QMainWindow {
Q_OBJECT
private:
    QLabel* m_plblX;
    QLabel* m_plblY;

protected:
    virtual void mouseMoveEvent(QMouseEvent* pe)
    {
        m_plblX->setText("X=" + QString().setNum(pe->x()));
        m_plblY->setText("Y=" + QString().setNum(pe->y()));
    }

public:
    MainWindow(QWidget* pwgt = 0) : QMainWindow(pwgt)
    {
       setMouseTracking(true);
```

```
        m_plblX = new QLabel(this);
        m_plblY = new QLabel(this);
        statusBar()->addWidget(m_plblY);
        statusBar()->addWidget(m_plblX);
    }
};
```

Окно заставки

При запуске многие приложения показывают так называемое *окно заставки* (Splash Screen). Это окно отображается на время, необходимое для инициализации приложения, и информирует о ходе его запуска. Зачастую такое окно используют для маскировки длительного процесса старта программы.

В библиотеке Qt окно заставки реализовано в классе `QSplashScreen`. Объект этого класса создается в функции `main()` до вызова метода `exec()` объекта приложения. Программа, приведенная в листинге 34.3, отображает перед запуском окно заставки, в котором осуществляется отсчет процесса инициализации в процентах (рис. 34.4).

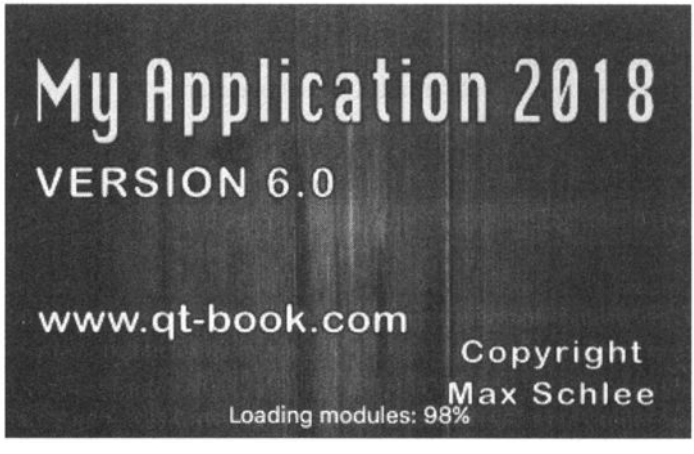


Рис. 34.4. Окно заставки

В листинге 34.3 объект окна заставки создается после объекта приложения. В конструктор передается растровое изображение, которое будет отображаться после вызова метода `show()`. Виджет `QLabel` представляет в этом примере само приложение, которое должно быть запущено. Функция `loadModules()` является эмуляцией загрузки модулей программы, в нее передается адрес объекта окна заставки, чтобы функция могла отображать информацию о процессе загрузки. Объект класса `QTime` (см. *главу 37*) служит для того, чтобы значение переменной `i` увеличивалось только по истечении 40 мсек. Отображение информации выполняется при помощи метода `showMessage()`, в который первым параметром передается текст, вторым — расположение текста (см. табл. 7.1), а третьим — цвет текста (см. табл. 17.1). Вызов метода `finish()` закрывает окно заставки. В этот метод передается указатель на главное окно приложения, и появление этого окна приводит к закрытию окна заставки. Если окно заставки не закрывать, то оно останется видимым до тех пор, пока пользователь не щелкнет на нем мышью.

Листинг 34.3. Создание окна заставки (файл main.cpp)

```
#include <QtWidgets>

// ----------------------------------------------------------------------
void loadModules(QSplashScreen* psplash)
{
    QTime time;
    time.start();

    for (int i = 0; i < 100; ) {
        if (time.elapsed() > 40) {
            time.start();
            ++i;
        }

        psplash->showMessage("Loading modules: "
                             + QString::number(i) + "%",
                             Qt::AlignHCenter | Qt::AlignBottom,
                             Qt::black
                            );
        qApp->processEvents();
    }
}

// ----------------------------------------------------------------------
int main (int argc, char** argv)
{
    QApplication  app(argc, argv);
    QSplashScreen splash(QPixmap(":/splash.png"));

    splash.show();

    QLabel lbl("<H1><CENTER>My Application<BR>"
               "Is Ready!</CENTER></H1>"
              );

    loadModules(&splash);

    splash.finish(&lbl);

    lbl.resize(250, 250);
    lbl.show();

    return app.exec();
}
```

SDI- и MDI-приложения

Существуют два типа приложений, базирующихся на документах. Первый тип — это SDI (Single Document Interface, *однодокументный интерфейс*), второй — MDI (Multiple Document Interface, *многодокументный интерфейс*). В SDI-приложениях рабочая область одновременно является окном приложения, а это значит, что в одном и том же таком приложении невозможно открыть сразу два документа. MDI-приложение предоставляет рабочую область (класса `QMdiArea`), способную размещать в себе окна виджетов, что дает возможность одновременной работы с большим количеством документов.

SDI-приложение

Типичным примером SDI-приложения является программа Блокнот (Notepad) из состава ОС Windows. Пример, приведенный в листингах 34.4–34.9, реализует упрощенный вариант этой программы — простой текстовый редактор (рис. 34.5).

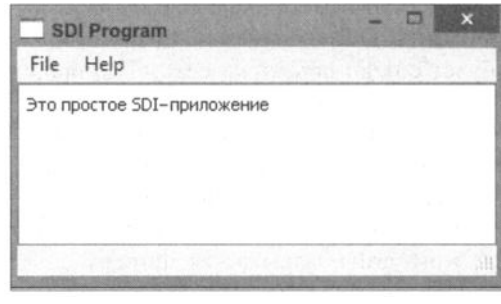


Рис. 34.5. SDI-приложение

Класс `DocWindow`, унаследованный от класса `QTextEdit`, представляет собой окно для редактирования (листинг 34.4). В его определении содержится атрибут `m_strFileName`, в котором хранится имя изменяемого файла. Сигнал `changeWindowTitle()` предназначен для информирования о том, что текстовая область заголовка должна быть изменена. Слоты `slotLoad()`, `slotSave()` и `slotSaveAs()` необходимы для проведения операций чтения и записи файлов.

Листинг 34.4. Определение класса `DocWindow` (файл DocWindow.h)

```
#pragma once

#include <QTextEdit>

// ============================================================================
class DocWindow: public QTextEdit {
Q_OBJECT
private:
    QString m_strFileName;

public:
    DocWindow(QWidget* pwgt = 0);

signals:
    void changeWindowTitle(const QString&);
```

```
public slots:
    void slotLoad  ();
    void slotSave  ();
    void slotSaveAs();
};
```

В конструктор класса, приведенный в листинге 34.5, передается указатель на виджет предка.

Листинг 34.5. Конструктор класса DocWindow (файл DocWindow.cpp)

```
DocWindow::DocWindow(QWidget* pwgt/*=0*/) : QTextEdit(pwgt)
{
}
```

Метод `slotLoad()`, приведенный в листинге 34.6, отображает диалоговое окно открытия файла вызовом статического метода `QFileDialog::getOpenFileName()`, с помощью которого пользователь выбирает файл для чтения. В том случае, если пользователь отменит выбор, нажав на кнопку **Cancel** (Отмена), этот метод вернет пустую строку. В нашем примере это проверяется с помощью метода `QString::isEmpty()`. Если метод `getOpenFileName()` возвратит непустую строку, то будет создан объект класса `QFile`, проинициализированный этой строкой. Передача `QIODevice::ReadOnly` в метод `QFile::open()` говорит о том, что файл открывается только для чтения. В случае успешного открытия файла создается объект потока `stream`, который в нашем примере используется для чтения текста из файла. Чтение всего содержимого файла выполняется при помощи метода `QTextStream::readAll()`, который возвращает его в объекте строкового типа `QString`. Текст устанавливается в виджете методом `setPlainText()`. После этого файл закрывается методом `close()`, а об изменении его местонахождения и имени оповещается отправкой сигнала `changeWindowTitle()`, для того чтобы использующее виджет `DocWindow` приложение могло отобразить эту информацию, изменив заголовок окна.

Листинг 34.6. Метод slotLoad() (файл DocWindow.cpp)

```
void DocWindow::slotLoad()
{
    QString str = QFileDialog::getOpenFileName();
    if (str.isEmpty()) {
        return;
    }

    QFile file(str);
    if (file.open(QIODevice::ReadOnly)) {
        QTextStream stream(&file);
        setPlainText(stream.readAll());
        file.close();

        m_strFileName = str;
        emit changeWindowTitle(m_strFileName);
    }
}
```

Приведенный в листинге 34.7 слот `slotSaveAs()` отображает диалоговое окно сохранения файла с помощью статического метода `QFileDialog::getSaveFileName()`. Если пользователь не нажал в этом окне кнопку **Cancel** (Отмена), и метод вернул непустую строку, то в атрибут `m_strFileName` записывается имя файла, указанное пользователем в диалоговом окне, и вызывается слот `slotSave()`.

Листинг 34.7. Метод `slotSaveAs()` (файл DocWindow.cpp)

```
void DocWindow::slotSaveAs()
{
    QString str = QFileDialog::getSaveFileName(0, m_strFileName);
    if (!str.isEmpty()) {
        m_strFileName = str;
        slotSave();
    }
}
```

Запись в файл представляет собой более серьезный процесс, чем считывание, так как она связана с рядом обстоятельств, которые могут сделать ее невозможной. Например, на диске не хватит места, или он будет не доступен для записи. Для записи в файл нужно создать объект класса `QFile` и передать в него строку с именем файла (листинг 34.8). Затем надо вызвать метод `open()`, передав в него значение `QIODevice::WriteOnly` (флаг, говорящий о том, что будет выполняться запись в файл). В том случае, если файла с таким именем на диске не существует, он будет создан, если существует — он будет открыт для записи. Если файл открыт успешно, то создается промежуточный объект потока, в который при помощи оператора << передается текст виджета, возвращаемый методом `toPlainText()`. После этого файл закрывается методом `QFile::close()`, и отсылается сигнал с новым именем и местонахождением файла. Это делается для того, чтобы эту информацию могло отобразить приложение, использующее наш виджет `DocWindow`.

Листинг 34.8. Метод `slotSave()` (файл DocWindow.cpp)

```
void DocWindow::slotSave()
{
    if (m_strFileName.isEmpty()) {
        slotSaveAs();
        return;
    }

    QFile file(m_strFileName);
    if (file.open(QIODevice::WriteOnly)) {
        QTextStream(&file) << toPlainText();
        file.close();
        emit changeWindowTitle(m_strFileName);
    }
}
```

Класс `SDIProgram` (листинг 34.9) унаследован от класса `QMainWindow`. В его конструкторе создаются три виджета: всплывающие меню **File** (Файл) — указатель `pmnuFile`, **Help** (По-

мощь) — указатель pmnuHelp и виджет созданного нами окна редактирования — указатель pdoc. Затем несколькими вызовами метода addAction() неявно создаются объекты действий и добавляются в качестве команд меню. Третьим параметром указываем слот, с которым должна быть соединена команда, во втором параметре указан сам объект, который содержит этот слот. Таким образом команда **Open...** (Открыть...) соединяется со слотом slotLoad(), команда **Save** (Сохранить) — со слотом slotSave(), а команда **SaveAs...** (Сохранить как...) — со слотом slotSaveAs(). Все эти слоты реализованы в классе DocWindow. Команда **About** (О программе) соединяется со слотом slotAbout(), предоставляемым классом SDIProgram. Метод menuBar() возвращает указатель на виджет меню верхнего уровня, а вызов методов addMenu() добавляет созданные всплывающие меню **File** (Файл) и **Help** (Помощь). Вызов метода setCentralWidget() делает окно редактирования центральным виджетом, то есть рабочей областью нашей программы. Для изменения текстового заголовка программы после загрузки файла или сохранения его под новым именем сигнал changeWindowTitle(), отправляемый виджетом окна редактирования, соединяется со слотом slotChangeWindowTitle(). Метод showMessage(), вызываемый из виджета строки состояния, отображает надпись **Ready** на время, установленное во втором параметре (в нашем примере это 2 сек).

Листинг 34.9. Определение класса SDIProgram (файл SDIProgram.h)

```
#pragma once

#include <QtWidgets>
#include "DocWindow.h"
#include "SDIProgram.h"

// ============================================================================
class SDIProgram : public QMainWindow {
Q_OBJECT
public:
    SDIProgram(QWidget* pwgt = 0) : QMainWindow(pwgt)
    {
        QMenu*      pmnuFile = new QMenu("&File");
        QMenu*      pmnuHelp = new QMenu("&Help");
        DocWindow* pdoc      = new DocWindow;

        pmnuFile->addAction("&Open...",
                            pdoc,
                            SLOT(slotLoad()),
                            QKeySequence("CTRL+O")
                           );
        pmnuFile->addAction("&Save",
                            pdoc,
                            SLOT(slotSave()),
                            QKeySequence("CTRL+S")
                           );
        pmnuFile->addAction("S&ave As...",
                            pdoc,
                            SLOT(slotSaveAs())
                           );
```

```
        pmnuFile->addSeparator();
        pmnuFile->addAction("&Quit",
                            qApp,
                            SLOT(quit()),
                            QKeySequence("CTRL+Q")
                           );
        pmnuHelp->addAction("&About",
                            this,
                            SLOT(slotAbout()),
                            Qt::Key_F1
                           );

        menuBar()->addMenu(pmnuFile);
        menuBar()->addMenu(pmnuHelp);

        setCentralWidget(pdoc);
        connect(pdoc,
                SIGNAL(changeWindowTitle(const QString&)),
                SLOT(slotChangeWindowTitle(const QString&))
               );

        statusBar()->showMessage("Ready", 2000);
    }

public slots:
    void slotAbout()
    {
        QMessageBox::about(this, "Application", "SDI Example");
    }

    void slotChangeWindowTitle(const QString& str)
    {
        setWindowTitle(str);
    }
};
```

MDI-приложение

MDI-приложение позволяет пользователю работать с несколькими открытыми документами. По своей сути оно очень напоминает обычный рабочий стол, только в виртуальном исполнении. Пользователь может разложить в его области несколько окон документов или свернуть их. Окна документов могут перекрывать друг друга, а могут быть развернуты на всю рабочую область.

Рабочая область, внутри которой размещаются окна документов (рис. 34.6), реализуется классом `QMdiArea`. Виджет этого класса выполняет «закулисное» управление динамически создаваемыми окнами документов. Упорядочивание таких окон осуществляется при помощи слотов `tileSubWindows()` и `cascadeSubWindows()`, определенных в этом классе. Метод `QMdiArea::subWindowList()` возвращает список всех содержащихся в нем окон, созданных от класса `QMdiSubWindow`.

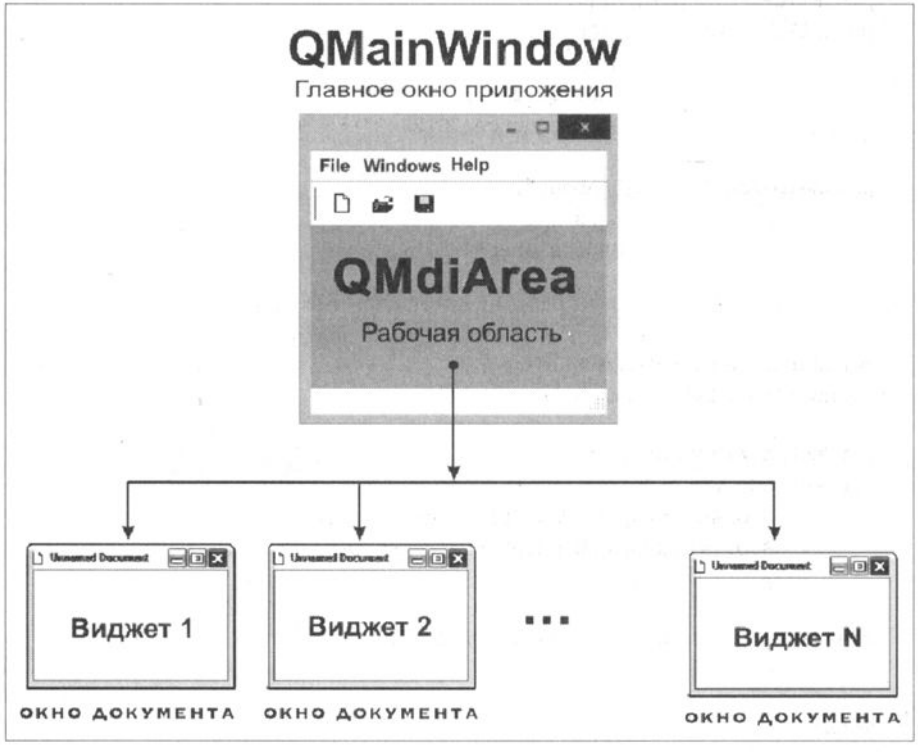


Рис. 34.6. Структура MDI-приложения

Программа (листинги 34.10–34.20), окно которой показано на рис. 34.7, реализует основные функции, присущие MDI-приложению. В качестве класса окна документа использован класс `DocWindow`, задействованный при реализации SDI-приложения (см. листинги 34.4–34.8).

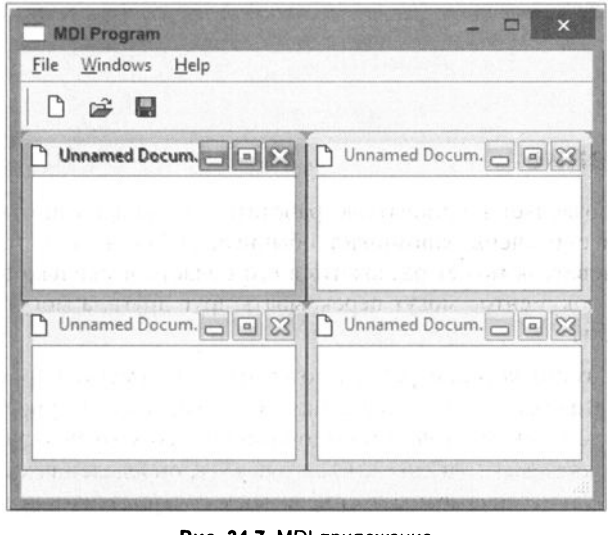


Рис. 34.7. MDI-приложение

Определение класса MDIProgram, приведенное в листинге 34.10, содержит атрибуты, хранящие указатели на виджет рабочей области — m_pma, на всплывающее меню **Windows** (Окна) — m_pmnuWindows и на сопоставитель сигналов — m_psigMapper.

> ***Класс сопоставления сигналов QSignalMapper***
>
> Объекты класса QAction отправляют сигналы triggered() только с булевыми значениями. Нам этого недостаточно, так как мы намереваемся отправлять указатели виджетов окон редактирования. Поэтому мы и прибегли к использованию класса сопоставления сигналов QSignalMapper. Этот класс описан в *разд. «Переопределение сигналов» главы 2.*

В классе MDIProgram определены слоты, предназначенные для работы с окнами документов — slotWindows(), для загрузки и сохранения файлов: slotLoad(), slotSave() и slotSaveAs(), для создания новых документов — slotNewDoc(), а также для предоставления информации о самом приложении — slotAbout().

Листинг 34.10. Определение класса MDIProgram (файл MDIProgram.h)

```
#pragma once

#include <QMainWindow>

class QMenu;
class QMdiArea;
class QSignalMapper;
class DocWindow;

// ============================================================================
class MDIProgram : public QMainWindow {
    Q_OBJECT
private:
    QMdiArea*      m_pma;
    QMenu*         m_pmnuWindows;
    QSignalMapper* m_psigMapper;

    DocWindow* MDIProgram::createNewDoc();

public:
    MDIProgram(QWidget* pwgt = 0);

private slots:
    void slotChangeWindowTitle(const QString&);

private slots:
    void slotNewDoc            (        );
    void slotLoad              (        );
    void slotSave              (        );
    void slotSaveAs            (        );
    void slotAbout             (        );
    void slotWindows           (        );
    void slotSetActiveSubWindow(QWidget*);
};
```

В конструкторе класса `MDIProgram` (листинг 34.11) создаются три объекта действий для команд создания, открытия и сохранения документов: указатели `pactNew`, `pactOpen` и `pactSave` соответственно. Сигнал объектов `triggered()` соединяется со слотами класса `MDIProgram`. Вызовами метода `addAction()` объекты действий добавляются к панели инструментов и в меню.

Команда меню **File | Quit** (Файл | Выход) соединяется со слотом объекта приложения `closeAllWindows()`, который закрывает все окна приложения.

Для создания рабочей области MDI-приложения необходимо создать виджет `QMdiArea`. Чтобы содержимое рабочей области можно было прокручивать, вызываются методы `setHorizontalScrollBarPolicy()` и `setVerticalScrollBarPolicy()`, в которые передается значение `Qt::ScrollBarAsNeeded` — чтобы горизонтальная и вертикальная полосы прокрутки не были постоянно видны, а возникали лишь при необходимости. Установка рабочей области в главном окне виджета выполняется методом `setCentralWidget()`.

После создания объекта сопоставителя сигналов (указатель `m_psigMapper`) его сигнал `mapped()` соединяется со слотом `slotSetActiveSubWindow()`, реализованным в нашем классе `MDIProgram`. Это позволит нам отсылать вместе с сигналами указатели на виджеты, которые будет обрабатывать слот `slotSetActiveSubWindow()`. Об этом слоте будет рассказано далее (см. листинг 34.20).

Метод `showMessage()`, вызываемый из виджета строки состояния, отображает надпись **Ready** в течение 3 сек.

Листинг 34.11. Конструктор `MDIProgram` (файл MDIProgram.cpp)

```
MDIProgram::MDIProgram(QWidget* pwgt/*=0*/) : QMainWindow(pwgt)
{
    QAction* pactNew = new QAction("New File", 0);
    pactNew->setText("&New");
    pactNew->setShortcut(QKeySequence("CTRL+N"));
    pactNew->setToolTip("New Document");
    pactNew->setStatusTip("Create a new file");
    pactNew->setWhatsThis("Create a new file");
    pactNew->setIcon(QPixmap(":/filenew.png"));
    connect(pactNew, SIGNAL(triggered()), SLOT(slotNewDoc()));

    QAction* pactOpen = new QAction("Open File", 0);
    pactOpen->setText("&Open...");
    pactOpen->setShortcut(QKeySequence("CTRL+O"));
    pactOpen->setToolTip("Open Document");
    pactOpen->setStatusTip("Open an existing file");
    pactOpen->setWhatsThis("Open an existing file");
    pactOpen->setIcon(QPixmap(":/fileopen.png"));
    connect(pactOpen, SIGNAL(triggered()), SLOT(slotLoad()));

    QAction* pactSave = new QAction("Save File", 0);
    pactSave->setText("&Save");
    pactSave->setShortcut(QKeySequence("CTRL+S"));
    pactSave->setToolTip("Save Document");
    pactSave->setStatusTip("Save the file to disk");
    pactSave->setWhatsThis("Save the file to disk");
```

```
    pactSave->setIcon(QPixmap(":/filesave.png"));
    connect(pactSave, SIGNAL(triggered()), SLOT(slotSave()));

    QToolBar* ptbFile = new QToolBar("File Operations");
    ptbFile->addAction(pactNew);
    ptbFile->addAction(pactOpen);
    ptbFile->addAction(pactSave);
    addToolBar(Qt::TopToolBarArea, ptbFile);

    QMenu* pmnuFile = new QMenu("&File");
    pmnuFile->addAction(pactNew);
    pmnuFile->addAction(pactOpen);
    pmnuFile->addAction(pactSave);
    pmnuFile->addAction("Save &As...", this, SLOT(slotSaveAs()));
    pmnuFile->addSeparator();
    pmnuFile->addAction("&Quit",
                        qApp,
                        SLOT(closeAllWindows()),
                        QKeySequence("CTRL+Q")
                       );
    menuBar()->addMenu(pmnuFile);

    m_pmnuWindows = new QMenu("&Windows");
    menuBar()->addMenu(m_pmnuWindows);
    connect(m_pmnuWindows, SIGNAL(aboutToShow()), SLOT(slotWindows()));
    menuBar()->addSeparator();

    QMenu* pmnuHelp = new QMenu("&Help");
    pmnuHelp->addAction("&About", this, SLOT(slotAbout()), Qt::Key_F1);
    menuBar()->addMenu(pmnuHelp);

    m_pma = new QMdiArea;
    m_pma->setHorizontalScrollBarPolicy(Qt::ScrollBarAsNeeded);
    m_pma->setVerticalScrollBarPolicy(Qt::ScrollBarAsNeeded);

    setCentralWidget(m_pma);

    m_psigMapper = new QSignalMapper(this);
    connect(m_psigMapper,
            SIGNAL(mapped(QWidget*)),
            this,
            SLOT(slotSetActiveSubWindow(QWidget*))
           );

    statusBar()->showMessage("Ready", 3000);
}
```

Слот `slotNewDoc()`, приведенный в листинге 34.12, создает новое окно документа и делает его видимым.

Листинг 34.12. Метод slotNewDoc() (файл MDIProgram.cpp)

```
void MDIProgram::slotNewDoc()
{
    createNewDoc()->show();
}
```

В листинге 34.13 в методе createNewDoc() создается виджет класса DocWindow, который вызовом метода addSubWindow() добавляется в рабочую область приложения. В метод setAttribute() передается значение Qt::WA_DeleteOnClose, говорящее виджету о том, что он должен быть уничтожен при закрытии своего окна.

Метод setWindowTitle() устанавливает заголовок окна. Небольшое растровое изображение в области заголовка устанавливается методом setWindowIcon(). Для изменения заголовка окна виджета, а также и окна программы, если виджет развернут, сигнал changeWindowTitle() соединяется со слотом slotChangeWindowTitle().

Листинг 34.13. Метод createNewDoc() (файл MDIProgram.cpp)

```
DocWindow* MDIProgram::createNewDoc()
{
    DocWindow* pdoc = new DocWindow;
    m_pma->addSubWindow(pdoc);
    pdoc->setAttribute(Qt::WA_DeleteOnClose);
    pdoc->setWindowTitle("Unnamed Document");
    pdoc->setWindowIcon(QPixmap(":/filenew.png"));
    connect(pdoc,
            SIGNAL(changeWindowTitle(const QString&)),
            SLOT(slotChangeWindowTitle(const QString&))
           );

    return pdoc;
}
```

Слот slotChangeWindowTitle() определен как private и не может быть вызван извне. Поэтому мы не проверяем успешность приведения к типу DocWindow, а сразу вызываем метод setWindowTitle() виджета окна редактирования, установив в нем имя и местонахождение ассоциированного с ним файла (листинг 34.14).

Листинг 34.14. Метод slotChangeWindowTitle() (файл MDIProgram.cpp)

```
void MDIProgram::slotChangeWindowTitle(const QString& str)
{
    qobject_cast<DocWindow*>(sender())->setWindowTitle(str);
}
```

Внутри слота slotLoad() вызывается метод createNewDoc(), который создает новый виджет документа и возвращает его указатель, после чего операция считывания делегируется далее, к виджету созданного документа (листинг 34.15).

Листинг 34.15. Метод slotLoad() (файл MDIProgram.cpp)

```
void MDIProgram::slotLoad()
{
    DocWindow* pdoc = createNewDoc();
    pdoc->slotLoad();
    pdoc->show();
}
```

Слот `slotSave()` получает указатель на текущее окно документа при помощи виджета рабочей области и, если приведение к типу `DocWindow` было успешным, делегирует операцию сохранения (листинг 34.16).

Листинг 34.16. Метод slotSave() (файл MDIProgram.cpp)

```
void MDIProgram::slotSave()
{
    DocWindow* pdoc = qobject_cast<DocWindow*>(m_pma->activeSubWindow());
    if (pdoc) {
        pdoc->slotSave();
    }
}
```

Действия слота `slotSaveAs()` аналогичны действиям слота `slotSave()` (см. листинг 34.16), только с делегированием метода `slotSaveAs()` (листинг 34.17).

Листинг 34.17. Метод slotSaveAs() (файл MDIProgram.cpp)

```
void MDIProgram::slotSaveAs()
{
    DocWindow* pdoc = qobject_cast<DocWindow*>(m_pma->activeSubWindow());
    if (pdoc) {
       pdoc->slotSaveAs();
    }
}
```

Слот `slotAbout()` отображает окно сообщения с информацией о приложении (листинг 34.18).

Листинг 34.18. Метод slotAbout() (файл MDIProgram.cpp)

```
void MDIProgram::slotAbout()
{
    QMessageBox::about(this, "Application", "MDI Example");
}
```

Одним из отличий MDI-приложения от приложения SDI является наличие всплывающего меню **Windows** (Окна), назначение которого — управление окнами документов, находящимися в рабочей области. Для отображения актуальной информации перед показом это меню

необходимо очистить методом `clear()` и затем заполнить списком команд (листинг 34.19). Первыми в меню **Windows** (Окна) добавляются команды меню **Cascade** (Каскад) и **Tile** (Мозаика). В зависимости от наличия в рабочей области окон (опрашивается методом `subWindowList()`), эти две команды при помощи вызова метода `setEnabled()` делаются доступными или нет. Затем в цикле `for` в меню добавляются команды с названиями окон документов. Каждая команда меню вызовом метода `setCheckable()` с параметром `true` получает возможность оснащения флажком, который устанавливается только у команды, ассоциирующейся с активным окном. Является ли окно активным, выясняется с помощью сравнения, результат которого для установки статуса активности передается в метод `setChecked()`.

Далее, сигнал `triggered()` объекта действия (указатель `pact`) соединяется со слотом `map()` сопоставителя сигналов `m_psigMapper`. Благодаря этому, при активации команды меню будет выслан сигнал `mapped()` (см. листинг. 34.11) с указателем на виджет окна редактирования, возвращаемым методом `at()`, который устанавливается методом `setMapping()`.

Листинг 34.19. Метод `slotWindows()` (файл MDIProgram.cpp)

```
void MDIProgram::slotWindows()
{
    m_pmnuWindows->clear();

    QAction* pact = m_pmnuWindows->addAction("&Cascade",
                                             m_pma,
                                             SLOT(cascadeSubWindows())
                                            );

    pact->setEnabled(!m_pma->subWindowList().isEmpty());

    pact = m_pmnuWindows->addAction("&Tile",
                                    m_pma,
                                    SLOT(tileSubWindows())
                                   );
    pact->setEnabled(!m_pma->subWindowList().isEmpty());

    m_pmnuWindows->addSeparator();

    QList<QMdiSubWindow*> listDoc = m_pma->subWindowList();
    for (int i = 0; i < listDoc.size(); ++i) {
        pact = m_pmnuWindows->addAction(listDoc.at(i)->windowTitle());
        pact->setCheckable(true);
        pact->setChecked(m_pma->activeSubWindow() == listDoc.at(i));
        connect(pact, SIGNAL(triggered()), m_psigMapper, SLOT(map()));
        m_psigMapper->setMapping(pact, listDoc.at(i));
    }
}
```

Метод `slotSetActiveSubWindow()` (листинг 34.20) в соответствии с переданным в него указателем виджета делает окно активным. Для начала мы убеждаемся в том, что значение указателя не нулевое, а затем приводим этот указатель к типу `QMdiSubWindow` и передаем его в метод `setActiveSubWindow()`.

Листинг 34.20. Метод slotSetActiveSubWindow() (файл MDIProgram.cpp)

```
void MDIProgram::slotSetActiveSubWindow(QWidget* pwgt)
{
    if (pwgt) {
        m_pma->setActiveSubWindow(qobject_cast<QMdiSubWindow*>(pwgt));
    }
}
```

Резюме

Панель инструментов служит для предоставления быстрого доступа к часто используемым командам меню. Этот виджет представляет собой дочернее окно, в котором помещены кнопки, позволяющие запускать команды одним нажатием левой кнопки мыши.

Виджет строки состояния располагается в нижней части главного окна программы и используется для отображения информационных сообщений трех типов: промежуточного, нормального и постоянного.

Класс действия `QAction` предоставляет возможность централизации всех элементов интерфейса, связанных с конкретной командой, в одном объекте. Это позволяет значительно сократить время разработки программы, а также уменьшить объем исходного кода.

Существуют два типа приложений: поддерживающие SDI (Single Document Interface, однодокументный интерфейс) и поддерживающие MDI (Multiple Document Interface, многодокументный интерфейс). Главное отличие приложения MDI от SDI-приложения состоит в том, что SDI-приложение содержит только одно окно документа, а MDI-приложение способно содержать несколько таких окон, что дает пользователю возможность параллельной работы с несколькими документами.

Класс `QMainWindow` предоставляет уже готовое окно приложения, в котором находятся виджеты, необходимые большинству программ. В центре размещена рабочая область, которая может содержать только один виджет. При помощи класса `QMdiArea` в этой области можно размещать сразу несколько виджетов, что позволяет реализовывать MDI-приложения. Виджеты находятся в рабочей области в виде отдельных окон, которые можно перемещать, изменять их размеры, сворачивать, разворачивать, упорядочивать и т. д.

При запуске программы можно отображать окно заставки, реализованное в классе `QSplashScreen`. Это позволяет скрыть длительный процесс инициализации приложения.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/34-510/** или с помощью следующего QR-кода (рис. 34.8):

Рис. 34.8. QR-код для доступа на страницу комментариев к этой главе

ГЛАВА 35

Рабочий стол (Desktop)

> Все следует делать настолько простым, насколько это возможно, но не проще.
>
> *Альберт Эйнштейн*

До настоящего времени все наши программы не выходили за пределы графической области своего окна, но в некоторых случаях в этом есть необходимость, поскольку преследуются особые цели. В этой главе мы рассмотрим приложение, расположенное в области уведомлений, которая представляет собой пример такой программы. Кроме того, мы расскажем о работе с виджетом экрана.

Область уведомлений

С областью уведомлений (Notification area) практически все хорошо знакомы. В Windows (рис. 35.1) она называется *панелью задач* (Taskbar Notification Area), хотя часто используется неофициальный термин «системный трей» (System Tray), и находится в нижнем правом углу рабочего стола (то же месторасположение принято в KDE (рис. 35.2), а вот в Mac OS X она называется Menu Extras и располагается в верхнем правом углу (рис. 35.3). Вот такие разные названия по своей сути одинаковых вещей. Основное назначение этой области — индикация состояния программ и системы.



Рис. 35.1. Windows



Рис. 35.2. KDE/Linux



Рис. 35.3. Mac OS X

В области уведомлений обычно расположены индикатор часов, индикатор заряда батареи (для ноутбуков), значок регулятора громкости звука, отображение сетевого подключения и многое другое. Такие приложения, как, например, Skype, WhatsApp и Viber, загружают в область уведомлений Windows свой собственный значок. Это позволяет быстро уведомлять пользователей о наступающих событиях — например, получении нового сообщения, а также дает возможность быстро открыть основное окно этих программ двойным щелчком на их значке. Такой прием, кроме того, позволяет не загромождать панель задач, поскольку посредством смены значка и вывода сообщений можно уведомить пользователя о текущем состоянии программы или об ошибках.

Итак, хотите воспользоваться этими возможностями в ваших программах? Тогда перейдем поскорее к классу `QSystemTrayIcon`. Именно он и реализует все эти замечательные возможности. Его использование довольно просто. Для того чтобы установить значок, нужно вызвать метод `setIcon()`, а для его отображения в области уведомления — вызвать метод `show()`. Если вам необходимо отобразить сообщение для пользователя, то следует вызвать метод `showMessage()`.

Желательно также установить всплывающую подсказку для пользователя при помощи метода `setToolTip()` — чтобы он мог увидеть имя вашей программы и, возможно, информацию о ее состоянии, наведя указатель мыши на ее значок. И, конечно же, в большинстве случаев может потребоваться установить контекстное меню, чтобы дать возможность пользователю взаимодействовать с вашей программой непосредственно из области уведомлений. Для этого предусмотрен метод `setContextMenu()`.

Проиллюстрируем все указанные возможности написанием простой программы. Эта программа (листинги 35.1–35.7) имеет контекстное меню, из которого пользователь может выбрать следующие команды (рис. 35.4):

- **Show/Hide Application Window** (Показать/скрыть окно приложения) — отображает показанное на рис. 35.5 окно. Если же окно до этого было видимо, то оно будет скрыто;
- **Show Message** (Выдать сообщение) — выводит сообщение, показанное на рис. 35.6;
- **Change Icon** (Сменить значок) — изменяет изображение снежинки на звездочку и наоборот;
- **Quit** (Выход).

Кроме того, наше приложение будет обладать подсказкой (рис. 35.7).

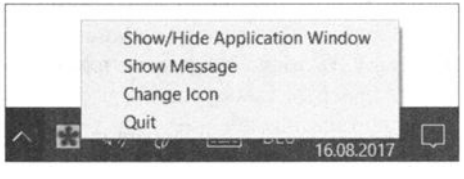


Рис. 35.4. Контекстное меню программы

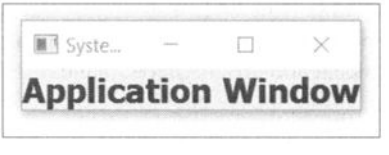


Рис. 35.5. Основное окно программы

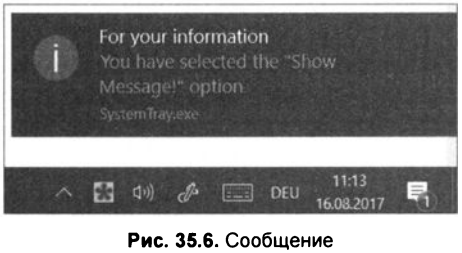


Рис. 35.6. Сообщение



Рис. 35.7. Всплывающая подсказка

В основной программе, приведенной в листинге 35.1, мы только создаем виджет. Поскольку закрытие окна приложения ни в коем случае не должно завершать нашу программу, мы вызываем статический метод `QApplication::setQuitOnLastWindow()` и передаем в него значение `false`. Окна виджетов Qt по умолчанию невидимы, и это замечательно, так как в основном приложения, предназначенные для области уведомлений, после запуска показы-

вают в этой области только свой значок и никаких окон. По этой причине в основной программе нет привычного вызова метода `show()`.

Листинг 35.1. Основная программа — создание виджета (файл main.cpp)

```
#include <QtWidgets>
#include "SystemTray.h"

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    SystemTray   st;

    QApplication::setQuitOnLastWindowClosed(false);

    return app.exec();
}
```

В заголовочном файле, приведенном в листинге 35.2, мы определяем класс окна нашего приложения. Он наследуется от класса `QLabel`, чтобы мы могли отобразить надпись. Наш класс содержит три атрибута:

- объект `QSystemTrayIcon` (указатель `m_ptrayIcon`), который и предоставит нам возможность использования области уведомлений;
- виджет `QMenu` (указатель `m_ptrayIconMenu`), который необходим для реализации контекстного меню;
- переменную булевого типа `m_bIconSwitcher`, нужную нам для управления сменой растровых изображений.

Мы перегрузили метод события закрытия окна `closeEvent()`, поскольку наше приложение должно реагировать на него иначе.

Три слота: `slotShowHide()`, `slotShowMessage()` и `slotChangeIcon()` — реализуют команды контекстного меню.

Листинг 35.2. Определение класса `SystemTray` (файл SystemTray.h)

```
#pragma once

#include <QLabel>

class QSystemTrayIcon;
class QMenu;

// ============================================================================
class SystemTray : public QLabel {
Q_OBJECT
private:
    QSystemTrayIcon* m_ptrayIcon;
    QMenu*           m_ptrayIconMenu;
    bool             m_bIconSwitcher;
```

```
protected:
    virtual void closeEvent(QCloseEvent*);

public:
    SystemTray(QWidget* pwgt = 0);

public slots:
    void slotShowHide   ();
    void slotShowMessage();
    void slotChangeIcon ();
};
```

В конструкторе (листинг 35.3) мы создаем объекты действий (`QAction`) для наших команд и соединяем их соответствующими их назначению слотами. Затем мы создаем меню (`QMenu`) и добавляем в него наши объекты действий. Последним создаем объект области уведомлений (`QSystemTrayIcon`), который в качестве предка получает указатель `this`. Метод `setContextMenu()` устанавливает только что созданное нами контекстное меню. Метод `setToolTip()` устанавливает надпись всплывающей подсказки **System Tray**. Установка растрового изображения происходит в слоте `slotChangeIcon()`, поэтому мы просто вызываем его. Наконец, метод `show()` добавит растровое изображение программы к области уведомлений.

Листинг 35.3. Конструктор `SystemTray()` (файл SystemTray.cpp)

```
SystemTray::SystemTray(QWidget* pwgt /*=0*/)
    : QLabel("<H1>Application Window</H1>", pwgt)
    , m_bIconSwitcher(false)
{
    setWindowTitle("System Tray")

    QAction* pactShowHide =
        new QAction("&Show/Hide Application Window", this);

    connect(pactShowHide, SIGNAL(triggered()),
            this,         SLOT(slotShowHide())
           );

    QAction* pactShowMessage = new QAction("S&how Message", this);
    connect(pactShowMessage, SIGNAL(triggered()),
            this,            SLOT(slotShowMessage())
           );

    QAction* pactChangeIcon = new QAction("&Change Icon", this);
    connect(pactChangeIcon, SIGNAL(triggered()),
            this,           SLOT(slotChangeIcon())
           );

    QAction* pactQuit = new QAction("&Quit", this);
    connect(pactQuit, SIGNAL(triggered()), qApp, SLOT(quit()));
```

```
    m_ptrayIconMenu = new QMenu(this);
    m_ptrayIconMenu->addAction(pactShowHide);
    m_ptrayIconMenu->addAction(pactShowMessage);
    m_ptrayIconMenu->addAction(pactChangeIcon);
    m_ptrayIconMenu->addAction(pactQuit);

    m_ptrayIcon = new QSystemTrayIcon(this);
    m_ptrayIcon->setContextMenu(m_ptrayIconMenu);
    m_ptrayIcon->setToolTip("System Tray");

    slotChangeIcon();

    m_ptrayIcon->show();
}
```

Большинство программ, рассчитанных на область уведомлений, скрывают свое окно, а не закрывают его. Для этого мы перегрузили метод обработки события закрытия окна виджета (листинг 35.4). В нем мы проверяем, является ли видимым значок в области уведомлений, и если да, то просто вызываем метод `hide()`, чтобы спрятать окно.

Листинг 35.4. Метод `closeEvent()` (файл SystemTray.cpp)

```
/*virtual*/void SystemTray::closeEvent(QCloseEvent* pe)
{
    if (m_ptrayIcon->isVisible()) {
        hide();
    }
}
```

Слот `slotShowHide()`, приведенный в листинге 35.5, закреплен за командой **Show/Hide Application Window**. Всякий раз при его вызове он меняет режим видимости на противоположный. То есть, если виджет был видимым, он станет невидимым, и наоборот.

Листинг 35.5. Слот `slotShowHide()` Файл SystemTray.cpp.

```
void SystemTray::slotShowHide()
{
    setVisible(!isVisible());
}
```

Слот `slotShowMessage()` закреплен за командой **Show Message**. Вызовом метода `showMessage()` он отображает сообщение (листинг 35.6). Первый параметр — это заголовок сообщения, второй — само сообщение, третий задает значок, который будет отображаться с информацией, и четвертый — промежуток времени, отведенный для отображения сообщения на экране, в нашем случае это 3 сек.

Листинг 35.6. Слот `slotShowMessage()` (файл SystemTray.cpp)

```
void SystemTray::slotShowMessage()
{
    m_ptrayIcon->showMessage("For your information",
```

```
                                "You have selected the "
                                "\"Show Message!\" option",
                                QSystemTrayIcon::Information,
                                3000
                               );
}
```

Слот `slotChangeIcon()` закреплен за командой **Change Icon** и предназначен для установки в области уведомления растрового изображения (листинг 35.7). При его вызове мы всякий раз меняем значение булевой переменной `m_bIconSwitcher` и, в зависимости от ее состояния, используем одно из двух растровых изображений: img1.bmp либо img2.bmp. Затем мы устанавливаем его в области уведомлений вызовом метода `setIcon()`.

Листинг 35.7. Слот `slotChangeIcon()` (файл SystemTray.cpp)

```
void SystemTray::slotChangeIcon()
{
    m_bIconSwitcher = !m_bIconSwitcher;
    QString strPixmapName = m_bIconSwitcher ? ":/images/img1.bmp"
                                            : ":/images/img2.bmp";
    m_ptrayIcon->setIcon(QPixmap(strPixmapName));
}
```

Виджет экрана

Класс `QDesktopWidget` отвечает за доступ к содержимому графической информации экрана. Кроме того, этот класс может оказаться очень полезен для правильного позиционирования окна вашего приложения, — ведь пользователь может установить различные разрешения экрана. Этот виджет существует только в одном экземпляре, и его нельзя создать, но зато можно получить на него указатель. Это достигается вызовом статического метода `desktop()`, определенного в классе `QApplication`. Таким образом, чтобы отобразить ваше окно строго посередине экрана, можно поступить следующим образом:

1. Получить указатель на виджет экрана, вызвав `QApplication::desktop()`.
2. Определить текущее разрешение экрана при помощи методов `width()` и `height()`.
3. Вычислить позицию окна посередине в соответствии с текущим разрешением и размерами окна приложения: `width()` и `height()` самого виджета окна.
4. Установить ее вызовом метода `move()`.

Все это можно реализовать одной программной строкой:

```
pwgt->move((QApplication::desktop()->width() - pwgt->width()) / 2,
           (QApplication::desktop()->height() - pwgt->height()) / 2
          );
```

Кроме того, этот класс предоставляет следующую информацию о количестве мониторов в системе, их разрешении и способен вернуть экран, на котором расположен определенный пиксел виджета (это может оказаться очень полезным, например, для запоминания в установках координат виджетов вместе с мониторами, на которых они расположены, чтобы впоследствии быть в состоянии восстановить все как было):

- `screenCount()` — возвращает количество мониторов системы;
- `primaryScreen()` — возвращает номер монитора, который отконфигурирован как основной;
- `isVirtualDesktop()` — позволяет распознать режим виртуального рабочего стола. В этом случае несколько мониторов используются как один общий экран, и окно приложения может перемещаться с одного монитора на другой или сразу находиться на нескольких мониторах одновременно;
- `screenNumber()` — принимает в качестве аргумента координаты (объект `QPoint`) и возвращает номер монитора, на котором расположен пиксел с этими координатами. Например:

```
int nScreen = desktopWidget->screenNumber(QPoint(320, 115));
```

У каждого из экранов можно опросить их актуальное горизонтальное и вертикальное разрешение, передав в метод `QDesktopWidget::screenGeometry()` номер экрана и вызвав методы `width()` и `height()` соответственно. Следующий пример выводит все имеющиеся в системе мониторы с их актуальными разрешениями:

```
QDesktopWidget* pwgt = QApplication::desktop();
for (int i = 0; i < pwgt->screenCount(); ++i) {
    qDebug() << "Screen:" << i;
    qDebug() << "width:" << pwgt->screenGeometry(i).width();
    qDebug() << "height:" << pwgt->screenGeometry(i).height();
}
```

Если пользователь изменит разрешение одного из экранов, то класс `QDesktopWidget` вышлет сигнал `resized()` с номером экрана, а при изменении количества используемых мониторов отправляется сигнал `screenCountChanged()`.

Теперь давайте вернемся к возможности доступа к графической информации экрана. Ее можно проиллюстрировать на примере программы (листинги 35.8–35.11), которая при нажатии на кнопку **Capture Screen** (Захватить экран) делает снимок с экрана компьютера и тут же показывает сделанный снимок (рис. 35.8).

В основной программе (листинг 35.8) мы создаем виджет определенного в листингах 35.9–35.11 класса `GrabWidget`, в котором реализуется возможность снятия снимка с экрана.

Листинг 35.8. Программа, делающая снимки с экрана (файл main.cpp)

```
#include <QtWidgets>
#include "GrabWidget.h"

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    GrabWidget   wgt;

    wgt.show();

    return app.exec();
}
```

В самом классе `GrabWidget` (листинг 35.9) мы определяем указатель на виджет надписи (`m_plbl`), который нам понадобится для отображения только что «сфотографированной»

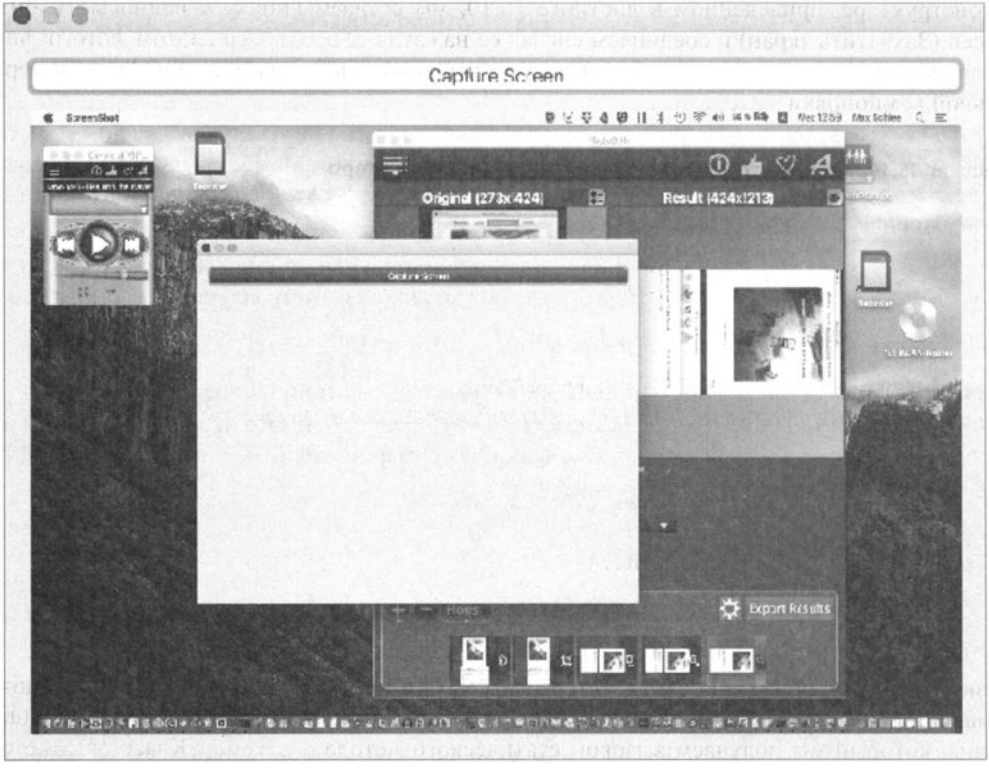


Рис. 35.8. Программа, делающая снимки с экрана

области экрана. Класс также содержит слот `slotGrabScreen()`, который будет реализовывать собственно процедуру фотографирования экрана.

Листинг 35.9. Определение класса `GrabWidget` (Файл GrabWidget.h)

```
#pragma once

#include <QWidget>

class QLabel;

// ============================================================================
class GrabWidget : public QWidget {
Q_OBJECT
private:
    QLabel* m_plbl;

public:
    GrabWidget(QWidget* pwgt = 0);

public slots:
    void slotGrabScreen();
};
```

В конструкторе, приведенном в листинге 35.10, мы создаем кнопку с надписью **Capture Screen** (Захватить экран) и соединяем сигнал ее нажатия `clicked()` со слотом фотографирования экрана `slotGrabScreen()`. Затем кнопка и надпись добавляются при помощи вертикальной компоновки `pvbxLayout`.

Листинг 35.10. Конструктор `GrabWidget()` (файл GrabWidget.cpp)

```
GrabWidget::GrabWidget(QWidget* pwgt /*=0*/) : QWidget(pwgt)
{
    resize(640, 480);

    m_plbl = new QLabel;

    QPushButton* pcmd = new QPushButton("Capture Screen");
    connect(pcmd, SIGNAL(clicked()), SLOT(slotGrabScreen()));

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(pcmd);
    pvbxLayout->addWidget(m_plbl);
    setLayout(pvbxLayout);
}
```

В листинге 35.11 приведена самая интересная, хотя и несложная, часть программы, позволяющая получить содержимое экрана. Сделать это можно с помощью указателя на виджет экрана, который мы получаем вызовом статического метода `desktop()`. Класс `QPixmap` предоставляет возможность получать графическое содержимое виджета в виде растрового изображения, что достигается вызовом метода `grabWindow()` и передачей в него идентификационного номера окна виджета, который, в свою очередь, возвращается методом `winId()`.

В метод `grabWindow()` можно передать также вторым аргументом и прямоугольную область, в пределах которой нам нужен снимок. Если второй аргумент отсутствует, то фотографируется сразу вся область виджета. Но сначала нам понадобится вызвать метод `screen()`, чтобы получить указатель на виджет конкретного экрана, — ведь пользователь может иметь не обязательно только одну, а две, три и более экранных областей, отображаемых на разных мониторах. Вызывая этот метод без аргументов, мы получаем указатель на виджет экранной области по умолчанию.

> ***Экранные области пользователя***
>
> Сколько именно пользователь имеет экранных областей, можно узнать вызовом метода `QDesktopWidget::screenCount()`.

Остался последний штрих — мы приводим размеры полученного изображения в соответствие с размером виджета надписи (вызывая метод `scaled()`), устанавливая его вызовом метода `setPixmap()`.

Листинг 35.11. Слот `slotGrabScreen()` (файл GrabWidget.cpp)

```
void GrabWidget::slotGrabScreen()
{
    QDesktopWidget* pwgt = QApplication::desktop();
    QPixmap         pic  = QPixmap::grabWindow(pwgt->screen()->winId());
```

```
    m_plbl->setPixmap(pic.scaled(m_plbl->size()));
}
```

Класс сервиса рабочего стола

За сервис рабочего стола отвечает класс `QDesktopServices`. Самое частое применение этого класса заключается в вызове почтового клиента или отконфигурированного в системе браузера по определенной веб-ссылке. Запустить веб-браузер по нужной ссылке можно, например, следующим образом:

```
bool bRes = QDesktopServices::openUrl(QUrl("http://www.bhv.ru"));
qDebug() << "Result:" << bRes;
```

С помощью этого же метода можно открыть и локальный файл. Исходя из его типа, система сама выберет программу для его открытия:

```
QDesktopServices::openUrl(QUrl::fromLocalFile("C:\\myfile.txt"));
```

> ***Пути к каталогам пользователя: класс QStandardPaths***
>
> Раньше с помощью класса `QDesktopServices` можно было найти пути к каталогам пользователя, например, узнать, где хранятся его музыкальные файлы, фильмы, документы, шрифты, фотографии и т. д. Теперь для этого нужно использовать класс `QStandardPaths`. Этот класс предоставляет метод `writableLocation()`, в параметре которого нужно указать нужный тип, например, для фильмов: `QStandardPaths::MoviesLocation`, для фото: `QStandardPaths::PicturesLocation` и так далее.

Резюме

Область уведомлений (в Windows — панель задач) информирует пользователей о заряде батареи, статусе приложения, а также позволяет выполнять некоторые настройки — например, регулировать громкость динамиков компьютера. Работа с областью уведомлений реализована классом `QSystemTrayIcon`.

За доступ к содержимому экрана отвечает класс `QDesktopWidget`. Его можно также использовать в целях правильного позиционирования окон приложения на экране компьютера.

Класс `QDesktopServices` отвечает за вызов веб-браузера и открытие заданных типов файлов программами, зарегистрированными в системе.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/35-510/** или с помощью следующего QR-кода (рис. 35.9):



Рис. 35.9. QR-код для доступа на страницу комментариев к этой главе



ЧАСТЬ VI

Особые возможности Qt

Определи последовательность занимающих тебя вопросов, начиная с самых важных. Потом настройся на то, чтобы воспринять ответы. Они повсюду — во всяком событии, во всякой вещи.

Борис Акунин

Глава 36. Работа с файлами, каталогами и потоками ввода/вывода

Глава 37. Дата, время и таймер

Глава 38. Процессы и потоки

Глава 39. Программирование поддержки сети

Глава 40. Работа с XML

Глава 41. Программирование баз данных

Глава 42. Динамические библиотеки и система расширений

Глава 43. Совместное использование Qt с платформозависимыми API

Глава 44. Qt Designer. Быстрая разработка прототипов

Глава 45. Проведение тестов

Глава 46. Qt WebEngine

Глава 47. Интегрированная среда разработки Qt Creator

Глава 48. Рекомендации по миграции программ из Qt 4 в Qt 5

ГЛАВА 36

Работа с файлами, каталогами и потоками ввода/вывода

> Цивилизация движется вперед путем увеличения чисел операций, которые мы можем осуществлять, не раздумывая над ними.
>
> *Альфред Норт Уайтхед*

Редко встречается приложение, которое не обращается к файлам. Работа с каталогами (или *папками*, в терминологии ОС Windows) и файлами — это та область, в которой не все операции являются платформонезависимыми, поэтому Qt предоставляет свою собственную поддержку таких операций, которая базируется на следующих классах:

- `QDir` — для работы с каталогами;
- `QFile` — для работы с файлами;
- `QFileInfo` — для получения файловой информации;
- `QIODevice` — абстрактный класс для ввода/вывода;
- `QBuffer` — для эмуляции файлов в памяти компьютера.

Ввод/вывод. Класс *QIODevice*

Класс `QIODevice` — это абстрактный класс, обобщающий устройство ввода/вывода, который содержит виртуальные методы для открытия и закрытия устройства ввода/вывода, а также для чтения и записи блоков данных или отдельных символов.

Реализация конкретного устройства происходит в унаследованных классах. В Qt есть четыре класса, наследующих класс `QIODevice` (рис. 36.1):

- `QFile` — класс для проведения операций с файлами;
- `QBuffer` — класс буфера, который позволяет записывать и считывать данные в массив `QByteArray`, как будто бы это устройство или файл;
- `QAbstractSocket` — базовый класс для сетевой коммуникации посредством сокетов (см. *главу 39*);
- `QProcess` — класс, предоставляющий возможность запуска процессов с дополнительными аргументами (см. *главу 38*) и позволяющий обмениваться информацией с этими процессами посредством методов, определенных в `QIODevice`.

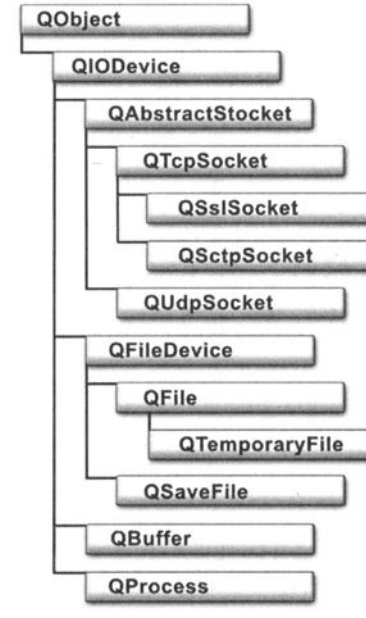


Рис. 36.1. Иерархия классов ввода/вывода

Для работы с устройством его необходимо открыть в одном из режимов, определенных в заголовочном файле класса `QIODevice`:

- `QIODevice::NotOpen` — устройство не открыто (это значение не имеет смысла передавать в метод `open()`);
- `QIODevice::ReadOnly` — открытие устройства только для чтения данных;
- `QIODevice::WriteOnly` — открытие устройства только для записи данных;
- `QIODevice::ReadWrite` — открытие устройства для чтения и записи данных (то же, что `ReadOnly | WriteOnly`);
- `QIODevice::Append` — открытие устройства для добавления данных;
- `QIODevice::Unbuffered` — открытие для непосредственного доступа к данным в обход промежуточных буферов чтения и записи;
- `QIODevice::Text` — служит для преобразования символов переноса строки в зависимости от платформы. В ОС Windows используется комбинация `"\r\n"`, а в Mac OS X и UNIX — `"\r"`;
- `QIODevice::Truncate` — все данные устройства, по возможности, должны быть удалены при открытии.

Для того чтобы в любой момент времени исполнения программы узнать, в каком из режимов было открыто устройство, нужно вызвать метод `openMode()`.

Считывать и записывать данные можно с помощью методов `read()` и `write()`. Для чтения всех данных сразу определен метод `readAll()`, который возвращает их в объекте типа `QByteArray`. Строку или символ можно прочитать методами `readLine()` и `getChar()` соответственно.

В классе `QIODevice` определен метод для смены текущего положения: `seek()`. Получить текущее положение можно вызовом метода `pos()`. Но не забывайте, что эти методы применимы только для прямого доступа к данным. При последовательном доступе, каким является сетевое соединение, они теряют смысл. Более того, в этом случае теряет смысл и метод `size()`, возвращающий размер данных устройства. Все эти операции применимы только для классов `QFile`, `QBuffer` и `QTemporaryFile`.

Для создания собственного класса устройства ввода/вывода, для которого Qt не предоставляет поддержки, необходимо унаследовать класс `QIODevice` и реализовать в нем методы `readData()` и `writeData()`. В большинстве случаев может потребоваться переопределить методы `open()`, `close()` и `atEnd()`.

Благодаря интерфейсу класса `QIODevice`, можно работать со всеми устройствами одинаково, и при работе обычно не имеет значения, является ли устройство файлом, буфером или дру-

гим устройством. Например, выводить на консоль данные из любого устройства можно с помощью такого метода:

```
void print(QIODevice* pdev)
{
    char    ch;
    QString str;

    pdev->open(QIODevice::ReadOnly);
    for (; !pdev->atEnd();) {
        pdev->getChar(&ch);
        str += ch;
    }
    pdev->close();
    qDebug() << str;
}
```

Класс `QIODevice` предоставляет ряд методов, с помощью которых можно получить информацию об устройстве ввода/вывода. Например, одни устройства могут только записывать информацию, другие — только считывать, а третьи способны делать и то, и другое. Чтобы проверить, какие операции доступны при работе с устройством, следует воспользоваться методами `isReadable()` и `isWriteable()`.

Работа с файлами. Класс *QFile*

Класс `QFile` унаследован от класса `QIODevice` (см. рис. 36.1). В нем содержатся методы для работы с файлами: открытия, закрытия, чтения и записи данных. Создать объект можно, передав в конструкторе строку, содержащую имя файла. Можно ничего не передавать в конструкторе, а сделать это после создания объекта вызовом метода `setName()`. Например:

```
QFile file;
file.setName("file.dat");
```

В процессе работы с файлами иногда требуется узнать, открыт файл или нет. Для этого вызывается метод `QIODevice::isOpen()`, который вернет значение `true`, если файл открыт, иначе — `false`. Чтобы закрыть файл, нужно вызвать метод `close()`. С закрытием осуществляется запись всех данных буфера. Если требуется выполнить запись данных буфера в файл без его закрытия, то вызывается метод `QFile::flush()`.

Проверить, существует ли нужный вам файл, можно статическим методом `QFile::exists()`. Этот метод принимает строку, содержащую полный или относительный путь к файлу. Если файл найден, то метод возвратит значение `true`, в противном случае — `false`. Для проведения этой операции существует и нестатический метод `QFile::exists()`, который проверяет существование файла, возвращаемого методом `fileName()`. Методы `QIODevice::read()` и `QIODevice::write()` позволяют считывать и записывать файлы блоками. Продемонстрируем применение некоторых методов работы с файлами:

```
QFile file1("file1.dat");
QFile file2("file2.dat ");
if (file2.exists()) {
    // Файл уже существует. Перезаписать?
}
```

```
if (!file1.open(QIODevice::ReadOnly)) {
    qDebug() << "Ошибка открытия для чтения";
}
if (!file2.open(QIODevice::WriteOnly)) {
    qDebug() << "Ошибка открытия для записи";
}
char a[1024];
while(!file1.atEnd()) {
    int nBlocksize = file1.read(a, sizeof(a));
    file2.write(a, nBlocksize);
}
file1.close();
file2.close();
```

Если требуется считать или записать данные за один раз, то используют методы `QIODevice::write()` и `QIODevice::readAll()`. Все данные можно считать в объект класса `QByteArray`, а потом записать из него в другой файл:

```
QFile file1("file1.dat");
QFile file2("file2.dat");

if (file2.exists()) {
    // Файл уже существует. Перезаписать?
}
if (!file1.open(QIODevice::ReadOnly)) {
   qDebug() << "Ошибка открытия для чтения";
}
if (!file2.open(QIODevice::WriteOnly)) {
   qDebug() << "Ошибка открытия для записи";
}
QByteArray a = file1.readAll();
file2.write(a);

file1.close();
file2.close();
```

Считывание всех данных сразу

Операция считывания всех данных сразу при большом размере файла может занять много оперативной памяти, а значит, к этому следует прибегать только в случаях острой необходимости или в том случае, когда файлы занимают мало места. Расход памяти при считывании сразу всего файла можно значительно сократить при условии, что файл содержит избыточную информацию. Тогда можно воспользоваться функциями сжатия `qCompress()` и `qUncompress()`, которые работают с классом `QByteArray`. Эти функции получают в качестве аргумента объект класса `QByteArray` и возвращают в качестве результата новый объект класса `QByteArray`.

Для удаления файла класс `QFile` содержит статический метод `remove()`. В этот метод необходимо передать строку, содержащую полный или относительный путь удаляемого файла.

Класс *QBuffer*

Класс `QBuffer` унаследован от класса `QIODevice` (см. рис. 36.1) и представляет собой эмуляцию файлов в памяти компьютера (memory mapped files). Это позволяет записывать информацию в оперативную память и использовать объекты как обычные файлы: открывать при помощи метода `open()` и закрывать методом `close()`. При помощи методов `write()` и `read()` можно считывать и записывать блоки данных. Это же можно сделать при помощи потоков, которые будут рассмотрены далее, например:

```
QByteArray arr;
QBuffer    buffer(&arr);
buffer.open(QIODevice::WriteOnly);
QDataStream out(&buffer);
out << QString("Message");
```

Как видно из этого примера, сами данные сохраняются внутри объекта класса `QByteArray`. При помощи метода `buffer()` можно получить константную ссылку на внутренний объект `QByteArray`, а посредством метода `setBuffer()` — устанавливать другой объект `QByteArray` для его использования в качестве внутреннего.

Метод `data()` идентичен методу `buffer()`, но метод `setData()` отличается от `setBuffer()` тем, что получает не указатель на объект `QByteArray`, а константную ссылку на него для копирования его данных.

Класс `QBuffer` полезен для проведения операций кэширования. Например, можно считывать файлы растровых изображений в объекты класса `QBuffer`, а затем, по необходимости, получать данные из них.

Класс *QTemporaryFile*

Иногда приложению может потребоваться создать временный файл. Это бывает связано, например, с промежуточным хранением большого объема данных или передачей этих данных какой-либо другой программе.

Класс `QTemporaryFile` предоставляет удобный способ работы с временными файлами. Этот класс самостоятельно создает имя файла, гарантируя его уникальность, — чтобы не возникало конфликтов, в результате которых могли бы пострадать уже существующие файлы. Сам файл будет расположен в каталоге для промежуточных данных, местонахождение которого можно получить вызовом метода `QDir::tempPath()`. С уничтожением объекта будет уничтожен и сам временный файл.

Работа с каталогами. Класс *QDir*

Разные платформы имеют различные способы представления путей. ОС Windows содержит буквы дисков, например: C:\Windows\System. ОС UNIX использует корневой каталог /, например: /usr/bin. Обратите внимание, что для разделения имен каталогов в обоих представлениях используются разные знаки. Для представления каталогов в платформонезависимом виде в Qt имеется класс `QDir`.

Этот класс содержит целый ряд статических методов, которые позволяют определить:

- `QDir::current()` — путь текущего каталога приложения;
- `QDir::root()` — корневой каталог;

- `QDir::drives()` — указатель на объект класса `QFileInfo`, содержащий список с узловыми каталогами (для ОС Windows это будут C:\, D:\ и т. д.);
- `QDir::home()` — персональный католог пользователя.

> ***Методы для определения пути к каталогам***
>
> Класс `QDir` не предоставляет методов для определения каталога, из которого было запущенно приложение. Если нужно узнать этот путь, то следует воспользоваться либо методом `QApplication::applicationDirPath()`, либо методом `QApplication::applicationFilePath()`, возвращающим еще и имя приложения.

Существование каталога можно проверить с помощью метода `exists()`. Чтобы перемещаться по каталогам, можно использовать метод `cd()`, передав в качестве параметра путь к каталогу, или метод `cdUp()`. Вызов метода `cd("..")` эквивалентен вызову метода `cdUp()`. Оба метода возвращают булево значение, сигнализирующее об успехе операции.

Для конвертирования относительного пути к каталогу в абсолютный можно вызвать метод `makeAbsolute()`.

Для создания каталога нужно вызвать метод `mkdir()`. При успешном проведении операции метод вернет значение `true`, в случае неудачи — `false`.

Если вам потребуется переименовать файл или каталог, то воспользуйтесь методом `rename()`. В этот метод первым параметром нужно передать старый путь, а вторым — новый. Если операция проведена успешно, то метод вернет значение `true`, иначе — `false`.

Удаление пустых каталогов осуществляется методом `rmdir()`, который получает путь, и в случае успеха возвращает значение `true`, а в случае неудачи — `false`.

Просмотр содержимого каталога

При помощи класса `QDir` можно получить содержимое указанного каталога. При этом допускается применять различные фильтры, исключающие из списка не интересующие вас файлы. Для этих целей в классе определены методы `entryList()` и `entryInfoList()`. Первый возвращает список имен элементов (`QStringList`), а второй — информационный список (`QFileInfoList`). Если вам нужно узнать лишь количество элементов, находящихся в каталоге, то просто вызовите метод `count()`.

Программа (листинги 36.1–36.4), окно которой показано на рис. 36.2, осуществляет рекурсивный поиск файлов в каталоге, указанном в текстовом поле **Directory** (Каталог). Нажатие кнопки с растровым изображением откроет диалоговое окно выбора нужного каталога. В текстовом поле **Mask** (Маска) задается фильтр для отображаемых файлов. Например, для отображения исходных файлов на языке C++ нужно задать в поле **Mask** (Маска) `*.cpp` и `*.h`. После нажатия кнопки **Find** (Поиск) выполняется поиск и отображение файлов в соответствии с заданными параметрами. Результаты выводятся в виджете многострочного текстового поля.

В конструкторе класса `FileFinder`, который приведен в листинге 36.1, создаются виджеты однострочного и многострочного текстовых полей (указатели `m_ptxtDir`, `m_ptxtMask` и `m_ptxtResult`). Первый виджет инициализируется абсолютным путем, возвращаемым методом `QDir::absolutePath()`, который, в свою очередь, инициализируется значением каталога исполняемого файла приложения, возвращаемым методом `QCoreApplication::applicationDirPath()`. Второй виджет инициализируется строкой, предназначенной для фильтрации найденных файлов. Для открытия диалогового окна выбора каталога и запуска операции поиска создаются две кнопки (указатели `pcmdFind` и `pcmdDir`), которые соединяют-

ся со слотами `slotFind()` и `slotBrowse()`. В конструкторе создаются два виджета надписей и с помощью метода `setBuddy()` ассоциируются с виджетами однострочных текстовых полей. Созданные виджеты размещаются в виде таблицы при помощи объекта класса `QGridLayout` (см. *главу 6*).

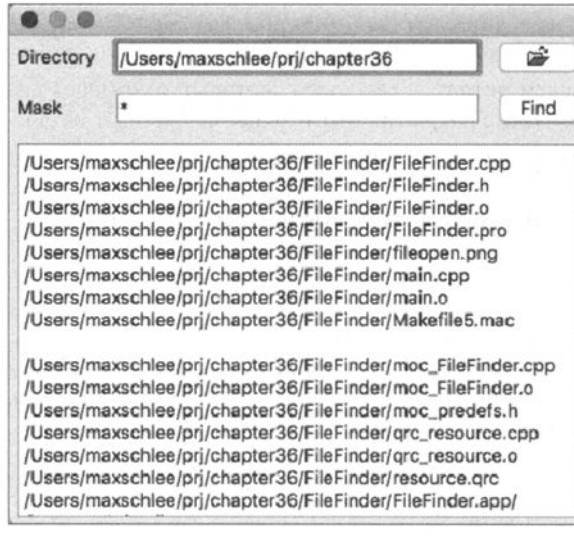


Рис. 36.2. Окно программы отображения файлов каталога по маске

Листинг 36.1. Конструктор класса `FileFinder()` (файл FileFinder.cpp)

```
FileFinder::FileFinder(QWidget* pwgt/*= 0*/) : QWidget(pwgt)
{
    m_ptxtDir    = new QLineEdit(QCoreApplication::applicationDirPath());
    m_ptxtMask   = new QLineEdit("*");
    m_ptxtResult = new QTextEdit;

    QLabel*      plblDir  = new QLabel("&Directory");
    QLabel*      plblMask = new QLabel("&Mask");
    QPushButton* pcmdDir  = new QPushButton(QPixmap(":/fileopen.png"), "");
    QPushButton* pcmdFind = new QPushButton("&Find");

    connect(pcmdDir, SIGNAL(clicked()), SLOT(slotBrowse()));
    connect(pcmdFind, SIGNAL(clicked()), SLOT(slotFind()));

    plblDir->setBuddy(m_ptxtDir);
    plblMask->setBuddy(m_ptxtMask);

    //Layout setup
    QGridLayout* pgrdLayout = new QGridLayout;
    pgrdLayout->setContentsMargins(5, 5, 5, 5);
    pgrdLayout->setSpacing(15);
    pgrdLayout->addWidget(plblDir, 0, 0);
    pgrdLayout->addWidget(plblMask, 1, 0);
    pgrdLayout->addWidget(m_ptxtDir, 0, 1);
```

```
    pgrdLayout->addWidget(m_ptxtMask, 1, 1);
    pgrdLayout->addWidget(pcmdDir, 0, 2);
    pgrdLayout->addWidget(pcmdFind, 1, 2);
    pgrdLayout->addWidget(m_ptxtResult, 2, 0, 1, 3);
    setLayout(pgrdLayout);
}
```

В листинге 36.2 приведен метод `slotBrowse()`, который открывает диалоговое окно для выбора каталога поиска. После закрытия этого окна выбранный каталог записывается в текстовое поле **Directory** (Каталог) методом `setText()`.

Листинг 36.2. Метод `slotBrowse()` (файл FileFinder.cpp)

```
void FileFinder::slotBrowse()
{
    QString str = QFileDialog::getExistingDirectory(0,
                                                     "Select a Directory",
                                                     m_ptxtDir->text()
                                                    );

    if (!str.isEmpty()) {
        m_ptxtDir->setText(str);
    }
}
```

Метод `slotFind()` запускает операцию поиска, для чего передает в метод `start()` выбранный пользователем каталог (листинг 36.3).

Листинг 36.3. Метод `slotFind()` (файл FileFinder.cpp)

```
void FileFinder::slotFind()
{
    start(QDir(m_ptxtDir->text()));
}
```

Метод `start()` для поиска по подкаталогам использует рекурсию (листинг 36.4). Процесс поиска заданных файлов бывает длительным, и так как мы выполняем его из основного потока программы, то это может привести к «замиранию» ее графического интерфейса. Поэтому для подавления такого нежелательного эффекта при каждом вызове метода мы вызываем метод `QApplication::processEvent()` и даем возможность обработаться накопившимся событиям. В методе `start()` переменная `listFiles` получает список файлов текущего каталога, соответствующих маске. Для этого в метод передаются два параметра. Первый представляет собой список шаблонов поиска, которые распространяются на имена и расширения файлов. В нашем случае мы преобразуем строку в список, разделив ее при помощи пробелов вызовом `QString::split()`. Второй параметр (флаг `QDir::Files`) говорит о том, что список должен содержать только файлы. Элементы полученного списка добавляются в виджет многострочного текстового поля с помощью метода `append()`. По завершении работы цикла добавления в метод `entryList()` передается флаг `QDir::Dirs` для получения списка каталогов. Для каждого из элементов списка, кроме `"."` и `".."`, вызывается метод `start()`.

Листинг 36.4. Метод `start()` (файл FileFinder.cpp)

```
void FileFinder::start(const QDir& dir)
{
    QApplication::processEvents();

    QStringList listFiles =
        dir.entryList(m_ptxtMask->text().split(" "), QDir::Files);

    foreach (QString file, listFiles) {
        m_ptxtResult->append(dir.absoluteFilePath(file));
    }

    QStringList listDir = dir.entryList(QDir::Dirs);
    foreach (QString subdir, listDir) {
        if (subdir == "." || subdir == "..") {
            continue;
        }
        start(QDir(dir.absoluteFilePath(subdir)));
    }
}
```

Флаг QDir::NoDotAndDotDot

Использование в листинге 36.4 флага `QDir::NoDotAndDotDot` избавило бы нас от необходимости сравнения имени каталога со строками `"."` и `".."`.

Метод `entryList()` можно использовать и без указания параметров. В этом случае сами критерии фильтрации могут быть заданы при помощи метода `setFilter()`. Дополнительно можно при помощи метода `setSorting()` задать и критерий сортировки списка. В следующем примере мы получаем список скрытых файлов, отсортированных по их величине:

```
dir.setFilter(QDir::Files | QDir::Hidden);
dir.setSorting(QDir::Size);
QStringList content = dir.entryList();
```

Информация о файлах. Класс *QFileInfo*

Задача класса `QFileInfo` состоит в предоставлении информации о свойствах файла: его имени, размере, времени последнего изменения, правах доступа и т. д. Объект этого класса создается передачей в его конструктор пути к файлу или объекта класса `QFile`.

Файл или каталог?

Иногда необходимо убедиться, что исследуемый объект является каталогом, а не файлом, и наоборот. Для этой цели существуют методы `isFile()` и `isDir()`. Если объект является файлом, метод `isFile()` возвращает значение `true`, иначе — `false`. Если объект является каталогом, то метод `isDir()` возвращает значение `true`, иначе — `false`. Кроме этих методов, класс `QFileInfo` содержит метод `isSymLink()`, возвращающий значение `true`, если объект

является символьной ссылкой (термин «symbolic link» используется в UNIX, в ОС Windows принято название *ярлык* (shortcut)).

> ***Символьные ссылки***
>
> Символьные ссылки используются в UNIX для обеспечения связи с файлами или каталогами. Создаются они при помощи команды `ln` с ключом `-s`.

Путь и имя файла

Чтобы получить абсолютный путь к файлу, нужно воспользоваться методом `absoluteFilePath()`, а для получения относительного пути — методом `filePath()`. Для получения имени файла надо вызвать метод `fileName()`, который возвращает имя файла вместе с его расширением. Если нужно только имя файла, то следует вызвать метод `baseName()`. Для получения расширения служит метод `completeSuffix()`.

Информация о дате и времени

Иногда нужно узнать время создания файла, время его последнего изменения или чтения. Для этого класс `QFileInfo` предоставляет методы `created()`, `lastModified()` и `lastRead()` соответственно. Эти методы возвращают объекты класса `QDateTime` (см. *главу 38*), которые можно преобразовать в строку методом `toString()`. Например:

```
// Дата и время создания файла
fileInfo.created().toString();

// Дата и время последнего изменения файла
fileInfo.lastModified().toString();

// Дата и время последнего чтения файла
fileInfo.lastRead().toString();
```

Получение атрибутов файла

Атрибуты файла дают информацию о том, какие операции можно проводить с файлом. Для их получения в классе `QFileInfo` существуют следующие методы:

- `isReadable()` — возвращает значение `true`, если из указанного файла можно читать информацию;
- `isWriteable()` — возвращает значение `true`, если в указанный файл можно записывать информацию;
- `isHidden()` — возвращает значение `true`, если указанный файл является скрытым;
- `isExecutable()` — возвращает значение `true`, если указанный файл можно исполнять. В ОС UNIX это определяется не на основании расширения файла, как принято в DOS и ОС Windows, а из свойств самого файла.

Определение размера файла

Метод `size()` класса `QFileInfo` возвращает размер файла в байтах. Размер файлов редко отображается в байтах, чаще используются специальные буквенные обозначения, сооб-

щающие об его размере. Например, для килобайта — это буква K, для мегабайта — M, для гигабайта — G, а для терабайта — T. Следующая функция позволяет сопровождать буквенными обозначениями размеры, лежащие даже в терабайтном диапазоне (вполне возможно, что через несколько лет это будет обычным размером некоторых типов файлов):

```
QString fileSize(qint64 nSize)
{
    int i = 0;
    for (; nSize > 1023; nSize /= 1024, ++i) {
        if(i >= 4) {
            break;
        }
    }
    return QString().setNum(nSize) + "BKMGT"[i];
}
```

Наблюдение за файлами и каталогами

Поскольку файлы и каталоги используются не только вашей программой, бывает очень полезно получать уведомления в случае их изменений. Например, вы написали файловый браузер и показываете содержимое текущего каталога. Но пользователь может воспользоваться другими программами и скопировать в этот каталог новые файлы, после чего отображаемая информация перестанет отвечать действительности. Не имея специального механизма слежения за изменением текущего каталога, вы не сможете узнать о необходимости обновить его содержимое.

Такой механизм реализует класс `QFileSystemWatcher`. Его использование предусматривает необходимость добавить методом `addPath()` нужный вам путь к файлу либо к каталогу, а когда необходимость в наблюдении за ними отпадет, удалить этот путь при помощи метода `removePath()`. При изменениях файла отправляется сигнал `fileChanged()`, а если изменен каталог — сигнал `directoryChanged()`. Оба сигнала передают в качестве параметра путь, по которому произошли изменения. Уведомления о изменениях осуществляются асинхронно и не блокируют выполнение основного потока.

Для иллюстрации сказанного реализуем программу (листинги 36.5–36.7), которая будет принимать в командной строке каталоги и файлы и отображать их при изменениях (рис. 36.3).

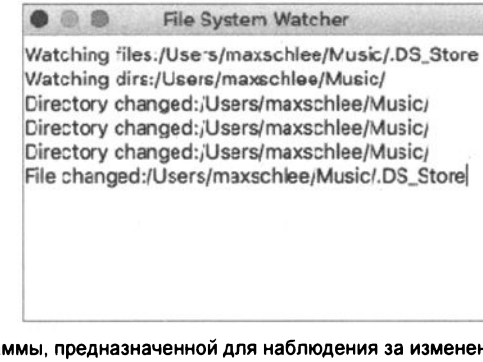


Рис. 36.3. Окно программы, предназначенной для наблюдения за изменениями файлов и каталогов

Прежде всего реализуем класс для отображения (листинг 36.5) — просто унаследуем `QTextEdit` и предоставим слоты для отображения пути измененного каталога `slotDirectory()` и пути и имени измененного файла `slotFileChaged()`.

Листинг 36.5. Определение класса `Viewer` (файл Viewer.h)

```
#pragma once

#include <QTextEdit>

// ============================================================================
class Viewer : public QTextEdit {
Q_OBJECT
public:
    Viewer(QWidget* pwgt = 0);

private slots:
    void slotDirectoryChanged(const QString&);
    void slotFileChanged     (const QString&);
};
```

В конструкторе (листинг 36.6) присваиваем надпись для основного окна приложения. При помощи метода `append()` выполняем отображение информации при вызовах слотов `slotDirectoryChanged()` и `slotFileChanged()`.

Листинг 36.6. Конструктор `Viewer()` (файл Viewer.cpp)

```
#include "Viewer.h"

// ----------------------------------------------------------------------
Viewer::Viewer(QWidget* pwgt /*=0*/) : QTextEdit(pwgt)
{
    setWindowTitle("File System Watcher");
}

// ----------------------------------------------------------------------
void Viewer::slotDirectoryChanged(const QString& str)
{
    append("Directory changed:" + str);
}

// ----------------------------------------------------------------------
void Viewer::slotFileChanged(const QString& str)
{
    append("File changed:" + str);
}
```

В листинге 36.7 мы создаем объекты наблюдателя (`watcher`) и виджета для отображения информации (`viewer`). Из объекта приложения (`app`) запрашиваем список параметров, пере-

данных пользователем в командной строке (метод `arguments()`). Убираем самый первый элемент списка, так как он является именем нашей программы. Полученный список передаем объекту наблюдателя и для этого вызываем метод `QFileSystemWatcher::addPaths()`. Объект наблюдателя автоматически анализирует и распознает, что из списка является каталогом, а что файлом. Для того чтобы увидеть результат этого анализа, мы отдельно отобразим файлы: метод `QFileSystemWatcher::files()` и каталоги: метод `QFileSystemWatcher::directories()`. Далее связываем сигналы объекта наблюдателя `QFileSystemWatcher::directoryChanged()` и `QFileSystemWatcher::fileChanged()` со слотами `slotDirectoryChanged()` и `slotFileChanged()` виджета отображения.

Листинг 36.7. Программа наблюдения за изменениями файлов и каталогов (файл main.cpp)

```
#include <QtWidgets>
#include "Viewer.h"

int main(int argc, char** argv)
{
    QApplication       app(argc, argv);
    QFileSystemWatcher watcher;
    Viewer             viewer;

    QStringList args = app.arguments();
    args.removeFirst();

    watcher.addPaths(args);

    viewer.append("Watching files:" + watcher.files().join(";"));
    viewer.append("Watching dirs:" + watcher.directories().join(";"));
    viewer.show();

    QObject::connect(&watcher, SIGNAL(directoryChanged(const QString&)),
                     &viewer, SLOT(slotDirectoryChanged(const QString&))
                    );
    QObject::connect(&watcher, SIGNAL(fileChanged(const QString&)),
                     &viewer, SLOT(slotFileChanged(const QString&))
                    );

    return app.exec();
}
```

Потоки ввода/вывода

Объекты файлов сами по себе обладают только элементарными методами для чтения и записи информации. Задействование потоков делает запись и считывание файлов более простым и гибким. Для файлов, содержащих текстовую информацию, следует воспользоваться классом `QTextStream`, а для двоичных файлов — классом `QDataStream`.

Используются классы `QTextStream` и `QDataStream` так же, как и стандартный поток ввода/вывода в языке C++ (`iostream`), с той лишь разницей, что они могут работать с объекта-

ми класса QIODevice. Благодаря этому, потоки можно использовать и для своих собственных классов, унаследованных от класса QIODevice. Для записи данных в поток служит оператор <<, а для чтения данных из потока — оператор >>.

Класс *QTextStream*

Класс QTextStream предназначен для чтения текстовых данных. В качестве текстовых данных могут выступать не только объекты, созданные классами, унаследованными от QIODevice, но и переменные типов char, QChar, char*, QString, QByteArray, short, int, long, float и double. Числовые данные, передаваемые в поток, автоматически преобразуются в текст. Можно управлять форматом их преобразования — например, метод QTextStream::setRealNumberPrecision() задает количество знаков после запятой. Этот класс следует использовать для считывания и записи текстовых данных в формате Unicode.

Чтобы считать текстовый файл, необходимо создать объект типа QFile и считать данные методом QTextStream::readLine(). Например:

```
QFile file("file.txt");
if (file.open(QIODevice::ReadOnly)) {
    QTextStream stream(&file);
    QString str;
    while (!stream.atEnd()) {
        str = stream.readLine();
        qDebug() << str;
    }
    if (stream.status() != QTextStream::Ok) {
        qDebug() << "Ошибка чтения файла";
    }
    file.close();
}
```

Методом QTextStream::readAll() можно считать в строку сразу весь текстовый файл. Например:

```
QFile file("myfile.txt");
if (file.open(QIODevice::ReadOnly)) {
    QTextStream stream(&file);
    QString str = stream.readAll();
    file.close();
}
```

Чтобы записать текстовую информацию в файл, необходимо создать объект класса QFile и воспользоваться оператором <<. Перед записью можно провести необходимые преобразования строки. Например:

```
QFile file("file.txt");
QString str = "This is a test";
if (file.open(QIODevice::WriteOnly)) {
    QTextStream stream(&file);
    stream << str.toUpper(); // Запишет THIS IS A TEST
    file.close();
```

```
    if (stream.status() != QTextStream::Ok) {
        qDebug() << "Ошибка записи файла";
    }
}
```

Класс `QTextStream` создавался для записи и чтения только текстовых данных, поэтому двоичные данные при записи будут искажены. Для чтения и записи двоичных данных без искажений следует пользоваться классом `QDataStream` (см. далее).

Класс *QDataStream*

Класс `QDataStream` является гарантом того, что формат, в котором будут записаны данные, останется платформонезависимым, и его можно будет считать и обработать на других платформах. Это делает класс незаменимым для обмена данными по сети с использованием сокетных соединений (см. *главу 39*). Формат данных, используемый `QDataStream`, в процессе разработки версий Qt претерпел множество изменений и продолжает изменяться. По этой причине этот класс содержит информацию о версии, и для того чтобы заставить его использовать формат обмена, соответствующий определенной версии Qt, нужно вызвать метод `setVersion()`, передав ему идентификатор версии. Текущая версия имеет идентификатор `Qt_5_3`.

Класс поддерживает большое количество типов данных, к которым относятся, например: `QByteArray`, `QFont`, `QImage`, `QMap`, `QPixmap`, `QString`, `QValueList` и `Variant`.

Следующий пример записывает в файл объект точки (`QPointF`), задающей позицию растрового изображения, вместе с самим объектом растрового изображения (`QImage`):

```
QFile file("file.bin");
if(file.open(QIODevice::WriteOnly)) {
    QDataStream stream(&file);
    stream.setVersion(QDataStream::Qt_5_3);
    stream << QPointF(30, 30) << QImage("image.png");

    if (stream.status() != QDataStream::Ok) {
        qDebug() << "Ошибка записи";
    }
}
file.close();
```

Для чтения этих данных из файла нужно сделать следующее:

```
QPointF pt;
QImage  img;
QFile   file("file.bin");
if(file.open(QIODevice::ReadOnly)) {
    QDataStream stream(&file);
    stream.setVersion(QDataStream::Qt_5_3);
    stream >> pt >> img;

    if (stream.status() != QDataStream::Ok) {
        qDebug() << "Ошибка чтения файла";
    }
}
file.close();
```

Резюме

В Qt имеется абстрактный класс `QIODevice`, который представляет собой устройство ввода/вывода. Классы, унаследованные от `QIODevice`, предоставляют возможность записи и чтения данных в файл байтами или блоками. Наследуют этот класс классы `QFile` и `QBuffer`:

- класс `QFile` содержит методы, необходимые для работы с файлами, — такие как открытие, закрытие, чтение и запись;
- класс `QBuffer` предназначен для эмуляции файлов в оперативной памяти. Такие файлы хранятся не на диске, а в памяти компьютера, что существенно ускоряет процесс обращения к ним.

Класс `QDir` предоставляет платформонезависимый подход для работы с каталогами. Этот класс содержит методы для создания и уничтожения каталогов, а также для получения главного, текущего и персонального каталога пользователя.

При помощи класса `QFileInfo` можно получить всю необходимую информацию о файлах и каталогах.

Для наблюдения за изменениями файлов и каталогов Qt предоставляет класс `QFileSystemWatcher`.

Qt также предоставляет классы потоков, которые делают удобной работу с данными файлов: `QDataStream` для двоичных данных и `QTextStream` для текстовых. Использование этих потоков очень похоже на использование обычного потока в языке C++. Как и в стандартных потоках, для чтения и записи перегружены операторы << и >>. Так как потоки принимают объекты классов, унаследованных от `QIODevice`, то вместо объекта класса `QFile` данные можно перенаправить, скажем, в объект класса `QBuffer`.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/36-510/** или с помощью следующего QR-кода (рис. 36.4):

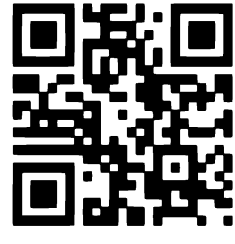


Рис. 36.4. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 37

Дата, время и таймер

Что часто повторяется, то вскоре становится доказанным.
О. Минье

В этой главе рассказано о часто необходимых при программировании объектах для манипулирования датами и временем, а также о реализации повторяющихся событий или задержек при помощи таймеров.

Дата и время

Приложениям часто требуется информация о дате и времени — например, для выдачи отчетной информации или для реализации часов. Qt предоставляет для работы с датой и временем три класса: `QDate`, `QTime` и `QDateTime`, определенных в заголовочных файлах QDate, QTime и QDateTime.

Класс даты *QDate*

Класс `QDate` представляет собой структуру данных для хранения дат и проведения с ними разного рода операций. В конструктор класса `QDate` передаются три целочисленных параметра. Первым передается год, вторым — месяц, а третьим — день. Например, создадим объект, который будет содержать дату 25 октября 2017 года:

```
QDate date(2017, 10, 25);
```

Эти значения с помощью метода `setDate()`можно установить и после создания объекта. Например:

```
QDate date;
date.setDate(2017, 10, 25);
```

Для получения значений года, месяца и дня, установленных в объекте даты, следует воспользоваться следующими методами:

- `year()` — возвращает год в диапазоне от 1752 до 8000;
- `month()` — возвращает целое значение месяца в диапазоне от 1 до 12 (с января по декабрь);
- `day()` — возвращает день месяца в диапазоне от 1 до 31.

С помощью метода `daysInMonth()` можно узнать количество дней в месяце, а с помощью метода `daysInYear()` — количество дней в году.

Для получения дня недели следует вызвать метод `dayOfWeek()`. Для получения порядкового номера дня в году служит метод `dayOfYear()`. Можно также узнать номер недели, для чего нужно вызвать метод `weekNumber()`.

Метод `toString()` позволяет получить текстовое представление даты, а в качестве параметра можно передать одно из значений, указанных в табл. 37.1.

Таблица 37.1. *Некоторые перечисления* `DateFormat` *пространства имен Qt*

Константа	Описание
`TextDate`	Специальный формат Qt (определен по умолчанию)
`ISODate`	Расширенный формат ISO 8601 (YYYY-MM-DD)
`SystemLocaleShortDate`	Формат, зависящий от установленного в операционной системе языка страны, в короткой форме
`SystemLocaleLongDate`	Формат, зависящий от установленного в операционной системе языка страны, в длинной форме

Если в таблице не приведен нужный вам формат, то можно определить свой собственный, передав в метод `toString()` строку-шаблон, его описывающую. Например:

```
QDate date(2017, 7, 3);
QString str;
str = date.toString("d.M.yy");         //str = "3.7.17"
str = date.toString("dd/MM/yy");       //str = "03/07/17"
str = date.toString("yyyy.MMM.ddd");   //str = "2017.июл.Пт"
str = date.toString("yyyy.MMMM.dddd");//str = "2017.Июль.пятница"
```

При помощи метода `addDays()` можно получить измененную дату, добавив или отняв от нее указанное количество дней. Действия методов `addMonths()` и `addYears()` аналогичны, но разница в том, что они оперируют месяцами и годами. Например:

```
QDate date(2017, 1, 3);
QDate date2 = date.addDays(-7);
QString str = date2.toString("dd/MM/yy"); //str = "27/06/17"
```

Класс `QDate` предоставляет статический метод `fromString()`, позволяющий проводить обратное преобразование из строкового типа к типу `QDate`. Для этого первым параметром метода нужно передать строку с датой, а вторым — можно передать формат в виде флага из табл. 37.1 или строки.

Одна из самых частых операций — получение текущей даты. Для этого нужно вызвать статический метод `currentDate()`, возвращающий объект класса `QDate`.

При помощи метода `daysTo()` можно узнать разницу в днях между двумя датами. Следующий пример определяет количество дней от текущей даты до Нового года:

```
QDate dateToday = QDate::currentDate();
QDate dateNewYear(dateToday.year(), 12, 31);
qDebug() << "Осталось "
         << dateToday.daysTo(dateNewYear)
         << " дней до Нового года";
```

Объекты дат можно сравнивать друг с другом, для чего в классе `QDate` определены операторы ==, !=, <, <=, > и >=. Например:

```
QDate date1(2017, 1, 3);
QDate date2(2017, 1, 5);
bool b = (date1 == date2); //b = false
```

Класс времени *QTime*

Контроль над временем — очень важная задача, с его помощью можно вычислять задержки в работе программы, отображать на экране текущее время, проверять время создания файлов и т. п.

Для работы со временем библиотека Qt предоставляет класс `QTime`. Как и в случае с объектами даты, с объектами времени можно проводить операции сравнения ==, !=, <, <=, > или >=. Объекты времени способны хранить время с точностью до миллисекунд. В конструктор класса `QTime` передаются четыре параметра. Первый параметр задает часы, второй — минуты, третий — секунды, а четвертый — миллисекунды. Третий и четвертый параметры можно опустить, по умолчанию они равны нулю. Например:

```
QTime time(20, 4);
```

Параметры можно устанавливать и после создания объекта времени посредством метода `setHMS()`. Например:

```
QTime time;
time.setHMS (20, 4, 23, 3);
```

Для получения значений часов, минут, секунд и миллисекунд, установленных в объекте времени, в классе `QTime` определены следующие методы:

- `hour()` — значение часа в диапазоне от 0 до 23;
- `minute()` — минуты в диапазоне от 0 до 59;
- `second()` — секунды в диапазоне от 0 до 59;
- `msec()` — миллисекунды в диапазоне от 0 до 999.

Класс `QTime` предоставляет метод `toString()` для передачи данных объекта времени в виде строки. В этот метод в качестве параметра можно передать одно из значений, указанных в табл. 37.1, или задать свой собственный формат. Например:

```
QTime time(20, 4, 23, 3);
QString str;
str = time.toString("hh:mm:ss.zzz"); //str = "20:04:23.003"
str = time.toString("h:m:s ap");     //str = "8:4:23 pm"
```

При помощи статического метода `fromString()` можно выполнить преобразование из строкового типа в тип `QTime`. Для этого первым параметром метода передается строка, представляющая время. Вторым параметром можно передать одно из значений форматов, приведенных в табл. 37.1, или строку с ожидаемым форматом.

Изменить объект времени можно, добавив или отняв значения секунд (или миллисекунд) от существующего объекта. Эти значения передаются в методы `addSecs()` и `addMSecs()`. Для получения текущего времени в классе `QTime` имеется статический метод `currentTime()`.

При помощи метода `start()` можно начать отсчет времени, а для того чтобы узнать, сколько времени прошло с момента начала отсчета, следует вызвать метод `elapsed()`. Например, на

базе этих методов можно сделать небольшой *профайлер*, то есть инструмент, позволяющий оценить эффективность работы кода и выявить его «узкие» места, нуждающиеся в оптимизации. Следующий пример вычисляет время работы функции `test()`:

```
QTime time;
time.start();
test();
qDebug() << "Время работы функции test() равно "
         << time.elapsed()
         << " миллисекунд"
         << endl;
```

Недостаток класса `QTime` состоит в ограничении 24-часовым интервалом, по истечении которого отсчет начинает осуществляться с нуля. Для решения этой проблемы можно воспользоваться классом `QDateTime`.

Класс даты и времени *QDateTime*

Объекты класса `QDateTime` содержат в себе дату и время. Вызовом метода `date()` можно получить объект даты (`QDate`), а вызов `time()` возвращает объект времени (`QTime`). Этот класс также содержит методы `toString()` для представления данных в виде строки. Для этого можно воспользоваться одним из форматов, указанных в табл. 37.1, или собственной строкой с форматом.

Таймер

В программах часто возникает потребность в периодическом повторении определенных действий через заданные промежутки времени. Конечно, в некоторых случаях для задания промежутка времени вызова функции можно воспользоваться и объектом класса `QTime` и сделать примерно следующее:

```
QTime time;
time.start();
for(;time.elapsed() < 1000;) {
}
function();
```

Но такой подход обладает огромным недостатком. Исполнение цикла на секунду приостанавливает выполнение всей программы. Из-за этого события интерфейса пользователя перестанут обрабатываться, и, скажем, если одно из окон перекроет окно приложения, то оно все это время перерисовываться не будет, то есть приложение как бы «замрет».

В подобных ситуациях можно, вызывая метод `processEvents()` класса приложения `QApplication`, приостанавливать исполнение цикла, чтобы программа получала возможность обработки поступивших событий. Например:

```
QTime time;
time.start();
for(;time.elapsed() < 1000;) {
    qApp->processEvents();
}
```

Но и такой подход тоже обладает недостатком — он не асинхронен, то есть наша программа будет обрабатывать поступающие события, но не сможет исполняться дальше, пока цикл не завершится до конца.

Решение этой проблемы представляет *таймер*. События таймера происходят асинхронно и не прерывают обработку других событий, выполняемых в том же потоке. Таймер — это гарант, обеспечивающий передачу управления программе. Однако длительная обработка событий влечет за собой задержки выполнения события таймера, то есть таймер ждет своего времени, как и остальные события. Период между событиями таймера носит название *интервал запуска* (firing interval). Таймер переходит в сигнальное состояние по истечении интервала запуска, который указывается в миллисекундах. Точность интервала запуска ограничивается, к сожалению, точностью системных часов, а это значит, что на таймер нельзя полагаться как на секундомер, поскольку точность его лежит в пределах 1 мс. Следовательно, при написании программы имитации часов будет нелишним после каждого сообщения таймера проверять текущее время (см. далее листинг 37.5). Поскольку временной интервал, задаваемый в таймере, представляет собой целое число, то самый большой интервал запуска, который можно установить, — 24 дня. Эту проблему можно решить введением для таймера дополнительного счетчика.

Существует много областей применения таймера. Например, его используют для автоматического сохранения файлов в текстовом редакторе или в качестве альтернативы многопоточности (см. *главу 38*) — разбив программу на части, каждая из которых начнет выполняться при наступлении события таймера. Также таймер используется для отображения информации о состоянии данных, изменяющихся с течением времени. Таймер незаменим для устранения различий, связанных с мощностью и возможностями разных компьютеров, то есть для исполнения программ в режиме реального времени.

События таймера можно использовать и в многопоточном программировании (см. *главу 38*) в каждом отдельно взятом потоке, который имеет *цикл сообщений* (event loop). Для запуска цикла сообщений в потоке нужно вызвать метод `QThread::exec()`.

Событие таймера

Каждый класс, унаследованный от `QObject`, содержит свои собственные встроенные таймеры. Вызов метода `QObject::startTimer()` запускает таймер. В качестве параметра ему передается интервал запуска в миллисекундах. Метод `startTimer()` возвращает идентификатор, необходимый для распознавания таймеров, используемых в объекте. По истечении установленного интервала запуска генерируется событие `QTimerEvent`, которое передается в метод `timerEvent()`. Вызвав метод `QTimerEvent::timerId()` объекта события `QTimerEvent`, можно узнать идентификатор таймера, инициировавшего это событие. Идентификатор можно использовать для уничтожения таймера, передав его в метод `QObject::killTimer()`. В программе (листинги 37.1 и 37.2), окно которой показано на рис. 37.1, отображается надпись, появляющаяся и исчезающая через заданные промежутки времени.

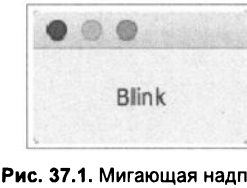


Рис. 37.1. Мигающая надпись

В функции `main()` (листинг 37.1) создается виджет класса `BlinkLabel`, в конструктор которого первым параметром передается отображаемый текст в формате HTML.

Листинг 37.1. Программа мигающей надписи — функция `main()` (файл main.cpp)

```
int main (int argc, char** argv)
{
    QApplication app (argc, argv);
    BlinkLabel   lbl("<FONT COLOR = RED><CENTER>Blink</CENTER></FONT>");

    lbl.show();

    return app.exec();
}
```

Класс `BlinkLabel`, приведенный в листинге 37.2, содержит атрибут булевого типа `m_bBlink`, управляющий отображением надписи, и атрибут `m_strText`, содержащий текст надписи. В конструктор класса `BlinkLabel` передается интервал мигания `nInterval`. По умолчанию он равен 200 мс. Вызов метода `startTimer()` запускает таймер со значением переданного интервала запуска. По истечении этого интервала создается событие `QTimerEvent`, которое передается в метод `timerEvent()`, где значение атрибута `m_bBlink` меняется на противоположное. В соответствии с установленным значением метод `setText()` выполняет одно из дей-ствий:

- `false` — вся область надписи очищается установкой пустой строки;
- `true` — текст надписи устанавливается заново.

Листинг 37.2. Определение класса `BlinkLabel` (файл main.cpp)

```
class BlinkLabel : public QLabel {
private:
    bool    m_bBlink;
    QString m_strText;

protected:
    virtual void timerEvent(QTimerEvent*)
    {
        m_bBlink = !m_bBlink;
        setText(m_bBlink ? m_strText : "");
    }

public:
    BlinkLabel(const QString& strText,
                     int      nInterval = 200,
                     QWidget* pwgt      = 0
              )
        : QLabel(strText, pwgt)
        , m_bBlink(true)
        , m_strText(strText)
```

```
    {
        startTimer(nInterval);
    }
};
```

Класс *QTimer*

Использование объекта класса `QTimer` гораздо проще, чем использование события таймера, определенного в классе `QObject`. К недостаткам работы с событием таймера относится необходимость наследования одного из классов, наследующих класс `QObject`. Затем в унаследованном классе требуется реализовать метод, принимающий объекты события таймера. А если в объекте создается более одного таймера, то возникает необходимость различать таймеры, чтобы узнать, который из них явился инициатором события.

Для ликвидации этих неудобств Qt предоставляет класс таймера `QTimer`, являющийся непосредственным наследником класса `QObject`. Чтобы запустить таймер, нужно создать объект класса `QTimer`, а затем вызвать метод `start()`. В параметре метода передается значение интервала запуска в миллисекундах.

Класс `QTimer` также содержит статический метод `singleShot()` для *одноразового режима отработки* таймера (режима `singleshot`). С его помощью можно запустить одноразовый таймер — без создания объекта класса `QTimer`. Первый параметр метода задает интервал запуска, а второй является указателем на объект, с которым должно осуществляться соединение. Слот для соединения передается в третьем параметре. Этим можно воспользоваться, например, для прекращения работы демоверсии программы через 5 минут после ее запуска (листинг 37.3).

Листинг 37.3. Завершение работы программы после пяти минут работы

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    MyProgram    myProgram;

    QTimer::singleShot(5 * 60 * 1000, &app, SLOT(quit()));

    myProgram.show();

    return app.exec();
}
```

По истечении интервала запуска таймера отправляется сигнал `timeout()`, который нужно соединить со слотом, выполняющим нужные действия. При помощи метода `setInterval()` можно изменить интервал запуска таймера. Если таймер был активен, то он будет остановлен и запущен с новым интервалом, и ему будет присвоен новый идентификационный номер. При помощи метода `isActive()` можно проверить, находится ли таймер в активном состоянии. Вызовом метода `stop()` можно остановить таймер.

Вот еще один пример того, как можно воспользоваться одноразовым таймером. Бывают случаи, когда нужно заблокировать выполнение основного потока программы на какое-то время, — для того чтобы, например, дождаться доступности ресурса, предоставляемого другой программой или сервером, без которого программа не может продолжать свою ра-

боту дальше. Реализуем для этого функцию `delay()`, которая будет принимать одно значение целого типа, означающее время продолжительности блокировки в миллисекундах (листинг 37.4).

Листинг 37.4. Функция задержки

```
void delay(int n)
{
    QEventLoop loop;
    QTimer::singleShot(n, &loop, SLOT(quit()));
    loop.exec();
}
```

В листинге 37.4 мы создаем блокирующую функцию при помощи объекта класса цикла событий `QEventLoop`, сам объект создан локально, и он автоматически будет разрушен в конце блокировки. Вызов метода `exec()` блокирует выполнение программы. Соединение, выполненное в методе `singleShot()`, прервет блокировку по истечении времени `n` вызовом слота `quit()` из объекта цикла событий (объект `loop`).

Программа (листинг 37.5), окно которой изображено на рис. 37.2, реализует часы, отображающие дату и время. Отображаемая информация актуализируется в соответствии с установленным полусекундным интервалом запуска таймера.



Рис. 37.2. Электронные часы

Здесь в конструкторе класса `Clock` создается объект таймера (указатель `ptimer`). Его сигнал `timeout()` соединяется со слотом `slotUpdateDateTime()`, ответственным за обновление отображаемой информации. Вызов метода `start()` запускает таймер. В него передается интервал запуска. Слот `slotUpdateDateTime()` получает актуальную дату и время с помощью метода `currentDateTime()`. Затем он, используя параметр локализации `Qt::SystemLocaleLongDate` (см. табл. 37.1), преобразует дату и время к строковому типу и передает в метод `setText()` для отображения.

Листинг 37.5. Программа реализации электронных часов (файл Clock.h)

```
#pragma once

#include <QtWidgets>

// ============================================================================
class Clock : public QLabel {
    Q_OBJECT

public:
    Clock(QWidget* pwgt = 0) : QLabel(pwgt)
    {
        QTimer* ptimer = new QTimer(this);
        connect(ptimer, SIGNAL(timeout()), SLOT(slotUpdateDateTime()));
```

```
        ptimer->start(500);
        slotUpdateDateTime();
    }
public slots:
    void slotUpdateDateTime()
    {
        QString str =
            QDateTime::currentDateTime().toString(Qt::SystemLocaleLongDate);
        setText("<H2><CENTER>" + str + "</CENTER></H2>");
    }
};
```

Класс *QBasicTimer*

В качестве еще одной альтернативы для таймера в Qt предусмотрено его минималистское решение — класс `QBasicTimer`, предоставляющий только четыре метода: `isActive()`, `start()`, `stop()` и `timerId()`, которые по своей функциональности аналогичны методам класса `QTimer`. Исключение составляет только метод `start()`. Помимо первого параметра, задающего интервал, в этот метод вторым параметром передается указатель на объект `QObject`, который будет получать события таймера. Таким образом, вам нужно будет реализовать в классе, унаследованном от класса `QObject`, метод обработки события таймера `QObject::timerEvent()`.

Резюме

Классы `QDate`, `QTime` и `QDateTime` предназначены для хранения дат и времени, а также для проведения с ними различных операций. Чаще всего требуется получение текущих даты и времени. Эти классы предоставляют методы для преобразования даты и времени в строку определенного формата. Существуют и методы для проведения обратного преобразования — из строки.

Таймер уведомляет приложение об истечении заданного промежутка времени. События таймера относятся к разряду внешних прерываний. *Внешние прерывания* — это прерывания, вызываемые асинхронными событиями, например устройствами ввода/вывода или самим устройством таймера. Интервалы запуска таймера устанавливаются в миллисекундах. Недостаток состоит в том, что если программа занята интенсивными вычислениями, то события таймера могут быть обработаны лишь по окончании процесса вычисления. При выходе из приложения таймеры автоматически уничтожаются.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/37-510/** или с помощью следующего QR-кода (рис. 37.3):



Рис. 37.3. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 38

Процессы и потоки

...каждый крупный результат является конечным продуктом длинной последовательности маленьких действий.
Кристофер Александер

В современных компьютерах одновременно выполняется множество программ. Одни из них запускаются при старте системы, другие — пользователем, а третьи — другими программами. Многие приложения также сами по себе часто выполняют несколько действий одновременно. Например, текстовый редактор проверяет орфографию, сохраняет резервные копии, и все это — не прекращая реагировать на действия пользователя. Для организации подобного поведения служат *потоки*. Как их можно реализовать в Qt, рассказано в этой главе.

Процессы

В том случае, когда пользователь или программа выполняют запуск другой программы, операционная система всегда создает новый процесс. *Процесс* — это экземпляр программы, загруженной для выполнения в память компьютера.

По своей сути, процессы — это независимые друг от друга программы, обладающие своими собственными данными. Коротко процесс можно охарактеризовать как совокупность кода, данных и ресурсов, необходимых для его работы. Под ресурсами подразумеваются объекты, запрашиваемые и используемые процессами в период их работы. Любая прикладная программа, запущенная на вашем компьютере, представляет собой не что иное, как процесс.

Создание процесса может оказаться полезным для использования функциональных возможностей программ, не имеющих графического интерфейса и работающих с командной строкой. Особенно это имеет смысл при запуске команд или программ, время работы которых не продолжительно.

Запускать другие программы из текущей Qt-программы довольно просто. Процессы можно создавать с помощью класса `QProcess`, который определен в заголовочном файле QProcess. Благодаря тому, что этот класс унаследован от класса `IODevice` (см. *главу 36*), объекты этого класса позволяют считывать информацию, выводимую запущенными процессами, и даже реагировать на их запросы ввода данных. Кроме того, класс `QProcess` содержит методы для манипулирования системными переменными процесса. Работа с объектами этого класса осуществляется в асинхронном режиме, что позволяет сохранять работоспособность графи-

ческого интерфейса программы в моменты, когда запущенные процессы находятся в работе. При появлении данных или других событий объекты класса `QProcess` отправляют сигналы. Например, при возникновении ошибок объект процесса вышлет сигнал `errorOccurred()` с кодом этой ошибки.

Для создания процесса его нужно запустить. Запуск процесса выполняется методом `start()`, в который необходимо передать имя команды и список ее аргументов, либо все вместе: команду и аргументы одной строкой. Как только процесс будет запущен, отправляется сигнал `started()`, а после завершения его работы — сигнал `finished()`. Сигнал `finished()` сообщает код и статус завершения работы процесса. Для получения статуса выхода можно вызвать метод `exitStatus()`, который возвращает только два значения: `NormalExit` (нормальное завершение) и `CrashExit` (аварийное завершение).

Для чтения данных запущенного процесса класс `QProcess` предоставляет два разделенных канала: канал стандартного вывода (`stdout`) и канал ошибок (`stderr`). Эти каналы можно переключать с помощью метода `setReadChannel()`. Если процесс готов предоставить данные по установленному текущему каналу, то отправляется сигнал `readyRead()`. Отправляются и сигналы для каждого канала в отдельности: `readyReadStandardOutput()` и `readyReadStandardError()`

Считывать и записывать данные в процесс можно с помощью методов класса `QIODevice`: `write()`, `read()`, `readLine()` и `getChar()`. Для чтения также можно воспользоваться методами, привязанными к конкретным каналам: `readAllStandardOutput()` и `readAllStandardError()`. Эти методы считывают данные в объекты класса `QByteArray`.

Приложение (листинг 38.1), окно которого изображено на рис. 38.1, иллюстрирует применение некоторых методов класса `QProcess`. В текстовом поле **Command** (Команда) может быть введена любая команда, понимаемая в операционной системе. Если запущенная команда или программа осуществляют вывод на консоль, то отображение будет выполняться в виджете многострочного тестового поля.

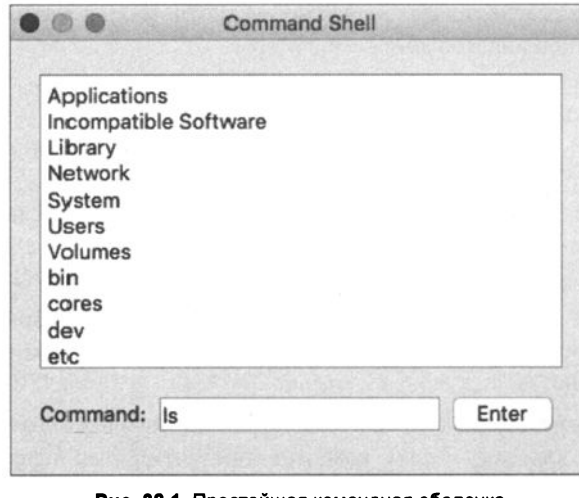


Рис. 38.1. Простейшая командная оболочка

Здесь в конструкторе класса `Shell` создается объект класса `QProcess`. Его сигнал `readyReadStandardOutput()` соединяется со слотом `slotDataOnStdout()`, в котором вызывается метод `readAllStandardOutput()` для считывания всего содержимого стандартного по-

тока. После считывания эти данные добавляются в виджет многострочного текстового поля m_ptxtDisplay вызовом метода append().

Слот slotReturnPressed() соединен с сигналом кнопки clicked() (указатель pcmd) и с сигналом однострочного текстового поля returnPressed() (указатель m_txtCommand). Некоторые команды ОС Windows, например dir, не являются отдельными программами, поэтому они должны быть выполнены посредством командного интерпретатора cmd. Поэтому для ОС Windows в командную строку сначала добавляется строка cmd /C. Во всех остальных операционных системах введенная в однострочном текстовом поле строка передается как есть, без дополнений. Для запуска процесса вызывается метод start().

> **МЕТОД STARTDETACHED()**
>
> Если вам нужно запустить программу и после этого завершить свою, а запущенная программа должна оставаться работать дальше, то используйте метод startDetached().

Метод start() исполняется асинхронно, а если логике работы вашей программы требуется блокирующий подход, то используйте методы waitForStarted() и waitForFinished(). Первый метод блокирует выполнение потока программы до тех пор, пока программа не будет запущена, а второй, пока программа не будет завершена. Например:

```
QProcess proc;
proc.start("cmd /k " + strProgram);
proc.waitForStarted();
```

Листинг 38.1. Простейшая командная оболочка (файл shell.h)

```
class Shell : public QWidget {
Q_OBJECT
private:
    QProcess*  m_process;
    QLineEdit* m_ptxtCommand;
    QTextEdit* m_ptxtDisplay;

public:
    // ------------------------------------------------------------------------
    Shell(QWidget* pwgt = 0) : QWidget(pwgt)
    {
        m_process     = new QProcess(this);
        m_ptxtDisplay = new QTextEdit;

        QLabel* plbl = new QLabel("&Command:");

#ifdef Q_OS_WIN
        QString strCommand = "dir";
#else
        QString strCommand = "ls";
#endif

        m_ptxtCommand = new QLineEdit(strCommand);
        plbl->setBuddy(m_ptxtCommand);
```

```
        QPushButton* pcmd = new QPushButton("&Enter");

        connect(m_process,
                SIGNAL(readyReadStandardOutput()),
                SLOT(slotDataOnStdout())
               );
        connect(m_ptxtCommand,
                SIGNAL(returnPressed()),
                SLOT(slotReturnPressed())
               );
        connect(pcmd, SIGNAL(clicked()), SLOT(slotReturnPressed()));

        //Layout setup
        QHBoxLayout* phbxLayout = new QHBoxLayout;
        phbxLayout->addWidget(plbl);
        phbxLayout->addWidget(m_ptxtCommand);
        phbxLayout->addWidget(pcmd);

        QVBoxLayout* pvbxLayout = new QVBoxLayout;
        pvbxLayout->addWidget(m_ptxtDisplay);
        pvbxLayout->addLayout(phbxLayout);
        setLayout(pvbxLayout);
    }

public slots:
    // -----------------------------------------------------------------------
    void slotDataOnStdout()
    {
        m_ptxtDisplay->append(m_process->readAllStandardOutput());
    }

    // -----------------------------------------------------------------------
    void slotReturnPressed()
    {
        QString strCommand = "";
#ifdef Q_OS_WIN
        strCommand = "cmd /C ";
#endif
        strCommand += m_ptxtCommand->text();
        m_process->start(strCommand);
    }
}
```

Потоки

Потоки становятся все более популярным средством программирования. Для управления потоком Qt предоставляет класс `QThread`. Но давайте сначала разберемся, что же они собой представляют.

Поток — это независимая задача, которая выполняется внутри процесса и разделяет с ним общее адресное пространство, код и глобальные данные.

Процесс сам по себе не исполняется, поэтому для выполнения программного кода он должен иметь хотя бы один поток (далее — *основной поток*). Конечно, можно создавать и более одного потока. Вновь созданные потоки начинают выполняться сразу же, параллельно с основным потоком, при этом их количество может изменяться — одни создаются, другие завершаются. Завершение основного потока приводит к завершению процесса, независимо от того, существуют другие потоки или нет. Создание нескольких потоков в процессе получило название *многопоточность*.

Многопоточность требуется для выполнения действий в фоновом режиме, параллельно с действиями основной программы, и позволяет разбить выполнение задач на параллельные потоки, которые могут быть абсолютно независимы друг от друга. А если приложение выполняется на компьютере с несколькими процессорами, то разделение на потоки может значительно ускорить работу всей программы, так как каждый из процессоров получит отдельный поток для выполнения. Практически все компьютеры сейчас оснащены многоядерными процессорами, что делает многопоточное программирование еще более популярным.

Приложения, имеющие один поток, могут выполнять только одну определенную операцию в каждый момент времени, а все остальные операции ждут ее окончания. Например, такие операции, как вывод на печать, считывание большого файла, ожидание ответа на посланный запрос или выполнение сложных математических вычислений, могут привести к блокировке или «зависанию» всей программы. Используя многопоточность, можно решить эту проблему, запустив такие операции в отдельно созданных потоках. Тем самым при «зависании» одного из потоков функционирование основной программы нарушено не будет.

Среди разработчиков встречается разное отношение к многопоточному программированию. Некоторые стараются сделать все свои программы многопоточными, а других многопоточность пугает. Важно учитывать и то обстоятельство, что использование потоков существенно усложняет процесс разработки приложения, а также его отладку. Но при правильном и обоснованном применении многопоточности можно существенно увеличить скорость работы приложения. Неправильное же ее применение, наоборот, может привести даже к снижению скорости его работы. Поэтому использование потоков должно быть обоснованным. А это значит, что если вы не можете сформулировать причину, по которой следует сделать приложение многопоточным, то лучше отказаться от этой идеи. Если вы сомневаетесь, то постарайтесь на начальных стадиях создавать два прототипа для тестирования: один многопоточный, а другой — нет. Тем самым, прежде чем тратить время на разработку программы, можно определить, является ли многопоточность решением поставленной задачи. В том случае, если достижения того же результата можно добиться без использования многопоточности, стоит отдать предпочтение этому варианту. В качестве заменителя многопоточности можно использовать только что рассмотренный класс `QProcess`, можно также обратиться к методу `QCoreApplication::processEvents()`, описанному в *главе 14*.

Перед тем как начать создание программы с использованием многопоточности, очень важно разобраться и понять фундаментальные принципы программирования потоков и подходы к нему, применяемые в Qt. Именно этому и посвящен следующий далее материал.

Первое, что нужно сразу взять на заметку, это то, что ни в коем случае нельзя делать в потоках (за исключением основного потока приложения):

- нельзя создавать объекты от класса `QWidget` и унаследованных от него классов, а также вызывать их методы;

- нельзя создавать объекты класса `QPixmap`. Вместо этого используйте класс `QImage`;
- нельзя вызывать `QCoreApplication::exec()` и `QApplication::exec()`;
- нельзя блокировать основной поток приложения — иначе замрет его пользовательский интерфейс. О блокировке пойдет речь далее, в *разд. «Синхронизация»*.

Так с чего же все-таки начинается многопоточное программирование? С наследования класса `QThread` и перезаписи в нем чисто виртуального метода `run()`, в котором должен быть реализован код, который будет выполняться в потоке. Например:

```
class MyThread : public QThread {
public:
    void run()
    {
        // Код, выполняемый в потоке
    }
}
```

Второй шаг заключается в создании объекта класса управления потоком и вызове его метода `start()`, который запустит, в свою очередь, реализованный нами метод `run()`.

Например:

```
MyThread thread;
thread.start();
```

Приоритеты

У каждого потока есть приоритет, указывающий процессору, как должно протекать выполнение потока по отношению к другим. Приоритеты разделяются по группам:

- *в первую группу* входят четыре приоритета, применяемые наиболее часто. Их значимость распределяется по возрастанию: `IdlePriority`, `LowestPriority`, `LowPriority`, `NormalPriority`. Они подходят для решения задач, которым процессор требуется только время от времени, — например, для фоновой печати или для каких-нибудь несрочных действий;
- *во вторую группу* входят два приоритета: `HighPriority` и `HighestPriority`. Пользуйтесь этими приоритетами с особой осторожностью. Обычно такие потоки большую часть времени ожидают каких-либо событий;
- *в третью группу* входит лишь один приоритет — `TimeCriticalPriority`. Потоки с ним нужно создавать в случае крайней необходимости. Этот приоритет нужен для программ, напрямую общающихся с аппаратурой или выполняющих операции, которые ни в коем случае не должны прерваться.

Для того чтобы запустить поток с нужным приоритетом, необходимо передать одно из приведенных ранее значений в метод `start()`. Например:

```
MyThread thread;
thread.start(QThread::IdlePriority);
```

А для того чтобы узнать, с каким приоритетом запущен поток, нужно вызвать метод `priority()`. Приоритет можно предварительно установить при помощи метода `setPriority()`.

Внимание: Linux vs Windows

В Linux ядро системы запустит поток даже в том случае, если в нем уже запущены более сотни потоков с максимальным приоритетом. Ядро системы не потеряет работоспособность, а вы сможете запускать и другие потоки. Но в Windows все обстоит иначе: если вы запустите поток с самым высоким приоритетом, то поток с самым низким приоритетом, возможно, будет просто проигнорирован, и это может привести к плачевному результату в тех случаях, когда используются потоки с самым низким приоритетом для проведения каких-либо вспомогательных операций, например загрузки данных с диска.

Обмен сообщениями

Один из важнейших вопросов при многопоточном программировании — это обмен сообщениями. Действительно, если вы, например, в одном потоке создаете растровое изображение и хотите переслать его объекту другого потока, то каким образом можно это сделать?

Каждый поток может иметь свой собственный цикл событий (рис. 38.2). Благодаря этому можно осуществлять связь между объектами. Такая связь может выполняться двумя способами: при помощи соединения сигналов и слотов или за счет обмена событиями.

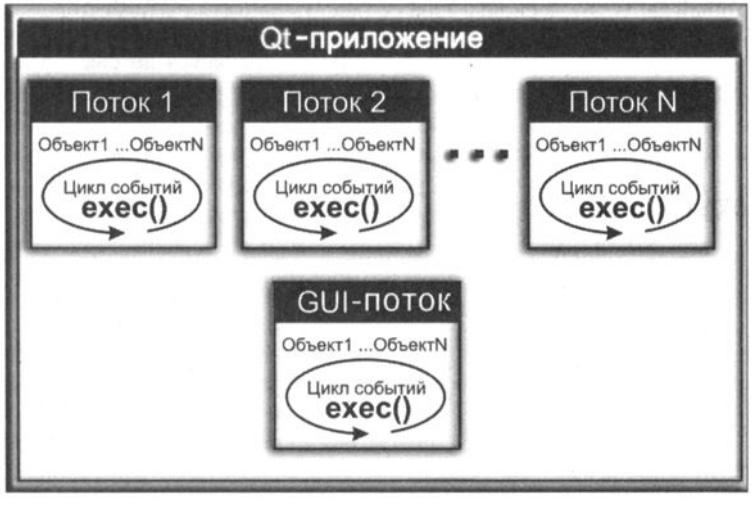


Рис. 38.2. Потоки с объектами и собственными циклами обработки событий

Преобразование сигнально-слотового соединения в событие

Если сигнально-слотовое соединение осуществляется между объектами разных потоков, то внутри оно преобразуется в событие.

Использование в классах управления потоком собственных циклов обработки событий позволяет снять ограничение, связанное с применением классов, способных работать только в одном цикле событий, и параллельно использовать необходимое количество объектов этих классов. К таким классам относятся класс `QTimer` (см. *главу 37*) и сетевые классы (см. *главу 39*).

Для того чтобы запустить собственный цикл обработки событий в потоке, нужно вызвать метод `exec()`. Этот метод может быть вызван неявно. Цикл обработки событий потока можно завершать посредством слота `quit()` или метода `exit()`. Это очень похоже на то, как мы обычно поступаем с объектом приложения в функции `main()`.

Класс QObject реализован так, что он обладает близостью к потокам. Каждый объект, созданный от унаследованного класса QObject, располагает ссылкой на поток, в котором был создан. Эту ссылку можно получить вызовом метода QObject::thread(). Потоки осведомляют свои объекты. Благодаря этому каждый объект знает, к какому потоку он принадлежит. Для того чтобы определить в каком потоке исполняется код, можно вызвать статический метод QThread::currentThreadId() и получить идентификационный номер потока. Можно так же получить и указатель на объект потока — для этого нужно вызвать статический метод QThread::currentThreadId().

Обработка событий осуществляется из контекста принадлежности объекта к потоку, то есть обработка его событий будет выполняться в том потоке, которому объект принадлежит. Объекты можно перемещать из одного потока в другой с помощью метода QObject::moveToThread().

Сигнально-слотовые соединения

Итак, мы можем взять сигнал объекта одного потока и соединить его со слотом объекта другого потока. Из *главы 2* мы уже знаем, что соединение при помощи метода connect() предоставляет дополнительный параметр режима обработки. Этот параметр по умолчанию имеет значение Qt::AutoConnection, что соответствует автоматическому режиму. Как только происходит отправка сигнала, Qt проверяет, осуществляется ли связь в одном и том же потоке или в разных. Если это один и тот же поток, то отправка сигнала приведет к прямому вызову метода. В том случае, если это разные потоки, сигнал будет преобразован в событие и доставлен нужному объекту. Реализация сигналов и слотов в Qt содержит механизм, обеспечивающий надежность при работе в потоках, а это означает, что вы можете отправлять сигналы и получать их, не заботясь о блокировке ресурсов. Вы также можете перемещать объект, созданный в одном потоке, в другой. А если вдруг обнаружится, что отправляющий объект находится в одном потоке с принимающим, то отправка сигнала будет сведена к прямой обработке соединения.

Во всем этом есть нюанс, связанный с сигнально-слотовыми соединениями и классом QThread. Очень важно понять и осознать тот факт, что класс QThread не является классом потока, — этот класс представляет собой только механизм для управления потоком, но не сам поток. И, как было сказано ранее, объектная иерархия (см. *главу 2*) принадлежности объектов к какому-либо потоку будет строиться согласно тому, в контексте какого из потоков эти объекты были созданы или помещены в потоки вызовом метода QObject::moveToThread(). Класс QThread унаследован от QObject, и, следовательно, ничем по своему принципу не отличается от других объектов Qt. А значит, если мы определим слоты в унаследованном от QThread классе и создадим от него объект в основном потоке, например, из функции main(), то сам объект управления потоком будет принадлежать основному потоку, и его слоты станут вызываться не в потоке, которым он управляет, а в основном потоке приложения.

Следовательно, если мы хотим, чтобы слоты обрабатывались в отдельном потоке, то нам нужно создавать объекты непосредственно из класса управления потоком, либо помещать их туда методом QObject::moveToThread(). Как это сделать, показывает следующий пример (листинги 38.2–38.4). В нем после запуска программы осуществляется отсчет таймера (рис. 38.3) от 10 к 0, после чего происходит завершение приложения. Объект-обладатель таймера будет вызывать слот внутри отдельного потока, в котором созданы он и таймер.

Класс MyWorker (листинг 38.2) представляет собой класс, объект которого мы будем исполнять в отдельном потоке. Он унаследован от класса QObject, и в его определении указан

макрос Q_OBJECT, что необходимо для использования сигналов и слотов. В конструкторе класса мы инициализируем атрибут m_nValue, создаем объект m_ptimer от класса QTimer и производим его соединение со слотом slotNextValue().

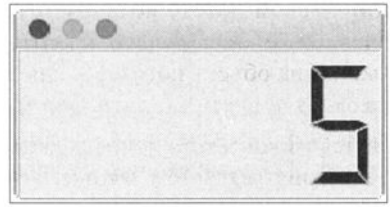


Рис. 38.3. Окно таймера

Слот slotNextValue() уменьшает значение атрибута m_nValue на единицу и отправляет сигнал valueChanged() с его актуальным значением. Если значение станет нулевым, то будет выслан сигнал finished(), информирующий о конце работы. Для запуска работы таймера мы реализуем в классе слот slotDoWork().

Листинг 38.2. Класс управления потоком MyWorker (файл MyWorker.h)

```
class MyWorker : public QObject {
Q_OBJECT
private:
    int     m_nValue;
    QTimer* m_ptimer;

public:
    MyWorker(QObject* pobj = 0) : QObject(pobj)
                                , m_nValue(10)
    {
        m_ptimer = new QTimer(this);

        connect(m_ptimer, SIGNAL(timeout()), SLOT(setNextValue()));
    }

signals:
    void valueChanged(int);
    void finished    (   );

public slots:
    void slotDoWork()
    {
        m_ptimer->start(1000);
    }

private slots:
    void setNextValue()
    {
        emit valueChanged(--m_nValue);
```

```
        if(!m_nValue){
            m_ptimer->stop();
            emit finished();
        }
    }
};
```

В листинге 38.3 приводится основная программа, в которой мы создаем виджет электронного индикатора `lcd`, объект управления потоком `thread` и объект класса `MyWorker`, который будет работать в отдельном потоке. Для отображения значений, высылаемых из потока, мы соединяем сигнал `valueChanged()` класса `MyWorker` со слотом виджета индикатора `display()`. Вызовом метода `moveToThread()` мы помещаем объект класса `MyWorker` в поток и соединяем его сигнал `finished()` со слотом `quit()` класса приложения — чтобы завершить работу всего приложения. Запускаем поток методом `start()`. Этот вызов, благодаря соединению сигнала `started()` объекта потока `thread`, приведет к вызову слота `MyWorker::slotDoWork()`. По завершении работы приложения мы вызываем из объекта потока метод `quit()` — это нужно, чтобы завершить цикл обработки сообщений и сам поток. Но цикл обработки не завершается моментально, поэтому нам необходимо дождаться, когда это произойдет, по этой причине мы вызываем метод `wait()`. Если этого не сделать, то по завершении программы может произойти аварийный выход, и будет показано некрасивое окно с цепочкой вызовов программы.

Листинг 38.3. Основная программа таймера — функция `main()` (файл main.cpp)

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QLCDNumber   lcd;
    QThread      thread;
    MyWorker     worker;

    QObject::connect(&worker, SIGNAL(valueChanged(int)),
                     &lcd,    SLOT(display(int))
                    );

    lcd.setSegmentStyle(QLCDNumber::Filled);
    lcd.display(10);
    lcd.resize(220, 90);
    lcd.show();

    worker.moveToThread(&thread);

    QObject::connect(&thread, SIGNAL(started()),
                     &worker, SLOT(slotDoWork())
                    );

    QObject::connect(&worker, SIGNAL(finished()),
                     &app, SLOT(quit())
                    );
```

```
    thread.start();

    int nResult = app.exec();

    thread.quit();
    thread.wait();

    return nResult;
}
```

Если поток должен только отправлять сигналы, то запуск цикла обработки событий не нужен.

Теперь давайте создадим более простой пример потока — без цикла сообщений. В приложении (листинги 38.5 и 38.6), выполнение которого показано на рис. 38.4, используется поток, отправляющий сигналы со значениями для индикатора выполнения.



Рис. 38.4. Поток, отправляющий сигналы

В методе нашего потока `run()`, показанном в листинге 38.4, мы запускаем цикл от 0 до 100, в котором через каждые 100 мс отправляется сигнал `progress()` с актуальным значением переменной цикла. Остановка выполнения цикла на 100 мс осуществляется при помощи метода `QThread::usleep()`, который принимает в качестве аргумента время задержки в микросекундах.

Листинг 38.4. Класс управления потоком `MyThread` (файл main.cpp)

```
class MyThread : public QThread {
Q_OBJECT

public:
    void run()
    {
        for (int i = 0; i <= 100; ++i) {
            usleep(100000);
            emit progress(i);
        }
    }

signals:
    void progress(int);
};
```

В листинге 38.5 после создания виджета индикатора выполнения осуществляется создание одного потока — `thread`, а его сигнал `progress()` соединяется для отображения высылаемых им значений со слотом `setValue()` виджета индикатора выполнения. Запуск потока приводится в действие методом `start()`.

Листинг 38.5. Основная программа индикатора — функция `main()` (файл main.cpp)

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QProgressBar prb;
    MyThread     thread;

    QObject::connect(&thread, SIGNAL(progress(int)),
                     &prb,    SLOT(setValue(int))
                    );

    prb.show();

    thread.start();

    return app.exec();
}
#include "main.moc"
```

Отправка событий

Отправка событий — это еще одна из возможностей для осуществления связи между объектами. Как мы знаем из *главы 16*, есть два метода для отправки событий: `QCoreApplication::postEvent()` и `QCoreApplication::sendEvent()`. Здесь присутствует небольшой нюанс, который нужно знать: отправка событий методом `postEvent()` обладает надежностью в потоках, а методом `sendEvent()` — нет. Поэтому при работе с разными потоками всегда используйте метод `postEvent()`. На рис. 38.5 показано, как с помощью механизма обмена событиями разных потоков можно осуществлять коммуникацию между двумя потоками. Поток может отправлять события другому потоку, который, в свою очередь, может ответить другим событием, и т. д. Сами же события, обрабатываемые циклами событий потоков, будут принадлежать тем потокам, в которых они были созданы.

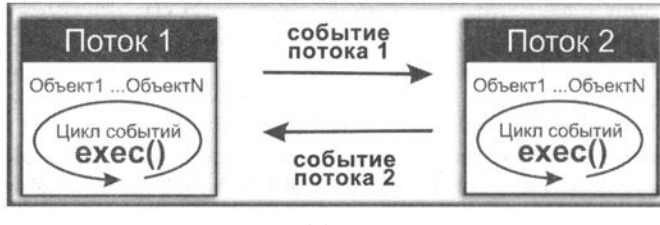


Рис. 38.5. Обмен событиями

Для того чтобы объект потока был в состоянии обрабатывать получаемые события, в классе потока нужно реализовать метод `QObject::event()`.

Если поток предназначен исключительно для отправки событий, а не для их получения, то реализацию методов обработки событий и запуск цикла обработки событий можно опустить. Для сравнения отправки событий с отправкой сигналов давайте реализуем программу отправки событий из потока (листинги 38.6–38.9), аналогичную программе, приведенной в листингах 38.4 и 38.5 (см. рис. 38.4).

Первое, что нам нужно сделать, — это создать класс для нашего события, которое мы будем отправлять из потока (листинг 38.6). Наш класс наследуется от класса `QEvent` и определяет атрибут целого типа `m_nValue`. Для получения и установки значений этого атрибута класс содержит методы `value()` и `setValue()`. В конструкторе мы задаем тип нашего события, передавая его целочисленный идентификатор в конструктор класса `QEvent()`.

Листинг 38.6. Класс события `ProgressEvent` (файл main.cpp)

```
class ProgressEvent : public QEvent {
private:
    int m_nValue;

public:
    enum {ProgressType = User + 1};

    ProgressEvent() : QEvent((Type)ProgressType)
    {
    }

    void setValue(int n)
    {
        m_nValue = n;
    }

    int value() const
    {
        return m_nValue;
    }
};
```

В листинге 38.7 приведена реализация класса `MyThread` нашего класса потока. Мы определяем в классе атрибут, указывающий на объект-получатель нашего события, создаем объект нашего класса события и отправляем его объекту-получателю с помощью метода `postEvent()`. Создание нашего события в методе `run()` без его уничтожения очень похоже на *утечку памяти* (memory leak), но ею не является, так как после обработки все объекты событий удаляются.

Решение с использованием сигналов

Очевидно, что если бы нам понадобилось выслать событие еще одному объекту, то пришлось бы создать второй объект класса и повторить действия, проделанные для первого события. Для третьего пришлось бы еще раз повторить код и т. д. При отправке сигналов нам этого делать не нужно, так как объект-получатель задается методом `QObject::connect()`. Поэтому решение с использованием сигналов, в нашем случае, смотрится более элегантно. Мы всего лишь один раз отправляем сигнал независимо от числа его получателей (см. листинг 38.2).

Листинг 38.7. Класс потока `MyThread` (файл main.cpp)

```
class MyThread : public QThread {
private:
    QObject* m_pobjReceiver;
```

```
public:
    MyThread(QObject* pobjReceiver) : m_pobjReceiver(pobjReceiver)
    {
    }

    void run()
    {
        for (int i = 0; i <= 100; ++i) {
            usleep(100000);

            ProgressEvent* pe = new ProgressEvent;
            pe->setValue(i);
            QApplication::postEvent(m_pobjReceiver, pe);
        }
    }
};
```

Для того чтобы виджет мог получать и правильно интерпретировать отправляемые потоком события, у нас есть две возможности. Первая заключается в реализации класса фильтра событий и установки его в виджете (см. *главу 15*). Вторая — в наследовании подходящего класса виджета и переопределении в нем метода `customEvent()`. Это решение представлено в листинге 38.8. В методе `customEvent()` мы проверяем тип полученного события и, если он соответствует типу нашего события `ProgressType`, приводим его к классу `ProgressEvent`, чтобы вызвать метод `value()`. Возвращаемое этим методом значение передается методу `QProgressBar::setValue()` для отображения виджетом хода процесса.

Листинг 38.8. Класс виджета индикации процесса `MyProgressBar` (файл main.cpp)

```
class MyProgressBar : public QProgressBar {
public:
    MyProgressBar(QWidget* pwgt = 0) : QProgressBar(pwgt)
    {
    }

    void customEvent(QEvent* pe)
    {
        if ((int)pe->type() == ProgressEvent::ProgressType) {
            setValue(((ProgressEvent*)(pe))->value());
        }
        QWidget::customEvent(pe);
    }
};
```

В основной программе, приведенной в листинге 38.9, мы создаем виджет индикации процесса и объект управления потоком `thread`, после чего выполняем запуск потока вызовом метода `start()`.

Листинг 38.9. Основная программа индикации процесса — функция main() (файл main.cpp)

```
int main(int argc, char** argv)
{
    QApplication  app(argc, argv);
    MyProgressBar prb;
    MyThread      thread(&prb);

    prb.show();

    thread.start();

    return app.exec();
}
```

Если сравнить реализации программы при помощи сигналов (см. листинги 38.4 и 38.5) и событий (см. листинги 38.6–38.9), то заметно, что подход с использованием сигналов более компактный. Но в целом оба подхода равнозначны, и оба имеют право на существование. Какой из подходов использовать, зависит от вас. В конкретных ситуациях следует оценить, насколько удобным будет применение каждого из них.

Синхронизация

Основные сложности возникают тогда, когда потокам нужно совместно использовать одни и те же данные. С одной стороны, это просто, так как несколько потоков могут одновременно обращаться и записывать данные в одну область. Но, с другой стороны, это может привести к нежелательным последствиям. Представьте себе такую ситуацию: один поток занимается вычислениями, задействуя значения какой-нибудь глобальной переменной, а в это время другой поток вдруг изменяет значение этой переменной. Поток, занимающийся вычислениями, ничего не подозревая, продолжает свою работу и по-прежнему использует исходное, еще не измененное значение, поэтому результат вычислений может оказаться совершенно бессмысленным. Для предотвращения подобных ситуаций требуется механизм, позволяющий блокировать данные, когда один из потоков намеревается их изменить. Этот механизм получил название *синхронизация*.

Синхронизация позволяет задавать *критические секции* (critical sections), к которым в определенный момент времени имеет доступ только один из потоков. Это гарантирует, что данные ресурса, контролируемые критической секцией, будут невидимы для других потоков и не будут непредвиденно изменены. И только после того, как поток выполнит всю необходимую работу, он освобождает ресурс, после чего доступ к этому ресурсу может получить любой другой поток. Например, если один поток записывает информацию в файл, то все другие не смогут использовать этот файл до тех пор, пока поток его не освободит.

Мьютексы

Взаимоисключающий доступ к ресурсам, гарантирующий, что критическая секция будет обрабатываться только одним потоком, обеспечивают так называемые *мьютексы* (mutex). Поток, владеющий мьютексом, обладает эксклюзивным правом на использование ресурса, защищенного мьютексом, и другой поток не может завладеть уже занятым мьютексом.

Мьютексы можно образно сравнить с дверью душевой кабинки, способной вместить только одного человека. Зайдя в кабинку, человек закрывает дверь. И если теперь кто-либо еще захочет воспользоваться этой душевой кабинкой, то он просто не сможет в нее попасть, так как дверь кабинки заперта. Когда кабинка освободится, ее дверь откроется, и в нее сможет войти следующий желающий. Войдя в нее, он закроет за собой дверь, сделав ее недоступной для других, и т. д.

> ***Исключения из любого правила...***
>
> Я не беру во внимание совместное принятие душа, ибо в реальном мире могут встретиться исключения из любого правила, которые на программирование не распространяются.

По этому принципу работает и мьютекс, только вместо людей выступают потоки, а вместо кабинок — ресурсы. Этот механизм реализован классом `QMutex`. Метод `lock()` класса `QMutex` выполняет блокировку ресурса. Для обратной операции существует метод `unlock()`, который открывает закрытый ресурс для других потоков.

Класс `QMutex` также содержит метод `tryLock()`. Его можно использовать для того, чтобы проверить, заблокирован ресурс или нет. Этот метод не приостанавливает исполнение потока и возвращается немедленно — со значением `false`, если ресурс уже захвачен другим потоком, не ожидая его освобождения. В случае успешного захвата ресурса этот метод вернет значение `true`, и это значит, что ресурс принадлежит потоку, и он вправе распоряжаться им по своему усмотрению.

Давайте воспользуемся мьютексом (листинг 38.10), чтобы реализовать класс с механизмом надежности использования в потоках (thread safety).

Листинг 38.10. Класс стека строк с механизмом надежности использования в потоках

```
class ThreadSafeStringStack {
private:
    QMutex          m_mutex;
    QStack<QString> m_stackString;

public:
    void push(const QString& str)
    {
        m_mutex.lock();
        m_stackString.push(str);
        m_mutex.unlock();
    }

    QString pop()
    {
        QMutexLocker locker(&m_mutex);
        return m_stackString.empty() ? QString()
                                     : m_stackString.pop();
    }
};
```

В классе `ThreadSafeStringStack`, приведенном в листинге 38.11, мы определяем два метода, один из которых: `push()` — служит для помещения строки в стек, а другой: `pop()` — для извлечения из стека. В секции `private` определены два атрибута: атрибут мьютекса `m_mutex`

и атрибут стека строк. Не забывайте, что класс QStack<T> не обладает механизмом надежности использования в потоках. Единственная возможность синхронизировать доступ к данным объекта класса QStack<T> — это блокировать их каждый раз, как только кто-то получит к ним доступ, и разблокировать после завершения операции. В методе push() самой первой строкой вызывается метод lock() объекта мьютекса, который блокирует доступ к ресурсу, а после помещения строкового значения в стек вызов метода unlock() разблокирует его. Теперь этот метод могут вызвать несколько потоков одновременно, и это не приведет к порче данных.

В методе pop() мы используем объект класса QMutexLocker. Иногда очень удобно применять именно этот класс. Для создания объекта класса QMutexLocker в его конструктор необходимо передать указатель на объект мьютекса. В конструкторе этого класса ресурс сразу же блокируется, а в деструкторе — разблокируется. Это означает, что не нужно явно вызывать метод для разблокирования ресурса, как мы это делали в методе push(), потому что завершение метода приведет к разрушению этого объекта, и будет вызван его деструктор.

> ***Применение блокировки может снизить эффективность работы приложения!***
>
> Золотое правило: никогда не используйте мьютексы, если вы не уверены в том, что это действительно необходимо. Бытует мнение, что классы, не обладающие надежностью для потока, — это плохо. Проблема, связанная с реализацией надежности использования в потоках, заключается в том, что ее невозможно осуществить без механизма блокировки. Однако применение блокировки ресурсов может снизить эффективность работы класса, а значит, и приложения в целом. Представьте себе, что каждый класс вашего приложения должен блокировать и разблокировать все ресурсы — это бы резко снизило быстродействие вашей программы. Именно по этой причине не все классы Qt реализованы с механизмом надежности.

Семафоры

Семафоры являются обобщением мьютексов. Как и мьютексы, они служат для защиты критических секций, чтобы доступ к ним одновременно могло иметь определенное число потоков. Все другие потоки обязаны ждать. Предположим, что программа поддерживает пять ресурсов одного и того же типа, одновременный доступ к которым может быть предоставлен только пяти потокам. Как только все пять ресурсов будут заблокированы, следующий поток, запрашивающий ресурс этого типа, будет приостановлен до освобождения одного из них. Принцип действия семафоров очень прост. Они начинают действовать с установленного значения счетчика. Каждый раз, когда поток получает право на владение ресурсом, значение этого счетчика уменьшается на единицу. И наоборот, когда поток уступает право владения этим ресурсом, счетчик на единицу увеличивается. При значении счетчика, равном нулю, семафор становится недоступным. Механизм семафоров реализует класс QSemaphore. Счетчик устанавливается в конструкторе при создании объекта этого класса. В целом все это весьма похоже на применение класса QMutex, только вместо метода lock() и unlock() используются acquire() и release() соответственно:

```
QSemaphore sem(1);
sem.acquire();
```

В этом примере мы запрашиваем доступ. В методе acquire() можно передать число объектов, к которым требуется получить доступ. В нашем случае мы вызываем этот метод без параметров и используем значение по умолчанию, которое равно единице. В общем случае взаимодействие выглядит следующим образом. Если значение счетчика семафора меньше значения, переданного в метод acquire(), то поток, запросивший доступ, войдет в цикл

ожидания, пока не произойдет освобождение нужных ресурсов. Если же значение счетчика семафора больше или равно значению метода `acquire()`, то оно уменьшится на единицу, и поток получит доступ к ресурсу. После выполнения требуемых действий нужно вызвать метод `release()` и указать количество ресурсов для освобождения. Вызов этого метода без параметров высвобождает один ресурс.

Класс семафора предоставляет также «неблокирующий» метод `tryAcquire()` для запроса доступа к ресурсу.

Ожидание условий

Библиотека Qt предоставляет класс `QWaitCondition`, обеспечивающий возможность координации потоков. Это еще одно обобщение мьютекса, где доступ к ресурсу разрешается только при выполнении некоторого условия. Если поток намеревается дождаться разблокировки ресурса, то он вызывает метод `QWaitCondition::wait()` и тем самым входит в режим ожидания.

С учетом сказанного, используя объект `waitCondition` класса `QWaitCondition`, в пример из листинга 38.10 можно было бы внести следующие изменения:

```
QString ThreadSafeStringStack::pop()
{
    QMutexLocker locker(&m_mutex);
    while(m_stackString.empty()) {
        waitCondition.wait(&m_mutex);
    }
    return m_stackString.pop();
}
```

Выводится поток из этого режима в том случае, если поток, который заблокировал ресурс, вызовет метод `QWaitCondition::wakeOne()` или `QWaitCondition::wakeAll()`. Разница этих двух методов в том, что первый выводит из состояния ожидания только один поток, а второй — все сразу. Также можно установить время, в течение которого поток будет ожидать разблокировки данных. Для этого нужно вторым параметром передать в метод `wait()` целочисленное значение, обозначающее временной интервал в миллисекундах.

Блокировка чтения/записи

Существуют особые случаи синхронизации потоков, при которых некоторому количеству потоков разрешено получать доступ к ресурсу только для чтения, а только одному разрешено получать доступ к ресурсу для записи. А также и такие случаи, когда один из потоков записывает данные и нужно, чтобы в это время ни один другой из потоков не смог получить доступ к этим данным для чтения. В подобных случаях лучше всего использовать класс `QReadWriteLock`. Этот класс предоставляет метод `lockForRead()` — для блокировки чтения ресурса и метод `lockForWrite()` — для блокировки записи ресурса.

С учетом сказанного, используя объект `readWriteLock` класса `QReadWriteLock`, в реализацию примера из листинга 38.10 можно было бы для последней ситуации внести следующие изменения:

```
QString ThreadSafeStringStack::push()
{
    readWriteLock.lockForRead();
```

```
    m_stackString.push(str);
    readWriteLock.unlock();
}
```

Возникновение тупиковых ситуаций

Работая с многопоточностью, нужно помнить о возможном возникновении тупиковых ситуаций, когда потоки могут заблокировать друг друга. Представьте себе такую ситуацию, когда поток заблокировал ресурс «A», а после работы над ним собирается работать с ресурсом «B». Другой же поток заблокировал ресурс «B» и по окончании намеревается работать с ресурсом «A». И вот один из потоков, закончив работу, обнаружил, что нужный ему ресурс заблокирован другим потоком. Он переходит в режим ожидания, надеясь дождаться разблокировки ресурса, но то же самое делает и другой поток. В итоге — оба ждут друг друга. Если ни один из этих потоков не освободит занятый им ресурс, то оба «зависнут» и не смогут продолжать свою работу дальше.

Это явление получило название *взаимной блокировки* (deadlock) (рис. 38.6). Существует множество решений такой проблемы. Например, можно организовать работу потока так, чтобы в случае, когда он не может получить доступ к необходимому ресурсу, он просто освободил бы все занятые им ресурсы, а позже повторил попытку их захвата.

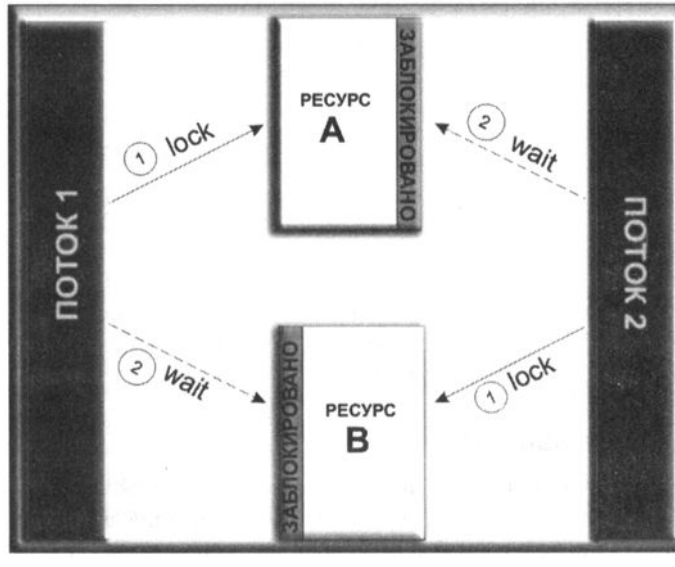


Рис. 38.6. Взаимная блокировка

Фреймворк *QtConcurrent*

В этой главе вы уже убедились, что реализовывать многопоточные программы не такое уж и простое дело, сопряженное не только с отладкой самих программ, но и с «подводными камнями», связанными с синхронизацией и с возникновением тупиковых ситуаций.

Для решения этих сложностей в Qt было добавлено новое пространство имен `QtConcurrent`. Это фреймворк высокого уровня, который создает уровень абстракции для управления по-

токами и синхронизацией. Он значительно упрощает написание мультипоточных приложений, что приводит к более быстрой разработке и уменьшению программного кода. Поэтому по возможности отдавайте предпочтение именно этому фреймворку. Для того чтобы его задействовать, необходимо включить в pro-файл строку:

```
QT += concurrent
```

В самом простом случае, когда нужно запустить функцию в отдельном потоке, ее нужно передать в качестве аргумента функции `QtConcurrent::run()`, как это показано в листинге 38.11.

Листинг 38.11. Использование фреймворка QtConcurrent (файл main.cpp)

```
#include <QtCore>
#include <QtConcurrent/QtConcurrent>

// ------------------------------------------------------------------------
QString myToUpper(const QString& str)
{
    return str.toUpper();
}

// ------------------------------------------------------------------------
int main(int argc, char** argv)
{
    QCoreApplication app(argc, argv);

    QFuture<QString> future =
         QtConcurrent::run(myToUpper, QString("test"));
    future.waitForFinished();
    qDebug() << future.result();

    return 0;
}
```

Обратите внимание, что функция, которую мы запускаем в потоке, принимает один аргумент. Аргументов можно использовать и больше, для этого их нужно передать третьим, четвертым и т. д. параметрами в ту же функцию `run()`. Наша функция `myToUpper()` возвращает значение строкового типа. Это значение будет содержаться в переменной класса `QFuture`, но лишь после того, как отработка функции в потоке окажется завершена. То есть, ее значение на момент присвоения неопределенно и будет известно только в будущем — отсюда и вытекает само название этого класса, которое переводится как «будущее». Класс `QFuture` — это шаблонный класс, он позволяет получать значения разных типов. Наша функция, например, возвращает `QString`. Сам запущенный процесс протекает асинхронно, но при помощи полезного метода `waitForFinished()` класса `QFuture` мы блокируем исполнение основного потока программы до тех пор, пока поток не завершит свою работу. Класс `QFuture` предоставляет и другие полезные методы, с помощью которых мы можем проследить за ходом выполнения работы наших задач, выполняемых в потоке:

- `QFuture::isFinished()` — выполнение задач завершено;
- `QFuture::isRunning()` — задачи в процессе выполнения;

- `QFuture::isStarted()` — задачи уже начали исполняться;
- `QFuture::isPaused()` — выполнение задач приостановлено методом `QFuture::pause()`.

Теперь представьте себе другую ситуацию, когда у нас имеются тысячи строк, которые мы хотим обработать в потоках при помощи нашей функции `myToUpper()`. Этого можно достичь, воспользовавшись функцией `QtConcurrent::mapped()`. В нее мы можем передать список аргументов, и она запустит нашу функцию `myToUpper()` для каждого элемента списка. Для этого просто изменим нашу функцию `main()` из листинга 38.11 (листинг 38.12).

Листинг 38.12. Использование `QtConcurrent` для списка (файл main.cpp)

```
int main(int argc, char** argv)
{
    QCoreApplication app(argc, argv);

    QStringList lst(QStringList() << "one" << "two" << "three");
    QFuture<QString> future =
          QtConcurrent::mapped(lst.begin(), lst.end(), myToUpper);
    future.waitForFinished();
    qDebug() << future.results();

    return 0;
}
```

В листинге 38.12 мы создаем список из трех элементов строк и передаем его в функцию `QtConcurrent::mapped()` первым аргументом. Вторым аргументом передаем имя функции, которая будет использовать для списка функцию `myToUpper()`. Так же, как и в предыдущем листинге, здесь мы блокируем основной поток программы методом `QFuture::waitForFinished()` и после выполнения всех задач выводим на консоль получившийся в результате список строк. Он будет выглядеть так:

```
"ONE", "TWO", "THREE"
```

Резюме

Процессы представляют собой программы, независимые друг от друга и загруженные для исполнения. Каждый процесс должен создавать хотя бы один поток, называемый *основным*. Основной поток процесса создается в момент запуска программы. Однако сам процесс может создавать несколько потоков одновременно.

Многопоточность позволяет разделять задачи и работать независимо над каждой из них для того, чтобы максимально эффективно задействовать процессор. Написание многопоточных приложений требует больше времени и усложняет процесс отладки, поэтому многопоточность нужно применять только тогда, когда это действительно необходимо. Многопоточность удобно использовать для того, чтобы блокировка или зависание одного из методов не стали причиной нарушения функционирования основной программы.

Для реализации многопоточности нужно унаследовать класс от `QThread` и переопределить метод `run()`, который должен содержать код для выполнения в потоке. Чтобы запустить поток, нужно вызвать метод `start()`. Каждый поток может иметь свой собственный цикл событий, который запускается вызовом метода `exec()` в коде метода `run()`.

Связь между объектами из разных потоков можно осуществлять при помощи сигналов и слотов или посредством обмена объектами событий.

При работе с потоками нередко требуется синхронизировать функционирование потоков. Причиной синхронизации является необходимость обеспечения доступа нескольких потоков к одним и тем же данным. Для этого библиотека Qt предоставляет классы `QMutex`, `QWaitContion` и `QSemaphore`.

В целях эффективности не все классы Qt обладают механизмом надежности использования в потоках.

Фреймворк `QtConcurrent` предоставляет возможность работать с потоками на более высоком уровне и освобождает разработчиков от необходимости реализации механизмов управления потоками.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/38-510/** или с помощью следующего QR-кода (рис. 38.7):

Рис. 38.7. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 39

Программирование поддержки сети

Информация есть информация, а не энергия и не материя.
Н. Винер, «Кибернетика»

Сокетное соединение

Сóкет (от англ. *socket* — гнездо, разъем) — это устройство пересылки данных с одного конца линии связи на другой. Другой конец может принадлежать процессу, работающему на локальном компьютере, а может располагаться и на удаленном компьютере, подключенном к Интернету и расположенном в другом полушарии Земли. *Сóкетное соединение* — это соединение типа *точка-точка* (point to point), которое осуществляется между двумя процессами.

Сокеты разделяют на *дейтаграммные* (datagram) и *поточные* (stream). Дейтаграммные сокеты осуществляют обмен пакетами данных. Поточные сокеты устанавливают связь и выполняют потоковый обмен данными через установленную ими линию связи. На практике поточные сокеты используются гораздо чаще, чем дейтаграммные, из-за того, что они предоставляют дополнительные механизмы, направленные против искажения и потери данных. Поточные сокеты работают в обоих направлениях — другими словами, то, что один из процессов записывает в поток, может быть считано процессом на другом конце связи, и наоборот.

Для дейтаграммных сокетов Qt предоставляет класс `QUdpSocket`, а для поточных — класс `QTcpSocket`.

Класс `QTcpSocket` содержит набор методов для работы с *TCP* (Transmition Control Protocol, протокол управления передачей данных) — сетевым протоколом низкого уровня, который является одним из основных протоколов в Интернете. Это самый лучший способ для установления связи между двумя компьютерами и передачи данных между ними с высокой степенью надежности. С его помощью можно реализовать поддержку для стандартных сетевых протоколов, таких как HTTP, FTP, POP3, SMTP, и даже для своих собственных протоколов. Этот класс унаследован от класса `QAbstractSocket`, который, в свою очередь, наследует класс `QIODevice`. А это значит, что для доступа (чтения и записи) к его объектам можно применять все методы класса `QIODevice` и использовать классы потоков `QDataStream` или `QTextStream` (см. *главу 36*).

Работа класса `QTcpSocket` асинхронна, что дает возможность избежать блокировки приложения в процессе его работы. Но если вам это не нужно, то вы можете воспользоваться серией методов, начинающихся со слова `waitFor`. Вызов этих методов приведет к ожиданию

выполнения операции и заблокирует на определенное время исполнение вашей программы, поэтому не рекомендуется вызывать эти методы в потоке графического интерфейса.

Класс `QUdpSocket` содержит набор методов для работы с *UDP* (User Datagram Protocol, протокол пользовательских дейтаграмм). Этот сокет реализует ненадежную передачу данных, не ориентированную на установление соединения между отправителем и получателем сообщения. Он не гарантирует доставки пакета, что является одновременно и недостатком, и преимуществом, так как позволяет быстрее и эффективнее доставлять данные для приложений, которым необходима большая пропускная способность линий связи либо нужно малое время доставки данных. Реализация в программах протокола UDP гораздо проще, чем реализация TCP, поэтому некоторые разработчики отдают предпочтение в его использовании так же и по этой причине.

Протокол UDP очень хорош для пересылки между компьютерами индивидуальных независимых пакетов, которые называются *дейтаграммами*. Выславшему нет необходимости знать, получен его пакет или нет, а получателю нет необходимости знать, получил ли он действительно все данные, которые были высланы. Это значит, что если доставка пакета в пункт назначения произошла неудачно, то никакого сообщения об ошибке отправителю передано не будет. Соответственно, UDP незаменим там, где быстрота важнее надежности. Например, при передаче видео лучше выбросить несколько кадров, чем отстать по времени.

Иногда в приложениях практикуют гибридное решение сразу из обоих протоколов: TCP и UDP. Смысл заключается в том, чтобы использовать только самые сильные их стороны. Такое решение выглядит следующим образом — все контрольные процедуры осуществляются посредством протокола TCP, а непосредственно для передачи данных используется UDP.

Модель «клиент-сервер»

Сценарий модели «клиент-сервер» выглядит очень просто: сервер предлагает услуги, а клиент ими пользуется (рис. 39.1). Программа, реализующая сокеты, может выполнять либо роль сервера, либо роль клиента. Для того чтобы клиент мог взаимодействовать с сервером, ему нужно знать IP-адрес сервера и номер порта, через который клиент должен сообщить о себе. Когда клиент начинает соединение с сервером, его система назначает этому соединению отдельный сокет, а когда сервер принимает соединение, сокет назначается со стороны сервера. После этого между двумя взаимодействующими сокетами устанавливается связь, по которой высылаются данные запроса к серверу. А сервер по тому же соединению, согласно запросу клиента, высылает готовые результаты. Сервер не ограничен связью только с одним клиентом — на самом деле он может обслуживать многих клиентов.



Рис. 39.1. Модель «клиент-сервер»

Каждому клиентскому сокету соответствует уникальный номер порта. Некоторые номера зарезервированы для так называемых *стандартных служб* (табл. 39.1).

Таблица 39.1. *Некоторые зарезервированные номера портов*

Порт	Сервис	Описание
11	systat	Показ зарегистрированных в системе пользователей
20, 21	FTP	Доступ к файлам по сети
22	SSH	Зашифрованное соединение с удаленным компьютером
23	Telnet	Удаленное соединение с компьютером
25, 587	SMTP	Отсылка электронной почты
43	WHOIS	Who is (кто это). Получение регистрационных данных о владельцах доменных имен и IP-адресов
53	DNS	Система доменных имен
80	HTTP	Веб-сервер (иногда в качестве альтернативы применяются порты 8080 или 8000)
110	POP3	Получение электронной почты
115	SFTP	SSH File Transfer Protocol — протокол прикладного уровня, обеспечивающий доступ к файлам по сети посредством надежного и зашифрованного соединения
139	Netbios-SSN	Разделение сетевых ресурсов
123	NTP	Network Time Protocol (протокол сетевого времени) — используется для синхронизации часов компьютера
143	IMAP	Пересылка электронной почты
194	IRC	Ретранслируемый интернет-чат
443	HTTPS	Зашифрованный HTTP
873	Rsync	Используется программой на UNIX-подобных системах, которая занимается синхронизацией файлов и каталогов
5800, 5900	VNC	Система удаленного доступа к рабочему столу компьютера
8000 и 8080	Web	Альтернативный HTTP-порт

Реализация TCP-сервера

Для реализации TCP-сервера Qt предоставляет удобный класс `QTcpServer`, который предназначен для управления входящими TCP-соединениями. Программа (листинги 39.1–39.6), окно которой показано на рис. 39.2, является реализацией простого сервера, принимающего и подтверждающего получение запросов клиентов.

В листинге 39.1 создается объект сервера. Чтобы запустить сервер, мы создаем объект определенного нами в листингах 39.2–39.6 класса `MyServer`, передав в конструктор номер порта, по которому должен осуществляться нужный сервис. В нашем случае передается номер порта, равный 2323.



Рис. 39.2. Реализация сервера

Листинг 39.1. Реализация простого сервера (файл main.cpp)

```
#include <QtWidgets>
#include "MyServer.h"

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    MyServer     server(2323);

    server.show();

    return app.exec();
}
```

В классе `MyServer`, определяемом в листинге 39.2, мы объявляем атрибут `m_ptcpServer`, который и является основой управления нашим сервером. Атрибут `m_nNextBlockSize` служит для хранения длины следующего полученного от сокета блока. Многострочное текстовое поле `m_ptxt` предназначено для информирования о происходящих соединениях.

Листинг 39.2. Определение класса `MyServer` (файл MyServer.h)

```
#pragma once

#include <QWidget>

class QTcpServer;
class QTextEdit;
class QTcpSocket;

// =============================================================================
class MyServer : public QWidget {
Q_OBJECT
private:
    QTcpServer* m_ptcpServer;
```

```
    QTextEdit*  m_ptxt;
    quint16     m_nNextBlockSize;

private:
    void sendToClient(QTcpSocket* pSocket, const QString& str);

public:
    MyServer(int nPort, QWidget* pwgt = 0);

public slots:
    virtual void slotNewConnection();
            void slotReadClient   ();
};
```

Для запуска сервера нам необходимо вызвать в конструкторе метод `listen()` (листинг 39.3). В этот метод нужно передать номер порта, который мы получили в конструкторе. При возникновении ошибочных ситуаций, например невозможности захвата порта, этот метод возвратит значение `false`, на которое мы отреагируем показом окна сообщения об ошибке. Если ошибки не произошло, мы соединяем определенный нами (см. далее листинг 39.4) слот `slotNewConnection()` с сигналом `newConnection()`, который отправляется при каждом присоединении нового клиента.

Для отображения информации мы создаем виджет многострочного текстового поля (указатель `m_ptxt`) и вызовом метода `setReadOnly()` устанавливаем в нем режим, в котором возможен только просмотр информации.

Листинг 39.3. Конструктор класса MyServer (файл MyServer.cpp)

```
MyServer::MyServer(int nPort, QWidget* pwgt /*=0*/) : QWidget(pwgt)
                                                    , m_nNextBlockSize(0)
{
    m_ptcpServer = new QTcpServer(this);
    if (!m_ptcpServer->listen(QHostAddress::Any, nPort)) {
        QMessageBox::critical(0,
                              "Server Error",
                              "Unable to start the server:"
                              + m_ptcpServer->errorString()
                             );
        m_ptcpServer->close();
        return;
    }
    connect(m_ptcpServer, SIGNAL(newConnection()),
            this,         SLOT(slotNewConnection())
           );

    m_ptxt = new QTextEdit;
    m_ptxt->setReadOnly(true);

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
```

```
    pvbxLayout->addWidget(new QLabel("<H1>Server</H1>"));
    pvbxLayout->addWidget(m_ptxt);
    setLayout(pvbxLayout);
}
```

Метод `slotNewConnection()`, показанный в листинге 39.4, вызывается каждый раз при соединении с новым клиентом. Для подтверждения соединения с клиентом необходимо вызвать метод `nextPendingConnection()`, который возвратит сокет, посредством которого можно осуществлять дальнейшую связь с клиентом. Последующая работа сокета обеспечивается сигнально-слотовыми связями. Мы соединяем сигнал `disconnected()`, отправляемый сокетом при отсоединении клиента, со стандартным слотом `QObject::deleteLater()`, предназначенным для его последующего уничтожения. При поступлении запросов от клиентов отправляется сигнал `readyToRead()`, который мы соединяем со слотом `slotReadClient()`.

Листинг 39.4. Метод `slotNewConnection()` (файл MyServer.cpp)

```
/*virtual*/ void MyServer::slotNewConnection()
{
    QTcpSocket* pClientSocket = m_ptcpServer->nextPendingConnection();
    connect(pClientSocket, SIGNAL(disconnected()),
            pClientSocket, SLOT(deleteLater())
           );
    connect(pClientSocket, SIGNAL(readyRead()),
            this,          SLOT(slotReadClient())
           );

    sendToClient(pClientSocket, "Server Response: Connected!");
}
```

В листинге 39.5 сначала выполняется преобразование указателя, возвращаемого методом `sender()`, к типу `QTcpSocket`. Цикл `for` нам нужен потому, что не все высланные клиентом данные могут прийти одновременно. Поэтому сервер должен «уметь» получать как весь блок целиком, так и только часть блока, а также и все блоки сразу. Каждый переданный сокетом блок начинается полем размера блока. Размер блока считывается при условии, что размер полученных данных не меньше двух байтов, и атрибут `m_nNextBlockSize` равен нулю (то есть размер блока неизвестен). Если размер доступных для чтения данных больше или равен размеру блока, тогда данные считываются из потока в переменные `time` и `str`. Затем значение переменной `time` преобразуется вызовом метода `toString()` в строку и вместе со строкой `str` записывается в строку сообщения `strMessage`, которая добавляется в виджет текстового поля вызовом метода `append()`. Анализ блока данных завершается присваиванием атрибуту `m_nNextBlockSize` значения 0, которое говорит о том, что размер очередного блока данных неизвестен. Вызовом метода `sendToClient()` мы сообщаем клиенту о том, что нам успешно удалось прочитать высланные им данные.

Листинг 39.5. Метод `slotReadClient()` (файл MyServer.cpp)

```
void MyServer::slotReadClient()
{
    QTcpSocket* pClientSocket = (QTcpSocket*)sender();
    QDataStream in(pClientSocket);
```

```
    in.setVersion(QDataStream::Qt_5_3);
    for (;;) {
        if (!m_nNextBlockSize) {
            if (pClientSocket->bytesAvailable() < sizeof(quint16)) {
                break;
            }
            in >> m_nNextBlockSize;
        }

        if (pClientSocket->bytesAvailable() < m_nNextBlockSize) {
            break;
        }
        QTime   time;
        QString str;
        in >> time >> str;

        QString strMessage =
            time.toString() + " " + "Client has sent — " + str;
        m_ptxt->append(strMessage);

        m_nNextBlockSize = 0;

        sendToClient(pClientSocket,
                     "Server Response: Received \"" + str + "\""
                    );
    }
}
```

В методе `sendToClient()` мы формируем данные, которые будут отосланы клиенту (листинг 39.6). Есть небольшой нюанс, заключающийся в том, что нам заранее не известен размер блока, а следовательно, мы не можем записывать данные сразу в сокет, так как размер блока должен быть выслан в первую очередь. Поэтому мы прибегаем к следующему трюку. Сначала создаем объект `arrBlock` класса `QByteArray`. На его основе создаем объект класса `QDataStream`, в который записываем все данные блока, причем вместо реального размера записываем 0. После этого перемещаем указатель на начало блока вызовом метода `seek()`, вычисляем размер блока как размер `arrBlock`, уменьшенный на `sizeof(quint16)`, и записываем его в поток (`out`) с текущей позиции, которая уже перемещена в начало блока. После этого созданный блок записывается в сокет вызовом метода `write()`.

Класс QDataStream — пересылка бинарных данных

Для пересылки обычных строк мы могли бы задействовать класс потока ввода `QTextStream`. В нашем же случае используется класс `QDataStream`, поскольку пересылка бинарных данных представляет собой общий случай. Нам бинарные данные также необходимы, чтобы переслать объект класса `QTime`. Используя класс `QDataStream`, вы можете отправлять не только строки, но и растровые изображения, объекты палитры и т. д.

Листинг 39.6. Метод `sendToClient()` (файл MyServer.cpp)

```
void MyServer::sendToClient(QTcpSocket* pSocket, const QString& str)
{
    QByteArray  arrBlock;
```

```
    QDataStream out(&arrBlock, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_5_3);
    out << quint16(0) << QTime::currentTime() << str;

    out.device()->seek(0);
    out << quint16(arrBlock.size() - sizeof(quint16));

    pSocket->write(arrBlock);
}
```

Реализация TCP-клиента

Для реализации клиента нужно создать объект класса `QTcpSocket`, а затем вызвать метод `connectToHost()`, передав в него первым параметром имя компьютера (или его IP-адрес), а вторым — номер порта сервера. Объект класса `QTcpSocket` сам попытается установить связь с сервером и, в случае успеха, вышлет сигнал `connected()`. В противном случае будет выслан сигнал `error(int)` с кодом ошибки, определенным в перечислении `QAbstractSocket::SocketError`. Это может произойти, например, в том случае, если на указанном компьютере не запущен сервер или не соответствует номер порта. После установления соединения объект класса `QTcpSocket` может высылать данные серверу или считывать их с него.

Следующий пример (листинги 39.7–39.13) демонстрирует взаимодействие клиента с сервером (рис. 39.3). Пересылка на сервер информации, введенной в однострочном текстовом поле окна клиента, осуществляется после нажатия на кнопку **Send** (Выслать).

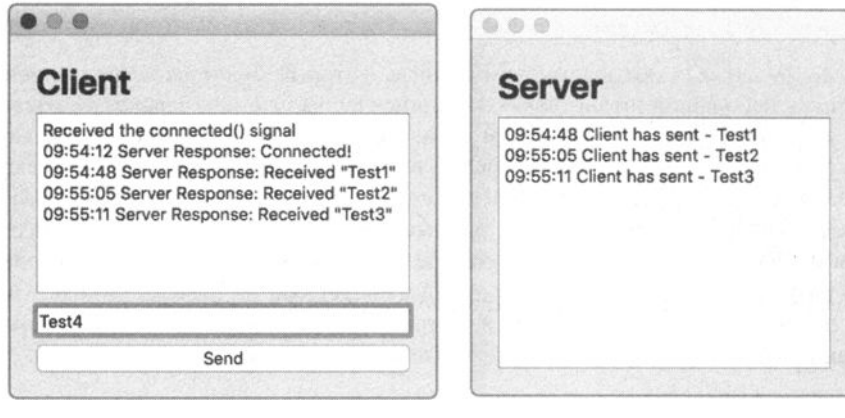


Рис. 39.3. Взаимодействие клиента с сервером

В функции `main()`, приведенной в листинге 39.7, мы создаем объект клиента (см. листинги 39.8–39.13). Если сервер и клиент запускаются на одном компьютере, то в качестве имени компьютера можно передать строку `"localhost"`. Номер порта в нашем случае равен 2323, так как это тот порт, который используется нашим сервером.

Листинг 39.7. Взаимодействие клиента с сервером (файл main.cpp)

```
#include <QApplication>
#include "MyClient.h"
```

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    MyClient     client("localhost", 2323);

    client.show();

    return app.exec();
}
```

В классе `MyClient`, определяемом в листинге 39.8, мы объявляем атрибут `m_pTcpSocket`, который нужен для управления нашим клиентом, и атрибут `m_nNextBlockSize`, необходимый нам для хранения длины следующего полученного от сокета блока. Остальные два атрибута: `m_txtInfo` и `m_txtInput` — служат для отображения и ввода информации соответственно.

Листинг 39.8. Определение класса `MyClient` (файл MyClient.h)

```
#pragma once

#include <QWidget>
#include <QTcpSocket>

class QTextEdit;
class QLineEdit;

// ====================================================================
class MyClient : public QWidget {
Q_OBJECT
private:
    QTcpSocket* m_pTcpSocket;
    QTextEdit*  m_ptxtInfo;
    QLineEdit*  m_ptxtInput;
    quint16     m_nNextBlockSize;

public:
    MyClient(const QString& strHost, int nPort, QWidget* pwgt = 0) ;

private slots:
    void slotReadyRead   (                            );
    void slotError       (QAbstractSocket::SocketError);
    void slotSendToServer(                            );
    void slotConnected   (                            );
};
```

В листинге 39.9 приведен конструктор, в котором прежде всего создается объект сокета (указатель `m_pTcpSocket`). Затем вызывается метод `connectToHost()` этого объекта, устанавливающий связь с сервером. Первым параметром в этот метод передается имя компьютера, а вторым — номер порта. Коммуникация сокетов асинхронна. Сокет отправляет сигнал `connected()`, как только будет создано соединение, а также сигнал `readyRead()` — при готовности предоставить данные для чтения. Мы соединяем эти сигналы со слотами

slotConnected() и slotReadyRead(). В случае возникновения ошибок сокет отправляет сигнал error(), который мы соединяем со слотом slotError(), предназначенным для отображения ошибок.

Затем создается пользовательский интерфейс программы, состоящий из надписи, кнопки, однострочного и многострочного текстовых полей. Сигнал clicked() виджета кнопки нажатия соединяется со слотом slotSendToServer() класса MyClient, ответственным за отправку данных на сервер. Для того чтобы к аналогичному действию приводило и нажатие клавиши <Enter>, мы соединяем сигнал returnPressed() виджета текстового поля (m_ptxtInput) с тем же слотом slotSendToServer().

Листинг 39.9. Конструктор класса MyClient (файл MyClient.cpp)

```
MyClient::MyClient(const QString& strHost,
                   int            nPort,
                   QWidget*       pwgt /*=0*/
                  ) : QWidget(pwgt)
                    , m_nNextBlockSize(0)
{
    m_pTcpSocket = new QTcpSocket(this);

    m_pTcpSocket->connectToHost(strHost, nPort);
    connect(m_pTcpSocket, SIGNAL(connected()), SLOT(slotConnected()));
    connect(m_pTcpSocket, SIGNAL(readyRead()), SLOT(slotReadyRead()));
    connect(m_pTcpSocket, SIGNAL(error(QAbstractSocket::SocketError)),
            this,         SLOT(slotError(QAbstractSocket::SocketError))
           );

    m_ptxtInfo  = new QTextEdit;
    m_ptxtInput = new QLineEdit;

    m_ptxtInfo->setReadOnly(true);

    QPushButton* pcmd = new QPushButton("&Send");
    connect(pcmd, SIGNAL(clicked()), SLOT(slotSendToServer()));
    connect(m_ptxtInput, SIGNAL(returnPressed()),
            this,        SLOT(slotSendToServer())
           );

    //Layout setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(new QLabel("<H1>Client</H1>"));
    pvbxLayout->addWidget(m_ptxtInfo);
    pvbxLayout->addWidget(m_ptxtInput);
    pvbxLayout->addWidget(pcmd);
    setLayout(pvbxLayout);
}
```

В листинге 39.10 приведен слот slotReadyToRead(), который вызывается при поступлении данных от сервера. Цикл for нужен потому, что с сервера не все данные могут прийти

одновременно. Поэтому клиент, как и сервер, должен быть в состоянии получать как весь блок целиком, так и только часть блока или даже все блоки сразу. Каждый принятый блок начинается полем размера блока.

Когда мы будем уверены, что блок получен целиком, можно без опасения использовать оператор >> объекта потока `QDataStream` (переменная `in`). Чтение данных из сокета осуществляется при помощи объекта потока данных. Полученная информация добавляется в виджет многострочного текстового поля (указатель `m_ptxtInfo`) с помощью метода `append()`.

В завершение анализа блока данных атрибуту `m_nNextBlockSize` присваиваем значение 0, которое говорит о том, что размер очередного блока данных неизвестен.

Листинг 39.10. Метод `slotReadyRead()` (файл MyClient.cpp)

```
void MyClient::slotReadyRead()
{
    QDataStream in(m_pTcpSocket);
    in.setVersion(QDataStream::Qt_5_3);
    for (;;) {
        if (!m_nNextBlockSize) {
            if (m_pTcpSocket->bytesAvailable() < sizeof(quint16)) {
                break;
            }
            in >> m_nNextBlockSize;
        }

        if (m_pTcpSocket->bytesAvailable() < m_nNextBlockSize) {
            break;
        }
        QTime   time;
        QString str;
        in >> time >> str;

        m_ptxtInfo->append(time.toString() + " " + str);
        m_nNextBlockSize = 0;
    }
}
```

Слот `slotError()`, приведенный в листинге 39.11, вызывается при возникновении ошибок. В нем мы преобразуем код ошибки в текст, а затем отображаем его в виджете многострочного текстового поля.

Листинг 39.11. Метод `slotError()` (файл MyClient.cpp)

```
void MyClient::slotError(QAbstractSocket::SocketError err)
{
    QString strError =
        "Error: " + (err == QAbstractSocket::HostNotFoundError ?
                     "The host was not found." :
                     err == QAbstractSocket::RemoteHostClosedError ?
                     "The remote host is closed." :
```

```
                    err == QAbstractSocket::ConnectionRefusedError ?
                    "The connection was refused." :
                    QString(m_pTcpSocket->errorString())
                   );
    m_ptxtInfo->append(strError);
}
```

В листинге 39.12 приведен метод отсылки запроса серверу. В нем используется тот же трюк, что и в аналогичном методе сервера (см. листинг 39.6), который уже был рассмотрен ранее. Единственное отличие от метода `MyServer::sendToClient()` заключается в том, что после отсылки данных мы дополнительно стираем текст в поле ввода сообщения, вызывая метод `setText()` с пустой строкой в качестве параметра (можно было бы также использовать метод `clear()`).

Напомню, что согласно установленным сигнально-слотовым связям слот `slotSendToServer()` вызывается после нажатия кнопки **Send** (Послать) или клавиши <Enter> в однострочном текстовом поле (указатель `m_ptxtInput()`).

Листинг 39.12. Метод `slotSentToServer()` (файл MyClient.cpp)

```
void MyClient::slotSendToServer()
{
    QByteArray  arrBlock;
    QDataStream out(&arrBlock, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_5_2);
    out << quint16(0) << QTime::currentTime() << m_ptxtInput->text();

    out.device()->seek(0);
    out << quint16(arrBlock.size() - sizeof(quint16));

    m_pTcpSocket->write(arrBlock);
    m_ptxtInput->setText("");
}
```

Как только связь с сервером будет установлена, вызывается метод `slotConnected()`. Этот слот в виджет текстового поля добавляет строку сообщения (листинг 39.13).

Листинг 39.13. Метод `slotConnected()` (файл MyClient.cpp)

```
void MyClient::slotConnected()
{
    m_ptxtInfo->append("Received the connected() signal");
}
```

Реализация UDP-сервера и UDP-клиента

При реализации UDP-сервера нужно начать с создания объекта класса `QUdpSocket` (потом вы сможете методом `writeDatagram()` задействовать его для записи в сокет дейтаграмм). Затем мы реализуем клиента, используя все тот же класс `QUdpSocket`, но соединяем его

с портом при помощи метода `bind()`. Всякий раз, как дейтаграмма поступает на порт, к которому подсоединен сокет, происходит отправка сигнала `readyRead()`. Считывать прибывшую дейтаграмму вы можете методом `readDatagram()`.

Клиент и сервер полностью независимы друг от друга — как только сервер начнет делать пересылку, клиент станет получать данные. Серверу абсолютно не важно, активны клиенты или нет.

Продемонстрируем создание простого UDP-сервера и UDP-клиента на примере, приведенном в листингах 39.14–39.19. В этом примере сервер показывает и передает текущую дату и время, а клиент ее получает и отображает (рис. 39.4).

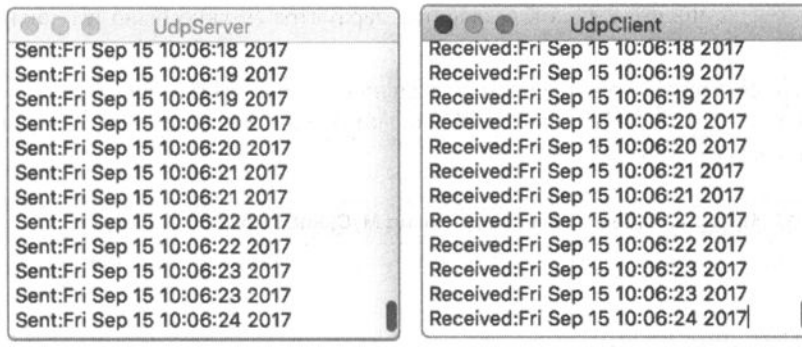


Рис. 39.4. Взаимодействие UDP-клиента с UDP-сервером

Сначала реализуем класс сервера. В определении класса `UdpServer` нам понадобится объект сокета `QUdpSocket` (листинг 39.14), поэтому мы объявляем указатель на него: `m_pudp`. Далее нам нужен слот для пересылки нашей дейтаграммы: `slotSendDatagram()`.

Листинг 39.14. Определение класса `UdpServer` (фрагмент файла UdpServer.h)

```
class UdpServer : public QTextEdit {
Q_OBJECT
private:
    QUdpSocket* m_pudp;

public:
    UdpServer(QWidget* pwgt = 0);

private slots:
    void slotSendDatagram();
};
```

Для того чтобы мы могли отличить приложение сервера от клиента, в конструкторе класса `UdpServer` (листинг 39.15) мы задаем заголовок окна методом `setWindowTitle()` и создаем объект сокета `QUdpSocket`. Пересылка дейтаграмм будет осуществляться через определенные интервалы времени при помощи таймера, поэтому мы создаем таймер, устанавливаем в нем интервал, равный 0,5 сек, запускаем его вызовом метода `start()` и соединяем со слотом пересылки дейтаграмм `slotSenDatagram()`.

Листинг 39.15. Конструктор UdpServer (файл UdpServer.cpp)

```
UdpServer::UdpServer(QWidget* pwgt /*=0*/) : QTextEdit(pwgt)
{
    setWindowTitle("UdpServer");

    m_pudp = new QUdpSocket(this);

    QTimer* ptimer = new QTimer(this);
    ptimer->setInterval(500);
    ptimer->start();
    connect(ptimer, SIGNAL(timeout()), SLOT(slotSendDatagram()));
}
```

В листинге 39.16 реализован метод `slotSendDatagram()`, который формирует и пересылает дейтаграммы. Наша дейтаграмма будет содержать только текущую дату и время. Воспользуемся классом потока `QDataStream` для помещения данных в объект `baDatagram`. Пересылка дейтаграмм осуществляется при помощи метода `writeDatagram()`, в который мы первым параметром передаем данные дейтаграммы `baDatagram`, а вторым и третьим параметрами — IP-адрес и номер порта партнера. В нашем случае мы не указываем IP-адрес, а используем константу `QHostAddress::LocalHost`, которая имеет строковое значение локального хоста 127.0.0.1.

> ***Определение IP-адреса по имени хоста***
>
> В отличие от класса `QAbstractSocket`, класс `QUdpSocket` не может работать с именами хостов и принимает только номера IP-адресов. Если вам понадобится определить IP-адрес по имени хоста, то можно воспользоваться статическим методом `fromName()` класса `QHostName`.

Листинг 39.16. Слот slotSendDatagram() (файл UdpServer.cpp)

```
void UdpServer::slotSendDatagram()
{
    QByteArray baDatagram;
    QDataStream out(&baDatagram, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_5_3);
    QDateTime dt = QDateTime::currentDateTime();
    append("Sent:" + dt.toString());
    out << dt;
    m_pudp->writeDatagram(baDatagram, QHostAddress::LocalHost, 2424);
}
```

На этом реализация нашего сервера закончена.

Теперь приступим к реализации UDP-клиента (листинг 39.17). Так же как и сервер, клиент будет содержать объект класса `QUdpSocket`, для этого мы объявляем указатель `m_pudp`. Объявляем слот `slotProcessDatagrams()`, предназначенный для получения дейтаграмм от сервера.

Листинг 39.17. Реализация UDP-клиента (файл UdpClient.h)

```
class UdpClient : public QTextEdit {
Q_OBJECT
```

```
private:
    QUdpSocket* m_pudp;

public:
    UdpClient(QWidget* pwgt = 0);

private slots:
    void slotProcessDatagrams();
};
```

Для того чтобы отличать окно клиента от окна сервера, в конструкторе для окна приложения клиента задаем надпись "UdpClient" (листинг 39.18). Создаем объект сокета QUdpSocket и при помощи метода bind() привязываем его к порту 2424. Поскольку мы не указали IP-адрес хоста, то для получения данных по умолчанию сокет будет использовать локальный хост. Для извлечения и отображения полученных дейтаграмм соединяем сигнал сокета readyRead() со слотом slotProcessDatagrams().

Листинг 39.18. Конструктор UdpClient (файл UdpClient.cpp)

```
UdpClient::UdpClient(QWidget* pwgt /*=0*/) : QTextEdit(pwgt)
{
    setWindowTitle("UdpClient");

    m_pudp = new QUdpSocket(this);
    m_pudp->bind(2424);
    connect(m_pudp, SIGNAL(readyRead()), SLOT(slotProcessDatagrams()));
}
```

В листинге 39.19 объект класса QUdpSocket при получении дейтаграмм ставит в очередь поступившие дейтаграммы и дает возможность последовательного доступа к ним в порядке очередности. Сама же очередь обычно состоит только из одной дейтаграммы, но никогда нельзя исключать возможность и того, что отправитель может последовательно передать сразу несколько дейтаграмм. Поэтому и нужен цикл do...while(), так как в этом случае необходимо проигнорировать все дейтаграммы, кроме самой последней, поскольку именно она и содержит самые актуальные данные. В теле цикла при помощи метода pendingDatagramSize() мы узнаем размер ждущей обработки дейтаграммы. Дейтаграммы всегда высылаются одним блоком, а это значит, что при любом размере дейтаграммы ее нужно считывать целиком. Следует учитывать, что если размер буфера окажется недостаточным, то данные будут обрезаны. Поэтому в соответствии с полученным размером дейтаграммы мы изменяем размер буфера для считывания дейтаграмм и вызовом метода readDatagram() копируем в буфер данные baDatagram. После завершения цикла считываем в поток in дату и время, полученные из дейтаграммы, и отображаем их в окне. Для этого вызываем метод QTextEdit::append().

Листинг 39.19. Слот slotProcessDatagrams() (файл UdpClient.cpp)

```
void UdpClient::slotProcessDatagrams()
{
    QByteArray baDatagram;
```

```
    do {
        baDatagram.resize(m_pudp->pendingDatagramSize());
        m_pudp->readDatagram(baDatagram.data(), baDatagram.size());
    } while(m_pudp->hasPendingDatagrams());

    QDateTime dateTime;
    QDataStream in(&baDatagram, QIODevice::ReadOnly);
    in.setVersion(QDataStream::Qt_5_3);
    in >> dateTime;
    append("Received:" + dateTime.toString());
}
```

Управление доступом к сети

Для того чтобы использовать классы высокого уровня, такие как `QHttp` и `QFtp`, необходимо понимание концепций, которые стоят за ними. Идея реализации нового механизма управления доступом к сети появилась вследствие необходимости упрощения взаимодействия с сетью и реализации классов еще более высокого уровня. Кроме того, созданный механизм обладает поддержкой «куки» (cookie), «прокси» (proxy), кэширования данных, аутентификации и одновременной пересылкой запросов.

Состоит он из трех основных классов: `QNetworkAccessManager`, `QNetworkRequest` и `QNetworkReply`.

Класс `QNetworkAccessManager` — это центр всего. Он поддерживает такие операции, как:

- `head` — получение статуса;
- `get` — загрузка данных;
- `put` — пересылка данных;
- `post` — это гибрид `get` и `put`, предназначен только для HTTP.

Для каждой операции есть одноименный метод. Сам класс управляет целой очередью запросов.

Класс `QNetworkRequest` содержит один подлежащий отправке запрос, основной его компонент — URL. Этот компонент содержит метаданные для запроса, такие как, например, HTTP-заголовок и другие опции. Для хранения и работы с URL библиотека Qt предоставляет класс `QUrl`.

Класс `QNetworkReply` содержит данные ответа, метаданные URL, HTTP-заголовок и статус шифрования, условные ошибки и т. д. Объекты этого класса отправляют сигналы:

- `readyRead()` — при поступлении данных;
- `finished()` — при завершении;
- `error()` — если возникли проблемы;
- `downloadProgress()` и `uploadProgress()` — сигналы процесса.

Проще назначение классов `QNetworkRequest` и `QNetworkReplay` можно описать так: класс `QNetworkRequest` — это данные запроса к удаленному серверу, а `QNetworkReply` — это ответ от сервера.

Проиллюстрируем все сказанное простым примером реализации класса, предназначенного для загрузки файлов с HTTP, и снабдим наше приложение (листинги 39.20–39.31) пользовательским интерфейсом (рис. 39.5), который будет реализован отдельно.

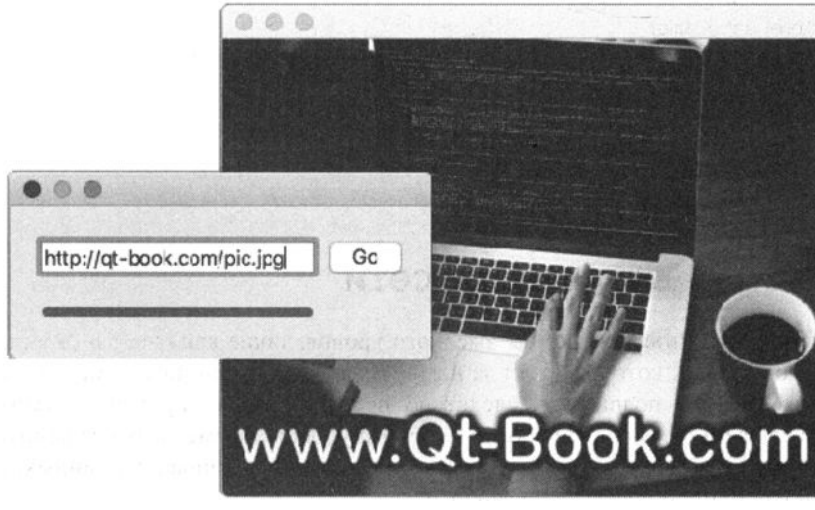


Рис. 39.5. Файловый загрузчик

В основном файле (листинг 39.20) приложения мы просто создаем виджет класса `DownloaderGui` и делаем его видимым вызовом метода `show()`.

Листинг 39.20. Загрузка файлов с HTTP (файл main.cpp)

```
#include <QApplication>
#include "DownloaderGui.h"

int main(int argc, char** argv)
{
    QApplication  app(argc, argv);
    DownloaderGui downloader;

    downloader.show();
    downloader.resize(280, 100);

    return app.exec();
}
```

Наш класс загрузчика, показанный в листинге 39.21, унаследован от класса `QObject` и содержит объект класса `QNeworkAccessManager`, являющийся основным звеном в механизме управления доступом к сети. Метод `download()` выполняет загрузку ресурса, заданного в параметре класса `QUrl`. Этот класс, как следует из его имени, представляет собой URL-ресурс. Далее идут сигналы, которые уведомляют о происходящем в процессе загрузки: `downloadProgress()`, `done()` и `error()`. Слот `slotFinished()` будет вызываться в конце загрузки и отправлять сигналы `done()` или `error()`.

Листинг 39.21. Определение класса Downloader (файл Downloader.h)

```
#pragma once

#include <QObject>
#include <QUrl>

class QNetworkAccessManager;
class QNetworkReply;

// ============================================================================
class Downloader : public QObject {
Q_OBJECT

private:
    QNetworkAccessManager* m_pnam;

public:
    Downloader(QObject* pobj = 0);

    void download(const QUrl&);

signals:
    void downloadProgress(qint64, qint64                );
    void done            (const QUrl&, const QByteArray&);
    void error           (                              );

private slots:
    void slotFinished(QNetworkReply*);
};
```

В конструкторе класса (листинг 39.22) мы создаем объект класса `QNetworkAccessManager` и соединяем его сигнал `finished()` со слотом `slotFinished()`. Этот сигнал отправляется по окончании загрузки и передает указатель на объект `QNetworkReply`, соответствующий текущей проведенной операции.

Листинг 39.22. Конструктор Downloader (файл Downloader.cpp)

```
Downloader::Downloader(QObject* pobj/*=0*/) : QObject(pobj)
{
    m_pnam = new QNetworkAccessManager(this);
    connect(m_pnam, SIGNAL(finished(QNetworkReply*)),
            this,   SLOT(slotFinished(QNetworkReply*))
           );
}
```

В методе загрузки `download()`, приведенном в листинге 39.23, мы создаем объект `QNetworkRequest`, который будет являться входным объектом нашего запроса для `QNetworkAccessManager`.

Задание приоритетов

Объектам класса `QNetworkRequest` можно при необходимости задавать приоритеты. Это даст возможность запросам с более высокими приоритетами быть обработанными в первую очередь. Приоритеты запросов устанавливаются вызовом метода `QNetworkRequest::setPriority()`.

Вот как бы мог выглядеть в программе запрос с высоким приоритетом:

```
request.setPriority(QNetworkRequest::HighPriority);
```

Вызов метода `get()` осуществляет выполнение нашего запроса. Этот метод вызывается асинхронно, то есть его исполнение происходит мгновенно еще до завершения самой операции, и возвращает указатель на объект класса `QNetworkReply`, которым мы воспользуемся для того, чтобы уведомлять о происходящем процессе загрузки, для чего соединим его с собственным сигналом `downloadProgress()`.

Листинг 39.23. Метод `download()` (файл Downloader.cpp)

```
void Downloader::download(const QUrl& url)
{
    QNetworkRequest request(url);
    QNetworkReply*  pnr = m_pnam->get(request);
    connect(pnr, SIGNAL(downloadProgress(qint64, qint64)),
            this, SIGNAL(downloadProgress(qint64, qint64))
           );
}
```

Слот `slotFinished()` вызывается по завершении операции загрузки (листинг 39.24). Нам нужно проверить, прошла загрузка успешно или нет, поэтому мы проверяем статус ошибок вызовом метода `error()` и, если статус не равен `QNetworkReply::NoError`, то есть произошла ошибка, отправляем сигнал `error()`. В другом же случае, то есть когда все прошло успешно, отправляем сигнал `done()`. В этом сигнале мы передаем использованный для загрузки URL и полученные данные. Сами данные мы считываем из объекта `QNetworkReply` при помощи метода `readAll()`. И в завершение удаляем объект класса `QNetworkReply`, так как он выполнил свое назначение и нам больше не нужен. Удаление осуществляем вызовом из объекта метода `deleteLater()`, поскольку этот способ более безопасный, чем непосредственное удаление при помощи оператора `delete` языка C++ (в случае вызова метода `deleteLater()` само удаление объекта произойдет не сразу, а только при следующем прохождении цикла событий). Ну вот, теперь наш класс загрузчика полностью готов, и можно переходить к реализации для него графического интерфейса.

Листинг 39.24. Слот `slotFinished()` (файл Downloader.cpp)

```
void Downloader::slotFinished(QNetworkReply* pnr)
{
    if (pnr->error() != QNetworkReply::NoError) {
        emit error();
    }
    else {
        emit done(pnr->url(), pnr->readAll());
    }
    pnr->deleteLater();
}
```

Наш пользовательский интерфейс для загрузчика, как видно из рис. 39.5 и листинга 39.25, состоит из виджета индикации процесса загрузки (указатель `m_ppb`), поля ввода ссылки (указатель `m_ptxt`) и кнопки исполнения (указатель `m_pcmd`). Кроме того, мы также нуждаемся в объекте нашего «свежеиспеченного» класса `Downloader` (указатель `m_dl`). Закрытый метод `showPic()` мы определили для автоматического показа загруженных растровых изображений. Далее идут слоты:

- `slotGo()` — осуществляет исполнение загрузки ресурса;
- `slotError()` — вызывается при возникновении ошибки;
- `slotDownloadProgress()` — отображает процесс загрузки, используя для этого виджет индикации процесса;
- `slotDone()` — выполняет конечные действия по окончании самой загрузки.

Листинг 39.25. Определение класса `DownloaderGui` (файл DownloaderGui.h)

```
#pragma once

#include <QWidget>
#include <QUrl>

class Downloader;
class QProgressBar;
class QLineEdit;
class QPushButton;

// ============================================================================
class DownloaderGui : public QWidget {
Q_OBJECT

private:
    Downloader*   m_pdl;
    QProgressBar* m_ppb;
    QLineEdit*    m_ptxt;
    QPushButton*  m_pcmd;

    void showPic(const QString&);
public:
    DownloaderGui(QWidget* pwgt = 0);

private slots:
    void slotGo              (                                  );
    void slotError           (                                  );
    void slotDownloadProgress(qint64, qint64                    );
    void slotDone            (const QUrl&, const QByteArray&);
};
```

В конструкторе, показанном в листинге 39.26, мы создаем виджеты нашего графического интерфейса, а также и сам объект загрузчика `Downloader`. Вызовом метода `setText()` инициализируем поле ввода строкой гиперссылки `strDownloadLink`, которая будет находиться

там по умолчанию при запуске программы. Осуществляем соединение кнопки со слотом запуска загрузки `slotGo()`, а также сигнала оповещения процесса `downladProgress()` и сигнала завершения загрузки объекта класса `Downloader` со слотами `slotDownloadProgress()` и `slotDone()`. Все элементы пользовательского интерфейса размещаем при помощи табличного размещения `QGridLayout`.

Листинг 39.26. Конструктор `DownloaderGui` (файл DownloaderGui.cpp)

```
DownloaderGui::DownloaderGui(QWidget* pwgt /*=0*/) : QWidget(pwgt)
{
    m_pdl  = new Downloader(this);
    m_ppb  = new QProgressBar;
    m_ptxt = new QLineEdit;
    m_pcmd = new QPushButton(tr("&Go"));

    QString strDownloadLink = "http://qt-book.com/pic.jpg";
    m_ptxt->setText(strDownloadLink);

    connect(m_pcmd, SIGNAL(clicked()), SLOT(slotGo()));
    connect(m_pdl, SIGNAL(downloadProgress(qint64, qint64)),
            this,  SLOT(slotDownloadProgress(qint64, qint64))
           );
    connect(m_pdl, SIGNAL(done(const QUrl&, const QByteArray&)),
            this,  SLOT(slotDone(const QUrl&, const QByteArray&))
           );

    QGridLayout* pLayout = new QGridLayout;
    pLayout->addWidget(m_ptxt, 0, 0);
    pLayout->addWidget(m_pcmd, 0, 1);
    pLayout->addWidget(m_ppb, 1, 0, 1, 1);
    setLayout(pLayout);
}
```

В листинге 39.27 показана реализация слота начала загрузки. Это всего одна строчка кода, с помощью которой мы передаем в метод `download()` содержимое текстового поля (метод `text()`) — то, что набрал пользователь, и запускаем тем самым процесс загрузки.

Листинг 39.27. Слот `slotGo()` (файл DownloaderGui.cpp)

```
void DownloaderGui::slotGo()
{
    m_pdl->download(QUrl(m_ptxt->text()));
}
```

В листинге 39.28 после получения данных о текущем процессе мы первым делом проверяем, чтобы переменная, которая содержит размер сгружаемого файла `nTotal`, не была равна нулю или меньше его. Это очень важно, потому что для вычисления хода процесса в процентах эта переменная будет использоваться в качестве делителя, и ее нулевое значение приведет к аварийному завершению приложения. Возникновение нулевого значения воз-

можно, например, в том случае, когда у пользователя нет доступа к Интернету, и он делает попытку загрузки. В подобных случаях мы просто вызываем слот `slotError()`, который отобразит сообщение об ошибке, и осуществляем выход. Если же все в порядке, то мы вычисляем значение хода процесса в процентах и вызовом метода `setValue()` устанавливаем его в виджете индикатора выполнения.

Листинг 39.28. Слот `slotDownloadProgress()` (файл DownloaderGui.cpp)

```
void DownloaderGui::slotDownloadProgress(qint64 nReceived, qint64 nTotal)
{
    if (nTotal <= 0) {
        slotError();
        return;
    }
    m_ppb->setValue(100 * nReceived / nTotal);
}
```

Слот `slotDone()` вызывается по завершении загрузки (листинг 39.29). В него мы получаем гиперссылку, или, иначе, ссылку (link), которая была использована для загрузки (переменная `url`), и сами данные (переменная `ba`). Эти данные мы просто записываем в файл в каталог пользователя, где он хранит фотографии. Именем этого файла становится последняя секция ссылки, которая является именем получаемого от сервера файла. Далее мы проверяем, является ли файл растровым изображением в формате JPG или PNG, и, если это так, вызываем метод для отображения растровых изображений `showPic()`, в который передаем имя записанного файла.

Листинг 39.29. Слот `slotDone()` (файл DownloaderGui.cpp)

```
void DownloaderGui::slotDone(const QUrl& url, const QByteArray& ba)
{
    QString strFileName =
      QStandardPaths::writableLocation(QStandardPaths::PicturesLocation)
      + "/" + url.path().section('/', -1);
    QFile file(strFileName);
    if (file.open(QIODevice::WriteOnly)) {
        file.write(ba);
        file.close();

        if (strFileName.endsWith(".jpg")
            || strFileName.endsWith(".png")
           ) {
            showPic(strFileName);
        }
    }
}
```

Отображение растровых файлов (листинг 39.30) мы выполняем в миниатюре, поэтому осуществляем уменьшение исходного изображения в 2 раза при помощи метода изменения размера `scaled()` в режиме сглаживания `Qt::SmoothTransforamtion`. Полученное изображе-

ние вызовом метода `setPixmap()` устанавливаем в виджете надписи `QLabel` и задаем неизменяемые размеры методом `setFixedSize()`. После чего осуществляем показ виджета надписи при помощи метода `show()`.

Листинг 39.30. Метод `showPic()` (файл DownloaderGui.cpp)

```
void DownloaderGui::showPic(const QString& strFileName)
{
    QPixmap pix(strFileName);
    pix = pix.scaled(pix.size() / 2,
                     Qt::IgnoreAspectRatio,
                     Qt::SmoothTransformation
                    );
    QLabel* plbl = new QLabel;
    plbl->setPixmap(pix);
    plbl->setFixedSize(pix.size());
    plbl->show();
}
```

Слот, представленный в листинге 39.31, просто показывает диалоговое окно с сообщением об ошибке.

Листинг 39.31. Слот `slotError()` (файл DownloaderGui.cpp)

```
void DownloaderGui::slotError()
{
    QMessageBox::critical(0,
                          tr("Error"),
                          tr("An error while download is occured")
                         );
}
```

Блокирующий подход

Мы уже знаем, что все методы, связанные с запросами и получением данных, выполняются асинхронно, не блокируя ход основного потока программы. Такой подход является в подавляющем большинстве желательным. Но иногда возникает необходимость в блокирующем подходе. Например, без полученных данных приложение не может продолжать работу, и поэтому пользователь должен дождаться получения данных.

Вы можете ждать до полного завершения операции и блокировать текущий поток посредством вызова одного из методов, начинающегося с `waitFor...()`. Большинство этих методов определено в классе `QAbstractSocket` и предназначено для того, чтобы:

- `waitForConnected()` — дождаться соединения;
- `waitForReadyRead()` — дождаться поступления данных;
- `waitForBytesWritten()` — дождаться, когда данные будут записаны;
- `waitForDisconnected()` — дождаться отсоединения;
- `waitForEncrypted()` — дождаться шифрования данных (только для `QSslSocket`).

Все указанные методы принимают один параметр, который задает время ожидания. По умолчанию время ожидания равно 30 сек.

Программа TCP-сервера с использованием блокирующего подхода может выглядеть так, как показано в листинге 39.32.

Листинг 39.32. TCP-сервер с применением блокирующего подхода

```
int main(int argc, char** argv)
{
    QCoreApplication app(argc, argv);
    QTcpServer       tcpServer;
    int              nPort = 2424;

    if (!tcpServer.listen(QHostAddress::Any, nPort)) {
        qDebug() << "Can't listen on port: " << nPort;
        return 0;
    }

    forever {
        while (tcpServer.waitForNewConnection(60000)) {
        do {
            QTcpSocket* pSocket = tcpServer.nextPendingConnection();
            QString strDateTime =
                QDateTime::currentDateTime()
                  .toString("yyyy.MM.dd hh:mm:ss");
                pSocket->write(strDateTime.toLatin1());
                pSocket->flush();
                qDebug() << "Server date & time:" + strDateTime;
                pSocket->disconnectFromHost();
                if (pSocket->state() == QAbstractSocket::ConnectedState){
                    pSocket->waitForDisconnected();
                }
                delete pSocket;
            } while (tcpServer.hasPendingConnections());
        }
    }

    return 0;
}
```

Здесь мы реализуем сервер, который возвращает подключаемым к нему клиентам текущую дату и время компьютера, на котором он запущен. Мы создаем объект класса `QTcpServer` и указываем ему, что он должен «слушать» входящие соединения от клиентов, находящихся на любом компьютере на порту 2424.

Запуск приложения сервера в Mac OS X

В Mac OS X некоторые порты, например порт 80 (HTTP), можно занимать только с правами суперпользователя. В подобных случаях запускайте ваше приложение сервера посредством команды `sudo`.

Мы запускаем циклы forever и while и устанавливаем время ожидания для соединения равным одной минуте, а это означает, что если в течение минуты запросов на соединение от клиентов получено не будет, то мы станем повторять эту попытку до бесконечности. При возникновении соединения мы получаем сокет для коммуникации вызовом метода nextPendingConnection(), а его адрес присваиваем указателю pSocket. Затем переводим в строку текущую дату и записываем ее в строковый объект strDateTime, а после этого записываем его с помощью вызова метода write() в сокет. После вызова метода write() мы вызываем метод flush() — он сообщает о том, что мы закончили запись данных, и сокет должен взять все переданные ему данные. На этом работа по передаче данных закончена, и мы выполняем отсоединение от клиента, для чего вызываем метод disconnectFromHost() и ждем, пока отсоединение произойдет. Ожидание осуществляется вызовом метода waitForDisconnected(). Иногда отсоединение происходит мгновенно, и выполнение метода waitForDisconnected() может привести к отображению на консоли сообщения об ошибке, которое будет выглядеть следующим образом:

```
QAbstractSocket::waitForDisconnected() is not allowed in UnconnectedState
```

Для того чтобы этого не произошло, мы проверяем текущий статус соединения, и только если оно соответствует значению QAbstractSocket::ConnectedState, то есть соединение еще существует, лишь тогда вызываем метод waitForDisconnected() и ждем его окончания. После этого оператором delete уничтожаем созданный объект сокета. И если на сервере есть еще один клиент, который стоит в очереди, ожидая соединения, то мы выполняем в цикле снова все те же операции, которые проделали с предыдущим клиентом. Если же ожидающего клиента нет, сервер сам ждет клиента на протяжении минуты, и если в течение этого времени он не появится, то продолжаем следующий виток нашего вечного цикла forever.

Наш TCP-клиент, показанный в листинге 39.33, выглядит просто. Мы создаем объект сокета (socket) и методом connectToHost() осуществляем попытку соединения с сервером. Значение первого параметра QHostAddress::LocalHost означает, что сервер должен быть запущен на том же компьютере, на котором запущен клиент. Второй параметр задает значение порта, на котором должно осуществиться соединение. Далее с помощью вызова метода waitForDisconnected() ожидаем момента, когда сервер осуществит отсоединение. После чего мы просто считываем и отображаем данные из сокета.

Листинг 39.33. TCP-клиент с применением блокирующего подхода

```
#include <QtCore>
#include <QtNetwork>

int main(int argc, char** argv)
{
    QCoreApplication app(argc, argv);
    QTcpSocket       socket;

    socket.connectToHost(QHostAddress::LocalHost, 2424);
    socket.waitForDisconnected();
    qDebug() << socket.readAll();

    return 0;
}
```

Класс `QNetworkAccessManager` не предоставляет возможности для использования методов `waitFor...()`. Но если блокирующий подход необходим, то можно прибегнуть к использованию объекта цикла событий. Мы могли бы изменить метод реализованного нами в листинге 39.25 загрузчика таким образом, чтобы метод `download()` не исполнялся асинхронно, а блокировал бы исполнения основного потока до тех пор, пока не будут получены все данные полностью:

```
void Downloader::download(const QUrl& url)
{
    QNetworkRequest request(url);
    QNetworkReply*  pnr = m_pnam->get(request);
    QEventLoop     loop;

    connect(&request, SIGNAL(finished()), &loop, SLOT(quit()));
    loop.exec();
}
```

Режим прокси

В программах, работающих с сетью, очень часто требуется предусмотреть возможность того, чтобы пользователь мог установить соединение с сетью через прокси-сервер. Эта возможность нужна не только для анонимности, но так же связана с тем, что в большинстве фирм, из соображений безопасности сети, нет другой возможности выхода в Интернет, как только через прокси-сервер предприятия. Для установки прокси-сервера можно воспользоваться классом `QNetworkProxy`. Пример его использования показан в листинге 39.34.

Листинг 39.34. Установка прокси-сервера

```
QNetworkProxy proxy;
proxy.setType(QNetworkProxy::HttpProxy);
proxy.setHostName("192.168.178.1");
proxy.setPort(8080);
proxy.setUser("user");
proxy.setPassword("password");
QNetworkProxy::setApplicationProxy(proxy);
```

Здесь мы создаем объект класса `QNetworkProxy` и устанавливаем тип, IP-адрес, порт, имя пользователя и пароль доступа к прокси-серверу. В завершение вызовом метода `setApplicationProxy()` устанавливаем созданный объект глобально для всего приложения.

Информация о хосте

Под хостом понимается любой компьютер, а также сервер, подключенный к локальной либо глобальной сети. Библиотека Qt предоставляет класс `QHostInfo`, который служит для получения информации о хосте. Благодаря ему, вы можете по имени хоста узнать, например, его IP-адрес и наоборот. Вот как это может выглядеть в виде программного кода:

```
QHostInfo::lookupHost(QStringLiteral("qt-book.com"),
                      this,
                      SLOT(slotPrintResults(QHostInfo))
                     );
```

Сам же слот для отображения информации можно реализовать следующим образом:

```
void HostInfoExample::slotPrintResults(QHostInfo info)
{
    qDebug() << "HostName:" << info.hostName();
    qDebug() << "Host IP:" << "Adrs:" << info.addresses();
}
```

Есть ли соединение с Интернетом?

Очень часто, перед тем как выполнять какие-либо операции, возникает потребность убедиться в том, что приложение имеет доступ к Интернету. Для этой цели существует класс `QNetworkConfigurationManager`. Проверить, существует ли соединение с Интернетом, можно вызовом из объекта этого класса метода `isOnline()`. Этот метод вернет значение `true` в том случае, если соединение с Интернетом есть, и `false` в случае, если его нет. Если же необходимо, чтобы приложение было проинформировано каждый раз при изменениях соединения с Интернетом, то можно подсоединить слот к сигналу `onlineStateChanged()`, высылаемые значения которого соответствуют значению метода `isOnline()`.

Резюме

В этой главе мы познакомились с протоколами TCP и UDP. Их основным различием является то, что первый — это протокол для передачи данных с установлением соединения, а второй — без установления соединения. Кроме того, протокол UDP не дает гарантии доставки пакетов получателю (то есть менее надежен), однако в силу этого обладает более высокой скоростью доставки данных, чем TCP.

Сокетные соединения — это стандартный механизм обмена данными через сеть в обоих направлениях. Каждому сокету соответствует пара значений: сетевой адрес и номер порта.

Сценарий «клиент-сервер» выглядит следующим образом: сервер занимает определенный порт, по которому он предоставляет свои услуги, и начинает ожидать поступления запросов от клиентов через этот порт. Чтобы подключиться к серверу, клиент должен знать его адрес и номер порта. Для соединения с сервером клиент должен создать сокет.

Для упрощения стандартных операций при работе с сетью Qt предоставляет универсальный класс `QNetworkAccessManager`.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/39-510/** или с помощью следующего QR-кода (рис. 39.6):

Рис. 39.6. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 40

Работа с XML

> Все воистину мудрые мысли были обдуманы уже тысячи раз, но чтобы они стали по-настоящему вашими, нужно честно обдумывать их еще и еще, пока они не укоренятся в вашем мозгу.
>
> *Гёте*

В настоящее время формат XML (Extensible Markup Language, расширяемый язык разметки) — одна из самых активно используемых технологий. Зайдя в книжный магазин вы, наверное, поразитесь количеству книг, посвященных XML. С распространением Интернета обмен данными между разными платформами стал необходимостью для работающих с данными программ. Это и послужило причиной создания XML.

Разработка формата XML началась в 1996 году, и в 1998 году организацией W3C (World Wide Web Consortium) был принят первый его стандарт. На самом деле, XML не представляет собой ничего нового и является упрощенной версией языка SGML, разработанного в начале 1980-х. Создатели XML переняли из него — с учетом опыта языка HTML — все самое лучшее и создали нечто, по своей мощности не уступающее SGML, но вместе с тем гораздо более удобное и простое как для понимания, так и для использования.

XML не имеет лицензии, что позволяет бесплатно использовать этот формат в своих программах. Всеобщая поддержка XML исключает зависимость его от отдельной фирмы или платформы, оказывающей поддержку для XML. Поскольку XML-документ представляет собой текст в формате ASCII или Unicode, то он является читабельным и для человека. Соответственно, XML-документ может быть изменен при помощи простых текстовых редакторов — например, программой Notepad (Блокнот) из стандартного набора ОС Windows.

Основные понятия и структура XML-документа

XML — это средство хранения структурированных данных в текстовом файле. Примером структурированных данных являются: адресная книга, генеалогическое древо, информация о продуктах и т. п. Язык XML очень похож на HTML. XML описывает структуру самого документа без детализации его отображения. Следует, впрочем, заметить, что XML гораздо строже, чем HTML, — например, при ошибке спецификация XML запрещает приложению, работающему с XML-документом, предпринимать попытку его корректировки. В подобных случаях приложение должно прекратить считывание дефектного файла и сообщить об ошибке.

Упрощенный вариант XML-документа, который представляет собой структуру адресной книги, предназначенную для хранения имен, телефонов и адресов электронной почты, мог бы выглядеть следующим образом:

```
<?xml version = "1.0"?>
<!-- My Address Book -->
<addressbook>
  <contact number = "1">
      <name>Piggy</name>
      <phone>+49 631322187</phone>
      <email>piggy@mega.de</email>
  </contact>
  <contact number = "2">
      <name>Kermit</name>
      <phone>+49 631322181</phone>
      <email>kermit@mega.de</email>
  </contact>
</addressbook>
```

Любой XML-документ начинается с заголовка, говорящего о принадлежности к этому формату и указывающего номер версии. Сам XML-документ состоит из множества *элементов* (elements). Имена элементов заключаются между символами < и >, образуя *теги*. Теги нужны для обозначения границ элемента. Начало элемента обозначается открывающим тегом, например `<name>`, а конец — закрывающим, в данном случае `</name>`. Между этими тегами может находиться содержание, например `<name>Piggy</name>`, но это необязательно. Иногда встречаются элементы, не имеющие содержания, например `<empty></empty>`. Для подобных случаев спецификация XML предусматривает сокращенную форму — так, предыдущую запись можно заменить на `<empty/>`. Может показаться, что подобные теги не могут содержать информацию, но это не так — информацию можно сохранить при помощи атрибутов, следующих за именем элемента, например `<empty number = "1"></empty>` или `<empty number = "1"/>`. Кроме того, информация содержится в самом наличии или отсутствии пустого тега.

В XML-документ можно вставлять *комментарии*, которые представляют собой от одной до нескольких строк текста, ограниченных тегами `<!--` и `-->`.

Основные отличия тегов HTML от тегов XML заключаются в следующем:

- теги используются только для сохранения информации, а интерпретация этих данных целиком лежит на приложении, которое их считывает. Например, в XML тег `<p>` необязательно является тегом параграфа, как в HTML, а может обозначать, например, `properties` (свойства);
- для описания документов можно использовать теги с любыми подходящими названиями. Следует учитывать, что строчные и прописные буквы в именах различаются, — так, например `<Tag></Tag>` и `<tag></tag>` являются разными тегами.

На рис. 40.1 показана часть графического эквивалента структуры XML. Обратите внимание, что элементы XML-документа задают иерархическую структуру в виде дерева.

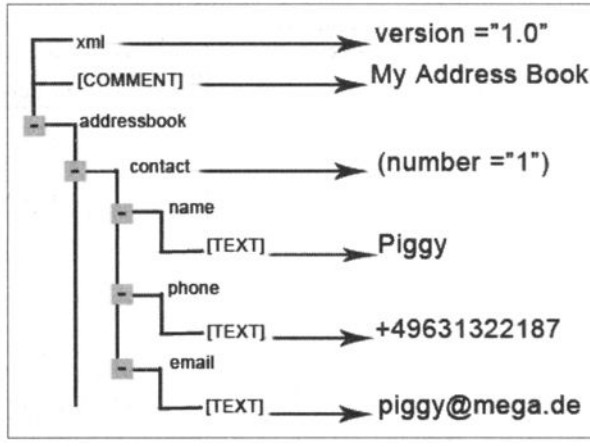


Рис. 40.1. XML-документ

XML и Qt

Библиотека Qt очень активно использует формат XML. Например, программа Qt Designer (см. *главу 44*) сохраняет файлы пользовательского интерфейса именно в этом формате. Также он используется утилитами Qt, предназначенными для интернационализации приложений (см. *главу 30*).

Поддержка XML в Qt — это отдельный модуль `QtXml`, для использования которого необходимо указать его имя в проектном файле. Сделать это несложно — нужно просто добавить туда следующую строку:

```
QT += xml
```

А для того чтобы работать с классами этого модуля, необходимо включить заголовочный метафайл QtXml:

```
#include <QtXml>
```

Qt предоставляет три возможности использования XML. Первая называется DOM, вторая — SAX, а третью реализует класс `QXmlStreamReader`. Нельзя однозначно сказать, что одна из них лучше другой, поскольку каждая имеет свои преимущества.

Работа с DOM

DOM (Document Object Model, объектная модель документа) — это стандартный API для анализа XML-документов, разработанный W3C. Qt поддерживает второй уровень реализации, следующий рекомендациям W3C и включающий в себя поддержку *пространства имен* (name spaces). Самое большое преимущество DOM состоит в возможности представления XML-документа в виде древовидной структуры в памяти компьютера. Цена этого удобства очевидна — большой расход памяти. Но если на компьютере, где запускается ваша программа, нет недостатка в оперативной памяти, то использование DOM станет наиболее подходящим решением.

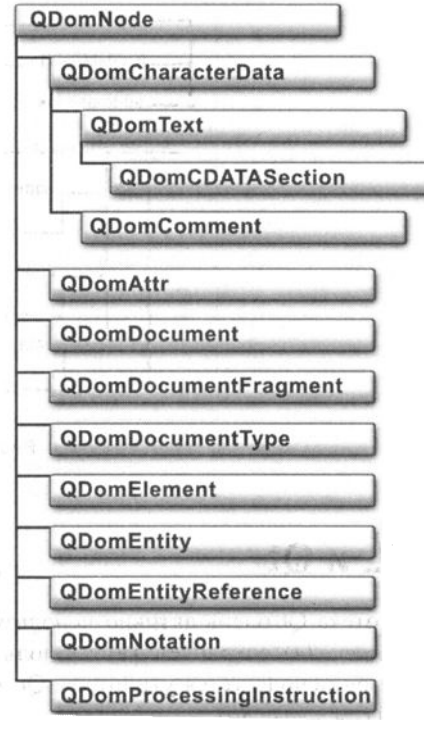


Рис. 40.2. Иерархия классов для работы с DOM

На рис. 40.2 отображена иерархия классов `QDomNode`, предоставляемая Qt для работы с DOM. Доступ ко всем классам DOM можно получить включением заголовочного файла QtXml. Самые используемые из этих классов: `QDomNode`, `QDomElement`, `QDomAttr` и `QDomText`.

Чтение XML-документа

Класс `QDomElement` создан для представления элементов. Иерархия DOM содержит узлы различного типа. Например, узел элемента соответствует открытому и закрытому тегу. Данные, находящиеся между этими тегами, представляют собой узлы потомков, также имеющие тип `QDomElement`. Все узлы иерархии DOM являются объектами класса `QDomNode`, которые способны содержать в себе любые типы узлов. Для проведения операций над узлом, его, прежде всего, необходимо преобразовать к нужному типу. Для преобразования объектов `QDomNode` в `QDomElement` следует воспользоваться методом `QDomNode::toElement()`. Необходимо всегда проверять возвращаемое этим методом значение — ведь в случае ошибки будет возвращено нулевое значение (это можно проверить методом `isNull()`).

Программа, приведенная в листингах 40.1 и 40.2, осуществляет чтение XML-документа и выводит его данные на экран в следующем виде:

```
Attr:  "1"
TagName:  "name"        Text:  "Piggy"
TagName:  "phone"       Text:  "+49 631322187"
TagName:  "email"       Text:  "piggy@mega.de"
Attr:  "2"
TagName:  "name"        Text:  "Kermit"
TagName:  "phone"       Text:  "+49 631322181"
TagName:  "email"       Text:  "kermit@mega.de"
```

Для поддержки XML библиотека Qt предоставляет отдельный модуль, который для его включения нужно указать в секции `QT` pro-файла (листинг 40.1). Обратите внимание на секцию `CONFIG`, в которой добавилось значение `console`. Это нужно для того, чтобы приложение могло осуществлять вывод на консоль в ОС Windows.

Листинг 40.1. Чтение XML-документа и вывод его данных на экран (файл XmlDOMRead.pro)

```
TEMPLATE        = app
QT             += xml
```

```
win32: CONFIG += console
SOURCES        = main.cpp
TARGET         = ../XmlDOMRead
```

В функции `main()`, показанной в листинге 40.2, создаются объекты классов `QDomDocument` и `QFile`. Объект класса `QDomDocument` представляет собой XML-документ. Для считывания XML-документа в метод `setContext()` объекта класса `QDomDocument` необходимо передать объект, созданный от класса, унаследованного от `QIODevice`. В нашем случае — это класс `QFile`. Далее, для получения объекта класса `QDomElement` выполняется вызов метода `documentElement()` объекта класса `QDomDocument`, возвращающий корневой элемент XML-документа. Полученный объект передается в функцию `traverseNode()`, которая обеспечивает прохождение по всем элементам XML-документа.

Для прохождения по иерархии DOM удобно применить рекурсию. Из рекурсивной функции `traverseNode()` вызываются методы `firstChild()` и `nextSibling()` объекта класса `QDomNode`. Если метод `nextSibling()` возвращает нулевое значение, то это значит, что узлов больше нет. Все получаемые узлы проверяются в цикле с помощью метода `isElement()`. Если полученный узел является элементом, то он преобразуется к типу `QDomElement` с помощью метода `toElement()` и результат присваивается объекту класса `QDomElement`. При обнаружении элемента с именем `"contact"` (получаемым методом `tagName()`) его метод `attribute()` используется для отображения значения его атрибута `number`. В остальных случаях вызываются методы `tagName()` и `text()` для отображения имени и данных элемента.

Листинг 40.2. Чтение XML-документа и вывод его данных на экран (файл main.cpp)

```
#include <QtXml>
// ------------------------------------------------------------------------
void traverseNode(const QDomNode& node)
{
   QDomNode domNode = node.firstChild();
   while(!domNode.isNull()) {
       if(domNode.isElement()) {
          QDomElement domElement = domNode.toElement();
          if(!domElement.isNull()) {
              if(domElement.tagName() == "contact") {
                  qDebug() << "Attr: "
                           << domElement.attribute("number", "");
              }
              else {
                  qDebug() << "TagName: " << domElement.tagName()
                           << "\tText: " << domElement.text();
              }
          }
       }
       traverseNode(domNode);
       domNode = domNode.nextSibling();
    }
}
```

```
// ----------------------------------------------------------------------
int main()
{
    QDomDocument domDoc;
    QFile        file("addressbook.xml");

    if(file.open(QIODevice::ReadOnly)) {
        if(domDoc.setContent(&file)) {
            QDomElement domElement= domDoc.documentElement();
            traverseNode(domElement);
        }
        file.close();
    }

    return 0;
}
```

Создание и запись XML-документа

При создании XML-документа необходимо иметь в своем распоряжении механизм создания элементов. Для этого класс `QDomDocument` содержит серию методов, например: `createElement()`, `createTextNode()`, `createAttribute()`. Каждый из этих методов возвращает объект узла.

Программа, приведенная в листинге 40.3, демонстрирует процесс создания XML-документа. Здесь после создания объекта класса `QDomDocument` необходимо создать начальный элемент, который представляет собой начальный узел создаваемой иерархии. В нашем случае этот элемент имеет имя `"addressbook"`. Вызов метода `appendChild()` добавляет созданный элемент. Вызов функции `contact()` создает элемент, содержащий контактную информацию (см. далее листинг 40.4). Эти элементы добавляются методом `appendChild()` в начальный элемент документа `domElement`. Для записи XML-документа в файл необходимо вызвать метод `toString()`, который возвратит текстовое представление XML-документа, и после этого осуществить запись в файл.

Листинг 40.3. Создание XML-документа — функция `main()` (файл main.cpp)

```
int main()
{
    QDomDocument doc("addressbook");
    QDomElement  domElement = doc.createElement("addressbook");
    doc.appendChild(domElement);

    QDomElement contact1 =
        contact(doc, "Piggy", "+49 631322187", "piggy@mega.de");

    QDomElement contact2 =
        contact(doc, "Kermit", "+49 631322181", "kermit@mega.de");

    QDomElement contact3 =
        contact(doc, "Gonzo", "+49 631322186", "gonzo@mega.de");
```

```
    domElement.appendChild(contact1);
    domElement.appendChild(contact2);
    domElement.appendChild(contact3);

    QFile file("addressbook.xml");
    if(file.open(QIODevice::WriteOnly)) {
        QTextStream(&file) << doc.toString();
        file.close();
    }
    return 0;
}
```

В листинге 40.4 приведена функция `contact()`. Она содержит статическую переменную, увеличивающую свое значение после каждого вызова функции. Это необходимо для присвоения контактным адресам определенного номера. Из этой функции вызывается функция `makeElement()`, которая создает элементы, представляющие собой составные части контактного адреса.

Листинг 40.4. Функция `contact()` (файл main.cpp)

```
QDomElement contact(      QDomDocument& domDoc,
                    const QString&      strName,
                    const QString&      strPhone,
                    const QString&      strEmail
                   )
{
    static int nNumber = 1;

    QDomElement domElement = makeElement(domDoc,
                                         "contact",
                                         QString().setNum(nNumber)
                                        );
    domElement.appendChild(makeElement(domDoc, "name", "", strName));
    domElement.appendChild(makeElement(domDoc, "phone", "", strPhone));
    domElement.appendChild(makeElement(domDoc, "email", "", strEmail));

    nNumber++;

    return domElement;
}
```

В листинге 40.5 приводится функция `makeElement()`, которая создает элемент с помощью метода `createElement()` объекта класса `QDomDocument`. Если третьим параметром было передано значение атрибута, то к элементу будет добавлен атрибут `number`. Его создание выполняется вызовом метода `createAttribute()` объекта класса `QDomDocument`. Вызов метода `setValue()` присвоит этому атрибуту переданное в метод значение. Если в четвертом параметре функции была передана строка текста, то методом `createTextNode()` объекта класса `QDomNode` будет создан текстовый узел. Вызов метода `appendChild()` внесет текстовую информацию в объект элемента.

Листинг 40.5. Функция makeElement() (файл main.cpp)

```
QDomElement makeElement(      QDomDocument& domDoc,
                        const QString&      strName,
                        const QString&      strAttr = QString(),
                        const QString&      strText = QString()
                       )
{
    QDomElement domElement = domDoc.createElement(strName);

    if (!strAttr.isEmpty()) {
        QDomAttr domAttr = domDoc.createAttribute("number");
        domAttr.setValue(strAttr);
        domElement.setAttributeNode(domAttr);
    }

    if (!strText.isEmpty()) {
        QDomText domText = domDoc.createTextNode(strText);
        domElement.appendChild(domText);
    }
    return domElement;
}
```

Работа с SAX

Работа с моделью DOM ввиду большого расхода памяти не всегда желательна или возможна. Существует принципиально иной способ для анализа XML-документов — это SAX.

SAX (Simple API for XML, простой API для XML) является стандартом Java API для считывания XML-документов. SAX применяется для последовательного считывания XML-данных, что позволяет без проблем работать с очень большими файлами.

Чтение XML-документа

Класс `QXmlSimpleReader` представляет собой XML-анализатор, базирующийся на SAX. Он читает XML-документ блоками и сообщает о нахождении и закрытии тегов в соответствующих методах. В этом и состоит его основное преимущество: в память помещаются только фрагменты, а не весь XML-документ. Но это и недостаток, так как информация не считывается целиком, и невозможно получить иерархию XML-документа.

Иерархия классов для работы с SAX показана на рис. 40.3. Класс `QXmlContentHandler` должен использоваться для соединения с объектом класса `QXmlSimpleReader`. Другие классы, такие как `QXmlEntityResolver`, `QXmlDTDHandler`, `QXmlErrorHandler`, `QXmlDeclHandler` и `QxmlLexicalHandler`, просто содержат определения виртуальных методов, соответствующих различным событиям анализа XML-документа.

В большинстве случаев, для считывания XML-документа можно прекрасно обойтись двумя классами: `QXmlContentHandler` и `QXmlErrorHandler`. Интерфейс класса `QXmlContentHandler` содержит методы, связанные с отслеживанием структуры документа. Вызов этих методов происходит в следующем порядке:

1. Метод `startDocument()` — вызывается при начале чтения XML-документа.
2. Метод `startElement()` — вызывается при начале чтения элемента.
3. Метод `characters()` — вызывается при чтении данных элемента.
4. Метод `endElement()` — вызывается при завершении обработки элемента.
5. Метод `endDocument()` — вызывается при завершении обработки документа.



Рис. 40.3. Классы для работы с SAX

Определенное удобство обеспечивает класс `QXmlDefaultHandler`, который унаследован от всех шести классов (см. рис. 40.3), — он содержит пустые реализации виртуальных методов этих классов. Для чтения XML-документа нужно унаследовать его и переопределить следующие методы: `startDocument()`, `startElement()`, `characters()`, `endElement()`, `endDocument()` и `fatalError()`. Определение последнего метода унаследовано от класса `QXmlErrorHandler`. Если указанные методы возвращают значение `true`, это говорит объекту класса `QXmlSimpleReader` о том, что он должен продолжить анализ файла. Значение `false` говорит об ошибке, а чтобы отобразить соответствующее сообщение о ней, необходимо переопределить метод `errorString()`.

Программа, приведенная в листингах 40.6 и 40.7, осуществляет чтение XML-документа. Вывод на консоль аналогичен выводу программы, приведенной в листингах 40.1 и 40.2.

В основной программе, приведенной в листинге 40.6, создаются объекты классов `AddressBookParser` (см. далее листинг 40.7), `QFile` (для чтения файла) и `QXmlSimpleReader` (для анализа файла). Чтобы поместить XML-документ в SAX-анализатор, нужно создать объект класса `QXmlInputSource`, передав ему указатель на `QIODevice`. Созданный объект нужно передать в метод `parse()` объекта класса `QXmlSimpleReader`. Этот метод запустит процесс анализа XML-документа. До его вызова необходимо установить в методе `setContentHandler()` объект, унаследованный от `QXmlContentHandler`, который выполняет анализ и отображение XML-документа.

Листинг 40.6. Чтение XML-документа — функция `main()` (файл main.cpp)

```
int main()
{
    AddressBookParser handler;
    QFile             file("addressbook.xml");
    QXmlInputSource   source(&file);
    QXmlSimpleReader  reader;
```

```
    reader.setContentHandler(&handler);
    reader.parse(source);

    return 0;
}
```

Приведенный в листинге 40.7 класс `AddressBookParser` реализует виртуальные методы, которые вызываются при прохождении по XML-документу. При нахождении тегов вызываются соответствующие методы `startElement()` и `endElement()`, которые должны быть переопределены для того, чтобы отреагировать надлежащим образом. Метод `startElement()` вызывается, когда при считывании встречается открытие тега. Первые два параметра, передаваемые в этот метод, относятся к пространствам имен XML, которые мы здесь не рассматриваем, третий параметр, также нами не используемый, — это имя тега, а четвертый — список атрибутов — нами просматривается. Если название атрибута совпадает со строкой `number`, то выводится его значение. Метод `characters()` записывает содержимое текущего элемента в переменную `m_strText`. Это необходимо для того, чтобы можно было получить доступ к указанному содержимому из любого метода класса `AddressBookParser`. Метод `endElement()` вызывается всегда, когда при чтении встречается закрытие тега. Третий параметр, передаваемый в этот метод, — имя тега. Если он не совпадает со строками `contact` и `addressbook`, то данные элемента выводятся на экран. Метод `fatalError()` вызывается в том случае, если не удается проанализировать XML-документ. Тогда метод осуществляет отображение предупреждающего сообщения с указанием номера строки и столбца XML-документа, в котором произошел сбой, а также текст ошибки анализатора.

Листинг 40.7. Класс `AddressBookParser` (файл main.cpp)

```
class AddressBookParser : public QXmlDefaultHandler {
private:
    QString m_strText;

public:
    // ---------------------------------------------------------------------
    bool startElement(const QString&,
                      const QString&,
                      const QString&,
                      const QXmlAttributes& attrs
                     )
    {
        for(int i = 0; i < attrs.count(); i++) {
            if(attrs.localName(i) == "number") {
                qDebug() << "Attr:" << attrs.value(i);
            }
        }
        return true;
    }

    // ---------------------------------------------------------------------
    bool characters(const QString& strText)
    {
        m_strText = strText;
```

```
        return true;
    }

    // ---------------------------------------------------------------------
    bool endElement(const QString&, const QString&, const QString& str)
    {

        if (str != "contact" && str != "addressbook") {
            qDebug() << "TagName:" << str
                     << "\tText:"  << m_strText;
        }
        return true;
    }

    // ---------------------------------------------------------------------
    bool fatalError(const QXmlParseException& exception)
    {
        qDebug() << "Line:"      << exception.lineNumber()
                 << ", Column:"  << exception.columnNumber()
                 << ", Message:" << exception.message();
        return false;
    }
};
```

Класс *QXmlStreamReader* для чтения XML

Qt предоставляет еще одну возможность для чтения XML-файлов — при помощи класса `QXmlStreamReader`. Его принцип очень похож на использование SAX, но он еще проще и быстрее, поскольку ему не нужен специальный обработчик, как в случае с SAX. Так что рассматривайте этот класс как более быструю замену для SAX. Простоту использования класса `QXmlStreamReader` демонстрирует листинг 40.8.

Листинг 40.8. Чтение при помощи класса `QXmlStreamReader` (файл main.cpp)

```
#include <QtXml>

int main()
{
    QFile file("addressbook.xml");
    if(file.open(QIODevice::ReadOnly)) {
        QXmlStreamReader sr(&file);
        do {
            sr.readNext();
            qDebug() << sr.tokenString() << sr.name() << sr.text();
        } while(!sr.atEnd());

        if (sr.hasError()) {
            qDebug() << "Error:" << sr.errorString();
        }
```

```
        file.close();
    }

    return 0;
}
```

Здесь для XML-данных мы используем все тот же файл addressbook.xml и открываем его для чтения. Адрес объекта файла мы передаем в конструктор класса `QXmlStreamReader` и создаем объект `sr`. Вызов метода `readNext()` в цикле `do...while()` выполняет считывание маркера и возвращает код его типа, который мы игнорируем. Далее мы выводим текущий маркер в виде строки (метод `tokenString()`), имя элемента (метод `name()`) и его текст (метод `text()`). И так продолжаем повторять те же действия, пока не достигнем конца документа (метод `atEnd()`).

Проверку на возникновение ошибок осуществляем вызовом метода `hasError()` и при появлении ошибки выводим текст сообщения о ней (метод `errorString()`). Вывод на консоли после запуска примера будет следующим:

```
"StartDocument" "" ""
"Comment" "" " My Address Book "
"StartElement" "addressbook" ""
"Characters" "" ""
"StartElement" "contact" ""
"Characters" "" ""
"StartElement" "name" ""
"Characters" "" "Piggy"
"EndElement" "name" ""
"Characters" "" ""
"StartElement" "phone" ""
"Characters" "" "+49 631322187"
"EndElement" "phone" ""
"Characters" "" ""
"StartElement" "email" ""
"Characters" "" "piggy@mega.de"
"EndElement" "email" ""
"Characters" "" ""
"EndElement" "contact" ""
"Characters" "" ""
"StartElement" "contact" ""
"Characters" "" ""
"StartElement" "name" ""
"Characters" "" "Kermit"
"EndElement" "name" ""
"Characters" "" ""
"StartElement" "phone" ""
"Characters" "" "+49 631322181"
"EndElement" "phone" ""
"Characters" "" ""
"StartElement" "email" ""
"Characters" "" "kermit@mega.de"
```

```
"EndElement" "email" ""
"Characters" "" ""
"EndElement" "contact" ""
"Characters" "" ""
"EndElement" "addressbook" ""
"EndDocument" "" ""
```

Использование XQuery

XQuery (XML Query Language, язык запросов XML) служит для того, чтобы из большого количества XML-данных выбирать только определенные интересующие вас данные. Он использует синтаксис, который является гибридом XSLT, SQL и C. Библиотека Qt реализацию XQuery предоставляет в классе `QXmlQuery`, который расположен в модуле `QtXmlPatterns`. В качестве примера использования запросов XQuery мы реализуем небольшой интерпретатор этого языка (листинг 40.9).

Начнем с pro-файла. Для использования модуля `QtXmlPatterns` в него необходимо добавить следующую строчку:

```
QT += xmlpatterns
```

В листинге 40.9 мы создаем консольное приложение, которое будет принимать три аргумента: имя программы, имя XML-файла, имя XQ-файла с запросами XQuery. В случае меньшего количества аргументов программа выведет на консоль формат ее использования. Мы считываем в строковую переменную `strQuery` содержимое XQ-файла, имя которого является третьим аргументом (индекс 2). Создаем объект для XML-файла (объект `xmlFile`) и, при удачном его открытии на прочтение, создаем объект класса `QXmlQuery`. Для того чтобы наши XQuery-запросы могли получать доступ к XML-документу, выполняем его связку с символьным именем `inputDocument` при помощи метода `QXmlQuery::bindVariable()`.

Вызовом метода `QXmlQuery::setQuery()` мы устанавливаем текст запроса, который был получен из XQ-файла, и проверяем его методом `QXmlQuery::isValid()` на содержание ошибок.

После чего вызовом метода `QXmlQuery::evaluateTo()` осуществляем запрос и передаем в него адрес объекта `QString`, в который будет записываться результат выполнения запроса. Если выполнение запроса произошло без ошибок, то осуществляется вывод содержимого строки на консоль при помощи метода `qDebug()`.

Листинг 40.9. Интерпретатор XQuery (файл main.cpp)

```
#include <QtCore>
#include <QtXmlPatterns>

int main(int argc, char** argv)
{
    QCoreApplication app(argc, argv);
    if (argc < 3) {
        qDebug() << "usage: XQuery any.xml any.xq";
        return 0;
    }
    QString strQuery = "";
    QFile   xqFile(argv[2]);
```

```
    if(xqFile.open(QIODevice::ReadOnly)) {
        strQuery = xqFile.readAll();
        xqFile.close();
    }
    else {
        qDebug() << "Can't open the file for reading:" << argv[1];
        return 0;
    }

    QFile xmlFile(argv[1]);
    if(xmlFile.open(QIODevice::ReadOnly)) {
        QXmlQuery query;
        query.bindVariable("inputDocument", &xmlFile);
        query.setQuery(strQuery);
        if (!query.isValid()) {
            qDebug() << "The query is invalid";
            return 0;
        }

        QString strOutput;
        if (!query.evaluateTo(&strOutput)) {
            qDebug() << "Can't evaluate the query";
            return 0;
        }

        xmlFile.close();
        qDebug() << strOutput;
    }

    return app.exec();
}
```

Программа, приведенная в листинге 40.10, отображает все контакты XML-документа. Мы декларируем переменную нашего XML-документа при помощи ключевого слова `declare variable`. Дополнение `external` означает, что наша переменная не является внутренней переменной, а используется извне. Далее мы ограничиваем наш документ только записями контактов `"/addressbook/contact/"` и объединяем отдельные данные при помощи функции `concat()` с указанием имен тегов.

Листинг 40.10. Отображение всех элементов (запрос _all.xq)

```
declare variable $inputDocument external;
doc($inputDocument)/addressbook/contact/
concat('(name:', name, ' email:', email, ' phone:', phone, ') ')
```

```
 (name:Kermit email:kermit@mega.de phone:+49 631322181)
```

Запустим интерпретатор:

```
XQuery addressbook.xml _all.xq
```

Вывод на консоль будет следующим:

```
(name:Piggy email:piggy@mega.de phone:+49 631322187)
```

Центральное место в языке занимает показанная в листинге 40.11 конструкция, которая строится на операторах `for`, `where`, `order by` и `return`. Эта конструкция может рассматриваться как аналог запроса `SELECT FROM WHERE` в SQL.

Листинг 40.11. Отображение элемента по совпадению с именем (запрос _kermit.xq)

```
declare variable $inputDocument external;
for $x in doc($inputDocument)/addressbook/contact/name
where data($x) = "Kermit"
order by $x
return data($x)
```

В листинге 40.11 к оператору `for` привязана переменная `x`, которая принимает значения записей из документа, ограниченного `"/addressbook/contact/name"`. Оператор `where` служит для исключения ненужных записей — для этого он всегда содержит выражение логического типа, но в нашем случае мы ограничиваемся только записью с тегом имени `"Kermit"`. Оператор `order by` служит для упорядочивания записей (в нашем примере он не имеет смысла, так как запись будет всего одна). И оператор `return` возвращает значения, которые нам нужны.

Запустим интерпретатор:

```
XQuery addressbook.xml _kermit.xq
```

Вывод на консоль будет:

```
"Kermit"
```

Листинг 40.2 демонстрирует, как можно делать выборку по атрибуту. Мы выбираем запись, у которой в теге контакта атрибут `number` равен 1. Далее мы просто формируем необходимую для передачи информацию с помощью функции `concat()`.

Листинг 40.12. Отображение элемента по атрибуту (запрос _piggy.xq)

```
declare variable $inputDocument external;
doc($inputDocument)/addressbook/contact[xs:integer(@number) = 1]/
concat(name, '; ', email, '; ', phone)
```

Запустим интерпретатор:

```
XQuery addressbook.xml _piggy.xq
```

Вывод на консоль будет:

```
"Piggy; piggy@mega.de; +49 631322187"
```

Наш XML-файл содержит всего один атрибут и два контакта. Если бы атрибутов было несколько и контактов больше, то можно было бы сделать более сложное комбинированное условие выбора, например:

```
doc($inputDocument)/addressbook/contact[xs:integer(@number) < 20 and @land =
                                                              "Germany"]/.
```

В этом примере мы бы хотели видеть только находящиеся в Германии контакты с номером меньше 20.

Критерий для выбора по атрибутам не обязательно должен находиться у самого последнего тега. Если бы тег "addressbook" содержал атрибуты, то мы могли бы составить наш запрос с их учетом. Например:

```
doc($inputDocument)/addressbook[@owner = "Max Schlee"]/contact/.
```

Здесь мы хотим получить все контакты только из адресной книги, владельцем которой является Max Schlee.

Язык предоставляет функции, дающие информацию о количестве, — например: функция `empty()` возвращает логическое значение `true`, если список пуст, и функция `count()`, которая возвращает количество найденных элементов. В листинге 40.13 функция `count()` осуществляет подсчет контактов, находящихся в XML-документе.

Листинг 40.13. Отображение количества контактов (запрос _count.xq)

```
declare variable $inputDocument external;
count(doc($inputDocument)//contact)
```

Запустим интерпретатор:

```
XQuery addressbook.xml _count.xq
```

Вывод на консоль будет:

```
"2"
```

Резюме

В этой главе мы познакомились с очень популярным форматом хранения и обмена данными — XML. В настоящее время XML очень распространен и привлекает на свою сторону все больше и больше разработчиков прикладного программного обеспечения.

Qt предоставляет три способа работы с XML-документами: DOM, SAX и класс `QXmlStreamReader`. Первый представляет данные XML-документа в виде иерархии (древовидной структуры), что очень удобно для работы. Второй способ считывает данные из XML-документа блоками и сообщает о результатах в определенные виртуальные методы. Третий способ похож на второй, но работает быстрее.

При выборе одного из них следует учитывать, что SAX — это низкоуровневый способ, поэтому он весьма быстр. Его лучше всего применять в тех случаях, когда не требуется выполнения очень сложных операций, а также для считывания больших XML-документов, поскольку SAX более экономно расходует ресурсы памяти.

Для языка запросов XQuery в библиотеке Qt есть отдельный класс `QtXmlPatterns`. С его помощью можно выполнять выбор нужных вам данных из XML-документа, используя запросы. Этот процесс похож на использование SQL в базах данных. Сам класс находится в отдельном модуле `QtXmlPatterns`.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/40-510/** или с помощью следующего QR-кода (рис. 40.4):



Рис. 40.4. QR-код для доступа на страницу комментариев к этой главе

ГЛАВА 41

Программирование баз данных

> Распознавание проблемы, которая может быть решена и достойна решения, есть... тоже своего рода открытие.
> *Макс Поланьи*

База данных представляет собой систему хранения записей, организованных в виде таблиц. База данных может содержать от одной до нескольких сотен таблиц, которые бывают связаны между собой. Таблица состоит из набора строк и столбцов (рис. 41.1). Столбцы таблицы имеют имена, и за каждым столбцом закреплен тип и/или область значений. Строки таблицы баз данных называются *записями*, а ячейки, на которые делится запись, — *полями*. *Первичный ключ* — это уникальный идентификатор записи, который может представлять собой не только один столбец, но и целую комбинацию столбцов.



Рис. 41.1. Таблица (отношение)

Пользователь может выполнять с таблицами множество разных операций: добавлять, изменять и удалять записи, вести поиск и т. д. Для составления подобного рода запросов был разработан язык SQL (Structured Query Language, язык структурированных запросов), который дает возможность не только осуществлять запросы и изменять данные, но и создавать новые базы данных.

Основные положения SQL

Основными действиями, выполняемыми с базой данных, являются: создание новой таблицы, чтение (выборка и проекция), вставка, изменение и удаление данных. Чтобы сохранить цельность изложения, мы опишем базовые команды SQL, позволяющие это сделать. Стан-

дарт SQL поддерживает более развернутый синтаксис для этих команд, чем описывается здесь, не говоря уже о конкретных СУБД (системах управления базами данных), производители которых обычно предоставляют расширенный синтаксис для наилучшего использования их особенностей. Этому предмету обычно посвящаются целые книги, но и изложенных далее основ будет вполне достаточно для написания простых программ, работающих с базами данных.

Сразу же следует сказать, что язык SQL нечувствителен к регистру, то есть буквы, набранные в разных регистрах, не различаются. Например, `SELECT`, `select`, `Select` и т. п. в языке SQL означают одно и то же. В дальнейшем для выделения ключевых слов SQL мы будем использовать верхний регистр.

Создание таблицы

Для создания таблицы, показанной на рис. 41.1, служит команда `CREATE TABLE`, в которой указываются имена столбцов таблицы, их тип, а также задается первичный ключ:

```
CREATE TABLE addressbook (
    number INTEGER PRIMARY KEY NOT NULL,
    name   VARCHAR(15),
    phone  VARCHAR(12),
    email  VARCHAR(15)
);
```

Операция вставки

После создания таблицы в нее можно добавлять данные. Для этого SQL предоставляет команду вставки `INSERT INTO`. Сразу после названия таблицы нужно указать в скобках имена столбцов, в которые будут заноситься данные. Сами данные указываются после ключевого слова `VALUES`:

```
INSERT INTO addressbook (number, name, phone, email)
VALUES(1, 'Piggy', '+49 631322187', 'piggy@mega.de');

INSERT INTO addressbook (number, name, phone, email)
VALUES(2, 'Kermit', '+49 631322181', 'kermit@mega.de');
```

Чтение данных

Составная команда `SELECT ... FROM ... WHERE` осуществляет операции выборки и проекции. Выборка соответствует выбору строк, а проекция — выбору столбцов. Команда возвращает созданную согласно заданным критериям таблицу с частью исходных данных. Команда составлена из трех частей:

1. Ключевое слово `SELECT` представляет собой команду для выполнения проекции — после него указываются столбцы, которые должны стать ответом на запрос. Если после `SELECT` указать знак `*`, то результирующая таблица будет содержать все столбцы таблицы, к которой был запрос адресован. Указание конкретных имен столбцов оставляет в проекции только эти столбцы.
2. После ключевого слова `FROM` задается таблица, к которой адресован запрос.

3. Ключевое слово WHERE является оператором выборки. Выборка осуществляется согласно условиям, указанным сразу же после оператора. Эту часть команды можно опустить, что приведет к включению в результат всех строк исходной таблицы.

Например, для получения адреса электронной почты мисс Piggy нужно сделать следующее:

```
SELECT email
FROM addressbook
WHERE name = 'Piggy';
```

Изменение данных

Для изменения данных таблицы служит составная команда UPDATE ... SET ... WHERE. После названия таблицы в операторе SET указывается название столбца, в который будет заноситься нужное значение, и через знак = само это значение. Указав несколько таких выражений через запятую, можно обновить сразу несколько столбцов. Изменение данных выполняется в строках, удовлетворяющих условию, указанному в ключевом слове WHERE. В приведенном далее примере осуществляется замена адреса электронной почты мисс Piggy с **piggy@mega.de** на **piggy@supermega.de**:

```
UPDATE addressbook
SET email = 'piggy@supermega.de'
WHERE name = 'Piggy';
```

Удаление

Удаление строк из таблицы выполняется при помощи команды DELETE FROM ... WHERE. После ключевого слова WHERE следует критерий, согласно которому осуществляется удаление строк. Например, удалить адрес мисс Piggy из таблицы можно следующим образом:

```
DELETE FROM addressbook
WHERE name = 'Piggy';
```

Использование языка SQL в библиотеке Qt

Для работы с базами данных библиотека Qt предоставляет отдельный модуль QtSql. Чтобы начать его использование, необходимо в проектный файл добавить следующую строку:

```
QT += sql
```

А чтобы работать с классами этого модуля, нужно включить и заголовочный метафайл QtSql:

```
#include <QtSql>
```

Классы модуля QtSql разделяются на три уровня:

1. Уровень драйверов.
2. Программный уровень.
3. Уровень пользовательского интерфейса.

К *первому уровню* относятся классы для получения данных на физическом уровне. Это такие классы, как QSqlDriver, QSqlDriverCreator<T*>, QSqlDriverCreatorBase, QSqlDriverPlugin и QSqlResult.

Классы *второго уровня* предоставляют программный интерфейс для обращения к базе данных. К классам этого уровня относятся следующие: `QSqlDatabase`, `QSqlQuery`, `QSqlError`, `QSqlField`, `QSqlIndex` и `QSqlRecord`.

Третий уровень предоставляет модели для отображения результатов запросов в представлениях интервью. К этим классам относятся: `QSqlQueryModel`, `QSqlTableModel` и `QSqlRelationalTableModel`.

К классам первого уровня вам обращаться не придется, если вы не собираетесь писать свой собственный драйвер для менеджера базы данных. В большинстве случаев все ограничивается использованием конкретной СУБД, поддерживаемой Qt. В настоящее время существует несколько десятков СУБД, а те из них, что поддерживаются Qt, приведены в табл. 41.1. Все СУБД из этой таблицы (за исключением SQLite) работают в режиме «клиент-сервер», когда сервер базы данных работает как отдельный процесс и взаимодействует с клиентской частью по сети. Если приложение рассчитано на работу только с локальными данными, то для базы данных удобнее всего использовать SQLite, кроме того, ее драйвер и сама база по умолчанию всегда находятся вместе с Qt.

***Таблица 41.1.** Идентификаторы менеджеров баз данных*

Идентификатор	Описание
QOCI	Доступ к базам данных Oracle через Oracle Call Interface (OCI). Поддерживаются версии 7, 8 и 9
QODBC	ODBC (Open Database Connectivity, открытый интерфейс доступа к базе данных) — стандартный ODBC-драйвер для Microsoft SQL Server, IBM DB2, Sybase SQL, iODBC и других баз данных
QMYSQL	MySQL — самый популярный в настоящее время бесплатный менеджер базы данных. Всю необходимую информацию можно получить на странице **www.mysql.com**
QTDS	Sybase Adaptive Server
QPSQL	Базы данных PosgreSQL с поддержкой SQL92/SQL3
QSQLITE2	SQLite версии 2
QSQLITE	SQLite версии 3 и выше
QIBASE	Borland InterBase
QDB2	DB/2 — платформонезависимая база данных, разработанная IBM
QOCI	Oracle Call Interface

Если в табл. 41.1 отсутствует нужная вам база данных, то вам потребуется самому написать для нее драйвер.

По умолчанию Qt собирается так, что драйверы баз данных подключаются к ней в виде файлов расширений (plug-ins). В процессе сборки библиотеки отыскиваются установленные в системе пакеты баз данных и к ним компилируются соответствующие файлы расширений.

> ***Передавайте клиентам файлы расширений вместе с самим приложением!***
>
> Если вы написали приложение с поддержкой базы данных и собираетесь передать его клиентам, то не забудьте позаботиться о том, чтобы файлы расширений драйверов базы были переданы вместе с этим приложением. Например, для Windows файлы расширений драй-

веров баз данных должны находиться в каталоге *<MyApplication>/sqldrivers/*. В Linux подкаталог тот же, а для Mac OS X используйте утилиту macdeployqt, которая сама автоматически разместит их как нужно. Не забудьте также вместе с вашим приложением предоставить и все файлы, необходимые для работы самой базы данных. Их можно скопировать, например, в основной каталог вашего приложения.

Соединение с базой данных (второй уровень)

Для соединения с базой данных прежде всего нужно активизировать драйвер. Для этого вызывается статический метод `QSqlDatabase::addDatabase()` (листинг 41.1). В него нужно передать строку, обозначающую идентификатор драйвера СУБД (см. табл. 41.1).

Листинг 41.1. Функция соединения с базой данных (файл main.cpp)

```
static bool createConnection()
{
    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("addressbook");

    db.setUserName("elton");
    db.setHostName("epica");
    db.setPassword("password");
    if (!db.open()) {
        qDebug() << "Cannot open database:" << db.lastError();
        return false;
    }
    return true;
}
```

Для того чтобы подключиться к базе данных, потребуется четыре следующих параметра:

- имя базы данных — передается в метод `QSqlDatabase::setDatabaseName()`;
- имя пользователя, желающего к ней подключиться, — передается в метод `QSqlDatabase::setUserName()`;
- имя компьютера, на котором размещена база данных, — передается в метод `QSqlDatabase::setHostName()`;
- пароль — передается в метод `QSqlDatabase::setPassword()`.

Методы должны вызываться из объекта, созданного с помощью статического метода `QSqlDatabase::addDatabase()` (см. листинг 41.1).

Само соединение осуществляется методом `QSqlDatabase::open()`. Значение, возвращаемое им, рекомендуется проверять. В случае возникновения ошибки, информацию о ней можно получить с помощью метода `QSqlDatabase::lastError()`, который возвращает объект класса `QSqlError`. Его содержимое можно вывести на экран с помощью метода `qDebug()`. Если вы хотите получить строку с ошибкой, то можно вызвать метод `text()` объекта класса `QSqlError`.

При помощи объекта класса `QSqlDatabase` можно также получить и метаинформацию о базе данных, например о таблицах. Это можно сделать конечно же только после подключения к самой базе данных. Следующий пример при помощи метода `QSqlDatabase::tables()`

получает информацию об именах всех таблиц, которые находятся в базе данных, и отображает их:

```
QStringList lst = db.tables();
foreach (QString str, lst) {
    qDebug() << "Table:" << str;
}
```

Исполнение команд SQL (второй уровень)

Для исполнения команд SQL после установки соединения можно использовать класс `QSqlQuery`. Запросы (команды) оформляются в виде обычной строки, которая передается в конструктор или в метод `QSqlQuery::exec()`. При передаче запроса в конструктор запуск команды будет выполняться автоматически при создании объекта.

Класс `QSqlQuery` предоставляет возможность навигации. Например, после выполнения запроса `SELECT` можно перемещаться по отобранным данным при помощи методов `next()`, `previous()`, `first()`, `last()` и `seek()`. С помощью метода `next()` мы перемещаемся на следующую строку данных, а вызов метода `previous()` перемещает нас на предыдущую строку данных. Методы `first()` и `last()` помогут нам установить первую и последнюю строку данных соответственно. Метод `seek()` устанавливает текущей строку данных, целочисленный индекс которой указан в параметре. Количество строк данных можно получить вызовом метода `size()`.

Дополнительные сложности возникают с запросом `INSERT`. Дело в том, что в запрос нужно внедрять данные. Для этого можно воспользоваться двумя методами: `prepare()` и `bindValue()`. В методе `prepare()` мы задаем шаблон, данные в который подставляются методами `bindValue()`. Например:

```
query.prepare("INSERT INTO addressbook (number, name, phone, email)"
              "VALUES(:number, :name, :phone, :email);");
query.bindValue(":number", 1);
query.bindValue(":name", "Piggy");
query.bindValue(":phone", "+49 631322187");
query.bindValue(":email", "piggy@mega.de");
```

Можно также прибегнуть и к известному из интерфейса ODBC методу использования безымянных параметров:

```
query.prepare("INSERT INTO addressbook (number, name, phone, email)"
              "VALUES(?, ?, ?, ?);");
query.bindValue(1);
query.bindValue("Piggy");
query.bindValue("+49 631322187");
query.bindValue("piggy@mega.de");
```

Есть и третий вариант — воспользоваться классом `QString`, в частности, методом `QString::arg()`, с помощью которого имеется возможность выполнить подстановку значений данных. Этот способ показан в листинге 41.2.

Разумеется, в рассматриваемом случае можно было бы сразу вставить данные в текст запроса, поскольку они известны заранее, но в реальных приложениях данные чаще всего представляют собой значения выражений, и такой подход не срабатывает.

Программа, приведенная в листинге 41.2, демонстрирует исполнение команд SQL — осуществляется создание базы, запись данных и их опрос. В результате на консоль будут выведены следующие данные:

```
1   "Piggy" ;    "+49 631322187" ;       "piggy@mega.de"
2   "Kermit" ;   "+49 631322181" ;       "kermit@mega.de"
```

Листинг 41.2. Исполнение команд SQL — функция `main()` (файл main.cpp)

```
int main(int argc, char** argv)
{
    QCoreApplication app(argc, argv);

    // Соединяемся с менеджером баз данных
    if (!createConnection()) {
        return -1;
    }

    // Создаем базу
    QSqlQuery query;
    QString   str  = "CREATE TABLE addressbook ( "
                        "number INTEGER PRIMARY KEY NOT NULL, "
                        "name   VARCHAR(15), "
                        "phone  VARCHAR(12), "
                        "email  VARCHAR(15) "
                    ");";

    if (!query.exec(str)) {
        qDebug() << "Unable to create a table";
    }

    // Добавляем данные в базу
    QString strF =
          "INSERT INTO  addressbook (number, name, phone, email) "
          "VALUES(%1, '%2', '%3', '%4');";

    str = strF.arg("1")
              .arg("Piggy")
              .arg("+49 631322187")
              .arg("piggy@mega.de");
    if (!query.exec(str)) {
        qDebug() << "Unable to make insert operation";
    }

    str = strF.arg("2")
              .arg("Kermit")
              .arg("+49 631322181")
              .arg("kermit@mega.de");
    if (!query.exec(str)) {
        qDebug() << "Unable to make insert operation";
    }
```

```
    if (!query.exec("SELECT * FROM addressbook;")) {
        qDebug() << "Unable to execute query - exiting";
        return 1;
    }

    // Считываем данные из базы
    QSqlRecord rec     = query.record();
    int        nNumber = 0;
    QString    strName;
    QString    strPhone;
    QString    strEmail;

    while (query.next()) {
        nNumber  = query.value(rec.indexOf("number")).toInt();
        strName  = query.value(rec.indexOf("name")).toString();
        strPhone = query.value(rec.indexOf("phone")).toString();
        strEmail = query.value(rec.indexOf("email")).toString();

        qDebug() << nNumber << " " << strName << ";\t"
                 << strPhone << ";\t" << strEmail;
    }

    return 0;
}
```

После удачного соединения с базой данных с помощью метода `createConnection()` в листинге 42.2 создается строка, содержащая команду SQL для создания таблицы. Эта строка передается в метод `exec()` объекта класса `QSqlQuery`. Если создать таблицу не удается, на консоль выводится предупреждающее сообщение. Ввиду того, что в таблицу будет внесена не одна строка, в строковой переменной `strF` при помощи символов спецификации определяется шаблон для команды `INSERT`. Вызовы методов `arg()` класса `QString` подставляют нужные значения, используя этот шаблон.

Затем, когда база данных создана, и все данные внесены в таблицу, выполняется запрос `SELECT`, помещающий все строки и столбцы таблицы в объект `query`. Вывод значений таблицы на консоль осуществляется в цикле. При первом вызове метода `next()` этот объект указывает на самую первую строку таблицы. Последующие вызовы приведут к перемещению указателя на следующие строки. В том случае, если записей больше нет, метод `next()` вернет значение `false`, что приведет к выходу из цикла.

Для получения результата запроса следует вызвать метод `QSqlQuery::value()`, в котором необходимо передать номер столбца. Для этого мы воспользуемся методом `record()`. Этот метод возвращает объект класса `QSqlRecord`, который содержит относящуюся к запросу `SELECT` информацию. Вызывая метод `QSqlRecord::indexOf()`, мы получаем индекс столбца.

Метод `value()` возвращает значения типа `QVariant`. `QVariant` — это специальный класс, объекты которого могут содержать в себе значения разных типов (см. *главу 4*). Поэтому в нашем примере полученное значение нужно преобразовать к требуемому типу, воспользовавшись методами `QVariant::toInt()` и `QVariant::toString()`.

Теперь вернемся к объекту класса `QSqlRecord`. Важно понимать, что в листинге 41.2 он просто содержит копию реальной записи, и если в базе произойдет какое-либо изменение, то значение копии останется неизменным. В случае когда, например, о структуре самой записи

мы ничего бы не знали, то могли бы вызвать метод `QSqlRecord::count()` и при помощи метода `QSqlRecord::fieldName()` последовательно опросить все имена полей записи, передав ему численный индекс. Обращаться к каждому полю в отдельности можно также по имени, передав его в метод `QSqlRecord::field()`. Этот метод возвращает объект класса `QSqlField`, который даст всю необходимую информацию об основных характеристиках записи, таких как, например, имя (метод `QSqlField::name()`), тип (метод `QSqlField::type()`), длина (метод `QSqlField::length()`) и значение (метод `QSqlField::value()`). Проверить же существование в записи поля с определенным именем можно при помощи метода `QSqlRecord::contains()`.

Классы SQL-моделей для интервью (третий уровень)

Модуль `QtSql` поддерживает концепцию *интервью* (см. *главу 12*), предоставляя целый ряд моделей для использования их в представлениях.

Использование интервью — это самый простой способ отобразить данные таблицы. В этом случае не потребуется цикла для прохождения по ее строкам. Библиотека Qt предоставляет три разные модели: это модель запроса, табличная модель и реляционная модель. Остановимся подробнее на каждой из них.

Модель запроса

Если вам понадобится осуществить отображение данных какого-либо конкретного запроса `SELECT`, то для этого можно воспользоваться классом `QSqlQueryModel`. Модель запроса предназначена только для чтения данных, и с ее помощью вы сможете быстро отображать результаты запросов без возможности их редактирования.

Листинг 41.3 иллюстрирует отображение только электронных адресов и телефонных номеров всех контактов с именем Piggy. В нашем случае найдется лишь одна соответствующая строка (рис. 41.2).



Рис. 41.2. Использование интервью для отображения выборочных данных

В листинге 41.3 мы создаем табличное представление `QTableView` и модель запроса `QSqlQueryModel`. Строка запроса передается в метод `setQuery()`, после чего результат выполнения запроса проверяется в операторе `if` на наличие проблем исполнения с помощью метода `lastError()`. Объект класса `QSqlError`, возвращаемый этим методом, означает, что ошибок нет, и никаких проблем не возникло. Это и проверяется методом `isValid()`. Возникновение проблем повлечет их отображение в `qDebug()`. В завершение модель устанавливается в представлении вызовом метода `setModel()`.

Листинг 41.3. Использование модели опроса — функция `main()` (файл main.cpp)

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
```

```
    if (!createConnection()) {
        return -1;
    }

    QTableView     view;
    QSqlQueryModel model;
    model.setQuery("SELECT phone, email "
                   "FROM addressbook "
                   "WHERE name = 'Piggy';"
                  );

    if (model.lastError().isValid()) {
        qDebug() << model.lastError();
    }

    view.setModel(&model);
    view.show();

    return app.exec();
}
```

Табличная модель

Класс табличной модели `QSqlTableModel` унаследован от рассмотренного ранее класса модели запроса `QSqlQueryModel`. Он обладает всеми его возможностями и позволяет работать с таблицами баз данных на более высоком уровне. Кроме того, он дает осуществлять редактирование данных и может отображать данные в табличной и иерархической форме. Отображение таблицы базы данных происходит целиком, и если нужно ограничить отображаемые столбцы, то следует передать необходимые столбцы в методы `removeColumn()`. Программа (листинг 41.4), окно которой отображено на рис. 41.3, демонстрирует использование табличной модели.

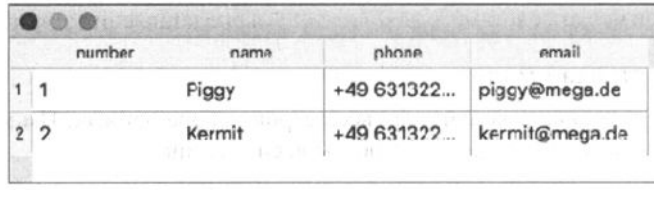


Рис. 41.3. Использование интервью для отображения данных

В листинге 41.4 после соединения с базой данных, которое осуществляется функцией `createConnection()` (см. листинг 41.1), создаются объект табличного представления `QTableView` и объект табличной модели `QSqlTableModel`. Вызовом метода `setTable()` мы устанавливаем в модели актуальную базу. Вызов метода `select()` выполняет заполнение данными.

Листинг 41.4. Использование табличной модели — функция `main()` (файл main.cpp)

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
```

```
    if (!createConnection()) {
        return -1;
    }

    QTableView     view;
    QSqlTableModel model;

    model.setTable("addressbook");
    model.select();
    model.setEditStrategy(QSqlTableModel::OnFieldChange);

    view.setModel(&model);
    view.show();

    return app.exec();
}
```

Теперь настало время немного рассказать о возможностях редактирования и записи данных. Класс `QSqlTableModel` предоставляет для этого три следующие *стратегии редактирования*, которые устанавливаются с помощью метода `setEditStrategy()`:

- `OnRowChange` — запись данных выполняется, как только пользователь перейдет к другой строке таблицы;
- `OnFieldChange` — запись данных происходит после того, как пользователь перейдет к другой ячейке таблицы;
- `OnManualSubmit` — данные записываются при вызове слота `submitAll()`. Если вызывается слот `revertAll()`, то данные возвращаются в исходное состояние.

Какая из стратегий вам больше подходит, должны выбрать вы сами. В листинге 41.4 мы используем стратегию `QSqlTableModel::OnFieldChange`, вызывая метод `setEditStrategy()` с этим аргументом. Теперь данные нашей модели можно изменять после двойного щелчка на ячейке. В завершение вызовом метода `setModel()` в представлении устанавливаем модель.

Прохождение по строкам модели

Метод `rowCount()` возвращает количество всех строк набора данных. Им можно воспользоваться, например, чтобы «пройтись» по строкам всей таблицы.

```
for (int nRow = 0; nRow < model.rowCount(); ++nRow) {
    QSqlRecord rec      = model.record(nRow);
    int        nNumber  = record.value("number").toInt();
    QString    strName  = record.value("name").toString();
...
}
```

Фильтрация и сортировка

Табличная модель для устранения показа ненужных строк записей позволяет устанавливать фильтры. Для этого существует метод `setFilter()`, в который мы можем передать строку с используемым в SQL условным выражением `WHERE`. Метод `setSort()` позволяет выполнять сортировку по нужному столбцу таблицы. Установим наш фильтр и проведем сортировку в убывающем порядке по первому столбцу:

```
model.setFilter("name = 'Piggy'");
model.setSort(0, Qt::DescendingOrder);
model.select();
```

Вставка новых записей

Для того чтобы вставить в таблицу новые записи, надо вызвать метод `insertRow()`, в первом параметре этого метода указать индекс строки, во втором — количество новых строк, а затем вызовами методов `setData()` в новые строки внести необходимые данные. Запомните, что при использовании стратегий обновления `OnFieldChange` и `OnRowChange` за один раз можно вставить только одну строку. Итак, внесем в нашу адресную книгу еще одного героя:

```
model.insertRows(0, 1);
model.setData(model.index(0, 0), 4);
model.setData(model.index(0, 1), "Sam");
model.setData(model.index(0, 2), "+49 63145476576");
model.setData(model.index(0, 3), "sam@mega.de");
if (!model.submitAll()) {
    qDebug() << "Insertion error!";
}
```

Удаление записей

Чтобы удалить записи, нам нужно сначала осуществить их выбор — для этого используется метод фильтра, а затем вызвать метод `removeRows()`. Например, удаление всех записей с именем Piggy могло бы выглядеть так:

```
model.setFilter("name = 'Piggy'");
model.select();
model.removeRows(0, model.rowCount());
model.submitAll();
```

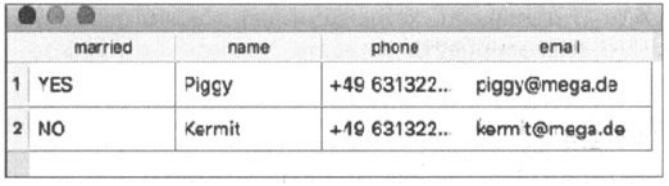
Реляционная модель

Реляционная модель представляет собой более высокий уровень реализации, чем тот, который предоставляет табличная модель. Она обладает механизмами связывания таблиц с помощью первичных (primary) и/или вторичных ключей (foreign keys). Это позволяет модели искать информацию сразу в нескольких таблицах и отображать их в одной. Класс для реляционной модели в Qt — `QSqlRelationalTableModel`. Он унаследован от только что рассмотренного класса `QSqlTableModel`. Этот класс к унаследованным методам добавляет только три новых метода, которые предназначены лишь для работы со связями таблиц: `relationModel()`, `relation()` и `setRelation()`. Для демонстрации дополним нашу базу данных `"addressbook"` еще одной таблицей `"status"`, которая будет содержать информацию о том, женат/замужем контакт или нет.

```
CREATE TABLE status (
    number INTEGER PRIMARY KEY NOT NULL,
    married VARCHAR(5) "
);
```

Внесем в нее данные:

```
INSERT INTO status (number, married) VALUES(1, 'YES');
INSERT INTO status (number, married) VALUES(2, 'NO');
```



	married	name	phone	email
1	YES	Piggy	+49 631322..	piggy@mega.de
2	NO	Kermit	+49 631322..	kermit@mega.de

Рис. 41.4. Использование реляционной модели

Теперь выполним (листинг 41.5) отображение данных из обеих таблиц, как это показано на рис. 41.4.

Листинг 41.5 отличается от листинга 41.3 только вызовом метода `setRelation()`. В этом методе первым параметром мы указываем номер столбца таблицы `"addressbook"`, в котором расположены первичные ключи. Во втором параметре передаем объект `QSqlRelation`. В нем мы указываем таблицу `"status"`, которую хотим объединить с таблицей `"addressbook"`, имя столбца первичных ключей таблицы `"status"` и третьим параметром указываем имя, которое должно быть отображено для пользователя.

Листинг 41.5. Использование реляционной модели — функция `main()` (файл main.cpp)

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    if (!createConnection()) {
        return -1;
    }

    QTableView               view;
    QSqlRelationalTableModel model;

    model.setTable("addressbook");
    model.setRelation(0, QSqlRelation("status", "number", "married"));
    model.select();

    view.setModel(&model);
    view.show();

    return app.exec();
}
```

Резюме

Базы данных — одна из самых объемных тем информатики. Поэтому описание всех возможностей работы с базой данных требует отдельной книги. Эта глава описывает лишь их малую часть, концентрируясь на основных особенностях Qt при работе с базами данных.

Здесь мы узнали, что база данных — это хранилище, организованное в виде таблиц, в каждой ячейке которых содержатся данные определенного типа: текст, числа и т. д.

Язык запросов SQL — это стандарт, который используется в большом количестве СУБД, что обеспечивает его платформонезависимость. Запросы (команды) — это своего рода вопросы, задаваемые базе данных, на которые она должна давать ответы.

Qt представляет модуль поддержки баз данных, классы которого разделены на три уровня: уровень драйверов, программный и пользовательский.

Для работы с базой данных нужно сначала активизировать драйвер, а затем установить связь с базой данных. После этого посредством запросов SQL можно получать, вставлять, изменять и удалять данные. Для отсылки запросов используется класс `QSqlQuery`.

При помощи концепции интервью можно очень просто отображать данные SQL-моделей в представлениях.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/41-510/** или с помощью следующего QR-кода (рис. 41.5):

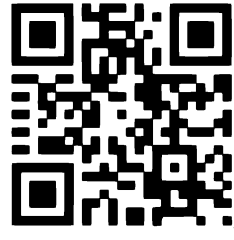


Рис. 41.5. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 42

Динамические библиотеки и система расширений

Что едино, разъедини.
Сунь Цзы

Зачастую приходится реализовывать свои дополнения, предназначенные для использования в одной или нескольких программах. Это можно сделать при помощи динамических библиотек, которые реализуют специальный интерфейс. Некоторые подобные дополнения поставляются с Qt и называются *расширениями* (plug-ins).

Динамические библиотеки

В самом деле, на практике очень часто возникают случаи, когда требуется совместное использование какой-либо функции сразу в нескольких программах, работающих на одном компьютере. Не совсем экономично, если каждая из этих программ будет содержать одинаковый код, — значит, необходим механизм для объединения общего кода таких функций в отдельных файлах (библиотеках), который позволял бы воспользоваться им в необходимых случаях. Такие файлы должны подгружаться в процессе работы самих программ динамически — по мере необходимости и в зависимости от потребностей. Совместно используемая динамически подключаемая библиотека (далее просто *динамическая библиотека*) по структуре представляет собой группу, содержащую объектные файлы. Использование динамических библиотек предоставляет следующие преимущества:

- программа занимает меньше места в оперативной памяти и на диске. Когда динамическая библиотека подключается к программе, то в исполняемый файл включается не код самой библиотеки, а лишь ссылка на нее;
- разные программы могут использовать одну и ту же библиотеку, только ссылаясь на нее;
- после обновления динамической библиотеки не обязательно перекомпилировать использующие ее программы, что позволяет обновлять динамические библиотеки, не затрагивая связанные с ними приложения. То есть, если в библиотеке будет обнаружена ошибка, достаточно просто заменить ее файл другим.

Очень важно также уточнить, что динамическая библиотека — это не просто группа объектных файлов, из которой компоновщиком выбираются нужные для разрешения ссылки. Все объектные файлы, которые содержатся в самой библиотеке, объединяются в единый объектный файл. Это дает преимущество для программ, которые компонуются вместе с библиотекой, потому что они всегда будут иметь доступ сразу ко всему содержимому библиотеки, а не какой-либо отдельной ее части.

Для того чтобы создать динамическую библиотеку, в проектный файл (файл с расширением pro) необходимо включить следующие строки:

```
TEMPLATE = lib
CONFIG  += dll
```

Первая опция (TEMPLATE) говорит о том, что наш проект — библиотека, однако этого недостаточно, потому что библиотеки бывают как динамические, так и статические. Поэтому нам нужна вторая опция (CONFIG) — в ней-то мы и указываем, что нам нужна именно динамическая библиотека. Теперь, после компоновки всего проекта, мы получим файл нашей динамической библиотеки. В Windows такие файлы носят расширение dll, в Mac OS X — dylib, а в Linux — so.

Чтобы использовать динамическую библиотеку в любом проекте другой библиотеки или исполняемой программы, нужно компоновать проект совместно с этой библиотекой. Допустим, нужная нам библиотека называется Tools. Тогда в pro-файл нужно включить ее вместе с путем ее размещения, а также и путь ее заголовочных файлов. Например:

```
LIBS        += -L../../lib/ -lTools
INCLUDEPATH = ../../include
```

Таким образом, если у вас в библиотеке Tools определены классы, то вы можете получать к ним доступ абсолютно так же, как если бы вы определили эти классы внутри самого проекта, который компонуется с этой библиотекой.

Для работы с динамическими библиотеками в среде операционной системы нужно описать место их размещения. Лучше всего это сделать, включив путь ваших динамических библиотек в переменную среды PATH. Например, так:

- в Windows:

```
set PATH=%PATH%;c:\Projects\cpp\lib\win32
```

- в Mac OS X:

```
export DYLD_LIBRARY_PATH=/Projects/cpp/lib/mac/:$DYLD_LIBRARY_PATH
```

- в Linux:

```
export LD_LIBRARY_PATH=/Projects/cpp/lib/x11/:$LD_LIBRARY_PATH
```

Впоследствии, когда программа будет готова для передачи ее заказчикам, то в Windows можно просто разместить ваши динамические программы в одном каталоге с исполняемым файлом. Для Mac OS X лучше всего воспользоваться утилитой macdeployqt. А в Linux можно написать файл сценария, который установит переменную LD_LIBRARY_PATH.

Динамическая загрузка и выгрузка библиотеки

Существуют два способа использования динамических библиотек. В первом способе связывание с динамической библиотекой осуществляется в процессе компоновки самой программы. В этом случае динамическая библиотека загружается автоматически при запуске использующего ее приложения. Этот способ был рассмотрен нами ранее. Теперь перейдем ко второму способу.

Второй способ предоставляет возможность загрузки некоторого кода без явной компоновки во время работы самой программы. Это необходимо, например, в случаях, когда нужно предоставить сторонним разработчикам возможность создавать дополнительные модули для расширения функциональности вашей программы. Суть способа заключается в исполь-

зовании класса QLibrary. Этот класс заботится и о том, чтобы загруженная библиотека оставалась в памяти на протяжении всего времени работы приложения.

> ***Не усложняйте свой проект!***
>
> Используйте этот способ только в тех случаях, когда это действительно необходимо, иначе вы можете необоснованно усложнить весь проект.

Следующий пример, приведенный в листингах 42.1–42.3, демонстрирует создание динамической библиотеки, содержащей только одну функцию.

Обратите внимание на файл проекта, показанный в листинге 42.1. Для создания динамической библиотеки нужно установить в секции TEMPLATE значение lib. Готовая библиотека будет расположена на один уровень выше каталога с ее исходными файлами, для чего в секции DESTDIR мы зададим значение .., при этом для операционной системы Mac OS X мы расположим библиотеку прямо в пакете приложения, поэтому укажем в секции DESTDIR значение MyApplication.app/Contents/libs.

Ввиду того, что создаваемая библиотека не нуждается в элементах пользовательского интерфейса, мы сделаем так, чтобы они были изъяты из компоновки, для чего оператором -= исключаем опцию gui в секции QT.

Листинг 42.1. Создание динамической библиотеки, содержащей одну функцию (файл dynlib.pro)

```
TEMPLATE = lib
mac {
DESTDIR  = ../MyApplication.app/Contents/libs
}
else {
DESTDIR  = ..
}
QT       -= gui
SOURCES  = dynlib.cpp
HEADERS  = dynlib.h
TARGET   = dynlib
```

В динамическую библиотеку должны быть экспортированы прототипы функций для их дальнейшего использования (листинг 42.2). Эти функции необходимо заключить в спецификатор extern "C" {...}. Тогда компилятор C++ не будет прикреплять информацию о типе к символьной сигнатуре функции. Без этого спецификатора компилятор может подставить вместо имени функции (oddUpper()) совсем другое имя, в котором будет закодирована дополнительная информация об этой функции. Такой спецификатор нужно указать, если вы хотите, чтобы вашу динамическую библиотеку можно было загружать в процессе работы основной программы, например, при помощи класса QLibrary, о котором речь пойдет далее. Для операционной системы Windows необходимо, чтобы спецификатор содержал определение: __declspec(dllexport).

Листинг 42.2. Экспорт прототипов функций (файл dynlib.h)

```
#include <QString>

#ifdef Q_OS_WIN
    #define MY_EXPORT __declspec(dllexport)
```

```
#else
    #define MY_EXPORT
#endif

extern "C" MY_EXPORT {
    QString oddUpper(const QString& str);
}
```

В листинге 42.3 показана реализация экспортируемой функции, которая преобразует каждый нечетный символ переданной строки в заглавный и возвращает полученный результат.

Листинг 42.3. Пример реализации экспортируемой функции (файл dynlib.cpp)

```
#include "dynlib.h"

QString oddUpper(const QString& str)
{
    QString strTemp;

    for (int i = 0; i < str.length(); ++i) {
        strTemp += (i % 2) ? str.at(i) : str.at(i).toUpper();
    }

    return strTemp;
}
```

В коде, приведенном в листинге 42.4, реализована загрузка динамической библиотеки и исполнение функции, экспортируемой этой библиотекой. Эта функция изменяет каждый нечетный символ на заглавный (рис. 42.1).

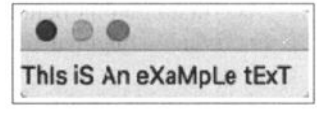


Рис. 42.1. Исполнение функции динамической библиотеки

В листинге 42.4 создается виджет надписи `lbl`, которому присваивается текст в конструкторе. Чтобы использовать динамическую библиотеку в программе, нужно создать объект класса `QLibrary` и передать в его конструктор имя файла динамической библиотеки, но, заметьте, — только имя, без расширения. Это связано с тем, что на разных платформах файлы динамических библиотек имеют различные расширения. Как уже отмечалось ранее, в ОС Windows файл нашей библиотеки будет иметь расширение dll, в UNIX/Linux — so, а в Mac OS X — dylib. Передавая только имя, мы возлагаем на библиотеку Qt ответственность на подстановку нужного расширения.

Указатели на экспортируемые функции извлекаются с помощью метода `resolve()`. В этот метод передается символьная сигнатура, по которой будет осуществляться поиск нужной функции. Возвращает метод указатель на тип `void`, представляющий собой адрес найденной функции. Если метод `resolve()` вернет нулевой указатель, это означает, что функция не найдена. Для вызова функции этот указатель необходимо привести к нужному типу. В слу-

чае успешной проверки указателя мы вызываем саму функцию. В завершение для отображения виджета надписи на экране вызывается метод `show()`.

Листинг 42.4. Загрузка динамической библиотеки и исполнение экспортируемой ею функции (файл main.cpp)

```
#include <QtWidgets>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);

    QLabel   lbl("this is an example text");
    QLibrary lib("dynlib");

    typedef QString (*Fct) (const QString&);
    Fct fct = (Fct)(lib.resolve("oddUpper"));
    if (fct) {
        lbl.setText(fct(lbl.text()));
    }
    lbl.show();

    return app.exec();
}
```

Расширения (plug-ins)

Использование расширений — это неотъемлемая часть любого профессионального приложения. По сути, расширение — это совместно используемая динамическая библиотека, предназначенная для загрузки в процессе исполнения основного приложения, которая обязательно должна реализовывать хотя бы один специальный интерфейс. Расширения делятся на две группы:

- расширения для Qt;
- расширения для собственных приложений.

Расширения для Qt

Библиотека Qt предоставляет различные типы расширений, предназначенных для дополнения ее возможностей, например для поддержки новых форматов растровых файлов, новых драйверов баз данных и т. д. Всего их более 20-ти. Вот некоторые из них:

- `QSqlDriverPlugin` — для драйверов баз данных;
- `QImageIOPlugin` — для поддержки различных форматов растровых изображений (см. *главу 19*);
- `QTextCodecPlugin` — для реализации кодировок текста;
- `QStylePlugin` — для стилей элементов управления (см. *главу 26*).

Все эти классы наследуют класс `QObject`, который является базовым для всей системы расширений Qt. Свои расширения для Qt нужно строить на базе этих классов, реализуя в них нужные методы. Возьмем для примера класс `QStylePlugin`. Реализация расширения с применением этого класса для созданного нами в *главе 26* стиля `CustomStyle` могла бы выглядеть так, как показано в листингах 42.5–42.7.

В классе, унаследованном от `QStylePlugin` (листинг 42.5), необходимо реализовать виртуальный метод `create()`. Чтобы класс расширений стал доступным для библиотеки Qt, его необходимо экспортировать, поэтому мы добавляем в его определение макрос `Q_PLUGIN_METADATA`. В первом параметре этого макроса мы передаем `IID` интерфейса, который реализует наше расширение, и ссылку на файл с метаинформацией о нем в формате JSON (см. *главу 50*). Этот файл придуман для того, чтобы получить метаинформацию о расширении, не прибегая к его загрузке, что дает выигрыш в скорости.

Листинг 42.5. Класс `CustomStylePlugin` (файл CustomStylePlugin.h)

```
class CustomStylePlugin : public QStylePlugin{
Q_OBJECT
Q_PLUGIN_METADATA(IID, "com.mycorp.Qt.CustomInterface" FILE mystyleplugin.json)
public:
    QStyle*     create(const QString &key);
};
```

JSON-файл mystyleplugin.json, показанный в листинге 42.6, содержит список строк с именами реализованных в классе расширений. В нашем примере их два: `CustomStyle` и `MyStyle`.

Листинг 42.6. Список строк с именами реализованных в классе расширений (файл mystyleplugin.json)

```
{
     "Keys": ["CustomStyle", "MyStyle"]
}
```

Метод `create()` создает запрашиваемый объект и возвращает указатель на него. Если запрашиваемый объект не найден, метод вернет нулевое значение (листинг 42.7).

Листинг 42.7. Метод `create()` (файл CustomStylePlugin.cpp)

```
QStyle* MyStylePlugin::create(const QString &key)
{
    if (key == "CustomStyle")
        return new CustomStyle;
    if (key == "MyStyle")
        return new MyStyle;
    return 0;
}
```

Созданный класс расширений должен быть скомпилирован в динамическую библиотеку. Его загрузку можно выполнить при помощи класса `QStyleFactory` (листинг 42.8).

Листинг 42.8. Загрузка класса расширений (файл main.cpp)

```
void main(int argc, char** argv)
{
    ...
    QApplication::setStyle(QStyleFactory::create("CustomStyle"));
    ...
}
```

Для того чтобы библиотека Qt могла воспользоваться нашим расширением, его необходимо поместить в каталог расширений соответствующего типа, расположенный в каталоге библиотеки. В нашем случае это будет каталог <Каталог Qt>/plugins/styles.

> ***ПЕРЕДАВАЙТЕ КЛИЕНТАМ ФАЙЛЫ РАСШИРЕНИЙ ВМЕСТЕ С САМИМ ПРИЛОЖЕНИЕМ!***
>
> Если вы написали приложение с поддержкой расширений Qt и собираетесь передать его клиентам, то не забудьте позаботиться о том, чтобы файлы расширений были переданы вместе с этим приложением. Например, в Windows сами файлы расширений должны находиться в каталогах *<MyApplication>/imageformats/* — для графических расширений и *<MyApplication>/sqldrivers/* — для расширений баз данных. В Linux подкаталоги те же, а в Mac OS X используйте утилиту macdeployqt, которая сделает эту работу за вас. Подобная утилита, способная проделать всю работу по сбору нужных библиотек, есть также и для Windows, называется она windeployqt. Имейте также в виду, что подключение расширений может быть неявным, — например, если вы используете Qt WebEngine (см. *главу 46*), может получиться так, что он не сможет отображать растровые изображения в JPG-, GIF- и TIFF-форматах, если файлы расширений для этих форматов будут недоступны.

Поддержка собственных расширений в приложениях

Связь с расширением осуществляется с помощью *интерфейса* (см. далее), поэтому приложение должно предоставлять по меньшей мере один интерфейс для использования расширения. Расширения загружаются приложением при помощи класса `QPluginLoader`, который содержит несколько методов. Самый часто используемый из них — это метод `instance()`, создающий и возвращающий указатель на объект расширения. Класс `QPluginLoader` автоматически загружает расширение, чье имя файла указывается в его конструкторе. Выгрузку расширения, если в этом есть необходимость, можно осуществить с помощью метода `unload()`.

В следующем примере (листинги 42.9–42.14) создается приложение, предоставляющее поддержку для использования расширений. Окно программы, показанное на рис. 42.2, демонстрирует интерфейс для операций над текстом.

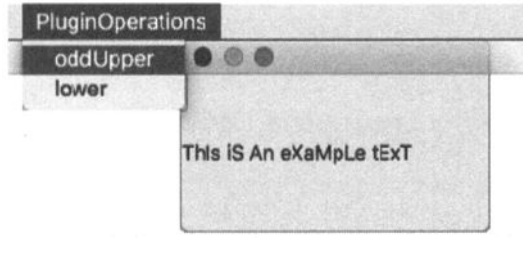


Рис. 42.2. Приложение, поддерживающее расширения

В данном случае *интерфейс* — это класс, который содержит только чисто виртуальные определения методов (листинг 42.9). В приводимом примере приложение предоставляет только один интерфейс — `StringInterface`, из названия которого ясно, что он предназначен для операций над строками. Этот интерфейс объявляет два прототипа методов:

- `operations()` — для получения списка операций расширения;
- `operation()` — для вызова операций над строками.

Виртуальный деструктор нам нужен, чтобы C++ не выдавал предупреждающее сообщение о том, что класс, имеющий виртуальные методы, не имеет виртуального деструктора.

Идентификация интерфейса должна быть задана при помощи макроса `Q_DECLARE_INTERFACE()`, в котором необходимо указать строку-идентификатор, для которой метаобъектный компилятор MOC должен сгенерировать метаинформацию. С ее помощью объект класса `QPluginLoader` проверяет версию расширения и другую информацию, заданную в этой строке. Строка идентификатора состоит из четырех компонентов, разделенных между собой точками:

- домен создателя интерфейса;
- имя приложения;
- имя интерфейса;
- номер версии.

Листинг 42.9. Определение класса интерфейса (файл interfaces.h)

```
#pragma once

class QString;
class QStringList;

// ============================================================================
class StringInterface {
public:
    virtual ~StringInterface() {}

    virtual QStringList operations() const = 0;

    virtual QString operation(const QString& strText,
                              const QString& strOperation
                             ) = 0;
};

Q_DECLARE_INTERFACE(StringInterface,
                    "com.mysoft.Application.StringInterface"
                   )
```

Класс основного окна приложения `PluginsWindow` (листинг 42.10) унаследован от класса `QMainWindow`. Это сэкономит нам время при работе с меню и компоновками.

Листинг 42.10. Определение класса основного окна приложения (файл PluginsWindow.h)

```
#pragma once

#include <QMainWindow.h>
#include "interfaces.h"

class QLabel;
class QMenu;
// ============================================================================
class PluginsWindow : public QMainWindow {
Q_OBJECT

private:
    QLabel* m_plbl;
    QMenu*  m_pmnuPlugins;

public:
    PluginsWindow(QWidget* pwgt = 0);

    void loadPlugins(             );
    void addToMenu  (QObject* pobj);

protected slots:
    void slotStringOperation();
};
```

В конструкторе класса, приведенном в листинге 42.11, создаются виджеты надписи и меню. Виджет надписи (указатель `m_plbl`) вносится в рабочую область приложения, а меню (указатель `m_pmnuPlugins`) вызовом метода `addMenu()` добавляется к основной строке меню. Вызов метода `loadPlugins()` осуществляет поиск и загрузку расширений.

Листинг 42.11. Конструктор `PluginsWindow` (файл PluginsWindow.cpp)

```
PluginsWindow::PluginsWindow(QWidget* pwgt/*=0*/) : QMainWindow(pwgt)
{
    m_plbl        = new QLabel("this is an example text");
    m_pmnuPlugins = new QMenu("&PluginsOperations");

    loadPlugins();
    setCentralWidget(m_plbl);
    menuBar()->addMenu(m_pmnuPlugins);
}
```

В приложении мы хотим использовать все возможные расширения. Поэтому при загрузке расширений (листинг 42.12) исходим из того, что они находятся в каталоге plugins, и ищем все файлы расширений там, для чего задействуем класс `QDir` и инициализируем его текущим каталогом исполняемого файла. В операционной системе Mac OS X мы располагаем каталог файлов расширений одним уровнем выше каталога исполняемого файла, поэтому здесь произведем вызов метода `cdUp()`, чтобы подняться на один уровень выше.

Найденные файлы передаются в конструктор класса `QPluginLoader`. Затем указатель, возвращенный из объекта `QPluginLoader` вызовом метода `instance()`, преобразуется к типу указателя на `QObject` и передается в метод `addToMenu()`, как возможный кандидат для добавления операций расширения к пунктам меню.

Листинг 42.12. Метод `loadPlugins()` (Файл PluginsWindow.cpp)

```
void PluginsWindow::loadPlugins()
{
    QDir dir(QCoreApplication::applicationDirPath());
#ifdef Q_OS_OSX
    dir.cdUp();
#endif
    if (!dir.cd("plugins")) {
        QMessageBox::critical(0, "", "plugins directory does not exist");
        return;
    }

    foreach (QString strFileName, dir.entryList(QDir::Files)) {
        QPluginLoader loader(dir.absoluteFilePath(strFileName));
        addToMenu(qobject_cast<QObject*>(loader.instance()));
    }
}
```

В методе `addToMenu()` первым делом проверяется допустимость указателя, то есть его неравенство нулю (листинг 42.13). Если указатель равен нулю, мы просто выходим из метода. В противном случае проверяем, используя шаблонную функцию `qobject_cast<StringInterface*>`, имеется ли в расширении нужный нашему приложению интерфейс. Поддерживаемых интерфейсов может быть несколько, и при помощи функции `qobject_cast<T>` можно отличить один от другого. Если проверка на поддержку интерфейса прошла удачно, то мы, вызывая метод `operations()`, опрашиваем список всех предоставляемых расширением операций и сохраняем их в переменной `lstOperations`. Затем для каждой операции создаем объект действия, в который передаются название операции и указатель на объект, являющийся расширением. Созданный объект действия соединяется со слотом `slotStringOperation()` и добавляется в меню.

Листинг 42.13. Метод `addToMenu()` (Файл PluginsWindow.cpp)

```
void PluginsWindow::addToMenu(QObject* pobj)
{
    if (!pobj) {
        return;
    }

    StringInterface* pI = qobject_cast<StringInterface*>(pobj);
    if (pI) {
        QStringList lstOperations = pI->operations();
        foreach (QString str, lstOperations) {
            QAction* pact = new QAction(str, pobj);
```

```
            connect(pact, SIGNAL(triggered()),
                    this, SLOT(slotStringOperation())
                   );
            m_pmnuPlugins->addAction(pact);
        }
    }
}
```

В слоте, приведенном в листинге 42.14, метод `sender()` возвращает указатель на объект, выславший сигнал. Этот указатель приводится к типу указателя на `QAction`. Вызовом метода `parent()` из объекта действия мы получаем указатель на объект расширения. Воспользовавшись методом `operation()`, выполняем действия над текстом виджета надписи. В этот метод мы передаем текст виджета надписи и название применяемой операции, которое соответствует названию объекта действия.

Листинг 42.14. Слот `slotStringOperation()` (Файл PluginsWindow.cpp)

```
void PluginsWindow::slotStringOperation()
{
    QAction* pact = qobject_cast<QAction*>(sender());

    StringInterface* pI = qobject_cast<StringInterface*>(pact->parent());

    m_plbl->setText(pI->operation(m_plbl->text(), pact->text()));
}
```

Создание расширения для приложения

Теперь, когда мы имеем приложение, поддерживающее систему расширений, самое время создать для него хотя бы один компонент расширения.

Для создания расширения в секции `CONFIG` pro-файла (листинг 42.15) необходимо добавить опцию `plugin`, а также включить в секцию `HEADERS` файл интерфейсов приложения interfaces.h.

Листинг 42.15. Создание расширения (файл MyPlugin.pro)

```
TEMPLATE = lib
CONFIG   += plugin
QT       -= gui
mac {
DESTDIR  = ../MyApplication.app/Contents/plugins
}
else {
DESTDIR  = ../plugins
}
SOURCES  = MyPlugin.cpp
HEADERS  = MyPlugin.h \
           ../Application/interfaces.h
TARGET   = myplugin
```

Наш класс расширения `MyPlugin`, показанный в листинге 42.16, наследует сразу два класса: `QObject` и `StringInterface`. Добавление макроса `Q_INTERFACES()` нужно для того, чтобы компилятор MOC сгенерировал всю необходимую для расширения метаинформацию. Далее мы должны экспортировать интерфейс, и это мы делаем при помощи макроса `Q_PLUGIN_METADATA()`, который задает точку входа нашего расширения, что делает его доступным для библиотеки Qt. В него мы должны передать IID интерфейса, который реализует наше расширение, и ссылку на файл с метаинформацией о нем в формате JSON (см. *главу 50*). В нашем случае метаданных нет, поэтому этот файл содержит только два символа "{}".

Виртуальный деструктор нам нужен, чтобы компилятор C++ «не жаловался» на то, что класс, имеющий виртуальные методы, не имеет виртуального деструктора.

Листинг 42.16. Определение класса `MyPlugin` (файл MyPlugin.h)

```
#pragma once

#include <QObject>
#include "../Application/interfaces.h"

// ============================================================================
class MyPlugin : public QObject, public StringInterface {
Q_OBJECT
Q_INTERFACES(StringInterface)
Q_PLUGIN_METADATA(IID "com.mysoft.Application.StringInterface" FILE
"stringinterface.json")

private:
    QString oddUpper(const QString& str);

public:
    virtual ~MyPlugin();

    virtual QStringList operations(                              ) const;
    virtual QString     operation (const QString&, const QString&);
};
```

Метод, приведенный в листинге 42.17, возвращает список поддерживаемых расширением операций. В нашем случае их две: `oddUpper` и `lower`.

Листинг 42.17. Метод `operations()` (файл MyPlugin.cpp)

```
/*virtual*/ QStringList MyPlugin::operations() const
{
    return QStringList() << "oddUpper" << "lower";
}
```

Метод `oddUpper()`, приведенный в листинге 42.18, нам уже знаком. Мы использовали его для нашей динамической библиотеки (см. листинг 42.3), а теперь применим и в нашем расширении.

Листинг 42.18. Метод oddUpper() (файл MyPlugin.cpp)

```
QString MyPlugin::oddUpper(const QString& str)
{
    QString strTemp;

    for (int i = 0; i < str.length(); ++i) {
        strTemp += (i % 2) ? str.at(i) : str.at(i).toUpper();
    }

    return strTemp;
}
```

В листинге 42.19 приведен метод `operation()`, который получает текст и имя операции. Этого достаточно, чтобы вызвать нужную реализацию, отвечающую за проведение операции. Если имя операции не будет найдено — приложение выдаст сообщение о том, что заданная операция не поддерживается.

Листинг 42.19. Метод operation() (файл MyPlugin.cpp)

```
/*virtual*/ QString MyPlugin::operation(const QString& strText,
                                        const QString& strOperation
                                       )
{
    QString strTemp;
    if (strOperation == "oddUpper") {
        strTemp = oddUpper(strText);
    }
    else if (strOperation == "lower") {
        strTemp = strText.toLower();
    }
    else {
        qDebug() << "Unsupported operation";
    }
    return strTemp;
}
```

Резюме

Динамическая библиотека содержит код, который может использоваться сразу несколькими приложениями. В отличие от статических библиотек, код, содержащийся в динамической библиотеке, не включается в основной код приложения, а находится в отдельном файле. Если при исполнении программы осуществляется вызов функции из динамической библиотеки, то в память компьютера загружаются только нужные функции. Это позволяет сэкономить объем дискового пространства и оперативной памяти, поскольку общедоступный код содержится в отдельном файле, совместно используемом программами.

Существуют два способа использования динамических библиотек. Первый способ заключается в компоновке программы вместе с библиотекой — в этом случае загрузка и использо-

вание библиотеки происходит вместе со стартом самой программы. Второй же способ не использует явную компоновку, и программа может загружать и выгружать библиотеки в процессе ее работы.

Библиотека Qt предоставляет все необходимые средства для создания и загрузки динамических библиотек. При помощи функции `resolve()` можно получить адрес нужной функции. После приведения адреса к нужному типу можно осуществить его вызов.

Расширение — это динамическая библиотека, реализующая специальный интерфейс. Qt предоставляет возможность реализации двух типов расширений:

- расширения для самой библиотеки;
- расширения для собственных приложений.

Для расширения *самой библиотеки* Qt предлагает классы для реализации драйверов баз данных, создания стилей и др. Чтобы создать свое собственное расширение, нужно унаследовать один из этих классов и переопределить все необходимые виртуальные методы.

Для поддержки расширений *собственных приложений* Qt предоставляет класс `QPluginLoader`, который способен как загружать, так и выгружать расширения.

Приложение, поддерживающее расширения, должно предоставлять хотя бы один интерфейс, посредством которого будет выполняться коммуникация с компонентом расширения. Созданный интерфейс должен быть зарегистрирован при помощи макроса `Q_DECLARE_INTERFACE`. В приложении, основываясь на метаданных, можно проверить, реализует ли расширение нужный нам интерфейс, используя шаблонную функцию приведения типа `qobject_cast<T>`.

Классы расширений должны наследоваться от интерфейсов, предоставляемых приложением, и реализовывать их. Поэтому при реализации расширения его необходимо унаследовать сразу от двух классов: `QObject` и класса интерфейса, предоставляемого приложением. В классе определения расширения необходимо разместить макрос `Q_INTERFACES`, чтобы препроцессор MOC смог сгенерировать метаданные. Для экспорта данных расширения используется макрос `Q_PLUGIN_METADATA()`, который указывается в определении расширения.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/42-510/** или с помощью следующего QR-кода (рис. 42.3):

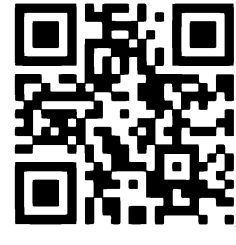


Рис. 42.3. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 43

Совместное использование Qt с платформозависимыми API

Алиса рассмеялась.
— Это не поможет! — сказала она. — Нельзя поверить в невозможное!
— Просто у тебя мало опыта, — заметила королева. — В твоем возрасте я уделяла этому полчаса каждый день. В иные дни я успевала поверить в десяток невозможностей до завтрака.

Льюис Кэрролл, «Алиса в Зазеркалье»

Несмотря на то, что библиотека Qt предоставляет практически весь инструментарий, необходимый для реализации программ, иногда возникает необходимость в использовании технологий, связанных с конкретной платформой, или необходимость в реализации кода низкого уровня. Прибегать к написанию платформозависимого кода нужно только в случаях острой необходимости. Помните, что если программа рассчитана на несколько платформ, то эти функции придется реализовывать для каждой поддерживаемой платформы. По возможности, рекомендуется весь платформозависимый код объединять в отдельных библиотеках (см. *главу 42*) — для отделения его от платформонезависимого.

Впрочем, можно выполнять и смешивание, а чтобы на той или иной системе была задействована только нужная часть кода, следует воспользоваться директивами препроцессора, определенными специально для каждой из них. Часть программы, поддерживающей несколько операционных систем, может выглядеть следующим образом:

```
#if defined(Q_OS_WIN)
// Реализация для Windows
#elif defined(Q_OS_UNIX)
// Реализация для UNIX
#elif defined(Q_OS_OSX)
// Реализация для Mac OS X
#else
// Не поддерживается
#endif
```

В табл. 43.1 указаны некоторые из препроцессорных определений.

Специальные макросы

Существуют также специальные макросы, с помощью которых можно распознать используемый на платформе C++ компилятор. Например: `Q_CC_INTEL`, `Q_CC_MSVC`, `Q_CC_MINGW`, `Q_CC_BOR` и др.

Таблица 43.1. *Препроцессорные идентификаторы платформ*

Константа	Описание
Q_OS_AIX	AIX (Advanced Interactive eXecutive) — операционная система, базирующаяся на UNIX System V, разработанная фирмой IBM
Q_OS_FREEBSD	FreeBSD — UNIX-подобная операционная система, унаследованная от UNIX-разработки фирмы AT&T
Q_OS_IRIX	IRIX — операционная система UNIX от фирмы SGI (Silicon Graphics Inc.)
Q_OS_LINUX	Linux — платформонезависимая многопользовательская операционная система, похожая на UNIX
Q_OS_OSX	Mac OS X — операционная система от фирмы Apple для компьютеров Macintosh
Q_OS_SOLARIS	SOLARIS — сертифицированная операционная система UNIX, разработанная фирмой Sun Microsystems
Q_OS_WIN32	32-битная операционная система Windows от фирмы Microsoft
Q_OS_WIN64	64-битная операционная система Windows от фирмы Microsoft
Q_OS_IOS	Мобильная операционная система iOS, разработанная Apple для iPhone, iPad и iPod touch
Q_OS_TVOS	Операционная система tvOS, разработанная Apple для «умных» телевизионных приставок AppleTV
Q_OS_WATCHOS	Операционная система watchOS, разработанная Apple для «умных» наручных часов AppleWatch
Q_OS_ANDROID	Операционная система Android, разработанная компанией Google для мобильных устройств
Q_OS_WINRT	Мобильная операционная система Windows RT от Microsoft
Q_OS_WINPHONE	Операционная система, разработанная Microsoft для смартфонов

Утилита qmake тоже предоставляет возможность отделения опций, предназначенных только для определенной платформы. В приведенном далее фрагменте pro-файла сборка библиотек выполняется в различных каталогах отдельно для каждой платформы, используются также специфичные для каждой из этих платформ библиотеки и ваши платформные реализации, а для Mac OS X и все необходимые фреймворки (frameworks). Реализации для Mac OS X в большинстве случаев содержатся в mm-файлах, которые написаны на языке Objective C++.

```
macx {
    DESTDIR = ../../lib/mac
    LIBS += -lAnyMacOSXLib -framework AnyFrameworkName
    OBJECTIVE_SOURCES += myfile_mac.mm
}
win32 {
    DESTDIR = ../../lib/win32
    LIBS += -lAnyWin32Lib
    SOURCE += myfile_win.cpp
}
!win32:!macx {
    DESTDIR = ../../lib/x11
```

```
    LIBS += -lAnyUnixLib
    SOURCES += myfile_lin.cpp
}
```

Есть также возможность более тонкого отделения опций, предназначенных только для определенной платформы, с учетом использования компилятора. Например, мы можем указать платформу и необходимый компилятор для файла:

```
win32-msvc.net {
    SOURCES += MyDotNetFile_win.cpp
}
```

Если количество опций не превышает одной, то можно воспользоваться вместо фигурных скобок `{}` оператором `:`. Таким образом мы можем предписать, например, чтобы для Windows в режиме отладки у приложения имелось для текстового вывода окно консоли:

```
win32:debug:CONFIG += console
```

Более подробную информацию об использовании утилиты qmake можно найти в *главе 3*.

Совместное использование с Windows API

Программа (листинг 43.1), окно которой показано на рис. 43.1, демонстрирует возможность использования функций GDI (Graphical Device Interface, интерфейс графического устройства) для графического вывода в ОС Windows. Аналогично можно реализовать рисование внутри окна виджета при помощи GDI+ или DirectX. По нажатию правой кнопки мыши в области окна приложения посредством Windows API вызывается окно сообщения.

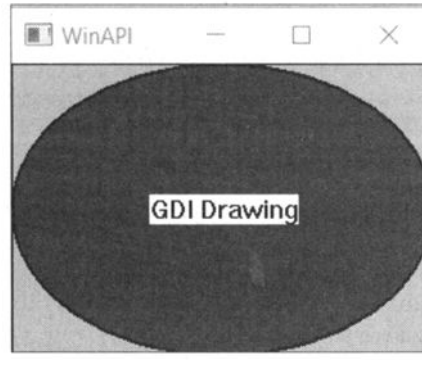


Рис. 43.1. Программа с использованием функций GDI

В листинге 43.1 приведен специальный метод обработки событий для ОС — `nativeEvent()`. Если не требуется дальнейшей обработки события с помощью библиотеки Qt, то из этого метода нужно вернуть значение `true`. В примере в конце метода мы возвращаем значение вызывного метода `nativeEvent()`, унаследованного класса `QWidget`. Реализация этого метода, по своей сути, очень похожа на реализацию *оконной функции* ОС Windows. В нашем случае отслеживается событие нажатия правой кнопки мыши и, когда оно происходит, вызывается функция `MessageBox()` Windows API, отображающая окно сообщения. Первым параметром в эту функцию в качестве родительского окна передается значение идентификатора окна Windows, возвращаемое методом `winId()`. Метод поддерживается для всех платформ и, в случае ОС Windows, возвращает идентификационный номер окна, соответствующий типу `HWND` (указатель на окно). Каждый виджет обладает своим собственным `HWND`.

В методе события перерисовки `paintEvent()` надпись и эллипс отображаются при помощи функций GDI. Обратите внимание на то, как мы получили идентификатор окна вызовом метода `QWidget::effectiveWinId()`, преобразовали его к типу `HWND` и вызовом глобальной функции Windows API `GetDC()` получили значение типа `HDC` (указатель на Device Context, контекст устройства), который нужен функциям GDI, чтобы они могли выполнять рисование. В конструкторе мы отключаем автоматическое заполнение фоном (метод `setAutoFillBackground()`) и включаем режим рисования в окне напрямую без закулисного хранения. Также обратите внимание на перегруженный метод `paintEngine()`, который возвращает указатель на ноль, и это значит, что мы не нуждаемся в этом виджете в движке для рисования и полностью берем контроль за отображения на себя.

Листинг 43.1. Класс `WinAPI` (файл main.cpp)

```
class WinAPI : public QWidget {
protected:
    virtual bool nativeEvent(const QByteArray& baType,
                             void* pmessage,
                             long* result
                            )
    {
        QString str1 = "Windows Version = "
                     + QString::number(QSysInfo::WindowsVersion);
        QString str2 = "Windows Message";

        WId         id     = effectiveWinId();
        HWND        hWnd   = (HWND)id;
        MSG* pmsg = reinterpret_cast<MSG*>(pmessage);

        switch(pmsg->message) {
        case WM_RBUTTONDOWN:
            ::MessageBox(hWnd,
                         (const WCHAR*)str1.utf16(),
                         (const WCHAR*)str2.utf16(),
                         MB_OK | MB_ICONEXCLAMATION
                        );
            break;
        default:
            ;
        }

        return QWidget::nativeEvent(baType, pmessage, result);
    }

    virtual void paintEvent(QPaintEvent*)
    {

        WId         id          = effectiveWinId();
        HWND        hWnd        = (HWND)id;
        HDC         hDC         = ::GetDC(hWnd);
```

```
        HBRUSH     hBrush     = ::CreateSolidBrush(RGB(255, 0, 255));
        HBRUSH     hBrushRect = ::CreateSolidBrush(RGB(200, 200, 200));
        HPEN       hPen       = ::CreatePen(PS_SOLID, 2, RGB(0, 0, 128));
        QString    str        = "GDI Drawing";
        TEXTMETRIC tm;

        ::SelectObject(hDC, hBrushRect);
        ::Rectangle(hDC, 0, 0, width(), height());
        ::GetTextMetrics(hDC, &tm);
        ::SelectObject(hDC, hBrush);
        ::SelectObject(hDC, hPen);
        ::Ellipse(hDC, 0, 0, width(), height());
        ::TextOut(hDC,
                  width() / 2 - (tm.tmAveCharWidth * str.length()) / 2,
                  (height() - tm.tmHeight) / 2,
                  (const WCHAR*)str.utf16(),
                  str.length()
                 );
        ::DeleteObject(hBrushRect);
        ::DeleteObject(hBrush);
        ::DeleteObject(hPen);
        ::ReleaseDC(hWnd, hDC);
    }

    virtual void resizeEvent(QResizeEvent* pe)
    {
        update();
        QWidget::resizeEvent(pe);
    }

public:
    WinAPI(QWidget* pwgt = 0) : QWidget(pwgt)
    {
        setAutoFillBackground(false);
        setAttribute(Qt::WA_PaintOnScreen, true);
    }

    QPaintEngine* paintEngine() const
    {
        return 0;
    }
};
```

Конвертирование строк из `QString` в листинге 43.1 выполняется следующим способом: `(const WCHAR*)str.utf16()`. Таблицы 43.2–43.3 демонстрируют, как можно конвертировать и в другие строковые типы, используемые в Windows (в табл. 43.3 показано конвертирование в сторону, обратную конвертированию из табл. 43.2).

Таблица 43.2. *Конвертирование из* `QString` *в другие строковые типы*

Тип строки	Пример конвертирования
C++ Standard String	`std::wstring wstr = qstr.toStdWString()`
BSTR (OLE String)	`BSTR bstr = ::SysAllocString(qstr.utf16())`
8bit (LPCSTR)	`LPCSTR lstr = qstr.toLocal8Bit().constData()`
LPCWSTR	`LPCWSTR wstr = qstr.utf16()`
CString (MFC)	`CString mfcstr = qstr.utf16()`

Таблица 43.3. *Конвертирование из других строковых типов в* `QString`

Тип строки	Пример конвертирования
C++ Standard String	`QString qstr = QString::fromStdWString(wstr)`
BSTR (OLE String)	`QString qstr((QChar*)bstr, wcslen(bstr))`
8bit (LPCSTR)	`QString qstr = QString::fromLocal8Bit(lstr)`
LPCWSTR	`QString qstr = QString::fromUtf16(wstr)`
CString (MFC)	`QString qstr = QString::fromUtf16((LPCTSTR(mfcstr)))`

Совместное использование с Linux

Так же как и для Windows, в ОС UNIX/Linux библиотека Qt предоставляет возможность доступа к событиям на низком уровне. Класс `QWidget` содержит метод `nativeEvent()`, который необходим для получения событий оконной системы X Window. Чтобы получать события, этот метод нужно переопределить. Если после завершения метода не требуется продолжать обработку события методами Qt, то из этого метода нужно вернуть значение `true`.

Кроме того, в состав библиотеки Qt входит модуль Qt X11 Extras, который содержит класс `QX11Info` и предоставляет информацию, специфичную для платформы X11. Для того чтобы задействовать этот модуль, необходимо включить в проектный файл строку: `QT += x11extras`.

Совместное использование с Mac OS X

Реализовывать платформозависимый код для Mac OS X удобнее всего на языке Objectve C++ (он является синтезом Objective C и C++). Этот язык очень хорошо продуман, и вы можете его рассматривать как улучшенный язык C++. В силу того, что фактически это надстройка над C++, для Objective C++ можно использовать те же самые h-файлы, что и для C++, естественно, задействуя макрос `Q_OS_OSX` в нужных местах.

Сам же язык Objective C по сравнению с языком C++ (без библиотеки Qt) обладает рядом преимуществ. Во-первых, Objective C является событийно-ориентированным языком, то есть в нем вызовы методов осуществляют пересылку сообщений объекту, в то время как в языке C++ все сводится к обычному вызову функций. Именно этот изъян языка C++ и

заполняет библиотека Qt с помощью механизма сигналов и слотов — чтобы объекты могли обмениваться сообщениями. Objective C поддерживает и работу с метаинформацией, кроме того, у объекта в процессе выполнения программы можно получить имя его класса, список методов, узнать, является ли этот класс потомком конкретного класса, и т. д. Библиотека Qt в большей части и тут заполняет этот недостаток языка C++. Все это в Objective C достигается благодаря его динамизму, при котором целый ряд решений выполняется не во время компиляции, как это происходит в C++, а откладывается на этап исполнения.

Objective C представляет собой настоящее расширение языка C объектно-ориентированными возможностями — то есть любая программа, написанная на языке C, всегда может быть откомпилирована на Objective C, в то время как для языка C++ это не так. Компилятор GCC поддерживает этот язык, а это значит, что вы можете использовать его и для Windows под управлением компилятора MinGW.

Как уже упоминалось ранее, Objective C++ — это Objective C, расширенный до уровня поддержки синтаксиса C++. Далее синтакис, не относящийся в C++, я буду называть Objective C. Следует также отметить, что язык Objective C является «родным» для компании Apple, и большинство кода программ для Mac OS X, iPhone, iPad и iPod touch пишутся на этом языке и на новом языке Swift, представляющем собой следующий виток эволюции Objective C.

Приведенный в листингах 43.2–43.7 пример демонстрирует использование элемента управления (флажка) Mac OS X (рис. 43.2).

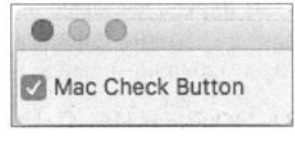


Рис. 43.2. Программа с использованием возможностей Mac OS X

В pro-файле (листинг 43.2) мы указываем файлы исходного кода на Objective C++ в секции `OBJECTIVE_SOURCES` и задаем нужный нам фреймворк `Cocoa` в секции `LIBS`.

Листинг 43.2. Использование флажка Mac OS X (файл MacButton.pro)

```
TEMPLATE = app
QT       += widgets
TARGET   = MacButton
macx {
    OBJECTIVE_SOURCES += MacButton.mm
    LIBS += -framework Cocoa
    HEADERS += MacButton.h
}

SOURCES += main.cpp
```

В основной программе (листинг 43.3) мы используем макрос `Q_OS_OSX` для того, чтобы разместить платформозависимый код (класс `MacButton`) и предусмотреть случай, если программу вдруг станут компилировать не на платформе Mac OS X, — тогда будет отображен виджет надписи с соответствующим сообщением.

Листинг 43.3. Использование флажка Mac OS X (файл main.cpp)

```
#include <QtWidgets>
#ifdef Q_OS_OSX

#include "MacButton.h"

#endif
int main(int argc, char** argv)
{
    QApplication app(argc, argv);

#ifdef Q_OS_OSX
    MacButton cmd;
    cmd.show();
#else
    QLabel label("This example requires Mac OS X");
    label.show();
#endif

    return app.exec();
}
```

В заголовочном файле (листинг 43.4) определен класс `ButtonContainer`, который мы наследуем от класса `QMacCocoaViewContainer`, чтобы использовать элемент управления кнопки Mac OS X. Этот класс находится в заголовочном файле qmaccocoaviewcontainer_mac.h, который мы включаем при помощи директивы `#import`. Эта директива языка Objective C полностью аналогична директиве `#include` языка C с той лишь разницей, что она гарантирует включение h-файла только один раз, что устраняет необходимость использования в самом заголовочном файле директивы типа:

```
#pragma once
```

Класс `QMacCocoaViewContainer`, к сожалению, не может использоваться в качестве виджета верхнего уровня, поэтому мы определяем класс `MacButton`, на поверхности которого и разместим виджет класса `ButtonContainer`.

> ***Реализации для других платформ***
>
> Если бы нам нужно было сделать реализации и для других платформ, то мы могли бы использовать тот же h-файл и продолжить описание классов после окончания действия макроса `Q_OS_OSX`.

Листинг 43.4. Определение класса `ButtonContainer` (файл MacButton.h)

```
#pragma once

#include <QtWidgets>

#import <qmaccocoaviewcontainer_mac.h>

// ========================================================================
class ButtonContainer : public QMacCocoaViewContainer {
Q_OBJECT
```

```
public:
    ButtonContainer(QWidget* pwgt = 0);

    QSize sizeHint() const;
};

// ============================================================================
class MacButton : public QWidget {
Q_OBJECT
public:
    MacButton(QWidget* pwgt = 0);
};
```

В конструкторе класса (листинг 43.5) нам нужно реализовать несколько строк на языке Objective C. Многие объекты фреймворка `Cocoa` создают временные объекты, поэтому нам надо создать объект пула (класс `NSButton`) для управления памятью и сбора этих объектов. Для создания объектов в языке Objective C необходимо выслать сначала сообщение `alloc`, а затем сообщение `init`. При этом сообщение `alloc` вы можете рассматривать как эквивалент оператора `new` на языке C++, а сообщение `init` — как эквивалент вызова конструктора. Так же мы поступаем при создании элемента управления кнопки флажка (класс `NSButton`). Далее мы высылаем созданному объекту кнопки сообщения:

- `setButtonType` с параметром значения `NSSwitchButton` — делает кнопку кнопкой флажка;
- `setTitle` с параметром строки — устанавливает в кнопке надпись;
- `setState` с параметром `YES` (в языке C++ эквивалентом для `YES` является значение `true`) — устанавливает кнопку флажка во включенное состояние.

Затем мы устанавливаем кнопку (указатель `pcmd`) в контейнере просмотра для фреймворка `Cocoa` вызовом метода `setCocoaView()`.

Пересылка сообщения `release` объекту кнопки (указатель `pcmd`) осуществляет освобождение ссылок, это мы делаем потому, что класс объекта владельца кнопки класса `ButtonContainer` и ссылка на нее нам больше не нужны. Мы выполняем очистку нашего пула пересылкой сообщения `relase` его объекту (указатель `ppool`).

> ***Программный код на Objective C***
>
> Если вы до этого не видели программный код на Objective C, то наверняка можете пребывать немного в растерянности, и сама программа может показаться вам несколько непривычной, непонятной или сложной. Это лишь первое, ошибочное впечатление. На самом деле, язык Objective C прост в изучении, и вам понадобится не более двух дней, чтобы его освоить, после чего вы сможете понимать его и писать на нем собственный программный код. Простота в изучении — это одна из сильных сторон языка Objective C, а изучение такого сложного языка, как, например, C++, потребует во много раз больше времени.

Листинг 43.5. Конструктор класса `ButtonContainer` (файл ButtonContainer.mm)

```
ButtonContainer::ButtonContainer(QWidget* pwgt/*=0*/)
    : QMacCocoaViewContainer(0, pwgt)
{
    NSAutoreleasePool* ppool = [[NSAutoreleasePool alloc] init];

    NSButton* pcmd = [[NSButton alloc] init];
```

```
    [pcmd setButtonType:NSSwitchButton];
    [pcmd setTitle:@"Mac Check Button"];
    [pcmd setState:YES];

    setCocoaView(pcmd);

    [pcmd release];
    [ppool release];
}
```

В методе `sizeHint()` (листинг 43.6) мы возвращаем рекомендуемые размеры, которые будет использовать менеджер размещения.

Листинг 43.6. Метод `sizeHint()` (файл ButtonContainer.mm)

```
QSize ButtonContainer::sizeHint() const
{
    return QSize(150, 40);
}
```

В листинге 43.7 при помощи менеджера вертикальной компоновки мы размещаем созданный нами виджет контейнера на поверхности виджета `MacButton`.

Листинг 43.7. Конструктор класса `MacButton` (файл ButtonContainer.mm)

```
MacButton::MacButton(QWidget* pwgt) :QWidget(pwgt/*=0*/)
{
    ButtonContainer* pcmd = new ButtonContainer(this);

    QVBoxLayout* pvbx = new QVBoxLayout;
    pvbx->setContentsMargins(0, 0, 0, 0);
    pvbx->addWidget(pcmd);
    setLayout(pvbx);
}
```

В качестве строк в Mac OS X используются объекты `NSString`. Конвертировать строки из объектов `NSString` в `QString` можно следующим образом:

```
NSString* ns  = @"Convert Me To QString";
QString   str = QString::fromNSString(ns);
```

А для того чтобы ковертировать из `QString` в `NSString`, можно использовать по аналогии метод `toNSString()`. Например:

```
QString   str = @"Convert Me To NSString";
NSString* ns  = str.toNSString();
```

В табл. 43.4 приведены остальные методы конвертирования данных из Cocoa в Qt, а также и в обратном направлении.

Как видно из табл. 43.4, класс `QImage` не предоставляет методов для конвертирования в `NSImage` или `CGImage`. Для того чтобы сделать это, нужно использовать отдельный мо-

Таблица 43.4. *Конвертирование из Qt в Cocoa и наоборот*

Cocoa-тип	Qt-тип	Конвертирование из Cocoa в Qt	Конвертирование из Qt в Cocoa
NSString и CFString	QString	fromNSString(), fromCFString()	toNSString(), toCFString()
NSData и CFData	QByteArra	fromNSData(), fromCFData()	toNSData(), toCFData()
NSUUID и CFUUID	QUuid	fromNSUUID(), fromCFUUID()	toNSUUID(), toCFUUID()
NSDate и CFDate	QDateTime	fromNSDate(), fromCFDate()	toNSDate(), toCFDate()
NSURL и CFURL	QUrl	fromCFURL(), fromCFURL()	toCFURL(), toCFURL()
CGImage и NSImage	QImage	-	toNSImage(), toCGImage()
CGPoint	QPointF	fromCGPoint()	toCGPoint()
CGSize	QSizeF	fromCGSize()	toCGSize()

дуль QtMac, который следует включить в проектный файл следующим образом: `QT += macextras`. Этот модуль предоставляет пространство имен `QtMac` с функциями `fromNSImage()` и `fromCGImageRef()` для конвертирования `NSImage` и `CGImage` в объекты класса `QPixmap` соответственно. Получив объекты `QPixmap`, при необходимости мы можем преобразовать их в объекты класса `QImage`. Это достигается вызовом метода `QPixmap::toImage()`.

Системная информация

Иногда бывает так, что приложению необходимо знать информацию о платформе, на которой оно запущено, — например, можно ограничить исполнение приложения до какой-то определенной версии операционной системы. Ведь вы могли протестировать его до выхода той или иной версии, и не можете гарантировать, что на операционной системе, которая может появиться в будущем, ваше приложение будет работать корректно и без побочных эффектов. На такой случай вы можете просто отобразить диалоговое окно, которое сообщит пользователю, что для текущей операционной системы эта версия приложения не тестировалась и может работать некорректно, и предложит пользователю загрузить из Всемирной паутины более актуальную версию вашей программы.

Класс `QSysInfo` предоставляет системную информацию в платформонезависимой форме и для целей получения версии операционных систем имеет следующие статические константы и методы:

- константа `WindowsVersion` — значение номера операционной системы Windows. Например: `WV_XP`, `WV_VISTA`, `WV_WINDOWS7`, `WV_WIDOWS8_1`, `WV_WINDOWS10` и т. д.;
- константа `MacintoshVersion` — значение номера операционной системы Mac OS X, а также и iOS. Например: `MV_SNOWLEOPARD`, `MV_LION`, `MV_MAVERICKS`, `MV_IOS_6_0`, `MV_IOS_7_1` и т. д.

Внимательный читатель уже наверняка обнаружил, что в приведенном списке не хватает операционной системы Linux. Это связано с тем, что до сих пор не достигнуто общего согласия о нумерации дистрибутивов различных версий Linux.

Следующий пример показывает, как получить номера версии для Windows, Mac OS X и iOS (листинг 43.8).

Листинг 43.8. Имя и номер версии операционной системы

```
QString strOSInfo = "";
#ifdef Q_OS_OSX
    int nVer = QSysInfo::macVersion();
    QString strVer = (nVer == QSysInfo::MV_10_5) ? "10.5 Leopard" :
                     (nVer == QSysInfo::MV_10_6) ? "10.6 Snow Leopard" :
                     (nVer == QSysInfo::MV_10_7) ? "10.7 Lion" :
                     (nVer == QSysInfo::MV_10_8) ? "10.8 Mountain Lion" :
                     (nVer == QSysInfo::MV_10_9) ? "10.9 Mavericks" :
                     (nVer == QSysInfo::MV_10_10) ? "10.10 Yosemite" :
                     (nVer == QSysInfo::MV_10_11) ? "10.11 EI Capitain" :
                     (nVer == QSysInfo::MV_10_12) ? "10.12 Sierra" :
                     ("(" + QString::number(nVer - 2) + ") Unknown");
    strOSInfo = "Mac OS X " + strVer;
#elif defined(Q_OS_WIN)
    int nVer = QSysInfo::windowsVersion();
    QString strVer = (nVer == QSysInfo::WV_5_0) ? "2000" :
                     (nVer == QSysInfo::WV_5_1) ? "XP" :
                     (nVer == QSysInfo::WV_5_2) ? "2003" :
                     (nVer == QSysInfo::WV_6_0) ? "Vista" :
                     (nVer == QSysInfo::WV_6_1) ? "7" :
                     (nVer == QSysInfo::WV_6_2) ? "8" :
                     (nVer == QSysInfo::WV_6_3) ? "8.1" :
                     (nVer == QSysInfo::WV_10_0) ? "10" :
                     ("(" + QString::number(nVer) + ") Unknown");
    strOSInfo = "Win " + strVer;
#elif defined(Q_OS_WINRT) || defined(Q_OS_WINPHONE)
    QString strVer = (nVer == QSysInfo::WV_6_2) ? "8" :
                     (nVer == QSysInfo::WV_6_3) ? "8.1" :
                     (nVer == QSysInfo::WV_10_0) ? "10" :
                     (nVer == QSysInfo::WV_None) ? "???" :
                     ("(" + QString::number(nVer) + ")");
    #ifdef Q_OS_WINRT
        strOSInfo = "WinRT " + strVer;
    #else
        strOSInfo = "WinPhone " + strVer;
    #endif
#elif defined(Q_OS_IOS)
    int nVer = QSysInfo::macVersion();
    QString strVer = (nVer == QSysInfo::MV_IOS_4_3) ? "4.3" :
                     (nVer == QSysInfo::MV_IOS_5_0) ? "5.0" :
                     (nVer == QSysInfo::MV_IOS_5_1) ? "5.1" :
                     (nVer == QSysInfo::MV_IOS_6_0) ? "6.0" :
                     (nVer == QSysInfo::MV_IOS_6_1) ? "6.1" :
                     (nVer == QSysInfo::MV_IOS_7_0) ? "7.0" :
```

```
                    (nVer == QSysInfo::MV_IOS_7_1) ? "7.1" :
                    (nVer == QSysInfo::MV_IOS_8_0) ? "8.0" :
                    (nVer == QSysInfo::MV_IOS_8_1) ? "8.1" :
                    (nVer == QSysInfo::MV_IOS_8_2) ? "8.2" :
                    (nVer == QSysInfo::MV_IOS_8_3) ? "8.3" :
                    (nVer == QSysInfo::MV_IOS_9_0) ? "9.0" :
                    (nVer == QSysInfo::MV_IOS_9_1) ? "9.1" :
                    (nVer == QSysInfo::MV_IOS_9_2) ? "9.2" :
                    (nVer == QSysInfo::MV_IOS_9_3) ? "9.3" :
                    (nVer == QSysInfo::MV_IOS_10_0) ? "10.0" :
                    ("(" + QString::number(nVer) + ") Unknown");
    strOSInfo = "iOS " + strVer;
#elif defined(Q_OS_ANDROID)
    strOSInfo = "Android";
#elif defined(Q_OS_LINUX)
    strOSInfo = "Linux";
#else
    strOSInfo = "Unknown";
#endif
qDebug() << "OS and Version:" << strOSInfo;
```

Класс `QSysInfo` также предоставляет информацию о том, какой порядок байтов применяется в операционной системе. Эту информацию можно получить, используя значение `QSysInfo::ByteOrder`. Например:

```
if  (QSysInfo::ByteOrder == QSysInfo::BigEndian) {
    qDebug() << "System is big endian";
}
else {
    qDebug() << "System is little endian";
}
```

Резюме

Библиотека Qt допускает возможность использования в своих программах платформозависимого кода. Это может быть полезно для реализации программ, использующих возможности, не предоставляемые библиотекой Qt. При помощи макросов или секций pro-файла вы можете снабдить код вашей программы платформозависимой реализацией, которая может находиться как внутри файла, так и в отдельно предназначенных для этого файлах.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/43-510/** или с помощью следующего QR-кода (рис. 43.3):

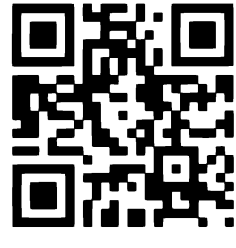


Рис. 43.3. QR-код для доступа на страницу комментариев к этой главе

ГЛАВА 44

Qt Designer. Быстрая разработка прототипов

> Наш век беспокойства в значительной мере является результатом попыток выполнить сегодняшнюю работу вчерашними средствами...
>
> *Маршал Мак-Люган*

Программа Qt Designer — это средство быстрой разработки приложений (Rapid Application Development, RAD). Прежде всего, этот инструмент предназначен для дизайнеров, и принцип его работы отвечает принципу WYSIWYG (What You See Is What You Get, «что видишь, то и получишь»). Он предоставляет возможность быстро создавать прототипы приложений, которые базируются на диалоговых окнах, а также могут иметь главное окно, меню, строку состояния и панель инструментов. Созданные в программе Qt Designer файлы описания интерфейса можно конвертировать в исходный код на языке C++. Кроме виджетов, уже содержащихся в библиотеке Qt, программа Qt Designer может дополняться виджетами, созданными самим разработчиком.

В этой главе для демонстрации возможностей Qt Designer мы создадим простое приложение, позволяющее изменять значения виджета электронного индикатора `QLCDNumber`.

Создание новой формы в Qt Designer

Окно программы Qt Designer (рис. 44.1) содержит меню и панели инструментов. Панель инструментов предоставляет общие операции для редактирования формы: скопировать, вставить и т. д., а также и режимы редактирования и размещения, о которых речь пойдет далее. Эти команды также доступны и в основном меню.

В левой части окна Qt Designer расположено окно **Widget Box** (Виджеты), в котором можно найти объекты компоновки и сами виджеты, сгруппированные в отдельные категории. Именно из этого окна посредством перетаскивания добавляются элементы на форму.

Справа расположены сразу пять окон:

- **Object Inspector** (Объекты) — содержит список используемых виджетов. В этом окне их можно выбирать для последующего изменения. Например, при помощи **Property Editor** (Редактор свойств) изменять их свойства;
- **Property Editor** (Редактор свойств) — определяет ряд свойств выбранного виджета. Это могут быть цвет фона, шрифт, максимальный/минимальный размер виджета и т. д.;
- **Signal/Slot Editor** (Редактор сигналов и слотов) — окно редактирования соединений сигналов со слотами;

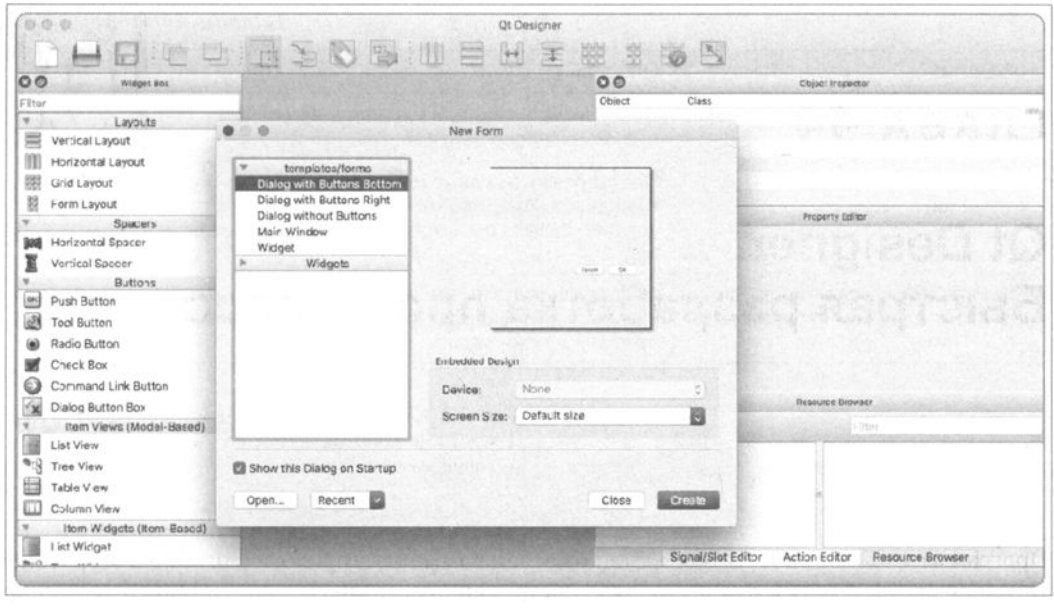


Рис. 44.1. Окно программы Qt Designer

- **Resource Browser** (Броузер ресурсов) — с помощью этого окна можно найти существующий ресурс;
- **Action Editor** (Редактор действий) — окно предназначено для создания и удаления действий команд создаваемой формы и управления ими.

> ***Еще об окнах Action Editor, Signal/Slot Editor и Resource Browser***
>
> Три последних окна: **Action Editor**, **Signal/Slot Editor** и **Resource Browser** — сгруппированы по умолчанию в виде вкладок, и при необходимости с помощью перетаскивания (drag&drop) их можно поместить в собственные области видимости.

После запуска программы в диалоговом окне **New Form** (Новая форма) будет предложен на выбор один из шаблонов (см. рис. 44.1). Если вам не нужно отображение этого окна при последующих запусках, то следует снять с него флажок **Show this Dialog on Startup** (Показывать это окно при старте). Впоследствии для вызова этого диалогового окна можно выбрать в меню команду **File | New...** (Файл | Новая) или нажать комбинацию «горячих» клавиш <Ctrl>+<N>.

В диалоговом окне **New Form** (Новая форма) предлагается пять вариантов создания форм. Выберите опцию **Widget** (Виджет), после чего будет создано окно нового виджета (рис. 44.2).

По умолчанию в программе Qt Designer установлен режим редактирования виджетов формы. В Qt Designer этих режимов четыре (табл. 44.1).

***Таблица 44.1.** Режимы редактирования*

Название	Описание	Вид
Edit Widgets (Редактирование виджетов)	В этом режиме можно изменять внешний вид формы, добавляя в нее, например, виджеты и компоновки, или редактируя свойства каждого виджета	

Таблица 44.1 (окончание)

Название	Описание	Вид
Edit Signals/Slots (Редактирование сигналов/слотов)	Этот режим позволяет соединять виджеты для обмена сообщениями	
Edit Buddies (Редактирование поручений)	В этом режиме виджет может быть «поручен» виджету надписи. Это делается для того чтобы этот виджет получал фокус тогда, когда его получит элемент надписи	
Edit Tab Order (Редактирование порядка следования табулятора)	Этот режим служит для установки порядка, в котором виджеты получают фокус клавиатуры при нажатии на клавишу табуляции	

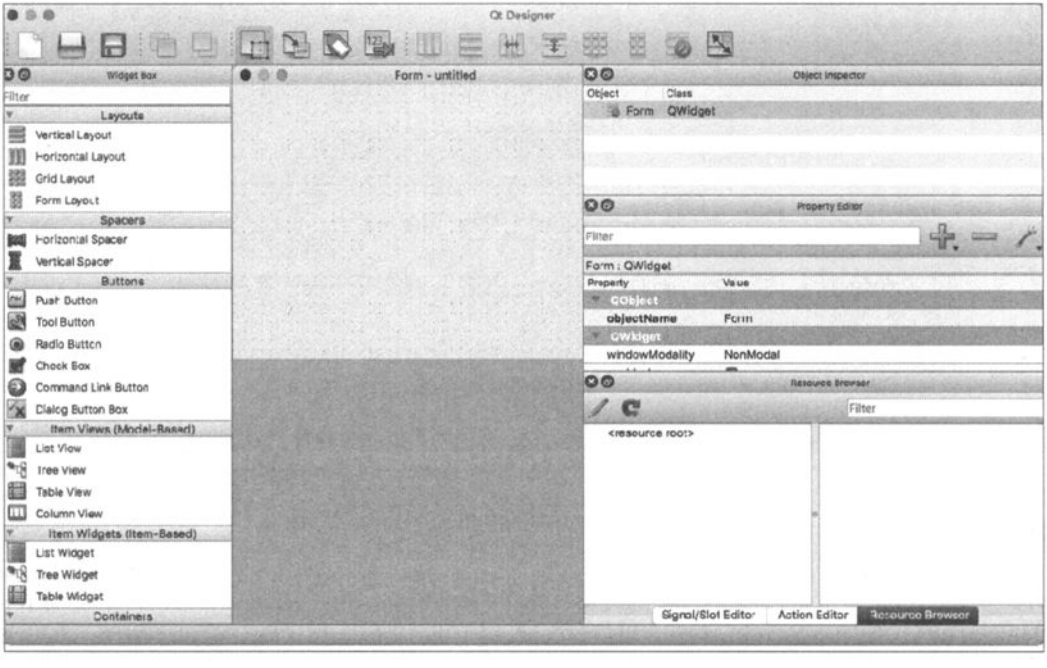


Рис. 44.2. Окно нового виджета

При помощи окна **Property Editor** (Редактор свойств) (рис. 44.3) в поле **windowTitle** (Заголовок окна) можно изменить заголовок окна виджета **Form**, например, на **DesignedWidget**. Нелишним будет в поле **objectName** (Имя объекта) задать имя, которое будет использовано при его соединении с сигналами и слотами других компонентов. Для нашей формы укажем имя **MyForm**.

> ***Дальнейшее использование имени формы и имен ее элементов***
>
> При обработке сохраненного файла формы (ui-файла) утилитой uic будет создан код на языке C++, в котором имя формы будет использовано для имени класса, а имена размещенных на ней элементов — для названий атрибутов этого класса.

Редактор свойств отображает свойства выделенного виджета, расположенного на форме, а если ни один из виджетов не выделен, то отображаются свойства самой формы. Поэтому для редактирования нужного виджета его необходимо сначала выделить.



Рис. 44.3. Окно свойств: изменение заголовка окна виджета

Для сохранения созданного диалогового окна выберите команду меню **File | Save As...** (Файл | Сохранить как...) или нажмите комбинацию «горячих» клавиш <Ctrl>+<S>.

Добавление виджетов

Чтобы добавить в диалоговое окно новые виджеты, нужно воспользоваться окном виджетов **Widget Box** (Виджеты) (см. рис. 44.2). Выберите нужный вам виджет, нажмите на нем левую кнопку мыши и перетащите его в область формы виджета окна на нужное вам место.

С вкладки **Buttons** (Кнопки) перетащите две кнопки **PushButton** (Кнопка), с вкладки **Input Widgets** (Виджеты ввода) — **Horizontal Slider** (Горизонтальный ползунок), а с вкладки **Display Widgets** (Виджеты отображения) — **LCDNumber** (Электронный индикатор) и разместите их в окне формы. Обратите внимание, что размещенные на форме виджеты стали видны не только в ней, но и в окне **Object Inspector** (Объекты). В этом окне виджеты отображаются в иерархическом виде (рис. 44.4).

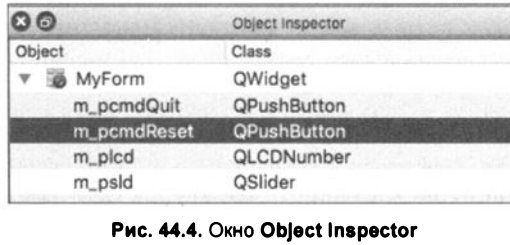


Рис. 44.4. Окно **Object Inspector**

Выберите одну из созданных кнопок, щелкнув на ней левой кнопкой мыши в окне **Object Inspector** (Объекты) или в самом диалоговом окне. Введите в поле **text** (Текст) окна **Property Editor** (Редактор свойств) текст `&Reset`, а в поле **objectName** (Имя объекта) — `m_pcmdReset`. Повторите ту же операцию со второй кнопкой, но в поле **objectName** (Имя объекта) введите `m_pcmdQuit`, а в поле **text** (Текст) — `&Quit`. Изменения, вносимые в поле **objectName** (Имя объекта), используются в дальнейшем для различения виджетов: это их имена, которыми вы будете пользоваться. Переименуйте также ползунок и индикатор (см. рис. 44.4). После этого окно формы должно выглядеть примерно так, как показано на рис. 44.5.

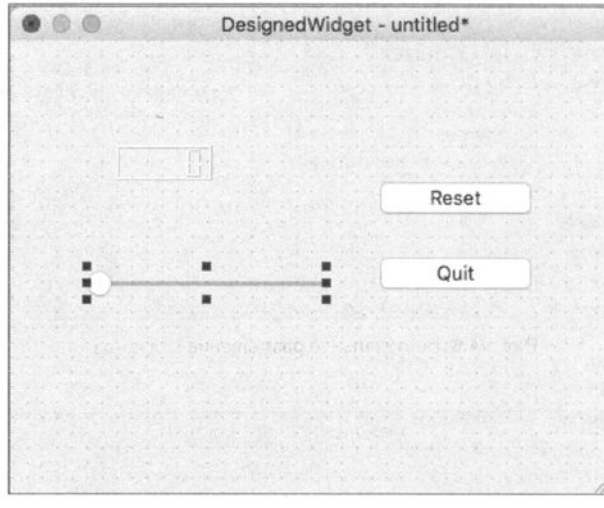


Рис. 44.5. Форма с добавленными виджетами

Компоновка (layout)

Добавив все нужные виджеты, можно заняться их размещением. Для этого поместите указатель мыши в область диалогового окна, нажмите левую кнопку и, не отпуская, переместите указатель — вы увидите рамку. Охватите этой рамкой виджет ползунка и виджет электронного индикатора. Отпустите левую кнопку мыши, и эти виджеты окажутся выделенными. Нажмите на выделенных элементах правую кнопку мыши, и вы увидите контекстное меню, в котором выберите команду **Lay out | Lay out Vertically** (Размещение | Разместить по вертикали) или просто нажмите комбинацию «горячих» клавиш <Ctrl>+<2>. Проделайте то же самое с кнопками **Reset** (Сброс) и **Quit** (Выход).

В места, которые должны оставаться свободными, следует поместить *заполнитель пространства* (Spacer). Для этого в окне **Widget Box** (Виджеты) на вкладке **Spacers** (Заполнители) выберите вертикальный заполнитель **Vertical Spacer** (Вертикальный заполнитель) и поместите его под самую нижнюю кнопку. У вас должно получиться так же, как показано на рис. 44.6.

Последний, завершающий штрих — поручите диалоговому окну размещать свои элементы в горизонтальном порядке. Для этого щелкните на одной из пустых областей диалогового окна указателем мыши или выберите опцию **My Form** в окне **Object Inspector** (Объекты). Затем нажмите правую кнопку мыши и выберите из контекстного меню команду **Lay out |**

Lay out Horizontally (Размещение | Разместить по горизонтали) или нажмите комбинацию «горячих» клавиш <Ctrl>+<1>. Окно изменится и примет вид, показанный на рис. 44.7. Теперь, при изменении размеров окна, размеры и позиции виджетов будут автоматически изменяться, заполняя всю рабочую площадь.

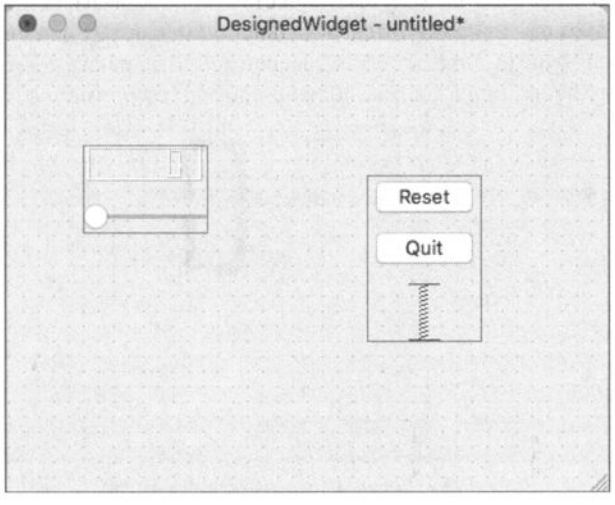


Рис. 44.6. Вертикальное размещение виджетов

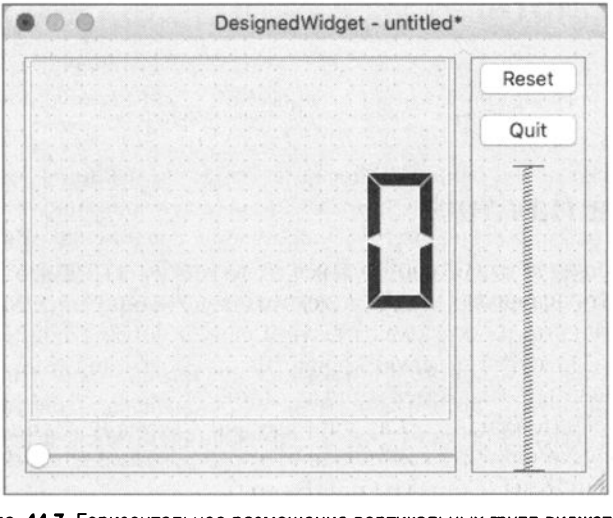


Рис. 44.7. Горизонтальное размещение вертикальных групп виджетов

Порядок следования табулятора

Порядок следования табулятора нужен для навигации по окну при помощи клавиатуры — нажатие на клавишу <Tab> перемещает фокус от одного виджета к другому. Для определения порядка следования табулятора нужно установить соответствующий режим, для чего выберите команду меню **Edit | Edit Tab Order** (Редактирование | Порядок следования) или

нажмите кнопку этой команды на панели управления. Окно примет вид, показанный на рис. 44.8. Нажимая указателем мыши на виджеты, обозначенные голубыми прямоугольниками, можно менять порядок следования табулятора.

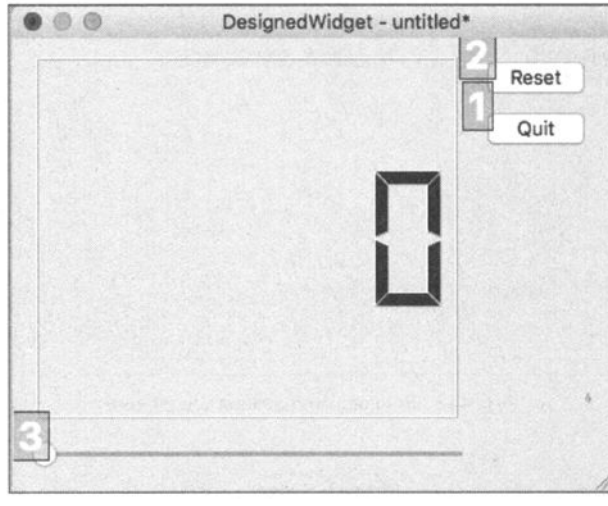


Рис. 44.8. Порядок следования табулятора

Сигналы и слоты

Для соединения сигналов одного виджета со слотами другого нужно установить соответствующий режим, для чего выберите команду меню **Edit | Edit Signals/Slots** (Редактирование | Редактирование сигналов/слотов) или нажмите клавишу <F4>.

Чтобы выполнить соединение, просто задержите указатель мыши на нужном вам элементе до тех пор, пока он не будет выделен, а затем нажмите левую кнопку и переместитесь к тому элементу, с которым должно быть осуществлено соединение. Вы увидите красную стрелку, показывающую в его сторону. Давайте задержим указатель на горизонтальном ползунке, нажмем левую кнопку мыши и переместим указатель в сторону виджета электронного индикатора.

После отпускания кнопки мыши будет показано диалоговое окно **Configure Connection** (Конфигурация соединения), предлагающее выбрать сигналы и слоты для связываемых виджетов. Выберите в левой области сигнал **valueChanged(int)**, а в правой — **display(int)** (рис. 44.9).

Чтобы закрывать окно виджета нажатием кнопки **Quit** (Выход), необходимо соединить эту кнопку со слотом формы `close()`. Выделите эту кнопку, нажмите левую кнопку мыши и переместите ее указатель в свободную область формы, но не отпускайте, пока не увидите большую стрелку. Ваши соединения должны выглядеть так, как это показано на рис. 44.10.

Отпустите кнопку мыши и выберите в правой области диалогового окна **Configure Connection** (Конфигурация соединения) сигнал `clicked()`. Для того чтобы соединить его со слотом `close()`, нужно сперва установить флажок **Show signals and slots inherited from QWidget** (Показать все сигналы и слоты, унаследованные от QWidget), в результате чего будут показаны сигналы и слоты унаследованных классов.

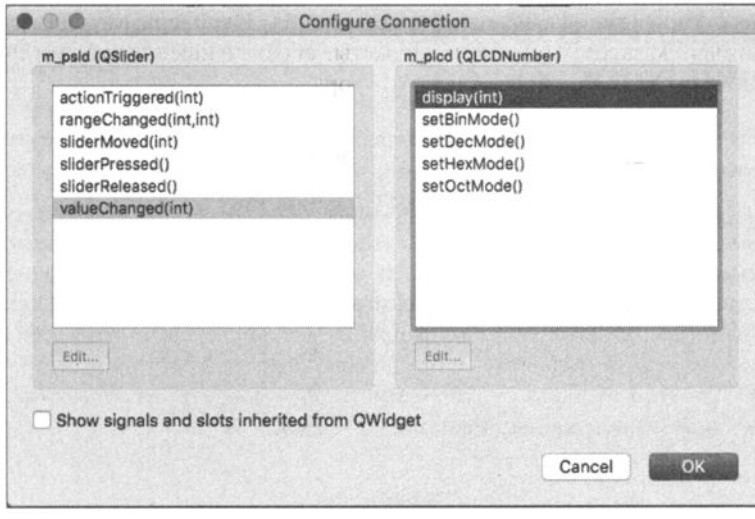


Рис. 44.9. Окно редактирования соединений

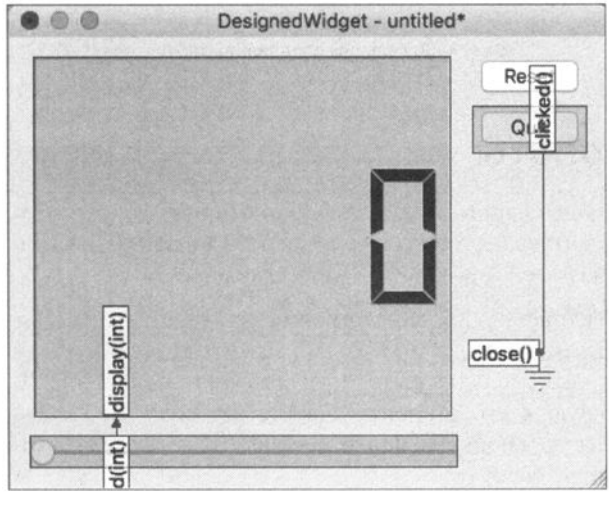


Рис. 44.10. Соединения

Чтобы полюбоваться на созданный нами виджет в действии, выберите команду меню **Form | Preview...** (Форма | Просмотр...). Теперь вы можете поэкспериментировать с изменениями положения ползунка, которые приведут к изменению информации, отображаемой виджетом электронного индикатора. Можете проверить установленный порядок следования табулятора и проследить за изменениями размеров виджетов, находящихся в компоновщике, при изменении размеров основного окна.

Если вы остались довольны просмотром виджета, то вам остается только сохранить его. Для этого нужно выбрать в меню команду **File | Save As...** (Файл | Сохранить как...), выбрать в диалоговом окне путь для сохранения формы и задать ее имя — например, MyFrom.ui.

Использование в формах собственных виджетов

Очень часто возникают ситуации, когда появляется потребность интегрировать в форму виджеты, которые были созданы в отдельном ui-файле или написаны на C++.

В подобных случаях можно поступить следующим образом: поместить в форму любой из стандартных виджетов, например QWidget, выделить его на форме, вызвать контекстное меню и выбрать в нем пункт **Promote to...**. В открывшемся диалоговом окне (рис. 44.11) вписать имя класса виджета, который вы хотите интегрировать в форму. Обратите внимание на то, чтобы название заголовочного файла, которое будет автоматически сгенерировано, соответствовало действительному, и при необходимости просто внесите в него коррективы вручную. Для того чтобы в будущем иметь возможность использовать этот виджет, нажмите кнопку **Add** и затем кнопку **Promote**.

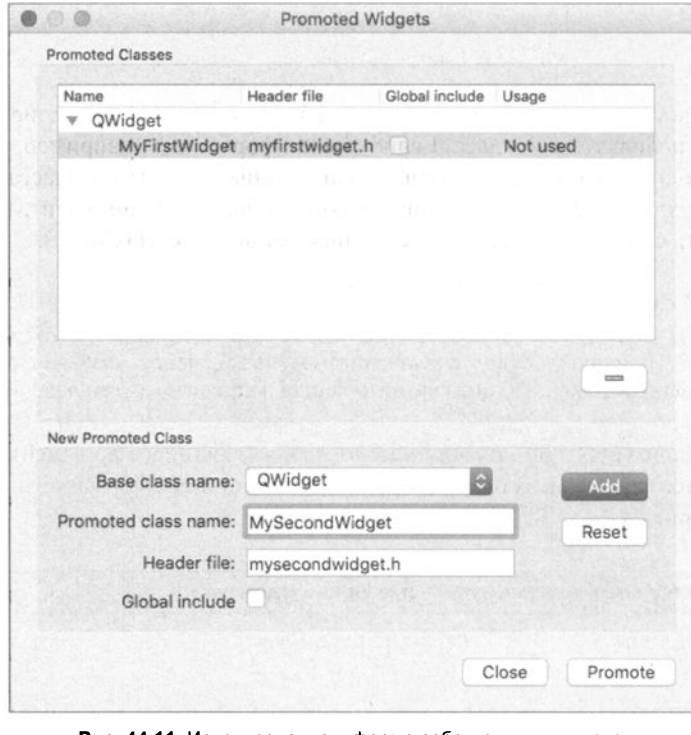


Рис. 44.11. Использование в форме собственных виджетов

Использование форм в проектах

Использование форм в проектах — это и есть то, ради чего мы их создаем. И для этого библиотека Qt предоставляет три способа.

Первый способ называют *прямым* (direct approach) — он является и самым простым. Все, что нам нужно, — это создать виджет и установить в методе `Ui::MyForm::setupUi()` созданную нами форму так, как это показано в листинге 44.1.

Листинг 44.1. Прямой способ использования формы в проекте

```
#include "ui_MyForm.h"
#include <QtWidgets>

int main(int argc, char* argv[])
{
    QApplication app(argc, argv);

    QWidget* form = new QWidget;
    Ui::MyForm ui;
    ui.setupUi(form);

    form->show();

    return app.exec();
}
```

Но недостаток такого подхода очевиден — мы не в состоянии инициализировать данные и дополнять кодом используемый класс. Если в вашей форме есть, например, виджеты, которые нуждаются в соединении со слотами, реализованными другими классами проекта, то этот способ — не для вас. Для нашего конкретного случая он не подходит, потому что нам нужно реализовать слот, с которым будет соединена кнопка **Reset** (Сброс).

> ***ФАЙЛ UI_MYFORM.H***
>
> У вас может возникнуть вопрос, откуда взять присутствующий в листинге 44.1 файл *ui_MyForm.h*? Этот файл будет создан автоматически, после указания формы в секции `FORMS` в проектном файле. Об этом читайте подробнее в следующем *разд. «Компиляция»*.

Второй способ реализуется при помощи *наследования* (inheritance approach). В этом случае мы наследуем класс от класса, за базу которого была взята наша форма, — в нашем примере это `QWidget` (листинг 44.2).

Листинг 44.2. Использование формы при помощи наследования

```
#include "ui_MyForm.h"

class MyForm : public QWidget {
    Q_OBJECT
private:
    Ui::MyForm m_ui;

public:
    MyForm(QWidget* pwgt = 0) : QWidget(pwgt)
    {
        m_ui.setupUi(this);

        connect(m_ui.m_cmdReset, SIGNAL(clicked()), SLOT(slotReset()));
    }
```

```
public slots:
    void slotReset()
    {
        m_ui.m_sld->setValue(0);
        m_ui.m_lcd->display(0);
    }
};
```

Здесь мы используем форму `Ui::MyForm` в качестве атрибута нашего класса и устанавливаем ее в конструкторе с помощью метода `Ui::MyForm::setupUi()`. Затем соединяем кнопку **Reset** (Сброс) (указатель `m_cmReset`) с реализованным нами слотом `slotReset()`. Если бы мы вдруг пришли к выводу, что наша форма нуждается еще в паре элементов, то просто добавили бы их при помощи программы Qt Designer, после чего реализовали бы недостающие слоты в нашем классе. То есть, дизайн графического интерфейса отделен от программной реализации.

Теперь давайте перейдем к последнему — третьему способу, который реализуется при помощи *множественного наследования* (multiple inheritance approach). По своей сути он очень похож на второй способ, но его достоинство заключается в том, что мы можем напрямую обращаться ко всем виджетам формы. Этот способ иллюстрирует листинг 44.3.

Листинг 44.3. Множественное наследование при использовании дизайна формы

```
#include "ui_MyForm.h"

class MyForm : public QWidget, public Ui::MyForm {
    Q_OBJECT

public:
    MyForm(QWidget* pwgt = 0) : QWidget(pwgt)
    {
        setupUi(this);

        connect(m_cmdReset, SIGNAL(clicked()), SLOT(slotReset()));
    }

public slots:
    void slotReset()
    {
        m_sld->setValue(0);
        m_lcd->display(0);
    }
};
#endif  //_MyForm_h_
```

Как видно из листинга 44.3, теперь наш класс `MyForm` наследуется не только от `QWidget`, но и от класса формы, созданной нами в программе Qt Designer.

Компиляция

Последнее, что осталось сделать, — откомпилировать проект с применением формы. Первым делом надо создать проектный файл и не забыть включить все используемые формы в секции FORMS. В нашем случае она только одна, и pro-файл должен выглядеть следующим образом:

```
TEMPLATE      = app
HEADERS      += MyForm.h
FORMS        += MyForm.ui
SOURCES      += main.cpp
QT           += widgets
win32:TARGET  = ../MyForm
```

После этого в каталогах, содержащих проектный файл, файл формы (MyForm.ui) и другие исходные файлы, выполните команды создания make-файла и компиляции:

```
qmake
make
```

В процессе компиляции из ui-файла будет автоматически создан h-файл с префиксом ui_. По завершении компилирования будет создана исполняемая программа, отображающая окно, показанное на рис. 44.12.

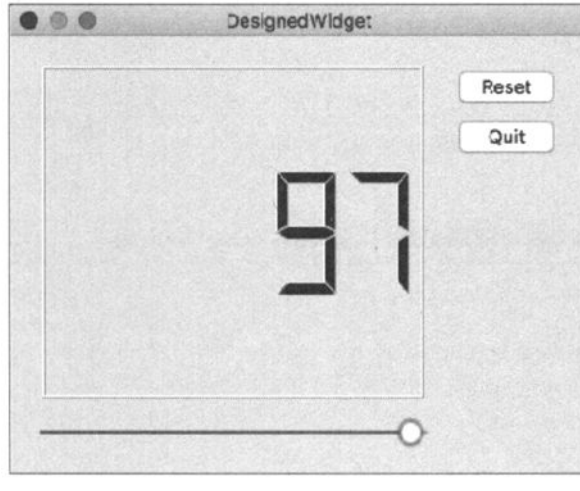


Рис. 44.12. Готовое приложение

Динамическая загрузка формы

Есть и четвертый способ использования форм, который в корне отличается от трех изложенных ранее. Его суть заключается в том, что XML-репрезентация формы (ui-файл) используется в программе как есть, без дополнительных преобразований. При помощи специального класса QUiLoader, содержащегося в модуле QtUiTools, данные репрезентации формы могут быть загружены, и в результате их интерпретации этот класс создаст соответствующий виджет. Рассмотрим класс QUiLoader в работе и создадим программу загрузки

созданной нами формы (листинги 44.4–44.6), функционально аналогичную программам из листингов 44.1–44.3.

В нашем проектном файле (листинг 44.5) должен быть подключен модуль `QtUiTools` — это мы делаем в секции `QT`. Саму форму мы располагаем в ресурсе (листинг 44.4).

Листинг 44.4. Форма (файл resource.qrc)

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>MyForm.ui</file>
</qresource>
</RCC>
```

Сам же ресурс подключаем к проекту в секции `RESOURCES` (листинг 44.5). Заметьте, что секция `FORMS` в нашем проектном файле отсутствует, так как мы будем использовать XML-данные формы напрямую.

Листинг 44.5. Проектный файл LoadMyForm.pro

```
TEMPLATE      = app
QT       += widgets uitools
HEADERS       = LoadMyForm.h
SOURCES       = main.cpp
RESOURCES     = resource.qrc
win32:TARGET  = ../LoadMyForm
```

В основной программе, показанной в листинге 44.6, мы просто создаем виджет нашего класса `LoadMyForm`.

Листинг 44.6. Создание виджета класса `LoadMyForm` (файл main.cpp)

```
#include <QApplication>
#include "LoadMyForm.h"

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    LoadMyForm   wgt;

    wgt.show();

    return app.exec();
}
```

Наш класс `LoadMyForm` унаследован от класса `QWidget` (листинг 44.7). Для того чтобы использовать виджеты `QSlider` и `QLCDNumber` в отдельном слоте, мы определяем указатели `m_psld` и `m_plcd` в качестве атрибутов класса. В конструкторе мы создаем объект класса загрузки форм `QUiLoader` (указатель `puil`) и передаем ему в качестве предка указатель `this`. Далее нам необходимо создать объект файла нашей формы, поэтому в конструкторе `QFile`

указываем имя расположенного в ресурсе файла ":/MyForm.ui". В метод load() передаем адрес объекта файла и получаем указатель на сконструированный виджет. Если попытка конструирования виджета закончилась неуспешно, а это может случиться, например, когда XML-данные содержат ошибки, то этот метод возвратит нулевое значение. Поэтому мы обязаны это значение проверить. В случае успеха приводим размеры виджета в соответствие с размерами нашей формы pwgtForm, для чего вызываем метод resize(). Теперь нам нужно получить доступ к виджетам формы, и для этой цели используется шаблонный метод findChild<Тип*>("ИмяОбъекта"), определенный в классе QObject. Таким образом мы получаем доступ к виджету ползунка (указатель m_psld), электронному индикатору (указатель m_pcd), кнопке **Reset** (указатель pcmdReset) и кнопке **Quit** (указатель pcmdQuit). Сигнал clicked() кнопки **Reset** соединяем со слотом slotReset(), в котором устанавливаются нулевые значения для виджетов ползунка и электронного индикатора. Для завершения работы приложения при нажатии кнопки **Quit** ее сигнал clicked() мы соединяем со слотом quit() объекта приложения qApp. Сам сконструированный виджет pwgtForm размещаем на поверхности нашего виджета LoadMyForm при помощи горизонтальной компоновки phbxLayout.

Листинг 44.7. Загрузка формы (файл LoadMyForm.cpp)

```
#pragma once

#include <QtWidgets>
#include <QtUiTools>

// ============================================================================
class LoadMyForm : public QWidget {
Q_OBJECT
private:
    QSlider*    m_psld;
    QLCDNumber* m_plcd;

public:
    LoadMyForm(QWidget* pwgt = 0) : QWidget(pwgt)
    {
        QUiLoader* puil = new QUiLoader(this);
        QFile      file(":/MyForm.ui");

        QWidget* pwgtForm = puil->load(&file);
        if (pwgtForm) {
            resize(pwgtForm->size());

            m_psld = pwgtForm->findChild<QSlider*>("m_sld");
            m_plcd = pwgtForm->findChild<QLCDNumber*>("m_lcd");

            QPushButton* pcmdReset =
                pwgtForm->findChild<QPushButton*>("m_cmdReset");
            connect(pcmdReset, SIGNAL(clicked()), SLOT(slotReset()));

            QPushButton* pcmdQuit =
                pwgtForm->findChild<QPushButton*>("m_cmdQuit");
            connect(pcmdQuit, SIGNAL(clicked()), qApp, SLOT(quit()));
```

```
            //Layout setup
            QHBoxLayout* phbxLayout = new QHBoxLayout;
            phbxLayout->addWidget(pwgtForm);
            setLayout(phbxLayout);
        }
    }

public slots:
    void slotReset()
    {
        m_psld->setValue(0);
        m_plcd->display(0);
    }
};
```

Резюме

Реализация пользовательского интерфейса отнимает много времени. Программа Qt Designer создана для ускорения этого процесса. Здесь с ее помощью мы создали приложение, базирующееся на диалоговом окне. К созданному диалоговому окну можно добавлять любые виджеты и изменять их свойства. При помощи компоновок можно задавать разные способы их размещения. В места, которые должны быть свободными, нужно помещать специальный заполнитель.

Программа Qt Designer предоставляет возможность быстрого просмотра созданного диалогового окна. Использовать созданные формы в своих проектах можно четырьмя разными способами:

- прямым способом;
- использованием наследования;
- использованием множественного наследования;
- динамической загрузкой ui-файла формы.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/44-510/** или с помощью следующего QR-кода (рис. 44.13):



Рис. 44.13. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 45

Проведение тестов

Сидит программист глубоко в отладке. Третий день сидит. Ничего не получается. Подходит к нему сынишка и говорит:
— Папа, а почему солнце встает на востоке?
— Ты это проверял?
— Да.
— Работает?
— Да.
— Каждый день работает?
— Да.
— Тогда сынок, ради Бога, ничего не трогай, ничего не меняй!

Тестирование — это фундамент разработки программ, позволяющий быстро продвигаться вперед.

На самом деле на написание кода тратится не так уж много времени. Много времени уходит на понимание задачи и проектирование. Ну а львиную долю занимает отладка. Каждый из читателей наверняка помнит часы, а может, и дни, которые ему пришлось провести в поисках закравшейся ошибки. Причем исправить ее можно, как правило, за считанные минуты, но поиск может отнять целую вечность.

Но и это еще не все. На исправлении ошибки история не заканчивается. Исправив ее, вы не можете дать гарантию того, что сделанное вами исправление не повлечет за собой появление других ошибок. На самом деле может оказаться так, что исправление ошибки вызовет возникновение не одной, а сразу нескольких ошибок, еще более коварных, чем исправленная.

Тесты являются решением подавляющего большинства подобных проблем. Чем чаще вы будете их выполнять, тем больше у вас шансов быстро обнаружить ошибку. Попробуйте выполнять тесты после каждой компиляции, и вы сами увидите, как резко возрастет производительность вашего труда. Вы перестанете тратить много времени на отладку, связанную с поиском ошибки. И если вы вдруг при исправлении сделали другую ошибку, то вы сразу же обнаружите ее, так как будете точно знать, что в предыдущей компиляции ее не было. В любом случае, вы без труда сможете внести необходимые исправления, снова откомпилировать свой модуль и провести следующий тест. Заметьте, важно не только создавать классы тестов, но еще и часто запускать сами тесты.

В идеале, для каждого класса должен быть написан тест. Но на практике это не всегда целесообразно. Всегда есть риск что-то пропустить, поэтому не стремитесь написать много тестов — их все равно будет недостаточно. Ваши опасения по поводу того, что тестирование

не выявит все ошибки, не должны отваживать вас от написания тестов — они в любом случае позволят обнаружить большинство ошибок. Впрочем, чрезмерное усердие при написании тестов может вызвать обратный эффект — вы решите, что написание тестов отнимает слишком много времени, и откажетесь от них. Поэтому необходимо писать тесты только для подозрительных критических мест. Подумайте, например, о граничных условиях, которые могут быть неправильно обработаны, и сосредоточьте свои тесты на них. Всегда помните — тестирование приносит ощутимую пользу, даже если осуществляется в небольшом объеме.

Для создания тестов библиотека Qt предоставляет специальный модуль `QtTest`, который разработан для того, чтобы упростить тестирование классов вашего приложения. Он также включает в себя возможности проведения тестов классов графического интерфейса и позволяет «симулировать» клавиатуру и мышь.

Тесты, о которых пойдет речь в этой главе, называются *модульными тестами* (unit tests). В подобных тестах каждый класс действует в рамках одного вашего модуля и исходит из того, что за его пределами все работает нормально. Эта процедура необходима для того, чтобы удостовериться, что исходный код конкретного модуля работает корректно. Если вам нужно провести *системный тест* (system test), то есть тест всего приложения в целом с взаимодействием с GUI, например с симуляцией нажатия кнопок, проведением операций перетаскивания (drag&drop) и т. д., то обращаю ваше внимание на продукт Squish, который вы можете найти по адресу: **www.froglogic.com**.

Создание тестов

Тесты полезно создавать до начала реализации кода. Это позволит вам при написании теста лучше осмыслить и понять задачу, задав себе вопрос: а что нужно сделать для добавления реализации? Скомпилированный тест представляет собой готовую к исполнению программу.

Для демонстрации рассмотрим простой пример: предположим, нам нужно реализовать класс с методами для нахождения максимума и минимума двух чисел. Первая задача заключается в подготовке тестовых данных, которые будут выступать в качестве образцов для тестирования. Возьмем для этой цели четыре пары чисел: (25, 0), (−12, −15), (2007, 2007) и (−12, 5). Теперь, когда тестовые данные готовы, можно начинать писать тесты. Существуют соглашения для названия тестирующего класса и его методов, которые успели закрепиться и зарекомендовать себя на практике с наилучшей стороны. А именно:

- называйте тестирующий класс именем тестируемого класса с префиксом `Test_`. Например, если мы тестируем класс `MyClass`, то тестирующий класс будет называться `Test_MyClass`;
- называйте тестовые слоты (методы) именами тестируемых методов.

В листинге 45.1 показана программа, которая должна проводить тест методов `min()` и `max()` класса `MyClass`. В тестовой программе необходимо включить заголовочный файл QTest. Тестирующий класс должен быть унаследован от класса `QObject` и, для создания специальной метаинформации, содержать в своем определении макрос класса `QOBJECT`. Это позволит вызывать слоты класса при исполнении, включая его тестовые слоты в секции `private`.

Макрос `QCOMPARE()` принимает два аргумента: полученный и ожидаемый результат, и сравнивает их. Если значения не совпадают, то исполнение тестового метода прерывается с сообщением о том, что тест не прошел.

Нам нужна функция `main()`, в которой будет исполняться каждый тест. Поскольку для проведения тестов эта функция всегда выглядит одинаково, Qt предоставляет для ее замены макрос `QTEST_MAIN()`.

В завершение мы должны включить метаинформацию, сгенерированную компилятором MOC.

Листинг 45.1. Программа для проведения теста (файл test.cpp)

```
#include <QtTest>
#include "MyClass.h"

// ======================================================================
class Test_MyClass : public QObject {
Q_OBJECT
private slots:
    void min();
    void max();
};

// ----------------------------------------------------------------------
void Test_MyClass::min()
{
    MyClass myClass;
    QCOMPARE(myClass.min(25, 0), 0);
    QCOMPARE(myClass.min(-12, -5), -12);
    QCOMPARE(myClass.min(2007, 2007), 2007);
    QCOMPARE(myClass.min(-12, 5), -12);
}

// ----------------------------------------------------------------------
void Test_MyClass::max()
{
    MyClass myClass;
    QCOMPARE(myClass.max(25, 0), 25);
    QCOMPARE(myClass.max(-12, -5), -5);
    QCOMPARE(myClass.max(2007, 2007), 2007);
    QCOMPARE(myClass.max(-12, 5), 5);
}

QTEST_MAIN(Test_MyClass)
#include "test.moc"
```

После создания теста можно приступить к реализации методов тестируемого класса.

Класс `MyClass` реализует два метода для нахождения минимума и максимума (листинг 45.2).

Листинг 45.2. Методы для нахождения минимума и максимума (файл MyClass.cpp)

```
#pragma once

// ======================================================================
class MyClass {
```

```
public:
    int min(int n1, int n2)
    {
        return n1 < n2 ? n1 : n2;
    }

    int max(int n1, int n2)
    {
        return n1 > n2 ? n1 : n2;
    }
};
```

В pro-файле (листинг 45.3) в секции `QT` должна быть добавлена опция `testlib`. Имя заголовочного файла тестируемого класса указано для того, чтобы при любых его изменениях можно было скомпилировать тест заново.

Листинг 45.3. Проектный файл TestLib.pro

```
SOURCES = test.cpp
HEADERS = MyClass.h
QT  += testlib
TARGET = ../TestLib
```

При первом проведении теста полезно начать с проверки на отказ, то есть модифицировать проверяемый метод так, чтобы тест завершался неудачей. Это поможет убедиться в том, что тест действительно выполняется и проверяет то, что требуется. Для этого в методах `min()` и `max()` класса `MyClass` поменяйте знаки сравнения на противоположные. Теперь откомпилируем и запустим тест. На экране появится следующее:

```
********* Start testing of Test_MyClass *********
Config: Using QtTest library 5.10.0, Qt 5.10.0 (x86_64-little_endian-lp64 shared
(dynamic) release build; by Clang 7.0.2 (clang-700.1.81) (Apple))
PASS   : Test_MyClass::initTestCase()
FAIL!  : Test_MyClass::min() Compared values are not the same
   Actual   (myClass.min(25, 0)): 25
   Expected (0)                 : 0
   Loc: [test.cpp(36)]
FAIL!  : Test_MyClass::max() Compared values are not the same
   Actual   (myClass.max(25, 0)): 0
   Expected (25)                : 25
   Loc: [test.cpp(46)]
PASS   : Test_MyClass::cleanupTestCase()
Totals: 2 passed, 2 failed, 0 skipped, 0 blacklisted, 2ms
********* Finished testing of Test_MyClass *********
```

Методы `initTestCase()` и `cleanupTest()` вызываются в начале и конце теста соответственно. Эти методы не трактуются как тест-методы. Они выполняются при запуске тестов и служат для инициализации и очистки теста. Кроме того, на экране мы видим информацию о том, что тест прошел неудачно: сообщение `FAIL!`, имена тестов `Test_MyClass::min()` и `Test_MyClass::max()`, актуальные значения (`Actual`) и ожидаемые (`Expected`). Отлично! Наш тест завершился неудачей, а это значит, что проверка работы теста удалась — он действи-

тельно способен отслеживать ошибки. Теперь поменяем операторы сравнения в классе `MyClass` так, как это показано в листинге 45.2. Скомпилируем и запустим тест еще раз. Мы увидим на экране следующие сообщения:

```
********* Start testing of Test_MyClass *********
Config: Using QtTest library 5.10.0, Qt 5.10.0 (x86_64-little_endian-lp64 shared
(dynamic) release build; by Clang 7.0.2 (clang-700.1.81) (Apple))
PASS   : Test_MyClass::initTestCase()
PASS   : Test_MyClass::min()
PASS   : Test_MyClass::max()
PASS   : Test_MyClass::cleanupTestCase()
Totals: 4 passed, 0 failed, 0 skipped, 0 blacklisted, 1ms
********* Finished testing of Test_MyClass *********
```

Это говорит о том, что все наши тесты прошли удачно.

Тесты с передачей данных

До настоящего момента мы вписывали данные для проведения теста в макрос `QCOMPARE()`. Подобный подход вызывает нежелательный эффект дублирования кода. Для минимизации дублирования кода модуль `QtTest` предоставляет возможность проведения тестов с передачей данных. Такой подход позволяет отделить тестовый код от данных, поместив их в отдельное место. Этим местом является слот, который предоставляется для каждого тестирующего слота. Он должен называться так же, как и тестирующий, но с постфиксом `_data`.

В тестирующий класс, показанный в листинге 45.4, мы ввели два дополнительных слота: `min_data()` и `max_data()`, которые будут обеспечивать данными наши тестирующие слоты.

Листинг 45.4. Определение класса `Test_MyClass` (файл test.cpp)

```
class Test_MyClass : public QObject {
Q_OBJECT
private slots:
    void min_data();
    void max_data();

    void min();
    void max();
};
```

В листинге 45.5 показан слот `min_data()`, в котором необходимо создать таблицу тестовых данных. Для задания ее столбцов используется метод `QTest::addColumn()`. Определив столбцы, мы добавляем данные в таблицу с помощью метода `QTest::newRow()`. Строка, переданная в этот метод, является идентификатором строки таблицы. Каждый вызов создает свою собственную строку, а при помощи оператора << в нее добавятся данные: два параметра и ожидаемый результат применения метода `min()`.

Листинг 45.5. Метод `min_data()` (файл test.cpp)

```
void Test_MyClass::min_data()
{
    QTest::addColumn<int>("arg1");
```

```
    QTest::addColumn<int>("arg2");
    QTest::addColumn<int>("result");

    QTest::newRow("min_test1") << 25 << 0 << 0;
    QTest::newRow("min_test2") << -12 << -5 << -12;
    QTest::newRow("min_test3") << 2007 << 2007 << 2007;
    QTest::newRow("min_test4") << -12 << 5 << -12;
}
```

Реализация слота данных `max_data()` (листинг 45.6) аналогична слоту `min_data()` (см. листинг 45.5). Единственное отличие — в идентификаторах строк таблицы и в ожидаемых результатах.

Листинг 45.6. Метод `max_data()` (файл test.cpp)

```
void Test_MyClass::max_data()
{
    QTest::addColumn<int>("arg1");
    QTest::addColumn<int>("arg2");
    QTest::addColumn<int>("result");

    QTest::newRow("max_test1") << 25 << 0 << 25;
    QTest::newRow("max_test2") << -12 << -5 << -5;
    QTest::newRow("max_test3") << 2007 << 2007 << 2007;
    QTest::newRow("max_test4") << -12 << 5 << 5;
}
```

В методе данных создается таблица из четырех строк, поэтому слот `min()`, приведенный в листинге 45.7, будет запускаться четыре раза — по разу для каждой строки таблицы данных. Мы используем три макроса `QFETCH()` для создания локальных переменных `arg1`, `arg2`, `result` и внесения в них данных. Заметьте, имя должно совпадать с именем элемента тестовых данных, а если элемент данных с этим именем не будет найден, то тест завершится с сообщением об ошибке. Теперь для проведения тестов нам нужен только лишь один макрос `QCOMPARE()`. Такой подход позволяет легко добавлять новые данные для теста без модификации самого теста.

Листинг 45.7. Метод `min()` (файл test.cpp)

```
void Test_MyClass::min()
{
    MyClass myClass;
    QFETCH(int, arg1);
    QFETCH(int, arg2);
    QFETCH(int, result);

    QCOMPARE(myClass.min(arg1, arg2), result);
}
```

Реализация тестового слота `max()` приведена в листинге 45.8 и аналогична реализации слота `min()`, с той разницей, что для проведения теста вызывается метод `MyClass::max()` вместо метода `MyClass::min()`.

Листинг 45.8. Метод max() (файл test.cpp)

```
void Test_MyClass::max()
{
    MyClass myClass;
    QFETCH(int, arg1);
    QFETCH(int, arg2);
    QFETCH(int, result);

    QCOMPARE(myClass.max(arg1, arg2), result);
}
```

В завершение мы должны предоставить функцию `main()` при помощи макроса `QTEST_MAIN()` и включить метаданные, сгенерированные компилятором MOC:

```
QTEST_MAIN(Test_MyClass)
#include "test.moc"
```

Запустим наш тест и на экране увидим:

```
********* Start testing of Test_MyClass *********
Config: Using QtTest library 5.10.0, Qt 5.10.0 (x86_64-little_endian-lp64 shared
(dynamic) release build; by Clang 7.0.2 (clang-700.1.81) (Apple))
PASS   : Test_MyClass::initTestCase()
PASS   : Test_MyClass::min(min_test1)
PASS   : Test_MyClass::min(min_test2)
PASS   : Test_MyClass::min(min_test3)
PASS   : Test_MyClass::min(min_test4)
PASS   : Test_MyClass::max(max_test1)
PASS   : Test_MyClass::max(max_test2)
PASS   : Test_MyClass::max(max_test3)
PASS   : Test_MyClass::max(max_test4)
PASS   : Test_MyClass::cleanupTestCase()
Totals: 10 passed, 0 failed, 0 skipped, 0 blacklisted, 2ms
********* Finished testing of Test_MyClass *********
```

Создание тестов графического интерфейса

Модуль `QtTest` предоставляет механизм для тестирования и графического интерфейса. Предположим, что мы хотим протестировать поведение виджета однострочного текстового поля `QLineEdit`. Прежде всего нам потребуется создать класс, содержащий тестовый метод.

В реализации тестового метода `edit()` мы создаем виджет `QLineEdit` (листинг 45.9). Затем имитируем ввод символов `ABCDEFGH`, используя метод `QTest::keyClicks()`, который симулирует серию нажатий на клавиши клавиатуры. В необязательных параметрах этого метода можно передавать модификаторы клавиатуры, а также задержку в миллисекундах после каждого нажатия на клавишу. Есть также методы для симулирования нажатия и отпускания отдельных клавиш: `keyClick()`, `keyPress()` и `keyRelease()` — в эти методы нужно передавать первым параметром указатель на виджет, вторым — символ или значение типа `Qt::Key`. Третий и четвертый параметры не обязательны и принимают модификатор (Shift, Ctrl и Alt) и время задержки.

Мы используем макрос QCOMPARE() для того, чтобы проверить на совпадение текст однострочного текстового поля с ожидаемым текстом. После того как текст в виджете поля был изменен, вызов метода isModified() должен вернуть значение true. Мы проверяем это при помощи макроса QVERIFY(), который оценивает переданное выражение, и, если оно истинно, исполнение теста продолжается. В противном случае осуществляется отображение сообщения об ошибке и выполнение теста прекращается.

Листинг 45.9. Тест на совпадение текста однострочного текстового поля с ожидаемым текстом (файл test.cpp)

```
#include <QtTest>
#include <QtWidgets>

// ======================================================================
class Test_QLineEdit : public QObject {
Q_OBJECT
private slots:
    void edit();
};

// ----------------------------------------------------------------------
void Test_QLineEdit::edit()
{
    QLineEdit txt;
    QTest::keyClicks(&txt, "ABCDEFGH");

    QCOMPARE(txt.text(), QString("ABCDEFGH"));
    QVERIFY(txt.isModified());
}

QTEST_MAIN(Test_QLineEdit)
#include "test.moc"
```

Запустим наш тест и на экране увидим:

```
********* Start testing of Test_QLineEdit *********
Config: Using QtTest library 5.10.0, Qt 5.10.0 (x86_64-little_endian-lp64 shared
(dynamic) release build; by Clang 7.0.2 (clang-700.1.81) (Apple))
PASS   : Test_QLineEdit::initTestCase()
PASS   : Test_QLineEdit::edit()
PASS   : Test_QLineEdit::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped, 0 blacklisted, 111ms
********* Finished testing of Test_QLineEdit *********
```

Класс QTest предоставляет методы, позволяющие симулировать события не только клавиатуры, но и мыши. Это следующие методы:

- mouseClick() — симулирует щелчок мыши;
- mouseDClick() — симулирует двойной щелчок;
- mousePress() — симулирует нажатие на кнопку мыши;
- mouseRelease() — симулирует отпускание кнопки мыши.

В первом параметре этих методов нужно передать указатель на виджет, во втором — кнопку мыши. Четвертый, пятый и шестой параметры не обязательны и принимают модификаторы клавиатуры, позицию указателя мыши и время задержки. Есть также метод `mouseMove()`, который выполняет перемещение указателя мыши. Использование метода `mouseClick()` может выглядеть следующим образом:

```
QPushButton cmd;
QTest::mouseClick(&cmd, Qt::LeftButton);
```

События могут быть записаны в объекте `QTestEventList` — так можно осуществить целую серию действий одним вызовом метода `simulate()`. Например, выполним три действия: запишем серию букв, подождем одну секунду и сотрем последнюю букву:

```
QTestEventList lst;
lst.addKeyClicks("ABCDE");
lst.addDelay(1000);
lst.addKeyClick(Qt::Key_Backspace);
QLineEdit txt;
lst.simulate(&txt);
```

Параметры для запуска тестов

Модуль QtTest предоставляет возможность использования параметров при запуске тестов. Это позволяет оказывать на каждый тест индивидуальное влияние. Например, при запуске теста без параметров будут выполнены все тестовые методы, но если бы мы, например, хотели убедиться в правильности работы только одного метода `MyTest::max()` (см. листинг 45.2), то нам нужно было бы запустить тест следующим образом:

```
TestLib max
```

В табл. 45.1 приведены некоторые опции, которые могут вам пригодиться при запуске тестов.

Таблица 45.1. *Некоторые опции для запуска тестов*

Опция	Объяснение
`-o filename`	Записывает результаты теста в файл `file`. После имени файла можно после запятой указать формат для вывода информации: `csv`, `xml`, `lightxml`. По умолчанию в качестве формата используется обычный текстовый формат `txt`
`-silent`	Ограничивает сообщения показом только предупреждений и ошибок
`-v1`	Отображает информацию о входе и выходе тестовых методов
`-v2`	Дополняет опцию `-v1` тем, что выводит сообщения для макросов `QCOMPARE()` и `QVERIFY()`
`-vs`	Отображает каждый высланный сигнал и вызванный слот
`-xml`	Осуществляет вывод всей информации в формате XML
`-eventdelay ms`	Заставляет тест остановиться и подождать определенное время (в миллисекундах). Эта опция полезна для нахождения ошибок в элементах графического интерфейса
`-help`	Выводит информацию о всех доступных опциях запуска теста

Резюме

Тесты — это мощный детектор ошибок, резко сокращающий время их поиска. Создав надежные тесты, можно значительно увеличить скорость программирования. Запускайте тесты как можно чаще.

Для создания теста нужен класс, который будет содержать тестовые слоты. Этот класс должен быть унаследован от класса `QObject`. Тестовая программа должна содержать макрос `QTEST_MAIN()`, который заменяет функцию `main()` для запуска всех тестовых методов.

Макрос `QCOMPARE()` сравнивает результирующие значения с ожидаемыми. Если значения идентичны, то исполнение теста продолжается, если нет — тест будет остановлен с отображением сообщения об ошибке.

Макрос `QVERIFY()` проверяет правильность условия. Если значение равно `true`, то выполнение теста продолжается. Если нет, то тест далее не исполняется, и отображается сообщение об ошибке.

Во избежание проблем, связанных с повторением кода, модуль `QtTest` предоставляет возможность создания тестов с передачей данных. Все, что нужно, — это просто добавить еще один слот в секцию `private` нашего класса. Слот для тестовых данных должен называться так же, как и тестовый слот, но с постфиксом `_data`. Тестовые данные имеют формат обычной таблицы. Назначение макроса `QFETCH()` состоит в создании локальных переменных и заполнении их данными.

Модуль `QtTest` предоставляет также возможность тестирования графического интерфейса.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/45-510/** или с помощью следующего QR-кода (рис. 45.1):



Рис. 45.1. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 46

Qt WebEngine

> Интернет похож на золотой пустяк, который действительно является золотом... Это означает, что он перевернет наши представления о бизнесе, образовании и даже развлечениях.
>
> *Билл Гейтс*

В наши дни Интернет приобрел невиданную популярность. Некоторые учебные заведения уже используют его для предоставления студентам доступа к учебным материалам и методическим пособиям. А приобретение товаров, заключение сделок, оплата счетов, торговля акциями через Интернет сегодня стали обычным делом. Поэтому многие приложения нуждаются в показе веб-содержимого, и почти каждый разработчик хочет создавать приложения, использующие Всемирную паутину.

В настоящее время из основной массы приложений, нацеленных на Интернет, выделяются веб-браузеры. Но вполне естественно, что вам может понадобиться отображать содержимое HTML-документов без необходимости запуска веб-браузера при каждом щелчке мыши пользователя. Поэтому ситуация все больше изменяется в сторону встраивания веб-клиентов в собственную программу.

Пользуясь, например, преимуществами программирования приложений и включив в него возможность использования встроенного веб-клиента, вы можете создать гибрид, объединяющий сильные стороны обоих миров, слаженно взаимодействующих и взаимодополняющих друг друга.

В создании гибридов открывается большой простор для фантазии. Так, например, вы можете встроить веб-клиент в свое приложение для показа погоды, курсов валют, акций, предоставить своему приложению ресурсы, находящиеся за пределами Сети, например в локальной системе, создать собственные веб-сервисы, используя информацию из Google, Yahoo и многое другое. Один из самых, пожалуй, ярких примеров такого гибрида — это приложение iTunes (рис. 46.1) фирмы Apple. Счастливые обладатели iPad, iPod и iPhone отлично знают эту программу, она есть как для Mac OS X, так и для Windows. Скачать ее можно по адресу **www.apple.com/itunes/download/**.

Приложение iTunes содержит встроенный веб-клиент, который позволяет пользователям открыть и загрузить из магазина iTunes, например, музыку. После загрузки композиции показываются в приложении в виде списка. И в этой же программе их можно проиграть с помощью медиаплеера. Скачивать можно не только музыку, но и видео, игры, программы — одним словом, все, что находится в электронном виде, но, естественно, не бесплатно. Отметим тот факт, что встроенный в приложении iTunes веб-клиент базируется на WebKit, кото-

рый использовался в предыдущих версиях Qt. Начиная с Qt версии 5.4, было решено заменить его другим и базироваться на проекте Chromium. Проект Chromium представляет собой веб-браузер с открытым исходным кодом, над которым работают такие известные компании, как Google, Яндекс, Opera и другие. Новый модуль для работы с Chromium в Qt было решено назвать Qt WebEngine. Ему и посвящена эта глава.

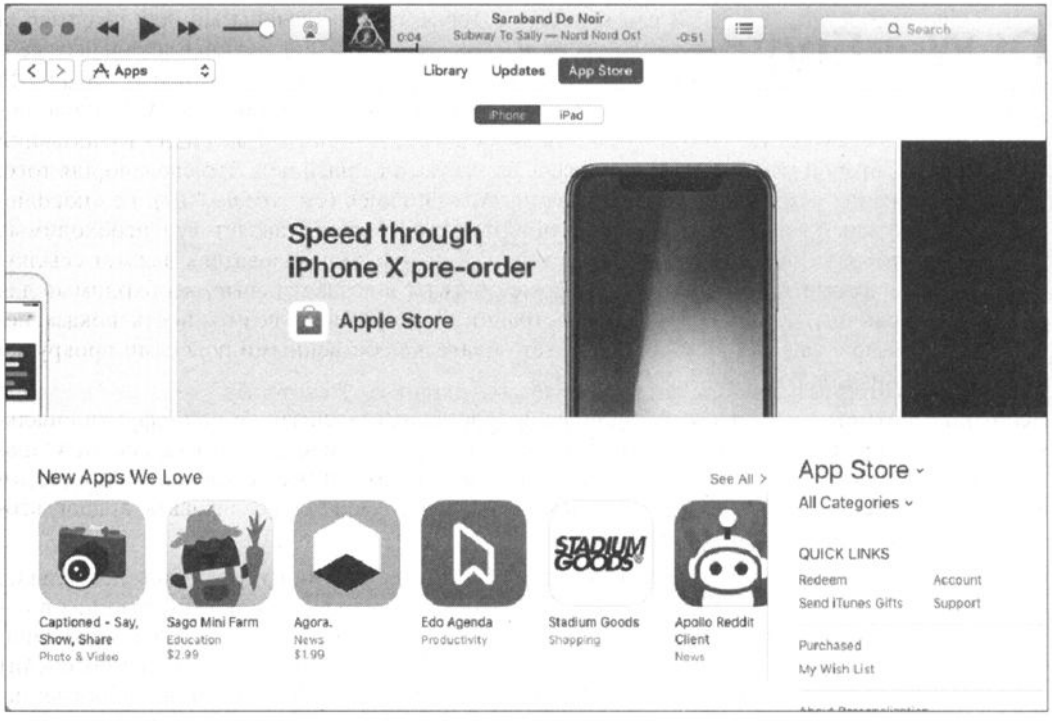


Рис. 46.1. Главное окно приложения iTunes

А зачем?

При разработке браузеров и других приложений, отображающих веб-страницы, до настоящего времени программисты сталкивались с целым рядом нелегких задач, связанных, например, с загрузкой и выводом содержимого веб-страниц. Такие программы должны базироваться на архитектуре «клиент-сервер», в которой клиент делает асинхронные запросы к веб-серверу для получения содержимого веб-страницы. Во время этого процесса может возникнуть масса проблем, связанных с отправкой запросов и получением ответов со стороны сервера через сеть. Например, могут возникнуть перебои в работе сети, страницы могут содержать неправильные гиперссылки, вовсе иметь дефектное содержание и т. д. Не легче и отображать полученное содержимое, которое может быть весьма сложным. Оно может включать несколько фреймов, серию MIME-типов, таких как, например, растровые изображения и фильмы.

Все это требует от разработчиков программного обеспечения либо колоссальных временных затрат, которые исчисляются месяцами, либо затрат денежных на приобретение библиотек от независимых производителей.

С появлением WebEngine в Qt все изменилось в лучшую сторону. Использование этой библиотеки сократило время разработки с нескольких месяцев до считанных минут и сняло дополнительные затраты при реализации коммерческого программного обеспечения.

Быстрый старт

WebEngine предоставляет целый ряд классов, которые предназначены как для реализации простых встраиваемых в приложение веб-клиентов, так и для полнофункциональных веб-браузеров. Вместе с тем он скрывает от разработчиков детали всех сложных задач. По умолчанию классы WebEngine прозрачно обращаются с запросами клиента. WebEngine поделен в Qt 5 на две части: `QtWebEngineCore` и `QtWebEngineWidgets`. Как видно из названий этих модулей, второй модуль содержит в себе виджеты, а первый нет. Это сделано для того, чтобы приложения без виджетов, реализованные на Qt Quick (см. *часть VIII*), не «носили» за собой ненужный балласт кода. Qt WebEngine также предоставляет все необходимые классы для отображения получаемых веб-станиц. Как только пользователь нажмет ссылку, Qt WebEngine автоматически разрушит старые объекты и создаст новые, необходимые для новой веб-страницы. Класс отображения страниц поддерживает возможность показа нескольких фреймов, каждый из которых может обладать собственными полосами прокрутки и содержать объекты различных MIME-типов.

Необходимо отметить, что Qt WebEngine не предоставляет реализацию полнофункционального браузера со всеми элементами графического интерфейса в виде одного виджета. Однако он предоставляет «конструктор», содержащий все необходимые компоненты, при помощи которых, используя также другие виджеты Qt, вы сможете реализовать аналог веб-браузера нужной вам конфигурации.

Использование Qt WebEngine настолько просто, что при помощи всего лишь нескольких строк кода можно реализовать программу, отображающую содержимое веб-страницы в соответствии с заданной ссылкой. Это приложение будет также обладать возможностью навигации по Сети посредством перехода по ссылкам, а наличие контекстного меню обеспечит функции возврата к предыдущей, перехода к последующей странице и их обновление. Рецепт создания простейшего веб-клиента прост — нам потребуется сделать лишь следующее:

1. Создать виджет `QWebEngineView`.
2. Отослать запрос загрузки.
3. Показать окно виджета.

Вот и все! Теперь для наглядности создадим программу (листинг 46.1), которая выполняет эти действия (рис. 46.2).

Класс `QWebEngineView` — это центральный класс в Qt WebEngineWidgets, который управляет отображением и всем взаимодействием с пользователем. Пользователи могут осуществлять навигацию самостоятельно, щелкая по ссылкам. Для этого нам совсем не понадобится усложнять текст программы — невидимый управляющий элемент возьмет все на себя. В листинге 46.1 мы создаем виджет веб-представления (`webView`) этого класса и вызываем его метод `load()`, передавая строку с начальной ссылкой. Обратите внимание, что строка должна быть преобразована в `QUrl`. Вызов метода `show()` отображает наш виджет на экране.

Листинг 46.1. Программа показа веб-содержимого

```
#include <QtWidgets>
#include <QtWebEngineWidgets>
```

```
int main(int argc, char** argv)
{
    QApplication   app(argc, argv);
    QWebEngineView webView;

    webView.load(QUrl("http://www.bhv.ru"));
    webView.show();

    return app.exec();
}
```



Рис. 46.2. Отображение содержимого веб-страницы

> ***Перед компилированием программы...***
>
> Для того чтобы программу можно было откомпилировать, не забудьте добавить в проектный pro-файл строку: `QT += widgets webenginewidgets`.

Хоть старт был и быстрым, но веб-браузером это приложение назвать нельзя. Нашей программе еще не хватает некоторых вещей, без которых пользователь «в Паутине» просто не сможет обойтись. Например, ему наверняка захочется иметь функции **Назад**, **Вперед** и **Обновление** в виде отдельных кнопок. Также обязательно нужно отдельное поле ввода, в котором он сам смог бы указывать необходимые ему интернет-адреса. Желательно, чтобы он видел процесс загрузки содержимого страницы и еще многое другое. Поэтому не будем останавливаться на достигнутом и отправимся дальше. Как говорят в Китае, дорогу осилит идущий.

Создание простого веб-браузера

Итак, мы установили, что для того чтобы наша программа могла гордо называться веб-браузером, в ней нужно реализовать как минимум три следующие возможности:

- ввод веб-адресов;
- управление историей;
- отображения процесса загрузки страницы и ее ресурсов.

Теперь давайте остановимся на каждой из этих возможностей подробнее.

Ввод адресов

В какой-то момент у пользователя может возникнуть необходимость задать новый адрес, чтобы перейти к странице, не связанной с текущей. Эта функция обязательна для веб-браузеров, но совсем не обязательна для гибридных приложений. Но если в вашем приложении предусмотрено, что адреса будут вводиться пользователем, необходимо позаботиться о том, чтобы эти адреса были введены в понятном для Qt WebEngine виде, — в частности, для метода загрузки `load()`. И позаботиться об этом нужно самому. Так, самой частой ошибкой или, точнее сказать, упущением является то, что пользователь забывает указывать тип сервиса в самом начале адреса. То есть пользователь напишет `www.bhv.ru` вместо того, чтобы написать `http://www.bhv.ru`. Такие ситуации необходимо отслеживать и при необходимости дополнять недостающий тип сервиса к введенному пользователем адресу.

Управление историей

История в Qt WebEngine формируется автоматически в процессе перехода пользователя в Сети со страницы на страницу. Если только что посещенная страница имеет адрес, совпадающий с другим элементом истории, то этот элемент истории изымается и заменяется новым, иначе бы подобные списки истории вырастали бы до огромных размеров и были бы непригодны. За список элементов истории в Qt WebEngine отвечает класс `QWebEngineHistory`. Доступ к объекту этого класса можно получить прямо из объекта класса `QWebEngineView` вызовом метода `history()`. Сами элементы истории хранятся в объектах класса `QWebHistoryItem`. Причем в этих элементах запоминаются также и даты посещения. Поэтому можно сгруппировать и показать пользователю все страницы, посещенные им в конкретные дни и часы. Например, можно создать подменю, в котором будут отображены страницы, посещенные пользователем за последние три дня.

Элементы истории можно удалить все сразу вызовом метода `clear()` объекта класса `QWebEngineHistory`.

Очевидно, что пользователю в веб-браузере необходимо предоставить возможность перемещаться по страницам вперед и назад согласно списку истории. Например, если он находится на некоторой веб-странице и хочет вернуться на страницу, на которой был до этого, вы должны предоставить ему эту возможность. В веб-браузерах обычно предусмотрены кнопки со стрелками вправо и влево, которые позволяют пользователям перемещаться вперед и назад. Для реализации этой возможности класс `QWebEngineHistory` предоставляет методы `back()` и `forward()`, но ими пользоваться необязательно, поскольку класс `QWebEngineView` располагает одноименными делегирующими слотами, с которыми очень удобно соединить сигналы таких кнопок.

Загрузка страниц и ресурсов

Ресурсы — это любые данные, связанные со страницей, которые должны быть загружены отдельно (дополнительно к самой странице), — например, растровые изображения, программы сценариев, таблицы CSS, а также и веб-страницы, содержащиеся во фреймах. Сама же веб-страница может содержать несколько ресурсов сразу, каждый из которых получается отсылкой собственного запроса и обработкой ответного сообщения. Запрошенные ресурсы поступают независимо и в любом порядке. Из всего этого становится понятно, что загрузка страниц — на самом деле весьма сложный процесс, который Qt WebEngine умышленно скрывает от разработчиков. Но ошибки происходят — ведь может получиться так, что запросы будут неуспешны. В подобных ситуациях приложение должно быть в состоянии соответствующим образом проинформировать пользователя. Для этих целей класс `QWebEngineView` предоставляет сигнал `loadFinished()`, который передает значение булевого типа. Это значение позволяет сделать вывод о том, успешно ли произошла загрузка страницы. Самая, пожалуй, частая причина ошибки — это неправильное указание пользователем адреса.

Если вы хотите, чтобы приложение уведомляло пользователя о процессе загрузки страницы и ее ресурсов, необходимо соединить ваш слот с сигналом `loadProgress()` класса `QWebView`. Этот сигнал передает целочисленное значение от 0 до 100, показывающее процесс загрузки.

Пишем веб-браузер: попытка номер два

Теперь подытожим все ранее изложенное конкретным примером и напишем наконец скромный, но настоящий веб-браузер (листинги 46.2–46.6), окно которого показано на рис. 46.3.



Рис. 46.3. Простой веб-браузер

В основной программе (листинг 46.2) мы просто создаем виджет нашего веб-браузера (webBrowser).

Листинг 46.2. Основная программа веб-браузера (файл main.cpp)

```
#include <QtWidgets>
#include "WebBrowser.h"

int main(int argc, char** argv)
{
    QApplication  app(argc, argv);
    WebBrowser    webBrowser;

    webBrowser.show();

    return app.exec();
}
```

В файле определений (листинг 46.3) мы задаем указатели на некоторые элементы управления нашего браузера: на текстовое поле для ввода адресов (указатель m_ptxt), на сам элемент веб-представления (указатель m_pwv) и на кнопки навигации по истории назад и вперед (указатели m_pcmdForward и m_pcmdBack). В нем также определены два слота: slotGo() и slotFinished(). Первый нам нужен, чтобы запускать процесс загрузки страницы, а второй будет отвечать за действия, которые надо выполнить после загрузки.

Листинг 46.3. Определение классов (файл WebBrowser.h)

```
#pragma once

#include <QWidget>

class QLineEdit;
class QWebEngineView;
class QPushButton;

// ============================================================
class WebBrowser : public QWidget{
Q_OBJECT
private:
    QLineEdit*      m_ptxt;
    QWebEngineView* m_pwv;
    QPushButton*    m_pcmdBack;
    QPushButton*    m_pcmdForward;

public:
    WebBrowser(QWidget* wgt = 0);

private slots:
    void slotGo      (    );
    void slotFinished(bool);
};
```

В конструкторе (листинг 46.4) мы создаем все необходимые для нашего веб-браузера элементы управления. Виджет для ввода адресов (указатель `m_ptxt`) сразу инициализируется текстом стартовой страницы. Далее создаются виджет веб-представления (указатель `m_pwv`) и кнопки вперед (указатель `m_pcmdForward`) и назад (указатель `m_pcmdBack`). При старте нашего веб-браузера эти кнопки должны быть недоступны, так как пользователь еще не успел посетить другие страницы. Это состояние мы устанавливаем вызовом метода `setEnabled()`, а сигналы нажатия кнопок соединяем со слотами веб-представления `back()` и `forward()`.

Для отображения процесса загрузки страницы мы создаем виджет индикации процесса (указатель `pprb`). Его слот `setValue()` соединяется с сигналом веб-представления `loadProgress()`.

Другими важными функциями, которые могут оказаться полезны пользователям, являются функции `Go`, `Stop` и `Refresh`. Функция `Go` (кнопка **Go**) осуществляет загрузку страницы. С нашим слотом `slotGo()` мы связываем сигнал нажатия этой кнопки, а также сигнал нажатия клавиши <Enter> в поле адресов. Кнопка **Stop** прерывает выполняемую в текущий момент загрузку содержимого веб-страницы, и ее сигнал нажатия связывается со слотом веб-представления `stop()`. Функция `Refresh` (кнопка **Refresh**) обновляет текущую веб-страницу, инициируя повторную загрузку ее данных с веб-сервера, для чего сигнал нажатия этой кнопки соединяется с одноименным слотом.

Далее мы размещаем виджеты в компоновке и вызываем слот `slotGo()`.

Листинг 46.4. Конструктор класса `WebBrowser` (файл WebBrowser.cpp)

```
WebBrowser::WebBrowser(QWidget* wgt/*=0*/) : QWidget(wgt)
{
    m_ptxt        = new QLineEdit("http://www.bhv.ru");
    m_pwv         = new QWebEngineView;
    m_pcmdBack    = new QPushButton("<");
    m_pcmdForward = new QPushButton(">");

    m_pcmdBack->setEnabled(false);
    m_pcmdForward->setEnabled(false);

    QProgressBar* pprb        = new QProgressBar;
    QPushButton*  pcmdGo      = new QPushButton("&Go");
    QPushButton*  pcmdStop    = new QPushButton("&Stop");
    QPushButton*  pcmdRefresh = new QPushButton("&Refresh");

    connect(pcmdGo, SIGNAL(clicked()), SLOT(slotGo()));
    connect(m_ptxt, SIGNAL(returnPressed()), SLOT(slotGo()));
    connect(m_pcmdBack, SIGNAL(clicked()), m_pwv, SLOT(back()));
    connect(m_pcmdForward, SIGNAL(clicked()), m_pwv, SLOT(forward()));
    connect(pcmdRefresh, SIGNAL(clicked()), m_pwv, SLOT(reload()));
    connect(pcmdStop, SIGNAL(clicked()), m_pwv, SLOT(stop()));
    connect(m_pwv, SIGNAL(loadProgress(int)), pprb, SLOT(setValue(int)));
    connect(m_pwv, SIGNAL(loadFinished(bool)), SLOT(slotFinished(bool)));

    //Layout setup
    QHBoxLayout* phbx = new QHBoxLayout;
    phbx->addWidget(m_pcmdBack);
```

```
    phbx->addWidget(m_pcmdForward);
    phbx->addWidget(pcmdStop);
    phbx->addWidget(pcmdRefresh);
    phbx->addWidget(m_ptxt);
    phbx->addWidget(pcmdGo);

    QVBoxLayout* playout = new QVBoxLayout;
    playout->addLayout(phbx);
    playout->addWidget(m_pwv);
    playout->addWidget(pprb);
    setLayout(playout);

    slotGo();
}
```

Слот `slotGo()` запускает загрузку страницы (листинг 46.5). Как мы уже упоминали, прежде чем передать строку в метод `load()`, необходимо подстраховаться. Для этого проверяем фрагмент, с которого начинается строка введенного адреса, чтобы узнать, не забыл ли пользователь указать в ней тип сервиса. В случае, если строка начинается с `ftp://`, `http://` или `gopher://`, мы оставляем ее без изменений, в противном случае исходим из того, что пользователь забыл указать сервис `http://`, и просто добавляем его в начало строки. Теперь все должно быть в порядке, и строку можно передать в метод `load()`.

Листинг 46.5. Слот `slotGo()` (файл WebBrowser.cpp)

```
void WebBrowser::slotGo()
{
    if (!m_ptxt->text().startsWith("ftp://")
        && !m_ptxt->text().startsWith("http://")
        && !m_ptxt->text().startsWith("gopher://")
       ) {
        m_ptxt->setText("http://" + m_ptxt->text());
    }
    m_pwv->load(QUrl(m_ptxt->text()));
}
```

В листинге 46.6 приведен слот `slotFinished()`, который вызывается по завершению загрузки страницы. Прежде всего нам необходимо убедиться в том, что загрузка прошла без ошибок. Для этого мы используем переданный в слот параметр. В случае возникновения ошибки мы сообщаем об этом пользователю, для чего вызываем метод `setHtml()` и устанавливаем в веб-представлении соответствующий текст.

Несмотря на то, что управляющий элемент делает для нас почти все, тем не менее есть моменты, о которых нужно позаботиться самим. Как только пользователь нажал на ссылку, адрес текущей страницы уже не соответствует предыдущему. Пользователь его наверняка захочет увидеть, скопировать или немного изменить. А это значит, что нам всякий раз после смены адреса необходимо актуализировать содержимое виджета текстового поля, предназначенного для ввода адресов (указатель `m_ptxt`), и помещать в него адрес актуальной страницы. Адрес актуальной страницы мы получаем в программе вызовом метода `url()` виджета веб-представления (указатель `m_pwv`).

Теперь нам осталось только правильно отобразить статус кнопок перехода назад и вперед. Для того чтобы узнать, возможно ли выполнить эти действия, в классе `QWebEngineHistory` определены два метода: `canGoBack()` и `canGoForward()`. В соответствии с их значениями нажатие на кнопки делаются доступными либо недоступными.

Листинг 46.6. Слот `slotFinished()` (файл WebBrowser.cpp)

```
void WebBrowser::slotFinished(bool bOk)
{
    if (!bOk) {
        m_pwv->setHtml("<CENTER>An error has occured"
                       " while loading the web page</CENTER>"
                      );
    }

    m_ptxt->setText(m_pwv->url().toString());

    m_pcmdBack->setEnabled(m_pwv->page()->history()->canGoBack());
    m_pcmdForward->setEnabled(m_pwv->page()->history()->canGoForward());
}
```

Резюме

Вы теперь можете в течение считанных минут создать свой веб-браузер. Модуль Qt WebEngine — это мощное средство в сфере программирования для Интернета, которое значительно упрощает сложный процесс загрузки веб-страниц, предоставляет набор классов для их отображения и реализует механизм перехода по нажатым гиперссылкам. В Qt центральным для Qt WebEngine классом является `QWebEngineView`, через который предоставляется основная часть возможностей этого модуля.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/46-510/** или с помощью следующего QR-кода (рис. 46.4):

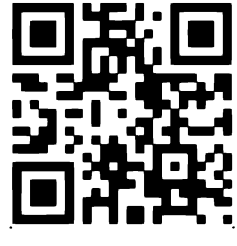


Рис. 46.4. QR-код для доступа на страницу комментариев к этой главе

ГЛАВА 47

Интегрированная среда разработки Qt Creator

> Мы должны сделать так, чтобы работа с компьютером стала столь же естественной, как с карандашом или ручкой.
>
> *Билл Гейтс*

Интегрированная среда разработки (IDE, Integrated Development Environment) — это набор инструментов, объединенных в одном приложении. Ее использование существенно облегчает работу разработчика. Такой инструмент есть и для Qt — он называется Qt Creator и входит в пакет поставки Qt. Цель создания интегрированной среды разработки Qt Creator — заполнить существовавшую нишу и предоставить платформонезависимую среду разработки, ориентированную на Qt-проекты как на языке C++, так и на языке QML (см. *часть VIII*). Теперь на любой поддерживаемой платформе вы можете использовать одну и ту же интегрированную среду разработки. IDE Qt Creator объединяет в себе девять основных компонентов (рис. 47.1).



Рис. 47.1. Состав компонентов интегрированной среды разработки Qt Creator

Как видно из рисунка, Qt Creator включает и систему помощи — интегрированное в среду разработки расширение Qt Help, которое предоставляет доступ к документации библиотеки. А сама библиотека Qt — это и есть тот краеугольный камень, ради которого и на котором разрабатывалась среда Qt Creator.

Среда Qt Creator не имеет своих собственных компилятора, компоновщика и отладчика, поэтому в ней задействуются доступные на платформе средства. В Windows — это MinGW

(Minimalist GNU for Windows), включенный в пакет Qt. Совместно с Qt Creator можно использовать и Visual C++ 2013 и 2015 — нужно только загрузить скомпонованную для этих компиляторов версию. В Mac OS X Qt Creator пользуется возможностями, входящими в поставку XCode.

Мастер проектов — это средство для автоматического создания минимальных стартовых проектов для ваших программ. Создаваемый проект состоит из ресурсов, исходного текста и заголовочного файла.

Текстовый редактор — это, собственно, средство, с помощью которого создается программный код. Редактор сам подсвечивает синтаксис, автоматически дополняет код и обладает рядом других достоинств, делающих программирование быстрее и удобнее.

С помощью встроенной в среду программы Qt Designer (см. *главу 44*) можно создавать или изменять формы, не покидая интегрированную среду разработки.

Первый запуск

После запуска интегрированной среды Qt Creator вы увидите начальную ее страницу (рис. 47.2), которая по умолчанию открывается на вкладке **Проекты**, где имеются две кнопки: **Новый проект** и **Открыть проект**, с помощью которых можно либо создать новый проект, либо открыть уже существующий. На этой странице можно так же восстановить один из последних используемых проектов. Вкладка **Примеры** начальной страницы среды отображает документацию по примерам Qt, а вкладка **Учебники** предоставляет доступ к учебным материалам и полезным видео по теме Qt.

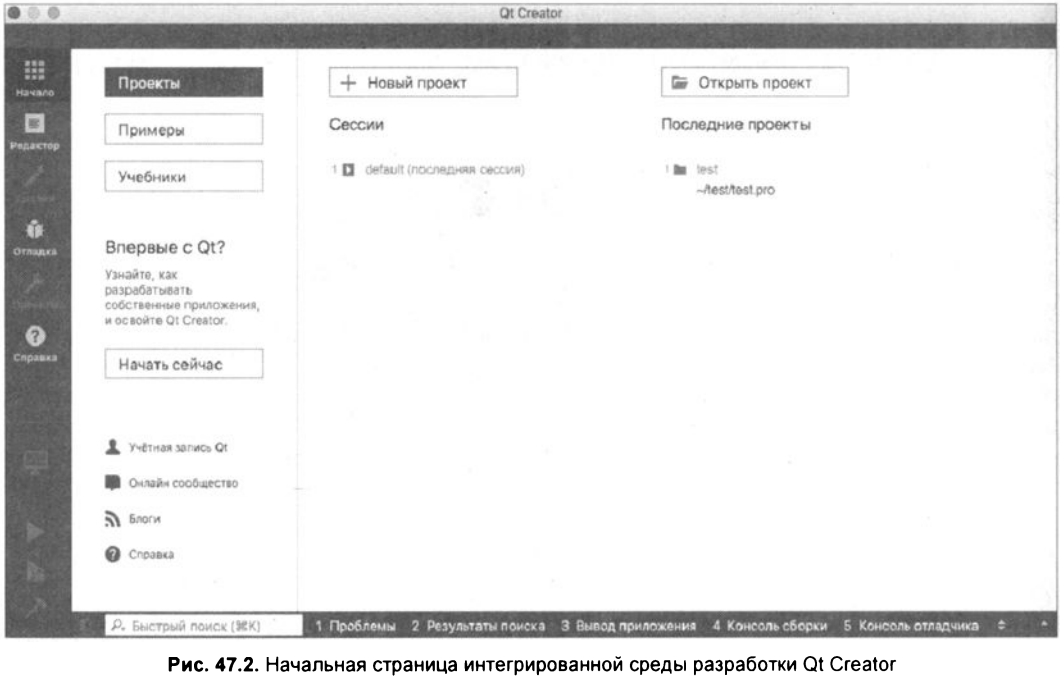


Рис. 47.2. Начальная страница интегрированной среды разработки Qt Creator

Создаем проект «Hello Qt Creator»

С «миром» мы уже поздоровались в *главе 1*. Давайте теперь создадим минимальный проект и поздороваемся с Qt Creator.

Проект — это собрание файлов реализации, заголовочных файлов, ресурсов и самого файла описания проекта (это уже знакомый нам pro-файл). Заметьте, для проектов не стали придумывать какого-либо нового формата файла специально под среду Qt Creator, и это очень хорошо тем, что вы можете использовать уже созданные вами pro-файлы и загружать их как проекты в Qt Creator.

Qt Creator предоставляет готовые шаблоны проектов, с помощью которых можно легко начать создание программы. Шаблоны — это «стержень-скелет», содержащий ресурсы, исходный текст и заголовочный файл.

Создать проект в Qt Creator очень просто. В меню **Файл** выберите команду **Новый проект** и в открывшемся диалоговом окне (рис. 47.3) выберите тип проекта **Приложение**, а затем одну из следующих альтернатив:

- **Приложение Qt Widgets** — приложение с графическим пользовательским интерфейсом, базирующемся на виджетах;
- **Консольное приложение Qt** — приложение без пользовательского интерфейса;
- **Приложение Qt Quick** — приложение с графическим пользовательским интерфейсом, которое может содержать код как QML, так и C++;
- **Приложение Qt Quick Controls 2** — приложение с графическим пользовательским интерфейсом, которое использует готовые элементы управления QML и может также содержать код как QML, так и C++;
- **Приложение Qt Canvas 3D** — приложение с использованием Qt Canvas 3D.

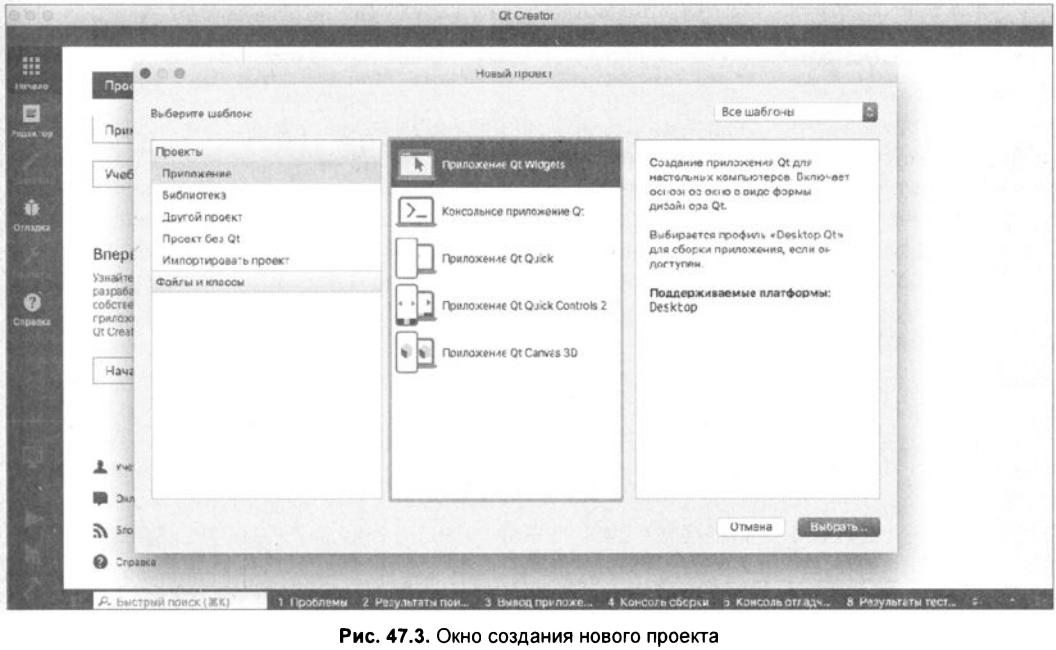


Рис. 47.3. Окно создания нового проекта

Давайте создадим приложение с графическим интерфейсом — для этого выделим **Приложение Qt Widgets** и нажмем кнопку **Выбрать**.

В следующем окне, показанном на рис. 47.4, задайте в текстовом поле ввода **Название** имя для своего проекта, например `MyApp`, и Qt Creator присвоит это имя новому проекту.

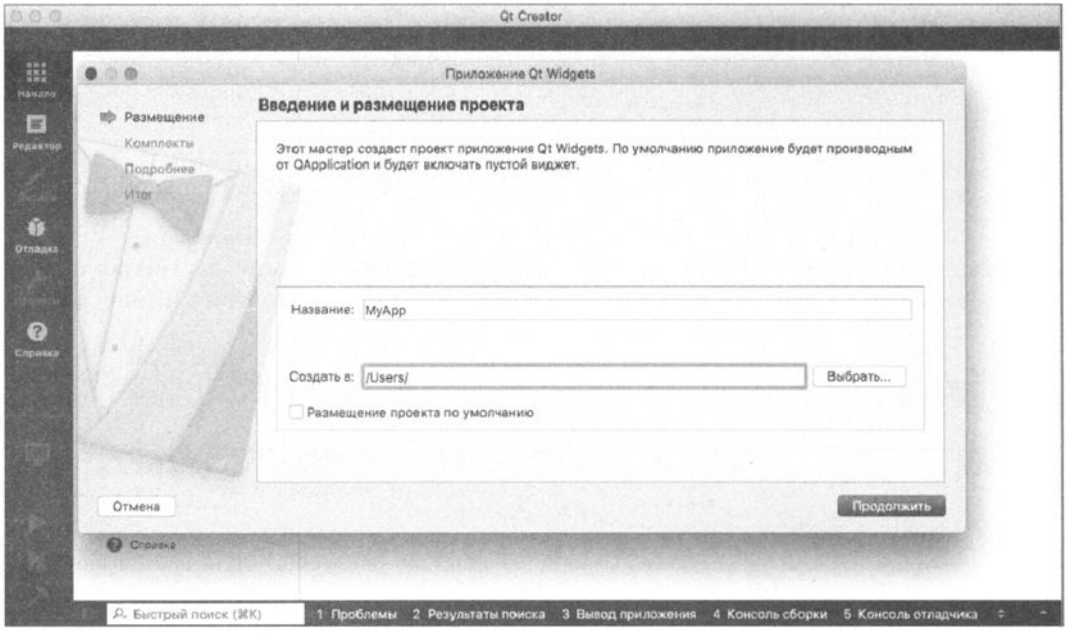


Рис. 47.4. Окно ввода имени проекта

Вы можете использовать для проекта путь, указанный в поле **Создать в** по умолчанию. Для его изменения укажите в этом поле путь к папке, в которой должен быть сохранен проект, или нажмите кнопку **Выбрать** и выберите папку. После этого нажмите кнопку **Продолжить**.

Если у вас установлены различные сборки (комплекты) Qt, например, для разнообразных устройств, то они будут показаны в соответствующем диалоговом окне (рис. 47.5). На данном этапе вы можете включить их в создаваемый проект или же исключить их участие в нем. В нашем примере доступны две сборки Qt. По умолчанию всегда включен режим использования теневой сборки — это очень удобно, когда вы работаете с различными версиями Qt, различными платформами и различными устройствами и хотите отделять скомпилированный код друг от друга в самостоятельные каталоги. После выполнения настроек нажмите кнопку **Продолжить**.

Теперь мастер проекта предложит вам выбрать для приложения базовый класс (рис. 47.6) и задать ему имя, а также имена для заголовочного файла (*.h), файла реализации (*.cpp) и файла формы (*.ui). Файл формы не обязателен, и если вы не собираетесь работать над формой во встроенной в Qt Creator программе Qt Designer, с которой мы уже успели познакомиться в *главе 44*, то вполне можно обойтись и без него, для чего достаточно снять флажок **Создать форму**. В нашем примере мы оставим эту опцию включенной, так как намереваемся воспользоваться программой Qt Designer.

На этом сбор информации для создания нового проекта подошел к концу, и в завершающем окне мастер проектов информирует нас о создаваемых файлах и об их местонахождении

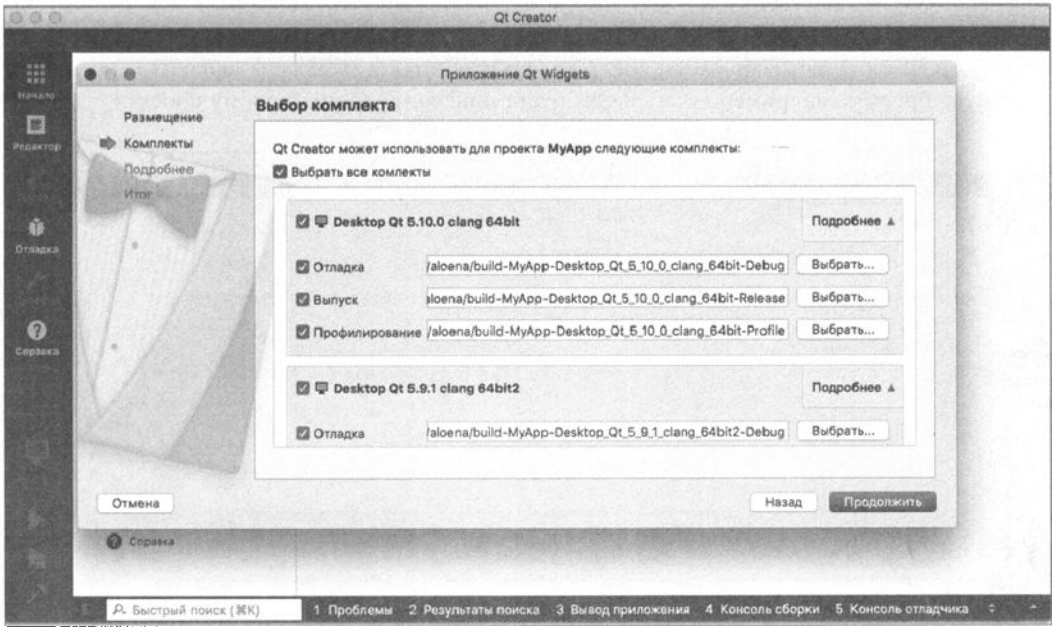


Рис. 47.5. Окно настройки цели проекта

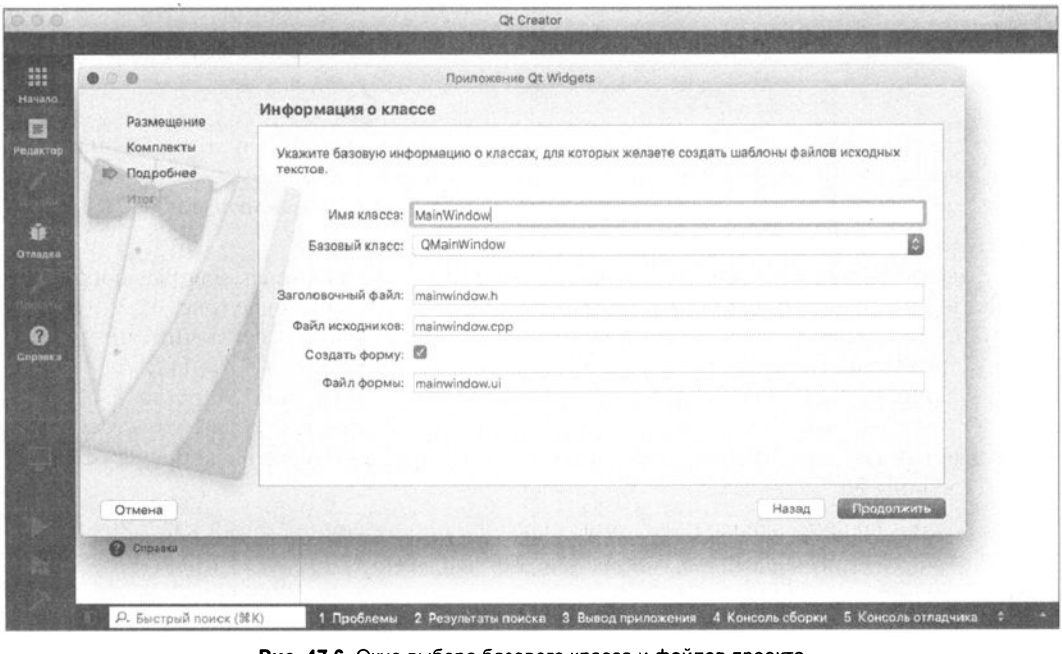


Рис. 47.6. Окно выбора базового класса и файлов проекта

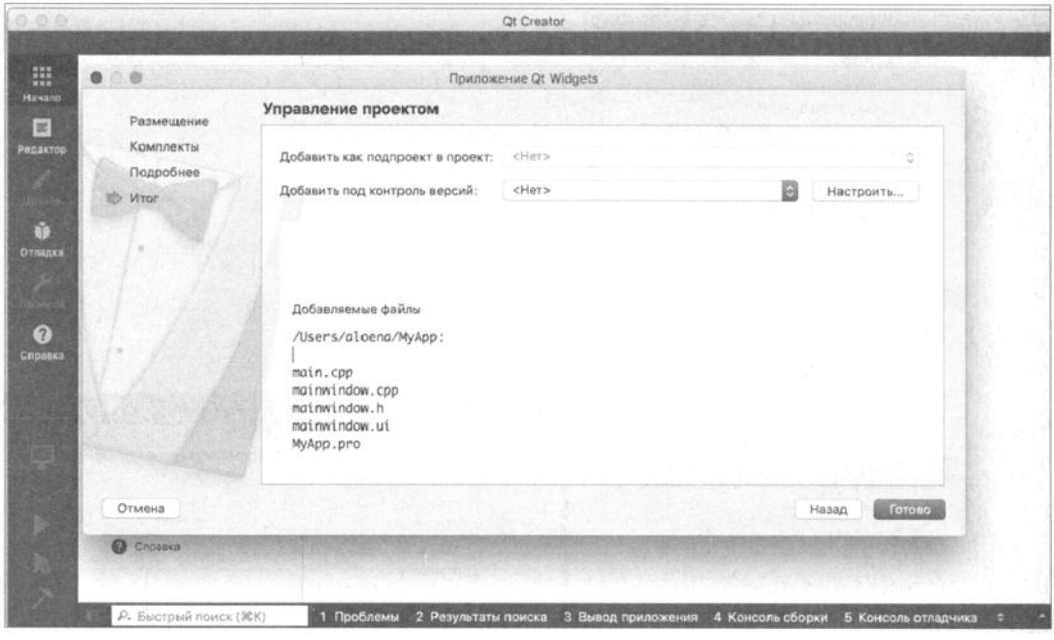


Рис. 47.7. Информационное окно проекта

(рис. 47.7). Проверив эту информацию, просто нажмите кнопку **Готово**, и интегрированная среда разработки Qt Creator создаст каталог MyApp и указанные файлы проекта. Сразу после этого мы увидим эти файлы в окне обозревателя проектов, показанном на рис. 47.8.

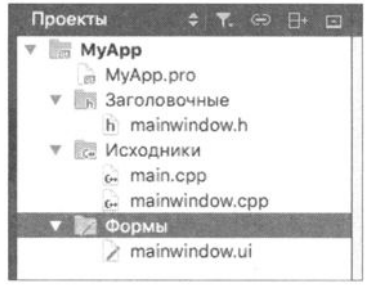


Рис. 47.8. Обозреватель проектов

Сам обозреватель проектов можно задействовать в различных аспектах представления информации и использовать неограниченное количество его окон. На рис. 47.9 показаны представления **Проекты**, **Обзор классов**, **Файловая система** и **Открытые документы**.

В окне обозревателя проектов (см. рис. 47.8) показаны исходные файлы: MyApp.pro, mainwindow.h, main.cpp, mainwindow.cpp и mainwindow.ui. Если их недостаточно, и нужно расширить проект дополнительными исходными файлами, то для этого выделите сам проект, как показано на рис. 47.10, и выберите из контекстного меню пункт **Добавить новый** или, если нужный файл или файлы уже находятся на диске, пункт **Добавить существующие файлы**. Если вы выбрали **Добавить новый**, то увидите уже знакомое вам диалоговое окно **Новый** (см. рис. 47.3), при помощи которого мы и создали наш проект. Здесь вам следует выбрать нужный тип файла. После создания файла будет задан вопрос о его добавлении в файл проекта. Если же вы выбрали пункт **Добавить существующие файлы**, то откроется стандартное диалоговое окно выбора файлов.

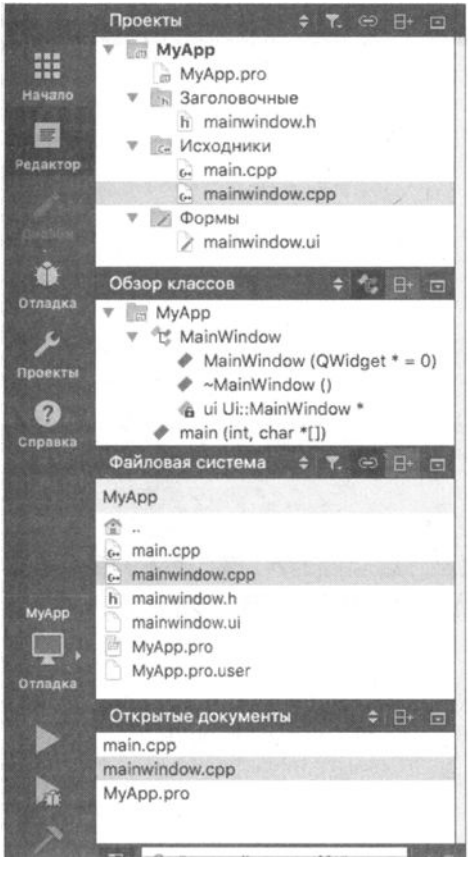


Рис. 47.9. Различные аспекты обозревателей проектов

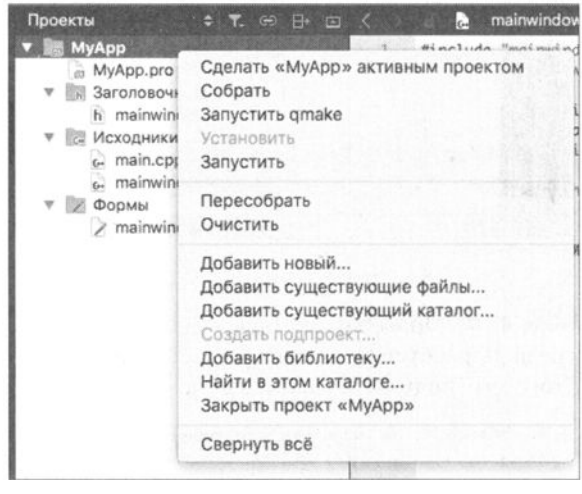


Рис. 47.10. Добавление в проект новых файлов

Пользовательский интерфейс Qt Creator

Теперь самое время рассмотреть компоненты графического интерфейса среды разработки Qt Creator.

Окно среды Qt Creator в режиме редактирования обладает следующими основными зонами (рис. 47.11):

- в самом низу находятся кнопки активации окон выводов с пояснениями справа от их номеров;
- на левом краю имеется полоса смены режимов (**Редактор**, **Отладка**, **Справка** и т. д.) и секция компилирования и запуска;
- правее полосы смены режимов расположено окно обозревателя проектов;
- самое большое окно — это окно редактора (основная рабочая область, в которой можно редактировать файлы).

Далее мы подробнее остановимся на некоторых из упомянутых зон.

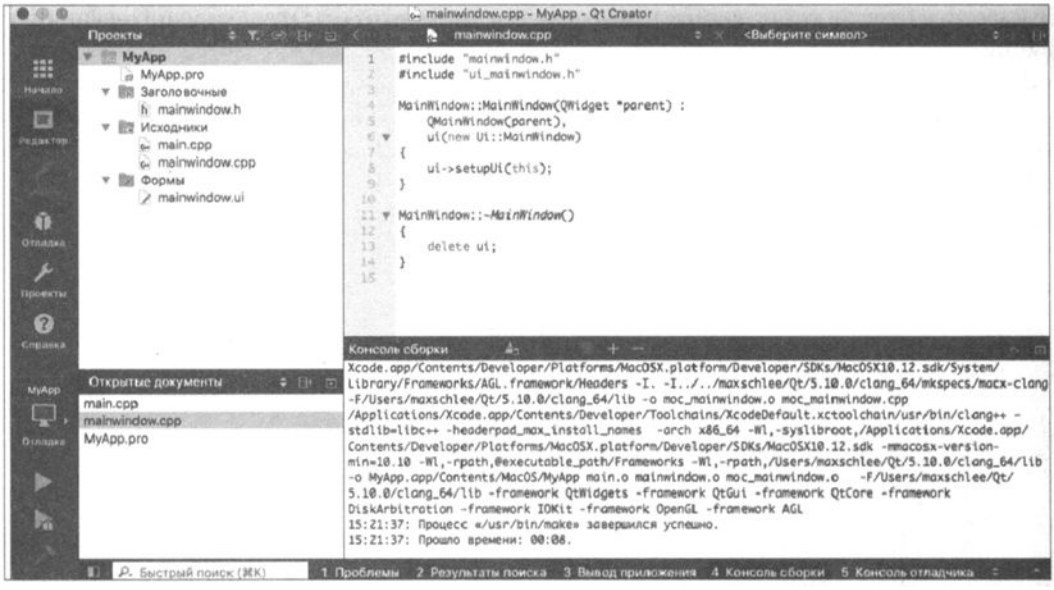


Рис. 47.11. Графический интерфейс интегрированной среды разработки Qt Creator

Окна вывода

Окна вывода (их всего семь) при запуске по умолчанию не видны и могут отображаться автоматически в зависимости от ваших действий. Например, как только вы начнете компилировать свое приложение, автоматически будет открыто окно **Консоль сборки**, в котором выводятся сообщения о ходе компиляции и компоновки программы. В частности, в нем отображаются сообщения о возникающих ошибках и предупреждения.

Если вы хотите увидеть то или иное окно вывода, то можете просто нажать на кнопку с именем нужного вам окна в нижней области окна Qt Creator. Если окно уже открыто, то нажатие на такую кнопку закроет его.

Окно проектного обозревателя

С этим окном мы уже успели познакомиться, оно показано на рис. 47.8. Примечательно, что оно может содержать более одного проекта, и это состояние можно сохранять в рабочей сессии. Для работы с созданными сессиями нужно открыть окно менеджера сессий из главного меню **Файл**.

Секция компилирования и запуска

Для демонстрации секции компилирования и запуска вернемся снова к нашему проекту, созданному для нас мастером проектов. Нам осталось только откомпилировать и запустить его. Обе операции можно выполнить сразу, нажатием всего лишь одной кнопки **Запуск**. Эта кнопка в виде зеленого треугольника находится слева — третья снизу (рис. 47.12).

Во время компиляции и сборки отображается ход процесса операции, а также при помощи индикаторов сообщается обо всех ошибках и предупреждениях компилятора или компоновщика. Для установки режимов компоновки можно выбрать используемую версию

библиотеки Qt и то, какая должна проводиться сборка: **Выпуск**, **Отладка** или **Профилирование**.

Если бы мы хотели только откомпилировать проект, то могли бы ограничиться самой нижней кнопкой в виде молоточка. Нажатие же второй снизу кнопки откомпилировало бы проект и запустило бы его в отладчике. Работу в режиме отладки мы рассмотрим немного позже.

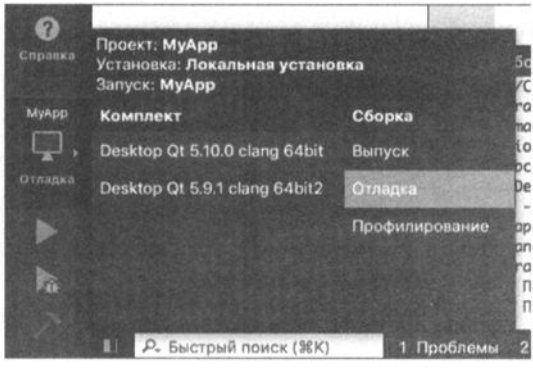


Рис. 47.12. Секция компиляции и запуска

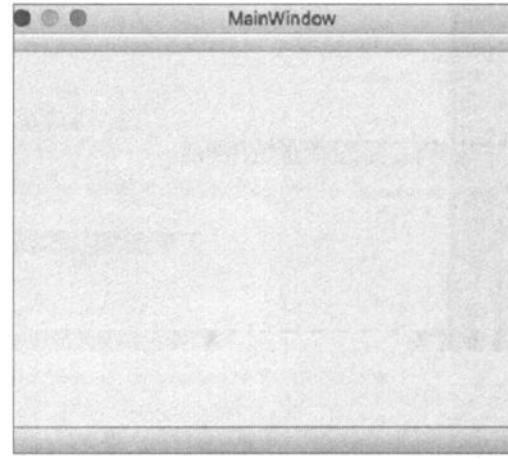


Рис. 47.13. Созданное приложение

Итак, мы нажали кнопку **Запуск**, и перед нами открылось окно приложения, сгенерированного мастером проектов (рис. 47.13). Само приложение ничего не делает, разве что отображает свое окно. Мы можем изменить это, модифицировав код исходных файлов проекта. Если вы помните, то нашим намерением было поздороваться с интегрированной средой разработки Qt Creator, поэтому давайте добавим в это окно надпись приветствия.

Добавить надпись можно двумя способами. Первый способ заключается в модифицировании исходных файлов программного кода (*.h, *.cpp) и добавлении в него виджета надписи. Второй способ заключается в использовании встроенной программы Qt Designer для добавления виджета надписи в файл формы (*.ui). Это очень просто сделать при помощи технологии drag & drop, и не требует написания ни единой строчки программного кода. Нужно щелкнуть двойным щелчком на файле нашей формы widget.ui, и она будет готова для изменений с помощью программы Qt Designer (рис. 47.14).

Теперь найдем в списке виджетов виджет надписи. Чтобы сделать это быстро, можно воспользоваться полем **Фильтр** (Filter) и вписать в него `Label`. По мере набора букв происходит отбрасывание всех виджетов, имена которых не удовлетворяют нашему критерию, и список виджетов уменьшается. В конечном итоге мы увидим только виджет надписи.

Теперь в секции **Display Widgets** выберем виджет **Label** и перетянем его в нашу форму. В результате этой операции вы увидите на форме надпись **TextLabel**. Это не тот текст, который мы хотели отобразить, поэтому его нужно изменить. Для этого щелкните двойным щелчком на тексте надписи либо нажмите на тексте правую кнопку мыши и выберите из появившегося контекстного меню пункт **Изменить текст**, после чего наша надпись перейдет в режим редактирования. Теперь можно набрать нужный нам текст: `Hello Qt Creator` (рис. 47.15).

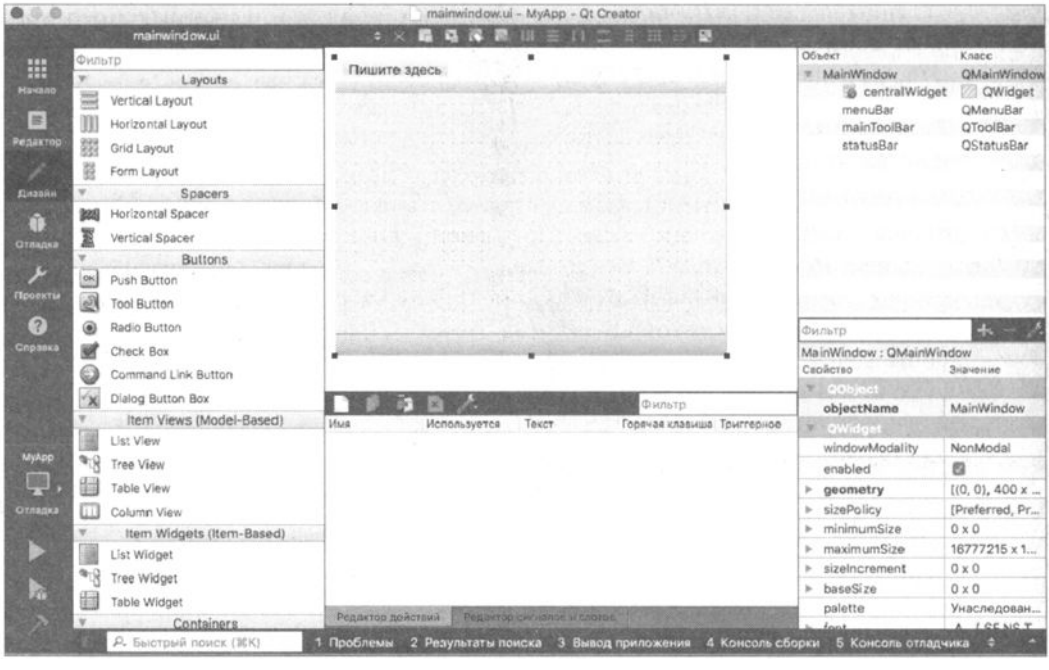


Рис. 47.14. Встроенная программа Qt Designer

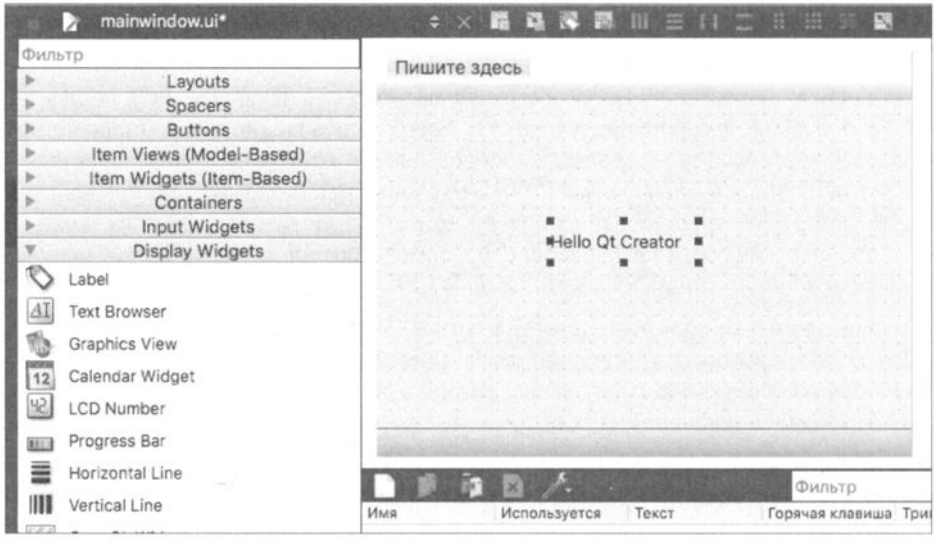


Рис. 47.15. Изменение надписи



Рис. 47.16. Приложение «Hello Qt Creator»

Снова нажмем кнопку **Запуск** и после успешно проведенного процесса компиляции и компоновки увидим окно с нужной нам надписью (рис. 47.16).

Редактирование текста

Теперь настало время познакомиться с возможностями и некоторыми особенностями инструмента, в котором разработчик проводит основное время, посвященное реализации программы. Это, конечно же, редактор.

Как подсвечен ваш синтаксис?

Редактор интегрированной среды разработки Qt Creator обладает прекрасной возможностью выделять разными цветами синтаксические элементы создаваемых программ, что значительно упрощает их чтение. Таким образом, выделив цветом отдельные элементы, например комментарии, ключевые слова, значения и переменные, мы быстро распознаем и визуально выделим их из текста программ. Это облегчает и обнаружение типичных синтаксических ошибок. Например, комментарии в программе по умолчанию выделяются зеленым цветом, и если мы вдруг откроем комментарий: `/*`, а потом в каком-то месте ошибемся с его закрытием, — например, поставим: `*)` вместо: `*/`, то это сразу же будет заметно, так как далее будет следовать бескрайняя череда зеленого текста. Или если вы опечатаетесь и наберете, скажем, `fir` вместо `for`, то это слово не будет окрашено в нужный цвет, и ошибка станет сразу же видна. Это позволяет выявлять многие ошибки тут же, а не в процессе компиляции.

По умолчанию в среде Qt Creator используются следующие основные цвета:

- фон — белый;
- текст — черный;
- числа — синий;
- строки — зеленый;
- ключевые слова — светло-коричневый;
- операторы — черный;
- директивы препроцессора — темно-синий;
- комментарии — зеленый.

Если вам не нравится эта расцветка, то вы можете изменить отдельные или все эти цвета, задав желаемые в меню **Настройки | Текстовый редактор**.

Скрытие и отображение кода

Иногда обилие исходного кода программы в открытых окнах редактора может создавать впечатление хаоса. Разработчики среды разработки Qt Creator учли это и предложили способ для решения проблемы. Он состоит в следующем: фрагменты программных кодов, которые вас в настоящий момент не интересуют, могут быть свернуты, и перестанут занимать место в редакторе. Если понадобится их посмотреть, то отображение свернутых фрагментов можно быстро восстановить. Для этого рядом с каждым фрагментом кода расположен знак стрелки (рис. 47.17), позволяющий либо отображать, либо сворачивать фрагмент.

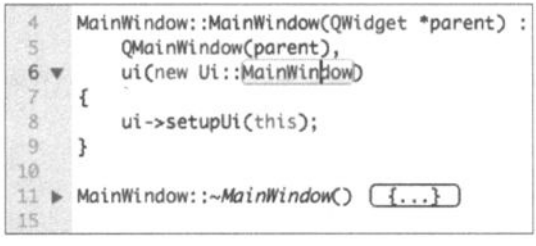


Рис. 47.17. Скрытие и отображение кода

Автоматическое дополнение кода

Если вы забыли точное название метода и/или количество и типы его параметров — не беда, при вводе текста программы в нужном месте среда разработки Qt Creator автоматически предложит вам выбор из списка соответствующих альтернатив. Если же вы хотите увидеть подсказку в определенном месте, то нажмите в нем комбинацию клавиш <Ctrl>+<Пробел>. Примечательно еще и то, что эта возможность распространяется также на сигналы и слоты, более того, дополнение выполняется не только после написанного имени объекта, а также и при их соединении. Эта возможность проиллюстрирована на рис. 47.18.

АВТОМАТИЧЕСКОЕ ДОПОЛНЕНИЕ ДЛЯ ФАЙЛОВ, НЕ ВХОДЯЩИХ В QT

Для того чтобы автоматическое дополнение работало так же хорошо и для используемых файлов, не входящих в Qt, необходимо следить за тем, чтобы их заголовочные файлы были указаны в секции `INCLUDEPATH` вашего проектного файла (**.pro*).

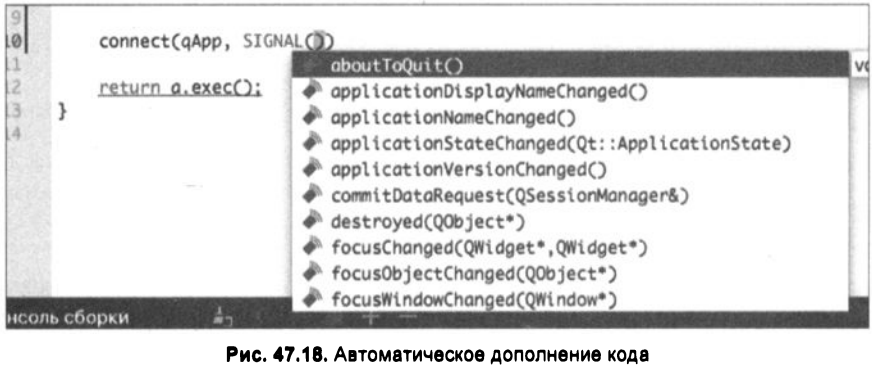


Рис. 47.18. Автоматическое дополнение кода

Поиск и замена

В процессе написания программ мы очень часто сталкиваемся с необходимостью поиска некоторых фрагментов в уже существующих файлах. Вместо того чтобы самостоятельно просматривать текст всей программы в поиске нужного места, доверьте это занятие редактору Qt Creator. Для выполнения поиска и замены текста можно открыть меню **Правка** и выбрать в нем пункт **Поиск/Замена** (рис. 47.19).

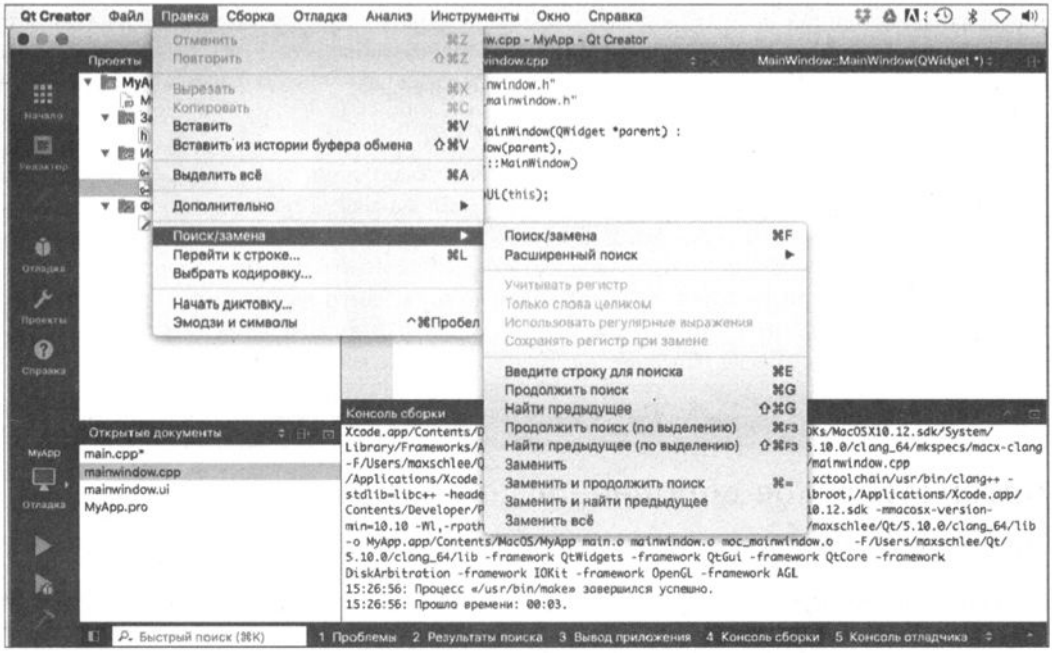


Рис. 47.19. Меню **Правка | Поиск/Замена**

Редактор войдет в режим поиска и замены (рис. 47.20), и внизу редактора появится дополнительная панель. В поле **Искать** можно задавать текст для поиска. Задав текст в поле **Заменить на**, можно заменять им найденные фрагменты, которые будут вам показаны перед заменой. Для того чтобы не просматривать все вхождения заменяемого кода, предусмотрена кнопка **Заменить все**. При ее нажатии все найденные вхождения текста, указанного в поле **Искать**, будут заменены значением поля **Заменить на**.



Рис. 47.20. Дополнительная панель для поиска в текущем файле

Чтобы найти какой-либо фрагмент текста программы, имеются шесть возможностей. Их предоставляет пункт меню **Правка | Поиск/Замена | Расширенный поиск** (см. рис. 47.19):

- **Все проекты** — осуществить поиск во всех загруженных проектах;
- **Открытые документы** — произвести поиск в файлах окна **Открытые документы** обозревателя проектов;
- **Символы C++** — поиск внутри кода C++ специфичных мест, таких как, например, перечисления, функции, классы и объявления;
- **Текущий проект** — провести поиск по текущему проекту;
- **Текущий файл** — искать в открытом редактором файле;
- **Файлы в системе** — найти в файлах на диске.

Выбрав режим **Файлы в системе**, вы увидите диалоговое окно, показанное на рис. 47.21. В этом окне в поле **Искать** можно задать текст для поиска. Также можно использовать регулярные выражения (см. *главу 4*), для чего нужно выбрать флажок **Использовать регулярные выражения**. Если вы хотите осуществить поиск не в текущем каталоге, то можете в раскрывающемся списке **Каталог** задать свой путь, а найти нужный путь вам поможет кнопка **Выбрать**. Если необходимо искать не только в файлах с расширениями cpp и h, надо в раскрывающемся списке **Шаблон** задать необходимые вам типы файлов.

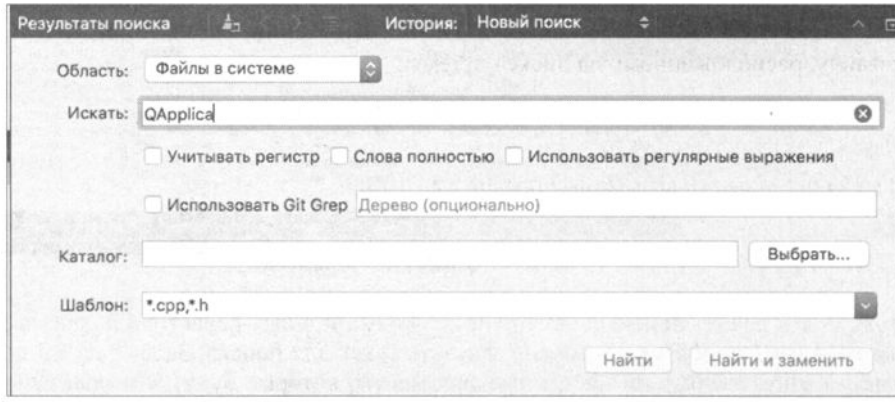


Рис. 47.21. Окно поиска в файлах на диске

Окна поиска текста при выборе режимов **Все проекты** и **Текущий проект** практически одинаковы (рис. 47.22). В них доступны все уже рассмотренные функции предыдущего окна, за исключением раскрывающегося списка **Каталог**, так как он не имеет в этом случае смысла, — ведь поиск осуществляется в проектах, местонахождение которых известно и изменению не подлежит. Обратите внимание: маска в раскрывающемся списке **Шаблон** настроена по умолчанию на все файлы (символ *).

Для поиска определенного класса, метода, строки кода и для просмотра документации Qt весьма полезна функция **Locator** (Локализатор). Кнопка ее запуска расположена возле кнопок окон вывода, а чтобы ее активировать, можно просто нажать комбинацию клавиш <Ctrl>+<K>.

Запуск функции Locator в Mac OS X

В Mac OS X вместо клавиши <Ctrl> нужно нажать клавишу <Command>.

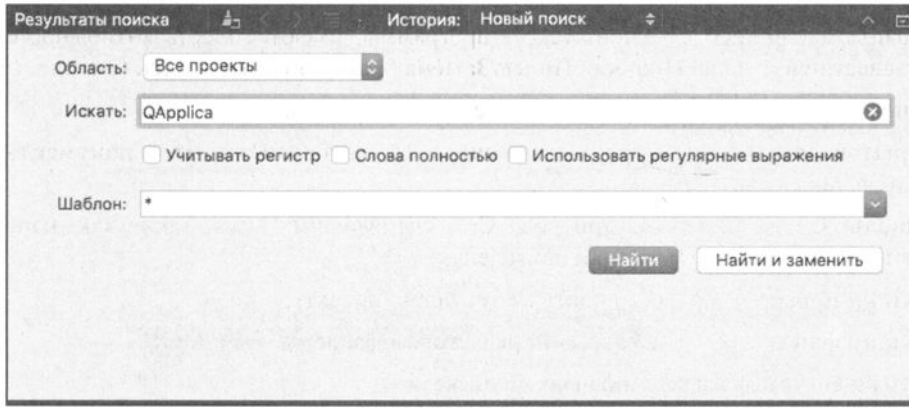


Рис. 47.22. Окно поиска в проектах

Функция предоставляет набор префиксов для локализатора, позволяющих переходить к следующим элементам проекта. Вот некоторые из них:

- l — к строке текущего документа;
- : — к классу и методу;
- o — к открытому файлу;
- ? — к статье помощи;
- f — к файлу, расположенному на диске системы;
- a — к файлу, расположенному в одном из загруженных проектов;
- p — к файлу текущего проекта.

На рис. 47.23 показаны все возможные префиксы.

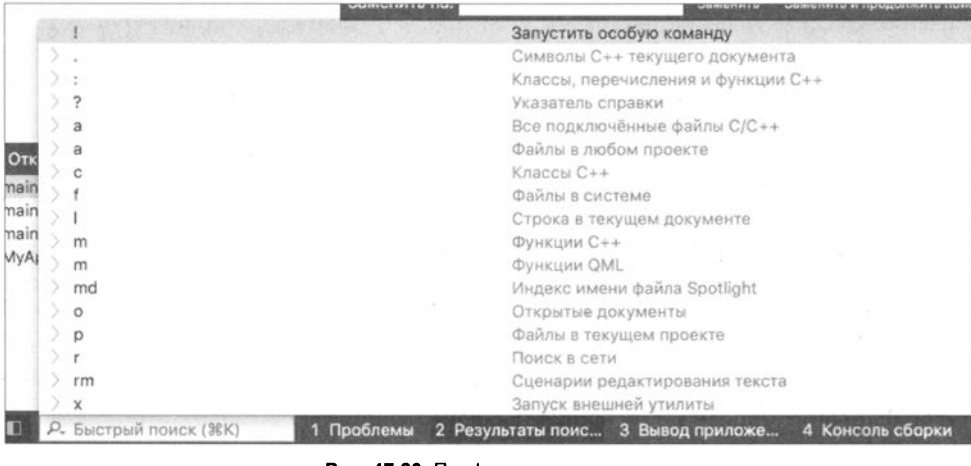


Рис. 47.23. Префиксы локализатора

Еще одна очень полезная функция, связанная с поиском, позволяет найти в проекте все места использования классов и методов. Вызвать ее можно, установив курсор редактора на нужном классе или методе, открыв контекстное меню и выбрав опцию **Найти использование** либо просто нажав комбинацию клавиш <Ctrl>+<Shift>+<U>.

Поиск мест использования в Mac OS X

В Mac OS X вместо клавиши <Ctrl> нужно нажать клавишу <Command>.

В окне результатов поиска будут отображены все файлы, в которых задействованы искомый класс или метод (рис. 47.24).

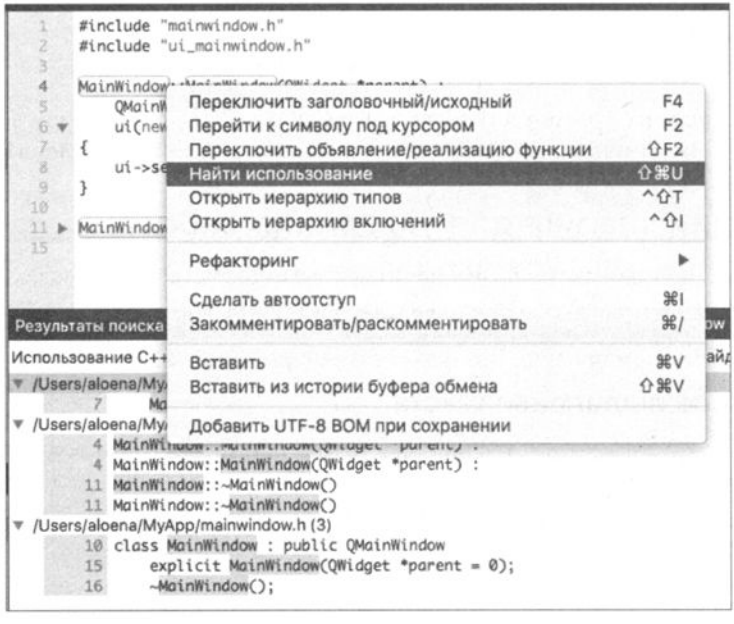


Рис. 47.24. Опция **Найти использование**

Бывает также, что возникает необходимость в изменениях имен классов, методов и/или переменных. Поскольку они используются в различных местах проекта, то сделать это вручную нелегко. Qt Creator предоставляет для этой цели функцию рефакторинга (рис. 47.25).

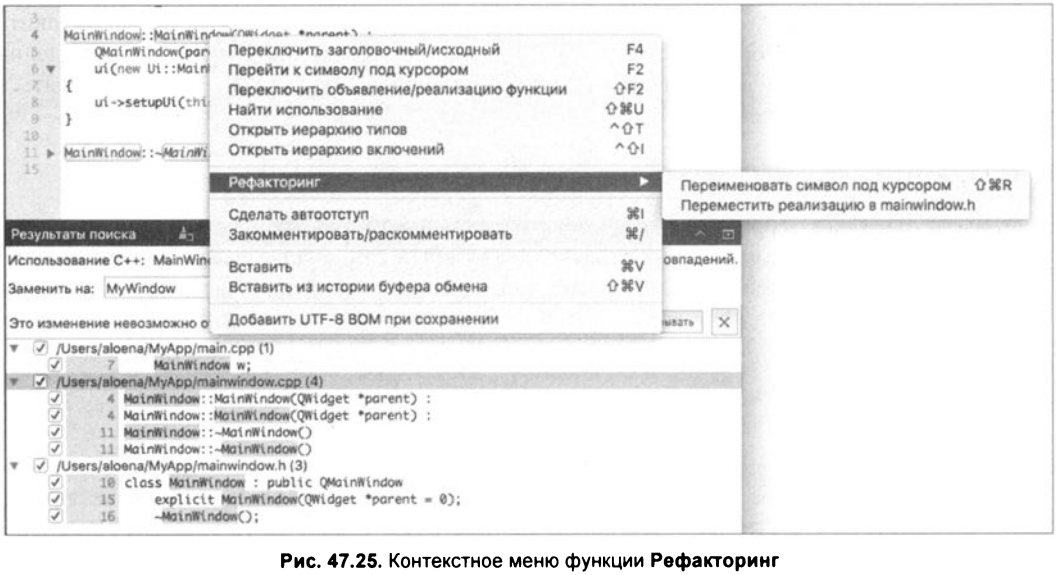


Рис. 47.25. Контекстное меню функции **Рефакторинг**

Чтобы ей воспользоваться, надо установить курсор редактора на нужный класс, метод или переменную и вызвать контекстное меню. Затем выбрать опцию **Рефакторинг | Переименовать символ под курсором** или нажать комбинацию клавиш <Ctrl>+<Shift>+<R> — в окне появятся результаты поиска всех мест использования класса, метода или переменной (см. рис. 47.25).

> ***Запуск рефакторинга в Mac OS X***
>
> В Mac OS X вместо клавиши <Ctrl> нужно нажать клавишу <Command>.

Нужно внимательно их просмотреть и снять флажки с тех мест, где замену делать не следует. Затем в поле **Заменить на** вписать новое имя и нажать на кнопку **Заменить**.

Комбинации клавиш для ускорения работы

Хотя некоторые программисты и предпочитают взаимодействовать с редактором, усердно работая мышью, знание всего нескольких клавиатурных комбинаций может значительно ускорить работу. Приведу несколько примеров.

Вертикальное выделение текста

Чтобы выделить нужные нам строчки текста, мы обычно удерживаем клавишу <Shift>, нажимаем левую кнопку мыши и, перемещая мышь, выделяем их. Если вместо клавиши <Shift> нажать <Alt> и повторить операцию, то мы сможем выделять вертикальные блоки программы (рис. 47.26).

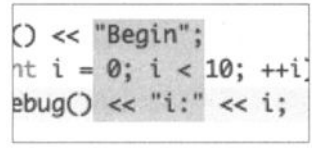


Рис. 47.26. Вертикальное выделение текста после нажатия клавиши <Alt> и перемещения мыши

Кусок выделенного кода теперь можно скопировать в буфер обмена и потом вставить в нужное место в редакторе Qt Creator либо в любом другом редакторе. Но если, выделив подобным образом текст, мы начнем вводить собственный текст, то он будет повторяться на каждой выделенной строке.

Автоматическое форматирование текста

Если вы встретились в программе с неотформатированным участком кода (подобный пример показан в левой части рис. 47.27), выделите нужный вам текст и нажмите комбинацию клавиш <Ctrl>+<I>, после чего текст примет вид, показанный в правой части рисунка.



Рис. 47.27. Автоформатирование комбинацией клавиш <Ctrl>+<I>

Автоматическое форматирование в Mac OS X

В Mac OS X вместо клавиши <Ctrl> нужно нажать клавишу <Command>.

Комментирование блоков

Чтобы не утруждать себя комментированием каждой строчки в отдельности, эту операцию можно осуществить сразу для всего выделенного фрагмента. Для этого просто выделите нужный вам фрагмент программного кода и нажмите комбинацию клавиш <Ctrl>+</> — выделенный блок будет закомментирован, как это показано на рис. 47.28.

Комментирование блоков в Mac OS X

В Mac OS X вместо клавиши <Ctrl> нужно нажать клавишу <Command>.

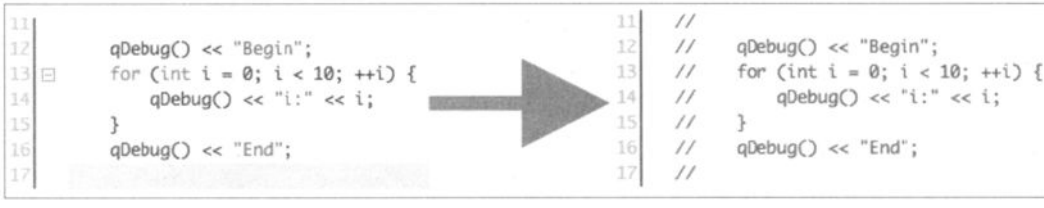


Рис. 47.28. Комментирование блоков программы с помощью комбинации клавиш <Ctrl>+</>

Просмотр кода методов класса, их определений и атрибутов

Очень часто, найдя в заголовочном файле (*.h) какой-либо метод, нам интересно посмотреть его реализацию. Для этого нам понадобится открыть файл реализации и найти нужный метод там. Но можно поступить и проще: если курсор находится на определении метода, то стоит нам нажать клавишу <F2>, как редактор откроет исходный файл с его реализацией (рис. 47.29). То же самое можно проделать в ситуации, когда мы находимся на месте реализации метода и хотим увидеть его определение.

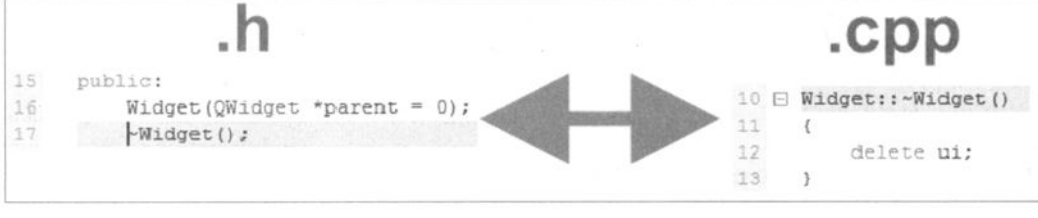


Рис. 47.29. Просмотр кода метода и место его определения с помощью клавиши <F2>

Аналогично только что описанной ситуации, часто возникает потребность найти место определения атрибута класса. Этот атрибут может располагаться и в заголовочном файле другого класса, от которого был унаследован текущий, что усложняет поиск вручную. При помощи Qt Creator наши действия сводятся к минимуму — достаточно установить курсор на имя интересующего нас атрибута, нажать клавишу <F2>, и нужный заголовочный файл будет открыт в редакторе.

Полезное подчеркивание...

Удерживая клавишу <Ctrl> (в Windows и Linux) или <Command> (в Mac OS X) и подводя указатель мыши к методам, функциям и атрибутам, вы увидите их подчеркивание, подобное

обозначению ссылок на веб-странице. Нажатие на такую «ссылку» произведет тот же эффект, что и при нажатии на клавишу <F2>. Преимущество заключается в том, что подчеркивание сразу сигнализирует о возможности получить дополнительную информацию.

Помощь, которая всегда рядом

Весьма удобно в процессе написания программ иметь оперативный доступ к помощи. Чтобы получить справочную информацию о классе объекта или о его методе, нужно установить курсор на интересующее имя и нажать клавишу <F1> — тут же откроется окно с необходимой справочной информацией, как это показано на рис. 47.30.

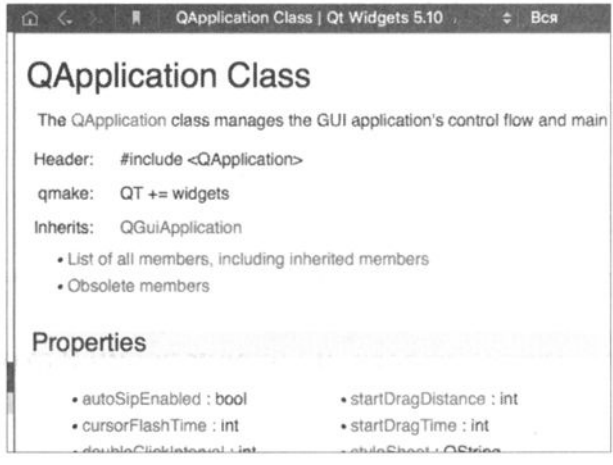


Рис. 47.30. Вывод помощи по нажатию клавиши <F1>

В табл. 47.1 указанные некоторые дополнительные клавиатурные комбинации, которые могут оказаться очень полезными для ускорения работы в Qt Creator.

Аналог клавиши <Ctrl> в Mac OS X

В Mac OS X вместо клавиши <Ctrl> нужно нажимать клавишу <Command>.

Таблица 47.1. *Некоторые полезные комбинации*

Комбинация	Выполняемое действие
<Esc>	Скрыть окна вывода
<Alt>+<Left>, <Alt>+<Right>	Навигация по истории
<Ctrl>+<E>	Выделить в программном коде все слова, совпадающие со словом, находящимся под курсором
<Ctrl>+<1>	Переключиться в режим **Начало**
<Ctrl>+<2>	Переключиться в режим **Редактирование**
<Ctrl>+<5>	Переключиться в режим **Проект**
<Ctrl>+<6>	Переключиться в режим **Помощь**
<Ctrl>+<0>	Увеличить рабочую область
<F4>	Переключиться между заголовочным файлом и файлом реализации

Интерактивный отладчик и программный экзорцизм

В *главе 3* мы уже познакомились с некоторыми приемами отладки программ при помощи отладчика GDB. Qt Creator предоставляет для отладчика GDB графический интерфейс. Огромное преимущество такого подхода — это возможность видеть и иметь доступ к большому количеству информации сразу. То есть, вы можете наблюдать и держать под контролем в процессе отладки много необходимой информации и в любое время обратиться к отладчику, чтобы отобразить значения определенных переменных. Поэтому такой отладчик и называется *интерактивным*.

Запустить свое приложение в отладчике Qt Creator очень просто — достаточно нажать клавишу <F5> (в Mac OS X — комбинацию клавиш <Command>+<Y>) или кнопку **Начать отладку**, и Qt Creator автоматически перейдет в специально созданный для отладки режим (рис. 47.31).

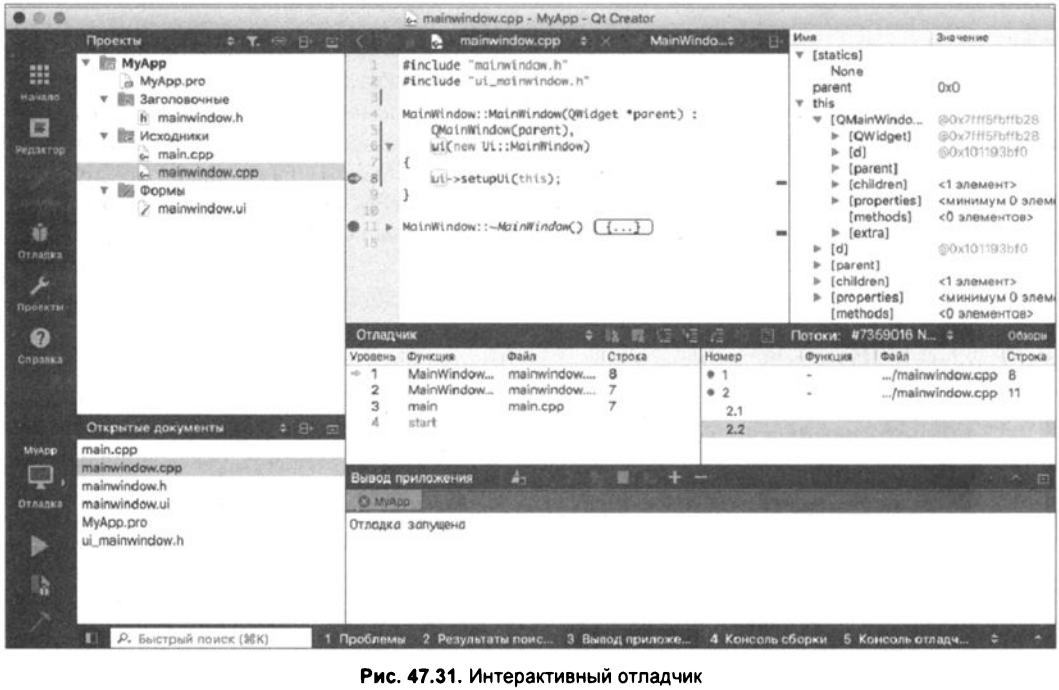


Рис. 47.31. Интерактивный отладчик

А теперь давайте поговорим немного об ошибках, ведь если бы не они, то отладчик нам бы был и не нужен. В идеальном мире, конечно же, каждая программа была бы совершенна при первом ее написании. Однако в действительности даже лучшие программисты делают ошибки или забывают обеспечить все ситуации. Чтобы минимизировать ошибки, чрезвычайно важно усвоить основы хорошего программирования и постоянно держать их в уме. Прекрасным источником для освоения хорошего стиля разработки является, на мой взгляд, книга Алена Голуба «Правила программирования на Си и Си++».

Главная цель отладчика — это обнаружение мест в программе, где находятся ошибки. Ошибки можно разделить на четыре типа:

- синтаксические ошибки;
- ошибки компоновки;
- ошибки времени исполнения;
- логические ошибки.

> ***Компилятор: сообщения об ошибках и предупреждения***
>
> В процессе компиляции помимо сообщений об ошибках вы можете сталкиваться также и с предупреждениями. Они появляются тогда, когда компилятор в состоянии правильно откомпилировать ваш код, но считает, что его выполнение может привести к возникновению проблем. Например, вы присваиваете переменной типа `int` значение типа `float` или создаете переменную, но ни разу ее не используете. В этих и подобных случаях компилятор предупредит вас и «поинтересуется», все ли здесь правильно. Помните, что в большинстве случаев предупреждения компилятора помогают избежать многих неприятностей, поэтому ни в коем случае не игнорируйте их, а относитесь к ним с должным вниманием.

Синтаксические ошибки

Это самые частые ошибки, которые возникают вследствие нарушения синтаксиса или грамматических правил языка C++. Ими могут быть передача в функцию неверного количества аргументов или аргументов не того типа, неправильно набранная команда, присвоение переменной или функции недопустимого имени и т. п. В подобных случаях компилятор не сможет вас понять, и программа не будет откомпилирована. И как только это случится, компилятор обратится к вам за помощью и отобразит список синтаксических ошибок в окне вывода (рис. 47.32). Все ошибки будут сопровождаться сообщениями о том, что вы что-то сделали неправильно. Щелкните двойным щелчком на таком сообщении, чтобы перейти к той строке программы, где эта ошибка произошла. Хотя почти всех программистов синтаксические ошибки раздражают, они на самом деле являются самыми безобидными. Просто внимательно изучите найденное место и, вполне возможно, вы сразу поймете, в чем дело. Если же сходу понять не удалось, то постарайтесь при помощи документации или другого источника распознать причину и откорректировать проблемные места. Ведь для того чтобы программа была успешно откомпилирована, все синтаксические ошибки должны быть обязательно исправлены.

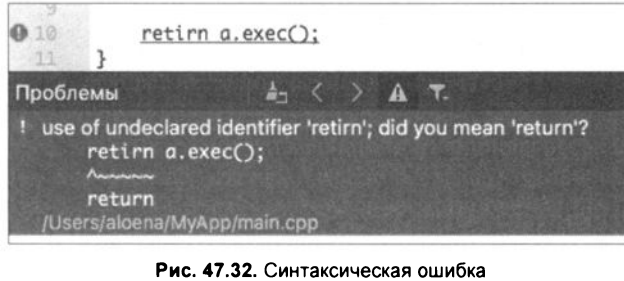


Рис. 47.32. Синтаксическая ошибка

Некоторые из синтаксических ошибок можно также увидеть при помощи расцветки синтаксиса. Так, в примере, показанном на рис. 47.32, мы можем сразу заметить, что слово `return` написано неправильно, и тут же на месте исправить ошибку до запуска компилятора. Много синтаксических ошибок случается также из-за недостающих или лишних фигурных и круглых скобок, и тут снова вам поможет расцветка синтаксиса.

Вот некоторые из рекомендаций, которые могут сэкономить вам время на поиск причин возникновения сообщения о синтаксической ошибке:

- перед компиляцией проверьте, сохранили ли вы все измененные вами файлы;
- если вы не можете найти ошибку в указанной строке, проверьте предыдущие, бывает, что ошибка именно там;
- сверьте каждую букву и убедитесь, что вы правильно набрали имена переменных и функций. Очень многие ошибки происходят именно из-за опечаток;
- если ничего не помогает, и вы никак не в силах понять причину возникновения ошибки, то попросите кого-нибудь о помощи. Свежий взгляд на вещи очень часто помогает найти то, что «замыленный» мог упустить.

Ошибки компоновки

Такой вид ошибок возникает при невозможности создания исполняемого файла. Это может быть неправильное использование библиотек или недостающие библиотеки, объектные файлы и т. п. Рассмотрим самый простой способ, которым можно вызвать этот тип ошибки. Просто попробуйте откомпилировать и запустить программу, а затем, не закрывая ее, перекомпилировать ее заново. Вы получите сообщение об ошибке, так как файл запущенной программы не может подвергаться изменениям до выхода из программы (впрочем, на Mac OS X и Linux это не распространяется).

На рис. 47.33 приведен еще один пример ошибки — на этот раз компоновщика: мы объявили функцию, но забыли ее реализовать. Заметьте, что сообщение об ошибке пришло не от компилятора, а от компоновщика, — создание компилятором объектного кода для показанного файла пройдет успешно, но на этапе компоновки не будет найден объектный код для функции `test()`.

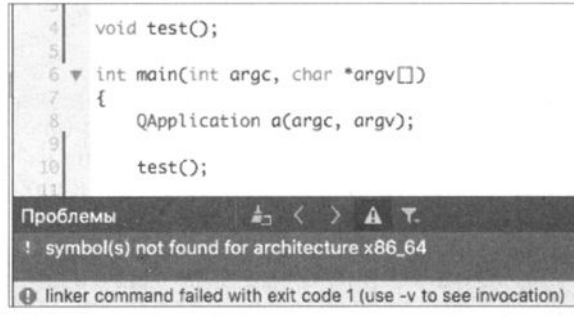


Рис. 47.33. Ошибка компоновки

Так же, как и в случае с синтаксическими ошибками, для успешного создания исполняемого файла программы необходимо устранить все ошибки компоновки.

Ошибки времени исполнения

После того как будут устранены все (если они были) синтаксические ошибки и ошибки компоновки, компилятор создаст исполняемый файл. Но ошибки могут случаться и во время исполнения программы, причем здесь они могут приводить к исключительным ситуациям и к аварийному завершению программы, — так называемому *крэшу*. Типичными причи-

нами крэша могут быть, например, деление на ноль, извлечение корня из отрицательного числа, выход за границы зарезервированной памяти, ошибки устройств и т. д. Подобные ошибки случаются не часто, но надежная, устойчивая система должна быть к ним готова.

Логические ошибки

Из всех четырех упомянутых нами категорий ошибок, логические ошибки найти наиболее трудно, так как их истоки кроются в ошибочном рассуждении разработчика на пути поиска решения поставленной задачи. Здесь сценарий такой: приложение вроде бы выполняется без ошибок, однако выдает неверные результаты. Еще хуже, если неверные результаты получаются не всегда, а только в редких ситуациях. Такие феномены обычно обнаруживаются лишь при анализе спецификации и в результате работы с самой программой, на основании чего можно сделать вывод, правильно ли она работает. К сожалению, компилятор не в состоянии оценить смысл написанного и проверяет только правильность использования синтаксиса, поэтому лишь тщательное тестирование в самых разнообразных условиях и с различными исходными данными может дать гарантию того, что программа не содержит логических ошибок. Модульное тестирование, описанное в *главе 45*, способно свести появление логических ошибок к минимуму.

Одним из самых простых способов локализации и нахождения в программе логических ошибок является *трассировка* — пошаговое прослеживание всех операторов программы. Отладчик может при этом отображать значения указанной переменной или выражений в любой точке программы. Другой, тоже очень простой способ — это использование вывода промежуточных результатов при помощи функции `qDebug()`. Ее вставка в стратегических точках программы помогает обнаружить необычное ее поведение, что часто приводит к раскрытию логических ошибок. Этот способ можно использовать и отдельно от отладчика. В силу своей универсальности он выручил много программистов и спас огромное количество программ. Использование функции `qDebug()` мы уже рассмотрели в *главе 3*, поэтому остановимся здесь лишь на трассировке.

Трассировка

Встаньте на первую строку функции `main()` и нажмите клавишу <F9> (в Mac OS X — клавишу <F8>). Вы увидите появившийся кружок красного цвета (рис. 47.34) — это одна из контрольных точек, их мы рассмотрим позже. А теперь нажмите клавишу <F5> (в Mac OS X — комбинацию клавиш <Command>+<Y>) — по завершении цикла компиляции и компоновки вы окажетесь в интегрированном отладчике Qt Creator, и программа начнет выполнение с первого оператора в функции `main()`. Если в программе были установлены контрольные точки, программа будет прерывать свое выполнение в них. Но если контрольных точек нет, то выполнение программы будет продолжаться до тех пор, пока она не закончится нормально или аварийно. В нашем случае установлена всего одна контрольная точка.

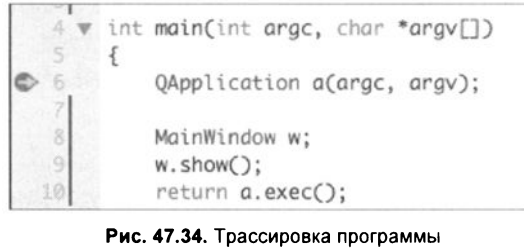


Рис. 47.34. Трассировка программы

На рис. 47.34 показана часть окна редактора, на которой видна трассировочная стрелка отладчика. Наличие такой стрелки информирует нас, что отладчик запущен. Как можно видеть, трассировочная стрелка находится напротив определения объекта класса `QApplication`. Важно помнить, что стрелка показывает не на строку, которую вы только что выполнили, а на ту, которая будет выполняться, когда вы сделаете следующий шаг трассировки.

Кнопки команд трассировки расположены на панели, показанной на рис. 47.35. С их помощью можно выполнить одну строку программы. После выполнения одной строки программа вновь приостанавливается, и вы можете посмотреть значения переменных или выполнить другие команды отладчика.



Рис. 47.35. Панель команд трассировки

Команд трассировки всего три:

- **Step Over** (Перешагнуть);
- **Step Into** (Войти);
- **Step Out** (Выйти).

Помимо кнопок трассировки первыми слева на панели трассировки (см. рис. 47.35) расположены кнопки: для продолжения выполнения программы — **Continue** и останова отладчика — **Stop Debugger**. Команда **Stop Debugger** приостанавливает выполнение программы в том месте кода, которое будет достигнуто на момент нажатия кнопки команды. Команда **Continue** служит для продолжения выполнения программы после ее приостановки командой **Stop Debugger** или при достижении контрольной точки. Самая крайняя кнопка справа осуществляет показ дизассемблированного машинного кода программы.

Команда *Step Over*

После подачи этой команды выполняется текущая строка программы и происходит остановка на следующей строке. Если текущая строка являлась вызовом функции, она будет выполнена вся, если, конечно, в этой функции не была расположена контрольная точка. Подав эту команду, мы увидим, что будет выполнена текущая строка, а трассировочная стрелка перейдет к следующей строке (рис. 47.36).

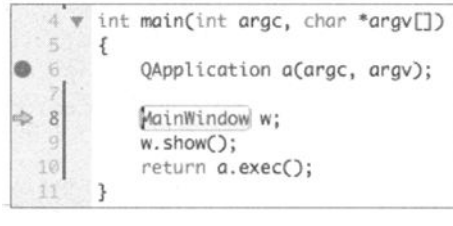


Рис. 47.36. Выполнение команды **Step Over**

Команда *Step Into*

Эта команда также выполняет текущую строку программы, но если она является вызовом, то отладчик проследует далее и остановится на первой строке внутри этой функции или

метода. То есть, этой командой нам предоставляется прекрасная возможность входить в вызываемую функцию или метод и затем выполнять их построчно. Подав эту команду, мы окажемся в конструкторе нашего класса `MainWindow` (рис. 47.37). Надо также иметь в виду, что при работе со стандартными функциями часто отдают предпочтение команде **Step Over**, так как содержимое этих функций редко представляет интерес.

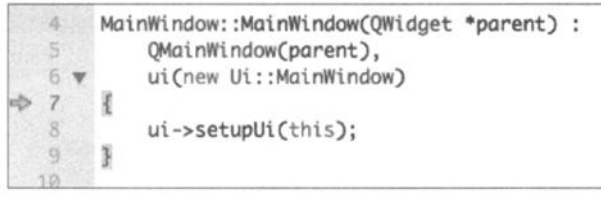


Рис. 47.37. Выполнение команды **Step Into**

Команда *Step Out*

Эта команда заставляет программу завершить функцию или метод, в которой находится отладчик, и приготовиться перейти к следующей строке. В нашем примере, как только мы воспользуемся этой командой, отладчик выйдет из конструктора и приготовится к переходу на вызов метода `show()`, который следует сразу за ним (рис. 47.38).

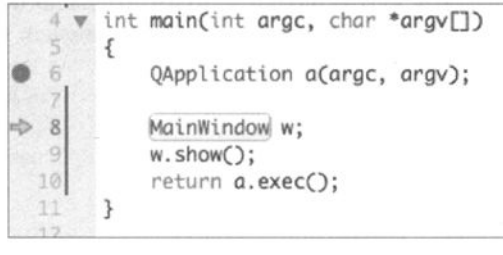


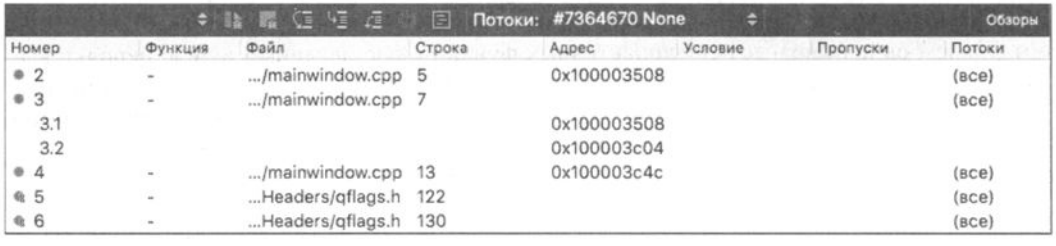
Рис. 47.38. Выполнение команды **Step Out**

Контрольные точки

Теперь поговорим немного о контрольных точках, которыми мы уже успели воспользоваться в нашем примере. Подчас, чтобы подойти к интересующему нас месту программы, приходится очень долго отдавать команды **Step Into** и **Step Over**, прогоняя отлаженные ранее участки программы. Контрольные точки представляют собой выделенные предложения, указывающие, где необходимо приостановить выполнение программы, для того чтобы разобраться, что происходит. В нужном месте можно просто установить контрольную точку и запустить отладчик — мгновение, и мы уже там.

Контрольные точки — фундамент для отладки, и Qt Creator делает их установку и удаление очень простыми. Чтобы установить точки контроля, нужно поставить курсор редактора на нужную строку кода и щелкнуть на левом ее поле или нажать клавишу <F9> (в Mac OS X — клавишу <F8>) — в начале строки появится метка в виде красного кружка, указывающая, что здесь установлена контрольная точка. Для удаления контрольной точки необходимо щелкнуть мышью на красном кружке или поставить курсор редактора на нужную строку с контрольной точкой и нажать клавишу <F9> (в Mac OS X — клавишу <F8>).

Количество контрольных точек не ограничено, и вы можете установить их столько, сколько вам нужно. Чтобы не запутаться в том, сколько контрольных точек в программе и где они находятся, существует специальное окно, отображающее их все (рис. 47.39). Двойной щелчок на интересующей вас точке сразу же откроет в редакторе нужный файл и поместит курсор на месте ее установки.



Номер	Функция	Файл	Строка	Адрес	Условие	Пропуски	Потоки
2	-	.../mainwindow.cpp	5	0x100003508			(все)
3	-	.../mainwindow.cpp	7				(все)
3.1				0x100003508			
3.2				0x100003c04			
4	-	.../mainwindow.cpp	13	0x100003c4c			(все)
5	-	...Headers/qflags.h	122				(все)
6	-	...Headers/qflags.h	130				(все)

Рис. 47.39. Окно контрольных точек

Окно переменных (Local and Watches)

При выполнении кода в отладчике основное внимание должно быть направлено прежде всего на проверку значений критических переменных. Именно для этого существует отдельное окно, где отображается вся информация, которую вам, возможно, потребуется знать о переменной (рис. 47.40). Это окно содержит три колонки переменного размера. Отладчик автоматически отображает в этих колонках имя, значение и тип переменных.

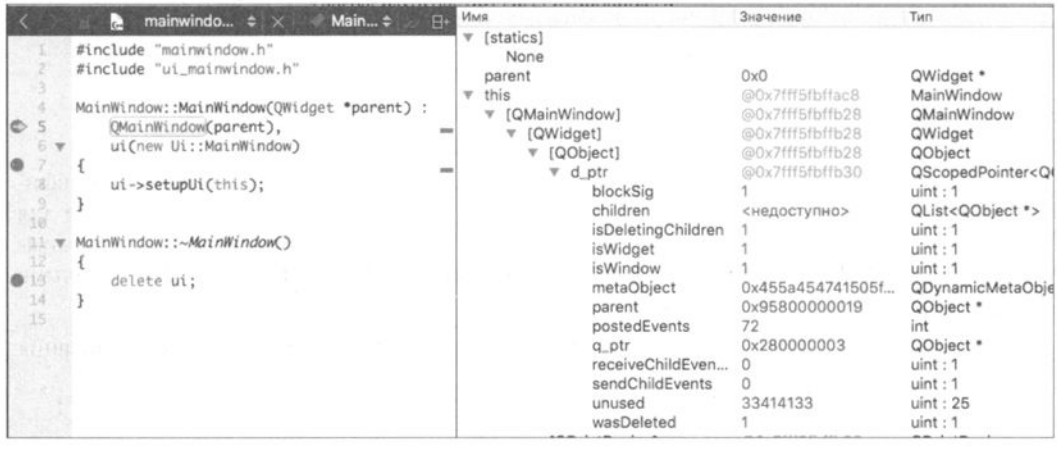


Рис. 47.40. Окно переменных (*справа*)

Если в строчке показан массив, объект или структурная переменная, то рядом с именем отображается треугольник. Щелкнув на нем, можно свернуть или развернуть представление переменной. При последовательном разворачивании переменной и ее компонентов отображается дерево из объектов, которые она содержит. Повторным щелчком можно свернуть переменную.

Поскольку среда Qt Creator разработана для упрощения процесса отладки Qt-приложений, она предоставляет рассчитанные на специфику Qt-классов дополнительные возможности, недоступные в других отладчиках. Соответственно, переменные, за которыми вы наблюдае-

те, при отладке предоставляются в удобной для чтения разработчиком форме. Например, если вы наблюдаете за переменной типа `QFile`, то первым вы увидите имя файла. То же касается и объектов класса `QList`, которые отображаются как настоящие списки со значениями.

Окно цепочки вызовов (Call Stack)

В отдельном окне выводится список активных функций, составленный в виде цепочки вызовов, которая отображает порядок обращения к функциям. Например, если при выполнении функции `main()` был вызван конструктор `MainWindow`, а в нем был вызван метод `setupUi()` и т. д., то цепочка вызовов будет выглядеть, как показано на рис. 47.41.

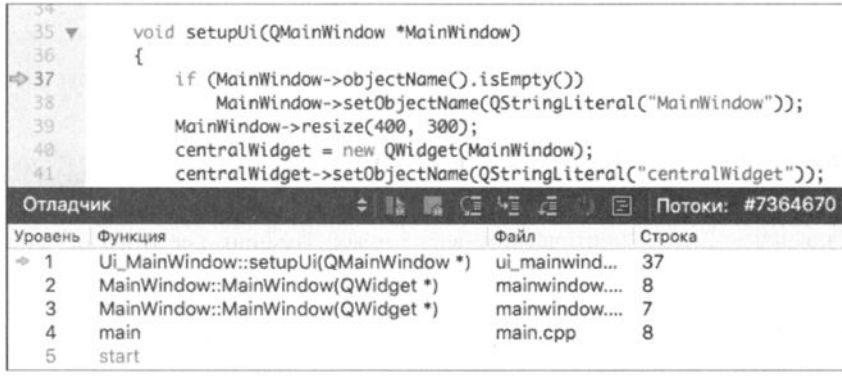


Рис. 47.41. Окно цепочки вызовов

Резюме

Qt Creator — это интегрированная среда разработки, специально рассчитанная на специфику библиотеки Qt. Она предоставляет разработчику весь необходимый инструментарий для создания программ. Ее использование облегчает работу и позволяет редактировать и компилировать программы без необходимости применения других приложений. Вот типичные действия, которые можно выполнить в среде разработки Qt Creator:

- создать базовое исполняемое приложение при помощи мастера проектов без написания кода;
- редактировать исходный код программ;
- добавлять и создавать новые файлы;
- создавать графическую оболочку для приложения интерактивно, при помощи встроенной программы Qt Designer;
- компилировать и компоновать программы;
- отлаживать готовые программы.

Для отладки программ Qt Creator предоставляет интерактивный отладчик. Существуют ошибки синтаксиса, компоновки, времени исполнения и логические ошибки. Отладчик используется для выявления места и причин возникновения двух последних типов ошибок. Для этого отладчик позволяет устанавливать контрольные точки, предоставляет команды

трассировки: **Step Over**, **Step Into**, **Step Out** и окна наблюдения за переменными и цепочками вызовов.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/47-510/** или с помощью следующего QR-кода (рис. 47.42):



Рис. 47.42. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 48

Рекомендации по миграции программ из Qt 4 в Qt 5

Раз, два, три, ЧЕТЫРЕ, ПЯТЬ, вышел зайчик погулять.
Народная считалка

Разработчики Qt сделали все возможное, чтобы переход с версии 4 на версию 5 был простым и гладким. И хотя результат совместимости Qt 5 с предшествующей версией был достигнут действительно очень хороший, полной совместимости добиться все-таки не удалось. О том, на что нужно обратить внимание при миграции ваших программ на Qt 5, а также и о сохранении обратной совместимости с Qt 4 (если вам она нужна), здесь и пойдет речь.

Эта глава может также оказаться полезной, чтобы вы могли понять, что вас ожидает, и примерно оценить количество времени и объем работ, которые вам могут понадобиться для миграции ваших проектов на Qt 5.

Основные отличия Qt 5 от Qt 4

Прежде чем приступить к переводу своих программ на Qt 5, необходимо разобраться в основных различиях версий. Это поможет более осознанно принимать решения в процессе перевода. Если вы ознакомились с предыдущими главами этой книги, то, наверняка, некоторые из этих различий уже заметили, но, тем не менее, я думаю, нелишним будет еще раз обратить на них ваше внимание.

Вот одно из существенных изменений — все виджеты «переехали» из модуля `QtGui` в новый модуль `QtWidgets`.

Qt 5 больше не содержит модуль `Qt3Support`, который использовался в Qt 4 для поддержки Qt 3. Поэтому, если ваш проект все еще использует его, то вам понадобится, наконец, разорвать эту зависимость и произвести рефакторинг, чтобы расстаться с этим модулем.

Модуль `Phonon` тоже больше не входит в состав Qt 5. Поэтому вместо него придется использовать модуль `QtMultimedia` (см. *главу 27*).

Теперь перейдем от общего к частному и рассмотрим подробности перевода более конкретно.

Подробности перевода на Qt 5

Первое, чем нужно запастись для начала перевода, — это терпением. Иногда, возможно, вам придется обращаться к оригинальной документации Qt — будьте к этому готовы, поэтому, если вы еще не успели подружиться с программой Qt Assistant, то сейчас самое вре-

мя это сделать. Особое внимание обратите в ней на разделы «Porting Guide» и «Porting C++ Application to Qt 5», а если вы реализовывали в Qt 4 QML-код, то обратите внимание еще и на раздел «Porting QML Applications to Qt 5». В этих разделах вы найдете ответы на большинство интересующих вас вопросов, связанных с переводом своих программ на Qt 5.

Виджеты

Итак, как мы уже знаем, все виджеты переместились в Qt 5 в отдельный модуль `QtWidgets`, и это изменение принесло с собой неприятные последствия, так как теперь появилась необходимость во всех исходных файлах, которые включали модуль `QtGui`, поменять его на `QtWidgets`. То есть, строка:

```
#include <QtGui>
```

должна быть заменена на:

```
#include <QtWidgets>
```

> ***Будет полегче...***
>
> Если же вы во всем проекте не использовали включение модулей, а включали всегда только заголовочные файлы классов виджетов, то эта операция замены вам не нужна.

Во всех проектных файлах необходимо добавить модуль виджетов — примерно так:

```
QT += widgets
```

> ***Утилита fixqt4headers.pl***
>
> Вносить изменения в большое количество исходных файлов вручную — утомительная задача, поэтому Qt 5 предоставляет утилиту *fixqt4headers.pl*, и ее можно найти в каталоге *qtbase/bin/*. Прежде чем вы запустите утилиту, позаботьтесь о резервной копии всех файлов проекта.

Класс `QWorkspace` удален, и теперь вместо него нужно использовать класс `QMdiArea`. Его замена не предоставляет собой трудностей. Нужно просто заменить строку:

```
#include <QWorkspace>
```

на строку:

```
#include <QMdiArea>
```

и далее произвести замену имен и нескольких методов.

Контейнерные классы

Класс `QString` больше не предоставляет методы `toAscii()` и `fromAscii()`. Поэтому, если вы эти методы использовали, то замените их на `toLatin1()` и `fromLatin1()` соответственно. Например, строку:

```
QByteArray ba = str.toAscii();
```

необходимо заменить строкой:

```
QByteArray ba = str.toLatin1();
```

Функция `qVariantValue()` не вошла в Qt 5. Теперь необходимо вместо нее использовать метод `QVariant::value<T>()`. Соответственно, следующий код с объектом `varPix`:

```
QPixmap pix = qVariantValue<QPixmap>(varPix);
```

нужно заменить вызовом метода `value<T>()`:

```
QPixmap pix = varPix.value<QPixmap>();
```

Функция *qFindChildren<T>()*

Глобальная функция `qFindChildren<T>()`, предназначением которой было возвращать список дочерних объектов, в Qt 5 ликвидирована. Ее необходимо теперь заменять везде методом `QObject::findChildren<T*>()`. Например, строку:

```
QList<QWidget*> lst = qFindChildren<QWidget*>(pwgt);
```

нужно заменить строкой:

```
QList<QWidget*> lst = pwgt->findChildren<QWidget*>();
```

Сетевые классы

Классы `QFtp` и `QHttp` из Qt 5 убраны. Вместо них нужно теперь использовать класс `QNetworkAccessManager` (см. *главу 39*).

WebKit

Модуль `QtWebKit` в Qt 5 разбит на два разных модуля: `QtWebKit` и `QtWebKitWidgets`, поэтому в исходных файлах необходимо заменить строку:

```
#include <QWebKit>
```

на строку:

```
#include <QWebKitWidgets>
```

А также не забыть добавить в проектный файл сам модуль:

```
QT += webkitwidgets
```

> ***Используйте модуль Qt WebEngine***
>
> Модуль WebKit признан устаревшим, поэтому он к использованию более не рекомендуется, вместо него теперь нужно использовать модуль Qt WebEngine (см. *главу 46*).

Платформозависимый код

Если вы реализовывали в проектах платформозависимый код и использовали макросы вида `Q_WS_*`, то их все нужно заменить макросами вида `Q_OS_*` следующим образом:

- для кода под Windows: `Q_WS_WIN` заменить на `Q_OS_WIN`;
- для кода под Mac OS X: `Q_WS_MACX` заменить на `Q_OS_OSX`;
- для кода под Linux: `Q_WS_X11` заменить на `Q_OS_LINUX`.

Система расширений Plug-ins

Плагины в Qt 5 с целью оптимизации реализованы иначе, чем в Qt 4 — макросы `Q_EXPORT_PLUGIN` и `Q_EXPORT_PLUGIN2` в Qt 5 более не используются. Появился новый макрос `Q_PLUGIN_METADATA`.

Преимущество нового подхода заключается в том, что теперь Qt может обращаться к метаданным плагина, не производя загрузку их динамических библиотек, и брать всю интересующую метаинформацию из JSON-файла, который должен быть у каждого плагина свой. Новый макрос `Q_PLUGIN_METADATA` объявляется сразу после объявления макроса `Q_OBJECT`, содержит IID и ссылку на JSON-файл и имеет следующий вид:

```
Q_PLUGIN_METADATA(IID "com.neonway.Aurochs.IndInterface" FILE "indicator.json")
```

Принтер *QPrinter*

Весь функционал для работы с принтером, включая его виджеты, перемещен в отдельный модуль `QPrintSupport`. Этот модуль необходимо включить в проектный файл следующим образом:

```
QT += printsupport
```

Мультимедиа

В классе `QAudioFormat` изменились имена двух методов: метод `QAudioFormat::setFrequency()` нужно везде заменить на `QAudioFormat::setSampleRate()`, а `QAudioFormat::setChannels()` — на `QAudioFormat::setChannelCount()`. При этом передаваемые в методы параметры должны остаться прежними.

Модульное тестирование

Включение модуля тестирования в проектном файле теперь должно происходить не в секции `CONFIG`, а в секции `QT` и с другим именем. То есть, строку:

```
CONFIG += qtestlib
```

необходимо заменить строкой:

```
QT += testlib
```

Реализация обратной совместимости Qt 5 с Qt 4

> За двумя зайцами погонишься — ни одного не поймаешь, но зато согреешься!
>
> *Народная мудрость*

Если вас заинтересовал этот раздел, то, скорее всего, вы еще до конца не уверены, что хотите полностью перейти на Qt 5 и, следовательно, хотите оставить возможность компилировать программы как в Qt 5, так и в Qt 4. Это может быть также связано с необходимостью поддерживать проекты на архитектурах, которые Qt 5 пока не поддерживает, — таких как, например, PowerPC, а, может, вы не хотите расставаться с модулем `Phonon`, собираетесь реализовывать программы для BlackBerry 10 на оригинальном API или имеете еще какие-либо другие причины. Вообще, причины для этого могут быть разными, но сразу оговорюсь, что поддержание обратной совместимости может стать «капризной вещью». Она может срываться каждый раз, когда в код программы добавляется что-то новое из Qt 5, чего нет в Qt 4. Поэтому за обратной совместимостью вам придется каждый раз следить отдельно и стараться, чтобы участки подобного кода под Qt 4 либо не компилировались, либо были реализованы как-то иначе. То есть, самый верный путь не поломать обратную совместимость — это использовать классы и методы только из арсенала Qt 4.

Первое, что нужно для обратной совместимости, — это pri-файл, который должен включаться в те проектные файлы, где она окажется необходима. В листинге 48.1 вы видите реализацию как раз такого файла. Он будет являться отправной точкой для всех проектов с обратной совместимостью, и вы всегда при необходимости сможете вносить в него собственные изменения.

Листинг 48.1. Обратная совместимость (файл main.pri)

```
greaterThan(QT_MAJOR_VERSION, 4) :
    DEFINES += Q_QT5
}
else: macx {
    DEFINES += Q_OS_OSX Q_QT4
}
else {
    DEFINES += Q_QT4
}
```

Основная часть реализации относится к Mac OS X — мы добавляем определение `Q_OS_X`, которого в Qt 4 нет. В остальном же мы просто на всех платформах добавляем определение `Q_QT4` или `Q_QT5` — в зависимости от того, какая версия Qt задействована для компиляции проекта. Это делается с тем расчетом, чтобы в исходных файлах не использовать более громоздкое препроцессорное выражение:

```
#if (QT_VERSION >= QT_VERSION_CHECK(5, 0, 0))
    // специфичный Qt5-код
#endif
```

Теперь вместо него мы можем включить наше определение:

```
#ifdef Q_QT5
    // специфичный Qt5-код
#endif
```

Так что, всякий раз, когда вы имеете дело с проектом, который нуждается в обратной совместимости, не забудьте включать в него pri-файл (см. листинг 48.1). А так же и модуль `QWidgets`, если проект собирается с Qt 5. Соответственно, если проект собирается с Qt 4, то модуль `QWidgets` включаться не должен. Вот пример того, как могут быть реализованы упомянутые ранее моменты в pro-файле:

```
include(main.pri)
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
```

Перейдем теперь к файлам реализации. Когда нам нужны виджеты, мы включаем в Qt 4 модуль `QtGui`, а в Qt 5 — модуль `QtWidgets`. То есть, в каждом файле нам нужно будет сделать следующее:

```
#ifdef Q_QT5
#include <QtWidgets>
#else
#include <QtGui>
#endif
```

Это выглядит достаточно громоздко, учитывая, что эти определения нужно прописывать всякий раз, когда нам понадобится включать модуль с виджетами. Так что, эту реализацию лучше всего поместить в отдельный файл и включать уже его. Назовем этот заголовочный файл QtWidgetsBridge.h и произведем в секции `Q_QT5` определение типа `WFlags` для совместимости с кодом Qt 4 (листинг 48.2). Рассматривайте этот файл как центральный файл для обратной совместимости виджетов и не стесняйтесь добавлять в него при необходимости новые определения.

Листинг 48.2. Обратная совместимость виджетов (файл QtWidgetsBridge.h)

```
#ifdef Q_QT5
#include <QtWidgets>
namespace Qt {
typedef WindowFlags WFlags;
}
#else
#include <QtGui>
#endif
```

В Qt 5 из класса `QDesktopServices` удалены методы получения стандартных путей системы, поэтому для этих целей необходимо использовать теперь класс `QStandardPaths`. Для того чтобы обеспечить обратную совместимость, необходимо реализовать свой собственный класс, который послужил бы «мостом» между `QDesktopServices` и `QStandardPath`. Назовем этот класс `MStandardPaths`. В листинге 48.3 показана его реализация.

Листинг 48.3. Класс «моста» между QStandardPaths и QDesktopServices (файл MStandardPaths.h)

```
#ifdef Q_QT5
#include<QStandardPaths>
#else
#include<QDesktopServices>
#include<QStringList>
#endif

class MStandardPaths {
public:
    enum MStandardLocation {
        DesktopLocation = 0,
        DocumentsLocation,
        FontsLocation,
        ApplicationsLocation,
        MusicLocation,
        MoviesLocation,
        PicturesLocation,
        TempLocation,
        HomeLocation,
        DataLocation,
        CacheLocation,
```

```
        // Qt5 specific locations
        GenericCacheLocation,
        GenericDataLocation,
        RuntimeLocation,
        ConfigLocation,
        GenericConfigLocation
    };

    static QStringList standardLocation(MStandardLocation type)
    {
#ifdef Q_QT5
        return
QStandardPaths::standardLocations((QStandardPaths::StandardLocation)type);
#else
        return QStringList()
         <<
QDesktopServices::storageLocation((QDesktopServices::StandardLocation)type);
#endif
    }
};
```

В классе `MStandardPaths` мы задаем перечисления `MStandardLocation`, совместимые с Qt 4 и с Qt 5. Создаем статический метод `MStandardPaths::standardLocation()`, из которого вызываем в зависимости от версии Qt либо метод `QStandardPaths::standardLocations()` — если это Qt 5, либо `QDesktopServices::storageLocation()` — если это Qt 4.

Резюме

Благодаря стараниям разработчиков переносимость кода программ из Qt 4 в Qt 5 не является уж очень трудоемким процессом. Но все-таки полной совместимости Qt 4 с Qt 5 нет. Поэтому, чтобы откомпилировать на Qt 5 проект, использовавший в прошлом Qt 4, придется немного потрудится.

А те, кто хочет использовать Qt 5, но также не желает расставаться с Qt 4, могут попробовать реализовать в своих проектах обратную совместимость.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/48-510/** или с помощью следующего QR-кода (рис. 48.1):

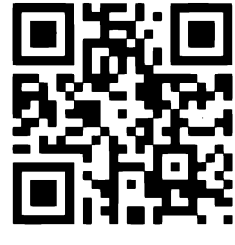


Рис. 48.1. QR-код для доступа на страницу комментариев к этой главе



ЧАСТЬ VII

Язык сценариев JavaScript

Поиск истины способен изрядно позабавить.
Вернон Говард

Глава 49. Основы поддержки сценариев JavaScript

Глава 50. Синтаксис языка сценариев

Глава 51. Встроенные объекты JavaScript

Глава 52. Классы поддержки JavaScript и практические примеры



ГЛАВА 49

Основы поддержки сценариев JavaScript

Хороший сценарий состоит в основном из диалога.
Ричард Уолтер

Как правило, при создании сложных программ не всегда можно предоставить для пользователей все необходимые им функции — кому-нибудь из них будет чего-либо не хватать. Выход из этой ситуации — реализация основных возможностей приложения на C++ и предоставление интерфейса для интерпретируемого языка сценариев, позволяющего пользователям самим реализовывать необходимые им функции. Такой подход способен удовлетворить те запросы пользователей, которые не нужны другим. Кроме того, приложения со встроенным языком сценариев упрощают предоставление поддержки для пользователей ваших программ. В таких приложениях можно реализовывать код устранения ошибок путем их обхода (workaround), а также добавлять некоторые дополнительные возможности, необходимые для какого-либо конкретного пользователя, прямо на месте и без вмешательства в исходный код основного приложения. Это свойство делает программы с поддержкой языка сценариев привлекательными и для других разработчиков, которые могут перепродавать программный продукт вместе с собственными решениями, реализовывая их в виде сценариев.

Полезное свойство использования языка сценариев заключается и в том, что для приложений, которые выполняют длительные операции, можно создать сценарий, способный автоматизировать этот процесс.

Еще одна возможность, открывающаяся с использованием в приложениях языка сценария, — это реализация гибридного приложения, часть компонентов которого реализована на C++, а другая часть — на языке сценария. При этом важно помнить о недостатке интерпретируемого языка. Этот недостаток состоит в том, что выполнение кода на нем потребует гораздо больше времени в сравнении с компилируемыми языками, такими как, например, C++. Поэтому реализовывать компоненты, которые необходимо выполнять особенно эффективно, лучше на языке C++. Но есть и преимущество — с помощью интерпретируемого языка можно легко и быстро усовершенствовать исходный код программы.

Поддержка языка сценариев в приложении — отнюдь не новая идея, она уже прошла проверку временем в таких программных продуктах, как Microsoft Office, CorelDRAW, Macromedia Flash, Emacs, Audacity и т. д.

Язык сценариев JavaScript встроен в модуль QtQml и представляет собой среду, которая обеспечивает встроенную поддержку для сценариев в приложениях, написанных на языке C++ с использованием Qt. Такая поддержка реализована в классах C++ , на которых мы более подробно остановимся в этой и *52-й главе*.

Язык сценариев JavaScript объектно-ориентирован и использует метаобъектную модель, подобную Qt. Он также обладает возможностями современных языков, такими как использование и создание классов и управление исключениями. Синтаксис этого языка подобен C++ и Java, но менее сложен. Одно из его достоинств заключается в том, что это сравнительно небольшой язык. И если вы поставляете свой продукт клиентам вместе с документацией, то это облегчит вам задачу при ее написании, поскольку информацию о JavaScript можно найти в избытке.

Благодаря тому, что поддержка языка сценариев встроена в библиотеку Qt, с ее помощью можно управлять Qt-программами без их перекомпиляции, вносить в них изменения, реализовывать тестовые сценарии и выполнять настройки приложения для специфических запросов пользователей.

Принцип взаимодействия с языком сценариев

Для реализации поддержки языка сценариев задействованы такие механизмы, как сигналы и слоты, объектные иерархии и свойства объектов (properties), с которыми хорошо знакомы все программисты, использующие библиотеку Qt. Никакого дополнительного кода не требуется, кроме того кода, который будет создан препроцессором MOC (Meta Object Compiler). Поэтому, чтобы сделать класс, унаследованный от класса `QObject`, доступным для использования в языке сценариев JavaScript, необходимо наличие метаинформации, а, значит, в определении класса для его свойств должны использоваться макросы `Q_OBJECT` и `Q_PROPERTY`, с которыми мы уже встречались в *главе 2*. В результате каждый класс и подкласс `QObject`, а также все их свойства, сигналы и слоты будут доступны из языка сценариев. Есть возможность сделать также и любой метод класса видимым для языка сценария — для этого при его объявлении нужно использовать макрос `Q_INVOKABLE` следующим образом:

```
Q_INVOKABLE void scriptAccessableMethod();
```

Несомненный плюс этого подхода заключается в том, что все унаследованные от `QObject` классы могут быть доступны для языка сценария без дополнительных усилий. Поэтому сделать доступными объекты уже существующих в Qt классов очень просто. А как сделать собственные классы, унаследованные от `QObject`, доступными для языка JavaScript, мы покажем на примере (листинг 49.1).

Листинг 49.1. Класс, доступный JavaScript

```
class MyClass : public QObject {
Q_OBJECT
Q_PROPERTY(bool readOnly WRITE setReadOnly READ isReadOnly)

private:
    bool m_bReadOnly;

public:
    MyClass(QObject* pobj = 0) : QObject(pobj)
                               , m_bReadOnly(false)
    {
    }
```

```
public slots:
    void setReadOnly(bool bReadOnly)
    {
        m_bReadOnly = bReadOnly;
        emit readOnlyStateChanged();
    }

    bool isReadOnly() const
    {
        return m_bReadOnly;
    }

signals:
   void readOnlyStateChanged();
};
```

В листинге 49.1 класс, унаследованный от класса `QObject`, управляет изменением состояния логического атрибута (`m_bReadOnly`). При создании объекта атрибут инициализируется значением `false`. В классе определены слоты для установки (`setReadOnly()`) и получения (`isReadOnly()`) значения атрибута `m_bReadOnly`. У вас наверняка возникнет вопрос: зачем мы определили эти методы как слоты? Все очень просто — если мы хотим сделать какие-либо методы класса видимыми для языка сценариев, то их необходимо определить как слоты. Но, в нашем случае, это не обязательно, так как для изменения и получения значений предусмотрено свойство `readOnly`, которое ассоциировано с этими методами. И мы могли бы определить `setReadOnly()` и `isReadOnly()` как обычные методы, тем самым скрыв их от использования в языке сценариев.

Манипулировать логическими значениями объекта класса из языка сценариев при помощи слотов можно следующим образом:

```
myObject.setReadOnly(true);
print("myObject is read only:" + myObject.isReadOnly());
```

Такая форма привычна для разработчиков на C++, но не для разработчиков на языке сценариев. Для них более привычна форма изменения значений при помощи свойств, которые наш класс также предоставляет:

```
myObject.readOnly = true;
print("myObject is read only:" + myObject.readOnly);
```

На практике обычно нужно создать сразу несколько подклассов `QObject`, которые будут использоваться для изменения состояния приложения. Поэтому модуль JavaScript дает разработчикам возможность делать выбранные ими объекты приложения, которые унаследованы от класса `QObject`, доступными для языка сценария. Это позволяет добиться динамического предоставления функциональности модулю JavaScript, и поэтому нет необходимости перекомпилировать код основной программы. Но с этим подходом связан и недостаток. Ввиду того, что мы получаем доступ из языка сценариев только к конкретным объектам приложения, невозможно получить из сценария доступ сразу ко всей функциональности библиотеки Qt или к функциональным особенностям всего приложения.

На рис. 49.1 показаны взаимосвязи между механизмом поддержки сценариев, приложением и сценариями приложения, в котором мы делаем доступными для сценариев два объекта:

Объект 3 и **Объект 4**. Сценарии получают доступ к этим объектам, как будто эти объекты встроены в сам язык JavaScript. Объекты могут быть либо объектами приложения, либо объектами, специально созданными с целью предоставления сценариям возможности простого управления приложением.

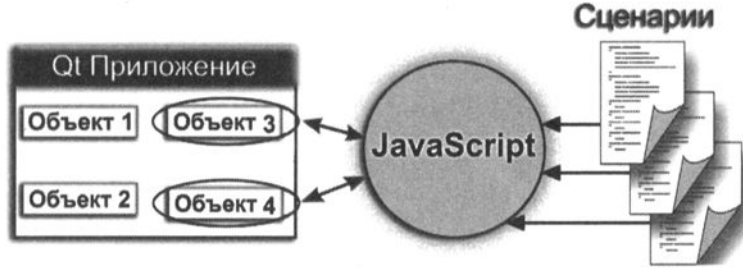


Рис. 49.1. Взаимодействие объектов приложения со сценариями

С унаследованными от `QObject` классами теперь, я думаю, все понятно, но как быть с классами, которые не унаследованы от `QObject`? Ведь может получиться так, что вы используете стороннюю библиотеку, не имеющую никакого отношения к библиотеке Qt, и хотите сделать некоторые ее классы доступными для языка сценариев. В подобных ситуациях вам придется реализовывать для каждого необходимого вам класса класс *обертки* (wrapper), унаследованный от `QObject`. Этот класс будет агрегировать объект нужного вам класса и предоставлять делегирующие методы. Предположим, что нам нужно сделать в языке сценария доступным обычный класс C++, который не использует библиотеку Qt (листинг 49.2).

Листинг 49.2. Обычный класс C++, который не использует Qt

```
class NonQtClass {
private:
    bool m_bReadOnly;

public:
    NonQtClass() : m_bReadOnly(false)
    {
    }

    void setReadOnly(bool bReadOnly)
    {
        m_bReadOnly = bReadOnly;
    }

    bool isReadOnly() const
    {
        return m_bReadOnly;
    }
};
```

Представленный в листинге 49.2 класс `NonQtClass` не наследует ни одного класса. Для того чтобы можно было использовать этот класс в языке сценариев, нам необходимо реализовать класс обертки, как показано в листинге 49.3.

Листинг 49.3. Класс обертки

```
class MyWrapper : public QObject {
Q_OBJECT
Q_PROPERTY(bool readOnly WRITE setReadOnly READ isReadOnly)

private:
    NonQtClass m_nonQtObject;

public:
    MyWrapper(QObject* pobj = 0) : QObject(pobj)
    {
    }

public slots:
    void setReadOnly(bool bReadOnly)
    {
        m_nonQtObject.setReadOnly(bReadOnly);
        emit readOnlyStateChanged();
    }

    bool isReadOnly() const
    {
        return m_nonQtObject.isReadOnly();
    }

signals:
   void readOnlyStateChanged();
};
```

Класс обертки `MyWrapper`, приведенный в листинге 49.3, агрегирует объект класса `NonQtClass` и предоставляет делегирующие слоты `setReadOnly()` и `isReadOnly()` для манипулирования его значениями. Также он определяет свойство `readOnly`, ассоциированное с этими методами. Использование класса `MyWrapper` из языка сценариев выглядит аналогично использованию класса, показанного в листинге 49.1.

Первый шаг использования сценария

Для начала удостоверимся, что все у нас работает правильно, и если кто-нибудь сомневается, то заодно проверим, действительно ли `2 * 2` равно 4? Для этого напишем самую минимальную программу с использованием модуля JavaScript (листинг 49.4). Не забудьте указать опцию для использования модуля QtQml в pro-файле:

```
QT += qml
```

Листинг 49.4. Минимальная программа использования JavaScript

```
#include <QtCore>
#include <QJSEngine>
```

```
int main(int argc, char** argv)
{
    QCoreApplication app(argc, argv);

    QJSEngine scriptEngine;
    QJSValue  value = scriptEngine.evaluate("2 * 2");
    qDebug() << value.toInt();

    return 0;
}
```

В листинге 49.4 мы создаем объект движка языка сценария `scriptEngine`. Затем в метод `evaluate()` передаем строку, которую должен выполнить интерпретатор языка. Этот метод после выполнения всегда возвращает значение результата. Само значение может быть разного типа: строкового, логического, массивом, произвольного типа `QVariant` и т. д. Принадлежность значения к конкретному типу мы можем всегда проверить вызовом методов `isT()`, где значение `T` является значением нужного нам типа. В нашем простом примере мы знаем, что значение, которое нам вернет интерпретатор, будет целого типа, но мы могли бы это проверить в листинге 49.4 следующим образом:

```
if (value.isNumber()) {
   qDebug() << "Number";
}
```

Если интерпретация закончилась неуспешно — например, в случае синтаксической ошибки, возвращенное значение будет содержать ошибку. Эту ситуацию мы могли бы отследить и аналогичным образом отобразить сообщение об ошибке:

```
if (value.isError()) {
   qDebug() << "Error:" << value.toString();
}
```

Привет, сценарий

От предоставления в программе поддержки языка сценариев любого Qt-разработчика отделяют всего лишь несколько шагов. Чтобы продемонстрировать это, возьмем простой пример, отображающий виджет надписи с текстом **Hello, JavaScript!** (рис. 49.2), и перепишем его для удобства в листинг 49.5.



Рис. 49.2. Виджет надписи

Первый шаг для предоставления поддержки сценариев заключается в добавлении следующей строки в проектный файл (это нужно, чтобы приложение собиралось с модулем QtQml):

```
QT += qml
```

Второй шаг — это включение в программу заголовочного файла QJSEngine:

```
#include <QJSEngine>
```

Возложим на язык сценариев ответственность за инициализацию виджета надписи текстом и его отображение.

Листинг 49.5. Виджет надписи, управляемый из языка сценариев

```
#include <QtWidgets>
#include <QJSEngine>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QLabel* plbl = new QLabel;

    QJSEngine scriptEngine;
    QJSValue  scriptLbl = scriptEngine.newQObject(plbl);
    scriptEngine.globalObject().setProperty("lbl", scriptLbl);
    scriptEngine.evaluate("lbl.text = 'Hello, JavaScript!'");
    scriptEngine.evaluate("lbl.show()");

    return app.exec();
}
```

В листинге 49.5 мы просто создаем виджет надписи. Далее создается объект класса `QJSEngine`, который является средой для исполнения команд языка сценариев. Из объекта этого класса мы вызываем метод `newQObject()` со ссылкой на виджет надписи `lbl`, который возвращает объект класса `QJSValue`. Этот объект является контейнером для хранения типа данных языка сценария. Поскольку добавлять свойства можно динамически, мы устанавливаем в глобальном объекте свойство нашего виджета надписи с помощью метода `QJSValue::setProperty()`.

После этого можно выполнять код сценариев на языке JavaScript, передавая его в метод `evaluate()`, что мы и делаем. При первом вызове этого метода модифицируем виджет надписи (`lbl`), присваивая его свойству `text` строку текста: `Hello, JavaScript!`. При втором — вызываем его метод `show()`.

Резюме

Сколько бы ни было реализовано функций в вашей программе, их все равно может оказаться недостаточно. Реализация приложений с поддержкой языка сценариев делает возможным динамическое расширение вашего приложения и его изменение под конкретные требования без необходимости перекомпиляции. Это позволяет разработчикам сосредоточиться на написании базовых функций и возможностях приложения.

Библиотека Qt при помощи модуля JavaScript предоставляет возможность для организации поддержки языка сценариев в ваших программах. Этот модуль содержит интерпретатор языка сценариев и классы C++ для его поддержки. Язык сценариев JavaScript базируется на ECMA Script — популярном стандарте, получившем распространение благодаря Интернету.

За счет использования метаобъектной модели Qt любой подкласс `QObject` можно сделать доступным для языка сценариев.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/49-510/** или с помощью следующего QR-кода (рис. 49.3):



Рис. 49.3. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 50

Синтаксис языка сценариев

Девушка-программист едет в трамвае, читает книгу. Старушка смотрит на девушку, смотрит на книгу, крестится и в ужасе выбегает на следующей остановке. Девушка читала книгу «Язык Ада».

Прежде чем приступить к написанию сценариев, нужно освоить их синтаксис. Если вы уже знакомы с программированием на JavaScript, то можете пропустить эту и следующую главы и перейти сразу к *главе 52*, посвященной практическим примерам.

Основное преимущество синтаксиса языка сценариев заключается в том, что он очень похож на C++ и Java. Но, в отличие от этих языков, программа языка сценария начинает выполнение не с конкретной функции, как, например, с функции `main()` в языке C и C++, а с кода, находящегося вне функции, начиная сверху.

Зарезервированные ключевые слова

В любом языке программирования присутствует своя специфика — она включает в себя и список ключевых и зарезервированных слов, составляющих ядро для программирования на этом языке. Ключевые слова всегда доступны программисту, но для их использования нужно придерживаться правильного синтаксиса. В табл. 50.1 приведен список ключевых слов JavaScript.

Таблица 50.1. *Ключевые слова JavaScript*

`break`	`case`	`catch`	`const`
`continue`	`default`	`do`	`else`
`finally`	`for`	`function`	`if`
`in`	`new`	`prototype`	`return`
`switch`	`this`	`typeof`	`throw`
`try`	`var`	`while`	`with`

Список зарезервированных слов, предназначенных для будущего использования, приведен в табл. 50.2.

Таблица 50.2. *Зарезервированные слова JavaScript*

boolean	short	interface	implements	extends
double	yield	private	long	import
float	byte	static	protected	native
int	enum	char	super	public
package	goto	export	class	throws

Ключевые и зарезервированные слова, приведенные в табл. 50.1 и 50.2, нельзя использовать в качестве идентификаторов.

Комментарии

Комментарии служат для пояснения кода сценариев. Не забывайте о том, что ваш код будут читать и изменять другие разработчики, да и вы сами по прошествии времени позабудете, что к чему, поэтому не скупитесь на них, и ваши усилия будут оправданы в будущем. Тем более, что комментарии никоим образом не влияют на эффективность исполнения вашего сценария и при интерпретации они будут просто проигнорированы. JavaScript предоставляет комментарии для одной строки (или даже ее части со знака комментария и до конца строки), а также и для целой серии строк:

```
// одна строка
/*
много строк
*/
```

Переменные

Переменная — это одна или несколько ячеек памяти компьютера, в которых хранятся определенные данные, а ее имя является своеобразной ссылкой на эти данные, при помощи которой данные можно считывать и изменять. JavaScript — слабо типизированный язык, и он не требует объявления всех переменных и определения их типов перед началом использования. Этим он отличается, например, от C++ и Java. В этом и состоит его гибкость, так как вы не обязаны объявлять переменные конкретного типа.

Переменные определяются при помощи ключевого слова `var`. Определение переменной может находиться в любом месте программы. Имя переменной должно отвечать следующим требованиям:

- имя должно начинаться со строчной или заглавной буквы или знака подчеркивания. После этого могут следовать остальные символы имени: цифры, буквы и знаки подчеркивания;
- имя не должно содержать никаких специальных символов, таких как, например: !, ?, |, < и т. д.;
- имя не должно совпадать с ключевыми словами языка (см. табл. 50.1 и 50.2);
- одно и то же имя нельзя определять в пределах одной области видимости более одного раза.

При определении переменные можно инициализировать значениями. Можно определять сразу несколько переменных с помощью одного ключевого слова `var`.

```
var x;          // Переменная без инициализации
var y = 100;    // Переменная с инициализацией
var i, j = 100; // Определение нескольких переменных
                // с инициализацией одной из них
```

Тип переменной задается при ее инициализации значением. Если переменную просто объявить и не инициализировать значением, то ее значение будет `undefined`, а тип — `Undefined`. Например:

```
var x; // имеет тип Undefined и значение undefined
```

Хотя определение переменных при помощи ключевого слова `var` и представляет собой элемент практики хорошего тона программирования, но оно не является обязательным, и можно прекрасно обойтись и без него. Например:

```
str = "Hello";
```

Имена переменных чувствительны к регистру, а это значит, например, что `x` и `X` являются двумя совершенно разными переменными.

> ***Точка с запятой***
>
> Вы наверняка заметили, что в конце каждого оператора стоит точка с запятой (`;`). На самом деле в языке сценария это совсем не обязательно, но все-таки рекомендуется их ставить — это поможет вам избежать неприятных сюрпризов.

Предопределенные типы данных

В JavaScript при объявлении задавать тип данных не нужно, так как он определяется автоматически при присвоении переменной значений и сохраняется до тех пор, пока не будет выполнено следующее присвоение. В JavaScript используются три основных типа данных: числовой (целый и вещественный), строковый и логический.

Целый тип

Целый тип представлен 32-битовыми значениями. Их диапазон лежит в пределах от −2 147 483 648 до 2 147 483 647. Значения могут задаваться в десятичной, восьмеричной и шестнадцатеричной системах счисления.

В десятичном формате задаваемое число может содержать любую последовательность цифр, но не может начинаться с нуля.

Для представления числа в шестнадцатеричном формате необходимо поставить в самом начале `0x` или `0X`. В шестнадцатеричном формате можно включать цифры в диапазоне от 0 до 9 и буквы от A до F.

Например:

```
var xDec = 123;  // Десятичная
var xHex = 0xFF; // Шестнадцатеричная
```

Вещественный тип

Вещественный тип также представлен 32 битами. Диапазон значений лежит в пределах от −1.17549435E−38 до 3.40282347E+38. Значения можно отображать в стандартной и в экспо-

ненциальной формах. В экспоненциальном представлении используются символы `e` или `E` для определения порядка числа, которое задает показатель степени. Порядок может быть как положительным, так и отрицательным. Например:

```
var fExp = 23.4524E-12; // Экспоненциальная форма записи
var f    = -13.3451;    // Стандартная форма записи
```

Строковый тип

Под этим типом понимается ряд символов в формате UNICODE. Символы строки должны заключаться в одинарные (') или в двойные (") кавычки. Например:

```
var str1 = "Hello"; // Можно так
var str2 = 'Hello'; // или так
var str2 = "80's";  // и так
```

Для форматирования текста особый интерес представляют специальные символы, приведенные в табл. 50.3. Например, используя только одинарные кавычки, мы могли бы при помощи специального символа `\'` переписать последнюю инструкцию следующим образом:

```
var str2 = '80\'s';
```

Таблица 50.3. *Специальные символы в строках*

Символ	Предназначение
`\b`	Возврат на один символ с его удалением
`\t`	Горизонтальная табуляция
`\n`	Новая строка
`\v`	Вертикальная табуляция
`\r`	Возврат каретки
`\"`	Двойная кавычка
`\'`	Одинарная кавычка
`\\`	Обратная косая черта

Логический тип

Переменные логического (или булевого) типа принимают только два значения: `true` (истина) и `false` (ложь). Как правило, этот тип используется для определения выполнения заданного условия:

```
var b = true;
b = (3 == 5); //=> b=false
```

Преобразование типов

Тип любой переменной можно узнать в процессе выполнения сценария при помощи оператора `typeof()`. Например, для четырех переменных `n`, `f`, `b`, `str` разного типа и одной переменной `unknown`, которой не присвоено какого-либо значения, это делается так:

```
var n       = 7;
var f       = 5.2018;
```

```
var b      = true
var str    = "Hello";
var unknown;
print(typeof(n)); //=> number
print(typeof(f)); //=> number
print(typeof(b)); //=> boolean
print(typeof(str)); //=> string
print(typeof(unknown)); //=> undefined
```

После того как переменной было присвоено значение, в соответствии со значением переменной ей присваивается тип. Мы видим, что в приведенном примере для переменной `unknown`, которую мы не инициализировали значением, оператор `typeof()` вернул значение `undefined`. Это значит, что для нее тип еще не задан. Тип переменной можно изменить в любой момент времени, присвоив переменной значение другого типа. Например:

```
var v = "Hello"; // переменная строкового типа
v = 32.3;        // стала переменной вещественного типа
```

Кроме того, часто тип переменной можно изменить, практически не меняя присвоенное ей значение. Различают *неявные* и *явные преобразования типа*. В первом случае, если переменная является результатом действий над переменными разных типов, то ее тип будет задан той переменной, тип которой позволяет представить полученный результат без искажений. Например, результатом перемножения значений целого и вещественного типа будет вещественное значение:

```
var n = 56;      // Целое значение
var f = 0.27;    // Вещественное значение
var res = n * f; // Результат является вещественным: 15.12
```

Можно самому влиять на процесс конвертирования путем явного преобразования типа. Это достигается при помощи функций `parseFloat()` и `parseInt()`, предназначенных для преобразований строковых значений в числа. Второй функцией можно преобразовывать и вещественные значения к целым, например:

```
var f = 3.5;
var n = parseInt(f);
print(n); //=> 3
```

При этом выполняется неявное преобразование переменной `f` к строке `3.5`, после чего из строки будет прочитано целое значение. Поскольку точка не может входить в целое число, она и все, что за ней следует, окажется проигнорировано. В результате произойдет «урезание» значения, и на экране будет отображено число 3.

С численными значениями все понятно, но что происходит, если сложить число со строкой?

```
var n = 7, f = 5.2018, b = true;
var res1 = n + " is a number";
var res2 = f + " is a float number";
var res3 = b + " is a boolean value";
print(res1); //=> 7 is a number
print(res2); //=> 5.2018 is a float number
print(res3); //=> true is a boolean value
```

Во всех трех случаях результат будет преобразован к строковому.

Разумеется, оператор + позволяет складывать и две строки, соединяя их в одну.

Операции

Самой важной синтаксической частью языка являются операции, с помощью которых можно сравнивать, изменять и получать доступ к данным.

Операторы присваивания

С этим оператором нам уже пришлось столкнуться в самом начале главы. С его помощью мы инициализировали переменные, присваивая им определенные значения. Присвоения переменным значений и есть основная функция этого оператора. Например: выражение `x = 13` присваивает переменной `x` значение `13`.

Арифметические операции

Для вычисления численных значений используются арифметические операции. Эти операции делятся на две группы: *бинарные* (двуместные) и *унарные* (одноместные). Первая группа состоит из следующих операций:

- *сложения* — операции, обозначаемой знаком плюс (+). Пример: `x + y`;
- *вычитания* — операции, обозначаемой знаком минус (-). Пример: `x - y`;
- *умножения* — операции, обозначаемой знаком звездочка (*). Пример: `x * y`;
- *деления* — операции, обозначаемой обратной косой чертой (/). Пример: `x / y`;
- *остатка от деления* — операции, обозначаемой знаком процента (%). Пример: `x % y`.

Приведенные здесь операции можно, в целях сокращения записи, комбинировать с оператором присваивания. В табл. 50.4 указаны примеры этих комбинаций.

Таблица 50.4. *Примеры комбинирования операций*

Операция	Описание
`x += y`	Сложение с присваиванием. Сокращенная запись для `x = x + y`
`x -= y`	Вычитание с присваиванием. Сокращенная запись для `x = x - y`
`x *= y`	Умножение с присваиванием. Сокращенная запись для `x = x * y`
`x /= y`	Деление с присваиванием. Сокращенная запись для `x = x / y`
`x %= y`	Остаток деления с присваиванием. Сокращенная запись для `x = x % y`

Вторую группу — унарных арифметических операций — составляют два оператора: *инкремента* (++) и *декремента* (--).

Первый оператор увеличивает переменную на единицу, а второй уменьшает ее на единицу. Таким образом, `++i` — это то же самое, что и `i = i + 1`, а `--i` — то же, что и `i = i - 1`.

Операторы инкремента и декремента допускается использовать в префиксной и постфиксной форме. Это изменяет порядок возврата значения, что важно при его дальнейшем использовании. Например, рассмотрим сначала *префиксную* форму записи:

```
var i = 0;
var a = ++i;
```

В этом случае результат переменной i будет сначала увеличен на единицу, а затем присвоен переменной a. В результате переменной a будет присвоено значение 1. Теперь давайте рассмотрим случай с *постфиксной* формой записи.

```
var i = 0;
var a = i++;
```

Здесь значение переменной i сначала присваивается переменной a, и только после этого значение переменной i увеличивается на единицу. Поэтому переменной a будет присвоено значение 0.

Поразрядные операции

Поразрядные операции используются для действий над целыми десятичными значениями как с последовательностью двоичных разрядов. По своей сути, эти операции сходны с обычными логическими операциями, только применяются к битам. Обычно эти операции служат для оптимизации кода программ. Они были унаследованы от языка C. Но при программировании на языке сценариев задачи оптимизации не стоят так остро, поэтому эти операторы используются крайне редко.

К ним относятся:

- операция поразрядного логического И (&);
- операция поразрядного логического ИЛИ (|);
- операция поразрядного исключающего ИЛИ (^);
- операция сдвига битов влево <<;
- операция сдвига битов вправо >>;
- операция сдвига битов вправо с заполнением нулями >>>.

Приведем несколько примеров:

```
var v1 = 5;  //0101
var v2 = 15; //1111
var v3 = 0;  //0000
print(v1 & v2);  //=>5 (0101)
print(v1 & v3);  //=>0 (0000)
print(v1 ^ v2);  //=>10 (1010)
print(v1 << 1);  //=>10 (1010)
print(v1 >>> 1); //=>2 (0010)
```

Эти операции можно комбинировать с оператором присваивания.

Операции сравнения

Как понятно из названия, эти операции применяются для сравнения значений выражений. Результат операций сравнения может иметь только два значения: true (истинное) и false (ложное). В табл. 50.5 приведен список всех операций сравнения.

Давайте рассмотрим несколько примеров:

```
true==1          // => true (true преобразуется в 1)
true===1         // => false (true не преобразуется в 1)
```

```
5 === "5"          // => false (строка "5" не преобразуется в число)
5 == "5"           // => true ("5" преобразуется в число)
5 > 4              // => true (5 больше 4)
null == undefined  // => true (null преобразуется к undefined)
null === undefined // => false (null не преобразуется к undefined)
```

Таблица 50.5. *Операции сравнения*

Операция	Описание
`a == b`	*Проверка равенства*. Возвращает значение `true`, если операнды между собой равны. В противном случае — `false`
`a != b`	*Проверка неравенства*. Возвращает значение `true`, если операнды не равны между собой. В противном случае — `false`
`a === b`	*Проверка на совпадение* (идентичность). Иногда называется операцией *жесткого равенства*. При проведении этой операции преобразование типов операндов не выполняется. Если значение переменной `a` равно значению переменной `b` и их типы совпадают, то возвращается значение `true`. В противном случае — `false`
`a !== b`	*Проверка на несовпадение*. Иногда называется операцией *жесткого неравенства*. При проведении этой операции преобразование типов операндов не выполняется. Если значение переменной `a` не равно значению `b`, то возвращается значение `true`. В противном случае — `false`
`a < b`	*Меньше*. Возвращает значение `true`, если переменная `a` меньше `b`. В противном случае — `false`
`a <= b`	*Меньше либо равно*. Возвращает значение `true`, если переменная `a` меньше или равна `b`. В противном случае — `false`
`a > b`	*Больше*. Возвращает значение `true`, если переменная `a` больше `b`. В противном случае — `false`
`a >= b`	*Больше либо равно*. Возвращает `true`, если переменная `a` больше или равна `b`. В противном случае — `false`

Приоритет выполнения операций

Каждая операция имеет свой приоритет выполнения, а это значит, что если в одном выражении задано несколько операторов, то сначала будут исполнены действия операторов, обладающих более высоким приоритетом. Поэтому важно знать, каким уровнем приоритета обладает та или иная операция, — иначе можно запутаться с получаемым значением. Вот конкретный пример — попробуйте угадать значение переменной `n`:

```
var i = 8;
var n = 11 + ++i * 9 % 5;
```

Воспользовавшись шкалой, показанной на рис. 50.1, мы можем вычислить результат. Итак, самая старшая в выражении — это операция инкрементации (++). Значит, сначала переменная будет увеличена на единицу и станет равна 9. Наш промежуточный результат выглядит следующим образом:

```
var n = 11 + 9 * 9 % 5;
```

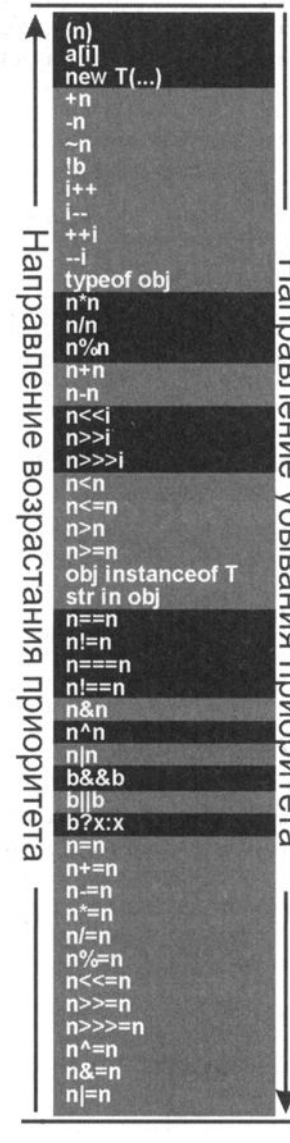


Рис. 50.1. Диаграмма старшинства приоритетов операций

Операции (`*`) и (`%`) имеют одинаковый приоритет, поэтому эти действия будут выполнены слева направо. Таким образом, сначала будет выполнять операция умножения 9 * 9. Промежуточный результат:

```
var n = 11 + 81 % 5;
```

А теперь будет вычислен остаток от деления 81 на 5 (`%`). Наш промежуточный результат:

```
var n = 11 + 1;
```

Значит, в итоге переменная `n` будет равна 12.

Управляющие структуры

Посредством управляющих структур можно управлять ходом выполнения действий программы.

Условные операторы

Условные операторы являются наиважнейшей структурой языка. Именно благодаря тому, что с их помощью в зависимости от условий можно направить выполнение программы по различным путям, удастся реализовать любую логику ваших программ. В качестве условий выступают любые логические выражения.

Логические выражения могут объединяться при помощи логических операций:

- `||` — логическое ИЛИ;
- `&&` — логическое И,

а также сводиться к противоположным логическим значениям при помощи оператора логического отрицания `!`.

Оператор *if ... else*

Этот оператор относится к числу наиболее часто используемых. Он направляет выполнение программы по определенному пути в зависимости от заданного в круглых скобках условного выражения. Его общий синтаксис выглядит следующим образом:

```
if (условное выражение) действие
[else действие]
```

Как вы заметили, применение `else` не является обязательным, но позволяет выполнить какие-то действия в том случае, если условное выражение возвратит значение `false`. Ключевое слово `else` не является самостоятельной частью языка и может использоваться только с `if`. Например:

```
var a = 3;
var b = 4;
if (a > b)
    print("a больше b");
else
    print("a меньше либо равно b");
```

Если действий много, то они должны быть заключены в фигурные скобки `{}`. Условные операторы `if` можно вкладывать друг в друга и комбинировать `else` с другим оператором `if`. Например:

```
if (a > b) {
// выполнить действия
}
else if(a == b) {
// выполнить действия
}
else {
// выполнить действия
}
```

Оператор *switch*

В качестве альтернативы оператору `if...else` можно воспользоваться оператором `switch`. Особенно эффективен этот оператор для проверки целого ряда значений. Общий синтаксис оператора `switch` выглядит следующим образом:

```
switch (выражение) {
case пункт1:
    действия
    [break]
case пункт2:
    действия
    [break]
...
[default: действия]
}
```

В отличие от C/C++, значения пунктов этого оператора не ограничены только целым типом. Его исполнение начинается с вычисления значения выражения в круглых скобках. Затем это значение сравнивается с пунктами, стоящими после ключевого слова `case`. В случае совпадения выполняются действия этого пункта. Если ни одного совпадения с пунктами `case` не найдено, то, в случае наличия секции `default`, выполняются ее действия. Если действия пункта завершаются ключевым словом `break`, то оператор `switch` после выполнения действий пункта будет покинут. Если ключевое слово `break` не указано, то выполняются действия следующего пункта и т. д., пока не встретится слово `break`, последний пункт `case` или секция `default`. В следующем примере переменной `value` присваивается значение, зависящее от значения переменной `num`.

```
var initialValue = 0;
switch (num) {
case 1:
case "one":
    value = "1";
    break;
case initialValue:
    value = "0";
    break;
default:
    value = "unknown";
}
```

Оператор условного выражения

Тернарный (трехместный) оператор условного выражения `? :` в качестве краткой формы записи оператора `if` может быть применен в тех случаях, когда не нужно изменять ход выполнения действий программы. Он возвращает, в зависимости от условия, одно из двух указанных альтернативных значений. В качестве простой демонстрации реализуем тот же самый пример, который был приведен для оператора `if`:

```
var a = 3;
var b = 4;
var str = (a > b) ? "a больше b" : "a меньше либо равно b";
print(str);
```

Тернарные операторы условных выражений можно вкладывать друг в друга. Это может быть полезно, например, для того, чтобы предотвратить выход какого-либо значения за заданный диапазон. Например, зададим диапазон допустимых значений от 0 до 100:

```
n = n > 100 ? 100 : n < 0 ? 0 : n;
```

Циклы

Циклы — это конструкции для выполнения части кода несколько раз.

Операторы *break* и *continue*

Прерывание цикла происходит при наступлении какого-либо условия, но иногда может потребоваться досрочно покинуть цикл. Это достигается при помощи ключевого слова `break`, помещенного внутри цикла.

Ключевое слово `continue` поступает похожим образом, только не осуществляет выход из цикла, а пропускает оставшиеся операторы текущей итерации и переходит к следующей итерации.

Цикл *for*

Это самый популярный цикл при программировании на C/C++, Java и JavaScript. Его популярность обоснована гибкостью, позволяющей выполнять дополнительные действия. Цикл `for` состоит из трех основных частей: инициализации, условного выражения и секции для изменения переменных. Его общий синтаксис выглядит следующим образом:

```
for (инициализация; условное выражение; изменения переменных) {
действия
}
```

При первом вызове цикла выполняется инициализация. До тех пор, пока условное выражение имеет значение `true`, цикл будет повторять действия в фигурных скобках, а затем проводить изменения переменных и снова проверять условное выражение. Рассмотрим этот цикл на простом примере подсчета суммы:

```
var sum = 0;
for (var i = 1; i <= 10; ++i) {
    sum = sum + i;
}
print(sum); //=> 55
```

При старте цикла выполняется определение переменной `i` и ее инициализация значением 1. Затем значение переменной `i` сравнивается со значением 10. Если оно превысит 10, работа цикла будет завершена. Иначе значение переменной `i` при каждой итерации цикла и при помощи оператора инкремента будет увеличиваться и прибавляться к переменной `sum`.

Цикл *while*

Оператор `while` действует подобно циклу `for`, но не включает в себя функций, отвечающих за инициализацию и изменение переменных. В остальном эти циклы идентичны. На практике не имеет значения, используете вы оператор `for` или `while`. Эти циклы всегда можно заменить один на другой. В общем виде цикл `while` выглядит следующим образом:

```
while (условное выражение) {
Действия
}
```

Давайте заменим цикл `for` в нашем примере подсчета суммы на цикл `while`:

```
var sum = 0;
var i   = 1;
while (i <= 10) {
    sum = sum + i;
    ++i;
}
```

Цикл *do...while*

Цикл `do...while` представляет собой разновидность цикла `while`. Разница с циклом `while` заключается в том, что перед проверкой условия тело цикла выполняется один раз. В общем виде этот цикл выглядит следующим образом:

```
do {
Действия
} while (условное выражение);
```

Наш пример подсчета суммы с использованием этого цикла будет выглядеть так:

```
var sum = 0;
var i   = 1;
```

```
do {
    sum = sum + i;
    ++i;
} while (i <= 10);
```

Оператор *with*

Оператор `with` задуман для ухода от многократного повторения ссылок на объект при доступе к его свойствам и методам. Например, вместо записи:

```
print(Math.PI);
print(Math.abs(-2));
print(Math.max(4, 10, 7, 6));
```

можно прибегнуть к эквивалентной записи с использованием оператора `with`:

```
with (Math) {
    print(PI);
    print(abs(-2));
    print(max(4, 10, 7, 6));
}
```

Исключительные ситуации

Исключительные ситуации могут возникнуть в тех случаях, когда какие-либо действия алгоритма программы не могут быть выполнены корректно. Например, произошло обращение по несуществующему индексу массива, файл, открываемый для чтения, не существует и т. д. Для того чтобы ваша программа в подобных случаях не потеряла «равновесия» и продолжала работать дальше с учетом этих ситуаций, необходимы конструкции, позволяющие перехватывать, анализировать и обрабатывать подобные ситуации.

Оператор *try...catch*

Этот оператор служит для отделения действий, в которых могут быть незапланированные ошибки, с последующей возможностью перехвата сгенерированных исключений, которые выполняются в секции `catch`. В общем виде оператор `try...catch` выглядит следующим образом:

```
try {
сомнительные действия
}
catch (ошибка) {
действия обработки ошибки
}
final {
завершающие действия
}
```

Итак, в теле `try` находятся некоторые «сомнительные» действия, выполнение которых может привести к ошибке. В случае ее возникновения будет осуществлен переход в секцию `catch` для ее дальнейшей обработки. В секции `final` находятся действия, которые должны быть выполнены независимо от того, возникло исключение или нет. Давайте рассмотрим пример перехвата и обработку исключения на примере открытия файла. Предположим, что

мы располагаем специальным классом File. В нашем случае мы просто отображаем код ошибки и закрываем файл.

```
var file = new File("file.dat");
try {
    file.open(File.ReadOnly);
}
catch(e) {
    print("Code error:" + e);
}
finally {
    file.close();
}
```

Оператор *throw*

При помощи оператора throw можно генерировать свои собственные исключения для их последующего перехвата в операторе try...catch. Исключение может быть числом, строкой или даже объектом собственного класса. Выброс исключения в общем виде выглядит следующим образом:

```
throw error;
```

Функции

Функции — это важный инструмент для разделения задач на подзадачи. В подавляющем большинстве функция представляет собой блок действий над полученными исходными данными с последующим возвращением результата. Объявление функции осуществляется при помощи ключевого слова function. Функции можно определять в любом месте программы. В общем виде, синтаксис объявления функций выглядит следующим образом:

```
function имя ([аргумент1][..., аргументN])
{
    [действия]
}
```

В качестве примера реализуем функцию, перемножающую два числа и возвращающую полученный результат:

```
function multiply(var1, var2)
{
    return var1 * var2;
}
```

Вызов функции осуществляется посредством указания ее имени. При вызове функции, если она это допускает, можно передать ей аргументы. Выполним вызов нашей функции перемножения двух чисел и сохраним возвращенный результат в переменной:

```
var f = multiply(2.3, 13.7);
```

Ключевое слово function позволяет создавать функции «на лету», как только в этом появится необходимость:

```
var myMultiplyFunction =
    function multiply(var1, var2){ return var1 * var2;};
var f = myMultiplyFunction(2.3, 13.7);
```

Кроме того, при помощи специального объекта `Function` можно создавать и изменять функции в процессе выполнения самой программы. Такой подход дает очень большое преимущество в силу предоставляемой им гибкости. Благодаря ему можно обеспечить пользователю возможность задавать действия функции самому, набрав ее в текстовом поле программы. Следующий пример демонстрирует создание подобной функции:

```
var myMultiplyFunction =
    new Function("var1", "var2", "return var1 * var2;");
var f = myMultiplyFunction(2.3, 13.7);
```

Количество и значения аргументов, переданных функции, можно получить в самой функции при помощи встроенной в язык переменной `arguments`. Эта переменная называется *объектом активизации функции* и инициализируется всякий раз при вызове функции. Таким образом можно определить функцию, скажем, для перемножения любого количества переданных в нее значений:

```
function multiply()
{
    var result = 1;
    for (var i = 0; i < arguments.length; ++i) {
        result *= arguments[i];
    }

    return result;
}
```

Теперь нашу функцию можно вызывать с любым количеством аргументов:

```
var f = multiply(34.5, 14.2, 8.7, 3.4, 7.1);
```

Функции могут быть вызваны рекурсивно. Классический пример тому — функция вычисления факториала:

```
function factorial(n)
{
    if ((n == 0) || (n == 1)) {
        return 1;
    }
    else {
        result = (n * factorial(n - 1));
    }
    return result;
}
print("Factorial_10=" + factorial(10)); //=> "Factorial_10=3628800"
```

Есть еще небольшой нюанс, связанный с использованием ключевого слова `var`, который необходимо учитывать. Все дело в том, что если мы внутри объявим переменную посредством `var`, то она будет являться *локальной переменной*, то есть по завершению работы функции будет разрушена. Но если мы просто используем переменную, не определяя ее, то она станет *глобальной* и после завершения работы функции продолжит существовать. Например:

```
function foo()
{
   m = 2;
}

foo();
print(m); //=>2
```

Если бы в этом примере переменная `m` была бы определена с помощью ключевого слова `var`, то это привело бы к исключительной ситуации, так как тогда мы бы пытались получить доступ к несуществующей переменной.

Встроенные функции

JavaScript предоставляет ряд функций, определенных в глобальном объекте и являющихся неотъемлемой частью самого языка. К таким функциям относятся:

- `eval()` — выполняет содержимое строки, понимаемое как код JavaScript. Это очень интересный метод. Он может использоваться, например, для того, чтобы пользователь в процессе работы самой программы сценария вводил целые программные фрагменты на JavaScript для их последующего выполнения. Например: `eval("for (var i = 0; i < 10; ++i) {print(i);}");`
- `isNaN()` — возвращает значение `true`, если переданное выражение не является числовым значением;
- `parseFloat()` — преобразует переданную строку к вещественному типу. В случае неудачи функция возвращает значение `NaN`;
- `parseInt()` — преобразует переданную строку к целому типу. В случае неудачи функция возвращает значение `NaN`.

Объектная ориентация

JavaScript является объектно-ориентированным языком. Класс в JavaScript — это своего рода шаблон для создания объектов. Классы определяются при помощи функций, которые являются конструкторами. Для создания объекта имени такой функции должен предшествовать оператор `new`. Конечно же, можно и просто вызвать функцию конструктора, но это не приведет к созданию объекта, и поэтому в подобном вызове смысла не будет.

Функция `Point()` в листинге 50.1 — это функция конструктора, в которую передаются два аргумента, — координаты точки: `x` и `y`. В конструкторе мы определяем два атрибута: `m_x` и `m_y`, предназначенные для хранения координат. Далее, при помощи ключевого слова `this` мы также задаем имена методов, предназначенных для изменения (`setX()` и `setY()`) и получения (`x()` и `y()`) значений координат. Ключевое слово `this` относится к созданному объекту и является ссылкой на него.

Листинг 50.1. Определение класса при помощи функций

```
function Point(x, y)
{
    this.m_x = x;
    this.m_y = y;
```

```
    this.setX = function(x)
    {
        this.m_x = x;
    }

    this.x = function()
    {
        return this.m_x;
    }
}

Point.prototype.setY = function(y)
{
   this.m_y = y;
}

Point.prototype.y = function()
{
    return this.m_y;
}
```

> ***Определение методов класса***
>
> Методы класса можно определять как внутри тела функции конструктора, так и за ее пределами. Для демонстрации этого в листинге 50.1 мы специально определили методы получения и установки атрибута `m_x` в теле функции конструктора, а методы атрибута `m_y` — при помощи ключевого слова `prototype` — за ее пределами.

Создание объектов при помощи функции `Point()` может выглядеть следующим образом:

```
var pt = new Point(20, 30);
print("X=" + pt.x() + ";Y=" + pt.y());
```

На экране будет отображено:

```
X=20;Y=30
```

Значение параметров по умолчанию, которое очень часто используется в C++, в JavaScript можно задать при помощи оператора `||`. Возьмем для примера конструктор листинга 50.1 и внесем в его начале следующие изменения:

```
this.m_x = x || 0;
this.m_y = y || 0;
```

Теперь, если мы вызовем наш конструктор без параметров:

```
var pt = new Point;,
```

его переменные члены `x` и `y` будут равны 0. Или, если мы укажем только первый параметр:

```
var pt = new Point(3);,
```

то `x` будет равен 3, а `y` будет равен 0. Абсолютно так же можно было поступить и с переменными не только целого типа, но и других, — например, строкового:

```
this.m_str = str || "";.
```

Для закрытия членов класса специального ключевого слова — как, например, `private` в C++, нет. Но можно поступить следующим образом: просто закрыть их в пределах функции конструктора при помощи ключевого слова `var` (листинг 50.2).

Листинг 50.2. Закрытие членов класса

```
function Point(x, y) {
    this.m_x = x || 0;
    this.m_y = y || 0;
    var privateVariable = 8;
    var privateMethod = function() {
        printOut("private variable value:" + privateVariable);
    }
    this.setX = function(x)
    {
        privateMethod();
        this.m_x = x;
    }
...
}
```

В листинге 50.2 члены класса: переменную `privateVariable` и метод `privateMethod()` — мы делаем закрытыми. Теперь, если мы попытаемся из созданного объекта `pt` обратиться к закрытому методу, то получим следующую ошибку:

```
Result of expression 'pt.privateMethod' [undefined] is not a function.
```

Но внутри нашего класса мы можем обращаться к ним из любого метода. В листинге 50.2, например, из метода `setX()` мы осуществляем вызов закрытого метода и отображаем значение закрытой переменной.

Есть также возможность для создания так называемых *буквальных объектов* без функции конструктора. Вот как мы могли бы обойтись без использования класса, приведенного в листинге 50.1, и создать объект точки буквально «на лету» (листинг 50.3).

Листинг 50.3. Создание буквального объекта

```
var myPoint = {
    m_x: 123,
    m_y: 321,
    x: function() {
        return myPoint.m_x;
    },
    y: function() {
        return myPoint.m_y;
    }
}
print("X=" + myPoint.x() + "; Y=" + myPoint.y());
```

На экране будет отображено:

```
X=123;Y=321
```

Статические классы

Иногда нужно сделать так, чтобы из программы можно было получать доступ к методам и переменным, не создавая самого объекта класса. В C++ для этого есть ключевое слово `static`, с помощью которого мы можем обозначить все нужные методы и переменные класса. Чтобы реализовать подобное в JavaScript, нужно сделать подобие статического класса при помощи только что рассмотренного (см. листинг 50.3) буквального объекта. Пример показан в листинге 50.4.

Листинг 50.4. Статический класс

```
Error = {
    nr1: 'Can not read',
    nr2: 'Can not write',
    message: function() {
        print('An error is occurred');
    }
}
Error.message();
print(Error.nr1);
```

Вывод на экран будет следующим:

```
An error is occurred
Can not read
```

Наследование

Для того чтобы унаследовать существующий класс, нам понадобится создать функцию конструктора нового класса и из него при помощи `call()` запустить конструктор наследуемого класса. Покажем это на примере создания класса трехмерной точки `ThreeDPoint`. Другими словами, расширим класс `Point` еще одной координатой `z` (листинг 50.5).

Листинг 50.5. Наследование класса `Point`

```
function ThreeDPoint(x, y, z)
{
    Point.call(this, x, y);
    this.m_z = z;

    this.setZ = function(z)
    {
        this.m_z = z;
    }

    this.z = function()
    {
        return this.m_z;
    }
}
```

Здесь наш класс принимает три аргумента: x, y, z. Два первых вместе с указателем this передаются в call() класса Point. Далее мы определяем атрибут m_z и инициализируем его в соответствии с переданным в конструктор значением, а так же реализуем два метода для установки и получения его значения: setZ() и z(). Создание объекта нашего нового класса выглядит следующим образом:

```
var pt = new ThreeDPoint(20, 30, 40);
print("X=" + pt.x() + ";Y=" + pt.y() + ";Z=" + pt.z());
```

На экране будет отображено:

```
X=20;Y=30;Z=40.
```

После того как объект создан, при необходимости можно добавлять к нему новые методы и переопределять уже существующие. Покажем это на следующем примере: добавим к только что созданному нами объекту pt метод test() и переопределим метод x(), унаследованный от класса Point:

```
pt.test = function()
{
    return "Test";
}

pt.x = function()
{
    return -1;
}

print("X=" + pt.x() + "; " + pt.test());
```

На экране вы увидите:

```
X=-1; Test
```

Наследование в JavaScript может быть также реализовано при помощи прототипов (листинг 50.6).

Листинг 50.6. Наследование класса Point при помощи прототипов

```
function Point(x, y)
{
    this.m_x = x;
    this.m_y = y;
}
function ThreeDPoint(x, y, z)
{
    this.base = Point;
    this.base(x, y);
    this.m_z = z;
}
ThreeDPoint.prototype = new Point;
var pt = new ThreeDPoint(1, 2, 3);
print("X=" + pt.m_x + ";Y=" + pt.m_y + ";Z=" + pt.m_z);
```

В листинге 50.6 мы расширяем уже существующий объект `Point` новой переменной — членом координаты `z`. Вывод на экране будет следующим:

```
X=1;Y=2;Z=3.
```

При желании добавленные переменные можно и убирать из объектов — например, если нам переменная `m_z` в объекте `pt` была бы нежелательна, то мы могли бы ее удалить из него следующим образом:

```
delete pt.m_z;
```

Для того чтобы проверить, содержится член в классе или нет, нужно вызвать из него метод `hasOwnProperty()`, например:

```
pt.hasOwnProperty('m_z'); //=>true
```

От какого класса был осуществлен объект, можно узнать при помощи оператора `instaceof` следующим образом:

```
pt instanceof ThreeDPoint; //=> true
pt instanceof Point; //=> true
pt instanceof Date; //=> false
```

Удостовериться в том, что перед нами объект, а не переменная, можно с помощью оператора `typeof`, который возвращает строки с обозначением типа:

```
typeof pt; //=> "object"
typeof "text"; //=> "string"
```

При помощи прототипов можно расширять возможности и уже существующих объектов — например, добавляя метод в стандартный объект `Date` для отображения года даты (листинг 50.7).

Листинг 50.7. Расширение стандартного объекта даты новым методом

```
Date.prototype.printFullYear = function() {
    print(this.getFullYear());
}
var dt = new Date();
dt.printFullYear();
```

В листинге 50.7 при помощи ключевого слова `prototype` мы дополняем стандартный объект новым методом `printFullYear()`. Создаем сам объект и вызываем из него новый метод. В выводе на экран будет показан текущий 2017 год.

При помощи метода `defineProperty()` глобального объекта `Object` можно реализовать полезную функцию, которая будет добавлять любые функции в качестве методов к объектам. В листинге 50.8 показан пример такой реализации, в которой функция `addMethodToObject()` принимает два аргумента. Первый аргумент — это функция, которую нужно добавить к объекту, второй аргумент — это сам объект, к которому будет добавлена функция.

Листинг 50.8. Расширение стандартных объектов при помощи функции

```
function addMethodToObject(fct, obj)
{
    Object.defineProperty(obj,
                          fct.name,
```

```
                        {value: fct,
                         enumerable: false
                        }
                       );
}
```

Пример использования функции, приведенной в листинге 50.8, может выглядеть следующим образом: мы создадим функцию `at()`, которая будет возвращать значение по заданному индексу, добавим ее к стандартным объектам `Array` и `String` и протестируем (листинг 50.9).

Листинг 50.9. Пример использования функции, приведенной в листинге 50.8

```
function at(i)
{
    return this[i];
}
addMethodToObject(at, Array.prototype);
addMethodToObject(at, String.prototype);

var  arr = new Array(4, 5, 6, 7);
var str = new String("TEST");
var n = arr.at(2); //=> n=6
var ch = str.at(2); //=> ch="S"
```

Перегрузка методов

Для того чтобы изменить поведение уже существующих объектов, можно воспользоваться перегрузкой методов. Например, если вас не устраивает какой-то метод объекта, то вы можете написать свою реализацию и выполнить замену существующего метода (листинг 50.10).

Листинг 50.10. Перегрузка метода

```
function Point(x, y)
{
    this.m_x = x;
    this.m_y = y;
    this.x = function()
    {
        return this.m_x;
    }
}

var pt = new Point(1, 2);
print("X=" + pt.x());

function myX()
{
    return 1234;
}
```

```
pt.x = myX;
print("X=" + pt.x());
```

В листинге 50.10 мы создаем объект нашего класса `Point` и вызываем метод `x()` для отображения его значения. Затем реализуем свою функцию `myX()` и при помощи присвоения выполняем перегрузку существующего метода `x()`. Теперь вызов того же метода `x()` будет нам всегда возвращать значение 1234.

Точно таким же образом мы можем перегрузить методы и любого стандартного объекта JavaScript и тем самым по своему усмотрению адаптировать саму среду использования языка.

Сказание о «джейсоне»

История об объектной ориентации в JavaScript не была бы полной, если бы не рассказать немного о «джейсоне» (JSON, JavaScript Object Notation). Вообще говоря, JSON — это текстовый формат для обмена данными, который позволяет конвертировать объекты JavaScript и массивы в строки. Сам принцип похож на XML, но в силу своей компактности JSON является более предпочтительным. Еще очень важен тот факт, что эти JSON-строки являются исходным кодом JavaScript, следовательно, получив такую строку в JavaScript-программе, ее можно сразу же, без каких-либо предварительных преобразований, исполнить как программный код. Это делает JSON более удобным в использовании, чем XML. Несмотря на то, что в его аббревиатуру первой буквой включен JavaScript, формат, в силу своих преимуществ, стал настолько популярным, что спектр его использования вышел далеко за рамки только одного языка JavaScript. В листинге 50.11 показан пример данных в формате JSON и взаимодействие с ними.

Листинг 50.11. Использование данных объекта в формате JSON

```
var json = ({
    "name": "Piggy",
    "phone": "+49 631322187",
    "email": "piggy@mega.de",
    "Details": {
        "age": 47,
        "lover": "Kermit",
        "male": false,
        "hobbies": ["Singing", "Dancing"],
        "car": null
    }
});
var jsonObj        = eval(json);
var jsonDetailsObj = jsonObj.Details;
print(jsonObj.name + " loves " + jsonDetailsObj.lover);
```

В листинге 50.11 мы используем уже знакомую нам структуру данных нашей адресной книги (см. *главы 40 и 41*). Переменная `json` содержит данные описания, которые должны быть исполнены из самой программы при помощи функции `eval()`. Объекты — это заключенные в скобки `{}` данные. Поэтому, после выполнения функции `eval()`, мы получим объект, в который встроен еще один объект `Details`. Переменные члены этих объектов задаются

следующим образом. Слева от : стоит имя переменной, а справа — ее значение. Значения могут быть строкой (в примере "name"), числом (в примере "age"), логической переменной (в примере "male"), массивом (в примере "hobbies"), а также быть null (в примере "car"). Доступ к атрибутам объекта осуществляется по привычной схеме с указанием имени объекта и нужной переменной члена. В листинге 50.11 мы вызываем на экран переменную name и переменную внутреннего объекта lover. Вывод на экран будет следующим:

```
Piggy loves Kermit
```

Этот объект мы также можем расширить нужными нам методами. Например, добавим в листинг 50.10 метод для возвращения значения переменной внутреннего объекта (переменной age):

```
jsonObj.getAge = function()
{
    return jsonObj.Details.age;
}
printOut(jsonObj.getAge());
```

На экране будет отображено число 47.

Резюме

Язык JavaScript очень похож на C++ и Java. Он является полноценным языком программирования и предоставляет все конструкции, необходимые для написания программ: от объявления переменных до объектной ориентации.

Переданная в JSON-строках информация является исходным кодом на JavaScript. Она гораздо компактнее XML и ее можно использовать принимающей стороной в JavaScript-сценариях без дополнительных преобразований.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/50-510/** или с помощью следующего QR-кода (рис. 50.2):



Рис. 50.2. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 51

Встроенные объекты JavaScript

> В любой науке, в любом искусстве лучший учитель — опыт.
>
> *Мигель Сервантес*

Язык JavaScript не содержит обычных классов, как к этому привыкли разработчики на C++ или Java, он предоставляет конструкторы, которые создают объекты при исполнении кода. Однако, в соответствии со стандартом ECMA-262, язык JavaScript предоставляет серию встроенных объектов, в число которых входят глобальный объект, объекты: `Object`, `Function`, `Array`, `String`, `Boolean`, `Number`, `Math`, `Date`, `RegExp`, а также объекты ошибок: `EvalError`, `RangeError`, `ReferenceError`, `SyntaxError`, `TypeError` и `URIError`. Рассмотрим некоторые из них.

Объект *Global*

Свойства и методы глобального пространства имен, о которых говорилось в *главе 50*, реализованы как свойства и методы базового объекта языка сценариев, а также объекта `Global`. Он является объектом верхнего уровня и не имеет родительского объекта. В нем определены свойства конструкторов для всех используемых в сценарии объектов, например: `Object`, `Number`, `String`, `Date` и пр., а также методы `eval()`, `isFinite()`, `isNaN()`, `parseFloat()`, `parseInt()` и др. Этот объект можно расширять новыми свойствами, что позволяет добавлять в глобальное пространство новые объекты прямо из Qt-программ.

Объект *Number*

Объект `Number` служит для хранения целых чисел. С помощью этих объектов можно определять системные пределы: `MAX_VALUE` и `MIN_VALUE`. Переменной числового типа могут быть присвоены нечисловые значения, в этом случае метод `isNaN()` возвращает значение `true`. Результаты арифметического выражения могут выйти за пределы, ограниченные максимальным и минимальным значениями, в этом случае значение будет равно `Infinity`, а для его проверки можно использовать метод `isFinite()`, который в этом случае вернет значение `false`.

Объект *Boolean*

Хотя JavaScript предоставляет этот объект, но его создание не рекомендуется. Вместо него лучше использовать константы `true` и `false`.

Каждое выражение может быть преобразовано к логическому типу. Значения `0`, `null`, `false`, `NaN`, `undefined` и пустая строка преобразуются к значению `false`, а во всех остальных случаях выражение преобразуется к значению `true`.

Объект *String*

До сих пор мы умели только создавать строки и объединять их. При помощи объекта `String` со строками можно проводить целую серию разнообразных операций.

Замена

Воспользовавшись методом `replace()`, можно заменить одну часть строки другой:

```
var str  = new String("QtScript");
var str2 = str.replace("Qt", "Java"); //=> str2="JavaScript"
```

Получение символов

Для получения символа на определенной позиции в строке можно вызвать метод `charAt()`. Номер позиции начинается с нуля:

```
var str = new String("JavaScript");
var c = str.charAt(2); //=> c='v'
```

Получение подстроки

Для получения подстроки можно воспользоваться методом `substring()`, в который необходимо передать начальную и конечную позиции. Например:

```
var str = new String("JavaScript");
var str2 = str.substring(4, str.length); //=> str2="Script"
```

Объект *RegExp*

Назначение объекта регулярного выражения заключается в проверке строк на соответствие заданному шаблону. Описание принципа работы регулярных выражений можно найти в *главе 4*, поэтому мы остановимся только на особенностях, присущих JavaScript. Итак, создавать регулярные выражения в языке сценариев можно двумя способами: инициализацией и созданием объекта. Первый способ заключается в присвоении переменной регулярного выражения, которое должно задаваться в стиле C#, то есть быть заключено в прямые слэши `/`. Например:

```
var myRegexp = /[0-9]+/;
```

Второй способ — это создание объекта регулярного выражения, что выполняется при помощи оператора `new`. Например:

```
var myRegexp = new RegExp("[0-9]+");
```

Проверка строки

Для того чтобы проверить строку на полное соответствие регулярному выражению, можно воспользоваться методом `test()`:

```
print(/[0-9]+/.test("number 95")); //=> true
print(/[0-9]+/.test("Hello")); //=> false
```

Поиск позиции совпадений

Для получения совпадений позиции строки с шаблоном регулярного выражения можно воспользоваться методом `search()`:

```
print("number 95".search(/[0-9]+/));//=> 7
```

Найденное совпадение

Для получения строки найденного совпадения используется метод `match()`:

```
print("number 95".search(/[0-9]+/)); //=> "95"
```

Объект *Array*

Объект массива `Array` является контейнером, содержащим элементы данных. В строго типизированных языках, таких как, например, C++ и Java, элементы массива должны иметь одинаковый тип, но в JavaScript это не обязательно. Таким образом, в одном и том же массиве можно размещать элементы различных типов. Создать массив можно посредством инициализации:

```
var arr = ["Evanescence", "Epica", "Xandria"];
```

Или вызовом оператора `new`:

```
var arr = new Array("Evanescence", "Epica", "Xandria");
```

Для работы с массивами предоставляется целый ряд методов, которые сведены в табл. 51.1.

***Таблица 51.1.** Методы работы с массивами*

Метод	Описание
`Concat()`	Вставка элементов в конец массива
`join()`	Объединение всех элементов массива в одну строку
`pop()`	Удаление последнего элемента массива
`push()`	Добавление элемента в конец массива
`reverse()`	Изменение порядка элементов в массиве на обратный
`shift()`	Удаление элементов массива в начале
`slice()`	Возвращение подмножества массива
`sort()`	Сортировка элементов массива
`splice()`	Вставка и удаление элементов

Таблица 51.1 (окончание)

Метод	Описание
toSource()	Преобразование массива в строку, заключенную в квадратные скобки
toString(	Преобразование массива в строку
unshift()	Вставка элементов в начало массива
unwatch()	Прекращение слежения за свойством
watch()	Установка слежения за свойством

Далее рассмотрим некоторые из этих методов.

Дополнение массива элементами

После создания и при ссылке на индекс несуществующей позиции массив может быть динамически дополнен элементами:

```
var arr = ["Evanescence", "Epica", "Xandria"];
arr[3] = "Therion";
    //=> arr=["Evanescence", "Nightwish", "Epica", "Therion"];
```

Массив можно дополнять элементами и при помощи методов: `push()`, добавляющего свои параметры в конец массива, `unshift()`, добавляющего их в начало, а также `splice()`, вставляющего их в середину. В последнем случае первым параметром нужно указать индекс начала вставки, вторым — количество символов, которые надо заменить при вставке, а далее перечислить вставляемые элементы, например:

```
var arr = new Array("Evanescence", "Epica", "Xandria");
arr.splice(1, 0, "Therion", "Nightwish");
//=> arr=["Evanescence", "Therion", "Nightwish", "Epica", "Xandria"];
```

Адресация элементов

Доступ к элементам массива осуществляется посредством их имен. Имена могут быть не только целыми числами, но и строками:

```
var arr = new Array(2);
arr["first"]     = "John";
arr["second"]    = "Paul";
var firstBeatle  = arr["first"];//=> firstBeatle=John
arr.second      += " McCartney";
var secondBeatle = arr.second; //=> secondBeatle=Paul McCartney
```

Изменение порядка элементов массива

Используя метод `reverse()`, можно менять порядок расположения элементов массива на противоположный:

```
var arr = ["Evanescence", "Epica", "Xandria"];
var arr2 = arr.reverse(); //=> arr2=["Xandria", "Epica", "Evanescence"]
```

Можно также обойтись и без первого объявления:

```
var arr2 = ["Evanescence", "Epica", "Xandria"].reverse();
```

Преобразование массива в строку

Вызов метода `join()` преобразует массив в строку, а в его необязательном параметре можно указать разделитель:

```
var arr = ["Evanescence", "Epica", "Xandria"];
var str = arr.join("**"); //=> str="Evanescence**Epica**Xandria"
```

Объединение массивов

Массивы можно объединять друг с другом при помощи метода `concat()`:

```
var str = ["Evanescence", "Epica", "Xandria"];
var str2 = str.concat(["Therion", "Nightwish"]);
   //=> str2=["Evanescence","Epica","Xandria","Therion","Nightwish"]
```

Упорядочивание элементов

Используя метод `sort()`, можно упорядочивать элементы массива:

```
var str = ["Xandria", "Evanescence", "Epica"];
var str2 = str.sort(); //=> str2=["Epica","Evanescence","Xandria"]
```

Многомерные массивы

Встроенных возможностей для создания многомерных массивов в языке JavaScript не предусмотрено. Но такие массивы можно легко создать с помощью одномерных массивов. Например, создать двумерный массив 3×3 можно следующим образом:

```
var arr = [[1, 1, 1],
           [2, 2, 2],
           [3, 3, 3]];
```

А обратиться к одному из его элементов можно так:

```
var n = arr[0][1];
```

По аналогии создадим трехмерный массив 3×3×3 и сразу же обратимся к одному из его элементов:

```
var arr = [[[1, 1, 1],
            [1, 1, 1],
            [1, 1, 1]],
           [[2, 2, 2],
            [2, 2, 2],
            [2, 2, 2]],
           [[3, 3, 3],
            [3, 3, 3],
            [3, 3, 3]],
          ];
var n = arr[0][1][2];
```

Объект *Date*

Объект `Date` предназначен для хранения даты и времени (с точностью до миллисекунд) и проведения над ними операций. Для создания объекта даты в его конструктор можно передать значение года, месяца, дня, часа, минуты, секунды и миллисекунды. Все эти параметры не обязательны, и если при создании объекта не передавать в конструктор никаких аргументов, то будет сконструирован объект с текущей датой и временем.

> ***Использовать даты до 1970 года запрещено!***
>
> Несмотря на то, что значения дат возвращаются в их привычной форме, фактическое значение хранится в миллисекундах, прошедших после полуночи 1 января 1970 года. Это обстоятельство запрещает использовать даты до 1970 года.

Создадим объекты дат — с параметрами и без них:

```
var moment = new Date(2017, 2, 6, 23, 55, 30);
var now    = new Date();
```

Для получения значения номера дня в месяце существует метод `getDate()`. Также можно воспользоваться методами: `getYear()`, `getMonth()`, `getDay()`, `getHours()`, `getMinutes()`, `getSeconds()` и `getMilliseconds()` — для получения года, номера месяца (январю соответствует 0), дня недели (воскресенье — 0, понедельник — 1 и т. д.), часов, минут, секунд и миллисекунд соответственно. Для установки этих параметров можно воспользоваться аналогичными методами, но с префиксом `set`.

> ***Нумерация значений представления даты***
>
> За исключением номера дня в месяце, все остальные значения представления даты нумеруются, начиная с нуля.

Следующий пример выводит на экран текстовое представление текущего дня недели. Номер дня недели возвращает метод `getDay()`:

```
var indexToDay = ["Sunday",
                  "Monday",
                  "Tuesday",
                  "Wednesday",
                  "Thursday",
                  "Friday",
                  "Saturday"
                 ];
var now = new Date();
print(indexToDay[now.getDay()]);
```

Следующий пример реализует отображение текущих даты и времени в виде строки и использует методы:

- `getDate()` — возвращает номер дня месяца (число в диапазоне от 1 до 31);
- `getMonth()` — возвращает номер месяца (число в диапазоне от 0 до 11);
- `getFullYear()` — возвращает год вместе с тысячелетием. Также этот метод эквивалентен значению 1900 + `getYear()`;
- `getHors()` — возвращает значение часов;

- `getMinutes()` — возвращает значение минут;
- `getSeconds()` — возвращает значение секунд.

```
var month = ["January", "February", "March", "April", "Mai", "Jun",
             "July", "August", "September", "October", "November",
             "December"
            ];
var now = new Date;
var strDateTime = now.getDate() + ". "
                  + month[now.getMonth()] + " "
                  + now.getFullYear()+ " "
                  + now.getHours()+ ":"
                  + now.getMinutes()+ ":"
                  + now.getSeconds();
print(strDateTime);
```

Вывод на экран будет иметь следующий вид:

```
"22. October 2017 16:33:32"
```

Объект *Math*

Объект `Math` содержит в себе атрибуты и методы, используемые для проведения математических операций. Доступ к этому объекту можно получить без использования конструктора. Все его атрибуты и методы определены как статические (табл. 51.2).

Таблица 51.2. *Атрибуты объекта* `Math`

Константа	Значение
`E`	Число E. Значение константы Эйлера (2,718281828459045)
`LN2`	Натуральный логарифм числа 2 (0,6931471805599453)
`LN10`	Натуральный логарифм числа 10 (2,302585092994046)
`LOG2E`	Логарифм числа E по основанию 2 (1,4426950408889633)
`LOG10E`	Логарифм числа E по основанию 10 (0,4342944819032518)
`PI`	Число π (3,141592653589793)
`SQRT1_2`	Квадратный корень из числа $^1/_2$ (0,7071067811865476)
`SQRT2`	Квадратный корень из числа 2 (1,4142135623730951)

Рассмотрим некоторые методы, предоставляемые объектом `Math`.

Модуль числа

Метод `abs()` возвращает абсолютное значение переданного в него числа. Передаваемые числа могут быть целого и действительного типа. Например:

```
var x1 = Math.abs(-123.5); //=> x1=123.5
var x2 = Math.abs(123);    //=> x2=123
```

Округление

Для округления в объекте Math есть три метода: ceil(), floor() и round(). Метод ceil() возвращает целое число, большее переданного значения или равное ему. Если в этот метод передать целое число, то он вернет его без изменений. Например:

```
var x1 = Math.ceil(5.2);  //=> x1=6
var x2 = Math.ceil(-5.5); //=> x2=-5
var x3 = Math.ceil(6);    //=> x3=6
```

Метод floor() также возвращает целое число, но меньшее переданного значения или равное ему.

```
var x1 = Math.floor(-5.4); //=> x1=-6
var x2 = Math.floor(5.9);  //=> x2=5
var x3 = Math.floor(6);    //=> x3=6
```

Метод round() округляет число до ближайшего целого значения:

```
var x1 = Math.round(5.4); //=> x1=5
var x2 = Math.round(5.5); //=> x2=6
var x3 = Math.round(6);   //=> x3=6
```

Определение максимума и минимума

Для вычисления максимума существует метод max(), который принимает несколько аргументов и возвращает значение большего из них. Аргументы могут быть как целого, так и вещественного типа. Например:

```
var x1 = Math.max(5.4, 5.5);    //=> x1=5.5
var x2 = Math.max(10, 2, 5, 3); //=> x2=10
```

Вычисление минимума осуществляется с помощью метода min(), который возвращает наименьшее из переданных значений. Например:

```
var x1 = Math.min(5.4, 5.5);    //=> x1=5.4
var x2 = Math.min(10, 2, 5, 3); //=> x2=2
```

Возведение в степень

Метод pow() принимает два аргумента: число и показатель степени (может быть как целым, так и дробным), и возвращает результат возведения числа в заданную степень. Например:

```
var x = Math.pow(2, 3); //=> x=8
```

Вычисление квадратного корня

Метод sqrt() является частным случаем операции возведения в степень с показателем $^1/_2$. Конечно, для вычисления квадратного корня можно было бы использовать метод pow(), но применение метода sqrt() будет более эффективным.

```
var x = Math.sqrt(2); //=> x= 1.4142135623730951
```

Если передать в этот метод отрицательное число, то он вернет значение NaN.

Генератор случайных чисел

Метод `random()` — это один из самых часто используемых методов, который возвращает случайное действительное число в диапазоне от 0 до 1. Если вам нужно число из произвольного диапазона, то на его базе можно реализовать следующую функцию:

```
function randomize(range)
{
    return (Math.random() * range)
}
print(randomize(1000));
```

Тригонометрические методы

Эти методы предназначены для геометрических вычислений. Самыми известными тригонометрическими функциями являются косинус (метод `cos()`), синус (метод `sin()`) и тангенс (метод `tan()`). Все эти методы принимают значения углов в радианах. А это значит, что если вы хотите использовать значения в градусах, то вам следует перевести их в радианы. Для этого надо умножить значение в градусах на число π и разделить на 180. Например, перевод 270 градусов (−90 градусов) в радианы и получение значения синуса будет выглядеть следующим образом:

```
var rad = -90 * Math.PI / 180;
print(Math.sin(rad)); //=> -1
```

Вы, наверное, уже заметили, что среди перечисленных методов не хватает метода для вычисления котангенса. Объект `Math` не содержит его, но его можно легко реализовать самому, разделив значение косинуса на значение синуса:

```
function ctg(angle)
{
    return Math.cos(angle) / Math.sin(angle);
}
```

Объект `Math` предоставляет методы и для обратных тригонометрических функций: арккосинус (`acos()`), арксинус (`asin()`) и арктангенс (`atan()`). Давайте теперь найдем угол, при котором значение синуса равно −1:

```
print(Math.asin(-1) / Math.PI * 180); //=> -90
```

Вычисление натурального логарифма

Метод `log()` осуществляет вычисление натурального логарифма:

```
var x = Math.log(Math.E); //=> x=1
```

В JavaScript не предусмотрен метод, вычисляющий десятичные логарифмы, но его несложно реализовать самому:

```
function log10(arg)
{
    return Math.log(arg) / Math.LN10;
}
var x = log10(10); //=> x=1
```

Объект *Function*

Этот объект уже был рассмотрен в *главе 50*. С его помощью можно использовать строку в качестве функции во время выполнения сценария. Для создания новой функции необходимо передать в конструктор параметры и текст. Давайте создадим свою собственную функцию для объединения строк, принимающую в качестве аргументов четыре строки, причем последняя строка будет использоваться в качестве разделителя:

```
var myJoin =
    new Function("a", "b", "c", "sep", "return a + sep + b + sep + c");
print(myJoin("Therion", "Epica", "Nightwish", "**"));
    //=> Therion**Epica**Nightwish
```

При создании объектов функций важно учитывать то обстоятельство, что трансляция объекта `Function` осуществляется при каждом его использовании, вследствие чего выполняться такой код будет гораздо медленнее, чем при реализации обычных функций языка сценариев.

Резюме

В этой главе кратко рассмотрены объекты, встроенные в язык JavaScript. Эти объекты предоставляют целый набор функциональных возможностей для проведения манипуляций со строками, массивами, выполнения математических операций и др.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/51-510/** или с помощью следующего QR-кода (рис. 51.1):



Рис. 51.1. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 52

Классы поддержки JavaScript и практические примеры

Разговор двух программистов:
— Что пишешь?
— Сейчас запустим — узнаем!

Чтобы использование языка сценариев в программах стало возможным, нужны классы, созданные на C++. Модуль `Qt Qml` предоставляет два основных класса для поддержки языка сценариев в Qt-программах, о которых мы уже успели вскользь упомянуть в *главе 49*:

- `QJSValue` — является контейнером для типов данных JavaScript;
- `QJSEngine` — представляет среду для выполнения программ, написанных на языке сценариев JavaScript.

Класс *QJSValue*

Этот класс поддерживает типы, определенные стандартом ECMA-262: `UndefinedValue` и `NullValue`.

Класс `QJSValue` дает возможность определять типы значений. Для этого он предоставляет серию методов `isT()` — например: для логического типа `isBool()`, для строки `isString()`, для массива `isArray()` и т. п. Имеются у него и методы `toT()` для преобразования значений к конкретному типу: `toString()`, `toBool()`, `toNumber()` и т. п.

Класс *QJSEngine*

Класс `QJSEngine` является главным классом модуля `Qt Qml` для языка сценариев. Любое Qt-приложение, использующее язык сценариев, должно создать хотя бы один объект этого класса. Класс `QJSEngine` как раз и предоставляет методы, связанные с выполнением кода сценария. Исполнение сценария запускается вызовом метода `QJSEngine::evaluate()`, который возвращает объект класса `QJSValue`. Вот пример выполнения простейшего сценария:

```
QJSValue value = engine.evaluate("5 * 5");
int number = value.toInt(); //number=25
```

Возникновение ошибки выполнения кода можно проверить вызовом метода `isError()` объекта класса `QJSValue`. Например:

```
QJSValue value = engine.evaluate("script error test");
if (value.isError()) {
```

```
    // Произошла ошибка при выполнении сценария
    qDebug() << value.toString();
}
```

Для того чтобы сделать объекты библиотеки Qt доступными в сценарии, нужно вызвать метод `newQObject()`. В этот метод передается указатель на объект `QObject` или на объект унаследованного от него класса. Сценарию будут доступны свойства, сигналы и слоты переданного в этот метод объекта.

Следующий шаг — установка свойств. Вся программа сценария выполняется в контексте глобального объекта. Такой объект создается автоматически и доступ к нему можно получить вызовом метода `globalObject()`. Установка свойств обычно осуществляется именно в этом объекте — чтобы сделать собственные расширения доступными для всей программы сценария. Вот пример установки свойства переменной целого значения:

```
QJSValue myInt = QJSValue(&scriptEngine, 2007);
engine.globalObject().setProperty("myInt", myInt);
```

Практические примеры

Теперь, когда мы познакомились с основными классами поддержки языка сценариев, самое время приступить к рассмотрению их применения в конкретных ситуациях. Так что давайте перейдем от теории к практике.

«Черепашья» графика

Многие из вас, наверное, помнят из школьных уроков информатики «черепашью» графику, которая используется в языке программирования Logo. Принцип рисования прост. «Черепаха» помещается в середину экрана, и перемещать ее можно при помощи специально предусмотренных команд. Двигаясь по экрану, «черепаха» оставляет за собой след, складывающийся в различные рисунки.

В следующем примере (листинги 52.5–52.10) мы предоставляем возможность для написания собственных сценариев с использованием «черепашьей» графики (рис. 52.1). Пользователь может вводить в левой части окна свои собственные сценарии и выполнять их нажатием кнопки **Evaluate** (Вычислить). В сценарии можно использовать следующие команды:

- `forward(n)` — перемещение «черепахи» вперед на `n` пикселов с рисованием следа;
- `back(n)` — перемещение «черепахи» назад на `n` пикселов с рисованием следа;
- `left(nAngle)` — поворот «черепахи» влево на угол `nAngle`;
- `right(nAngle)` — поворот «черепахи» вправо на угол `nAngle`;
- `reset()` — очистка экрана и установка «черепахи» в середину.

В основной функции, приведенной в листинге 52.1, мы создаем виджет рабочей области для «черепашьей» графики и показываем его на экране.

Листинг 52.1. Создание «черепашьей» графики — функция `main()` (файл main.cpp)

```
#include <QApplication>
#include "TurtleWorkArea.h"
```

```
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    TurtleWorkArea turtleWorkArea;
    turtleWorkArea.show();

    return app.exec();
}
```

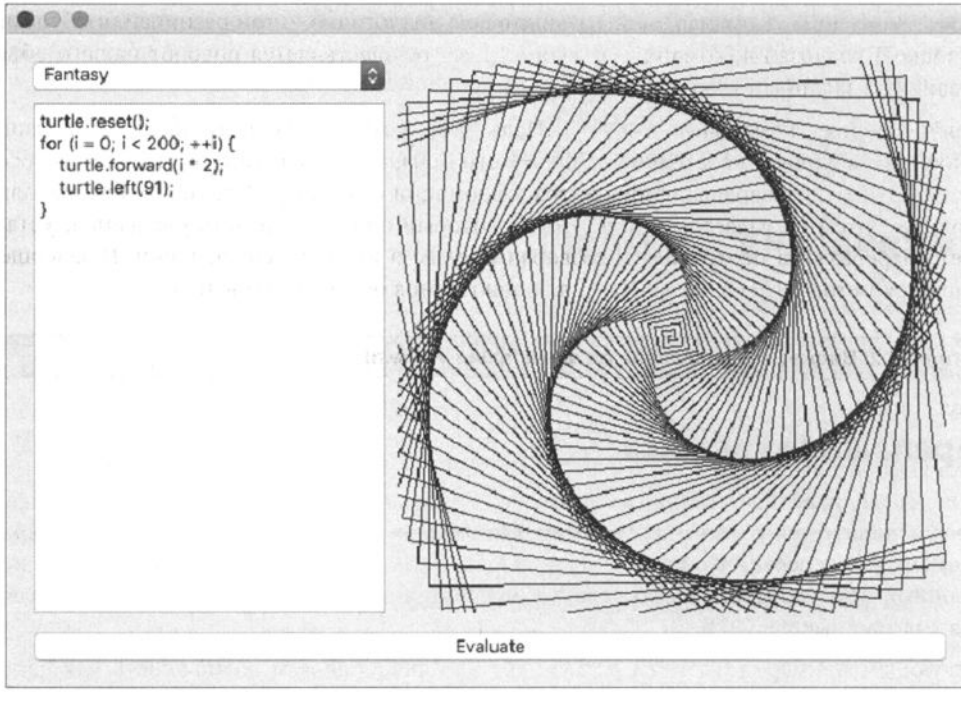


Рис. 52.1. «Черепашья» графика

В листинге 52.2 приведен класс `Turtle`, реализующий команды рисования «черепашьей» графики. Этот класс определяет два атрибута: `m_pix` и `m_painter`. Первый атрибут — это растровое изображение, которое является полотном для нашей «черепахи». Второй атрибут — это объект, с помощью которого наша «черепаха» будет рисовать. При инициализации мы задаем в конструкторе размер растрового изображения для нашего полотна: 400×400. Для того чтобы все операции объекта рисования (`m_painter`) ассоциировались с растровым изображением (`m_pix`), при инициализации объекта рисования мы передаем ссылку на растровое изображение. Мы также запрещаем изменять размер растрового изображения в нашем виджете при помощи метода `QWidget::setFixedSize()`. В общем-то ничего страшного не случилось бы, если бы мы позволили изменять размеры, так как в методе `QWidget::paintEvent()` эта возможность предусмотрена методом `QPainter::drawPixmap()`, который растягивает наше полотно на всю область виджета. Однако фиксированный размер позволяет показывать результат рисования в масштабе 1:1, то есть без искажений, связанных с изменением размера. После установки размера вызываем метод инициализации `init()`. В методе `init()`, чтобы можно было визуально отличить область рисования, при

помощи метода `QPixmap::fill()` закрашиваем ее желтым цветом. С помощью метода `QPainter::translate()` выполняем трансформацию нашего объекта рисования и смещаем его начало координат в центр полотна. Вызов метода `QPixmap::rotate()` осуществляет поворот на 90 градусов. Теперь наша «черепаха» приведена в исходное положение и готова для рисования, дело осталось только за реализацией команд.

Чтобы команды были доступны из сценария, мы объявляем их как слоты. Реализация команды `forward()` заключается в вызове метода рисования из нулевой точки линии длиной `n` с последующим перемещением объекта рисования на эту длину. Чтобы результат быть виден сразу после выполнения команды, вызываем метод перерисовки `QWidget::repaint()`. Команда `back()` реализована аналогично, с той разницей, что у `n` меняется знак. В реализации команд `left()` и `right()` осуществляется поворот нашего объекта рисования на заданный угол `nAlpha` в градусах.

Самая последняя команда — `reset()`. Цель этой команды вернуть нашу «черепашью» графику в исходное состояние. Для этого сначала вызывается метод `QPainter::resetTransform()`, возвращающий все трансформации объекта рисования в первоначальное состояние. Затем вызывается метод `init()`, который очищает область рисования, устанавливает «черепаху» в центр поля и разворачивает ее в нужном направлении. В завершение вызывается метод `QWidget::repaint()` для обновления области рисования.

Листинг 52.2. Виджет «черепашьей» графики (файл Turtle.h)

```
#pragma once

#include <QtWidgets>

// ====================================================================
class Turtle : public QWidget {
Q_OBJECT
private:
    QPixmap  m_pix;
    QPainter m_painter;

public:
    Turtle(QWidget* pwgt = 0) : QWidget(pwgt)
                              , m_pix(400, 400)
                              , m_painter(&m_pix)
    {
        setFixedSize(m_pix.size());
        init();
    }

protected:
    void init()
    {
        m_pix.fill(Qt::yellow);
        m_painter.translate(rect().center());
        m_painter.rotate(-90);
    }
```

```
    virtual void paintEvent(QPaintEvent*)
    {
        QPainter painter(this);
        painter.drawPixmap(rect(), m_pix);
    }

public slots:
    void forward(int n)
    {
        m_painter.drawLine(0, 0, n, 0);
        m_painter.translate(n, 0);
        repaint();
    }

    void back(int n)
    {
        m_painter.drawLine(0, 0, -n, 0);
        m_painter.translate(-n, 0);
        repaint();
    }

    void left(int nAngle)
    {
        m_painter.rotate(-nAngle);
    }

    void right(int nAngle)
    {
        m_painter.rotate(nAngle);
    }

    void reset()
    {
        m_painter.resetTransform();
        init();
        repaint();
    }
};
```

Чтобы разрешить отдавать нашей «черепахе» команды и выполнять их, нам нужно объединить необходимые для этого компоненты. Именно эту роль и выполняет класс `TurtleWorkArea`, приведенный в листинге 52.3. В этом классе определены: атрибут области написания программ сценария (указатель `m_ptxt`), атрибут для исполнения программ (`m_scriptEngine`) и область рисования «черепашьей» графики (указатель `m_pTurtle`).

Листинг 52.3. Рабочая область «черепахи» (файл TurtleWorkArea.h)

```
#pragma once

#include <QWidget>
#include <QJSEngine>
```

```
class QTextEdit;
class Turtle;

// ============================================================================
class TurtleWorkArea : public QWidget {
Q_OBJECT
private:
    QTextEdit* m_ptxt;
    QJSEngine  m_scriptEngine;
    Turtle*    m_pTurtle;

public:
    TurtleWorkArea(QWidget* pwgt = 0);

private slots:
    void slotEvaluate (   );
    void slotApplyCode(int);
};
```

В конструкторе (листинг 52.4) мы создаем виджет для работы с «черепашьей» графикой `m_pTurtle` и виджет выпадающего списка для выбора готовых примеров «черепашьей» графики. Их четыре: **Haus vom Nikolaus** (Дом Николауса), **Curly** (Кудряшка), **Circle** (Окружность) и **Fantasy** (Фантазия). За смену примеров отвечает слот `slotApplyCode()`, поэтому мы соединяем его с сигналом `activated()`, который высылает выпадающий список при смене элемента. Вызовом слота `slotApplyCode()` мы устанавливаем текущий пример. После этого вызываем метод `newQObject()` объекта класса `QJSEngine` со ссылкой на виджет «черепашьей» графики. Этот метод возвращает объект типа данных языка сценария, который сохраняется в объекте класса `QJSValue`. Мы передаем это значение вторым параметром в метод `QJSValue::setProperty()` глобального объекта, а в первом параметре устанавливаем имя, по которому этот объект можно будет использовать.

Затем мы создаем кнопку (указатель `pcmd`), при нажатии на которую будет выполняться программа, введенная или модифицированная в виджете многострочного текстового поля. Для этого кнопка соединяется со слотом `slotEvaluate()`. В завершение объекты размещаются в виджете рабочей области с использованием табличной компоновки.

Листинг 52.4. Конструктор класса `TurtleWorkArea()` (файл TurtleWorkArea.cpp)

```
TurtleWorkArea::TurtleWorkArea(QWidget* pwgt/*=0*/) : QWidget(pwgt)
{
    m_pTurtle = new Turtle;
    m_pTurtle->setFixedSize(400, 400);
    m_ptxt = new QTextEdit;

    QComboBox*  pcbo = new QComboBox;
    QStringList lst;

    lst << "Haus vom Nikolaus" << "Curly" << "Circle" << "Fantasy";
    pcbo->addItems(lst);
```

```
    connect(pcbo, SIGNAL(activated(int)), SLOT(slotApplyCode(int)));
    slotApplyCode(0);

    QJSValue scriptTurtle =
        m_scriptEngine.newQObject(m_pTurtle);
    m_scriptEngine.globalObject().setProperty("turtle", scriptTurtle);

    QPushButton* pcmd = new QPushButton("&Evaluate");
    connect(pcmd, SIGNAL(clicked()), SLOT(slotEvaluate()));

    QGridLayout* pgrdLayout = new QGridLayout;

    pgrdLayout->addWidget(pcbo, 0, 0);
    pgrdLayout->addWidget(m_ptxt, 1, 0);
    pgrdLayout->addWidget(m_pTurtle, 0, 1, 2, 1);
    pgrdLayout->addWidget(pcmd, 2, 0, 1, 2);
    setLayout(pgrdLayout);
}
```

В слоте `slotEvalute()`, приведенном в листинге 52.5, методом `QTextEdit::toPlainText()` извлекается текст, находящийся в виджете многострочного текстового поля, и передается на исполнение в метод `QJSEngine::evaluate()`. Возвращаемое этим методом значение анализируется на наличие ошибок в операторе `if` при помощи вызова метода `isError()`. В случае возникновения ошибок при выполнении сценария выводится окно сообщения.

Листинг 52.5. Слот `slotEvaluate()` (файл TurtleWorkArea.cpp)

```
void TurtleWorkArea::slotEvaluate()
{
    QJSValue result =
        m_scriptEngine.evaluate(m_ptxt->toPlainText());
    if (result.isError()) {
        QMessageBox::critical(0,
                              "Evaluating error",
                              result.toString(),
                              QMessageBox::Yes
                             );
    }
}
```

Слот `slotApplyCode()`, показанный в листинге 52.6, предоставляет четыре готовых примера для «черепашьей» графики, написанные на языке сценария. Вы можете экспериментировать и изменять их исходный текст в виджете многострочного текстового поля. Проявите немного творчества, и вам наверняка удастся сделать их еще более оригинальными.

Листинг 52.6. Слот `slotApplyCode()` (файл TurtleWorkArea.cpp)

```
void TurtleWorkArea::slotApplyCode(int n)
{
    QString strCode;
```

```
    switch (n) {
    case 0:
        strCode = "var k = 100;\n"
                  "turtle.reset();\n"
                  "turtle.right(90);\n"
                  "turtle.back(-k);\n"
                  "turtle.left(90);\n"
                  "turtle.forward(k);\n"
                  "turtle.left(30);\n"
                  "turtle.forward(k);\n"
                  "turtle.left(120);\n"
                  "turtle.forward(k);\n"
                  "turtle.left(30);\n"
                  "turtle.forward(k);\n"
                  "turtle.left(135);\n"
                  "turtle.forward(Math.sqrt(2)*k);\n"
                  "turtle.left(135);\n"
                  "turtle.forward(k);\n"
                  "turtle.left(135);\n"
                  "turtle.forward(Math.sqrt(2)*k);\n";
        break;
    case 1:
        strCode = "turtle.reset();\n"
                  "for (i = 0; i < 2; ++i) {\n"
                  "   for(j = 0; j < 100; ++j) {\n"
                  "       turtle.forward(15);\n"
                  "       turtle.left(100 - j);\n"
                  "   }\n"
                  "   turtle.right(180);\n"
                  "}";
        break;
    case 2:
        strCode = "turtle.circle = function()\n"
                  "{\n"
                  "    for (var i = 0; i < 360; ++i) {\n"
                  "        this.forward(1);\n"
                  "        this.left(1);\n"
                  "    }\n"
                  "}\n"
                  "turtle.reset();\n"
                  "turtle.circle();\n";
        break;
     default:
        strCode = "turtle.reset();\n"
                  "for (i = 0; i < 200; ++i) {\n"
                  "    turtle.forward(i * 2);\n"
                  "    turtle.left(91);\n"
                  "}";
    }
    m_ptxt->setPlainText(strCode);
}
```

Давайте перейдем теперь к рассмотрению примеров, приведенных в листинге 52.6:

- пример самой первой секции `switch` (`case 0`) — это рисунок дома Николауса. В нем использованы все команды, предоставляемые нашим виджетом «черепашьей» графики, поэтому он является прекрасным тестом проверки работоспособности всех команд нашей «черепашки»;
- пример во второй секции `switch` (`case 1`) рисует две кудряшки;
- третий пример (`case 2`) отображает окружность. Обратите внимание, что в этом примере мы расширяем виджет нашей «черепахи» еще одним методом `circle()`. Таким способом можно расширять объекты и виджеты Qt дополнительными методами прямо из языка сценариев JavaScript;
- код последней секции `switch` (`default`) я назвал *фантазией*. Поэкспериментируйте и замените значение 91 в методе `left()` на 120, и вы увидите египетскую пирамиду. Значение 171 отобразит солнце, 160 — звезду, 181 — веер, 45 — лабиринт, 115 — подсолнух, а 31 — спираль. Быть может, вы найдете еще какие-либо интересные узоры, изменяя этот параметр, попробуйте!

Сигналы, слоты и функции

Добавив объект, автор сценария может вызывать слоты объекта, изменять любые его свойства и получать доступ к его потомкам. Но это еще не все! В программе сценариев можно присоединяться к сигналам Qt-объектов. Эти сигналы могут быть присоединены к Qt-слотам, а также к функциям и методам классов, определенных в самом сценарии.

Для того чтобы соединить сигнал, нужно вызвать метод `connect()` отправляющего сигнал объекта и передать в него слот, функцию или метод объекта сценария. Возьмем, например, соединение сигнала `clicked()` кнопки с функцией сценария `buttonClicked()`:

```
function buttonClicked()
{
    label.text = "The button was clicked";
}
cmd.clicked.connect(buttonClicked);
```

Для отсоединения сигнала нужно поступить аналогичным образом, только вместо метода `connect()` надо вызвать метод `disconnect()`. Например:

```
cmd.clicked.disconnect(buttonClicked);
```

В следующем примере (листинги 52.7 и 52.8) организуется соединение сигнала из языка сценария четырьмя способами: со слотом Qt-виджета, функцией и методом объекта класса сценария (рис. 52.2). Этот пример также демонстрирует возможность доступа к потомкам.



Рис. 52.2. Пример соединения сигнала разными способами

В основной программе, приведенной в листинге 52.7, мы создаем виджет верхнего уровня wgt. Отметьте очень важный момент — для всех создаваемых далее виджетов мы вызываем метод setObjectName(), предназначенный для установки имени объекта. Это необходимо для того, чтобы можно было получить доступ к ним из сценария. Создаваемых виджетов пять: виджет текстового поля (указатель ptxt) и четыре виджета надписи (указатели plbl1, plbl2, plbl3 и plbl4). После размещения этих виджетов при помощи вертикальной компоновки мы осуществляем отображение виджета wgt посредством метода show(). Затем загружаем файл сценария script.js, и поскольку этот файл не является составляющей ресурса, нам необходимо проверить успешность операции загрузки. В цикле foreach устанавливаем виджеты pwgt, ptxt, plbl1, plbl2, plbl3 и plbl4 в глобальном объекте сценария (метод globalObject()) под именами "wgt", "lineedit", "label1", "label2", "label3", и "label4", вызываем метод evaluate() и выполняем код сценария, приведенный в листинге 52.8. Этот метод возвращает объект класса QJSValue, который нам нужен для того, чтобы узнать о возникновении ошибок при выполнении и отобразить их в окне сообщения.

Листинг 52.7. Программа для демонстрации соединения сигнала разными способами (файл main.cpp)

```
#include <QtWidgets>
#include <QJSEngine>

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget*     pwgt = new QWidget;

    QLineEdit* ptxt = new QLineEdit;
    ptxt->setObjectName("lineedit");

    QLabel* plbl1 = new QLabel("1");
    plbl1->setObjectName("label1");

    QLabel* plbl2 = new QLabel("2");
    plbl2->setObjectName("label2");

    QLabel* plbl3 = new QLabel("3");
    plbl3->setObjectName("label3");

    QLabel* plbl4 = new QLabel("4");
    plbl4->setObjectName("label4");

    //Layout Setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(ptxt);
    pvbxLayout->addWidget(plbl1);
    pvbxLayout->addWidget(plbl2);
    pvbxLayout->addWidget(plbl3);
    pvbxLayout->addWidget(plbl4);
    pwgt->setLayout(pvbxLayout);
```

```
    pwgt->show();

    //Script part
    QJSEngine se;
    QFile     file(":/script.js");
    if (file.open(QFile::ReadOnly)) {
        QJSValue sw = se.newQObject(pwgt);

        se.globalObject().setProperty("wgt", sw);

        QList<QObject*> lst = pwgt->findChildren<QObject*>();
        foreach(QObject* pobj, lst) {
            sw = se.newQObject(pobj);
            se.globalObject().setProperty(pobj->objectName(), sw);
        }

        QJSValue result =
            se.evaluate(QLatin1String(file.readAll()));
        if (result.isError()) {
            QMessageBox::critical(0,
                                  "Evaluating error",
                                  result.toString(),
                                  QMessageBox::Yes
                                 );
        }
    }
    else {
        QMessageBox::critical(0,
                              "File open error",
                              "Can not open the script file",
                              QMessageBox::Yes
                             );
    }

    return app.exec();
}
```

В сценарии, приведенном в листинге 52.8, мы определяем класс `MyClass`, содержащий метод `updateText()` и атрибут виджета надписи `label`, который инициализируется виджетом, переданным в конструкторе в качестве аргумента. При помощи метода `updateText()` и посредством атрибута `label` мы получаем доступ к виджету надписи. В объекте надписи в качестве текста (свойство `text`) устанавливаем строку, переданную в этот метод.

Реализация функции `updateText()` по своей сути похожа на метод, описанный ранее, с той лишь разницей, что мы выполняем изменение текста у атрибута `label1`.

После этого в виджете верхнего уровня `wgt` при помощи свойства `windowsTitle` устанавливаем заголовок окна `"Signal And Slots"`. Свойство `text` нашего текстового поля мы инициализируем строкой `"Test"`. Затем соединяем его сигнал `textChanged` с функцией `updateText()`. Теперь эта функция будет вызываться при любых изменениях текста этого виджета.

Далее мы создаем объект `myObject` нашего класса `MyClass`, передаем в его конструктор виджет `label2` и соединяем его метод `updateText()` с сигналом `textChanged()`.

Следующее соединение осуществляется между двумя виджетами Qt напрямую, для чего в метод `connect()` передаем слот виджета надписи (`label3`) `setText()`.

Самое последнее соединение с виджетом надписи `label4` мы производим с функцией, которую реализуем прямо в методе `connect()`.

Теперь, после изменения текста в виджете текстового поля, новый текст будет отображен сразу в четырех виджетах надписей (см. рис. 52.2).

Листинг 52.8. Соединение сигнала разными способами. Сценарий на JavaScript (файл script.js)

```
function MyClass(label)
{
    this.label = label
}

MyClass.prototype.updateText = function(str)
{
    this.label.text = str;
}

function updateText(str)
{
    label1.text = str;
}

wgt.windowTitle = "Signal And Slots";
lineedit.text = "Test";
lineedit.textChanged.connect(updateText);
var myObject = new MyClass(label2);
lineedit.textChanged.connect(myObject, myObject.updateText);
lineedit.textChanged.connect(label3.setText);
lineedit.textChanged.connect(function(str){label4.text = str});
```

Полезные дополнительные функции

В качестве следующего примера дополним JavaScript несколькими полезными функциями, которые могут очень сильно пригодиться при программировании на этом языке. Для этого создадим отдельный файл C++ JSTools.h (листинг 52.9), в который поместим эти функции. В будущем вы можете расширять этот файл, дополняя его по мере необходимости своими собственными функциями. Программа, показанная на рис. 52.3, демонстрирует использование реализованных функций из JavaScript.

Листинг 52.9. Полезные функции для JavaScript (файл JSTools.h)

```
#pragma once

#include <QtWidgets>
```

```
// ============================================================================
class JSTools : public QObject {
Q_OBJECT
public:
    JSTools(QObject* pobj = 0) : QObject(pobj)
    {
    }

public slots:
    void print(const QString& str)
    {
        qDebug() << str;
    }

    void alert(const QString& strMessage)
    {
        QMessageBox::information(0, "", strMessage);
    }

    void quitApplication()
    {
        qApp->quit();
    }

    QStringList dirEntryList(const QString& strDir,
                             const QString& strExt
                            )
    {
        QDir dir(strDir);
        return dir.entryList(strExt.split(" "), QDir::Files);
    }
};
```

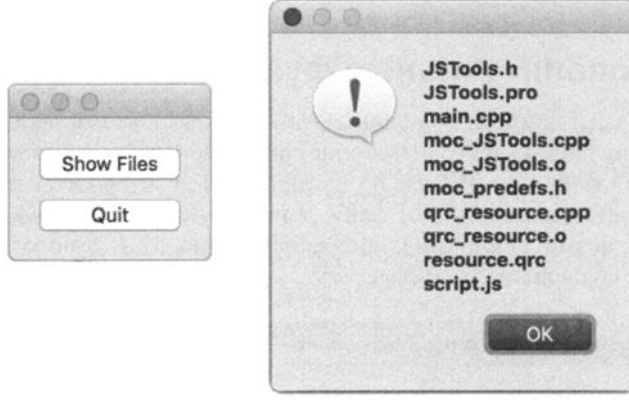


Рис. 52.3. Пример использования дополнительных функций для JavaScript

В листинге 52.9 все функции, которые мы хотим сделать доступными для языка JavaScript, нам необходимо реализовать в виде слотов. Первая функция — это функция вывода `print()`, ее мы реализуем на базе `qDebug()`. Вторая функция — это очень популярная в JavaScript функция `alert()`, с помощью которой отображают сообщения в отдельном диалоговом окне, эту функцию используют в языке JavaScript очень часто также и для отладки. Третья функция — `quitApplication()` — даст возможность завершать программу прямо из JavaScript-кода. Все наши функции не возвращают значений, поэтому для демонстрации класс `JSTools` включает функцию `dirEntryList()`, которая будет возвращать список файлов указанного в первом параметре каталога `strDir` и с расширениями во втором параметре `strExt`.

Листинг 52.10. Реализация функций в виде слотов (файл main.cpp)

```
#include <QtWidgets>
#include <QJSEngine>
#include "JSTools.h"

int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QWidget*     pwgt = new QWidget;

    QPushButton* pcmdFiles = new QPushButton("Show Files");
    pcmdFiles->setObjectName("cmdFiles");

    QPushButton* pcmdQuit = new QPushButton("Quit");
    pcmdQuit->setObjectName("cmdQuit");

    //Layout Setup
    QVBoxLayout* pvbxLayout = new QVBoxLayout;
    pvbxLayout->addWidget(pcmdFiles);
    pvbxLayout->addWidget(pcmdQuit);
    pwgt->setLayout(pvbxLayout);

    pwgt->show();

    //Script part
    QJSEngine se;
    QFile     file(":/script.js");
    if (file.open(QFile::ReadOnly)) {
        QJSValue sw = se.newQObject(pwgt);

        se.globalObject().setProperty("wgt", sw);

        QList<QObject*> lst = pwgt->findChildren<QObject*>();
        foreach(QObject* pobj, lst) {
            sw = se.newQObject(pobj);
            se.globalObject().setProperty(pobj->objectName(), sw);
        }
```

```
        JSTools* pjt = new JSTools;
        sw = se.newQObject(pjt);
        QString strClassName = pjt->metaObject()->className();
        se.globalObject().setProperty(strClassName, sw);

        QJSValue result =
            se.evaluate(QLatin1String(file.readAll()));
        if (result.isError()) {
            QMessageBox::critical(0,
                                  "Evaluating error",
                                  result.toString(),
                                  QMessageBox::Yes
                                 );
        }
    }
    else {
        QMessageBox::critical(0,
                              "File open error",
                              "Can not open the script file",
                              QMessageBox::Yes
                             );
    }

    return app.exec();
}
```

Принцип действия основной программы, приведенной в листинге 52.10, аналогичен уже рассмотренному нами листингу 52.7 с той лишь разницей, что мы делаем доступными для языка JavaScript две кнопки: `pcmdFiles`, `pcmdQuit` и объект, произведенный от класса `JSTools`. За действия кнопок будет отвечать программа на JavaScript, которая приведена в листинге 52.11.

Листинг 52.11. Использование дополнительных функций в JavaScript (файл script.js)

```
function showFiles()
{
    var files = JSTools.dirEntryList(".", "*");
    JSTools.alert(files.join("\n"));
}

JSTools.print("JSTools test!");
cmdFiles.clicked.connect(showFiles);
cmdQuit.clicked.connect(function(){JSTools.quitApplication()});
```

В функции `showFiles()` (листинг 52.11) мы при помощи объекта `JSTools`, вызовом `dirEntryList()`, получаем список всех файлов текущего каталога. Этот список мы объединяем в одну строку, вызовом «родного» метода JavaScript `join()`. Затем передаем результат для отображения в функцию `alert()`, объекта `JSTools`.

Первым в программе JavaScript будет отработан вызов функции `print()` из объекта `JSTools`, которая отобразит на консоли текст: `JSTools test!`. Затем сигнал `clicked()` кнопки нажатия `cmdFiles` будет соединен с только что рассмотренной функцией `shotFiles()`. Сигнал кнопки `cmdQuit` мы соединяем с функцией, в которой будет происходить вызов функции `quitApplication()` из объекта `JSTools`, и это должно привести к завершению приложения.

Резюме

В этой главе мы познакомились с Qt-классами, при помощи которых можно обеспечивать поддержку сценариев в приложениях. Их использование позволяет сделать любые свойства, сигналы и слоты и потомки объекта доступными для JavaScript. Центральным классом поддержки является `QJSEngine`, который дает возможность выполнять код написанных на JavaScript сценариев.

Язык сценариев поддерживает механизм сигналов и слотов. Сигналы, которые посылает объект, могут быть связаны как с функциями и методами классов JavaScript, так и с обычными слотами Qt-объектов.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/52-510/** или с помощью следующего QR-кода (рис. 52.4):



Рис. 52.4. QR-код для доступа на страницу комментариев к этой главе



ЧАСТЬ VIII

Технология Qt Quick

Хорошо летают только красивые самолеты.
А. Н. Туполев

Глава 53. Знакомство с Qt Quick

Глава 54. Элементы

Глава 55. Управление размещением элементов

Глава 56. Элементы графики

Глава 57. Пользовательский ввод

Глава 58. Анимация

Глава 59. Модель/Представление

Глава 60. Qt Quick и C++

Глава 61. 3D-графика Qt 3D



ГЛАВА 53

Знакомство с Qt Quick

> Настройте сознание на будущие достижения, и вы увидите, что прошлые ошибки часто способствуют удаче в будущем.
>
> *Наполеон Хилл*

Qt Quick — это набор технологий, предназначенных для создания анимированных, динамических, пользовательских интерфейсов нового поколения, которые становятся нормой уже не только для мобильных устройств, но и настольных компьютеров. Кроме того, это абсолютно новый подход к их разработке. Сам набор технологий состоит в основном из следующих составляющих:

- *QML* — новый язык и сразу же движок для его интерпретации. Легкий в изучении и обладает похожим на JSON синтаксисом (см. *главу 50*);
- *Qt* — библиотека, которой и посвящена вся эта книга;
- *JavaScript* — язык программирования. С его синтаксисом можно ознакомиться в *главах 50* и *51* этой книги;
- *Qt Creator* — интегрированная среда для разработки (см. *главу 47*).

А зачем?

Действительно, должны быть веские причины для того, чтобы учить что-то новое, тем более, что сама библиотека Qt — это мощный инструмент, и с ее помощью можно реализовать практически все. Давайте проясним причины, которые могут вызвать ваш интерес к изучению этой технологии.

Опыт показывает, что дизайнеры и программисты довольно тяжело понимают друг друга и, как следствие, стараются избегать общения. Так что язык QML (Qt Meta-Object Language, метаобъектный язык Qt) разрабатывался как средство для связи дизайнеров с программистами. Благодаря QML, дизайнер говорит с разработчиком на одном и том же языке, и им ничего дополнительно не приходится объяснять друг другу, — они могут просто модифицировать исходный код. А это дает возможность быстро создавать прототипы (Rapid Prototyping) программных продуктов. Выполняя роль связующего звена, QML позволяет разработчикам программного обеспечения работать совместно с дизайнерами. Это очень важно, так как дизайнеру обычно требуется много времени, чтобы изготовить нужные картинки прототипа и передать их программисту для реализации. И важно не забывать и учитывать еще и то обстоятельство, что дизайнер может создавать прототип без каких бы то ни было ограничений, используя имеющиеся у него средства редактирования. Но в C++ есть

собственные ограничения, и у вас, как у программиста, наверняка возникнут проблемы, а также отнимут уйму времени перфекционистские[1] устремления, согласно которым вы будете изо всех сил стараться приблизить свое творение к предоставленному вам прототипу. Даже если вам дали прототип не в картинках и описании, а в каком-нибудь анимационном формате, все равно автоматических средств переработки кода из этого формата в C++ вы вряд ли найдете.

Увы, но это горькая правда типичного цикла разработки пользовательского интерфейса. А вот если дизайнер предоставит прототип на языке QML, то все сразу будет выглядеть иначе, потому что созданный таким образом прототип уже является стартовой версией для готового приложения, на базе которого разработчики могут работать дальше. Сам же прототип, а значит, и приложение, сразу же будут тестироваться в настоящих эксплуатационных условиях, у дизайнера появятся те же ограничения, что и у разработчиков, и вам не придется больше заниматься подгонкой. Кроме того, есть средства для конвертирования дизайна созданного в графических редакторах Adobe Photoshop и GIMP прямо в QML.

А вот и следующий аргумент в пользу Qt Quick — натуральная, окружающая нас природа намного разнообразнее и сложнее, и она не работает по принципу обычных виджетов Qt, когда виджет мгновенно появляется в нужном месте. Это выглядит неправдоподобно и неестественно. Желательно, чтобы интерфейс вел себя иначе, используя поведенческие реакции, присущие реальному миру, и тем самым как можно больше приближался к интуитивному восприятию человека. А это значит, что в приложение необходимо добавить немного анимации, которая на Qt может стоить большого количества программного кода (некоторые примеры использования в Qt анимации вы можете найти в *главе 22*).

То, что технология Qt Quick является интерпретируемой, дает еще одно преимущество — отсутствует промежуточный процесс компиляции. Компиляция отнимает у разработчика время, так как он вынужден ждать, когда полностью будет откомпилирован и скомпонован исполняемый модуль. Отсутствие компиляции позволяет вам быстро изменять программу, сразу же запускать ее и тут же смотреть на сделанные вами изменения в действии. То есть каждый разработанный элемент сразу же будет доступен к использованию.

Еще один положительный аргумент касается проектов, код которых пишется на «чистом» языке QML с JavaScript, но без использования C++, либо с C++, но без изменения исходного кода, поскольку двоичный платформозависимый код не должен изменяться. Дело в том, что для законченных программных продуктов требуется специальная инсталляционная программа, устанавливающая их на компьютер пользователя. Если учесть, что компьютеры работают под управлением разных операционных систем, то задача еще больше усложняется, — ведь приходится создавать разные инсталляционные пакеты для каждой платформы. Например, если вы захотите распространять свою программу для трех настольных операционных систем: Windows, Mac OS X и Linux, то вам потребуется не только откомпилировать ее на все эти платформы, но еще и позаботиться о том, чтобы она могла устанавливаться на эти платформы при помощи специальных программ инсталляции. Следует заметить, что в веб-разработках таких проблем нет, поэтому Qt Quick использует парадигму веб-подхода: ее пакеты содержат при себе все необходимое, поэтому инсталляция совсем не обязательна, более того, выполнение самой программы возможно даже посредством компьютерной сети или Интернета. Единственным условием является наличие на исполняющей стороне приложения, способного интерпретировать эти Qt Quck-программы. Такой программой может быть, например, программа qmlscene, которая входит в комплект поставки Qt.

---

[1] Перфекционизм (от фр. *perfection* — совершенствование) — попросту говоря, это стремление сделать свой продукт максимально приближенным к идеалу. — *Ред.*

Еще один аргумент — это мобильные устройства. Известно, что они обладают ограниченными ресурсами, и в распоряжении программиста нет мощного центрального процессора, память ограниченна, и экран обладает низким разрешением. Одна из целей, которая преследовалась при создании Qt Quick, была возможность работы именно в этих условиях.

И наконец, Qt Quick предоставляет легкие, легко изменяемые и расширяемые элементы как со стороны самого Qt Quick, так и со стороны C++.

Мне удалось разбудить ваш интерес, дорогой читатель? Тогда читайте дальше!

Введение в QML

Теперь остановимся немного подробнее на языке QML (Qt Meta-Object Language, метаобъектный язык Qt), лежащем в основе технологии Qt Quick.

QML — это описательный язык, он описывает, как выглядят и как взаимодействуют друг с другом элементы пользовательского интерфейса. В силу своего описательного характера этот язык не предполагает в себе использования конструкций программирования — например, циклов. И если вам требуется переместить с одного места на другое какой-либо объект, то этот процесс нужно описать. Впрочем, использовать те же циклы возможно благодаря встроенному в QML языку JavaScript. Если вы веб-дизайнер или обладаете соответствующими навыками, то окажетесь в привычной для вас среде, а если работали с AdobeFlash, то тоже очень легко сможете освоиться с QML.

Если же вы до настоящего момента создавали пользовательский интерфейс традиционными методами с помощью C++ или какого-либо другого языка программирования, то, скорее всего, вам потребуется небольшой переворот в сознании, чтобы понять философию создания приложений на QML, а также, возможно, понадобится слегка очистить свою память и научиться думать немного иначе. Но не спешите делать преждевременные выводы. QML — простой, легко осваиваемый и, в то же время, мощный язык программирования, который обладает элегантным синтаксисом. Он очень гибок, и рассчитан на создание пользовательских интерфейсов с использованием анимации. С его помощью вы можете создавать и воплощать собственные идеи и экспериментировать.

В QML встроен доступ ко всем имеющимся технологиям Qt — таким как, например, Qt/C++, `QtWebEngine`, `Qt3D`, `QtLocation` и так далее, имеется также и доступ к метаинформации объектов — например, к свойствам, вызываемым методами (invokable methods), декларируемыми при помощи макроса `Q_INVOKABLE` и т. п. QML обладает системой версионализации модулей. Интегрированная среда разработки Qt Creator предоставляет хорошую поддержку этого языка.

В табл. 53.1 описаны модули, которые можно использовать в QML. Для этого перед указанием модуля и номера его версии в QML-программе должна предшествовать директива `import`.

Таблица 53.1. *Модули для QML*

Модуль	Описание
`Qt3D.Animation 2.9`	Набор элементов, предназначенных для анимации в Qt 3D
`Qt3D.Core 2.0`	Основные возможности для поддержки 3D в режиме реального времени
`Qt3D.Extras 2.0`	Набор дополнительных элементов для работы в Qt 3D

Таблица 53.1 (продолжение)

Модуль	Описание
Qt3D.Input 2.0	Элементы поддержки ввода пользователя для приложений, использующих Qt 3D
Qt3D.Logic 2.0	Синхронизация кадров с движком Qt 3D
Qt3D.Renderer 2.0	Содержит функциональные возможности для поддержки 2D- и 3D-рендеринга с использованием Qt 3D
Qt3D.Scene2D 2.9	Предоставляет возможность рендеринга содержимого из QML в текстуру Qt 3D
QtBuetooth 5.2	Поддержка работы с беспроводными устройствами Bluetooth
QtCanvas3D 1.1	Предоставляет возможность производить рисование в трехмерном пространстве способом, схожим с WebGL
QtCharts 2.2	Предоставляет набор простых в использовании элементов для отображения диаграмм и графиков
QtGraphicsEffects 1.0	Набор визуальных элементов для создания и добавления графических эффектов
QtLocation 5.6, QtPositioning 5.5	Поддержка определения местоположения
QtMultimedia 5.4	Набор элементов для работы с мультимедиа
QtNfc 5.2	Поддержка устройств ближней бесконтактной связи (Ner field communication)
QtPurchasing 1.0	Механизмы для покупки внутри приложения
QtQml 2.8	Основной модуль для разработки приложений на QML. Предоставляет язык и инфраструктуру
QtQml.Models 2.2	Набор элементов моделей данных
QtQml.StateMachine 1.0	Модуль элементов для работы с состояниями конечного автомата (state machine)
QtQuickControls 2.3 QtQuickControls.Styles 1.4	Элементы управления и стили для создания графических пользовательских интерфейсов
QtQuick.Dialogs 1.2	Набор элементов диалоговых окон
QtQuick.Extras 1.4	Дополнительные элементы управления для графических пользовательских интерфейсов
QtQuick.LocalStorage 2.0	Чтение и запись данных из/в базу данных SQLite
QtQuick.Particles	Набор небольших компонентов различного применения
QtQuick.Templates 2.3	Набор невизуальных элементов, которые используются для создания графических пользовательских интерфейсов
QtQuick.VirtualKeyboard 2.3 QtQuick.VirtualKeyboard.Styles 2.2	Модули реализации виртуальных клавиатур как для мобильных устройств, так и для настольных компьютеров
QtScxml 5.8	Реализация для создания конечных автоматов (state machines) из файлов SCXML (State Chart XML)
QtSensors 5.0	Поддержка работы с датчиками устройств, такими как, например, акселерометр, компас, барометр и т. п.

Таблица 53.1 (окончание)

Модуль	Описание
QtTest 1.1	Модуль для проведения модульного тестирования
QtWayland.Compositor 1.0	Модуль для работы с дисплейным серверным протоколом Wayland
QtWebEngine 1.5	Модуль реализации веб-браузера
QtWebSockets 1.1	Поддержка протокола для двусторонней связи между клиентским приложением и удаленным компьютером
QWebView 1.1	Модуль для отображения веб-контента без необходимости включения в программу модуля QtWebEngine. Это необходимо для сокращения размера приложения и удовлетворения требований мобильных операционных систем, придерживающихся философии, что веб-контент должен отображаться средствами мобильной операционной системы
Qt.labs.calendar 1.0	Модуль элементов календарей. Работает совместно с модулями QtQuick и QtQuick.Countrols
Qt.labs.folderlistmodel 2.1	Модуль модели локальной файловой системы
Qt.labs.handlers 1.0	Модуль для работы с касаниями и жестами пользователя
Qt.labs.platform 1.0	Набор «родных» элементов операционной системы, таких как, например, область уведомлений, диалоговые окна, доступ к путям пользовательских каталогов
Qt.labs.settings 1.0	Модуль для сохранения настроек приложения
Qt.labs.sharedimage 1.0	Модуль для сохранения памяти в случаях, когда несколько приложений используют одни и те же файлы растровых изображений

Быстрый старт

Стартовой площадкой для разработки QML служит интегрированная среда разработки Qt Creator (см. *главу 47*). Поэтому, если вы ее еще не установили Qt и не запускали на своем компьютере Qt Creator, то сделайте это сейчас (см. *приложение 1*). Давайте создадим с ее помощью проект и, по традиции, скажем «здравствуй», но на этот раз на языке QML. Для этого выберите меню **Файл** и затем пункт **Создать новый проект или файл**, и перед вашим взором предстанет диалоговое окно, показанное на рис. 53.1.

В этом диалоговом окне имеются сразу три возможности для создания Qt Quick-проекта:

- **Приложение Qt Quick** — проект может содержать код QML и C++;
- **Приложение Qt Quick Controls 2** — проект базируется на готовых элементах пользовательского интерфейса QML (см. *главу 54*);
- **Приложение Qt Canvas 3D** — трехмерное приложение на QML.

Выбираем для нашего случая второй вариант проекта: **Приложение Qt Quick** и нажимаем кнопку **Выбрать**.

В открывшемся окне (рис. 53.2), в поле **Название** вписываем имя нашего проекта: HelloWorld. Нажимаем кнопку **Продолжить**, пока не откроется окно **Определение данных**

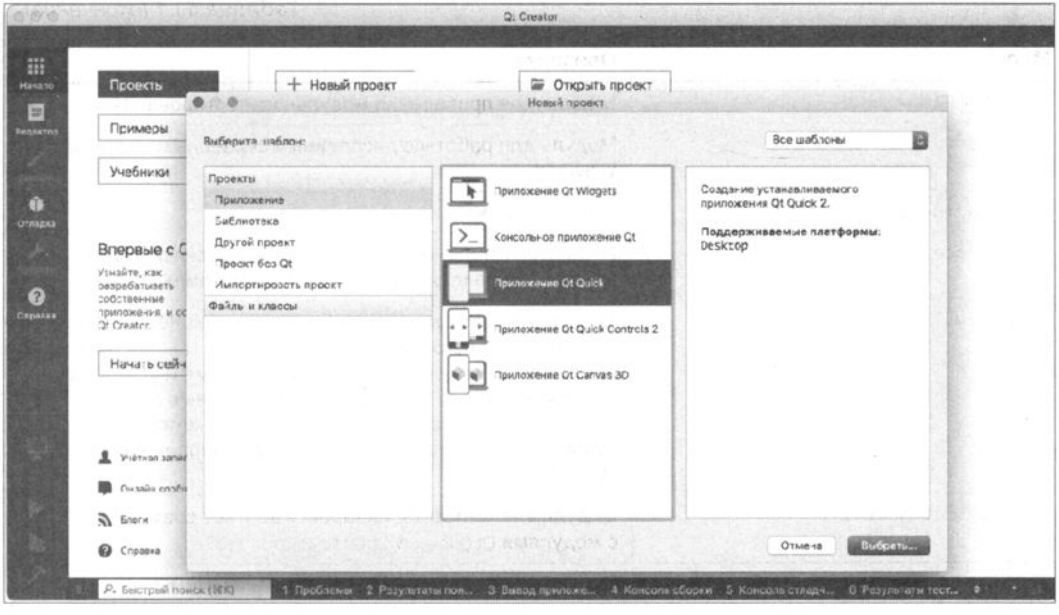


Рис. 53.1. Выбор проекта Qt Quick

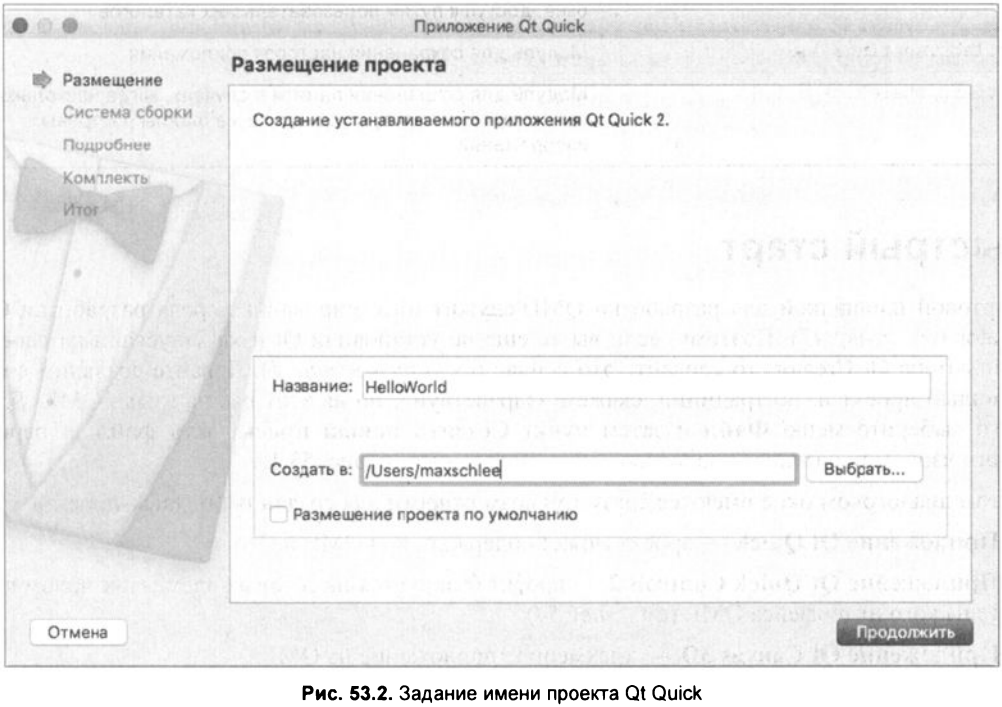


Рис. 53.2. Задание имени проекта Qt Quick

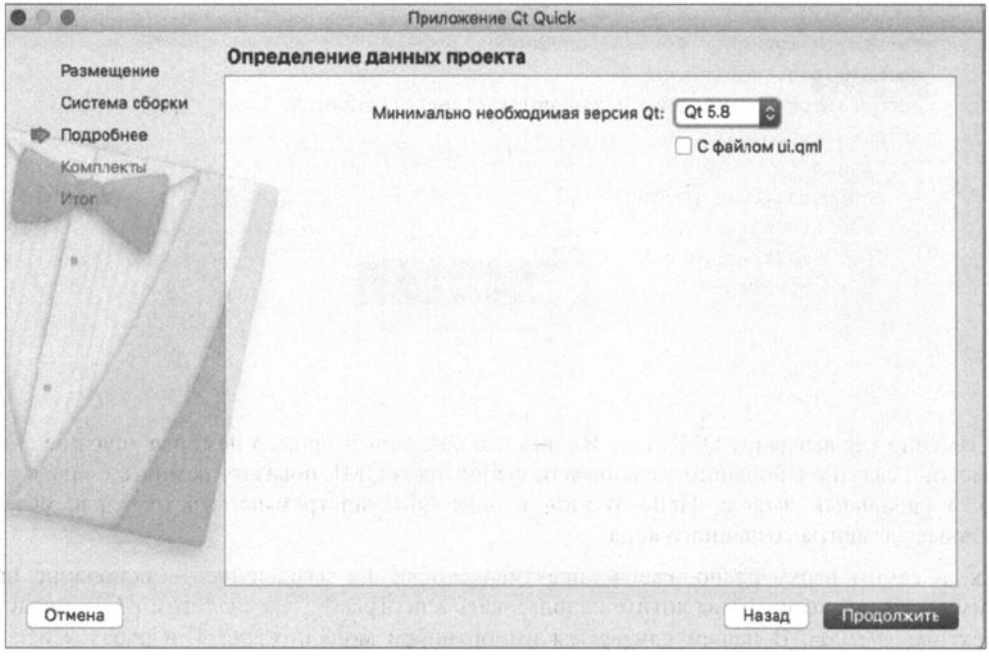


Рис. 53.3. Отключение опции для создания дополнительного QML-файла

проекта (рис. 53.3). В этом окне убираем флажок **С файлом ui.qml**, так как мы ходим иметь всего один QML-файл, и нажимаем кнопку **Продолжить**.

Следуем до завершающего окна с кнопкой **Готово** и нажимаем ее — программа Qt Creator сгенерирует QML-код листинга 53.1.

Листинг 53.1. Созданный QML-код (файл main.qml)

```
import QtQuick 2.6
import QtQuick.Window 2.2

Window {
    visible: true
    width: 640
    height: 480
    title: qsTr("Hello World")

    MouseArea {
        anchors.fill: parent
        onClicked: {
            console.log(qsTr('Clicked on background. Text: "'
                             + textEdit.text + '"')
                        )
        }
    }

    TextEdit {
        id: textEdit
```

```
        text: qsTr("Enter some text...")
        verticalAlignment: Text.AlignVCenter
        anchors.top: parent.top
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.topMargin: 20
        Rectangle {
            anchors.fill: parent
            anchors.margins: -10
            color: "transparent"
            border.width: 1
        }
    }
}
```

Да! Именно так выглядит QML-код. Видно, что созданный пример немного усложнен, что, возможно, связано с большим желанием разработчиков QML показать немного больше, чем просто банальный вывод «Hello World» в окне QML-программы. Поэтому рассмотрим основные элементы созданного кода.

Итак, в самом верху расположена директива `import`. Ее назначение — включение программного кода, который вы хотите использовать в программе. Ее аналогом в C++ является директива `include`. В нашем случае мы импортируем модули `QtQuick` и `QtQuick.Window`, чтобы иметь возможность использовать их функции и элементы.

Рядом с именем модуля `QtQuick` определен также и номер. А это значит, что будет использоваться именно та версия, которая соответствует этому номеру: в нашем случае для `QtQuick` это 2.6, а для `QtQuick.Window` — 2.2, функции и элементы из более ранних версий модулей будут нам недоступны. Такой механизм гарантирует, что поведение ваших QML-программ не будет изменено даже с появлением более новых версий, так как вы целенаправленно используете версию с указанным номером.

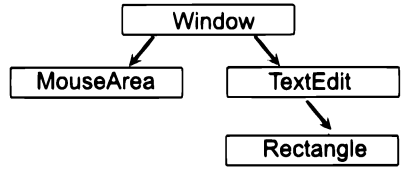


Рис. 53.4. Дерево элементов файла листинга 53.1

Пользовательский интерфейс описывается как дерево элементов с их свойствами (рис. 53.4). Каждый QML-файл должен содержать один корневой элемент. В нашем примере корневым элементом является элемент основного окна приложения `Window`, который имеет высоту 480 и ширину 640 пикселов. Это элемент основного окна приложения. Все элементы QML выполняются по указанию их типа, содержимое элемента заключено в его тело, ограниченное фигурными скобками.

Элемент содержит свойства, которые задаются именем и значением в следующем виде: *`name: value`*. В нашем примере:

- для `Window` — это `visible`, `width`, `height` и `title`;
- для элемента `TextEdit` — это `id`, `text`, `verticalAlignment`, `anchors.top`, `anchors.horizontalCenter` и `anchors.topMargin`;

- для элемента `Rectangle` — это `anchors.fill`, `anchors.margins`, `color` и `border.width`;
- для элемента `MouseArea` — это `anhors.fill` и `onClicked`.

Сами свойства могут быть дополнены при помощи JavaScript, как это сделано со свойством `onClicked` в элементе `MouseArea`. Получить доступ к элементам можно при помощи их идентификатора `id`, который должен быть уникален. К элементу предка можно получить доступ и без идентификатора — при помощи свойства-ссылки `parent`, как это делается в листинге 53.1 в элементах `MouseArea`, `TextEdit` и `Rectangle`.

Комментарии можно добавлять так же, как в C++ — для блока: `/* ... */` и для одной строки: `//`. Заметьте, что тип свойств не указывается, и это очень удобно, потому что программа будет в этом случае понятна даже для людей, не особенно разбирающихся в программировании. Точка с запятой в конце строки, как и в JavaScript, вовсе не обязательна, ее нужно использовать только в тех случаях, когда вам необходимо разделить несколько инструкций в одной строке.

Пока не будем вдаваться в подробности функционирования программы, оставим это на потом. Просто запустим ее, нажав на кнопку пуска в Qt Creator. Приложение покажет окно с заголовком **Hello World** (рис. 53.5) и поле для ввода текста. Нажатие мышью на область ввода текста переведет ее в режим редактирования текста, показанного в рамке, а нажатие на область окна произведет вывод на консоль сообщения следующего вида:

qml: Clicked on background. Text: "Enter some text..."

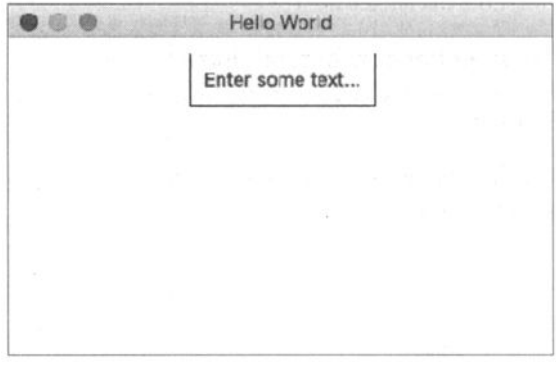


Рис. 53.5. Запущенный Qt Quick-проект

Теперь вернемся снова к Qt Creator. Эта интегрированная среда разработки предоставляет отличную поддержку для Qt Quick-проектов. Конечно, можно использовать и текстовый редактор, но вы рискуете остаться без массы удобных функций, предназначенных специально для работы с QML. Вот только лишь некоторые из них:

- запуск QML-файлов;
- подсвечивание идентификационных номеров элементов, о которых я расскажу в *главе 54*;
- поддержка расцветки синтаксиса QML;
- встроенный отладчик, который позволяет наблюдать за значениями QML-свойств и переменными JavaScript в процессе выполнения;
- автоматические подсказки для автоматического дополнения кода. Для получения подсказки можно также нажать комбинации клавиш:

- <Alt>+<Space> — покажет все доступные свойства элемента;
- <Ctrl>+<Space> — покажет все свойства, которые можно изменить;

♦ визуальные инструменты, с помощью которых вы можете в интерактивной форме, не выходя из редактора исходного кода программы, изменять параметры для цветов, градиентов, шрифтов, растровых изображений и смягчающих кривых (см. *главу 56*);

♦ визуальный интерактивный редактор (рис. 53.6), с помощью которого можно изменять QML-программы в интерактивной форме. Этот редактор рекомендуется использовать для того, чтобы еще больше повысить скорость разработки QML-программ. Вы можете перетаскивать в форму элементы управления, изменять значения их свойств. Для того чтобы войти в этот режим редактирования, нужно нажать кнопку с изображением карандаша, которая расположена на панели слева. Входным и выходным форматом для этого редактора является исходный QML-код, поэтому вы можете свободно переключаться между текстовым и визуальным режимами редактирования, работая с одним и тем же кодом. Если вдруг будет необходимо производить работу над некоторыми или всеми QML-файлами только в интерактивном редакторе, то таким файлам необходимо будет дать расширение ui.qml.

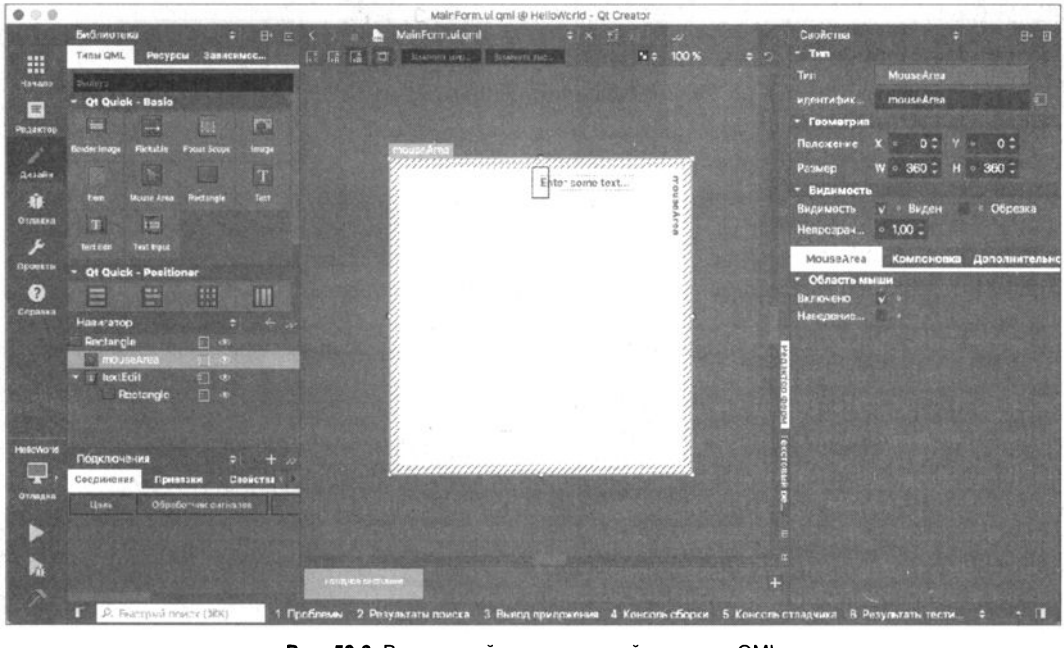


Рис. 53.6. Визуальный интерактивный редактор QML

Qt Creator — это очень полезное средство, которое будет постоянно опекать вас и следить за тем, чтобы вы не допустили погрешностей и ошибок в создании QML-кода. Например, если вы забудете директиву `import`, то сразу это заметите, потому что Qt Creator предупредит вас об этом и покажет целую серию неопределенных типов. Поэтому смело используйте интегрированную среду разработки Qt Creator в своей работе!

Использование JavaScript в QML

Как уже было сказано ранее, QML — это описательный язык и для того, чтобы реализовать логику программы, QML непригоден. Но в программе может понадобиться использовать циклы или произвести какие-либо вычисления. Что делать? Именно для этого в язык QML встроен движок JavaScript.

Для ясности рассмотрим следующий код. В нем для вычисления первым делом задействован JavaScript, и только после этого вычисленное значение будет присвоено свойству `width` элемента `Rectangle`:

```
Rectange {
    width: parent.width / 5
}
```

Если выражение JavaScript содержит две или более строк, то оно помещается в фигурные скобки. В следующем примере мы присваиваем вычисленное значение промежуточной переменной, отображаем ее значение на консоли и затем возвращаем его для присвоения свойству `width`:

```
Rectangle {
    width: {
        var w = parent.width / 5
        console.log("Current width:" + w)
        return w
    }
}
```

В только что рассмотренном отрывке программы мы отображали сообщение и переменную `w` из JavaScript при помощи `console.log()`. В JavaScript существует целый ряд возможностей для отображения информации, которые могут очень пригодиться для оптимизации и отладки QML-программ. Некоторые из этих возможностей приведены в табл. 53.2.

Таблица 53.2. *Некоторые возможности отображения информации на консоли из JavaScript*

Метод	Описание
`console.assert()`	Проверяет переданное логическое выражение. Если оно равное `true`, то отображается цепочка вызовов QML-программы и дополнительное сообщение, которое может быть передано в качестве второго аргумента
`console.count().`	Отображает количество раз, которое был вызван код
`console.debug()`, `console.log()`, `console.info()`, `console.error()`, `console.warn()`, `print()`	Служат для вывода информационных сообщений, которые могут быть использованы для отладки программы
`console.profile()`, `console.profileEnd()`	Используются для оптимизации кода. Позволяют включить и отключить сообщения профилировщика
`console.time()`, `console.timeEnd()`	Позволяют замерить и отобразить время, прошедшее с момента вызова метода `time()` до вызова метода `timeEnd()`
`console.trace()`	Отображает цепочку вызовов javascript, ограниченную 10 вызовами

Функции JavaScript можно интегрировать в сами элементы. В следующем примере мы реализуем в элементе функцию определения максимума и используем ее:

```
Rectangle {
    function maximum(a, b)
    {
        return a > b ? a : b;
    }
    width: maximum(parent.width, 100)
}
```

Для более объемных по коду функций JavaScript можно создать отдельные файлы и импортировать их в QML-файле элемента. Возьмем в качестве примера функцию вычисления максимума и поместим ее в отдельный файл myfunctions.js. Теперь воспользуемся ей из QML-файла:

```
import "myfunctions.js" as MyScripts
Rectangle {
    function maximum(a, b)
    {
        return a > b ? a : b;
    }
    width: MyScripts.maximum(parent.width, 100)
}
```

Стоящее в директиве `import` после ключевого слова `as` имя `MyScripts` является своеобразным идентификатором, с помощью которого мы можем получить доступ ко всем функциям, определенным в файле. Этот момент требует пояснения — представьте себе, что вы импортируете два файла с функциями, в которых встречаются две функции с одинаковыми именами. Идентификатор убережет вас от получившейся путаницы: вы можете представить его с неким локальным пространством имен, который вы задаете для использования в каждом отдельном QML-файле сами.

Внутри самого файла myfunctions.js, где расположены JavaScript-функции, самой первой строкой нелишне будет указать следующую директиву:

```
.pragma myfunctions
```

Это нужно для того, чтобы файл не включался более одного раза и не растрачивал понапрасну драгоценные ресурсы QML.

Использовать JavaScript в QML-программах нужно обдуманно, для сложных алгоритмов рекомендуется все же отдавать предпочтение C++ (см. *главу 60*) — это поможет сделать QML-программу более быстрой и экономнее использовать ресурсы компьютера.

Резюме

Технология Qt Quick представляет собой средство для создания пользовательского интерфейса нового поколения с поддержкой анимационных эффектов. Времена, когда дизайнеры создавали картинки, а программисты реализовывали по ним код, уходят в прошлое. Набор технологий Qt Quick позволяет дизайнерам и разработчикам программного кода работать вместе настолько тесно, как этого не было никогда ранее. В основе технологии Qt Quick лежит язык QML. Этот язык описательный, то есть он в основном описывает то, что должно

быть, а не как это должно быть запрограммировано. Язык QML определяет пользовательский интерфейс, используя элементы и свойства.

Чтобы овладеть языком QML, желательно иметь минимальное понимание основ программирования, хотя и без них тоже можно довольно быстро в нем освоиться. Включение в состав технологии Qt Quick языка JavaScript делает ее доступным для понимания подавляющего большинства веб-дизайнеров и расширяет язык QML возможностью реализовывать алгоритмы и производить вычисления.

Язык QML очень хорошо оптимизирован, уверенно функционирует на мобильных устройствах и может быть расширяем из C++. Кроме того, QML обладает массой возможностей, которые предоставляет библиотека Qt, — такими как: метаобъектная модель, свойства и сигналы.

Использование Qt Quick сокращает время, необходимое для разработки приложений, и позволяет быстро создавать полностью функциональные прототипы, которые можно тестировать в реальных условиях и также на мобильных устройствах.

Интегрированная среда разработки Qt Creator обладает целым рядом удобных средств, направленных на то, чтобы сделать создание QML-программ быстрым и удобным не только программистам, но и дизайнерам.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/53-510/** или с помощью следующего QR-кода (рис. 53.7):

Рис. 53.7. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 54

Элементы

Каждая мечта тебе дается вместе с силами, необходимыми для ее осуществления.
Однако, тебе, возможно, придется ради этого потрудиться.
Ричард Бах

Элементы — это «строительный материал» приложений, выполненных с помощью языка QML. Они делятся на две группы. Первая группа — это *визуальные элементы*. Вторая группа элементов — это *объекты*, то есть те элементы, которые не обладают визуальным представлением и не выполняют никакой вспомогательной роли. Эти объекты делятся на группы:

- объекты для размещения элементов: `Column`, `Row`, `Grid`, `Flow`, `Positioner` и т. д. (см. *главу 55*);
- объекты-инструменты: `Timer`, `Loader`, `Connections`, `WorkerScript`, `Binding` и т. д.;
- объекты для анимации: `PropertyAnimation`, `NumberAnimation`, `ColorAnimation`, `ParallelAnimation` и т. д. (см. *главу 58*);
- объекты моделей: `ListModel`, `ListElement`, `XmlModel`, `VisualDataModel`, `VisualItemModel` и т. д. (см. *главу 59*);
- объекты пользовательского ввода: `MouseArea`, `DropArea`, `MultiPointTouchArea`, `PinchArea`, `MouseEvent`, `KeyEvent` и т. д. (см. *главу 57*).

В этой главе мы остановимся на группе визуальных элементов.

Визуальные элементы

Чаще всего используются следующие визуальные элементы:

- `Item` — представляет собой базовый тип для всех элементов. Элемент `Item` можно сравнить с классом `QWidget` в Qt. Хоть элемент `Item` невидим, он имеет позицию, ширину и высоту. Обычно он служит для того, чтобы объединить в группу видимые элементы, а также часто применяется как элемент высшего уровня, то есть основное окно;
- `Rectangle` — представляет заполненную прямоугольную область с необязательным контуром;
- `Image` — представляет растровое изображение;
- `BorderImage` — представляет растровое изображение с контуром;
- `Text` — позволяет добавлять форматированный текст;

- `TextEdit`, `TextInput` — ввод и отображение текста (см. *главу 57*);
- `Window` — элемент главного окна приложения;
- `WebView` — элемент, предназначенный для отображения веб-содержаний.

В визуальных элементах можно выделить отдельную подгруппу — это *представления* (их мы более подробно рассмотрим в *главе 59*):

- `ListView` — осуществляет представление в виде списка;
- `GridView` — осуществляет представление в виде таблицы;
- `PathView` — самое мощное и вместе с тем самое сложное из имеющихся представлений. Оно позволяет располагать элементы вдоль произвольного пути и контролировать их масштаб, прозрачность и другие атрибуты.

Заметьте, что все элементы, согласно конвенции, всегда начинаются с заглавных букв. Создаются они одинаково. В QML так же, как и в Qt, существует механизм объектной иерархии, и элементы тоже могут иметь предков и потомков. В качестве примера создадим элемент `Rectangle` как потомка элемента `Item` (листинг 54.1). Зададим элементу прямоугольника местоположение, размеры и цвет (рис. 54.1).

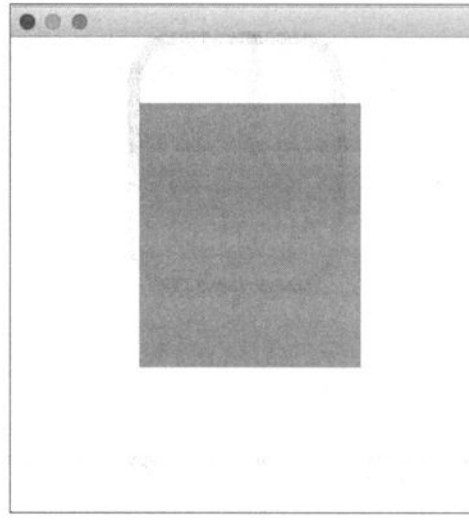


Рис. 54.1. Элемент прямоугольника как потомка элемента основного окна

В листинге 54.1 элемент `Item` является элементом верхнего уровня и выполняет роль основного окна, начальные размеры которого 360×360 задаются при помощи свойств `width` и `height`. Элемент `Rectangle` является его потомком и отображается относительно и внутри предка на позиции `x: 100` и `y: 50` (свойства `x` и `y`) с размерами 170×200 (свойства `width` и `height`). Свойством `color` ему присваивается цвет заполнения — темное хаки: `"darkkhaki"`.

Листинг 54.1. Отображение потомка

```
import QtQuick 2.8
Item {
    width: 360
    height: 360
```

```
    Rectangle {
        color: "darkkhaki"
        x: 100
        y: 50
        width: 170
        height: 200
    }
}
```

Как вы заметили из листинга 54.1, свойства — это очень важная составляющая языка QML, поэтому далее мы рассмотрим их более подробно.

В элементе прямоугольника можно при помощи свойств `border.color`, `border.width`, `radius` и `smooth` задать соответственно: цвет рамки (бордюра), ее толщину, радиус закругления ее углов и включить режим сглаживания отображения (листинг 54.2).

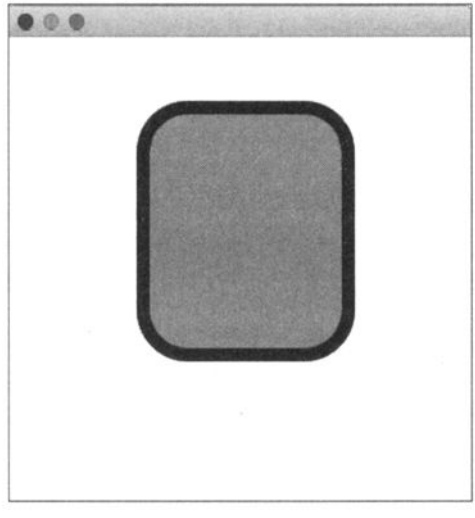


Рис. 54.2. Элемент прямоугольника с закруглениями и рамкой

Листинг 54.2. Отображение потомка

```
import QtQuick 2.8
Item {
    width: 360
    height: 360
    Rectangle {
        color: "darkkhaki"
        x: 100
        y: 50
        width: 170
        height: 200
        border.color: "black"
        border.width: 10
```

```
        radius: 40
        smooth: true
    }
}
```

Свойства элементов

Мы уже успели убедиться в том, что свойства служат для изменения поведения и внешнего вида элементов. Все элементы содержат стандартные свойства и могут также быть расширены дополнительными собственными свойствами. К стандартным свойствам относятся:

- свойства для позиционирования: `x`, `y`, `z`, `position`;
- свойства для задания и получения размеров: `width`, `height`, `implicitWidth`, `implicitHeight`;
- свойства для графических операций и преобразований: `rotation`, `scale`, `clip`, `transform`, `transformOrgin`, `antialiasing`, `smooth`;
- серия свойств фиксации: `anchors` (см. *главу 55*);
- свойства для ссылок на элемент: `id`, `parent`;
- свойства доступности/недоступности: `enabled`;
- свойства установки фокуса: `focus`;
- свойства управления видимостью: `visible`, `opacity`, `visibleChildren`;
- свойства управления состояниями и переходами: `states`, `state`, `transitions` (см. *главу 58*);
- серия свойств для работы со слоями: `layer`;
- свойства для работы с дочерними элементами: `children`, `childrenRect`.

Свойства позиционирования и задания размера мы уже рассмотрели в листинге 54.1. Свойствам фиксации посвящена следующая глава этой книги.

Очень важна группа свойств для ссылок на элемент. В языке C++ вы имеете указатели, а в QML указателей нет, и вы не смогли бы получать доступ к элементам, если бы не было этих свойств. Ссылаться на элементы обычно бывает нужно для позиционирования элементов, а также для использования и изменения их свойств.

Первое свойство идентификации — `id` — задает элементу имя, с помощью которого можно будет ссылаться на этот элемент. Если мы хотим сослаться на элемент, то при помощи свойства `id` обязаны дать ему имя.

> ***Имена для идентификаторов***
>
> Имена для идентификаторов (для свойств `id`) должны начинаться либо с маленькой буквы, либо со знака подчеркивания и могут содержать только буквы, числа и знаки подчеркивания.

Второе свойство — `parent` — позволяет нам сослаться на элемент предка.

В следующем примере (листинги 54.3 и 54.4) мы продемонстрируем использование свойств `parent` и `id` с помощью двух элементов — прямоугольников, один из которых будет ссылаться на `parent`, а другой на `id` другого прямоугольника (рис. 54.3).

В листинге 54.3 мы создаем два элемента прямоугольников. Первому присваиваем идентификатор `redrect` (свойство `id`), задаем красный цвет (свойство `color`), позицию 0, 0 (свойст-

ва x и y). Ширина и высота (свойства width и height) станут вычисляться в зависимости от ширины и высоты предка и будут в два раза меньше. Для этого мы при помощи свойства parent ссылаемся на эти свойства и присваиваем их. Второму прямоугольнику присваиваем зеленый цвет (свойство color) и присваиваем свойства позиции x и y свойствам размеров первого прямоугольника, а для того чтобы получить доступ к этим свойствам, используем заданный идентификатор redrect. Так же мы поступаем с его размерами и присваиваем размеры первого прямоугольника (свойства width и height). Теперь начинается самое интересное — попробуйте изменить при помощи мыши размеры окна, и вы увидите, что вместе с ним изменяются размеры обоих прямоугольников. Каким же образом это так получается? Такое «волшебство» называется *связыванием свойств*, оно происходит автоматически тогда, когда QML «видит», что значения свойств одного объекта изменяются, и после этого осуществляет связывание со свойствами, использующими это значение.

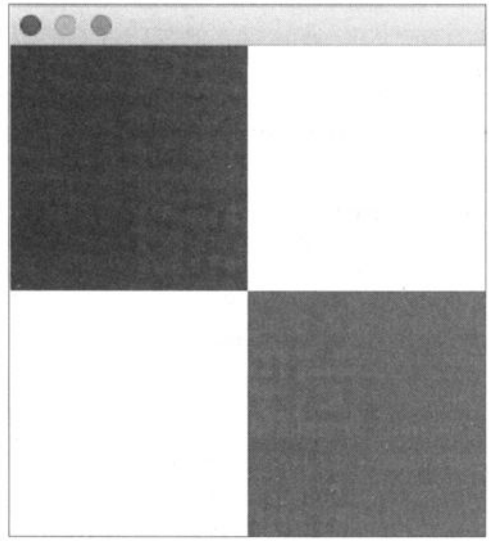


Рис. 54.3. Ссылки по свойствам id и parent

Это не просто свойства, а нечто особенное, потому что они находятся под наблюдением. QML наблюдает за ними и выполняет уведомления всякий раз, когда они изменяются, и вы так же можете сделать связывание с ними отдельного кода, который при изменениях свойства будет отрабатывать необходимые действия в специальных свойствах типа on*X* (onWidthChanged, onHeightChanged и т. п.). Для реакции на изменение ширины и высоты окна можно, например, вывести их текущее значение на консоль.

Листинг 54.3. Использование свойств parent и id

```
import QtQuick 2.8
Item {
    width: 360
    height: 360
    Rectangle {
        id: redrect;
        color:  "red"
```

```
        x: 0
        y: 0
        width: parent.width / 2
        height: parent.height / 2
    }

    Rectangle {
        color: "green";
        x: redrect.width
        y: redrect.height
        width: redrect.width
        height: redrect.height
    }
}
```

Листинг 54.4 использует свойства `onWidthChanged` и `onHeightChanged` для отображения текущих размеров. Эти свойства вызывают код при каждом изменении ширины и высоты. Обратите внимание, что в блоках этих свойств задействован код на JavaScript, вследствие чего вывод значений будет виден на консоли.

Листинг 54.4. Вывод текущей ширины и высоты при изменении размеров окна

```
import QtQuick 2.8
Item {
    width: 200
    height: 200
    Rectangle {
       width: parent.width
       height: parent.height
       onWidthChanged: {
           console.log("width changed:" + width)
       }
       onHeightChanged: {
           console.log("height changed:" + height)
       }
    }
}
```

Собственные свойства

Как уже говорилось ранее, мы можем не только использовать уже существующие свойства, но и добавлять свои собственные. Этой цели служит ключевое слово `property`, которое имеет следующий синтаксис:

```
property <тип> <имя>[: <значение>]
```

Первым идет тип, за ним следует имя свойства и в последнюю очередь — необязательное значение либо выражение. Свойства строго типизированы и свойствам одного типа не могут быть присвоены значения других типов. В табл. 54.1 приведены некоторые типы для свойств QML.

Таблица 54.1. *Типы свойств QML*

Тип	Описание
`bool`	Логический тип, принимает значения `false` или `true`
`double`	Число двойной точности (double precision)
`enumeration`	Тип перечисления
`int`	Целочисленный тип (например, 400)
`list`	Список объектов
`real`	Числа с плавающей запятой (например, 1.7)
`string`	Строка (например «Hello QML»)
`time`	Время в формате HH:MM:SS (например, 09:23:01)
`url`	Строка, соответствующая стандарту URL (Uniform Resource Locator, единый указатель ресурсов)
`var`	Тип, эквивалентный `QVariant`, используется для переменных любого типа

В табл. 54.2 приведены некоторые типы, которые доступны из модуля `QtQuick`.

Таблица 54.2. *Типы Qt Quick 2.8*

Тип	Описание
`color`	Тип для обозначения цвета
`date`	Дата в формате YYYY-MM-DD
`font`	Тип шрифта
`matrix4x4`	Матрица из четырех строк и четырех столбцов
`point`	Тип точка. Состоит из пары значений: `x` и `y`
`quaternion`	Состоит из четырех атрибутов: `scalar`, `x`, `y` и `z`
`rect`	Тип прямоугольник. Состоит из четырех значений: `x`, `y`, `width` (ширина) и `height` (высота)
`size`	Тип размер. Состоит из двух значений: `width` и `height`
`Vercor2d`	Состоит из `x`-, `y`-координат
`vercor3d`	Состоит из `x`-, `y`- и `z`-координат
`vector4d`	Состоит из `x`-, `y`- `z`- и `w`-координат

В следующем примере (листинг 54.5) мы расширим один из элементов новыми свойствами, считаем и покажем их значения из другого элемента: `Text`, предназначенного для отображения текста (рис. 54.4).

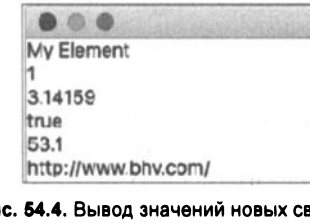


Рис. 54.4. Вывод значений новых свойств

В листинге 54.5 мы присваиваем идентификатор `myelement` элементу `Item` и расширяем его пятью новыми свойствами разных типов. Из элемента `Text` мы обращаемся к их значениям и отображаем их присвоением текстовой строки свойству `text`. Каждое из новых свойств после определения тоже будет иметь соответствующее свойство типа `onX`, которое станет реагировать на каждое изменение свойства. Для примера, в элемент с идентификатором `myelement` можно внести свойство `onConditionChanged`, которое будет реагировать на изменение значений свойства `condition`:

```
onConditionChanged: {
    //do something
}
```

Листинг 54.5. Дополнение элемента свойствами

```
import QtQuick 2.8

Item {
    width: 200
    height: 100

    Item {
        id: myelement
        property string name: "My Element"
        property int ver: 1
        property real pi: 3.14159
        property bool condition: true
        property var variant: 53.1
        property url link: "http://www.bhv.com/"
    }

    Text {
        x: 0
        y: 0
        text: myelement.name + "<br>"
              + myelement.ver + "<br>"
              + myelement.pi + "<br>"
              + myelement.condition + "<br>"
              + myelement.variant + "<br>"
              + myelement.link
    }
}
```

Все новые свойства вместе с их `onX`-свойствами будут также доступны в Qt Creator для автоматического дополнения.

Все объявленные свойства элементов открыты для доступа другим элементам. Если это нежелательно, и требуется инкапсулировать данные и скрыть их от доступа извне, то можно поступить следующим образом: указать в элементе `QtObject` все свойства, которые необходимо закрыть для доступа извне, и обращаться к ним при помощи идентификатора элемента `QtObject`. Вот как это может выглядеть в коде, в котором закрыли два свойства: `nX` и `nY`:

```
Rectangle {
...
    QtObject {
        id: priv
        readonly property int nX: 23
        readonly property int nX: 50
    }
    x: priv.nX
    y: priv.nY
...
}
```

Создание собственных элементов

Собственные элементы определяются в отдельных файлах в соотношении один к одному. То есть, для каждого нового элемента нужен свой отдельный файл. Новые элементы используются таким же образом, как и стандартные элементы. Элементы, расположенные в одном и том же каталоге, автоматически доступны друг для друга.

Создадим (листинги 54.5 и 54.6) собственный элемент для отображения текста, который обладает свойством цвета, рамки и другими свойствами элемента `Rectangle` (рис. 54.5). Сначала сгенерируем новый проект с помощью Qt Creator. Потом создадим и добавим в него новый файл TextField.qml, расположенный в том же каталоге, где расположена основная программа main.qml.

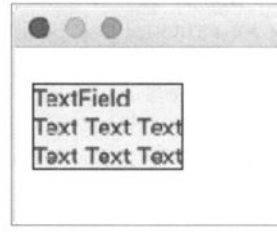


Рис. 54.5. Собственный элемент отображения текста с фоном и рамкой

В листинге 54.6 реализован наш новый элемент для отображения текста на базе элемента `Rectangle`, который является предком для элемента `Text`. Все его свойства будут доступны извне. Недоступными будут только свойства элемента потомка `Text`. Нам бы очень хотелось предоставить возможность изменять текст со стороны. Этого можно добиться, определив синоним к свойству в элементе `Rectangle`.

Синоним (alias) — это механизм для опубликования свойств под другим именем на более высоком уровне иерархии. Таким образом им можно придать возможность быть изменяемыми извне. В листинге мы определяем свойство-синоним `text` в элементе `Rectangle` и используем идентификатор `txt`, чтобы связать его со свойством `text` элемента `Text`. Так мы сделали это свойство доступным для внешнего изменения.

Обратите внимание, что свойство `width` элемента `Rectangle` не нуждается в синониме. Поскольку это элемент верхнего уровня, к его «родным» свойствам можно обратиться извне. В синонимах нуждаются только свойства вложенных элементов — если вы хотите сделать их доступными для других элементов.

Теперь нам нужно определиться с размерами элемента. Очевидно, что размеры изменяются в зависимости от содержимого текста и рассчитываются этим элементом. Значит, размер элемента прямоугольника должен соответствовать размеру текстового элемента. Поэтому мы присваиваем свойствам элемента `Rectangle` значения свойств элемента `Text` и устанавливаем нулевую позицию (свойства `x` и `y`) для текстового элемента, относительно элемента предка `Rectangle`.

Листинг 54.6. Элемент отображения текста с фоном и рамкой (Файл TextField.qml)

```
import QtQuick 2.8
Rectangle {
    property alias text: txt.text
    property string name: "TextField"

    width: txt.width
    height: txt.height

    Text {
        id: txt
        x: 0
        y: 0
    }
}
```

В листинге 54.7 мы используем наш новый элемент `TextField`. Поскольку содержащий его файл TextField.qml находится в том же каталоге ресурса, что и файл main.qml, он автоматически доступен. Расположим наш элемент на позиции `x: 10` и `y: 20` главного окна (свойства `x` и `y`). Установим цвет фона желтым (свойство `color`). Текст устанавливаем при помощи введенного нами нового свойства `text` — заметьте, что разницы в присвоении нет никакой, и только мы знаем, что это новое свойство. И, наконец, устанавливаем толщину рамки, равную одному пикселу (свойство `border.width`).

Листинг 54.7. Использование нового элемента отображения текста (файл main.qml)

```
import QtQuick 2.8
Item {
    width: 150
    height: 100

    TextField {
        x: 10
        y: 20
        color: "yellow"
        text: "Text Text Text<br>Text Text Text"
        border.width: 1
    }
}
```

Создание собственных модулей

Для повторного использования коллекции QML-элементов их можно объединять в отдельные модули. Эти модули можно импортировать для использования в QML-программах с помощью уже знакомой нам директивы `import`. Средства QML позволяют назначать модулям номера версий. Благодаря этому механизму, импортируется заданная версия модуля, что гарантирует ожидаемый результат в использовании элементов этого модуля.

Для того чтобы создать модуль, нужно создать каталог и разместить в нем QML-файлы элементов и файл описания, который должен называться qmldir. В этом файле должны быть указанные все входящие в модуль QML-файлы с указанием номеров их версий. В листинге 54.8 приводится отрывок из файла описания модуля.

Листинг 54.8. Файл описания модуля (файл qmldir)

```
# Module description "qmldir"
module QtBookControls
ColorPicker 1.1 ColorPicker-1.1.qml
ColorPicker 1.0 ColorPicker-1.0.qml
ToolTip 1.1 ToolTip-1.1.qml
ToolTip 1.0 ToolTip-1.0.qml
```

Как видно из листинга 54.8, комментарии обозначаются в этом файле символом #. Обратите внимание, как организуется поддержка множественных версий в этом файле: более новая версия должна всегда указываться выше предыдущей версии.

Использовать каталог с компонентами модуля в программе QML мы можем следующим образом: `import "QtBookControls"`.

Или если каталог расположен уровнем выше, то так: `import "../QtBookControls"`.

Можно также задать и идентификатор для модуля: `import "QtBookControls" as QBC`.

Этот подход можно также использовать, если вы намерены поместить свой модуль с компонентами на удаленном сервере и получать доступ к нему по сети, например через Интернет.

Динамическое создание элементов

Язык QML предоставляет удобный механизм, который позволяет избежать повторного копирования кода элементов. Это делает программы более компактными и легко читаемыми. В книге мы будем иногда им пользоваться, поэтому постарайтесь понять принцип его действия. Этот элемент, как и следует из его назначения, называется *повторителем* (Repeater). Он позволяет производить элементы, используя любой тип модели данных, которая может быть указана статически в виде числа, массива или набора элементов JSON-типа. То есть, в данном конкретном случае, модель является инструкцией к созданию элементов, а содержащийся внутри повторителя элемент представляет собой шаблон, по образцу которого будут создаваться элементы. Вот так, например, можно создать 10 элементов `Rectangle`:

```
Repeater {
    model: 10
    Rectangle {}
}
```

А вот так можно создать из массива строк 5 элементов `Text`, каждый из которых будет отображать свой отдельный элемент массива, доступ к которому получается при помощи свойства `modelData`:

```
Repeater {
    model: ["one", "two", "three", "four", "five"]
    Text {text: modelData}
}
```

Думаю, идея в применении элемента повторителя теперь понятна. Далее мы рассмотрим этот элемент на практическом примере и заодно познакомимся с элементом `Flickable`.

Элемент *Flickable*

Элемент `Flickable` напоминает известный из библиотеки Qt класс `QScrollArea` (см. *главу 5*). Так же, как и этот класс, он предназначен для показа элементов или их частей, в том случае, если размеры превышают размеры области показа. Перемещать части элемента для показа можно при помощи пальца на сенсорном экране или мыши на компьютере. Само перемещение сопровождается также эффектом анимации.

Пример использования этого элемента, а также элемента повторителя `Repeater`, показан в листинге 54.9. Мы создаем с помощью повторителя 5 элементов `Rectangle` разных цветов и размещаем их внутри элемента `Flickable`. Размеры элементов `Rectangle` превышают размеры самого элемента `Flickable`, поэтому показана только часть изображения. Для того чтобы увидеть другую его часть, необходимо переместить область просмотра в нужное место (рис. 54.6).

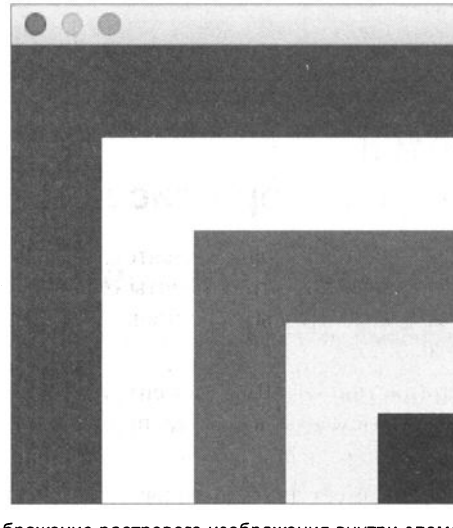


Рис. 54.6. Отображение растрового изображения внутри элемента `Flickable`

В листинге 54.9 мы задаем элементу `Flickable` размеры 250×250. Для содержимого мы задаем размеры 500×500 при помощи свойств `contentWidth` и `contentHeight`. Внутри элемента `Flickable` создаем при помощи повторителя 5 элементов `Rectangle`. В качестве модели указан массив из строковых обозначений цветов. Каждый создаваемый элемент получает цветовое значение из этого массива при помощи свойства `modelData` и присваивает его свойст-

ву color. Мы можем также узнать и текущий индекс элемента массива нашей модели — это достигается при помощи свойства index: мы используем индекс, чтобы уменьшить размеры каждого последующего элемента в свойствах width и height. Для свойств управления позицией x и y мы вычисляем и присваиваем значения расположения в середине области содержимого элемента Flickable.

Листинг 54.9. Отображение потомка

```
import QtQuick 2.8

Flickable {
    id: view
    width: 250
    height: 250
    contentWidth: 500
    contentHeight: 500

    Repeater {
        model: ["red", "white", "green", "yellow", "blue"]
        Rectangle {
            color: modelData
            width: view.contentWidth - index * 100
            height: view.contentHeight - index * 100
            x: view.contentWidth / 2 - width / 2
            y: view.contentHeight / 2 - height / 2
        }
    }
}
```

Готовые элементы пользовательского интерфейса

Технология Qt Quick предоставляет целый ряд элементов, специально ориентированных для создания пользовательского интерфейса. Эти элементы способны полностью заменить виджеты, описанные в *части II* книги. Чтобы использовать готовые элементы, необходимо включить в QML-файл модуль QtQuick.Controls.

Рассмотрим простой пример (листинг 54.10) с элементами кнопок различного типа. Кнопки — это самые используемые элементы в любом приложении. На рис. 54.7 мы видим (*сверху вниз*):

- кнопку флажка — она обычно имеет два состояния: включено или выключено;
- кнопку задержки. Процесс задержки показывается при нажатии на саму кнопку, и при достижении конечного значения задержки кнопка считается нажатой;
- кнопку переключателя. Группы из этих кнопок представляют собой взаимоисключающие состояния, поэтому в программе их должно быть минимум две в группе;
- закругленную кнопку — представляет собой обычную кнопку нажатия с закругленными углами;

- кнопку выключателя — ее принято использовать на мобильных устройствах вместо кнопки флажка;
- кнопку инструментов — эта кнопка специально предназначена для расположения на панели инструментов;
- обычную кнопку нажатия — эта кнопка идет последней и расположена в самом низу. Нажатие на нее осуществляет выход из приложения.

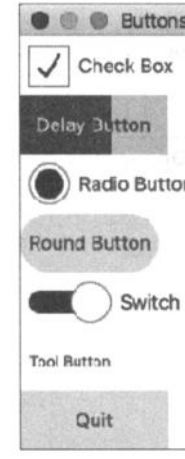


Рис. 54.7. Окно приложения с различными типами кнопок

В листинге 54.10 мы включаем модуль `QtQuick.Controls`. Затем нам нужно создать элемент окна приложения `ApplicationWindow`. Его размеры будут заданы в соответствии с размерами области элемента вертикального размещения `Column`, в котором будут расположены кнопки (элементам размещения посвящена *глава 55*). Чтобы сделать видимыми виджеты верхнего уровня, нужно было вызвать метод `show()`, но в QML мы просто устанавливаем свойству `visible` значение `true`. Заголовок окна устанавливается при помощи свойства `title`.

Затем в элементе размещения `Column` помещаем 7 элементов кнопок и присваиваем им с помощью свойства `text` надписи с именами типов кнопок — кроме последней кнопки: ей мы присваиваем надпись **Quit** и в свойстве обработки события нажатия `onClicked` устанавливаем код для вызова функции завершения приложения `Qt.quit()`.

Листинг 54.10. Элемент окна приложения с кнопкой (файл main.qml)

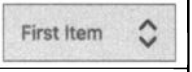
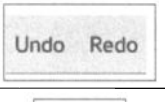
```
import QtQuick 2.8
import QtQuick.Controls 2.2

ApplicationWindow {
    width: buttons.width
    height: buttons.height
    visible: true
    title: "Buttons"

    Column {
        id: buttons
        CheckBox{text: "Check Box"}
        DelayButton{text: "Delay Button"}
        RadioButton{text: "Radio Button"}
        RoundButton{text: "Round Button"}
        Switch{text: "Switch"}
        ToolButton{text: "Tool Button"}
        Button{text: "Quit"; onClicked: Qt.quit()}
    }
}
```

В табл 54.3 приведены еще некоторые из интересных элементов управления, которые предоставляет модуль `QtQuick.Controls`.

Таблица 54.3. *Элементы пользовательского интерфейса QtQuick.Controls 2.2*

Элемент	Описание	Вид
`BusyIndicator`	Элемент для отображения состояния занятости приложения. Используется в тех случаях, когда вычислить момент завершения операции не представляется возможным	
`ComboBox`	Элемент выпадающего списка. Предоставляет возможность выбора одного элемента из нескольких	
`Dial`	Элемент установщик. Для установки значений его требуется вращать. Он не задуман для установки точных значений	
`Label`	Элемент надписи. Представляет собой текстовое поле, которое служит для отображения поясняющего текста	
`PageIndicator`	Элемент индикатора страниц. Информирует об индексе отображаемой в текущий момент страницы	
`ProgressBar`	Элемент индикации выполнения. Демонстрирует процесс выполнения операции тем, что заполняет область слева направо. Полное заполнение области сигнализирует о завершении операции	
`RangeSlider`	Элемент для установки диапазона значений. Элемент содержит два элемента ползунка, с помощью которых задаются два значения, находящиеся между начальным и конечным значениями диапазона, установленного в этом элементе	
`Slider`	Элемент ползунок. Используется для установки в заданном диапазоне значений, которые не требуют высокой точности	
`SpinBox`	Элемент счетчика. Предоставляет пользователю доступ к ограниченному диапазону чисел. Используется для установки значений с высокой точностью	
`ToolBar`	Элемент панели инструментов. Он предназначен для размещения элементов в нижней либо в верхней зоне главного окна приложения	
`Tumbler`	Элемент предоставления выбора в виде столбцов. Может состоять из одного или нескольких столбцов, при использовании которых нужные значения помещаются в средний ряд	

Область элемента основного окна приложения `ApplicationWindow` делится на три зоны: верхнюю (`header`), нижнюю (`footer`) и рабочую, которая располагается в середине. В следующем примере (рис. 54.8) эти зоны отчетливо видны. В верхней зоне располагается кнопка **Quit**, при нажатии на которую произойдет выход из программы. В нижней зоне располагается надпись с числом. А в рабочей области задействованы индикатор выполнения (элемент `ProgressBar`) и установщик (элемент `Dial`), которые мы заимствовали из табл. 54.3.



Рис. 54.8. Окно приложения с элементами меню, индикатора выполнения и ползунка

В листинге 54.11, как и в прошлый раз (см. листинг 54.10), мы начинаем с включения модуля `QtQuick.Controls` и определения основного окна приложения `ApplicationWindow`. Верхняя зона задается в свойстве `header`. В нем мы создаем элемент кнопки нажатия, который обеспечивает выход из программы. Мы задаем ему в опции `text` значение `"Quit"` и реализуем реакцию на его нажатие: `onClicked`.

Наша программа предоставляет два функционально связанных друг с другом элемента: `ProgressBar` и `Slider`. При помощи свойств `x`, `y`, `width`, `height` мы располагаем элемент индикатора выполнения `ProgressBar` в верхней части окна приложения и соединяем свойство установки текущего значения `value` со свойством актуального значения элемента установщика `Dial value`, используя его идентификатор `slider`.

Элементу ползунка присваиваем идентификатор `slider` и размещаем при помощи свойств `x`, `y`, `width`, `height` в нижней части окна приложения. В качестве начального значения устанавливаем 0.75.

Элемент основного окна `ApplicationWindow` содержит также строку состояния. Ее мы реализуем следующим образом: задаем в нижней зоне при помощи свойства `footer` элемент `ToolBar` и помещаем в него элемент надписи `Label`. Элемент надписи снабжаем идентификатором `statuslbl`. С помощью этого идентификатора мы передаем актуальные значения элемента установщика `Dial` для их отображения в строке состояния.

Листинг 54.11. Окно приложения с элементами (файл main.qml)

```
import QtQuick 2.8
import QtQuick.Controls 2.2

ApplicationWindow {
    visible: true
    width: 200
    height: 200
    title: "Controls"
    header: ToolBar {
        Button {
            text: "Quit"
            onClicked: Qt.quit();
        }
    }
```

```
    footer: ToolBar {
        id: statusbar
        Label {
            id: statuslbl
        }
    }
    ProgressBar {
        x: 0
        y: 0
        width: parent.width
        height: parent.height / 2
        value: slider.value
    }
    Dial {
        id: slider
        x: 0
        y: parent.height / 2
        width: parent.width
        height: parent.height / 2
        value: 0.75
        stepSize: 0.1
        onValueChanged: statuslbl.text = slider.value
    }
}
```

Помимо модуля `QtQuick.Controls`, существует еще один дополнительный модуль с элементами управления. Этот модуль называется `QtQuick.Extras`. Все элементы, входящие в этот модуль, представлены в табл. 54.4. Их можно использовать совместно с элементами управления табл. 54.3. Чтобы избежать конфликтов с элементами, находящимися в обоих модулях с одинаковыми именами, — например: `DelayButton`, `Dial` и `Tumbler`, рекомендуется снабдить после импорта этот модуль отдельным идентификатором и затем обращаться к его элементам с помощью идентификатора модуля. Например, так:

```
import QtQuick.Extras 1.4 as QQE
QQE.DelayButton{}
```

Таблица 54.4. *Дополнительные элементы пользовательского интерфейса QtQuick.Extras 1.4*

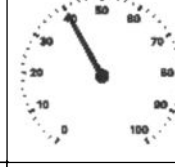
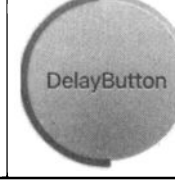
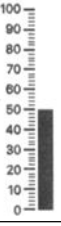
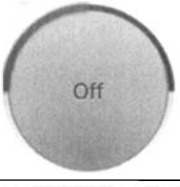
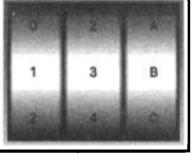
Элемент	Описание	Вид
`CircularGauge`	Элемент отображения значений в виде измерительного прибора круговой формы. Может использоваться для отображения значений скорости, а также для показа уровня громкости звука	
`DelayButton`	Элемент кнопки задержки. Процесс задержки при нажатии на эту кнопку отображается в виде линии на внешней дуге элемента	

Таблица 54.4 *(окончание)*

Элемент	Описание	Вид
`Dial`	Элемент установщик. Похож на регулятор громкости, который нужно для регулировки вращать. Используется для установки в заданном диапазоне значений, не требующих высокой точности	
`Gauge`	Элемент отображения значений в виде измерительного прибора типа термометра. Может быть использован вместо элемента индикации выполнения или для отображения уровня громкости звука	
`PieMenu`	Элемент меню круговой формы. Альтернативное представление для элементов меню	
`StatusIndicator`	Элемент показа статуса. Имеет внешний вид и то же информационное значение, что и обычный светодиод	
`ToggleButton`	Кнопка выключателя круглой формы. Может пребывать в двух состояниях: нажатом и не нажатом. Ее состояния отображаются в верхней левой и верхней правой частях кнопки в виде цветных дуг, а также опционально в виде текста в середине элемента	
`Tumbler`	Элемент предоставления выбора в виде колес. Содержит одно или несколько колес. Пользователи выбирают нужные значения, вращая колеса так, чтобы нужные значения находились в подсвеченном среднем ряду	

Диалоговые окна

Технология Qt Quick предоставляет стандартные диалоговые окна для задания цвета: `ColorDialog`, открытия файлов: `FileDialog`, установки шрифта: `FontDialog` и вывода сообщений: `MessageDialog`. Для использования диалоговых окон необходимо включить модуль `QtQuick.Dialogs`.

В следующем примере (листинг 54.12) мы продемонстрируем использование всех трех стандартных диалоговых окон и окно для вывода сообщений. На рис. 54.9 показаны три окна. *Левое верхнее* окно содержит три кнопки с надписями. При нажатии на первую кнопку (с надписью **Select color**) открывается диалоговое окно выбора цвета, показанное *справа*. Когда пользователь выбрал в этом диалоговом окне нужный цвет и нажал кнопку **OK**, окно выбора цвета закрывается и открывается окно отображения сообщения, показанное *внизу слева*, — оно демонстрирует код выбранного цвета. Аналогичным образом работают и две

остальные кнопки: **Select font** и **Select file** — с той лишь разницей, что первое из них открывает стандартное диалоговое окно выбора шрифта, а второе — стандартное диалоговое окно для открытия файла.

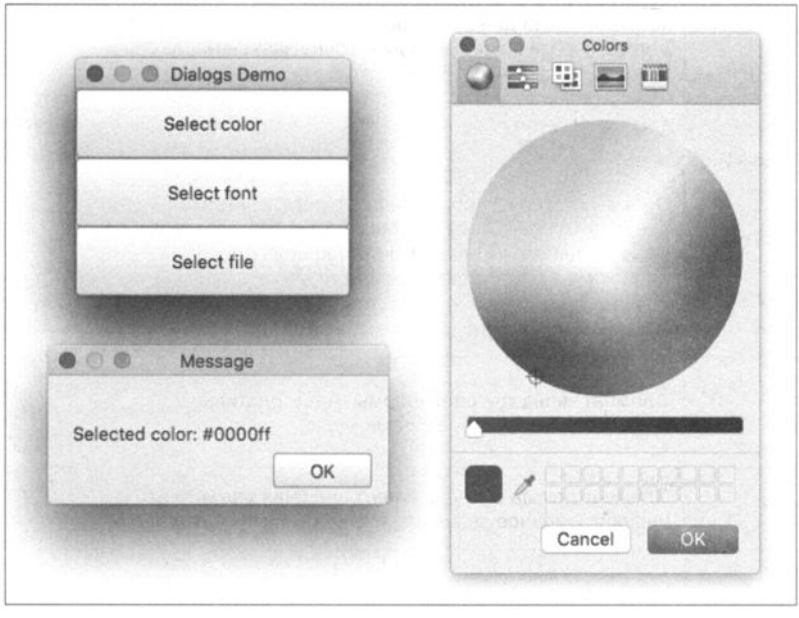


Рис. 54.9. Диалоговые окна выбора цвета и вывода сообщений

В листинге 54.12 мы включаем модуль для диалоговых окон `QtQuick.Dialogs` и в окне приложения создаем при помощи элемента повторитель три кнопки нажатия — в качестве моделей для создания кнопок используются идентификаторы диалоговых окон. С помощью повторителя мы обращаемся к свойствам диалоговых окон `title` и `visible`: в первом свойстве мы задаем кнопке надпись, а второе свойство используем для управления показом диалогового окна.

Все три элемента стандартных диалогов программы работают похожим образом, поэтому объясним принцип их действия на примере диалогового окна выбора цвета. Мы присваиваем элементу диалогового окна выбора цвета `ColorDialog` идентификатор `colorDialog`, присваиваем свойству `visible` значение `false` и делаем его невидимым. Свойство `modality` управляет модальностью диалогового окна — возможностью взаимодействия пользователя с основным окном приложения после открытия этого диалогового окна. Присвоив этому свойству значение `Qt.WindowModal`, мы делаем его модальным. В свойствах `title` и `color` мы устанавливаем заголовок окна и выбор синего цвета по умолчанию. При нажатии на кнопку **OK** вызывается код, заданный в свойстве `onAccepted`. Там мы прописываем в диалоговом окне информационного сообщения (свойство `informativeText`) текст с информацией о выбранном цвете и делаем окно отображения сообщений видимым (свойство `visible`). Свойству `visible` элемента диалогового окна отображения сообщений `MessageDialog` мы присваиваем `false` и делаем его по умолчанию невидимым. Мы также делаем его немодальным, присваивая его свойству `modality` значение `Qt.NonModal`. Это позволит нам нажимать на кнопку запуска окна выбора цвета, расположенную в основном окне приложения, без закрытия окна сообщения.

Листинг 54.12. Диалоговые окна выбора цвета и вывода сообщений (файл main.qml)

```
import QtQuick 2.8
import QtQuick.Controls 1.3
import QtQuick.Dialogs 1.2

ApplicationWindow {
    width: 200
    height: 150
    visible: true
    title: "Dialogs Demo"

    Repeater {
        id: repeater
        model: [colorDialog, fontDialog, fileDialog]
        Button {
            y: index * (parent.height / repeater.count)
            height: parent.height / repeater.count
            width: parent.width
            text: modelData.title
            onClicked: {
                messageDialog.visible = false;
                modelData.visible = true;
            }
        }
    }

    ColorDialog {
        id: colorDialog
        visible: false
        modality: Qt.WindowModal
        title: "Select color"
        color: "blue"
        onAccepted: {
             messageDialog.informativeText = "Selected color: " + color
             messageDialog.visible = true
        }
    }

    FontDialog {
        id: fontDialog
        visible: false
        modality: Qt.WindowModal
        title: "Select font"
        onAccepted: {
             messageDialog.informativeText = "Selected font: " + font
             messageDialog.visible = true
        }
    }

    FileDialog {
        id: fileDialog
        visible: false
```

```
        modality: Qt.WindowModal
        title: "Select file"
        folder: "file://Users/"
        nameFilters: ["Doc (*.txt *.html)", "All files (*)"]
        onAccepted: {
            messageDialog.informativeText = "Selected file: " + fileUrls
            messageDialog.visible = true
        }
    }

    MessageDialog {
        id: messageDialog
        visible: false
        modality: Qt.NonModal
        title: "Message"
    }
}
```

Резюме

Элементы — это структуры в исходном коде QML. Видимые элементы обладают рядом стандартных свойств, нужных для позиционирования, задания размеров, фиксации и ссылок на элементы.

Свойства соединяются друг с другом, и если значение свойства изменилось, то свойства, которые ссылаются на него, будут тоже изменены.

Свойства элементов можно дополнять своими собственными свойствами. Свойства строго типизированы, и необходимо из предложенных типов выбрать нужный. Значениями свойств могут быть выражения и функции, которые исполняются при помощи встроенного интерпретатора JavaScript.

Модули объединяют в себе коллекции элементов и позволяют использовать их на более высоком уровне.

Элемент повторитель позволяет динамически создавать элементы, используя модель и элемент шаблона.

Технология Qt Quick предоставляет для реализации пользовательского интерфейса целый ряд готовых элементов, которые могут полностью заменить виджеты, описанные в *части II* книги. Qt Quick предоставляет также ряд диалоговых окон для выбора файлов, цвета, шрифтов и отображения сообщений.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/54-510/** или с помощью следующего QR-кода (рис. 54.10):



Рис. 54.10. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 55

Управление размещением элементов

Пища для размышления кормит сообразительных.
Тамара Клейме

Несмотря на то, что QML предоставляет возможности табличного, вертикального и горизонтального размещения элементов, к которым вы уже, используя Qt, привыкли, необходимо учитывать, что подобные подходы обладают и недостатками. Так, например, очень трудно делать нестандартные размещения, и если вы хотите использовать анимационные эффекты вместе с размещениями или осуществлять перекрытие одних элементов другими, то вам придется изрядно потрудиться.

Поэтому для размещения элементов в QML, в основном, используется другой механизм, который получил название *фиксация*.

В предыдущей главе для размещения элементов мы обходилось без фиксаторов и при изменениях размеров окна высчитывали позиции для элементов при помощи JavaScript. Фиксаторы более удобны в использовании, обладают лучшей читаемостью, реализованы на C++ и, следовательно, более эффективны.

Фиксаторы

Фиксатор (anchor) задает позиции одного элемента относительно других. Его принцип работы таков: вы сами определяете расположение элементов относительно фиксатора. Этот механизм позволяет располагать элементы более интуитивно и с учетом связей самих элементов. В качестве показательного примера выполним фиксацию текстового элемента по центру, как это показано на рис. 55.1 (листинг 55.1).

Элементы QML практически всегда вложены друг в друга, то есть один элемент содержит другие элементы, и каждый элемент расположен относительно своего родителя. В нашем примере (листинг 55.1) элемент `Text` вложен в элемент `Rectangle`, и мы фиксируем элемент `Text` относительно его родителя `Rectangle`, что реализуется всего лишь одной строчкой кода. В этой строчке происходит присвоение значения `parent` свойству `anchors.centerIn` элемента `Text`. На рис. 55.2 схематично показаны свойства координат элементов для фиксации.

Листинг 55.1. Размещение элемента в центре

```
import QtQuick 2.8
Rectangle {
```

```
    width: 360
    height: 360
    Text {
        text: "Centered"
        anchors.centerIn: parent
    }
}
```



Рис. 55.1. Размещение элемента в центре

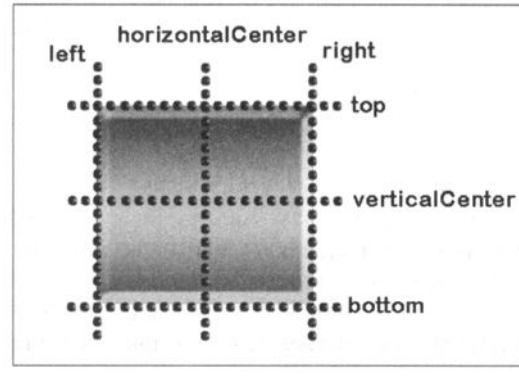


Рис. 55.2. Свойства координат для фиксации

Используя свойства, показанные на рис. 55.2, можно было бы отцентрировать элемент, применяя также свойства `verticalCenter` и `horizontalCenter` (листинг 55.2). Результат работы кода будет аналогичен результату, показанному на рис. 55.1.

Листинг 55.2. Размещение элемента в центре

```
import QtQuick 2.8
Rectangle {
    width: 360
    height: 360
    Text {
        text: "Centered"
        anchors.horizontalCenter: parent.horizontalCenter
        anchors.verticalCenter: parent.verticalCenter
    }
}
```

Отражение фиксаторов

Заметьте, что фиксаторы ссылаемых элементов имеют прямое отражение, и мы используем `parent.horizontalCenter`, а не `parent.anchors.horizontalCenter`.

В рассмотренных случаях мы фиксировали элемент в нужных местах, не изменяя его ширину и высоту. Как же быть, когда нам нужно заполнить какую-нибудь область и тем самым изменить не только позицию, но и размеры элемента? Можно связать свойства фиксатора со значениями свойств нужного нам элемента, как это показано в листинге 55.3.

Листинг 55.3. Заполнение всей области элемента

```
import QtQuick 2.8
Rectangle {
    width: 360
    height: 360
    Text {
        text: "Text"
        anchors.left: parent.left
        anchors.right: parent.right
        anchors.top: parent.top
        anchors.bottom: parent.bottom
    }
}
```

В арсенале `anchors` имеется свойство `fill`, которое способно заменить эти четыре строчки (см. листинг 55.3) всего одной. После связывания этого свойства с идентификационным номером нужного элемента оно будет заполнять его область целиком. Так что, результат работы листинга 55.4 будет полностью эквивалентен работе листинга 55.3.

Листинг 55.4. Заполнение всей области элемента

```
import QtQuick 2.8
Rectangle {
    width: 360
    height: 360
    Text {
        text: "Text"
        anchors.fill: parent
    }
}
```

В листинге 55.3 мы столкнулись также с *группированными свойствами*. Конечно, мы их видели и раньше, но их не было так много. Группированные свойства — это свойства, которые тематически подходят друг к другу, поэтому объединяются в отдельную группу, — в нашем случае группа называется `anchors`. Вы можете думать о группах как о пространстве имен C++. Таким образом, при помощи точки вы можете обратиться к каждому отдельно взятому свойству группы элемента. Нам придется часто сталкиваться с группами свойств, так как их в QML много. Существует и более компактная форма обращения к группированным свойствам. Давайте используем ее и изменим часть листинга 55.3 для присвоения значений свойствам группы `anchors` следующим образом:

```
anchors {
    left: parent.left
    right: parent.right
    anchors.top: parent.top
    anchors.bottom: parent.bottom
}
```

Для того чтобы убедиться в приведенных утверждениях, введем элемент прямоугольника, который изменит свои размеры в соответствии с элементом текста (листинг 55.5).

Листинг 55.5. Проверка фиксаций при помощи элемента прямоугольника

```
import QtQuick 2.8
Rectangle {
    width: 360
    height: 360

    Rectangle {
         color: "lightgreen"
         anchors.fill: text
    }

    Text {
        id: text
        text: "Text"
        anchors.fill: parent
    }
}
```

Мы здесь присвоили элементу текста идентификатор для того, чтобы элемент прямоугольника мог принимать его размеры. Для этой цели используем в прямоугольнике свойство `fill`, которое свяжем с этим идентификатором, а для того, чтобы увидеть границы элемента прямоугольника, присвоим ему светло-зеленый фон.

Запускаем код на выполнение и убеждаемся, что вся область окна закрашена в светло-зеленый цвет (рис. 55.3). Она также остается вся закрашенной и при изменениях размеров окна — это достигается вследствие того, что свойствам не присваивается какое-то конкретное значение, а осуществляется их связка, — то есть при изменении значений свойств выполняется автоматическое изменение значений, связанных с ними свойств.

Теперь в элемент текста из листинга 55.3 подставьте четыре строчки фиксации — их выполнение должно дать аналогичный результат.

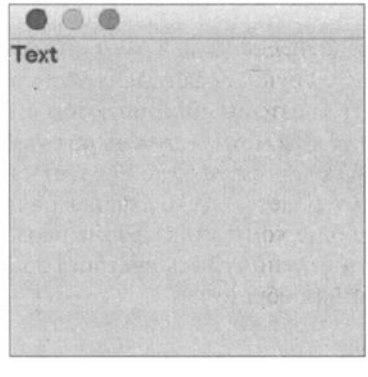


Рис. 55.3. Проверка заполнения элемента

Затем подставим в листинг 55.5 фиксацию центрирования из листинга 55.1: `anchors.centerIn: parent`. Теперь прямо в середине окна мы должны увидеть прямоугольную светло-зеленую область, обрамляющую только текст (рис. 55.4).

Далее рассмотрим, как можно использовать свойства `anchors` для выполнения размещения с перекрытием. Продемонстрируем это (листинг 55.6) на двух прямоугольных элементах красного и зеленого цвета (рис. 55.5).



Рис. 55.4. Проверка центрирования элемента

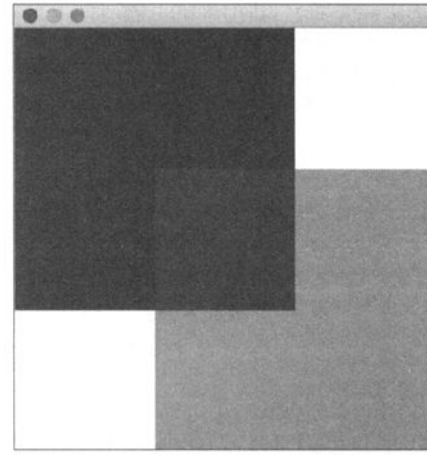


Рис. 55.5. Размещение с перекрытием

В листинге 55.6 мы задаем идентификатор `redrect` первому прямоугольнику и присваиваем ему красный цвет. Присваиваем размеры (свойства `width` и `height`) таким образом, чтобы он не занимал полностью всю площадь окна. Чтобы видеть область перекрытия, второй прямоугольник делаем полупрозрачным (свойство `opacity`) и присваиваем ему зеленый цвет. Связываем его вершину (свойство `top`) с вертикальным центром (свойство `verticalCenter`) красного прямоугольника. Его низ (свойство `bottom`) связываем с низом элемента родителя. Левую границу (свойство `left`) связываем с горизонтальным центром (свойство `horizontalCenter`) красного прямоугольника, а правую границу (свойство `right`) — с правой границей элемента родителя.

Листинг 55.6. Фиксация с перекрытием элемента

```
import QtQuick 2.8
Item {
    width: 360
    height: 360

    Rectangle {
        id: redrect
        color: "red"
        width: parent.width / 1.5
        height: parent.height / 1.5
        anchors.top: parent.top
        anchors.left: parent.left
    }
    Rectangle {
        opacity: 0.5
        color: "green"
        anchors.top: redrect.verticalCenter
        anchors.bottom: parent.bottom
        anchors.left: redrect.horizontalCenter
        anchors.right: parent.right
    }
}
```

Когда мы задаем вертикальные или горизонтальные расположения с помощью фиксаторов, то можем контролировать размеры элементов, которые находятся между элементами. На рис. 55.6 показаны три элемента. У двух крайних ширина задана постоянной, ширина же среднего элемента высчитывается на основании левой и правой границ соседних элементов (листинг 55.7).

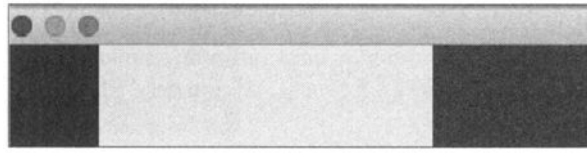


Рис. 55.6. Контроль размеров среднего элемента

В листинге 55.7 мы создаем три элемента прямоугольников: красного, желтого и зеленого цветов. Красному и зеленому прямоугольникам задаем постоянную ширину 60 и 100. Два крайних (красный и зеленый) снабжаем идентификаторами `redrect` и `greenrect`, чтобы можно было ссылаться на них из желтого прямоугольника. Ключевой момент листинга заключается в связывании свойств желтого прямоугольника `left` и `right` со свойствами `right` красного прямоугольника и `left` зеленого прямоугольника соответственно. Тем самым желтый прямоугольник полностью заполняет пространство между этими прямоугольниками, и изменение размеров окна приведет только к увеличению либо к уменьшению ширины желтого прямоугольника.

Листинг 55.7. Контроль размеров среднего элемента

```
import QtQuick 2.8
Item {
    width: 360
    height: 60

    Rectangle {
        id: redrect
        color: "red"
        anchors.left: parent.left
        anchors.top: parent.top
        anchors.bottom: parent.bottom
        width: 60
    }

    Rectangle {
        color:  "yellow"
        anchors.top: parent.top
        anchors.bottom: parent.bottom
        anchors.left: redrect.right
        anchors.right: greenrect.left
    }

    Rectangle {
        id: greenrect
        color: "green"
```

```
        anchors.right: parent.right
        anchors.top: parent.top
        anchors.bottom: parent.bottom
        width: 100
    }
}
```

Отступы от краев элемента можно задать при помощи свойств `topMargin`, `bottomMargin`, `leftMargin` и `rightMargin`, которые определены в свойстве `anchors`, как это показано на рис. 55.7.

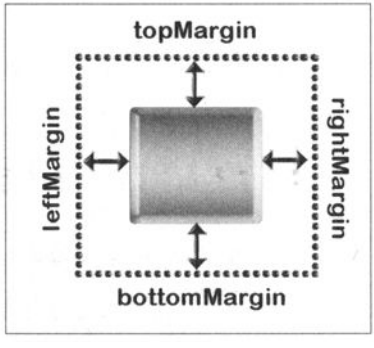


Рис. 55.7. Отступы

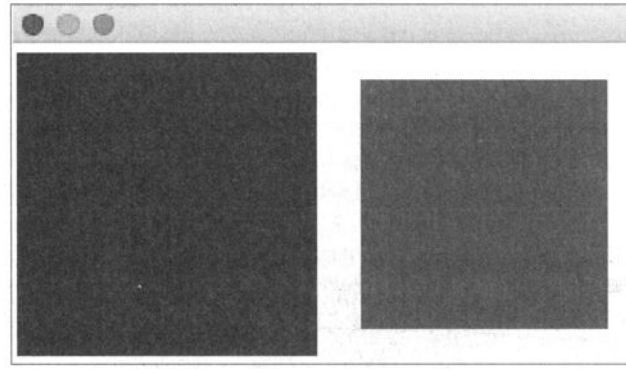


Рис. 55.8. Отступы с четырех сторон

Проиллюстрируем использование отступов (листинг 55.8) на примере двух элементов, а именно — прямоугольников, показанных на рис. 55.8.

В листинге 55.8 мы задаем цвет прямоугольникам: одному — красный, другому — зеленый (свойство `color`). Ограничиваем правую границу красного прямоугольника серединой окна (свойство `right`), в остальном его размеры соответствуют размерам основного окна. Затем при помощи свойств `leftMargin`, `topMargin`, `rightMargin` и `bottomMargin` задаем одинаковые отступы, равные пяти пикселам. Аналогично поступаем и с зеленым прямоугольником с той разницей, что ограничиваем его левую границу серединой основного окна и со всех сторон отступаем на 20 пикселов. В этом примере мы отступаем со всех сторон, но если будет необходимо сделать пространство между элементами лишь с одной стороны, используйте только один отступ с нужной вам стороны. Также можно сделать относительный отступ при помощи свойств, имена которых оканчиваются словом `Offset`, — если поставить следующие инструкции в блок `anchors` листинга 55.8, то будет осуществлен отступ элемента на десять пикселов вниз:

```
verticalCenterOffset: 10
verticalCenter: parent.verticalCenter
```

Листинг 55.8. Использование отступов

```
import QtQuick 2.8
Item {
    width: 360
    height: 180
```

```
    Rectangle {
        color: "red"
        anchors {
            right: parent.horizontalCenter
            left: parent.left
            top: parent.top
            bottom: parent.bottom
            leftMargin: 5
            topMargin: 5
            rightMargin: 5
            bottomMargin: 5
        }
    }
    Rectangle {
        color: "green"
        anchors {
            left: parent.horizontalCenter
            right: parent.right
            top: parent.top
            bottom: parent.bottom
            leftMargin: 20
            topMargin: 20
            rightMargin: 20
            bottomMargin: 20
        }
    }
}
```

Традиционные размещения

Теперь рассмотрим традиционные методы размещения, которые похожи на используемые в Qt. Эти размещения являются тоже элементами. Укажем основные размещения — их по два для каждого из видов:

- `Row`, `RowLayout` — область для горизонтального размещения элементов, аналогом в Qt является класс `QHBoxLayout`;
- `Column`, `ColumnLayout` — область для вертикального размещения элементов, аналогом в Qt является класс `QVBoxLayout`;
- `Grid`, `GridLayout` — область для табличного размещения элементов, аналогом в Qt является класс `QGridLayout`;
- `StackLayout` — стековая область для размещения, в которой одновременно можно видеть только один элемент. Аналогом подобного размещения в Qt является класс `QStackedLayout`.

Объясним разницу между двумя элементами одинаковых видов размещений. Элементы размещений с коротким именем обладают свойствами `spacing` и `layoutDirection`. Эти свойства служат для установки промежутков между элементами и изменения направления размещения соответственно.

Элементы размещений, оканчивающиеся словом Layout, содержатся в отдельном модуле QtQuick.Layouts и обладают, помимо указанных свойств, дополненным свойством Layout. Это свойство дает возможность устанавливать и получать минимальную, максимальную и предпочтительную высоту и ширину:

- Layout.minimumWidth — минимальная ширина;
- Layout.minimumHeight — минимальная высота;
- Layout.maximumWidth — максимальная ширина;
- Layout.maximumHeight — максимальная высота;
- Layout.preferredWidth — предпочтительная ширина;
- Layout.preferredHeight — предпочтительная высота.

А так же заполнение по ширине и высоте:

- Layout.fillWidth — заполнение по ширине элемента размещения;
- Layout.fillHeight — заполнение по высоте элемента размещения.

Продемонстрируем работу горизонтального размещения сначала на примере элемента Row на трех элементах (листинг 55.9) и упорядочим их, как это показано на рис. 55.9.

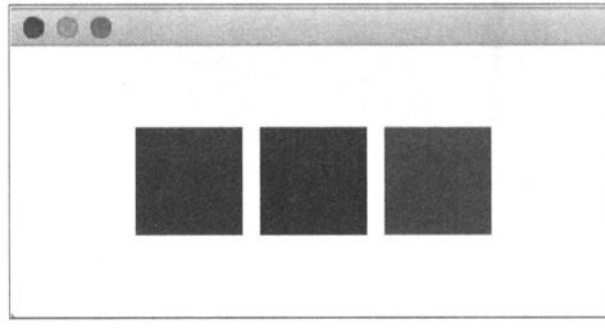


Рис. 55.9. Горизонтальное размещение элементов с помощью элемента Row

В листинге 55.9 мы фиксируем сам элемент размещения с центром окна (свойство centerIn). Задаем расстояние между элементами, равное 10 пикселам. Внутри элемента горизонтального размещения Row определяем три элемента прямоугольников красного, голубого и зеленого цветов с размерами 64×64 пиксела.

Листинг 55.9. Горизонтальное размещение с использованием элемента Row

```
import QtQuick 2.8
Item {
    width: 360
    height: 160

    Row {
        anchors.centerIn: parent
        spacing: 10
        Rectangle {
            width: 64; height: 64; color: "red"
        }
```

```
        Rectangle {
            width: 64; height: 64; color: "blue"
        }
        Rectangle {
            width: 64; height: 64; color: "green"
        }
    }
}
```

Теперь продемонстрируем возможности размещения с элементом `RowLayout` (листинг 55.10). Установим минимальные размеры элементов и разрешим первому и последнему элементам вытягиваться на всю длину окна, а среднему элементу — заполнять пространство между крайними элементами, как это показано на рис. 55.10.

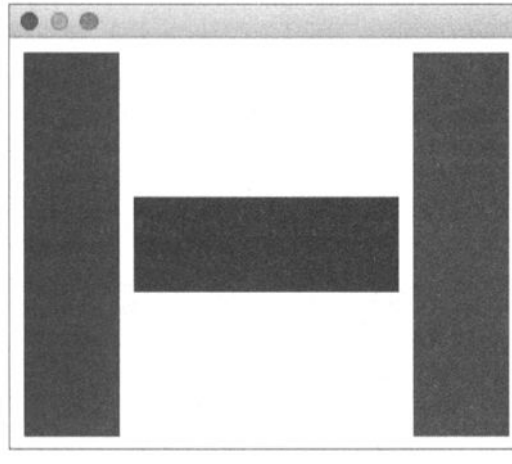


Рис. 55.10. Горизонтальное размещение элементов с помощью элемента `RowLayout`

В листинге 55.10 мы включаем модуль `QtQuick.Layouts` для использования размещений. Заполняем всю область окна элемента размещения `RowLayout` присвоением `anchords.fill` элемента предка (`parent`). Устанавливаем рамку (бордюр), равную 10 (свойство `margins`), и фиксированное растояние между элементами (свойство `spacing`), тоже равное 10. Встроенному свойству размещения `fillHeight` первого и последнего элементов `Rectangle` мы присваиваем значение `true` — что дает этим элементам возможность увеличиваться по высоте. В среднем элементе для возможности увеличения в ширину мы присваиваем свойству `fillWidth` значение `true`. Всем элементам `Rectangle` мы устанавливаем минимальные размеры 64×64 при помощи свойств `minimumWidth` и `minimumHeight`.

Листинг 55.10. Горизонтальное размещение с использованием элемента `RowLayout`

```
import QtQuick 2.8
import QtQuick.Layouts 1.3

Item {
    width: 320
    height: 240
```

```
    RowLayout {
        anchors.fill: parent
        anchors.margins: 10
        spacing: 10

        Rectangle {
            Layout.fillHeight: true
            Layout.minimumWidth: 64;
            Layout.minimumHeight: 64;
            color: "red"
        }
        Rectangle {
            Layout.fillWidth: true
            Layout.minimumWidth: 64;
            Layout.minimumHeight: 64;
            color: "blue"
        }
        Rectangle {
            Layout.fillHeight: true
            Layout.minimumWidth: 64;
            Layout.minimumHeight: 64;
            color: "green"
        }
    }
}
```

Размещение в вертикальном порядке работает аналогично, и чтобы в этом убедиться, в листинге 55.9 просто замените имя элемента `Row` на `Column`.

В табличном размещении есть дополнительные свойства: `rows` и `column`, которые задают количество строк и столбцов таблицы. Выполним табличное размещение четырех элементов (листинг 55.11), как это показано на рис. 55.11.

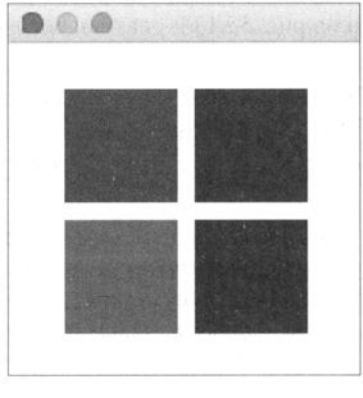


Рис. 55.11. Размещение элементов в виде таблицы

Листинг 55.11 практически аналогичен листингу 55.9 с той лишь разницей, что в этом случае при помощи свойств `rows` и `columns` мы присваиваем количество строк и столбцов, равное 2, и используем для размещения четыре элемента вместо трех.

Листинг 55.11. Табличное размещение

```
import QtQuick 2.8
Item {
    width: 200
    height: 200

    Grid {
        rows: 2
        columns: 2
        anchors.centerIn: parent
        spacing: 10

        Rectangle {
            width: 64; height: 64; color: "red"
        }
        Rectangle {
            width: 64; height: 64;color: "blue"
        }
        Rectangle {
            width: 64; height: 64; color: "green"
        }
        Rectangle {
            width: 64; height: 64; color: "black"
        }
    }
}
```

Размещение в виде потока

Одним из интересных размещений можно назвать размещение в виде потока (элемент `Flow`). Оно упорядочивает элементы «змейкой», которая пытается разместить как можно большее количество элементов в заданной области окна. На рис. 55.12 показано размещение из десяти элементов до изменения размеров окна, а на рис. 55.13 — размещение тех же элементов, но после изменения его размеров.

В листинге 55.12 мы создаем элемент размещения `Flow` внутри главного элемента `Item`. В элементе `Flow` мы устанавливаем равные 20 пикселам отступы (`anchors.margins`) от всех сторон главного окна приложения и промежутки между элементами (`spacing`) также в 20 пикселов.

Элемент-повторитель содержит встроенную модель (`model`), которая возвращает 10 элементов строк с цветовыми значениями, причем элемент с четным индексом получает значение красного цвета «red», а с нечетным — значение зеленого цвета «green». Для того чтобы отобразить элементом `Rectangle` не прямоугольник, а окружность, мы задаем радиус закругления прямоугольника (`radius`), равный половине одной из его сторон, то есть 32 пиксела. Основной цвет задается в прямоугольнике посредством текущего значения модели данных (`modelData`). Цифровое значение отображается при помощи дочернего элемента `Text`, в котором само значение является текущим индексом элемента (`index`) встроенной модели.

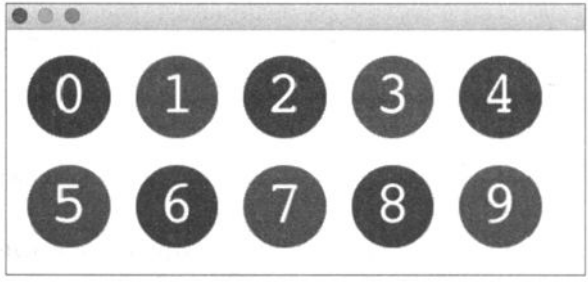


Рис. 55.12. Начальное размещение элементов

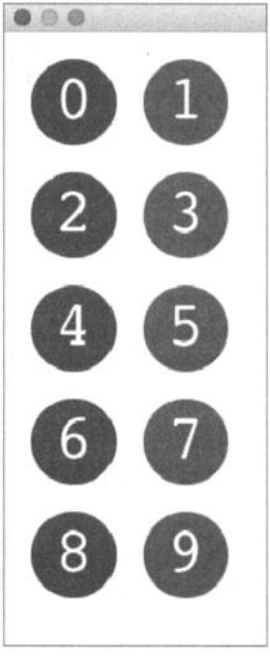


Рис. 55.13. Размещение элементов после изменения размеров окна

Листинг 55.12. Поточное размещение

```
import QtQuick 2.8
Item {
    width: 480
    height: 200
    Flow {
        anchors.fill: parent
        anchors.margins: 20
        spacing: 20
        Repeater {
            model: {var v = new Array(10);
                    for (var i = 0; i < v.length; ++i) {
                        v[i] = i % 2 ? "green" : "red"
                    }
                    return v;
                   }
            Rectangle {
                width: 64
                height: 64
                radius: 32
                color: modelData
                Text {
                    color: "white"
                    font.pixelSize: 48
```

```
                font.family: "Courier"
                anchors.centerIn: parent
                text: index
            }
        }
    }
}
}
```

Резюме

Применяя традиционный подход, реализованный в классах размещений Qt, очень трудно использовать анимационные эффекты и осуществлять перекрытие одних элементов другими. Именно поэтому для размещения элементов в QML принят другой подход, который называется *фиксацией*. Этот подход не имеет указанных недостатков и позволяет располагать элементы более интуитивно с учетом их взаимосвязей.

Следует заметить, что традиционный подход размещения элементов в QML тоже поддерживается.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/55-510/** или с помощью следующего QR-кода (рис. 55.14):

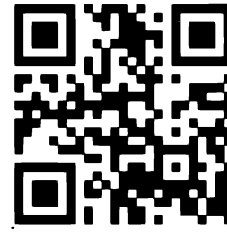


Рис. 55.14. QR-код для доступа на страницу комментариев к этой главе

ГЛАВА 56

Элементы графики

Осуществление Великого начинается с малого.
Дао де Дзин

Язык QML предоставляет возможности задания цветов, шрифтов, манипулирования растровыми изображениями, создания градиентов и рисования графических примитивов на элементах холста.

Цвета

Цвета в QML можно задавать в виде строк или использовать встроенную функцию Qt. Строковое задание цвета осуществляется по имени либо в формате числового кода.

Строки имен — это стандарт, используемый в SVG. Вот, например, как могут выглядеть такие имена: `"red"`, `"green"`, `"darkkhaki"`, `"snow"` и т. д. Более подробную информацию о именах 148 цветов можно найти на странице: **www.qt-book.com/colors/** или с помощью следующего QR-кода (рис. 56.1)

Рис. 56.1. QR-код для доступа на страницу с таблицей цветов

Строки числовых кодов цвета задаются в следующем формате: `#rrggbb`, принятом в HTML: `"#FF0000"`, `"#00FE00"`, `"#0000AF"` и т. д. Однако мало кто может представить себе, как будет выглядеть тот или иной цвет, вводя эти значения. Именно поэтому Qt Creator дает нам в помощь соответствующий интерактивный инструмент (рис. 56.2). Для того чтобы им воспользоваться, просто встаньте в коде на позицию свойства `color`, выполните вызов контекстного меню и в этом меню выберите пункт **Show Qt Quick Toolbar**. В открывшемся окне вы можете настроить нужное вам значение цвета — его выбор сразу же осуществит изменение выбранного значения в исходном коде программы.

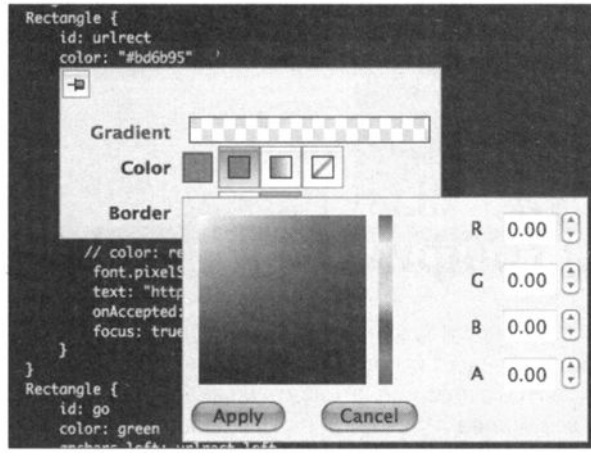


Рис. 56.2. Интерактивная настройка значения цвета из Qt Creator

Благодаря встроенной функции `rgba()` можно также получать различные цветовые значения. Ее использование выглядит следующим образом:

```
Rectangle {
    color: Qt.rgba(0.3, 0.45, 0.21)
    opacity: 0.5
}
```

Для задания прозрачности можно использовать свойство `opacity`. Диапазон значений этого свойства лежит в пределах от нуля до единицы. Значение, равное нулю, означает полную прозрачность, а равное единице — полную непрозрачность. В приведенном ранее примере мы этому свойству задаем значение 0.5 и делаем тем самым элемент прямоугольника полупрозрачным.

Более подробную информацию о цветовых моделях можно получить в *главе 17*.

Растровые изображения

Для растровых изображений в QML существуют сразу два элемента: `Image` и `BorderImage`. Для того чтобы файлы растровых и векторных изображений можно было использовать в языке QML, они должны быть в формате JPG, PNG или SVG.

Элемент *Image*

Элемент `Image` отображает файл изображения, указанный в свойстве `source`. Этот файл может находиться не только на локальном диске компьютера, но и в сети, поэтому ссылка осуществляется либо при помощи URL, либо по относительному пути к файлу. Кроме всего, элемент `Image` поддерживает не только растровые, но и векторные изображения в формате SVG.

Для настройки свойств элемента `Image` (рис. 56.3) программа Qt Creator предоставляет интерактивный редактор. Чтобы его открыть, встаньте на сам элемент `Image` и вызовите контекстное меню, в котором выберите пункт **Show Qt Quick Toolbar**. Далее вы можете просто поэкспериментировать с изменением свойств.

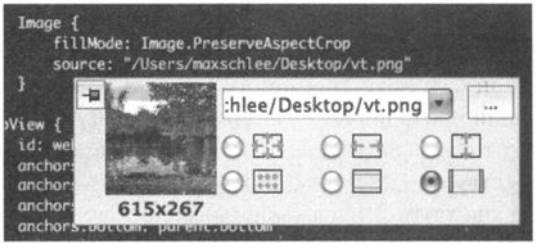


Рис. 56.3. Интерактивная настройка свойств элемента `Image` из Qt Creator

Само изображение можно впоследствии подвергать трансформациям, например уменьшению/увеличению (свойство `scale`) и повороту (свойство `rotation`). Эти трансформации по умолчанию осуществляются относительно центральной точки самого элемента. Для изменения точки для трансформации и поворота нужно задать соответствующее значение свойства `transformOrgin`. Например, чтобы изображение поворачивалось относительно верха элемента, можно поступить так:

```
Image {
...
   transformOrigin: Item.Top
...
}
```

Следующий пример (листинг 56.1) демонстрирует окно, отображающее растровое изображение при помощи элемента `Image` с уменьшением и поворотом (рис. 56.4).

В листинге 56.1 мы создаем элемент `Rectangle`, который станет предком для нашего элемента `Image`. Задаем ему цвет воды для заполнения фона `"aqua"` и размер, равный размеру

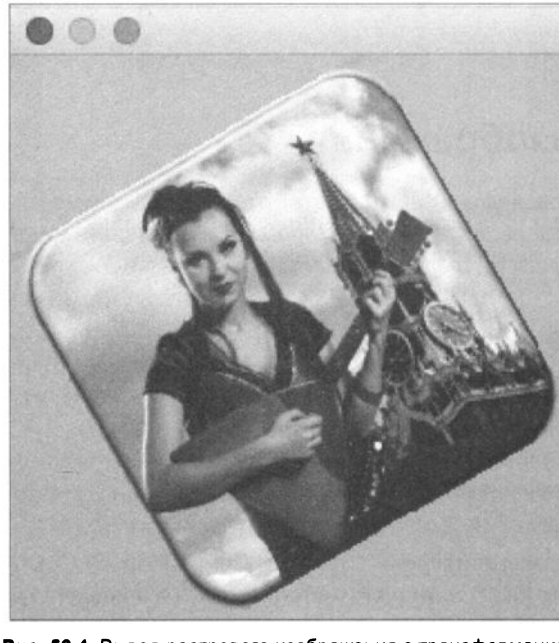


Рис. 56.4. Вывод растрового изображения с трансформацией

оригинала изображения (свойства `width` и `height`), — для этого ссылаемся на них при помощи идентификатора `img`. Теперь переходим к элементу `Image`. Его относительную позицию устанавливаем равной 0, 0 (свойства `x` и `y`). Свойству `smooth` устанавливаем значение `true` — чтобы все операции проводились со сглаживанием, и картинка имела приятный вид. Свойство `source` содержит имя файла `"balalaika.png"`, и так как этот файл находится в ресурсе, то имени предшествует идентификатор ресурса `qrc:`. Свойство `scale` определяет, насколько большим должно быть изображение: значения до единицы уменьшают изображение, а значения больше единицы — увеличивают его. В нашем примере мы уменьшаем изображение относительно центра элемента. Свойство `rotation` поворачивает изображение относительно центра на 30 градусов, а так как значение градусов отрицательно, то поворот выполняется против направления часовой стрелки. Свойства `width` и `height` не инициализируются значениями явно, поскольку их инициализация происходит неявно при загрузке, и соответствуют размеру изображения, находящегося в файле.

Листинг 56.1. Отображение графического файла

```
import QtQuick 2.8
Rectangle {
    color: "aqua"
    width: img.width
    height: img.height

    Image {
        id: img
        x: 0
        y: 0
        smooth: true
        source: "qrc:/balalaika.png"
        scale: 0.75
        rotation: -30.0
    }
}
```

Для более тонкой настройки трансформации ее можно задавать при помощи элементов. В листинге 56.2 приведен элемент `Image`, использующий эти элементы, — его действия полностью эквивалентны элементу `Image` из листинга 56.1.

Все трансформации здесь присваиваются свойству `transform`. Если трансформаций больше одной, то они должны указываться в виде *списка*. Итак, мы первый раз столкнулись со списками. Списки в QML задаются квадратными скобками, внутри скобок вписываются все входящие в список элементы, которые должны разделяться между собой запятыми. В нашем случае мы осуществляем две трансформации, и в наш список входят два элемента для трансформации: `Scale` и `Rotation`. В каждом случае в качестве точки для проведения трансформации устанавливаем середину изображения (свойства `origin.x` и `origin.y`). В трансформации размера `Scale` присваиваем одно и то же значение 0.75 двум размерностям: `x` и `y` (свойства `xScale` и `yScale`). А в элементе `Rotation` устанавливаем свойству `angle` значение 30 градусов.

Листинг 56.2. Уменьшение и поворот

```
import QtQuick 2.8
Image {
    id: img
    x: 0
    y: 0
    smooth: true
    source: "qrc:/balalaika.png"
    transform: [
        Scale {origin.x: width / 2
               origin.y: height / 2
               xScale: 0.75
               yScale: 0.75
        },
        Rotation {origin.x: width / 2
                  origin.y: height / 2
                  angle: -30.0
        }
    ]
}
```

Теперь продемонстрируем загрузку и показ картинки из Интернета. Для транспортировки растрового изображения по Сети требуется время, и для того, чтобы пользователь не нервничал и не задавал вопросы, что это за окно бирюзового цвета и зачем оно, нужно проинформировать его о том, что программа находится в процессе выполнения работы, и ему необходимо подождать (рис. 56.5). После завершения процесса загрузки в окне отобразится загруженное растровое изображение (рис. 56.6).

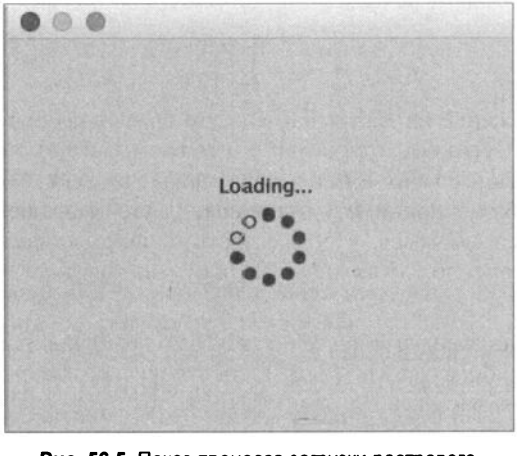


Рис. 56.5. Показ процесса загрузки растрового изображения из Интернета



Рис. 56.6. Вывод растрового изображения, полученного из Интернета

В листинге 56.3 мы создаем элемент `Image` и задаем веб-ссылку на источник растрового изображения в свойстве `source`. В элементе размещения `Column` мы задаем два элемента: `Text` и `BusyIndicator`, которые и будут сигнализировать о выполнении программой процес-

са загрузки. Элемент размещения с этими двумя элементами будет виден до тех пор, пока статус элемента `Image` равен `Loading`, в противном случае он и его дочерние элементы станут невидимыми (см. свойство `visible`).

Листинг 56.3. Загрузка и показ растрового изображения из Интернета

```
import QtQuick 2.8
import QtQuick.Controls 2.2

Rectangle {
    color: "aqua"
    width: 320
    height: 240

    Image {
        id: img
        anchors.fill: parent
        smooth: true
        source: "http://qt-book.com/pic.jpg"
        Column {
            anchors.centerIn: parent
            visible: img.status == Image.Loading ? true : false
            Text {
                text: "Loading..."
            }
            BusyIndicator {
            }
        }
    }
}
```

Элемент *BorderImage*

Элемент `BorderImage` позволяет разбить изображение на девять частей. Это бывает необходимо для создания масштабируемой графики. Основные трудности с изменяемыми размерами возникают у элементов, имеющих закругленные углы. Благодаря элементу `BorderImage` можно создать базовые компоненты, которые будут принимать различные размеры без некрасивых искажений. Этим элементом мы воспользуемся, чтобы создать полностью масштабируемый элемент кнопки, показанный на рис. 56.7. Как видно из рисунка, эти разбиения управляются свойствами `left`, `right`, `top` и `bottom`.

Реализуем пример, в котором используется растровое изображение как раз такой кнопки с закругленными углами (листинг 56.4). Эта кнопка должна быть в состоянии принимать любые размеры без искажений. На рис. 56.8 показано, как выглядит такая кнопка при изменении размеров во всех направлениях.

В листинге 56.4 мы указываем в свойстве `source` имя файла — это осуществляет загрузку растрового изображения для кнопки. Так как мы сделали элемент `BorderImage` элементом верхнего уровня, то он отвечает за размеры окна, поэтому устанавливаем их в явном виде (свойства `width` и `height`). Далее следует разбивка самого изображения на девять частей, а ее границы устанавливаются свойствами `left`, `top`, `right` и `bottom`.

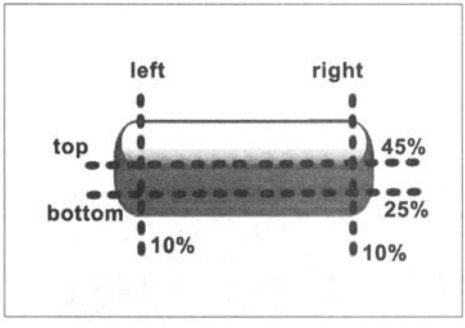


Рис. 56.7. Девять частей разбивки растрового изображения

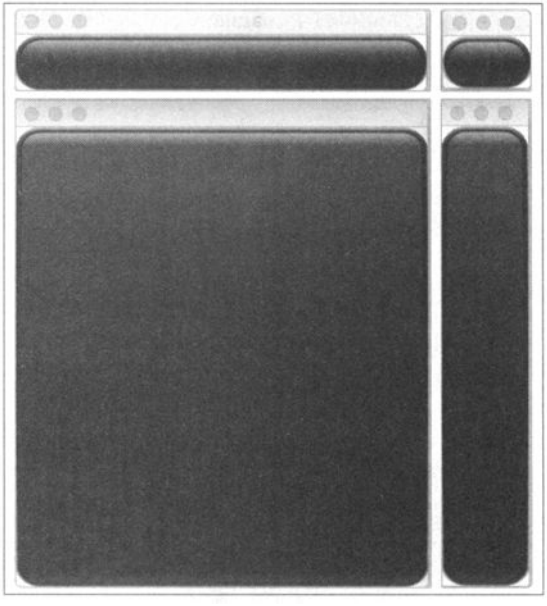


Рис. 56.8. Различные размеры растрового изображения кнопки

Листинг 56.4. Масштабируемая кнопка

```
import QtQuick 2.8
BorderImage {
    source: "qrc:/button.png"
    width: 100
    height: 45
    border {left: 30; top: 15; right: 30; bottom: 15}
}
```

Градиенты

В язык QML включен только один градиент — линейный. Для его задания существует свойство `gradient`, которому в качестве значения необходимо присвоить элемент `Gradient`. Этот элемент содержит две или более точек останова (элемент `GradientStop`). Каждая точка останова имеет позицию: номер между 0 (стартовая точка) и 1 (конечная точка) и цвет. Стартовая и конечная точки располагаются в верхних и нижних углах и не могут быть перемещены. Поэтому если нужно сделать линейный градиент по диагонали, то следует использовать элементы трансформации.

Следующий пример (листинг 56.5) показывает, как можно сделать такой градиент (рис. 56.9).

В элементе `Rectangle` листинга 56.5 мы присваиваем свойству `gradient` элемент `Gradient`, в котором определены три точки останова `GradientStop`: начальная точка — 0, промежуточная — находится на расстоянии 0.7 и конечная точка — 1 (свойства `position`). В каждой из этих точек задан свой собственный цвет (свойства `color`). В завершение выполняется трансформация поворота и увеличение нашего градиента (свойства `rotation` и `scale`).

Листинг 56.5. Линейный градиент

```
import QtQuick 2.8
Rectangle {
    width: 200
    height: 200
    gradient: Gradient {
        GradientStop {position: 0.0; color: "blue"}
        GradientStop {position: 0.7; color: "gold"}
        GradientStop {position: 1.0; color: "silver"}
    }
    rotation: 30;
    scale: 1.5
}
```

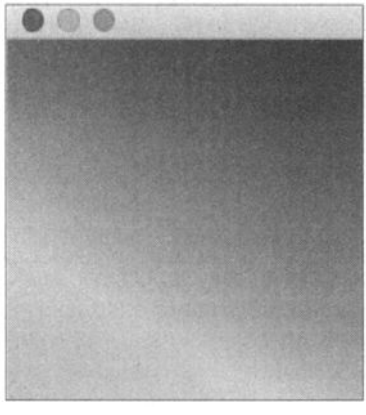


Рис. 56.9. Линейный градиент

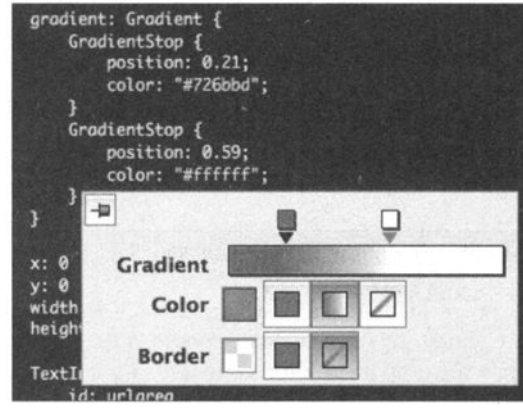


Рис. 56.10. Интерактивный редактор для изменения градиентов в программе Qt Creator

Программа Qt Creator предоставляет возможность интерактивной настройки градиентов. Просто вызовите контекстное меню из позиции свойства `gradient`, выберите пункт **Show Qt Quick Toolbar**, и вы увидите окно, показанное на рис. 56.10. Все изменения в этом окне приводят к мгновенным изменениям исходного кода выбранного градиента в тексте программы.

При задании градиентов важно помнить, что их создание может потребовать много ресурсов процессора. Из этих соображений, если ваше приложение может использоваться на мобильном устройстве, желательно применять уже готовые изображения градиентов, вместо того, чтобы каждый раз создавать их. Это поможет также более экономно расходовать заряд аккумулятора устройства мобильного устройства. Заменить градиент на изображение можно следующим образом:

```
import QtQuick 2.8
Rectangle {
    width: 200
    height: 200
    Image {
        x: 0;
        y: 0
```

```
        source: "qrc:/images/gradient.png"
    }
}
```

Создание градиентов не ограничивается только стандартными возможностями модуля `QtQuick`. Градиенты можно создавать еще и при помощи элементов холста, о которых речь пойдет дальше, а также с помощью модуля `QtGraphicalEffects`, который дает возможность создания не только линейных градиентов, но и конических, и радиальных (см. далее табл. 56.1 в *разд. «Шейдеры и эффекты»*).

Шрифты

Все свойства настройки шрифтов расположены в группе `font`. Следующий пример устанавливает в элементе `Text` шрифт Helvetica (свойство `family`) с размером 24 пиксела (свойство `pixelSize`) и полужирными символами (свойство `bold`):

```
Text {
...
    font {
        family: "Helvetica"
        pixelSize: 24
        bold: true
    }
...
}
```

Проще всего выполнять изменение свойств шрифтов сразу в интерактивном окне программы Qt Creator. Для его вызова просто поместите курсор мыши на область свойства группы `font`, нажмите правую кнопку мыши и в открывшемся контекстном меню выберите пункт **Show Qt Quick Toolbar**, — вы увидите окно, показанное на рис. 56.11, в котором можно осуществить все необходимые изменения свойств шрифта.

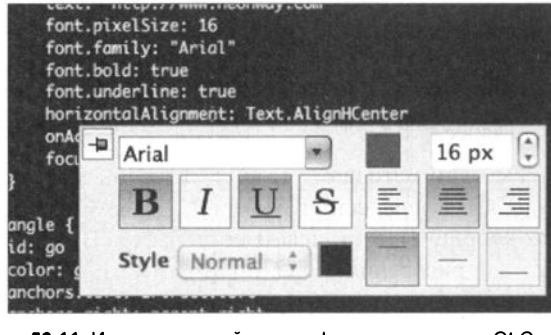


Рис. 56.11. Изменение свойств шрифта из программы Qt Creator

Рисование на элементах холста

Элемент `Canvas` представляет собой элемент холста, на котором можно выполнять растровые операции. С некоторыми допущениями он аналогичен классу контекста рисования `QPaintDevice`, на котором мы в *части IV* книги рисовали при помощи объекта `QPainter`.

Мы знаем, что QML — это описательный язык, а значит, на этом языке мы не сможем реализовать алгоритмы для рисования, и для этой цели используется встроенный язык JavaScript. Элемента `Canvas` предоставляет свойство обработки `onPaint`, которое можно сравнить с событием `QWidget::paintEvent()`. Внутри этого свойства и необходимо реализовать алгоритм рисования на JavaScript.

Следующий пример (листинг 56.6) демонстрирует рисование на холсте узора, показанного на рис. 56.12. Этот узор мы уже ранее использовали в *главе 52*.

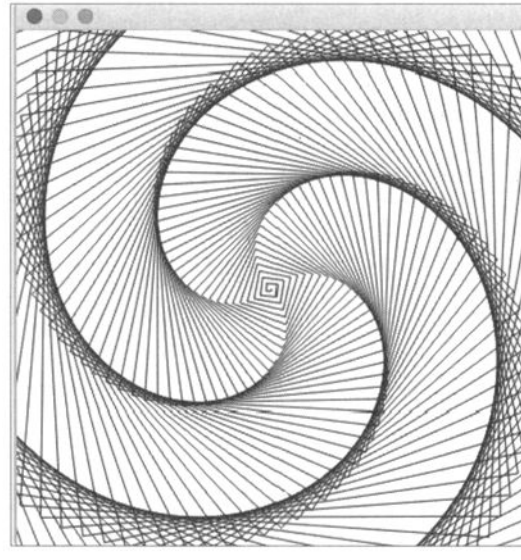


Рис. 56.12. Рисование узора на элементе холста `Canvas`

В листинге 56.6 мы создаем элемент основного окна размерами 400×400, который содержит элемент `Canvas`. При помощи свойства `anchors.fill` мы заполняем все пространство окна. На этом QML закончился, и после свойства обработки `onPaint` начинается JavaScript.

Алгоритм нашего узора мы оформим в виде отдельной функции `drawFantasy()`. В самой функции мы будем ссылаться на объект контекста рисования `ctx`, этот контекст мы получим в основной программе далее.

В самом начале функции вызовом метода `beginPath()` из объекта контекста рисования мы объявляем начало рисования графической траектории. Далее методом `translate()` производим трансформацию координат и смещаем ее в центр. Вычисляем угол в радианах `fAngle` и запускаем цикл для рисования узора, в котором используем `moveTo()` для установки начальной точки, из которой будем рисовать линию методом `lineTo()`. Потом проводим последующие трансформации перемещения `translate()` и вращения `rotate()` с постоянным углом `fAngle`. В завершение цикла мы вызываем метод `closePath()` и тем самым заканчиваем формирование нашей графической траектории.

В основной программе, которая следует сразу после реализованной функции, мы вызовом `getContext()` получаем контекст рисования. Строковый аргумент `"2d"` говорит о том, что мы хотим рисовать на плоскости, используя две координаты: *X* и *Y*.

Вызов метода `clearRect()` очищает всю область окна. Вызов `save()` из объекта контекста сохраняет его актуальное состояние — это очень важно, так как мы имеем дело не

с объектом рисования, а непосредственно с самим контекстом, а поскольку он у элемента `Canvas` всего один, то необходимо следить за правильным сохранением его состояний.

Свойству `strokeStyle` мы устанавливаем черный цвет для рисования линии. Свойство `lineWidth` контекста рисования управляет толщиной линий, которую мы устанавливаем равной 1.

Вызываем функцию `drawFantasy()` для задания траектории узора, после чего вызываем `stroke()` из объекта контекста рисования для его отображения.

В завершение мы восстанавливаем вызовом `restore()` исходное состояние контекста рисования, которое мы сохранили методом `save()`.

Листинг 56.6. Рисование на элементе холста

```
import QtQuick 2.8
Item {
    width: 400
    height: 400
    Canvas {
        anchors.fill: parent
        onPaint: {
            function drawFantasy()
            {
                ctx.beginPath()
                ctx.translate(parent.width / 2, parent.height / 2)
                var fAngle = 91 * 3.14156 / 180
                for (var i = 0; i < 300; ++i) {
                    var n = i * 2
                    ctx.moveTo(0, 0)
                    ctx.lineTo(n, 0)
                    ctx.translate(n, 0)

                    ctx.rotate(fAngle)
                }
                ctx.closePath()
            }
            var ctx = getContext("2d");
            ctx.clearRect(0, 0, parent.width, parent.height)
            ctx.save();
            ctx.strokeStyle = "black"
            ctx.lineWidth = 1

            drawFantasy();

            ctx.stroke();
            ctx.restore();
        }
    }
}
```

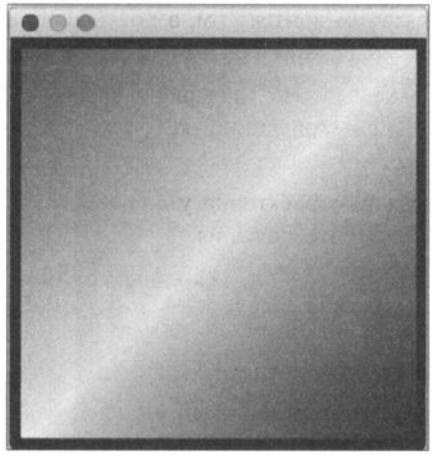


Рис. 56.13. Рисование градиента на элементе холста `Canvas`

При помощи элемента холста можно отображать линейные градиенты. Окно программы, показанное на рис. 56.13, отображает линейный градиент в прямоугольнике с рамкой.

В листинге 56.7 в секции свойства `onPaint` элемента холста `Canvas` вызовом метода `createLinearGradient()` мы создаем объект градиента и присваиваем его переменной `gradient`. Создаем три точки останова методами `addColorStop()`. Присваиваем объекту графического контекста свойство `fillStyle`, которое отвечает за заполнение созданного нами градиента, и отображаем заполненный прямоугольник (метод `fillRect()`). В завершении поверх заполненного прямоугольника отображаем прямоугольный контур (метод `strokeRect()`). Для цвета контурной линии используется голубой цвет, заданный в свойстве `strokeStyle` объекта контекста `ctx`. Оба прямоугольника имеют размеры, равные основному окну приложения, которые мы получаем посредством идентификатора `canv`.

Листинг 56.7. Рисование прямоугольника с градиентом на элементе холста

```
import QtQuick 2.8

Canvas {
    id: canv
    width: 320
    height: 320
    onPaint: {
        var ctx = getContext("2d")
        ctx.strokeStyle = "Blue"
        ctx.lineWidth = 15
        var gradient =
          ctx.createLinearGradient(canv.width, canv.height, 0, 0)
        gradient.addColorStop(0, "Indigo")
        gradient.addColorStop(0.5, "Bisque")
        gradient.addColorStop(1, "ForestGreen")
        ctx.fillStyle = gradient
        ctx.fillRect(0, 0, canv.width, canv.height)
        ctx.strokeRect(0, 0, canv.width, canv.height)
    }
}
```

Для вывода текста на холсте можно использовать метод `fillText()`. Контекст рисования предоставляет свойства, при помощи которых можно придать отображаемым объектам немного стилевых эффектов. На рис. 56.14 отображается текст с эффектом свечения (листинг 56.8).

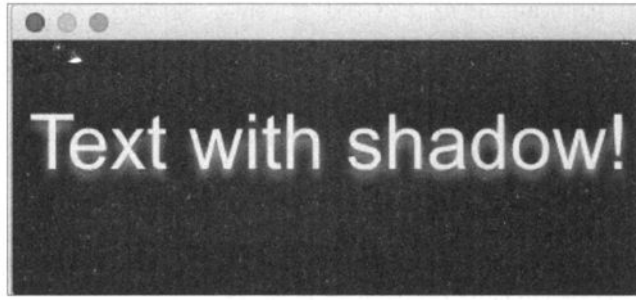


Рис. 56.14. Шрифт со свечением на элементе холста `Canvas`

В секции `onPaint` листинга 56.8 мы задаем свойству заполнения `fillStyle` черный цвет и отображаем заполненный прямоугольник методом `fillRect()`. Затем задаем желтый цвет для рисования контуров (свойство `strokeStyle`) и устанавливаем цвет для тени тоже желтым (свойство `shadowColor`). В нашем случае это конечно не тень, а свечение, так как оно отображается светлым цветом на темном фоне. Для достижения большего визуального эффекта осуществляем сдвиг нашего свечения по оси Y на 5 пикселов (свойство `shadowOffsetY`). Уровню размытия, которым управляет свойство `shadowBlur`, присваиваем значение 5. Далее устанавливаем тип и размер шрифта в свойстве `font`, а также и цвет его заполнения в свойстве `fillStyle`. Для отображения текста с эффектом свечения мы просто вызываем метод `fillText()`.

Листинг 56.8. Рисование текста со свечением на элементе холста

```
import QtQuick 2.8

Canvas {
    id: canv
    width: 400
    height: 160
    onPaint: {
        var ctx = getContext("2d")
        ctx.fillStyle = "Black"
        ctx.fillRect(0 , 0, canv.width, canv.height);

        ctx.strokeStyle = "Yellow"
        ctx.shadowColor = "Yellow";
        ctx.shadowOffsetY = 5;
        ctx.shadowBlur = 5;
        ctx.font = "48px Arial";
        ctx.fillStyle = "Yellow";
        ctx.fillText("Text with shadow!", 10, canv.height / 2);
    }
}
```

Шейдеры и эффекты

Под термином *шейдер* понимается программа, которая предназначена для исполнения не центральным процессором компьютера (CPU), а процессором видеокарты (GPU). Сама программа должна быть написана на языке GLSL (OpenGL Shading Language). Это своеобразный диалект языка C, обладающий целым рядом особенностей, которые потребуют дополнительного времени для изучения. Мощь шейдеров заключается в скорости выполнения графических алгоритмов, которая гораздо выше в сравнении с выполнением их центральным процессором компьютера. Шейдеры работают с текстурами и фрагментами растровых изображений, изменяют их и в результате создается новая текстура.

Изучать язык GLSL мы не станем, а рассмотрим лишь небольшой пример того, как использовать шейдеры в QML. Тех же, кого интересует язык GLSL, я отсылаю к книге Алексея Борескова «Разработка и отладка шейдеров» (**www.bhv.ru/books/book.php?id=13335**), найти ее также можно по следующему QR-коду (рис. 56.15):

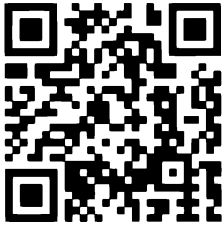


Рис. 56.15. QR-код для доступа на страницу книги «Разработка и отладка шейдеров»

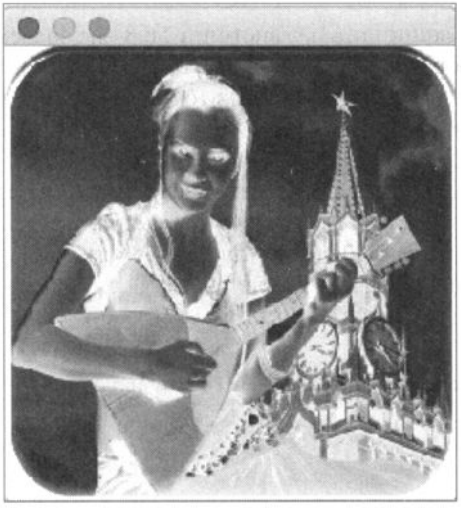


Рис. 56.16. Инвертирование растрового изображения при помощи шейдера

Для использования шейдеров Qt Quick предоставляет элемент `ShaderEffect`. Этот элемент переводит источник растрового изображения в текстуру и обрабатывает ее при помощи заданного алгоритма шейдера, а затем отображает получившуюся новую текстуру. На рис. 56.16 показано применение шейдера с алгоритмом инвертирования.

В листинге 56.9 мы создаем элемент `ShaderEffect`, внутри которого размещаем элемент `Image`. Элемент `Image` будет нам служить источником растрового изображения, поэтому мы его делаем невидимым, для чего присваиваем свойству `visible` значение `false`. Размеры элемента `Image` задаются в соответствии с загруженным в него растровым изображением (свойство `source`) при помощи свойств `sourceWidth` и `sourceHeight`.

Для того чтобы использовать элемент `Image` в алгоритме шейдера, необходимо создать свойство `source` и передать ему идентификатор элемента источника (`sourceImage`). Сам алгоритм шейдера помещается в строку и присваивается свойству `fragmentShader`. Строка с кодом шейдера будет при выполнении отправлена процессору графической карты (GPU),

который ее скомпилирует и выполнит. В конечном итоге у нас получилось что-то вроде шампуня «Проктер энд Гэмбл» в том смысле, что «два в одном флаконе». То есть мы смешиваем два исходных кода вместе: код QML с кодом шейдера.

Листинг 56.9. Использование шейдера инвертирования

```
import QtQuick 2.8
Rectangle {
    width: sourceImage.width
    height: sourceImage.height
    color: "Black"
    ShaderEffect {
        Image {
            id: sourceImage
            width: sourceWidth
            height: sourceHeight
            visible: false
            source: "qrc:/balalaika.png"
        }

        width: sourceImage.width
        height: sourceImage.height

        property variant source: sourceImage
        fragmentShader: "
          uniform sampler2D source;
          uniform lowp float qt_Opacity;
          varying highp vec2 qt_TexCoord0;
          void main() {
              gl_FragColor =
                abs(texture2D(source, qt_TexCoord0) * qt_Opacity - 1.0);
          }"
    }
}
```

На базе элемента шейдера в Qt Quick реализована целая библиотека эффектов `QGraphicalEffects`. Вы можете не только посмотреть их исходный код и использовать его в качестве учебного материала или для экспериментов, но также и применять эти эффекты в своих программах. Список доступных эффектов в версии 1.0 модуля `QGraphicalEffects` приведен в табл. 56.1.

***Таблица 56.1.** Эффекты модуля `QtGraphicsalEffects`*

Тип	Описание
`Blend`	Смешивает два источника изображений вместе
`BrightnessContrast`	Регулировка яркости и контраста
`ColorOverlay`	Изменяет цвет элемента, применяя к нему цвет наложения
`Colorize`	Устанавливает цвет в пространстве HSL-модели

Таблица 56.1 *(окончание)*

Тип	Описание
ConicalGradient	Рисует конический градиент
Desaturate	Уменьшает насыщенность цветов
DirectionalBlur	Эффект размытия в указанном направлении
Displace	Перемещает пикселы исходного элемента в соответствии с указанным источником смещения
DropShadow	Рисует тень за исходным элементом
FastBlur	Быстрый эффект размытия
GammaAdjust	Регулировка яркости
GaussianBlur	Эффект размытия высокого качества
Glow	Генерирует эффект сияния вокруг исходного элемента
HueSaturation	Изменяет цвета источника в пространстве HSL-модели
InnerShadow	Рисует внутреннюю тень
LevelAdjust	Регулировка уровней цвета в пространстве RGBA-модели
LinearGradient	Рисует линейный градиент
MaskedBlur	Эффект размытия с изменяющейся интенсивностью
OpacityMask	Маскирует исходный элемент другим элементом
RadialBlur	Направленное размытие в круговом направлении вокруг центральной точки
RadialGradient	Рисует радиальный градиент
RecangularGlow	Создает эффект свечения прямоугольной области
ResursiveBlur	Сильное размытие, которое достигается многократным применением размытия
ThresholdMask	Маскирует исходный элемент другим элементом и применяет пороговое значение
ZoomBlur	Направленный эффект размытия, который применяется к центральной точке источника

Теперь возьмем из табл. 56.1 один из эффектов и реализуем пример того, как можно его использовать в QML-программе. На рис. 56.17 показано использование эффекта размытия `FastBlur` с возможностью регулирования уровня размытия.

Итак, для эффекта быстрого размытия мы используем элемент `FastBlur` (листинг 56.10). В этом элементе создаем элемент источника растрового изображения `Image` с идентификатором `sourceImage` и делаем его при помощи свойства `visible` невидимым, так как он нам нужен только в роли «поставщика» растровых данных и не более. Указываем идентификатор элемента `Image` в свойстве `source` элемента эффекта размытия `FastBlur`. Далее мы просто создаем элемент ползунка `Slider` с диапазоном от 0 до 64, что соответствует минимальному и максимальному уровням размытия элемента `FastBlur`. При изменениях положения ползунка (свойство `onValueChanged`) мы задаем новое значение свойству `radius` элемента `FastBlur` и тем самым применяем заданный уровень размытия.

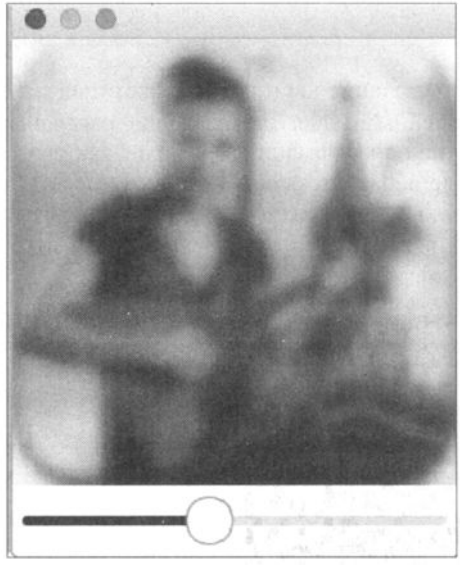


Рис. 56.17. Эффект размытия

Листинг 56.10. Эффект размытия

```
import QtQuick 2.8
import QtQuick.Controls 2.2
import QtGraphicalEffects 1.0

Column {
    FastBlur {
        id: blur
        Image {
            id: sourceImage
            visible: false
            source: "qrc:///Alina.jpg"
        }

        width: sourceImage.width;
        height: sourceImage.height
        source: sourceImage
    }
    Slider {
        id: sld
        width: sourceImage.width
        value: 0
        from: 0
        to: 64
        stepSize: 1
        onValueChanged: {
            blur.radius = value
        }
    }
}
```

Резюме

В этой главе мы рассмотрели варианты задания цветов и познакомились с двумя элементами: `Image` и `BorderImage`, предназначенными для отображения растровых изображений. Кроме того, мы изучили возможности трансформации и узнали, как создавать масштабируемые элементы управления, которые обладают способностью не искажаться при изменении их размеров. Мы также создали линейный градиент и проверили возможность рисования на элементе холста. А в заключение рассмотрели увлекательные возможности применения шейдеров и основанных на них эффектов.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/56-510/** или с помощью следующего QR-кода (рис. 56.18):



Рис. 56.18. QR-код для доступа на страницу комментариев к этой главе

ГЛАВА 57

Пользовательский ввод

Прежде чем подумать..., подумай.
Древняя мудрость

QML содержит механизм взаимодействия с пользователем. Существуют два концепта: для взаимодействия с мышью и взаимодействия с клавиатурой. Для обмена информацией между элементами можно использовать сигналы.

Область мыши

Для получения событий мыши в QML служат специальные элементы, которые называются `MouseArea` — *область мыши*. То, что они являются элементами, дает возможность устанавливать их расположение и изменять размеры как у обычных элементов. По своей сути они представляют собой прямоугольные области, определяющие регионы, в которых должен осуществляться ввод информации от мыши. Приведем в качестве примера прямоугольную область (листинг 57.1), которая реагирует на нажатие левой и правой кнопок мыши и отпускание кнопки мыши изменением цвета (рис. 57.1).



Рис. 57.1. Пример программы с использованием области мыши

В листинге 57.1 мы задали светло-зеленый прямоугольник с текстом (элемент `Text`), который отцентрирован по вертикали и горизонтали. И все, что мы хотим, — это иметь возможность реагировать на нажатия кнопок мыши пользователем. Для этого мы создаем область мыши (элемент `MouseArea`), которая является потомком элемента верхнего уровня `Rectangle`. Мы присваиваем свойству `fill` элемент предка, что позволяет провести фиксацию, которая осуществит заполнение всей области предка. Она и станет областью для получения и обработки событий, которые будет совершать пользователь с помощью мыши.

Мы задаем также свойства (`onPressed` и `onReleased`), которые будут исполнять код при нажатии и при отпускании кнопки мыши. Эти свойства являются на самом деле обработчика-

ми сигналов, а все обработчики сигналов имеют префикс on. В нашем примере, по аналогии с Qt, вы можете представить, что это слоты, которые будут вызываться при нажатии и отпускании кнопок мыши. В свойствах onPressed и onReleased мы осуществляем изменения цвета текста. Некоторые сигналы содержат в себе дополнительную информацию, к которой можно получить доступ, используя имя. В нашем примере обработчик сигнала onPressed получает дополнительный параметр mouse, который мы используем, чтобы узнать, какая именно из кнопок мыши была нажата. При нажатии правой кнопки заполняем окно красным цветом, в ином случае — синим. В свойстве acceptedButtons ограничиваем срабатывание наших обработчиков сигналов только этими кнопками.

Листинг 57.1. Область мыши

```
import QtQuick 2.8
Rectangle {
    width: txt.width + 20
    height: txt.height + 20
    color:  "lightgreen"
    Text {
        id: txt
        anchors.centerIn: parent
        text: "<h1>Click Me!<br>(use left or right mouse button)</h1>"
        horizontalAlignment: Text.AlignHCenter
    }

    MouseArea {
        anchors.fill: parent
        acceptedButtons: Qt.LeftButton | Qt.RightButton
        onPressed: {
            if (mouse.button == Qt.RightButton) {
                parent.color = "red"
            }
            else {
                parent.color = "blue"
            }
        }
        onReleased: parent.color = "lightgreen"
    }
}
```

Вот еще один способ сделать обработку события мыши (листинг 57.2) — теперь при наведении курсора мыши на область окна в нем должно будет происходить только изменение цвета (рис. 57.2).

В листинге 57.2 мы задали область мыши и идентификатор mousearea, которым воспользовались, чтобы сослаться на него из предка (элемент Rectangle) и получить информацию о том, находится ли курсор мыши поверх этой области или нет. В зависимости от этого мы и присваиваем цвет для заполнения окна.

В области мыши свойству hoverEnabled присваиваем значение true. Это позволяет элементу MouseArea реагировать на подведение мыши.

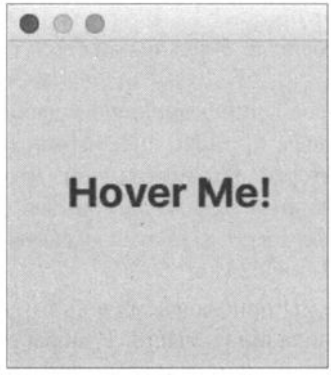


Рис. 57.2. Обработка события при наведении курсора мыши

Листинг 57.2. Обработка события мыши hover

```
import QtQuick 2.8
Rectangle {
   width: 200
   height: 200
   color: mousearea.containsMouse ? "red" : "lightgreen"

   Text {
      anchors.centerIn: parent
      text: "<h1>Hover Me!</h1>"
   }

   MouseArea {
      id: mousearea
      anchors.fill: parent
      hoverEnabled: true
   }
}
```

Абсолютно идентичного результата можно добиться и при помощи свойств `onEntered` и `onExit`. Эти свойства исполняют код, который вызывается при попадании курсора мыши в область элемента (то есть в область мыши) и при покидании им этой области. Это альтернативное решение реализовано в листинге 57.3.

Листинг 57.3. Обработка события мыши hover — альтернативное решение

```
import QtQuick 2.8
Rectangle {
    width: 200
    height: 200
    color: "lightgreen"
    Text {
        text: "<h1>Hover Me!</h1>"
        anchors.centerIn: parent
    }
```

```
    MouseArea {
        id: mousearea
        anchors.fill: parent
        hoverEnabled: true
        onEntered: parent.color = "red"
        onExited: parent.color = "lightgreen"
    }
}
```

В листинге 57.3 так же, как и в листинге 57.2, мы присваиваем свойству `hoverEnabled` значение `true` и разрешаем обработку события попадания мыши. В обработчике вхождения курсора мыши `onEntered` в область мыши `MouseArea` устанавливаем красный цвет, а в обработчике покидания `onExited` устанавливается светло-зеленый цвет.

Сигналы

В Qt многие объекты отсылают сигналы и элементы. Следует заметить, что язык QML не является исключением. Сигналы в QML — это просто события, которые прикреплены к свойствам с кодом для исполнения. В языке QML эти свойства называются *обработчиками сигналов*. С ними мы уже встречались в начале этой главы, и, как мы уже знаем, они имеют префикс `on`. Таким образом, сигнал представляет собой событие, а слот — это свойство с функцией, которое выполняется по этому событию.

Стандартные элементы определяют сигналы и обработчики — например, только что рассмотренный элемент `MouseArea` содержит свойства `onPressed`, `onReleased`, `onClicked` и т. д. Но этим все не ограничивается, и вы при желании и при необходимости можете добавлять в код свои собственные сигналы. Для добавления собственных сигналов существует ключевое слово `signal`. Его синтаксис выглядит следующим образом:

```
signal <name>[(<type> <value>, ...)]
```

К каждому сигналу автоматически создается обработчик со следующим синтаксисом:

```
on<Name>: <expression>
```

Теперь мы владеем теорией и самое время перейти к практике. Для этого реализуем пример с собственным сигналом (листинг 57.4). Сигнал нашего примера отправляется при изменении координат курсора мыши и отображает текущие его координаты (рис. 57.3).

В листинге 57.4, используя ключевое слово `signal`, мы вводим новый сигнал. Этот сигнал передает дополнительную информацию о текущей позиции курсора мыши в двух перемен-

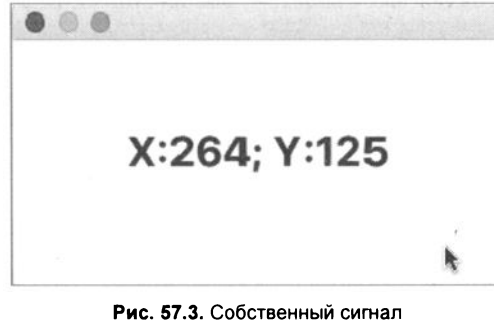


Рис. 57.3. Собственный сигнал

ных: x и y. Поскольку обработчики сигналов для сигналов создаются автоматически, сразу же после введения нового сигнала, только лишь введя начальные буквы on, мы увидим, что Qt Creator для выбора обработчиков отобразил нам список-подсказку, в котором мы обнаружим и обработчики для нашего сигнала с именами onMouseXChanged и onMouseYChanged. В этих обработчиках при помощи идентификатора txt мы будем присваивать элементу текста строку с текущим местоположением курсора и ссылаться на него.

Выслать сигнал очень просто. Для этого надо всего лишь выполнить вызов функции, который осуществляется из области мыши в свойстве обработки onMousePositionChanged из объекта предка. В сигнал передаются два значения: mouseX и mouseY — с текущим местоположением курсора мыши. Мы в этом примере предоставляем сигнал, которым можно пользоваться со стороны, а в целях демонстрации используем его внутри самого элемента.

Листинг 57.4. Собственный сигнал

```
import QtQuick 2.8
Rectangle {
    width: 300
    height: 150

    signal mousePositionChanged(int x, int y)

    onMousePositionChanged:
        txt.text = "<h1>X:" + x + "; Y:" + y + "</h1>"

    Text {
        id: txt
        text: "<h1>Move the Mouse</h1>"
        anchors.centerIn: parent
    }
    MouseArea {
        anchors.fill: parent
        hoverEnabled: true
        onMouseXChanged: parent.mousePositionChanged(mouseX, mouseY)
        onMouseYChanged: parent.mousePositionChanged(mouseX, mouseY)
    }
}
```

В следующем примере (листинг 57.5) мы реализуем элемент кнопки с текстом (рис. 57.4), которая при нажатии отправляет сигнал clicked. Этот сигнал на этот раз мы будем использовать извне.

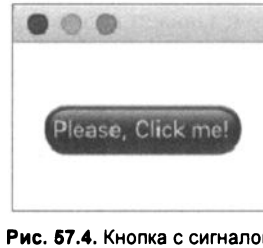


Рис. 57.4. Кнопка с сигналом

Элементы для повторного использования должны быть помещены в отдельные файлы. Имя файла является именем элемента. Код, приведенный в листинге 57.5, находится в файле Button.qml. Наш элемент кнопки базируется на элементе `BorderImage`. За текст кнопки отвечает внутренний элемент `Text`, поэтому, чтобы дать возможность его изменять извне, предоставляем свойство-синоним. Далее декларируем сигнал `clicked`, который будет отправляться при нажатии на нашу кнопку, поэтому он отправляется из обработчика `onClicked` области мыши. Размер кнопки должен зависеть от ее текста, для этого соединяем свойства размера текста с размерами элемента `BorderImage` и увеличиваем полями в 15 пикселов с обеих сторон. При нажатии и отпускании мыши нам нужны разные визуальные представления кнопки. Для того чтобы в зависимости от состояния кнопки загружать различные изображения, используем обработчики `onPressed` и `onReleased`.

Листинг 57.5. Кнопка с сигналом (файл Button.qml)

```
import QtQuick 2.8
BorderImage {
    property alias text: txt.text
    signal clicked;

    source: "qrc:/mybutton.png"
    width: txt.width + 15
    height: txt.height + 15
    border {left: 15; top: 12; right: 15; bottom: 12}

    Text {
        id: txt
        color:  "white"
        anchors.centerIn: parent
    }

    MouseArea {
        anchors.fill: parent
        onClicked: parent.clicked();
        onPressed: parent.source = "qrc:/mybuttonpressed.png"
        onReleased: parent.source = "qrc:/mybutton.png"
    }
}
```

Использование нового элемента нашей кнопки, который применен в листинге 57.6, ничем не отличается от использования любого другого стандартного элемента. С помощью свойства `text` мы присваиваем элементу кнопки начальный текст, а при нажатии кнопки в свойстве обработки сигнала `onClicked` присваиваем кнопке другой текст.

Листинг 57.6. Использование кнопки

```
import QtQuick 2.8
Item {
    width: 150
    height: 100
```

```
    Button {
        anchors.centerIn: parent
        text: "Please, Click me!"
        onClicked: {
            text = "Clicked!"
        }
    }
}
```

Так как связанные свойства тоже генерируют события, то сигналы можно практически всегда заменить свойствами. Разница проистекает из их натуры. Сигналы отправляются в одном направлении — от отправителя к получателю. Получатель при этом не может изменить в выславшем элементе принятые значения. Со свойством же все обстоит иначе — значения могут быть изменены, что также может привести к изменениям в поведении других элементов, которые подсоединены к этим свойствам. Поэтому сигналы лучше использовать в случаях, например, когда взаимодействие должно осуществляться между автономными элементами. Свойства же более целесообразно использовать для связи элементов внутри элемента родителя. Но это не правило, а только рекомендация, и поэтому, что вы предпочтете использовать в том или ином случае: сигналы или свойства — решать вам!

Для того чтобы идея была более понятна, реализуем альтернативное решение для нашей кнопки и заменим сигнал `clicked` свойством (листинг 57.7).

Рассмотрим этот листинг и остановимся только на его различиях с листингом 57.5. Мы заменили сигнал `clicked` одноименным свойством и задали ему логический тип. Значения этого свойства изменяем в элементе области мыши в обработчиках `onPressed` и `onReleased`.

Листинг 57.7. Кнопка со свойством (файл Button.qml)

```
import QtQuick 2.8
BorderImage {
    property alias text: txt.text
    property bool clicked;

    source: "qrc:/mybutton.png"
    width: txt.width + 15
    height: txt.height + 15
    border {left: 15; top: 12; right: 15; bottom: 12}

    Text {
        id: txt
        color:  "white"
        anchors.centerIn: parent
    }

    MouseArea {
        anchors.fill: parent
        onPressed: {
            parent.source = "qrc:/mybuttonpressed.png"
            parent.clicked = false
        }
```

```
        onReleased: {
            parent.source = "qrc:/mybutton.png"
            parent.clicked = true
        }
    }
}
```

Разница в использовании элемента кнопки, приведенного в листингах 57.6 и 57.8, заключается только в том, что свойство обработки события называется уже не `onClicked`, а `onClickedChanged`, так как теперь генерируется событие изменения значения свойства.

Листинг 57.8. Использование кнопки

```
import QtQuick 2.8
Item {
    width: 150
    height: 100
    Button {
        anchors.centerIn: parent
        text: "Please, Click me!"
        onClickedChanged: {
            text = "Clicked!"
        }
    }
}
```

Ввод с клавиатуры

В основном ввод с клавиатуры можно обрабатывать двумя элементами: `TextInput` и `TextEdit`. Как видно из названий, оба элемента предназначены для работы с текстом. Элемент `TextInput` работает с одной строкой текста, а элемент `TextEdit` — с многострочным текстом, аналогично тому, как это в библиотеке Qt делают классы `QLineEdit` и `QTextEdit`.

В следующем примере (листинг 57.9) возьмем элемент `TextInput` и отобразим его в окне (рис. 57.5).

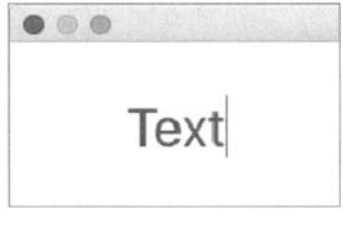


Рис. 57.5. Поле текстового ввода

В листинге 57.9 мы задаем элемент однострочного ввода. Присваиваем ему начальный текст и, чтобы пользователь мог делать ввод, устанавливаем фокусу (свойство `focus`) значение `true`. Размер самого элемента будет соответствовать введенному в него тексту. Что же произойдет в том случае, если элемент `TextInput` не содержит текста совсем? Тогда его ширина окажется равна 0, и поэтому не будет возможности его нажать. Для того чтобы избежать подобных ситуаций, элементу с помощью свойства `width` нужно задать ширину.

Листинг 57.9. Поле для ввода текста

```
import QtQuick 2.8
Rectangle {
    width: 200
    height: 100
    TextInput {
        anchors.centerIn: parent
        color:  "red"
        text: "Text"
        font.pixelSize: 32
        focus: true
    }
}
```

Фокус

Фокус между элементами ввода может быть перемещен клавишами управления курсором и табуляции. Присвоение фокуса работает следующим образом. Если содержится всего лишь один элемент `TextInput`, то он получает фокус автоматически. Если же их больше одного, то пользователь может сам изменять фокус нажатиями мыши. Если необходимо установить фокус, то, как мы уже знаем из прошлого примера, нужно воспользоваться свойством `focus`.

Следующий пример (листинг 57.10) иллюстрирует использование двух текстовых полей одновременно. Текстовый элемент с активным фокусом изменяет цвет текста с черного на красный (рис. 57.6).

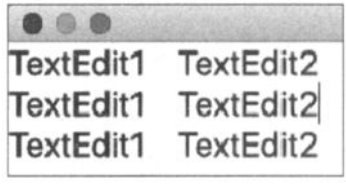


Рис. 57.6. Фокус между двумя полями ввода

В листинге 57.10 имеются два элемента текстового ввода. В первом из них с помощью свойства `focus` устанавливаем фокус. Здесь мы используем фиксаторы — для уверенности в том, что ширина никакого из них не станет равной нулю. Это значит, что в любом случае у пользователя останется возможность нажатием клавиши табуляции поменять фокус даже тогда, когда элемент не будет содержать никакого текста.

Листинг 57.10. Фокус между двумя полями ввода

```
import QtQuick 2.8
Item {
    width: 200
    height: 80
    TextEdit {
        anchors.left: parent.left
        anchors.right: parent.horizontalCenter
        anchors.top: parent.top
```

```
        anchors.bottom: parent.bottom
        text: "TextEdit1\nTextEdit1\nTextEdit1"
        font.pixelSize: 20
        color: focus ? "red" : "black"
        focus: true
    }

    TextEdit {
        anchors.left: parent.horizontalCenter
        anchors.right: parent.right
        anchors.top: parent.top
        anchors.bottom: parent.bottom
        text: "TextEdit2\nTextEdit2\nTextEdit2"
        font.pixelSize: 20
        color: focus ? "red" : "black"
    }
}
```

Теперь продемонстрируем возможность управления фокусом, используя нетекстовые элементы, так как они могут тоже иметь фокус. В следующем примере (листинг 57.11) представлены два прямоугольника: один внешний, а другой внутренний (рис. 57.7). При получении фокуса прямоугольник изменяет свой цвет со светло-зеленого на красный. Изменение происходит при нажатии пользователем клавиши табуляции.

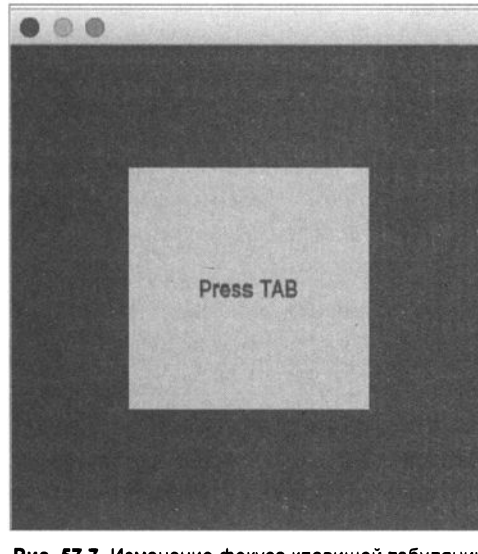


Рис. 57.7. Изменение фокуса клавишей табуляции

В листинге 57.11 для управления фокусом при помощи клавиши табуляции мы используем так называемое *прикрепляемое свойство* `KeyNavigation.tab`. Это свойство не является стандартным свойством элемента `Rectangle`. Для того чтобы можно было внешне отличить прямоугольник с фокусом от прямоугольника без фокуса, мы в обоих прямоугольниках в зависимости от значения `focus` свойству `color` присваиваем цвет. По умолчанию даем фокус внутреннему прямоугольнику (свойство `focus`). При возникновении ситуации нажатия на

клавишу табуляции отрабатывается код свойства `KeyNavigation.tab`, и в нашем примере, если внутренний элемент имел фокус, происходит передача фокуса внешнему прямоугольнику, и наоборот.

Листинг 57.11. Изменение фокуса клавишей табуляции

```
import QtQuick 2.8
Rectangle {
    width: 300
    height: 300
    color:  focus ? "red" : "lightgreen"
    KeyNavigation.tab: childrect

    Rectangle {
        id: childrect
        width: 150
        height: 150
        anchors.centerIn: parent
        color:  focus ? "red" : "lightgreen"
        KeyNavigation.tab: parent
        focus: true

        Text {
            anchors.centerIn: parent
            text: "Press TAB"
        }
    }
}
```

Для управления фокусом мы могли бы вместо клавиши табуляции использовать, например, клавиши управления курсором, задействовав свойства `KeyNavigation.right`, `KeyNavigation.left`, `KeyNavigation.up` и `KeyNavigation.down`.

«Сырой» ввод

Очень часто нужно обеспечить возможность доступа на уровне событий клавиатуры с полной информацией о событии. Элементы такой возможностью не обладают, поэтому для этого применяется прикрепляемое свойство `Keys`.

В следующем примере (листинг 57.12) мы используем элемент текста, позицию которого можно изменять при помощи клавиш управления курсором (рис. 57.8).

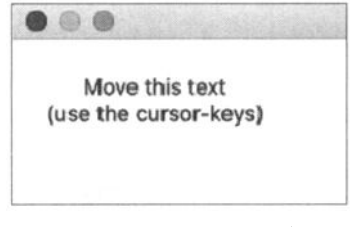


Рис. 57.8. Перемещение элемента при помощи клавиш управления курсором

В листинге 57.12 мы реализовали четыре обработчика: один — для нажатия клавиши «стрелка влево» <←> (Keys.onLeftPressed), другой — для нажатия клавиши «стрелка вправо» <→> (Keys.onRightPressed), а также для нажатия клавиш «стрелка вниз» <↓> (Keys.onDownPressed) и «стрелка вверх» <↑> (Keys.onUpPressed). В этих обработчиках мы увеличиваем позиции в нужном направлении на 3 пиксела. Основному текстовому элементу мы задаем фокус.

Листинг 57.12. Перемещение элемента при помощи клавиш управления курсором

```
import QtQuick 2.8
Rectangle {
    width: 200
    height: 100
    Text {
        x: 20;
        y: 20;
        text: "Move this text<br>(use the cursor-keys)"
        horizontalAlignment: Text.AlignHCenter
        Keys.onLeftPressed: x -= 3
        Keys.onRightPressed: x += 3
        Keys.onDownPressed: y += 3
        Keys.onUpPressed: y -= 3
        focus:  true
    }
}
```

Листинг 57.13 демонстрирует, как можно обойтись и одним обработчиком, а также контролировать все нажатия в одном свойстве onPressed, получая дополнительную информацию события (event.key) и сравнивая его значение с перечислениями клавиатуры Qt (см. табл. 14.2).

Листинг 57.13. Обработка событий клавиатуры

```
Keys.onPressed: {
    if (event.key == Qt.Key_Left) {
        x -= 3;
    }
    else if (event.key == Qt.Key_Right) {
        x += 3;
    }
    else if (event.key == Qt.Key_Down) {
        y += 3;
    }
    else if (event.key == Qt.Key_Up) {
        y -= 3;
    }
}
```

События клавиатуры при помощи Keys.forwardTo могут пересылаться и другим элементам для дальнейшей обработки, причем допускаются также и списки объектов. Если мы допол-

ним цепочку if-операторов листинга 57.13 двумя следующими if-операторами листинга 57.14, то у нас появится, благодаря константам Key_Plus и Key_Minus, возможность увеличивать и уменьшать размер шрифта нажатием на клавиши <+> и <–>.

Листинг 57.14. Увеличение шрифта нажатием на клавиши <+> и <–>

```
else if (event.key == Qt.Key_Plus) {
    font.pixelSize++;
}
else if (event.key == Qt.Key_Minus) {
     font.pixelSize--;
}
```

А для распознавания нажатий клавиш с буквами можно воспользоваться константами от Key_A до Key_Z.

Мультитач

Термин *мультитач* (Multi-touch) в переводе с английского языка означает множественное касание. Как работать с этим типом взаимодействия с пользователем в C++-программах, мы уже подробно разобрались в *главе 14*. Теперь пришло время немного прояснить, как это делается в QML.

В QML за область региона, где будет осуществляться мультитач, отвечает элемент MultiPointTouchArea. Этот элемент содержит в себе элементы обработки события касания TouchPoint — их ровно столько, сколько одновременных касаний мы намереваемся обрабатывать. По своей функциональной сути TouchPoint можно грубо сравнить с элементом обработки события мыши MouseArea. Представьте себе ситуацию, когда к вашему устройству подключены сразу несколько мышек, и их всех перемещают одновременно. В этом случае был бы необходим механизм, для того чтобы обрабатывать все эти события в отдельных элементах. Именно такими элементами и являются элементы TouchPoint. Только в роли мышек выступают пальцы и, в отличие от мышек, не всегда все они должны высылать события, потому что пользователь не обязательно соприкасается своими пальцами с поверхностью сенсорной панели (тачпада) своего компьютера или с сенсорным экраном мобильного устройства, а если и соприкасается, то количество пальцев в разные промежутки времени может изменяться. Более того, соприкосновения одновременно могут производиться сразу несколькими пользователями, то есть теоретически это могут быть 10 и более пальцев одновременно. На рис. 57.9 показано окно приложения, способное обрабатывать до пяти одновременных прикосновений и отображать их места.

В листинге 57.15 мы создаем элемент MultiPointTouchArea и заполняем им всю рабочую область окна (элемент Rectangle). При помощи свойств minimumTouchPoints и maximumTouchPoint мы задаем диапазон одновременно возможных нажатий: от 1 до 5. Далее необходимо создать 5 элементов для обработки каждого из возможных событий касания TouchPoint, и массив из этих элементов присваивается свойству touchPoints. Для визуализации информации элементов TouchPoint мы создаем с помощью повторителя (элемент Repeater) прямоугольные области белого цвета. Эти прямоугольные области будут отображаться только в случае прикосновений (свойство pressed элемента TouchPoint) и на позициях, на которых произошло нажатие (свойства x и y элемента TouchPoint).

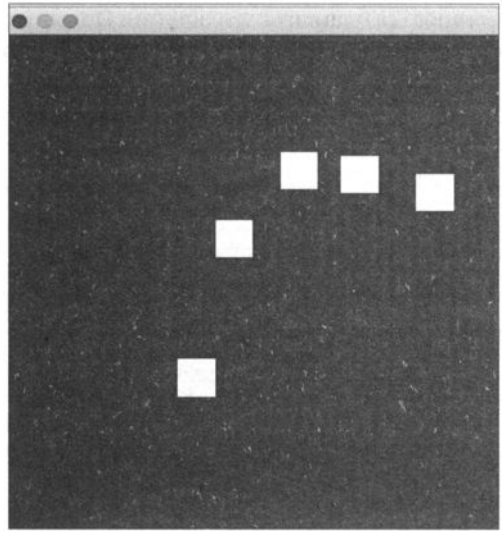


Рис. 57.9. Прикосновение к области элемента `MultiPointTouchArea` пятью пальцами

Листинг 57.15. Обработка мультитач-событий (файл main.qml)

```
import QtQuick 2.8
Rectangle {
    width: 400
    height: 400
    color:  "black"

    MultiPointTouchArea {
        anchors.fill: parent
        minimumTouchPoints: 1
        maximumTouchPoints: 5
        touchPoints: [
            TouchPoint {},
            TouchPoint {},
            TouchPoint {},
            TouchPoint {},
            TouchPoint {}
        ]
        Repeater {
            model: parent.touchPoints
            Rectangle {
                color: "white";
                x: modelData.x;
                y: modelData.y;
                width: 30;
                height: 30
                visible: modelData.pressed
            }
        }
    }
}
```

В табл. 57.1 приведены некоторые свойства элемента `TouchPoint`. Помните, что каждое из свойств в QML всегда обладает соответствующим методом обработки `on<ИмяСвойства>Changed`.

***Таблица 57.1.** Свойства элемента `TouchPoint`*

Имя свойства	Описание
`pressed`	При касании имеет значение `true`, в противном случае `false`
`pressure`	Сила нажатия (не все устройства предоставляют эту информацию)
`previousX, previousY`	Предыдущие координаты позиций касания
`startX, startY`	Начальные координаты позиций касания
`x, y`	Текущие координаты позиций касания

Резюме

Взаимодействие с мышью реализуется при помощи специального элемента `MouseArea`. Этот элемент ничего не отображает, а просто задает регион для получения и обработки событий мыши.

Для всех заданных сигналов автоматически создаются обработчики с префиксом `on`. Благодаря автодополнению, в программе Qt Creator мы сразу увидим сгенерированное свойство обработки.

Для ввода текста предоставляются элементы `TextInput` и `TextEdit`, но очень часто нужна работа с фокусом и вводом с клавиатуры на уровне событий. Это достигается при помощи прикрепляемых свойств `KeyNavigation` и `Keys`.

Мы познакомились также с возможностями обработки событий множественного касания «мультитач».

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/57-510/** или с помощью следующего QR-кода (рис. 57.10):



Рис. 57.10. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 58

Анимация

> Если хочешь иметь то, чего раньше не имел, научись делать то, чего раньше не умел.
>
> *Дж. Рон*

Об анимации мы уже говорили в *главе 22*. В связи с этим у вас, наверное, закрался вопрос, а зачем же нужны в одной и той же книге две главы об анимации? Конечно, все эффекты, описанные в этой главе, можно реализовать и с помощью Qt, но на языке QML это достигается гораздо меньшими усилиями, так как здесь используется абсолютно иной подход. Поэтому и возникла необходимость в написании еще одной главы.

На самом деле анимация нужна не только для создания визуальных эффектов — она может помочь вам обратить внимание пользователя на ту или иную часть действий, выполняемых вашей программой. Следовательно, анимация является дополнением пользовательского интерфейса, которое помогает сделать его еще более понятным. Но нужно сразу же предостеречь — не добавляйте необдуманно слишком много эффектов, потому что вы можете получить прямо противоположный результат. Если вам нужно заострить внимание пользователя, то анимация — самый лучший способ добиться этого. Но если надо, чтобы пользователь проигнорировал что-либо, то от применения анимации в этом случае лучше отказаться, так как пользователи не любят, когда их внимание отвлекают понапрасну. Не забывайте, что анимация прежде всего призвана произвести впечатление естественности, поэтому и используйте ее соответствующим образом. И тогда ваша программа приобретет магнетизм и будет притягивать к себе с каждым разом все больше и больше пользователей.

Как вы знаете, QML — это описательный язык, и вы наверняка уже озадачены: как же можно с его помощью описать динамическое поведение анимации? Наберитесь терпения — сейчас мы разберемся, какие возможности анимации есть на «вооружении» в языке QML, и узнаем, что все варианты анимации применяются к свойствам элементов стандартных типов QML: `real`, `int`, `color`, `rect`, `point`, `size` и `vector3d`.

Анимация при изменении свойств

Для анимации свойств существует элемент `PropertyAnimation`. С его помощью можно изменять в один и тот же момент времени сразу несколько свойств одновременно, например размер и прозрачность.

В следующем примере (листинг 58.1) покажем, как одновременно изменить два свойства: `x` и `y`, в результате чего растровое изображение проделает путь из верхнего левого угла в нижний правый угол (рис. 58.1).

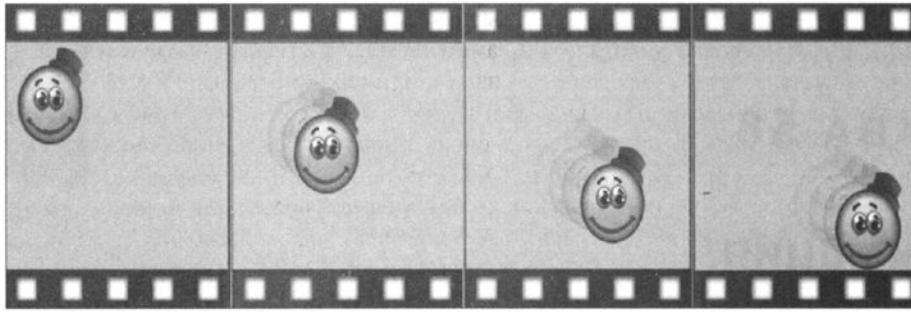


Рис. 58.1. Анимация при изменении координат x и y

В листинге 58.1 мы создаем элемент растрового изображения `Image` и элемент анимации `PropertyAnimation` внутри элемента `Rectangle`. В свойстве `target` элемента анимации определяется, к свойствам какого элемента должна применяться анимация. В свойстве `properties` в виде строки указываются имена свойств, которые будут подвергаться изменениям. Свойство `from` задает начальное значение, а свойство `to` — конечное значение для анимации. В нашем примере указано, что значение для обоих свойств должно изменяться от 0 до высоты прямоугольника минус высота растрового изображения, — чтобы оно не исчезло за пределы прямоугольника. Время, которое потребуется элементу для перемещения из одного угла в другой, мы задаем в свойстве `duration` в миллисекундах, и в нашем примере оно составляет 1,5 сек. Анимация не запускается по умолчанию, поэтому, чтобы произошел ее запуск при старте программы, необходимо присвоить свойству `running` значение `true`. Свойство `loop` управляет числом повторений анимации — в нашем случае мы устанавливаем значение `Animation.Infinite`, что соответствует бесконечному числу повторений.

Листинг 58.1. Анимация при изменении координат x и y

```
import QtQuick 2.8
Rectangle {
    color: "lightgreen"
    width: 300
    height:  300
    Image {
        id: img
        x: 0
        y: 0
        source: "qrc:/happyos.png"
    }
    PropertyAnimation {
        target: img
        properties: "x,y"
        from: 0
        to: 300 - img.height
        duration: 1500
        running: true
        loops: Animation.Infinite
        easing.type: Easing.OutExpo
    }
}
```

Для создания эффекта изменяющейся скорости анимации применяются смягчающие линии. Например, анимация может ускоряться в самом начале, достигать максимальной скорости в середине, а затем начать замедляться. В нашем примере (см. листинг 58.1) мы используем стандартный идентификатор типа смягчающей линии и присваиваем его свойству `easing.type`. Библиотека Qt предоставляет очень много готовых типов динамик смягчающих линий, которые можно использовать в QML-коде, все они приведены в табл. 22.1 этой книги. Кроме того, в выборе нужного типа динамики смягчающей линии вам может помочь Qt Creator — нужно только лишь поставить курсор мыши на свойство `easing.type`, вызвать контекстное меню и выбрать в нем пункт **Show Qt Quick Toolbar**, после чего вашему взору предстанет очень симпатичное окно, изображенное на рис. 58.2.

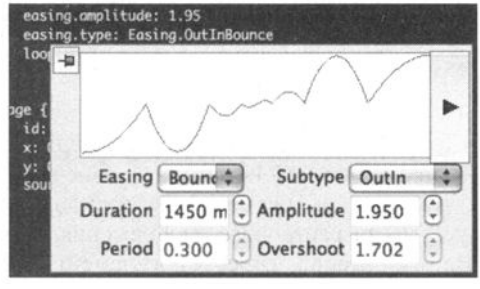


Рис. 58.2. Окно интерактивной настройки анимации

Это окно имеет различные элементы настройки — не бойтесь поэкспериментировать с ними. Изменение значений в этом окне сразу же вызывает изменение их значений в исходном тексте программы. Обратите внимание на кнопку со стрелкой у правой границы окна — это очень полезная функция. Нажав на эту кнопку, вы сразу же увидите, как будет выглядеть динамика анимации, и сможете принять решение, подходит она вам или нет.

Итак, мы рассмотрели элемент `PropertyAnimation`. Этот элемент является базовым для более специфичных типов элементов анимаций:

- `NumberAnimation` — для изменения числовых свойств;
- `ColorAnimation` — для изменения свойств цвета;
- `RotationAnimation` — для поворота элементов.

Анимация для изменения числовых значений

Элемент анимации числовых значений `NumberAnimation` предоставляет, в сравнении с элементом `PropertyAnimation`, более эффективную реализацию для анимирования свойств типа `real` и `int`.

Следующий пример (листинг 58.2) демонстрирует использование этой анимации для изменения значения свойства ширины элемента (рис. 58.3).

В листинге 58.2 у нас создаются два прямоугольника: один основной — светло-зеленого цвета и другой, встроенный в него, — красного цвета (свойства `color`). Оба имеют одинаковую высоту 100 (свойство `height`). Элемент `NumberAnimation` изменяет значения численных свойств и применяется к свойству с помощью ключевого слова `"on"` — в нашем случае это свойство `width`. Далее все инструкции (`from`, `to`, `duration`, `easing.type`) прописаны аналогично их использованию с элементом `PropertyAnimation` (см. описание листинга 58.1).

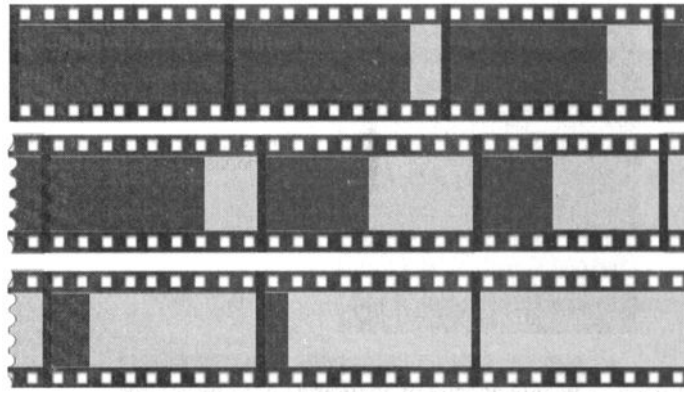


Рис. 58.3. Пример анимации с использованием элемента `NumberAnimation`

Листинг 58.2. Анимация с использованием элемента `NumberAnimation`

```
import QtQuick 2.8
Rectangle {
    width: 300
    height: 100
    color: "lightgreen"

    Rectangle {
        x: 0
        y: 0
        height: 100
        color: "red"
        NumberAnimation on width {
            from: 300
            to: 0
            duration: 2000
            easing.type: Easing.InOutCubic
        }
    }
}
```

Анимация с изменением цвета

Элемент `ColorAnimation` управляет изменением цвета элементов и для изменения значения цвета предоставляет свойства `from` и `to`.

В следующем примере (листинг 58.3) осуществляется изменение цвета основного элемента окна от одного значения цвета к другому (рис. 58.4).

В листинге 58.3 мы задаем в свойствах `from` и `to` начальное и конечное значения цветов, то есть значения, начиная от которого и заканчивая которым должно произойти изменение цвета, а также время продолжительности выполнения этого изменения — 1,5 сек (свойство `duration`). Запускаем анимацию установкой значения `true` в свойстве `running` и устанавливаем бесконечное количество раз ее повторения.

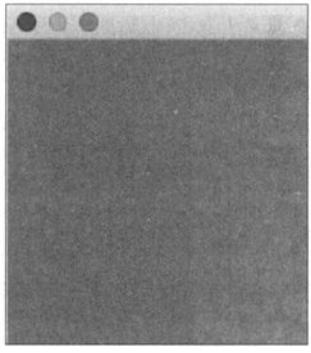


Рис. 58.4. Пример анимации с использованием элемента `ColorAnimation`

Листинг 58.3. Анимация с использованием элемента `ColorAnimation`

```
import QtQuick 2.8
Rectangle {
    width: 200
    height: 200
    ColorAnimation on color {
        from: Qt.rgba(1, 0.5, 0, 1)
        to: Qt.rgba(0.5, 0, 1, 1)
        duration: 1500
        running: true
        loops: Animation.Infinite
    }
}
```

Анимация с поворотом

Элемент `RotationAnimation` описывает поворот элемента. Он предоставляет свойство `direction`, с помощью которого можно задавать направление поворота. Это свойство может принимать следующие значения:

- `RotationAnimation.Clockwise` — поворот по часовой стрелке;
- `RotationAnimation.Counterclockwise` — поворот против часовой стрелки;
- `RotationAnimation.Shortest` — поворот в сторону наименьшего угла поворота от значения, заданного в свойстве `from`, и до значения — `to`.

По умолчанию поворот осуществляется в направлении часовой стрелки.

В следующем примере (листинг 58.4) мы выполняем поворот элемента растрового изображения при помощи элемента анимации поворота `RotationAnimation` (рис. 58.5).

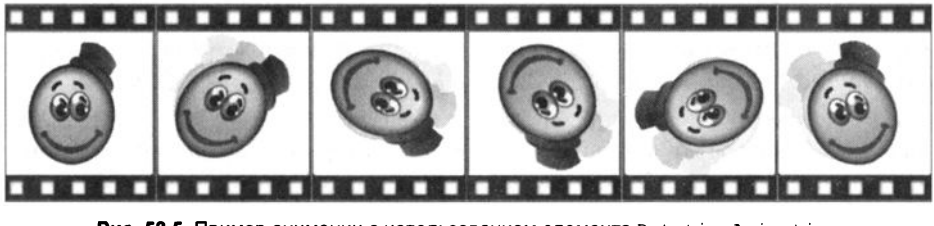


Рис. 58.5. Пример анимации с использованием элемента `RotationAnimation`

В листинге 58.4 мы задаем элемент анимации поворота `RotationAnimation` внутри элемента растрового изображения `Image`. Анимация длится 2 секунды и осуществляется по часовой стрелке (действие по умолчанию) от 0 до 360 градусов.

Листинг 58.4. Анимация с использованием элемента RotationAnimation

```
import QtQuick 2.8
Rectangle {
    width: 150
    height: 150
    Image {
        source: " qrc:/happyos.png"
        anchors.centerIn: parent
        smooth: true

        RotationAnimation on rotation {
            from: 0
            to: 360
            duration: 2000
            loops: Animation.Infinite
            easing.type: Easing.InOutBack
        }
    }
}
```

Анимации поведения

Часто бывает так, что хочется выполнить анимацию в момент, когда происходит изменение какого-либо из свойств элемента. Например, чтобы растровое изображение следовало за указателем мыши из одной позиции в другую, и не просто было бы примитивно «приклеено» к указателю, а красиво анимировало при смене позиций. Это значит, что каждый раз, когда свойство изменяется, должна запускаться анимация. Именно для этого и существует элемент анимации поведения `Behavior`.

Следующий пример (листинг 58.5) выполняет анимацию растрового изображения при перемещении указателя мыши (рис. 58.6).

В листинге 58.5 мы задаем две отдельные анимации поведения `Behavior`, которые реагируют на изменение свойств координат `x` и `y` элемента растрового изображения `Image`. Внутри этих элементов используются элементы `NumberAnimation` и задается длительность проведения анимации, равная 1 сек. Тем самым мы создали поведение, которое каждый раз при изменении позиции растрового изображения запускает эту анимацию. Изменение же самих свойств элемента растрового изображения `Image` осуществляется из элемента области мыши `MouseArea` в свойствах `onMouseXChanged` и `onMouseYChanged`.

Листинг 58.5. Анимация поведения

```
import QtQuick 2.8
Rectangle {
    id: rect
    width: 360
    height: 360
```

```
    Image {
        id: img
        source: " qrc:/happyos.png"
        x: 10
        y: 10
        smooth: true
        Text {
            anchors.verticalCenter: img.verticalCenter
            anchors.top: img.bottom
            text: "Move the mouse!"
        }
        Behavior on x {
            NumberAnimation {
                duration: 1000
                easing.type: Easing.OutBounce
            }
        }
        Behavior on y {
            NumberAnimation {
                duration: 1000
                easing.type: Easing.OutBounce
            }
        }
    }
    MouseArea {
        anchors.fill: rect
        hoverEnabled: true

        onMouseXChanged: img.x = mouseX
        onMouseYChanged: img.y = mouseY
    }
}
```

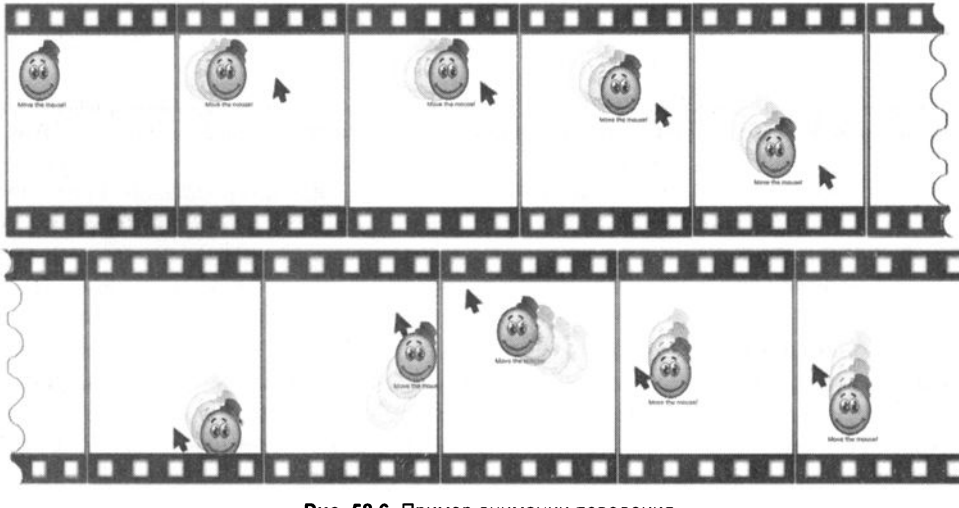


Рис. 58.6. Пример анимации поведения

Параллельные и последовательные анимации

До сих пор мы рассматривали анимации, которые выполняют лишь одно действие. Как же быть в тех случаях, когда нужно выполнить с объектом несколько различных анимаций? В этих случаях анимации могут быть объединены в одну общую анимацию. Это делается при помощи специальных элементов групп. Группы анимаций могут выполняться параллельно и последовательно:

- *последовательные анимации* задаются при помощи элемента `SequentialAnimation`. В этом элементе каждый потомок анимации выполняется в порядке очереди. Например, анимация изменения размеров следует перед анимацией изменения прозрачности;
- элемент `ParallelAnimation` задает *параллельную группу*, в которой потомки анимации запускаются и исполняются одновременно. Например, изменение размеров будет происходить одновременно с изменением прозрачности.

Параллельные и последовательные анимации могут быть вложены друг в друга.

Если элемент группы опущен, то анимации будут выполняться параллельно, например:

```
Rectangle {
....
    PropertyAnimation on x {to: 50; duration: 1000}
    PropertyAnimation on y {to: 50; duration: 1000}
....
}
```

Сначала мы реализуем пример параллельной анимации, в которой станем изменять размеры растрового изображения и вместе с тем изменять его прозрачность (листинг 58.6). При этом наша картинка (рис. 58.7) должна будет в основном окне одновременно как бы появляться из ниоткуда, увеличиваться в размерах и растворяться в никуда, а потом все заново.

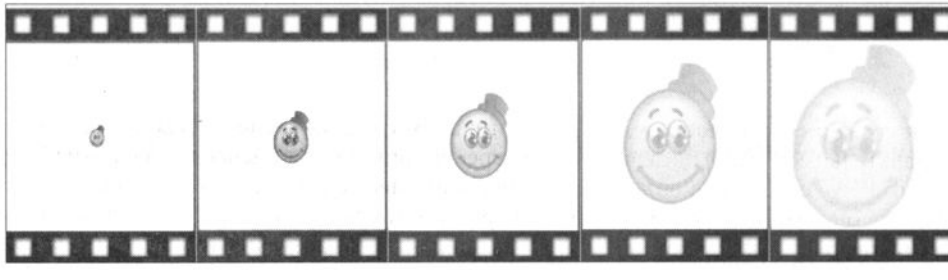


Рис. 58.7. Пример параллельной анимации

В листинге 58.6 мы изменяем сразу два свойства одновременно: это увеличение изображения (свойство `scale`) с фактором от 1 до 3 и изменение прозрачности (свойство `opacity`) от полностью прозрачного до совсем непрозрачного (от 1 до 0). В обоих случаях в качестве объекта, по отношению к которому применяются анимации, используется элемент растрового изображения `Image`, для этого мы устанавливаем его в свойстве `target` при помощи идентификатора `img`. Продолжительность у обеих анимаций одинакова и составляет 2 сек (свойство `duration`). В самом конце мы делаем так, чтобы наша анимация запускалась сразу же после создания элемента, присваивая свойству `running` значение `true`, и задаем бесконечное количество раз ее выполнения, присвоив свойству `loop` значение `Animation.Infinite`.

Листинг 58.6. Параллельная анимация

```
import QtQuick 2.8
Rectangle {
    width: 400
    height: 400
    Image {
        id: img
        source: " qrc:/happyos.png"
        smooth: true
        anchors.centerIn: parent
    }

    ParallelAnimation {
        NumberAnimation {
            target: img
            properties: "scale"
            from: 0.1;
            to: 3.0;
            duration: 2000
            easing.type: Easing.InOutCubic
        }
        NumberAnimation {
            target: img
            properties: "opacity"
            from: 1.0
            to: 0;
            duration: 2000
        }
        running: true
        loops: Animation.Infinite
    }
}
```

Теперь, я думаю, что с параллельными анимациями все понятно, и мы можем перейти к последовательным. Чтобы стало более интересно, придумаем сценарий посложнее. Представим, что у нас есть некий объект, который висит наверху, и если мы нажатием мыши его отпустим, то он упадет. После падения он перевернется вокруг своей оси, полежит некоторое время на полу, а затем самостоятельно поднимется вверх (листинг 58.7). На рис. 58.8 изображены все стадии перемещения объекта.

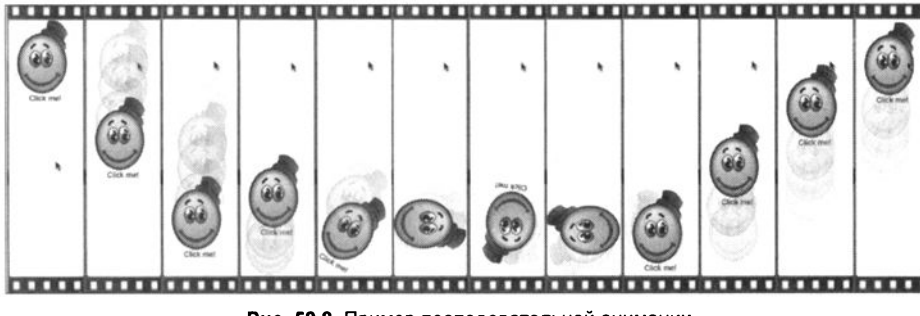


Рис. 58.8. Пример последовательной анимации

В листинге 58.7 мы делаем так, чтобы наша анимация стартовала при щелчке мыши. Для этого используем свойство `onClicked` области мыши `MouseArea`. В этом свойстве мы запускаем анимацию (свойство `running`), ссылаясь на ее идентификатор `anim`.

Теперь перейдем к самóй последовательной анимации (элемент `SequentialAnimation`). В первом ее элементе `NumberAnimation` мы реализуем падение нашего объекта от начальной точки 20 (свойство `from`) и до конечной 300 (свойство `to`) по вертикали (свойство `y`). Падение длится 1 сек (свойство `duration`).

После падения наш объект должен повернуться вокруг своей оси, поэтому вторым элементом анимации мы задействуем элемент `RotationAnimation`, в котором при помощи свойств `from` и `to` повернем объект от 0 до 360 градусов. Направление поворота мы задаем при помощи свойства `direction`, хотя в этом случае мы могли бы его и не указывать, так как поворот по часовой стрелке (значение `RotationAnimation.Clockwise`) осуществляется по умолчанию. Поворот объекта вокруг своей оси выполняется тоже в течение 1 сек.

На следующем шаге наш объект должен полежать в спокойствии, что выполняется при помощи элемента паузы. Это довольно специфический элемент анимации. Он позволяет на определенный промежуток времени как бы сказать: «необходимо подождать». Этот тип анимации имеет только одно свойство `duration` для задания времени ожидания, которому мы устанавливаем значение 500, то есть 0,5 сек.

И напоследок наш объект должен вернуться в свою исходную позицию. Для этого мы используем элемент `NumberAnimation`, в котором задаем изменение свойства `y` от нашей нижней точки 300 до верхней 20 (свойства `from` и `to`). Продолжительность этой анимации будет тоже составлять 0,5 сек (свойство `duration`).

Листинг 58.7. Последовательная анимация

```
import QtQuick 2.8
Rectangle {
    width: 130
    height: 450
    Image {
        id: img
        source: " qrc:/happyos.png"
        smooth: true
        Text {
            anchors.horizontalCenter: img.horizontalCenter
            anchors.top: img.bottom
            text: "Click me!"
        }
        MouseArea {
            anchors.fill: img
            onClicked: anim.running = true
        }
        SequentialAnimation {
            id: anim
            NumberAnimation {
                target: img
                from: 20
                to: 300
```

```
            properties: "y"
            easing.type: Easing.OutBounce
            duration: 1000
        }
        RotationAnimation {
            target: img
            from: 0
            to: 360
            properties: "rotation"
            direction: RotationAnimaion.Clockwise
            duration: 1000
        }
        PauseAnimation {
            duration: 500
        }
        NumberAnimation {
            target: img
            from: 300
            to: 20
            properties: "y"
            easing.type: Easing.OutBack
            duration: 1000
        }
    }
  }
}
```

Состояния и переходы

Состояния похожи на шаги в истории, и вы можете их образно сравнить с отдельными кадрами на кинопленке. Наблюдая за кадрами кинофильма, можно сказать, например, что только минуту назад супермен был в состоянии полета в пункте «А», а теперь он уже находится в состоянии приземления в пункт «В».

Теперь немного проясним назначение *переходов*, используя тот же самый пример. Понятно, что на пленке, демонстрируемой со скоростью 25–30 кадров в секунду, наш герой-супермен движется плавно. Но если бы у нас было всего два кадра из этого фильма: кадр состояния «А» и кадр состояния «В»? Иначе говоря, произошла бы просто смена одного кадра другим, и показ такой анимации оказался бы не очень-то впечатляющим. Для того чтобы исправить ситуацию, нужно внедрить между этими двумя состояниями какой-нибудь красивый переход.

Состояния

Состояния в языке QML представлены при помощи элемента `State`. Каждое отдельно взятое состояние — это конфигурация, так как с его помощью можно свойствам элементов присвоить целые наборы значений. Из состояний можно сформировать даже целые списки, но этим назначение состояний не ограничивается. С их помощью можно также запускать на выполнение функции JavaScript, управлять изменением фиксации и изменять элементы предков.

Что ж, перейдем от теории к практике и реализуем пример (листинг 58.8), который использует два состояния. Смена состояний будет осуществляться нажатием мыши на область элемента. На рис. 58.9 изображено окно нашего примера в его начальном состоянии, а на рис. 58.10 — конечное состояние с изменением размера, цвета и надписи элемента.

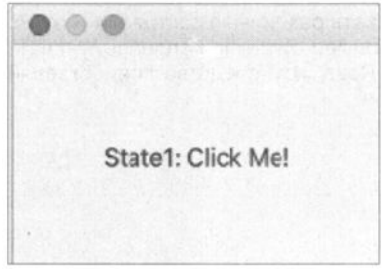


Рис. 58.9. Первое (начальное) состояние

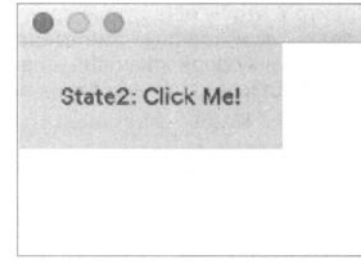


Рис. 58.10. Второе (конечное) состояние

Давайте создадим в Qt Creator новый файл main.qml (листинг 58.8) и зададим в нем прямоугольник (элемент `Rectangle`), в середине которого есть надпись (элемент `Text`). Каждому элементу при помощи свойства `id` необходимо задать идентификатор. Это связано с тем, что состояния могут выполнять изменение свойств только у именованных элементов, поэтому первый элемент назван `rect`, а второй — `txt`.

Теперь перейдем к самим состояниям. Список состояний мы задаем при помощи квадратных скобок — именно так задаются списки элементов в языке QML. Список состояний должен быть присвоен свойству `states`. Элементы списка разделяются запятыми. То, какое состояние должны взять наши визуальные элементы для инициализации, задается свойством `state`. В нашем примере для инициализации мы выбираем второе состояние, используя имя `State2`. Как видите, состояния должны обязательно иметь имя, иначе мы на них не сможем ссылаться и отличать одно состояние от другого.

> ***Состояние инициализации***
>
> Состояние, определенное в списке первым, является состоянием инициализации и может вызываться при помощи пустой строки `""`. То есть если бы мы в нашем примере свойству `state` присвоили пустую строку, то это имело бы тот же эффект, что и присвоение `"State1"`.

Что же теперь мы можем сделать с состояниями? Мы можем задать изменения, которые должны происходить при входе в определенное состояние. Изменение свойств внутри состояний осуществляется при помощи элементов `PropertyChanges`. То, в каком элементе должны быть проведены изменения, задается свойством `target`. Для каждого элемента необходимо создать свой отдельный элемент, внутри которого мы можем изменять любое количество его свойств. Сам этот элемент описывает новые значения для свойств элемента, которые присваиваются сразу же при смене состояния. В нашем примере мы изменяем свойства `color`, `width` и `height` для элемента прямоугольника `Rectangle`, а для элемента текста `Text` изменяем свойство `text`.

Чтобы дать пользователю возможность входить в созданные нами состояния, создаем область мыши. С ее помощью мы будем сменять состояния при каждом нажатии на область элемента `Rectangle`. В свойстве `onClicked` проверяем имя текущего состояния и устанавливаем состояние, отличное от текущего состояния. Для этого мы просто присваиваем свойству `state` имя `"State1"`, если его текущее имя было `"State2"`, и наоборот.

Обратите внимание, что состояния определены в листинге 58.8 отдельно от остальных элементов. Это большое преимущество, так как позволяет отделить логику от графического интерфейса.

Графические средства языка UML

Для того чтобы еще лучше продумать и спланировать различные состояния, можно использовать также графические средства языка *UML* (Unified Modeling Language, унифицированный язык моделирования) — например, Rational Rose. Эти средства предоставляют хорошую поддержку для рисования диаграмм состояний.

Листинг 58.8. Состояния (файл main.qml)

```
import QtQuick 2.8
Rectangle {
    id: rect
    width: 360
    height: 360
    state: "State2"
    Text {
        id: txt
        anchors.centerIn: parent
    }
    states: [
        State {
            name: "State1"
            PropertyChanges {
                target: rect
                color: "lightgreen"
                width: 150
                height: 60
            }
            PropertyChanges {
                target: txt
                text: "State2: Click Me!"
            }
        },
        State {
            name: "State2"
            PropertyChanges {
                target: rect
                color: "yellow"
                width: 200
                height: 120
            }
            PropertyChanges {
                target: txt
                text: "State1: Click Me!"
            }
        }
    ]
```

```
    MouseArea {
        anchors.fill: parent
        onClicked:
         parent.state = (parent.state == "State1") ? "State2" : "State1"
    }
}
```

Переходы

Мы уже убедились, рассматривая состояния, что переключение из одного состояния в другое похоже на простую смену картинок и выглядит не очень привлекательно. Переходы применяются к двум и более состояниям и описывают, как между состояниями должна проходить анимация, то есть определяют, как элементы изменяются со сменой одного состояния другим.

Если вы хотите, чтобы графический интерфейс вашей программы вел себя натурально, как в реальном мире, то необходимо внедрять переходы из одного состояния в другое. Используя переходы, мы как бы говорим языку QML: «если хочешь изменить позицию элемента, делай это, но делай это, пожалуйста, красиво».

Анимационный движок выполнит всю необходимую работу и решит, как наилучшим образом распределить кадры по времени. Он определит и то, какие кадры должны быть созданы и показаны, а какие можно не создавать и не показывать, а это очень важно, так как влияет на производительность. Синтаксис для определения переходов практически идентичен синтаксису определения состояний.

Теперь самое время перейти к практике. В следующем примере (листинг 58.9) мы создадим анимацию из двух состояний и соединим их переходами (рис. 58.11).

Состояния мы рассмотрели в предыдущем листинге 58.8, поэтому для листинга 58.9 мы ограничимся объяснением только переходов.

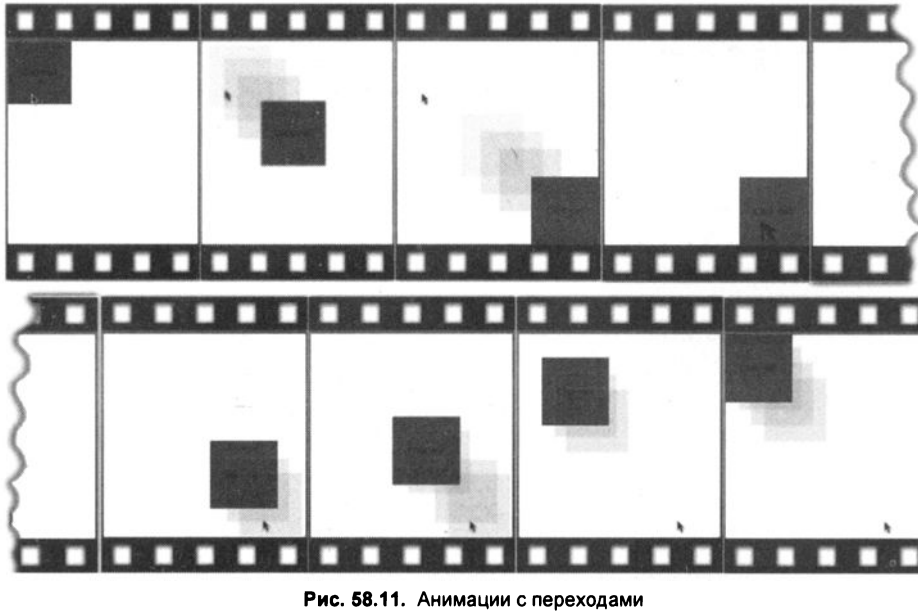


Рис. 58.11. Анимации с переходами

Свойство `transitions` задает список переходов. В списке переходов создаем элементы переходов `Transition` и разделяем их запятой. Каждый элемент перехода отмечен двумя важными моментами:

- первый важный момент — это свойства `from` и `to`, которые задают для перехода начальное и конечное состояния. В нашем случае имеются два перехода. Первый переход применяется при смене состояний от состояния `State1` и до состояния `State2`, во втором переходе все происходит в обратную сторону, то есть от состояния `State2` до `State1`;
- второй важный момент — это элемент `PropertyAnimation`, который осуществляет «магию» изменения свойств по времени и выполняет сложные вычисления. Но в это вам вникать не нужно — от вас требуется лишь задать время для проведения самой анимации в свойстве `duration`. В нашем примере анимация выполняется в течение 1 сек. Можно также свойством `easing.type` задать закон для изменения поведения динамики.

Листинг 58.9. Переходы

```
import QtQuick 2.8
Item {
    width: 300
    height: 300
    Rectangle {
        id: rect
        width: 100
        height:  100
        color: "magenta"
        state: "State1"
        Text {
            anchors.centerIn: parent
            text: "Click me!"
        }
        MouseArea {
            anchors.fill: rect
            onClicked:
             rect.state = (rect.state == "State1") ? "State2" : "State1"
        }
        states: [
            State {
                name: "State1"
                PropertyChanges {target: rect; x: 0; y: 0}
            },
            State {
                name: "State2"
                PropertyChanges {target: rect; x: 200; y: 200}
            }
        ]
        transitions: [
            Transition {
                from: "State1"; to: "State2"
                PropertyAnimation {
                    target: rect;
                    properties: "x,y";
```

```
                easing.type: Easing.InCirc
                duration: 1000
            }
        },
        Transition {
            from: "State2"; to: "State1"
            PropertyAnimation {
                target: rect
                properties: "x,y";
                easing.type: Easing.InBounce
                duration: 1000
            }
        }
    ]
}
}
```

В нашем примере (см. листинг 58.9) мы используем для каждого перехода две разные смягчающие линии: `Easing.InCirc` и `Easing.InBounce`. Но если бы мы использовали одинаковые смягчающие линии, то код обоих элементов `PropertyAnimation` был бы идентичен, и тогда мы могли бы сократить его и использовать *шаблонный переход*. Просто замените свойство `transitions` в листинге 58.9 на код, приведенный в листинге 58.10, запустите пример на выполнение и посмотрите на изменение работы программы.

Листинг 58.10. Шаблонный переход

```
transitions:
    Transition {
        from: "*"; to: "*"
        PropertyAnimation {
            target: rect;
            properties: "x,y";
            easing.type: Easing.InCirc
            duration: 1000
        }
    }
```

Шаблонные переходы создаются при помощи символа `"*"` (см. листинг. 58.10). Этот символ представляет любое состояние, и анимация будет проводиться при смене любого из состояний. Заметьте, что у нас нет необходимости задавать два отдельных перехода для наших состояний, как мы это делали в листинге 58.9. Но в обоих случаях будет использоваться одна и та же смягчающая линия `Easing.InCirc`.

Модуль частиц

Модуль частиц `QtQuick.Particles` позволяет создавать потрясающие визуальные эффекты. Эти эффекты производятся отображением большого количества частиц и могут очень пригодиться в реализации видеоигр для симулирования взрывов, летающих метеоритов, разлетающихся космических кораблей и многого другого.

Использование модуля частиц только лишь играми, конечно же, не ограничивается, и эти эффекты можно с успехом применять везде, где требуется задействовать много двигающихся частиц для их отображения на дисплее. С их помощью можно также придать пользовательскому интерфейсу приложения особый шарм, при условии, если уместно и умело воспользоваться ими.

Четыре следующих компонента составляют стержень модуля частиц:

- элемент `ParticleSystem` является центральным — он запускает системный таймер для управления временной линией;
- элемент `Emitter` (эмиттер) — излучает частицы;
- элемент `ParticlePainter` — служит для отображения частиц. Сам элемент может быть элементом растрового изображения `ImageParticle`, а также и элементом шейдера, представленным элементом `CustomParticle`. В *главе 56* можно найти больше информации о том, как использовать шейдеры в QML;
- элемент `Affector` (эффектор) — изменяет поведение частиц после их создания эмиттером.

Для демонстрации того, как указанные элементы взаимодействуют друг с другом, реализуем в листинге 58.11 пример симуляции падения снежинок, показанных в окне программы (рис. 58.12).

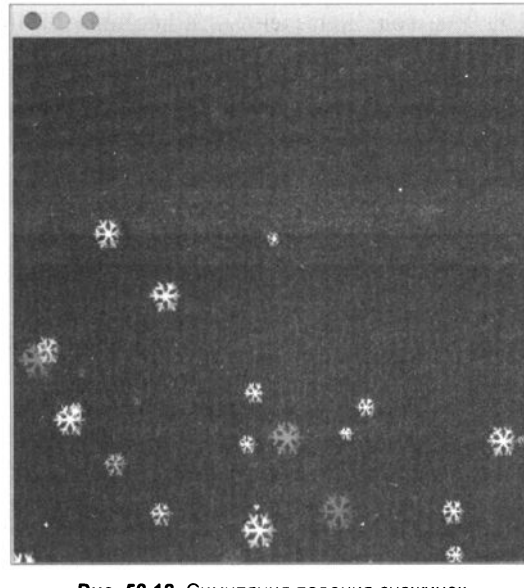


Рис. 58.12. Симуляция падения снежинок

Первый элемент модуля частиц, который мы создаем в листинге 58.11, это сама система `ParticleSystem`. Она необходима, чтобы инициировать работу таймера и запустить процесс управления системой частиц.

Чтобы падение снежинок происходило во всей области окна, мы при помощи фиксатора и присваиваем заполнение всей области окна. Если бы мы хотели отобразить их только в левой половине окна, можно было бы присвоить элементу `ParticleSystem` половинную ширину и высоту окна, разделив ее на 2.

Для растровых изображений снежинок, которые являются частицами, мы используем элемент `ImageParticle`.

Создаем эмиттер — он нам необходим не только для излучения частиц. В его обязанности в нашей программе так же входит:

- управление сроком жизни частиц (свойство `lifeSpan`). В нашем случае срок жизни частиц равен 10 секундам;
- управление размерами частиц (свойства `size`, `endSize` и `sizeVariation`). В нашем случае мы используем свойство `sizeVariation`, потому что нам необходим метод создания снежинок со случайным размером, — поскольку в реальном мире снежинки находятся от нас на разном расстоянии, их размеры должны различаться между собой. Для этого в свойстве `sizeVariation` мы устанавливаем значение 16 — это максимальный размер диапазона для генератора случайного размера;
- управление частотой излучения частиц в секунду (свойство `emitRate`). По умолчанию излучается 10 частиц в секунду, но нам нужно больше, поэтому мы присваиваем этому свойству значение 20, что соответствует 20 частицам в секунду;
- управление скоростью (свойство `velocity`). На самом деле это свойство может управлять не только скоростью — в нашем случае с помощью элемента `AngleDirection` оно так же управляет и направлением движения снежинок. Как это происходит, мы уточним чуть позже.

Поскольку снежинки должны падать сверху вниз, нам нужно указать угол падения, — для этого мы присваиваем свойству `velocity` элемент `AngleDirection`. Угол излучения частиц задается свойством `angle` и варьируется в диапазоне от 0 до 360 градусов, при этом 0 градусов означает, что движение будет производиться вправо. Но нам нужно чтобы снежинки перемещались сверху вниз, поэтому указываем угол 90 градусов.

Скорость задается свойством `magnitude`, и по умолчанию она равна 0, поэтому если это свойство не указать, то наши снежинки будут возникать и оставаться на месте. Это явно не то, чего мы добиваемся, поэтому зададим свойству `magnitude` значение 100 — теперь наши снежинки будут падать сверху вниз со скоростью 100 пикселов в секунду.

Листинг 58.11. Симуляция снегопада

```
import QtQuick 2.8
import QtQuick.Particles 2.0

Rectangle {
    width: 360
    height: 360
    color: "MidnightBlue"

    ParticleSystem {
        anchors.fill: parent

        ImageParticle {
            source: "qrc:/snowflake.png"
        }

        Emitter {
            width: parent.width
```

```
            height: parent.height
            anchors.bottom: parent.buttom
            lifeSpan: 10000
            sizeVariation: 16
            emitRate: 20
            velocity:
                AngleDirection {
                    angle: 90
                    angleVariation: 10
                    magnitude: 100
                }
        }
    }
}
```

Если добавить к симуляции немного ветра — чтобы снежинки, пройдя какую-то часть пути, резко ускорились в каком-либо направлении, например вправо, — можно сделать нашу анимацию еще более реалистичной и впечатляющей. А это значит, что пришло, наконец, время задействовать четвертый из основных элементов модуля частиц — эффектор. Мы уже упоминали о его невероятных способностях изменять поведение частиц после их создания эмиттером. Для того чтобы продемонстрировать это в действии, возьмем основанный на нем элемент `Gravity` (листинг 58.12). Его назначение, как и следует из его имени, это применение силы «гравитации», с помощью которой он способен притягивать объекты в ту или иную сторону.

Начало действия силы гравитации мы установим со второй половины окна приложения (свойства `y` и `height`) — это значит, что когда наши снежинки попадут в эту область, то они подвергнутся эффекту гравитации и резко отклонятся от своего курса вправо (свойство `angle`) со скоростью 250 пикселов в секунду (свойство `acceleration`). И теперь все получается, как в старой доброй песне ВИА «Пламя»: «Снег кружится, летает, летает...».

Листинг 58.12. Добавляем ветер: эффект гравитации

```
Gravity {
    y: parent.height / 2
    width: parent.width
    height: parent.height
    angle: 0
    acceleration: 250
}
```

Для того чтобы задействовать эффект гравитации, вставьте код листинга 58.12 сразу после элемента эмиттера (`Emitter`) листинга 58.11. Для дальнейших экспериментов можно воспользоваться эффекторами, указанными в табл. 58.1.

Таблица 58.1. *Некоторые эффекторы модуля QtQuick.Partices 2.0*

Тип	Описание
`Age`	Изменение срока жизни частиц
`Friction`	Сила трения

Таблица 58.1 (окончание)

Тип	Описание
`Turbulence`	Сила турбулентных потоков
`Wander`	Изменение траектории частиц случайным образом

Резюме

Подведем итог. Как вы уже убедились, язык QML — на самом деле очень мощная технология, и с ее помощью можно делать изумительные анимации. Код программ получается сравнительно небольшой — просто представьте себе то количество кода, которое пришлось бы написать, чтобы сделать подобные анимации на языке C++.

Анимации в QML можно применять к любому элементу пользовательского интерфейса. В основе анимаций лежит элемент `PropertyAnimation`. Он и все базирующиеся на нем элементы управляют изменением свойств элементов в заданном промежутке времени. Изменением динамики проведения анимации управляют смягчающие линии. Программа Qt Creator предоставляет возможность проведения интерактивной настройки параметров смягчающих линий.

Отдельный тип анимаций — это анимации поведения. Они выполняют анимацию при изменении значений определенных свойств.

Несколько анимаций можно объединять в одну анимацию при помощи элементов групп. Группа может запускать входящие в нее анимации параллельно и последовательно. Для создания более сложных сценариев группы также можно вкладывать друг в друга.

Состояния — это хранилище для набора значений свойств определенных элементов. Присвоение элементу определенного состояния приводит к мгновенному присвоению новых значений свойствам, входящим в это состояние элементов.

Переходы отвечают за проведение анимации между сменой состояний.

Мы познакомились также с модулем частиц, с помощью которого можно создавать анимации из множества объектов — например, симулировать падение снежинок.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/58-510/** или с помощью следующего QR-кода (рис. 58.13):



Рис. 58.13. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 59

Модель/Представление

> Секрет успеха в жизни — быть готовым для возможностей, когда они приходят к тебе.
>
> *Бенджамин Дизраэли*

Язык QML, так же как и концепт «интервью» (Interview), реализованный в библиотеке Qt (см. *главу 12*), предоставляет механизм отделения данных от представления. За показ данных отвечают элементы представлений и делегаты, а за поставку данных — элементы моделей, которые мы сейчас и рассмотрим.

Модели

Модель — это элемент, который предоставляет интерфейс для обращения к данным, в отдельных случаях этот элемент может также содержать и сами данные, но это совсем необязательно.

Элементы моделей не располагают информацией о том, как отображать их данные. К типичным моделям, представленным языком QML, относятся модели `ListModel` и `XmlListModel`. Но вы можете также применять модель, которая реализована на C++ (см. *главу 60*). Кроме того, в QML в качестве модели может выступать число целого и вещественного типа, массивы и данные в формате JSON.

Рассмотрим возможности реализации и использования моделей QML.

Модель списка

Модель списка представлена элементом `ListModel` и содержит последовательности элементов в следующем виде:

```
ListModel {
    ListElement{...}
    ListElement{...}
...
}
```

Каждый ее подэлемент `ListElement` содержит одно или более свойств для данных. Элемент `ListElement` не содержит ни одного предопределенного свойства, и все они задаются пользователем. Создадим модель списка для коллекции компакт-дисков (листинг 59.1) и зададим свойства `artist` (исполнитель), `album` (альбом), `year` (год) и `cover` (обложка).

Листинг 59.1. Модель данных (файл CDs.qml)

```
import QtQuick 2.8
ListModel {
    ListElement {
        artist: "Amaranthe"
        album: "Amaranthe"
        year: 2011
        cover: "qrc:/covers/Amaranthe.jpg"
    }
    ListElement {
        artist: "Dark Princess"
        album: "Without You"
        year: 2005
        cover: "qrc:/covers/WithoutYou.jpg"
    }
    ListElement {
        artist: "Within Temptation"
        album: "The Unforgiving"
        year: 2011
        cover: "qrc:/covers/TheUnforgiving.jpg"
    }
...
}
```

В листинге 59.1 мы каждое заданное нами свойство снабжаем данными. Первые три свойства — это данные, которые содержат информацию описательного характера о компакт-дисках (CD), последнее свойство: `cover` — это путь к файлу растрового изображения обложки CD.

Несмотря на то, что в листинге 59.1 мы прописываем все данные вручную, элемент `ListModel` — это динамический список элементов. Элементы этого списка могут быть добавлены, вставлены, удалены и перемещены при помощи интегрированных в элемент `ListModel` методов `append()`, `insert()`, `remove()` и `move()`. Например, код для удаления текущего элемента может выглядеть так:

```
CDs.remove(view.currentIndex)
```

Наша модель реализована в отдельном файле CDs.qml. Если же вы разместите ее в одном и том же файле вместе с представлением, то для того чтобы иметь возможность к ней обратиться, необходимо при помощи свойства `id` снабдить ее идентификатором. В целях экономии места файл CDs.qml в листинге 59.1 полностью не показан.

XML-модель

Имеется очень много источников, которые предоставляют данные в XML-формате. Классическим примером является интерфейс с веб-серверами. Это и послужило причиной для создания отдельного элемента модели. Элемент `XmlListModel` — это тоже модель списка и используется для XML-данных. Модель `XmlListModel` задействует для заполнения данными опросы `XPath` (см. *главу 40*) и присваивает данные свойствам.

Для того чтобы сказанное было понятно, проиллюстрируем создание XML-модели на примере, и для начала создадим локальный файл с XML-данными (листинг 59.2). Хотя таким мог быть и ответ сервера на запрос о нужной вашему приложению информации.

Листинг 59.2. XML-данные (файл CDs.xml)

```
<?xml version = "1.0"?>
<CDs>
    <CD>
        <artist>Amaranthe</artist>
        <album>Amaranthe</album>
        <year>2011</year>
        <cover>qrc:/covers/Amaranthe.jpg</cover>
    </CD>
    <CD>
        <artist>Dark Princess</artist>
        <album>Without You</album>
        <year>2005</year>
        <cover>qrc:/covers/WithoutYou.jpg</cover>
    </CD>
    <CD>
        <artist>Within Temptation</artist>
        <album>The Unforgiving</album>
        <year>2011</year>
        <cover>qrc:/covers/TheUnforgiving.jpg</cover>
    </CD>
...
</CDs>
```

В листинге 59.2 первым идет стандартный заголовок XML-документа. Далее расположены теги с данными. Имена и назначение тегов и данных аналогичны листингу 59.1. В целях экономии места файл CDs.xml в листинге полностью не показан.

Теперь реализуем модель, которая работает с данными из файла CDs.xml. Эта модель базируется на элементе `XmlListModel` (листинг 59.3).

Листинг 59.3. XML-модель (файл CDs.qml)

```
import QtQuick 2.8
import QtQuick.XmlListModel 2.0
XmlListModel {
    source: "qrc:///CDs.xml"
    query:  "/CDs/CD"
    XmlRole {name: "artist"; query: "artist/string()"}
    XmlRole {name: "album"; query: "album/string()"}
    XmlRole {name: "year"; query: "year/string()"}
    XmlRole {name: "cover"; query: "cover/string()"}
}
```

В листинге 59.3 мы реализуем элемент нашей модели. Элемент `XmlListModel` имеет свойство `source`, а это значит, что XML-данные могут быть получены не только с локального но-

сителя, но и из Интернета. Любой элемент с этим свойством имеет такую способность. Мы присваиваем этому свойству в качестве источника XML-данных файл CDs.xml, который мы создали в листинге 59.2.

Чтобы иметь возможность получать данные, нужен механизм XPath. Он позволяет легко запросить нужную нам информацию. Этот механизм вы можете представить как механизм запросов к базе данных, которая представлена XML-файлом. Нас интересуют определенные имена свойств, поэтому, чтобы получить их данные, мы создаем элементы `XmlRole`. В этих элементах запрашивается любая часть данных. В свойстве `name` мы задаем имя, которое будет представлять данные, а в свойстве `query` указываем тег и его тип. Свойство `query` идентифицирует части данных в модели. В нашем случае все типы строковые. Все эти данные берутся относительно пути тегов `"/Cds/CD"`, заданного в свойстве `query` родительского элемента `XmlListModel`.

JSON-модель

Очень удобным типом модели в QML является JSON. В *главе 50* мы уже познакомились с этим форматом.

Как и в случае с XML-моделью, в качестве данных модели могут выступать данные, полученные приложением от веб-сервисов. Благодаря тому, что в QML интегрирован JavaScript, данные в формате JSON могут использоваться напрямую, то есть вы можете считать JSON-данные в переменную и использовать ее в качестве модели данных. Наша модель для CD в JSON могла бы иметь следующий вид (листинг 59.4):

Листинг 59.4. JSON-данные (файл CDs.js)

```
var jsonModel = [
    {
        artist: "Amaranthe",
        album: "Amaranthe",
        year: 2011,
        cover: "qrc:/covers/Amaranthe.jpg",
    },
    {
        artist: "Dark Princess",
        album: "Without You",
        year: 2005,
        cover: "qrc:/covers/WithoutYou.jpg",
    },
    {
        artist: "Within Temptation",
        album: "The Unforgiving",
        year: 2011,
        cover: "qrc:/covers/TheUnforgiving.jpg",
    },
...
]
```

Как использовать файл с JSON-данными из листинга 59.4, мы покажем на примере следующего листинга 59.5.

Представление данных моделей

Для отображения данных моделей QML предоставляет три основных элемента:

- `ListView` — показывает классический список элементов, расположенных в горизонтальном или вертикальном порядке;
- `GridView` — отображает элементы в виде таблицы подобно тому, как это делается в обозревателе в режиме отображения значков;
- `PathView` — отображает элементы в виде замкнутой ленты.

Следует заметить, что все эти элементы базируются на элементе `Flickable`.

Элемент *ListView*

За отображение данных в виде столбца или строки отвечает элемент `ListView` (рис. 59.1). Отображение данных модели мы рассмотрим на конкретном примере (листинг 59.5) — здесь осуществляется отображение данных JSON-модели из листинга 59.4.

В листинге 59.5 мы импортируем JS-файл с данными JSON-модели и указываем идентификатор для пространства имен CDs, чтобы получить доступ к переменной, которой эта модель присвоена. Далее задаем элемент верхнего уровня `Rectangle`, присваиваем ему серый цвет (свойство `color`) и размеры 200×360.



Рис. 59.1. Отображение данных в элементе представления списка `ListView`

За отображение каждого элемента списка в отдельности всегда отвечает элемент делегата. За основу для делегата берем элемент `Component` и присваиваем ему идентификатор `delegate`, который имеет свойство `id`. Это нужно для того, чтобы мы могли сослаться на него из элемента представления. Внутри этого элемента мы в элементе `Item` определяем то, как должен отображаться отдельный элемент данных. При помощи элемента `Row` отображаем данные в виде строки, причем слева — растровое изображение обложки (элемент `Image`), а справа — вся текстовая информация, расположенная в виде столбца при помощи элемента `Column`. Элемент `Image` загружает файл, местоположение и имя которого содержится в свойстве модели `cover`. Элементы `Text` отображают разными цветами и размерами свойства моделей `artist`, `album` и `year`. Заметьте, доступ к свойствам модели мы получаем посредством текущего элемента `modelData`.

Теперь мы переходим к описанию собственно самого элемента представления `ListView`. Представления позиционируются внутри других элементов точно так же, как и обычные

элементы (свойство anchors). В нашем примере представление расположено внутри элемента серого прямоугольника. Для того чтобы разрешить в представлении навигацию, выполняемую при помощи клавиатуры, свойством focus осуществляем установку фокуса.

По умолчанию элемент ListView применяется без декорации. Добавить декорации можно с помощью свойств header и footer, а также свойства highlight — для показа текущего элемента. В верхней декорации с помощью свойства footer мы создаем градиентный заголовок (элемент Gradient) и в его центре позиционируем надпись "CDs" (элемент Text). В нижней декорации (свойство footer) мы тоже задаем элемент прямоугольника с градиентом, но без надписи. Для декорирования выделения элементов (свойство highlight) представления списка мы ограничиваемся только установкой голубого цвета (свойство color).

И в заключение два ключевых момента: об установке модели и делегата. Они осуществляются при помощи соответствующих свойств: model и delegate. Как мы уже знаем, модель содержится в JS-файле и присвоена переменной jsonModel. Для того чтобы получить к ней доступ, нам нужно использовать идентификатор пространства имен, поэтому мы присваиваем CDs.jsonModel свойству model. Созданный нами делегат представлен элементом Component и имеет идентификатор delegate. Чтобы его задействовать, мы присваиваем идентификатор делегата свойству delegate представления ListView.

Листинг 59.5. Использование элемента ListView (файл main.qml)

```
import QtQuick 2.8
import "qrc:/CDs.js" as CDs

Rectangle {
    id: mainrect
    color:  "gray"
    width: 200
    height: 360

    Component {
        id: delegate
        Item {
            width: mainrect.width
            height: 70
            Row {
               anchors.verticalCenter: parent.verticalCenter
               Image {
                   width: 64
                   height: 64
                   source: modelData.cover
                   smooth: true
               }
               Column {
                   Text {color: "white"
                         text: modelData.artist
                         font.pointSize: 12
                        }
```

```
                Text {color: "lightblue"
                      text: modelData.album
                      font.pointSize: 10
                     }
                Text {color: "yellow"
                      text: modelData.year
                      font.pointSize: 8
                     }
            }
        }
    }
}

ListView {
    focus:  true
    header: Rectangle {
        width: parent.width
        height: 30
        gradient: Gradient {
            GradientStop {position: 0; color: "gray"}
            GradientStop {position: 0.7; color: "black"}
        }
        Text{
            anchors.centerIn: parent;
            color: "gray";
            text: "CDs"
            font.bold: true;
            font.pointSize: 20
        }
    }
    footer: Rectangle {
        width: parent.width
        height: 30
        gradient: Gradient {
            GradientStop {position: 0; color: "gray"}
            GradientStop {position: 0.7; color: "black"}
        }
    }
    highlight: Rectangle {
        width: parent.width
        color: "darkblue"
    }

    anchors.fill: parent
    model: CDs.jsonModel
    delegate: delegate
}
}
```

Элемент *GridView*

Элемент `GridView` автоматически заполняет всю область отображаемыми элементами в табличном порядке (рис. 59.2), поэтому нет необходимости устанавливать количество столбцов и строк. Здесь и далее будет осуществляться отображение данных модели из листинга 59.1 либо идентичной модели из листинга 59.3.



Рис. 59.2. Отображение данных в элементе табличного представления `GridView`

Использование элемента табличного размещения `GridView` (листинг 59.6) практически идентично использованию элемента `ListView`, которое мы рассмотрели в листинге 59.5.

Листинг 59.6. Использование элемента `GridView` (файл main.qml)

```
import QtQuick 2.8
Rectangle {
    id: mainrect
    color:  "gray"
    width: 380
    height: 420
    Component {
        id: delegate
        Item {
            width: 120
            height: 120
            Column {
               anchors.centerIn: parent
```

```
            Image {
                anchors.horizontalCenter: parent.horizontalCenter
                width: 64
                height: 64
                source: cover
                smooth: true
            }
            Text {color: "white"; text: artist; font.pointSize: 12}
            Text {color: "lightblue"; text: album; font.pointSize: 10}
            Text {color: "yellow"; text: year; font.pointSize: 8}
        }
    }
}
GridView {
    cellHeight: 120
    cellWidth: 120
    focus:  true
    header: Rectangle {
        width: parent.width
        height: 30
        gradient: Gradient {
            GradientStop {position: 0; color: "gray"}
            GradientStop {position: 0.7; color: "black"}
        }
        Text{
            anchors.centerIn: parent;
            color: "gray";
            text: "CDs";
            font.bold: true;
            font.pointSize: 20
        }
    }
    footer: Rectangle {
        width: parent.width
        height: 30
        gradient: Gradient {
            GradientStop {position: 0; color: "gray"}
            GradientStop {position: 0.7; color: "black"}
        }
    }
    highlight: Rectangle {
        width: parent.width
        color: "darkblue"
    }
    anchors.fill: parent
    model: CDs{}
    delegate: delegate
}
}
```

Отличие листинга 59.6 от листинга 59.5 заключается в том, что здесь мы используем элемент `GridView` вместо `ListView`, и что растровое изображение и текстовые надписи располагаются в вертикальном порядке только с помощью элемента `Column`.

Элемент *PathView*

Элемент `PathView` показывает элементы в виде замкнутой линии. Другими словами, пользователь может до бесконечности прокручивать элементы в определенную сторону, и они будут просто повторяться. Продемонстрируем эту особенность на примере с использованием все той же модели списка коллекции CD (листинг 59.7). На рис. 59.3 показано окно программы, в котором мы можем одновременно наблюдать четыре элемента и прокручивать весь список элементов с помощью мыши в правую или левую сторону.



Рис. 59.3. Отображение данных в элементе представления `PathView` в виде замкнутой ленты

Делегат листинга 59.7 идентичен делегатам листингов 59.5 и 59.6. Ключевой момент заключается в создании элемента `Path`. Этот элемент задает форму замкнутой линии. В нашем случае мы определяем горизонтальную линию от 0 до 500 с одинаковым удалением сверху 80. Если бы, например, в элементе `Path` мы присвоили бы свойству `startY` значение 0, то все элементы пошли бы по наклонной линии. Созданный элемент `Path` устанавливается в представлении `PathView` при помощи свойства `path`. В завершение мы устанавливаем свойством `pathItemCount` количество элементов, которые должны быть одновременно видимы.

Листинг 59.7. Использование элемента `PathView` (файл main.qml)

```
import QtQuick 2.8
Rectangle {
    color: "gray"
    width: 450
    height: 170
    Component {
        id: delegate
        Item {
            width: item.width
            height: item.height
            Column {
                id: item
                Image {
                    width: 90
```

```
                height: 90
                source: cover
                smooth: true
            }
            Text {color: "white"; text: artist; font.pointSize: 12}
            Text {color: "lightblue"; text: album; font.pointSize: 10}
            Text {color: "yellow"; text: year; font.pointSize: 8}
        }
    }
}
Path {
    id: itemsPath
    startX: 0
    startY: 80
    PathLine {x: 500; y: 80}
}
PathView {
    id: itemsView
    anchors.fill: parent
    model: CDs {}
    delegate: delegate
    path: itemsPath
    pathItemCount: 4
}
}
```

Теперь воспользуемся этим замечательным свойством замкнутости и реализуем другой элемент `Path`, который будет отображать элементы в виде 3D-карусели, показанной на рис. 59.4. Для этого в листинге 59.7 выполним замену элемента `Path` на такой же элемент `Path` из листинга 59.8.

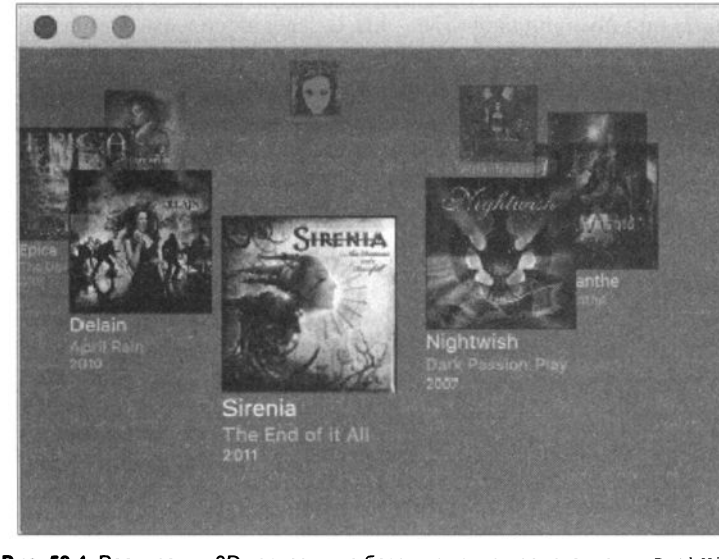


Рис. 59.4. Реализация 3D-карусели на базе элемента представления `PathView`

В листинге 59.8 в элементе `Path` добавился новый элемент `PathQuad`, который задает дугу. Для создания окружности нам потребуется два таких элемента: первый — для правой половины окружности, а второй — для левой. В зависимости от расположения элементов, отображаемых на дуге, мы при помощи элементов `PathAttribute` выполняем изменение их прозрачности и размера. Эти действия и создают иллюзию 3D.

Листинг 59.8. 3D-карусель (файл main.qml)

```
Path {
    id: itemsPath
    startX: 150
    startY: 150
    PathAttribute {name: "iconScale"; value: 1.0}
    PathAttribute {name: "iconOpacity"; value: 1.0}
    PathQuad {x: 150; y: 25; controlX: 460; controlY: 75}
    PathAttribute {name: "iconScale"; value: 0.3}
    PathAttribute {name: "iconOpacity"; value: 0.3}
    PathQuad {x: 150; y: 150; controlX: -80; controlY: 75}
}
```

Визуальная модель данных

Глава о моделях не была бы полной, если бы мы оставили без внимания особый вид модели, которая сама отвечает за представление своих данных и не нуждается в делегате. Этот тип модели может очень пригодиться, например, в тех случаях, когда каждый из элементов нужно отображать в индивидуальной манере. На рис. 59.5 показано использование этой модели для уже знакомого нам собрания компакт-дисков с умышленно внедренным элементом, который отображается кардинально иначе, чем все остальные элементы. Это желтый прямоугольник с надписью **Blank!**.

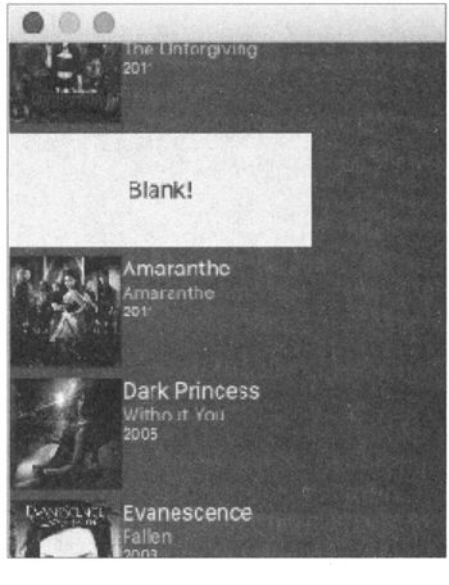


Рис. 59.5. Использование визуальной модели данных `VisialItemModel`

В листинге 59.9 представлен фрагмент кода визуальной модели. Каждый элемент этой модели содержит полную реализацию своего представления, включая размещения составных элементов, а также размеры их шрифтов и цвета. Для всех элементов компакт-дисков нашего примера мы используем одинаковые цвета и размеры шрифтов. На самом деле это совсем не обязательно, потому что визуальная модель позволяет каждый из элементов представить полностью по-своему. И для того чтобы это продемонстрировать, мы внедряем в модель единственный элемент, который будет представлен элементом `Rectangle` — он в модели второй по счету.

Листинг 59.9. Фрагмент визуальной модели данных (файл CDs.qml)

```
import QtQuick 2.8

VisualItemModel {
    Row {
       Image {
           width: 64
           height: 64
           source: "qrc:/covers/Fallen.jpg"
           smooth: true
       }
       Column {
           Text {color: "white"; text: "Evanescence"; font.pointSize: 12}
           Text {color: "lightblue"; text: "Fallen"; font.pointSize: 10}
           Text {color: "yellow"; text: "2003"; font.pointSize: 8}
       }
    }
    Rectangle {
        width: parent.width
        height: 64
        color: "Yellow"
        Text {
            anchors.centerIn: parent
            color: "Red"
            text: "Blank!"
        }
    }
    Row {
       Image {
           width: 64
           height: 64
           source:  "qrc:/covers/Rubicon.jpg"
           smooth: true
       }
       Column {
           Text {color: "white"; text: "Tristania"; font.pointSize: 12}
           Text {color: "lightblue"; text: "Rubicon"; font.pointSize: 10}
           Text {color: "yellow"; text: "2010"; font.pointSize: 8}
       }
    }
...
}
```

Листинг 59.10 — это пример того, как мы используем модель. Для возможности прокрутки элементов мы задействуем элемент `Flickable`, который заключаем внутри элемента `Rectangle`. Элемент `Rectangle` нам нужен для того, чтобы создать темный цвет фона `"DarkSlateGray"`. Высоту области просмотра мы делаем зависимой от высоты элемента `Column` (см. `contentHeight`) — именно этот элемент и будет содержать все элементы, которые будут созданы репитером (`Repiter`) из нашей визуальной модели.

Листинг 59.10. Отображение визуальной модели (файл main.qml)

```
import QtQuick 2.8

Rectangle {
    width: 250;
    height: 250;
    color: "DarkSlateGray"

    Flickable {
        id: view
        width: 250
        height: 500
        contentWidth: 250
        contentHeight: column.height
        anchors.fill: parent

        Column {
            id: column
            anchors.fill: view
            spacing: 5
            Repeater {
                model: CDs{}
            }
        }
    }
}
```

Резюме

Для использования в языке QML технологии *Модель/Представление* применяются три основных типа моделей: `ListModel`, `XmlModel` и JSON. Все три модели представляют данные в виде списка. Последние две часто нужны для получения данных из Интернета.

Для отображения данных существуют три основных класса: `ListView`, `GridView` и `PathView`. Первый отображает данные в виде строки или столбца, второй размещает элементы в виде таблицы, третий же объединяет элементы в замкнутую линию. Элементы представлений отвечают за расположение и управление элементами, а за отображение каждого элемента в отдельности отвечает элемент делегата.

Мы познакомились с визуальной моделью, которая сама отвечает за представления своих элементов и не нуждается в делегате.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/59-510/** или с помощью следующего QR-кода (рис. 59.6):



Рис. 59.6. QR-код для доступа на страницу комментариев к этой главе

ГЛАВА 60

Qt Quick и C++

Для того чтобы расширить горизонты возможного, нужно сделать шаг в невозможное.

Артур Кларк

У меня к вам, дорогой читатель, есть сразу четыре вопроса:

- вам понравились возможности QML?
- вы хотите разрабатывать компоненты на Qt Quick и использовать их в ваших Qt/C++ проектах?
- у вас много разработанных на Qt/C++ компонентов?
- вы хотите использовать их в QML дальше?

Если хоть один из этих важных вопросов занимает ваше сознание, то эта глава для вас.

Итак, вы еще со мной? Тогда не будем терять время и начнем по порядку.

Использование языка QML в C++

Класс `QQuickWidget` интегрирован в QML и предоставляет хорошую среду для показа и визуализации QML-элементов. Он базируется на классе `QWidget`, то есть это не что иное, как виджет, а следовательно, его и надо использовать как обычный виджет в вашем коде на Qt/C++. Таким образом, вы можете расположить этот виджет в области другого виджета, используя класс размещения или задав позицию. Этот класс расположен в отдельном модуле `quickwidgets`, который необходимо включить в проектный файл.

Проиллюстрируем сказанное на отдельном примере (листинг 60.1) создания области, в которой расположен виджет, исполняющий QML-код и имеющий светло-зеленый цвет (рис. 60.1).



Рис. 60.1. Размещение `QQuickWidget` в виджете

Опция включения модулей нашего проектного файла будет выглядеть следующим образом:

```
QT += quick qml widgets quickwidgets
```

Листинг 60.1. Использование QQuickWidget (файл MyWidget.cpp)

```
MyWidget::MyWidget(QWidget* pwgt/*=0*/) : QWidget(pwgt)
{
    QQuickWidget* pv = new QQuickWidget(QUrl("qrc:/main.qml"));
    QVBoxLayout* pvbx = new QVBoxLayout;
    pvbx->addWidget(pv);
    setLayout(pvbx);
}
```

В конструкторе класса `MyWidget` (листинг 60.1) мы создаем объект класса `QQuickWidget` и загружаем в конструкторе из ресурса файл main.qml (листинг 60.2), в котором находится исходный текст программы на языке QML. И в завершение размещаем его на поверхности виджета класса `MyWidget` при помощи класса размещения `QVBoxLayout`.

Листинг 60.2. Исходный текст программы на языке QML (файл main.qml)

```
import QtQuick 2.8
Rectangle {
    color: "lightgreen"
    width: 100
    height: 100
    Text {
        objectName: "text"
        anchors.centerIn: parent
        text: "Hello QML"

        function setFontSize(newSize)
        {
            font.pixelSize = newSize
            return font.family + " Size=" + newSize
        }
    }
}
```

В листинге 60.2 представлена QML-программа, которая просто отображает центрированный текст (элемент `Text`) и содержит функцию изменения размера шрифта `setFontSize()`, которую мы пока вызывать не будем. Для того чтобы лучше была видна область нашего QML-виджета, мы задали элементу `Rectangle` светло-зеленый цвет.

Взаимодействие из C++ со свойствами QML-элементов и вызов их функций

Вот мы и разместили QML-элемент внутри виджета. Теперь возникает вопрос, каким образом можно взаимодействовать с его свойствами и вызывать функции из C++? Ответим на

этот вопрос еще одним примером и внесем некоторые дополнения в уже существующий программный код.

Вы наверняка заметили в листинге 60.2, что мы в элементе `Text` использовали свойство `objectName` и присвоили ему строковое значение `"text"`. Что это за свойство? Дело в том, что базовый QML-элемент `Item` реализован в C++ классом `QQuickItem`, который унаследован от `QObject` точно так же, как и класс `QWidget`. Свойство `objectName` принадлежит классу `QObject`, и, значит, если получить доступ к объекту класса `QQuickItem`, то с ним можно работать как с обычным объектом `QObject`. Для этого мы и присвоили свойству `objectName` строку, с помощью которой сможем найти текстовый элемент. В следующем примере (листинг 60.3) мы получим доступ к обоим QML-элементам нашей программы и изменим значения некоторых из их свойств. На рис. 60.2 показан результат: с измененным цветом фона, измененной надписью, измененным цветом надписи и измененным размером шрифта.

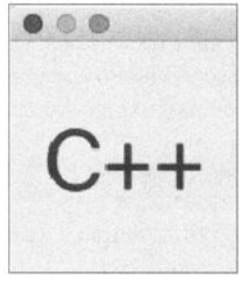


Рис. 60.2. Изменение свойств QML-элементов из C++

Листинг 60.3 представляет собой измененную версию листинга 60.1. Мы добавили в конец код, в котором получаем вызовом метода `QQuickWidget::rootObject()` указатель на узловой объект `QQuickItem`. Этим объектом, если посмотреть листинг 60.2, является элемент `Rectangle`. Так как класс `QQuickItem` унаследован от класса `QObject`, мы вызываем его метод `setProperty()`, в который передаем имя свойства, его новое значение и изменяем тем самым цвет фона этого элемента на желтый. Вызвав метод `QObject::findChild()` из узлового объекта `pqiRoot` и передав ему имя искомого объекта в виде строки, мы получаем указатель на элемент текста. В элементе текста, при помощи двух методов `QObject::setProperty()`, мы присваиваем другую строку свойству `text` и устанавливаем неиспользованному свойству `color` голубой цвет.

Теперь пришла очередь вызвать функцию `setFontSize()` элемента `Text`. Это JavaScript-функция, которая принимает в качестве аргумента целочисленный размер шрифта, устанавливает его и возвращает строку с информацией о предыдущем размере шрифта. Все функции в QML-программе представлены в метаобъектной информации и, благодаря ей, могут быть вызваны из C++. Для этого используется метод `QMetaObject::invokeMethod()`. Как можно видеть в листинге 60.3, в этот метод мы первым передаем указатель на объект, который содержит эту функцию, а затем имя функции. В макросе `Q_RETURN_ARG` указывается переменная, в которую функция должна вернуть значение. Последним следует макрос `Q_ARG` с аргументом, который мы передаем функции, — это размер шрифта в 48 пикселов. Если бы функция принимала два аргумента и более, то нужно было бы указать соответствующее количество макросов `Q_ARG`. Обратите внимание на то, что все аргументы функции передаются посредством типа `QVariant`. После вызова размер шрифта будет изменен, а строку с информацией о предыдущем размере шрифта, сохраненной в переменной `varRet`, мы отобразим при помощи `qDebug()`.

Листинг 60.3. Изменение свойств QML-элементов из C++ (файл MyWidget.cpp)

```
MyWidget::MyWidget(QWidget* pwgt/*=0*/) : QWidget(pwgt)
{
    QQuickWidget* pv = new QQuickWidget(QUrl("qrc:/main.qml"));
    QVBoxLayout* pvbx = new QVBoxLayout;
    pvbx->addWidget(pv);
    setLayout(pvbx);

    QQuickItem* pqiRoot = pv->rootObject();
    if(pqiRoot) {
        pqiRoot->setProperty("color", "yellow");

        QObject* pObjText = pqiRoot->findChild<QObject*>("text");
        if (pObjText) {
            pObjText->setProperty("text", "C++");
            pObjText->setProperty("color", "blue");

            QVariant varRet;
            QMetaObject::invokeMethod(pObjText,
                                      "setFontSize",
                                      Q_RETURN_ARG(QVariant, varRet),
                                      Q_ARG(QVariant, 52)
                                     );
            qDebug() << varRet;
        }
    }
}
```

Соединение QML-сигналов со слотами C++

Для демонстрации соединения QML-сигналов со слотами C++ возьмем простой пример, в котором создадим окно с элементами QML-кнопок **Info** и **Quit** (рис. 60.3) и соединим их сигналы со слотами объекта, реализованного на C++. В прошлых примерах мы использовали QML-элементы внутри виджетов. В этот раз мы полностью исключим их использование и уберем все ненужные модульные зависимости из проектного файла, оставив лишь только два модуля: `QtQuick` и `QtQml`. Опция включения модулей нашего проектного файла будет выглядеть следующим образом:

```
QT += quick qml
```

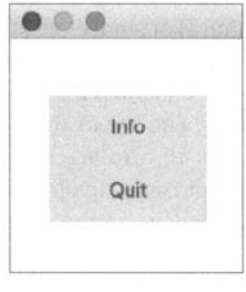


Рис. 60.3. Соединение сигналов QML-кнопок со слотами C++

В листинге 60.4 мы реализуем элемент главного окна QML-приложения и размещаем в элементе `Column` две кнопки (элементы `Button`) с надписями **Info** и **Quit** в виде столбца в центре окна приложения (свойство `anchors.centerIn`). Для получения доступа к кнопкам из C++ присваиваем их свойствам `objectName` уникальные строковые значения. Каждая из кнопок содержит сигнал, который мы впоследствии соединим с объектом C++. Сигнал `infoClicked()` высылается при нажатии на кнопку (свойство `onClicked`) и высылает строку `"Information"`. Сигнал `quitClicked()` высылается без аргументов.

Листинг 60.4. Основная программа на QML (файл main.qml)

```
import QtQuick 2.8
import QtQuick.Controls 2.2
import QtQuick.Window 2.2

Window {
    visible: true
    width: 150
    height: 150
    Column {
        anchors.centerIn: parent
        Button {
            signal infoClicked(string str)
            objectName: "InfoButton"
            text: "Info"
            onClicked: infoClicked("Information")
        }
        Button {
            signal quitClicked()
            objectName: "QuitButton"
            text: "Quit"
            onClicked: quitClicked()
        }
    }
}
```

В листинге 60.5 реализован обычный класс, унаследованный от класса `QObject`, с двумя слотами: `slotQuit()` и `slotInfo()`. Первый слот завершает приложение, а второй выводит на консоль текст, переданный ему в аргументе. Эти слоты мы намереваемся впоследствии соединить с сигналами QML-элементов листинга 60.4.

Листинг 60.5. Класс C++ со слотами (файл CppConnect.h)

```
#pragma once
#include <QtCore>

// ============================================================================
class CppConnection : public QObject {
Q_OBJECT
```

```
public:
    CppConnection(QObject* pobj = 0)
        : QObject(pobj)
    {
    }

public slots:
    void slotQuit()
    {
        qApp->quit();
    }

    void slotInfo(const QString& str)
    {
        qDebug() << str;
    }
};
```

В основной программе приложения (листинг 60.6) мы соединяем QML-сигналы со слотами C++. Вместо класса `QQuickWidget` мы используем два класса: `QQmlApplicationEngine` и `QQmlComponent`, поскольку эти классы дают эквивалентные возможности для показа QML-элементов и получения доступа к ним, как к объектам класса `QObject`. Мы создаем сначала объект `eng` класса `QQmlApplicationEngine`, а затем передаем ссылку на этот объект в конструктор класса `QQmlComonent` вместе с исходным кодом QML-программы и создаем объект `comp`. После чего создаем объект с именем `cc` от класса `CppConnection`, который предназначен для соединения с QML-элементами.

Вызовом метода `QQmlComponent::create()` мы получаем указатель на узловой элемент в виде объекта класса `QObject`, а далее поступаем аналогично тому, как уже делали это в листинге 60.3, а именно: вызываем методы `findChild()` и с помощью заданных строк (свойство `objectName` — см. листинг 60.4) получаем указатели на элементы двух кнопок: `pcmdQuitButton` и `pcmdInfoButton`. Все сигналы, объявленные в QML, автоматически доступны в C++. Поэтому мы используем привычные методы `QObject::connect()` и соединяем сигналы элементов кнопок со слотами объекта `cc`. Теперь при нажатии на кнопку **Info** мы увидим, что на консоли отобразится текст **Information**, а нажатие на кнопку **Quit** приведет к завершению приложения (см. слоты листинга 60.5).

Листинг 60.6. Соединение QML-сигналов со слотами C++ (файл main.cpp)

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include <QQmlComponent>
#include "CppConnection.h"

int main(int argc, char** argv)
{
    QGuiApplication       app(argc,argv);
    QQmlApplicationEngine eng;
    QQmlComponent         comp(&eng, QUrl("qrc:/main.qml"));
    CppConnection         cc;
```

```
    QObject* pobj           = comp.create();
    QObject* pcmdQuitButton = pobj->findChild<QObject*>("QuitButton");
    if (pcmdQuitButton) {
        QObject::connect(pcmdQuitButton, SIGNAL(quitClicked()),
                         &cc, SLOT(slotQuit())
                        );
    }
    QObject* pcmdInfoButton = pobj->findChild<QObject*>("InfoButton");
    if (pcmdInfoButton) {
        QObject::connect(pcmdInfoButton, SIGNAL(infoClicked(QString)),
                         &cc, SLOT(slotInfo(QString))
                        );
    }

    return app.exec();
}
```

Использование компонентов языка C++ в QML

Это самое популярное направление использования QML. Именно благодаря ему можно реализовать красивый пользовательский интерфейс с анимационными эффектами, а реализацию функциональных компонентов возложить на C++. При таком подходе задействуются самые выигрышные стороны обоих инструментов: и QML, и C++.

В языке QML реализована возможность расширения при помощи C++. Благодаря ей можно осуществлять расширение этого языка новыми элементами из C++.

О некоторых расширениях уже позаботились сами разработчики библиотеки Qt. Так, например, если вы хотите использовать уже существующие технологии Qt, такие как `Qt3D`, `QtCharts`, `QtWebEngine`, а также прочие модули Qt QML (см. табл. 53.1), то для этого нужно просто воспользоваться директивой `import`. Например, для `QtWebEngine`:

```
import QtWebEngine 1.5
```

Если же вы имеете свои собственные наработки или модули других разработчиков, базирующиеся на C++ и Qt, и хотите их использовать, то их необходимо сделать доступными для этого языка.

Так или иначе, вам придется иметь дело с классом контекста `QQmlContext`. Чтобы получить доступ к объекту этого класса, нужно из объекта класса `QQuickWidget` вызвать метод `rootContext()`, который возвращает указатель на корневой контекст. Используя этот указатель, вы можете в дерево контекста ввести новые объекты классов, унаследованных от класса `QObject`. Публикация объектов в контексте осуществляется с помощью метода `setContextProperty()`. Этот метод принимает два аргумента. Первый аргумент — это имя, под которым объект будет доступен в QML, второй аргумент — это адрес на сам объект. Если вы проделаете эту операцию, то свойства класса `QObject` станут свойствами QML, а слоты и методы, декларированные с помощью макроса `Q_INVOKABLE`, станут методами, которые могут вызываться из вашего нового QML-элемента.

Класс `QQuickWidget` содержит также и объект класса `QQmlEngine`, который предоставляет среду для QML-компонентов и является сердцевиной для исполнения QML-кода. Доступ к нему можно получить вызовом метода `engine()`.

Экспорт объектов и виджетов из C++ в QML

Чтобы продемонстрировать механизм использования объектов библиотеки Qt в QML, реализуем программу таким образом, чтобы осуществлять из нее публикацию Qt-объектов в QML (листинги 60.7 и 60.8). QML-программа будет, помимо прямоугольной желтой области, содержать элемент представления списка и область мыши. При нажатии мышью на область, реализованную на QML, будет вызываться слот из виджета `MyWidget` и отображаться диалоговое окно с текстом **It's my message** (рис. 60.4).



Рис. 60.4. Публикация Qt-объектов в QML

Чтобы мы могли публиковать наши объекты, нам нужен указатель на объект класса контекста `QQmlContext`. Для этого мы вызываем метод `rootContext()` из виджета `QQuickWidget`. Затем создаем объект списка `QStringList` и в цикле `for` добавляем в него сто текстовых элементов. Этот список вызовом метода `setStringList()` устанавливаем в созданной модели `QStringListModel`. Самые последние четыре вызова методов `setContextProperty()` в конструкторе и есть публикация наших объектов. Мы публикуем объекты разных типов: `QStringListModel`, `QString`, `QColor` и `QWidget` и передаем указатели на них во втором параметре метода. А в первом параметре указываем имя, под которым опубликованный объект будет доступен в QML. Для того чтобы наши объекты можно было без труда заметить в исходном тексте QML, снабжаем их префиксом `"my"`.

В листинге 60.7 также реализован слот `slotDisplayDialog()`, который мы будем вызывать из QML-программы.

Листинг 60.7. Публикация объектов Qt (файл MyWidget.cpp)

```
MyWidget::MyWidget(QWidget* pwgt/*=0*/) : QWidget(pwgt)
{
    QQuickWidget* pv = new QQuickWidget;
    pv->setSource(QUrl("qrc:/main.qml"));

    QVBoxLayout* pvbx = new QVBoxLayout;
    pvbx->addWidget(pv);
    setLayout(pvbx);
```

```
    QQmlContext* pcon = pv->rootContext();
    QStringList lst;
    for (int i = 0; i < 100; ++i) {
        lst << "Item" + QString::number(i);
    }
    QStringListModel* pmodel = new QStringListModel(this);
    pmodel->setStringList(lst);

    pcon->setContextProperty("myModel", pmodel);
    pcon->setContextProperty("myText", "It's my text!");
    pcon->setContextProperty("myColor", QColor(Qt::yellow));
    pcon->setContextProperty("myWidget", this);
}

void MyWidget::slotDisplayDialog()
{
    QMessageBox::information(0, "Message", "It's my message");
}
```

В листинге 60.8 мы используем опубликованные объекты 4 раза. Сначала это установка цвета фона (объект `myColor`) для элемента `Rectangle` (свойство `color`). Затем установка строки текста (объект `myText`) в элементе текста `Text`. Далее это установка модели (объект `myModel`) в элементе представления списка `ListView`. И, наконец, это область мыши `MouseArea`. В ее свойстве `onPressed` первым вызываем объект `myWidget` — «родной» слот виджета `setWindowTitle()`, и устанавливаем в заголовке окна надпись `"Hello from QML"`. Вторым вызываем реализованный нами в классе `MyWidget` слот `slotDisplayDialog()`, который и запускает диалоговое окно.

Листинг 60.8. Использование опубликованных объектов (файл MyWidget.cpp)

```
import QtQuick 2.8
Rectangle {
    color: myColor
    width: 200
    height: 200
    Text {
        anchors.centerIn: parent
        text: myText
    }
    ListView {
        anchors.fill:parent
        model: myModel
        delegate: Text {text: model.display}
    }
    MouseArea {
        anchors.fill: parent
        onPressed: {
            myWidget.setWindowTitle("Hello from QML");
            myWidget.slotDisplayDialog();
        }
    }
}
```

Использование зарегистрированных объектов C++, их свойств и методов в QML

В качестве следующего примера продемонстрируем возможность использования свойств Q_PROPERTY и метода, определенного как Q_INVOKABLE. Программа (листинги 60.9–60.17) вычисляет значение факториала, исходя из значений, введенных в элементы счетчика, расположенного в левой части окна. Результаты вычисления отображаются в правой части окна (рис. 60.5).

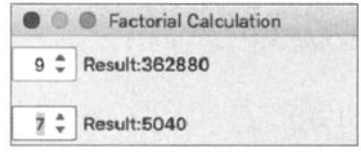


Рис. 60.5. Использование Q_PROPERTY и Q_INVOKABLE

В основной программе, приведенной в листинге 60.9, мы регистрируем при помощи функции qmlRegisterType<T>() наш класс Calculation. В первом параметре этой функции мы передаем строку, которая задает идентификатор модуля с его именем для включения в QML-программу. Вторым параметром передаем номер версии, третьим — номер уровня версии. В последнем, четвертом, параметре мы передаем имя элемента.

Листинг 60.9. Регистрация класса Calculation (файл main.cpp)

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include "Calculation.h"

int main(int argc, char** argv)
{
    QGuiApplication app(argc, argv);

    qmlRegisterType<Calculation>("com.myinc.Calculation", 1, 0, "Calculation");

    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

    return app.exec();
}
```

В классе Calculation (листинг 60.10) мы определяем два свойства: input и output. Обратите внимание, что класс обязательно должен быть унаследован от класса QObject.

Свойство input — это параметр ввода, поэтому в определении Q_PROPERTY мы разрешаем для него запись (WRITE), чтение (READ) и уведомление о его изменении (NOTIFY). Второе свойство — result — это выходное значение результата, поэтому оно не должно подвергаться изменениям извне, и для этого мы разрешаем только его чтение (READ) и уведомление о его изменении.

Мы также определяем в классе метод factorial() — для вычисления значения факториала и декларируем его как Q_INVOKABLE.

После чего определяем методы: `inputValue()`, `resultValue()`, `setInputValue()` — для чтения и установки значений свойств и сигналы: `inputValueChanged()`, `resultValueChanged()` — для уведомления об изменении значений свойств.

Листинг 60.10. Определение класса `Calculation` (файл Calculation.h)

```
#pragma once
#include <QObject>

// ============================================================================
class Calculation : public QObject {
Q_OBJECT
private:
    Q_PROPERTY(qulonglong input WRITE  setInputValue
                                READ   inputValue
                                NOTIFY inputValueChanged
              )
    Q_PROPERTY(qulonglong result READ   resultValue
                                 NOTIFY resultValueChanged
              )

    qulonglong m_nInput;
    qulonglong m_nResult;

public:
    Calculation(QObject* pobj = 0);

    Q_INVOKABLE qulonglong factorial(const qulonglong& n);

    qulonglong inputValue   (                ) const;
    void       setInputValue(const qulonglong&);
    qulonglong resultValue  (                ) const;

signals:
    void inputValueChanged (qulonglong);
    void resultValueChanged(qulonglong);
};
```

В конструкторе (листинг 60.11) мы инициализируем свойства `input` и `result` начальными значениями.

Далее опишем методы класса `Calculation`:

- метод `factorial()` производит рекурсивное вычисление факториала;
- метод `inputValue()` — это реализация чтения свойства `input`, он возвращает его значение;
- метод `resultValue()` — это реализация чтения свойства `result`, он возвращает его значение;
- метод `setInputValue()` — это реализация записи свойства `input`. Он производит присвоение этому свойству новых значений. После присвоения нового значения вызовом

метода `factorial()` осуществляется повторное вычисление нового значения факториала. В результате изменяются два значения, и в конце выводится уведомление об изменении двух свойств высылкой сигналов: `inputValueChanged()` и `resultValueChanged()`.

Листинг 60.11. Конструктор класса `Calculation` (файл Calculation.cpp)

```
#include "Calculation.h"
// ----------------------------------------------------------------------
Calculation::Calculation(QObject* pobj) : QObject(pobj)
                                        , m_nInput(0)
                                        , m_nResult(1)
{
}

// ----------------------------------------------------------------------
qulonglong Calculation::factorial(const qulonglong& n)
{
    return n ? (n * factorial(n - 1)) : 1;
}

// ----------------------------------------------------------------------
qulonglong Calculation::inputValue() const
{
    return m_nInput;
}

// ----------------------------------------------------------------------
qulonglong Calculation::resultValue() const
{
    return m_nResult;
}

// ----------------------------------------------------------------------
void Calculation::setInputValue(const qulonglong& n)
{
    m_nInput  = n;
    m_nResult = factorial(m_nInput);

    emit inputValueChanged(m_nInput);
    emit resultValueChanged(m_nResult);
}
```

В QML-программе, показанной в листинге 60.12, мы включаем наш модуль `com.myinc.Calculaition` и создаем элемент `Calculation` с идентификатором `calc`. В элементе вертикального размещения `ColumnLayout` мы используем два элемента горизонтального размещения `RowLayout`. В верхнем элементе мы демонстрируем вызов метода `factorial()`, который был определен в классе `Calculation` как `Q_INVOKABLE`. Его вызов происходит в элементе `Text` (см. свойство `text`) при изменениях значения счетчика. Доступ к измененному

значению счетчика мы получаем при помощи свойства `value` идентификатора `sbx`, а доступ к элементу вычислений `Calculation` — при помощи идентификатора `calc`.

В нижнем элементе горизонтального размещения `RowLayout` мы демонстрируем другой подход с использованием свойств. При каждом элементе счетчика в его свойстве обработки события `onValueChanged` мы присваиваем свойству `input` элемента `Calculation` актуальное значение. Мы знаем из листинга 60.11 (см. метод `setInputValue()`), что подобное изменение повлечет за собой также и изменение свойства `result` элемента `Calculation`, поэтому отображаем это свойство в элементе `Text` (см. свойство `text`).

Листинг 60.12. QML-программа (файл main.qml)

```
import QtQuick 2.8
import QtQuick.Controls 2.2
import QtQuick.Layouts 1.3
import com.myinc.Calculation 1.0

ApplicationWindow {
    title: "Factorial Calculation"
    width: 250
    height: 80
    visible: true

    Calculation {
        id: calc
    }

    ColumnLayout {
        anchors.fill: parent
        RowLayout { // 1. call of an invokable method
            SpinBox {
                id: sbx
                value: 0
            }
            Text {
                text: "Result:" + calc.factorial(sbx.value)
            }
        }
        RowLayout { // 2. using of the properties
            SpinBox {
                value: 0
                onValueChanged: calc.input = value
            }
            Text {
                text: "Result:" + calc.result
            }
        }
    }
}
```

Как вы уже заметили, мы не использовали в наших двух предыдущих подходах уведомлений об изменении свойств, то есть сигналов, которые высылаются в листинге 60.11 (см. метод `setInputValue()`). Давайте теперь реализуем третий подход, в котором мы будем отлавливать изменения свойства `result` (листинг 60.13).

Листинг 60.13. Альтернативное решение с использованием сигналов

```
ApplicationWindow {
    title: "Factorial Calculation"
    width: 250
    height: 40
    visible: true

    Calculation {
        input: sbx.value
        onResultValueChanged: txt.text = "Result:" + result
    }
    RowLayout {
       SpinBox {
           id: sbx
           value: 0
       }
       Text {
           id: txt
       }
   }
}
```

В листинге 60.13 мы соединяем свойство `input` в элементе `Calculation` со свойством `value` элемента счетчика `SpinBox`. В свойстве `onResultValueChanged` мы отлавливаем изменения вычисленных значений факториала и отображаем их в текстовом элементе при помощи идентификатора `txt` (свойство `text`).

Реализация визуальных элементов QML на C++

Модуль `QtQuick` предоставляет базовые визуальные элементы, такие как, например, `Rectangle` и `Text`. Интересно знать, что все они реализованы на C++ классами `QQuickRectangle` и `QQuickText`. Оба этих класса, в свою очередь, унаследованы от знакомого нам класса `QQuickItem`. Для реализации собственного визуального элемента на C++ можно наследовать этот класс, но есть и более удобный класс `QQuickPaintedItem`, который уже унаследован от класса `QQuickItem`. Этот класс создавался специально для упрощения написания визуальных элементов. Так давайте воспользуемся этим классом и заодно дополним язык QML еще одним полезным визуальным элементом — эллипсом, который показан на рис. 60.6.

Наш новый класс `Ellipse` будет унаследован от класса `QQuickPaintedItem`. Для установки цвета, заполняющего площадь эллипса, мы добавим свойство `color` (см. листинг 60.14) при помощи макроса `Q_PROPERTY`. Этому свойству мы предоставим возможности чтения и записи значений (`WRITE` и `READ`): чтение обеспечит метод `colorValue()`, а запись — метод `setColorValue()`. Метод `paint()` отвечает за рисование элемента и вызывается всякий раз,

когда его требуется перерисовать снова, — это может потребоваться, например, при изменениях размеров элемента. Этот метод можно грубо сравнить с методом события рисования `QWidget::paintEvent()` с той лишь разницей, он получает не указатель на объект события `QPaintEvent`, а указатель на сам объект рисования `QPainter`, так что создавать отдельный объект `QPainter` нам не придется.

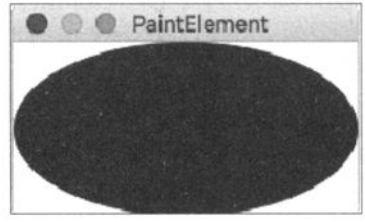


Рис. 60.6. QML-элемент эллипс, реализованный на C++

Листинг 60.14. Определение класса `Ellipse` (файл Ellipse.h)

```
#pragma once
#include <QQuickPaintedItem>
class QPainter;

// ====================================================================
class Ellipse : public QQuickPaintedItem {
Q_OBJECT
private:
    Q_PROPERTY(QColor color WRITE  setColorValue
                           READ   colorValue
               )
    QColor m_color;

public:
    Ellipse(QQuickItem* pqi = 0);
    void paint(QPainter* ppainter);

    QColor colorValue   (            ) const;
    void   setColorValue(const QColor&);
};
```

В конструкторе класса `Ellipse` (см. листинг 60.15) мы инициализируем значение атрибута цвета `m_color` черным цветом `Qt::black`. В методе `paint()` мы устанавливаем режим сглаживания вызовом метода `setRenderHint()`. Затем устанавливаем в объекте кисти `QBrush` значение цвета заполнения в соответствии с его текущим значением, которое возвращает метод `colorValue()`. Для того чтобы не отображалась контурная линия, мы присваиваем в методе `setPen()` нулевой объект пера `Qt::NoPen`. Последним следует вызов метода отображения эллипса — его размеры устанавливаются вызовом метода `boundingRect()`, который управляется из QML свойствами `x`, `y`, `width` и `height`. Методы `setColorValue()` и `colorValue()` прикреплены к свойству `color` (см. `Q_PROPERTY` в листинге 60.14) — они отвечают за установку и возвращение значения цвета, хранящегося в атрибуте `m_color`.

Листинг 60.15. Конструктор класса Ellipse (файл Ellipse.cpp)

```
#include <QPainter>
#include "Ellipse.h"

// ------------------------------------------------------------------------
Ellipse::Ellipse(QQuickItem* pqi /*= 0*/) : QQuickPaintedItem(pqi)
                                          , m_color(Qt::black)
{
}

// ------------------------------------------------------------------------
void Ellipse::paint(QPainter* ppainter)
{
    ppainter->setRenderHint(QPainter::Antialiasing, true);
    ppainter->setBrush(QBrush(colorValue()));
    ppainter->setPen(Qt::NoPen);
    ppainter->drawEllipse(boundingRect());
}

// ------------------------------------------------------------------------
QColor Ellipse::colorValue() const
{
    return m_color;
}

// ------------------------------------------------------------------------
void Ellipse::setColorValue(const QColor& col)
{
    m_color = col;
}
```

Наш класс C++ называется `Ellipse`, и для того чтобы сделать его доступным в QML, нам необходимо зарегистрировать его в основной программе (листинг 60.16). Для этого мы вызываем шаблонную глобальную функцию `qmlRegister<T>()` и типизируем ее нашим классом `Ellipse`. В первом параметре в эту функцию мы передаем имя для модуля, что обеспечит возможность его включения в QML-программу. Вторым параметром следует номер версии, третьим — номер уровня версии модуля, в который входит наш элемент. А в последнем, четвертом, параметре мы передаем имя, под которым будет доступен наш элемент в QML-программе.

Листинг 60.16. Регистрация класса Ellipse (файл main.cpp)

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include "Ellipse.h"

int main(int argc, char** argv)
{
    QGuiApplication app(argc, argv);
```

```
    qmlRegisterType<Ellipse>("com.myinc.Ellipse", 1, 0, "Ellipse");

    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));

    return app.exec();
}
```

Воспользоваться нашим новым элементом эллипсом просто. Нужно импортировать элемент `Ellipse` при помощи директивы `import` (листинг 60.17) и добавить его в элемент основного окна программы. Для того чтобы наш элемент всегда заполнял область основного окна приложения, мы используем фиксатор `anchors.fill` и при помощи реализованного свойства `color` задаем эллипсу голубой цвет. При изменении размеров окна наш элемент будет автоматически изменять свои размеры и заполнять всю его область.

Листинг 60.17. Использование элемента `Ellipse` (файл main.qml)

```
import QtQuick 2.8
import QtQuick.Controls 2.2
import QtQuick.Window 2.2
import com.myinc.Ellipse 1.0

Window {
    title: "PaintElement"
    visible: true
    width: 200
    height: 100

    Ellipse {
        anchors.fill: parent
        color: "blue"
    }
}
```

Класс *QQuickImageProvider*

Для операций с растровыми изображениями можно было бы, как и в предыдущем случае, использовать класс `QQuickPaintedItem`, но в этот раз мы продемонстрируем другой подход и воспользуемся классом `QQuickImageProvider`. Унаследуем его и реализуем алгоритм для обработки изображения. Интересно то, что в QML элемент от этого класса будет представлен как обычная ссылка на файл.

Следующий пример (листинги 60.18–60.23) демонстрирует использование класса `QQuickImageProvider` (рис. 60.7). В этом примере элементы управления и отображения растрового изображения реализованы на QML, а алгоритм изменения яркости — на C++.

В основной программе, показанной в листинге 60.18, мы создаем объект класса `QQmlApplicationEngine`. Этот объект предоставляет среду исполнения QML-кода с элементом основного окна приложения `ApplicationWindow`. Вызов метода `addImageProvider()` добавляет объект класса `ImageProvider`, который мы унаследовали от `QQuickImageProvider` и снабдили алгоритмом обработки изображения. Первый параметр — это строка



Рис. 60.7. Использование класса `QQuickImageProvider` в QML

`"brightness"`, по которой мы будем обращаться к нашему объекту. Метод `load()` производит загрузку QML-файла из ресурса.

Листинг 60.18. Создание объекта класса `QQmlApplicationEngine` (файл main.cpp)

```
#include <QGuiApplication>
#include <QQmlApplicationEngine>
#include "ImageProvider.h"

int main(int argc, char *argv[])
{
    QGuiApplication app(argc, argv);

    QQmlApplicationEngine eng;
    eng.addImageProvider(QLatin1String("brightness"), new ImageProvider);
    eng.load(QUrl(QStringLiteral("qrc:/main.qml")));

    return app.exec();
}
```

Класс `ImageProvider` наследуется от класса `QQuickImageProvider` (листинг 60.19). В его методе `brightness()` мы реализуем алгоритм для увеличения/уменьшения яркости. Метод `requstedImage()` передает сгенерированное изображение и имеет три параметра для коммуникации с QML.

Листинг 60.19. Определение класса ImageProvider (файл ImageProvider.h)

```
#pragma once
#include <QObject>
#include <QImage>
#include <QQuickImageProvider>

// ============================================================================
class ImageProvider : public QQuickImageProvider {
private:
    QImage brightness(const QImage& imgOrig, int n);

public:
    ImageProvider();

    QImage requestImage(const QString&, QSize*, const QSize&);
};
```

Конструктор (листинг 60.20) передает в конструктор базового класса `QQuickImageProvider` значение `Image`. Это значение говорит о том, что мы будем возвращать растровые изображения в объектах класса `QImage`.

> ***Возвращение растровых изображений в объектах класса QPixmap***
>
> Если бы мы намеревались возвращать растровые изображения в объектах класса `QPixmap`, то в конструктор `QQuickImageProvider` нужно было бы передать значение `Pixmap` и реализовать метод `requestedPixmap()`.

Листинг 60.20. Конструктор класса ImageProvider (файл ImageProvider.cpp)

```
ImageProvider::ImageProvider()
    : QQuickImageProvider(QQuickImageProvider::Image)
{
}
```

На алгоритме увеличения/уменьшения яркости, который приведен в листинге 60.21, мы останавливаться не будем, — его описание можно найти в *разд. «Класс QImage» главы 19.*

Листинг 60.21. Алгоритм увеличения/уменьшения яркости (файл ImageProvider.cpp)

```
QImage ImageProvider::brightness(const QImage& imgOrig, int n)
{
    QImage imgTemp = imgOrig;
    qint32 nHeight = imgTemp.height();
    qint32 nWidth  = imgTemp.width();

    for (qint32 y = 0; y < nHeight; ++y) {
        QRgb* tempLine = reinterpret_cast<QRgb*>(imgTemp.scanLine(y));

        for (qint32 x = 0; x < nWidth; ++x) {
            int r = qRed(*tempLine) + n;
```

```
            int g = qGreen(*tempLine) + n;
            int b = qBlue(*tempLine) + n;
            int a = qAlpha(*tempLine);

            *tempLine++ = qRgba(r > 255 ? 255 : r < 0 ? 0 : r,
                                g > 255 ? 255 : g < 0 ? 0 : g,
                                b > 255 ? 255 : b < 0 ? 0 : b,
                                a
                               );
        }
    }

    return imgTemp;
}
```

Метод `requestImage()` является связующим звеном между QML и C++ (листинг 60.22). Его задача — создание растровых изображений. Он получает первым параметром идентификационную строку, в которой мы станем передавать два значения: имя растрового файла и значение яркости. Эти значения мы будем разделять в QML символом `;`. Второй агрумент (указатель `ps`) будет содержать информацию о размере изображения. Третий аргумент мы игнорируем, он нужен на тот случай, если мы захотим получить из QML изображение с заданными размерами.

В методе при помощи разделителя `;` мы разбиваем методом `QString::slit()` строку `strId` на два строковых значения и преобразуем второе значение из строки в тип `int` (переменная `nBrightness`) — это значение яркости. Передаем в метод `brighness()` первым параметром объект класса `QImage`, который сконструирован из имени файла — в нашем случае файл находится в ресурсе. Вторым параметром в метод `brightness()` передается значение яркости `nBrightness`. Этот метод возвращает объект `QImage`, мы его сохраняем в переменной `img`. Далее, если указатель `ps` не нулевой, мы заполняем объект `QSize`, на который он указывает, размером созданного растрового изображения. В завершение мы возвращаем объект растрового изображения.

Листинг 60.22. Метод создания растрового изображения (файл ImageProvider.cpp)

```
QImage ImageProvider::requestImage(const QString& strId, QSize* ps, const QSize&
/*requestedSize*/)
{
    QStringList lst = strId.split(";");
    bool        bOk = false;
    int         nBrightness = lst.last().toInt(&bOk);
    QImage      img = brightness(QImage(":/" + lst.first()), nBrightness);

    if (ps) {
        *ps = img.size();
    }

    return img;
}
```

Листинг 60.23 демонстрирует использование в QML класса изменения яркости растрового изображения. Размеры основного окна приводятся в соответствие с размерами вертикального размещения (идентификатор `controls`) при помощи свойств `width` и `height`. Элемент вертикального размещения содержит три элемента: `Image`, `Slider` и `Text`. Элемент `Image` показывает растровое изображение. Обратите внимание, что мы получаем изображение, которое вычисляется с помощью ранее реализованного класса `ImageProvider`, обычной строкой в свойстве `source`. В этой строке сразу после указания типа ссылки `image` мы приводим ссылку `brightness` на наш объект создания растровых изображений с изменением яркости. Далее указываем имя файла `Alina.png`, который у нас содержится в ресурсе, и после знака `;` — значение яркости. Значение яркости мы берем из элемента ползунка с идентификатором `sld`. Элемент ползунка имеет длину, одинаковую с шириной растрового изображения, которую мы устанавливаем в свойстве `width`. Текущее положение позиции ползунка 75%, его мы устанавливаем свойством `value`. Так как диапазон хода ползунка лежит в диапазоне от 0 до 1, то, чтобы создать количество шагов, равное 100, нужно присвоить свойству `stepSize` значение `0.01`. Для удобства мы создаем новое свойство `brightnessValue`, которое будет предоставлять значение яркости в зависимости от положения ползунка. Это значение мы используем как в элементе `Image` для вывода изображения с выбранной яркостью, так и в элементе `Text` для отображения самого значения яркости в виде числа.

Листинг 60.23. Использование класса `ImageProvider` (файл main.qml)

```
import QtQuick 2.8
import QtQuick.Controls 2.2

ApplicationWindow {
    title: qsTr("Image Brightness")
    width: controls.width
    height: controls.height
    visible: true

    Column {
        id: controls
        Image {
            id: img
            source: "image://brightness/Alina.png;" + sld.brightnessValue
        }
        Slider {
            id: sld
            width: img.width
            value: 0.75
            stepSize: 0.01
            property int brightnessValue: (value * 255 - 127)
        }
        Text {
            width: img.width
            text: "<h1>Brightness:" + sld.brightnessValue + "</h1>"
        }
    }
}
```

Резюме

Технология Qt Quick разработана с учетом возможности расширения из библиотеки Qt и языка C++. Поскольку класс `QQuickWidget`, который отвечает за отображение QML-элементов, базируется на классе `QWidget`, то его допускается использовать в Qt-программах как обычный виджет. С его помощью можно получать доступ к элементам QML и их свойствам.

Предоставление языку QML возможностей, реализованных на основе библиотеки Qt и языка C++, осуществляется посредством класса `QQmlContext`. Этот класс предоставляет метод `setContextPropety()`, с помощью которого можно опубликовать объекты C++ и сделать их доступными для использования в языке QML. При помощи функции `qmlRegisterType<T>()` можно публиковать классы и делать их свойства, сигналы, слоты и методы, объявленные как `Q_INVOKABLE`, также доступными для QML.

Класс `QQuickPaintedItem` предоставляет возможности для реализации визуальных элементов QML из C++.

Класс `QQuickImageProvider` содержит механизм для создания в C++ растровых изображений, которые можно использовать в QML в виде ссылки на файл.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/60-510/** или с помощью следующего QR-кода (рис. 60.8):



Рис. 60.8. QR-код для доступа на страницу комментариев к этой главе

ГЛАВА 61

3D-графика Qt 3D

Да здравствует все, благодаря чему, мы, несмотря ни на что!
Зиновий Паперный

Набор модулей Qt 3D предоставляет общую структуру для моделирования трехмерных сцен в режиме реального времени. Все эти модули реализованы на C++ и их можно использовать как из самого языка C++, так и из языка QML. Как известно, язык QML является описательным языком, и поэтому он как нельзя лучше, по своей природе, подходит для описания трехмерных сцен. Именно его использование даст нам возможность создавать трехмерные сцены на более высоком уровне, а это значит более просто и быстро.

Основы

Для начала проясним, что же такое *трехмерное пространство*. Трехмерное пространство — это естественная среда нашего обитания, в нем мы перемещаемся, взаимодействуем друг с другом и объектами, находящимися в нем, словом, живем там. Трехмерное пространство на дисплее компьютера или мобильного устройства — это виртуальная модель нашей среды обитания, и оно, так же, как и реальное пространство, неограниченно велико. Для того чтобы не потеряться и быть способным ориентироваться в виртуальном трехмерном пространстве, для определения местоположения там необходимы три направления: X, Y и Z (рис. 61.1). Первое задает ширину, второе — высоту, а третье — глубину. Виртуальное пространство определено сценой, в центре которой расположена начальная точка отсчета по этим направлениям.

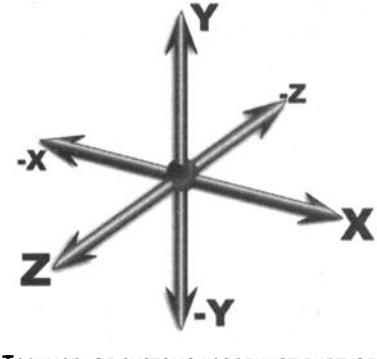


Рис. 61.1. Трехмерная система координат виртуальной сцены

За реализацию виртуальной сцены — показ трехмерной сцены с объектами — в Qt 3D отвечает элемент `Scene3D` из модуля `QtQuick.Scene3D`. Все, что находится внутри сцены: 3D-объекты, камера, свет — все это должно быть выражено в форме *компонентов сущностей*.

Система компонентов сущностей (Entity Component System, ECS) — это популярный шаблон, который хорошо зарекомендовал себя и получил признание в игровых движках, симуляторах и поэтому был заимствован в Qt 3D. Ее основное преимущество — в гибкости изменения поведения сущностей, которое может проводиться в режиме реального времени, путем добавления либо удаления компонентов без прерывания процесса работы программы.

Сущности реализуются в QML элементами `Entity`. Сама по себе сущность не несет в себе ни поведения, ни характеристик. *Поведение* может быть добавлено к сущности путем объединения одного или нескольких компонентов. Сущности — это, по сути, контейнеры, в которые можно добавлять компоненты. Например, можно добавить геометрический образ объекта и к нему добавить элемент поведения, скажем, трансформации.

Для того чтобы отобразить в трехмерной сцене элемент, его необходимо оформить в виде сущности. Таким образом, для отображения 3D-объекта нам понадобится объединить в элементе сущности его геометрический образ (элемент `Mesh`), материал (элемент `Material`), дающий оболочку, и трансформации (элемент `Transform`). Вот, как это может выглядеть в программном коде (листинг 61.1).

Листинг 61.1. Объединение элементов в сущность

```
Entity {
    Mesh{id: myMesh; ...}
    PhongMaterial{id: myMaterial...}
    Transform{id: myTansform;...}
    component: [myMesh, myMaterial, myTransform]
}
```

Свет

Поскольку глубина тени 3D-объектов очень зависит от источников света, то свет играет в процессе визуализации трехмерных сцен весьма важную роль. Именно он придает 3D-объектам реалистичность и создает настроение восприятия всей трехмерной сцены. Например, чрезмерно сильно освещение может сделать сцену плоской. А использование света в виде луча прожектора может привлечь внимание к определенному месту сцены или объекта. Интенсивный цвет способен сформировать острые и четкие тени от объектов, а мягкий свет — более размытые, что подчеркнет пространство и создаст еще большую реалистичность восприятия.

Итак, да будет свет! Qt 3D предоставляет три вида источников света (рис. 61.2):

- *точечный свет* (рис. 61.2, *слева*) — реализован элементом `PointLight`, излучает свет от центра и сразу во всех направлениях. Более далеко расположенные от источника объекты освещены меньше всего. Этот тип света отлично подходит для имитации освещения от лампочки, свечи или факела;
- *направленный свет* (рис. 61.2, *в центре*) — реализован элементом `DirectionalLight`, излучает множество лучей, которые поступают извне и освещают сцену с бесконечного расстояния. Все объекты сцены получают одинаковые порции света вне зависимости от их расположения. Этот тип света подходит для имитации света от Солнца и Луны;

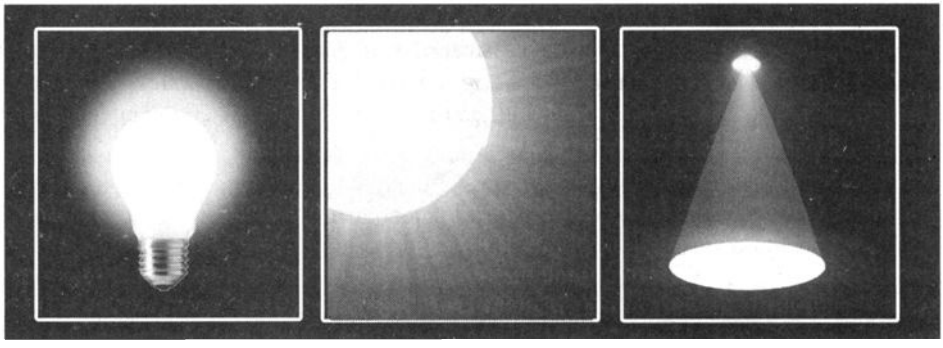


Рис. 61.2. Три вида источников света Qt 3D: точечный (*слева*), направленный (*в центре*), прожекторный (*справа*)

- *прожекторный свет* (рис. 61.2, *справа*) — реализован элементом `SpotLight`, излучает свет из центральной точки, лучи откуда расходятся в виде конуса. Этот тип света подходит для имитации настольных ламп, прожекторов, фар автомобилей, студийного освещения и фонарей.

Все источники света имеют общие свойства:

- *цвет* (свойство `color`) — управление цветом освещения. Для большей реалистичности не рекомендуется использовать однотонные цвета. Например, не стоит задавать в качестве дневного цвета чисто белый цвет, будет гораздо лучше, если вы добавите к нему какой-нибудь оттенок, предположим, желтый;
- *интенсивность* (свойство `intensity`) — управляет яркостью освещения. Для достижения хороших результатов рекомендуется начинать с минимального значения интенсивности и понемногу ее увеличивать, пока не будет достигнут оптимальный результат.

Листинг 61.2 демонстрирует, как можно создать простейший точечный источник света с голубоватым оттенком.

Листинг 61.2. Создание элемента точечного света

```
PointLight {
    color: "#afafff"
}
```

Камера

Теперь проясним, что такое *камеры* и для чего они нужны. В реальном мире камера нам нужна, чтобы снять фотографию. То же самое назначение и у камер в виртуальном пространстве, но если выражаться более точно, то они нужны, чтобы получить проекцию реального трехмерного мира на плоскости.

Камеры в Qt 3D, в отличие от камер реального мира, это не визуальные объекты, то есть их не видно на сцене. Их можно размещать на любой позиции трехмерной сцены и регулировать угол захвата для получения необходимого изображения. Положение камеры, как и любого другого объекта сцены, можно изменять во времени, то есть анимировать. Эту возможность можно использовать, например, для того чтобы показать движение внутри или вдоль объекта, скажем, здания, туннеля и т. д.

Камера Qt 3D реализуется элементом `Camera`. Простой вариант ее создания показан в листинге 61.3. В первом свойстве мы указываем тип для проекции «перспективный»: `PerspectiveProjection`, затем назначаем угол захвата объектива (свойство `fieldOfView`) 90 градусов и отводим камеру по оси Z на 40 единиц от центра трехмерной сцены (свойство `position`).

Листинг 61.3. Создание элемента камеры

```
Camera {
    projectionType: CameraLens.PerspectiveProjection
    fieldOfView: 90
    position: Qt.vector3d( 0.0, 0.0, 40.0 )
}
```

Наряду с перспективной проекцией Qt 3D предоставляет так же ортогональную (`OrthographicProjection`) и пирамидальную проекции (`FrustrumProjection`).

3D-объекты

3D-объекты состоят из многоугольников, которые задаются координатами вершин, и описания поверхностей и нормальных векторов к ним. Для загрузки 3D-объектов Qt 3D использует формат OBJ. Это очень простой текстовый формат, который был разработан компанией Wavefront Technoligies, стал очень популярен и теперь является общепринятым форматом. Он может импортироваться и экспортироваться практически во все популярные программы создания 3D-графики — такие как, например, Maya, 3ds Max и Blender. Последняя программа является свободно доступной для Windows, Mac OS X и Linux, и вы можете загрузить ее по ссылке: **https://www.blender.org/download/**.

Рассмотрим формат OBJ на примере геометрического образа пирамиды. Пирамида — это фигура, которая имеет прямоугольное основание и четыре боковые грани, сходящиеся на вершине (рис. 61.3).

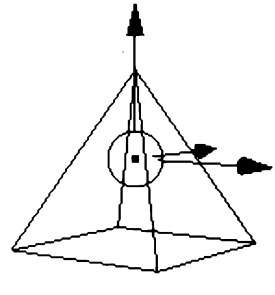


Рис. 61.3. Пирамида в трехмерном пространстве

Геометрические данные для пирамиды в формате OBJ показаны в листинге 61.4. В этом формате комментарии следуют после символа «`#`». Символ «`o`» задает имя объекта. За символами «`v`» следуют координаты вершин объекта. Символами «`vn`» задаются описания векторов, перпендикулярных к лицевой поверхности (нормали). За символом «`f`» идут описания поверхностей объекта.

Листинг 61.4. Пирамида в формате OBJ (Файл pyramid.obj)

```
# Blender v2.79 (sub 0) OBJ File: ''
# www.blender.org
o Shape_IndexedFaceSet
v 1.000000 -1.000000 -1.000000
v -1.000000 -1.000000 -1.000000
v -1.000000 -1.000000 1.000000
v 1.000000 -1.000000 1.000000
v 0.000000 1.000000 -0.000000
vn 0.0000 0.4472 -0.8944
vn -0.8944 0.4472 -0.0000
vn -0.0000 0.4472 0.8944
vn 0.8944 0.4472 0.0000
f 1//1 2//1 5//1
f 2//2 3//2 5//2
f 3//3 4//3 5//3
f 4//4 1//4 5//4
```

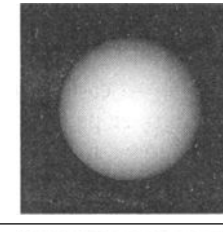
Загружать данные из OBJ-файлов позволяет элемент `Mesh`. Вот как можно загрузить только что рассмотренный нами файл `pyramid.obj` из листинга 61.4:

```
Mesh {
    id: mesh
    source: "pyramid.obj"
}
```

Совсем необязательно создавать элемент `Mesh` и загружать геометрии объектов. Модуль `Qt3D.Extras` предоставляет коллекцию стандартных форм трехмерных объектов. К этим формам относятся такие наиболее распространенные объекты, как, например, сфера, параллелепипед, цилиндр и др. (табл. 61.1).

Таблица 61.1. *Стандартные 3D-объекты модуля Qt3D.Extras 2.0*

Элемент	Пример	Вид

Сфера	`SphereMesh {`
	`  radius: 3 // радиус`
	`}`
		

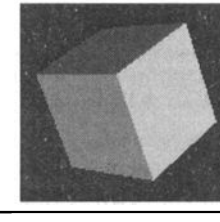
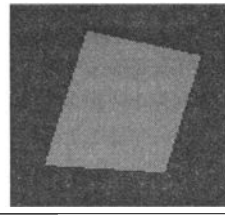
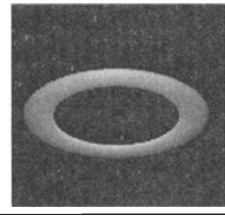
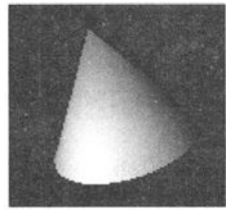
Параллелепипед	`CuboidMesh {`
	`  // размеры лицевых сторон`
	`  yzMeshResolution: Qt.size(2, 2)`
	`  xzMeshResolution: Qt.size(2, 2)`
	`  xyMeshResolution: Qt.size(2, 2)`
	`}`
		

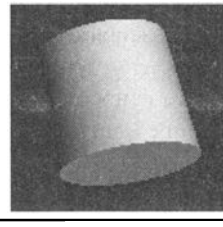
Таблица 61.1 *(окончание)*

Элемент	Пример	Вид

Плоскость	`PlaneMesh {`
	`  width: 50 // ширина`
	`  height: 50 // высота`
	`}`
		

Тор	`TorusMesh {`
	`  radius: 5 // внешний радиус`
	`  minorRadius: 1 // внутренний радиус`
	`  rings: 100 // количество колец`
	`  //корпуса`
	`}`
		

Конус	`ConeMesh {`
	`  topRadius: 0 // радиус верхнего`
	`  //основания`
	`  bottomRadius: 1 // радиус нижнего`
	`  //основания`
	`  length: 3 // высота`
	`  rings: 50 // количество колец`
	`  //корпуса`
	`}`
		

Цилиндр	`CylinderMesh {`
	`  radius: 1 // радиус оснований`
	`  length: 3 // высота`
	`  rings: 100 // количество колец`
	`  //корпуса`
	`}`
		

В столбце «Пример» табл. 61.1 для каждого элемента приведен код, который можно копировать в программы 3D-сцен. Теперь рассмотрим, как можно использовать некоторые из этих элементов:

- *сфера* — самый простой элемент. Для того чтобы ее задать, необходимо только лишь указать ее радиус. В реальной жизни объекты в форме сфер можно встретить очень часто — это, например, теннисные мячики, бильярдные шары, жемчуг, воздушные шары и т. п.;
- *параллелепипед* — встречается очень часто, и его можно использовать, например, для формирования основания зданий, а если поставить рядом вертикальные цилиндры, то может получиться что-то, похожее на древнеримское здание с колоннами;
- *тор* — из него можно сделать бублик, а также шину для велосипеда;
- *конус* — из этого объекта может получиться прекрасный стаканчик для мороженого. Да вы, наверное, уже заметили, что из конуса можно сделать и *цилиндр* — для чего всего лишь нужно свойствам `topRadius` и `bottomRadius` присвоить одинаковые значения;

- *плоскость* — может оказаться очень полезной для экспериментов со светом и тенью, если расположить на ее поверхности 3D-объекты.

Другими словами, стандартные 3D-объекты могут быть использованы в качестве строительного материала для более сложных объектов. Например, из пяти сфер, двух цилиндров и одного конуса можно составить такого вполне симпатичного снеговика (рис. 61.4).

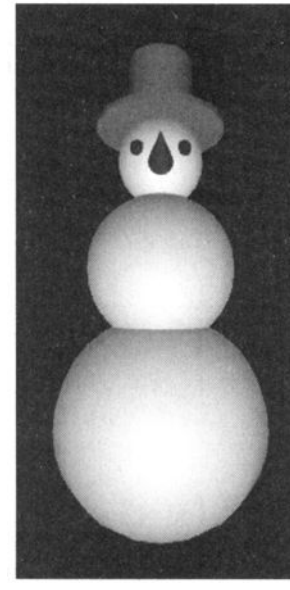


Рис. 61.4. Снеговик, составленный из пяти сфер, двух цилиндров и одного конуса

Материалы

Отобразить геометрический образ нам пока не представляется возможным, так как для его отображения нужно добавить в сущность *материал* и тем самым создать для трехмерного объекта оболочку. Материалы отличаются друг от друга способностью отражать свет. Из реального мира мы знаем, что металлическая поверхность отражает свет более ярко, чем поверхность, покрытая бумагой. Qt 3D представляет следующие элементы для материалов:

- элемент `PhongMaterial` — представляет гладкие поверхности. Это довольно яркий материал, обеспечивающий на поверхности объектов формирование бликов, которые придают им блестящий или глянцевый вид. В качестве эквивалента этому материалу хорошо подойдут пластмасса и металл;
- элемент `GoochMaterial` — представляет поверхность без сглаживания, цвет которой остается постоянным. Отображение этого материала производится быстрее, чем в случае с `PhongMaterial`, поэтому его замечательно можно использовать для тестовых визуализаций. В качестве эквивалента этому материалу хорошо подойдут картон, бумага и некоторые типы древесины.

Эти два материала имеют следующие общие свойства:

- свойство `diffuse` (Диффузия) — задает основной цвет поверхности объекта значением типа `color`;
- свойство `shininess` (Свечение) — задает уровень свечения поверхности от 0 до 1;
- свойство `alpha` (Прозрачность) — управляет способностью поверхности объекта пропускать через себя определенное количество света. Задается значениями от 0 до 1.

Теперь у нас есть все необходимое, чтобы отобразить наш первый трехмерный объект, и это будет, конечно же, сфера, показанная на рис. 61.5. Нажатием на клавиши управления курсором, кнопки мыши или тачпад можно будет управлять изменением позиции камеры — чтобы иметь возможность рассмотреть наш объект на разных удалениях и в разных положениях.

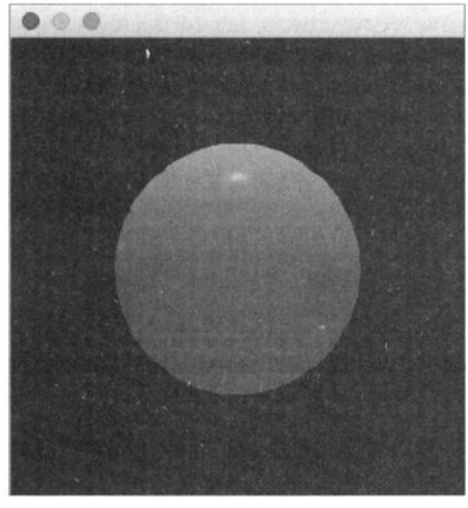


Рис. 61.5. Отображение сферы

В этом проекте мы воспользуемся некоторыми модулями пакета Qt 3D, поэтому секция `QT` проектного файла должна выглядеть следующим образом:

```
QT += quick qml 3dcore 3drender 3dinput 3dextras 3dquick 3dquickextras
```

В основной программе приложения на C++ (листинг 61.5) мы используем класс `QQuickView`. Вызов метода `setResizeMode()` из объекта этого класса со значением `QQuickView::SizeRootObjectToView` делает так, чтобы при изменении окна `QQuickView` автоматически производились изменения размеров основного QML-элемента в соответствии с новыми размерами окна. Метод `setSource()` загружает исходный код QML-программы из листинга 61.6 на выполнение.

Листинг 61.5. Основная программа приложения на C++ (файл main.cpp)

```
#include <QGuiApplication>
#include <QQuickView>
int main(int argc, char** argv)
{
    QGuiApplication app(argc,argv);
    QQuickView view;
    view.resize(300, 300);
    view.setResizeMode(QQuickView::SizeRootObjectToView);
    view.setSource(QUrl("qrc:/main.qml"));
    view.show();
    return app.exec();
}
```

В программе из листинга 61.6 мы импортируем целый ряд модулей из пакета Qt 3D. Затем создаем главный элемент окна приложения `Rectangle` и устанавливаем фон черного цвета (свойство `color`). Внутри этого элемента мы располагаем элемент трехмерной сцены `Scene3D` и при помощи фиксатора (свойство `anchors.fill`) делаем так, чтобы он заполнил всю его область. Для того чтобы сцена гарантированно получала ввод пользователя, уста-

навливаем свойству `focus` значение `true`, а для того чтобы сцена могла его интерпретировать, присваиваем свойству `aspects` массив из двух значений: `"input"` и `"logic"`. Мы не хотим, чтобы в случае изменения размеров окна при отображении происходило искажение объектов сцены, поэтому устанавливаем в свойстве `cameraAspectRatioMode` значение пропорционального отображения `Scene3D.AutomaticAspectRatio`. Далее идет элемент сущности `Entity`, который объединяет в себе все элементы трехмерной сцены:

- элемент камеры `Camera` — задаем значения ближнего (свойство `nearPlane`) и дальнего (свойство `farPlane`) фокусных расстояний и помещаем камеру на 50 единиц от начала положения координат вдоль оси Z (свойство `position`);
- элемент управления камерой `FirstPersonCameraController` — устанавливаем в свойстве `camera` уникальный идентификатор нашей камеры, чтобы дать возможность пользователям управлять созданной камерой при помощи мыши, курсоров, тачпада, сенсорного экрана и т. п.;
- свойство `components` — объединяет в сущности сцены настройки рендеринга с пользовательским вводом. *Рендеринг* — это процесс создания двумерного изображения из трехмерной сцены. Для осуществления рендеринга необходима камера, идентификатор которой мы присваиваем в свойстве `camera`. Для того чтобы был виден черный цвет фона главного элемента окна `Rectange`, мы делаем цвет очистки прозрачным (свойство `clearColor`);
- элемент света `DirectionaLight` — задает направленный источник света со слегка голубоватым оттенком (свойство `color`).

Сущность сферы состоит из двух элементов: элемента материала `PhongMaterial` и геометрии объекта сферы `SphereMesh` с радиусом 6 единиц. В элементе материала `PhongMaterial` установлены два свойства: `ambient` и `diffuse` — первое свойство управляет способностью поверхности отражать окружающую среду, а второе задает основной цвет поверхности объекта.

Листинг 61.6. QML-код описания сцены (файл main.qml)

```
import QtQuick 2.8
import Qt3D.Core 2.0
import Qt3D.Render 2.0
import Qt3D.Input 2.0
import Qt3D.Extras 2.0
import QtQuick.Scene3D 2.0

Rectangle {
    color: "black"
    Scene3D {
        anchors.fill: parent
        focus: true
        aspects: ["input", "logic"]
        cameraAspectRatioMode: Scene3D.AutomaticAspectRatio
        Entity {
            Camera { //Камера
                id: camera
                nearPlane : 0.1
```

```
                farPlane : 1000.0
                position: Qt.vector3d( 0.0, 0.0, 50.0 )
            }
            FirstPersonCameraController { //Управление положением камеры
                camera: camera
                linearSpeed: 1000.0
                acceleration: 0.1
                deceleration: 1.0
            }
            components: [
                RenderSettings {
                    activeFrameGraph:
                        ForwardRenderer {
                            camera: camera
                            clearColor: "transparent"
                        }
                },
                InputSettings { }
            ]
            DirectionalLight { //Свет
                color: "#afafff"
            }
            Entity {  //Сущность сферы
                PhongMaterial {
                    id: phongMaterial
                    ambient: Qt.rgba( 0.3, 0.3, 0.3, 1.0 )
                    diffuse: Qt.rgba( 1, 1, 1, 1 )
                }
                SphereMesh {
                    id: sphereMesh
                    radius: 6
                }
                components: [sphereMesh, phongMaterial]
            }
        }
    }
}
```

Повторное использование кода листингов

Далее для всех остальных примеров мы будем использовать похожий исходный код основной программы (листинг 61.5) и QML-код описания сцены (листинг 61.6), в которые станем вставлять другие сущности трехмерных объектов.

Трансформация

Трансформация — это изменения положения объекта вдоль любых из трех осей или плоскостей, а также его повороты вокруг собственной оси и изменение его размеров. За все эти операции отвечает элемент `Transform`, осуществляющий их с помощью свойств: `translate`,

rotation и scale. Трансформации помогут вам расположить объекты в пространстве так, чтобы из простых объектов составить более сложный. Показанный ранее на рис. 61.4 снеговик составлен следующим образом:

- тело — три сферы (элементы SphereMesh);
- глаза — две сферы (элементы SphereMesh);
- шапка — два цилиндра (элементы CylinderMesh);
- нос — конус (элемент ConeMesh).

В листинге 61.7 мы приведем только отрывок исходного кода снеговика, а именно то место, где мы задаем конус с материалом, и задействуем трансформацию для изображения носа снеговика в виде морковки. В качестве морковки мы используем конус (элемент ConeMesh), в котором задаем радиус верхнего основания равным 0 (свойство topRadius) и нижнего основания — равным 1 (свойство bottomRadius). Длине конуса (свойство lenght) присваиваем значение 5.

За трансформацию конуса отвечает элемент Transform. Мы присваиваем свойству scale значение 0.5 и тем самым уменьшаем размер конуса в два раза. Затем передаем свойству перемещения translation объект vector3d, параметры которого задаются в формате X, Y, Z, и смещаем конус на 14 единиц в положительном направлении оси Y и на 4 единицы в положительном направлении по оси Z. Поворот по оси X на 90 градусов производим в свойстве rotationX. Красноватый цвет морковки задаем в свойстве diffuse элемента материала PhongMaterial.

Листинг 61.7. Фрагмент программы «Конус с трансформацией» (нос снеговика)

```
Entity {
    ConeMesh { //Геометрия объекта
        id: coneMesh1
        topRadius: 0
        bottomRadius: 1
        length: 5
        rings: 50
    }
    Transform { //Изменение места положения
        id: coneTransform1
        scale: 0.5
        translation: Qt.vector3d(0, 14, 4)
        rotationX: 90
    }
    PhongMaterial {//Материал
        id: coneMaterial1
        diffuse: Qt.rgba( 1, 0.3, 0.2, 1 )
    }
    components: [coneMesh1, coneMaterial1, coneTransform1]
}
```

Применение трансформаций для камер и света

Трансформации изменения местоположения и угла поворота можно также применять к элементам света и камерам.

Анимация

Для анимации трехмерных объектов можно задействовать уже знакомые нам из *главы 58* приемы, которые мы использовали там для анимации элементов на плоскости.

Воспользуемся знакомым нам элементом `NumberAnimation` и применим его к трехмерным объектам сферы и пирамиды (рис. 61.6). В этой анимации объект сферы будет облетать вокруг объекта пирамиды на удалении, а пирамида, оставаясь на одном месте, станет вращаться по всем трем осям: X, Y и Z.

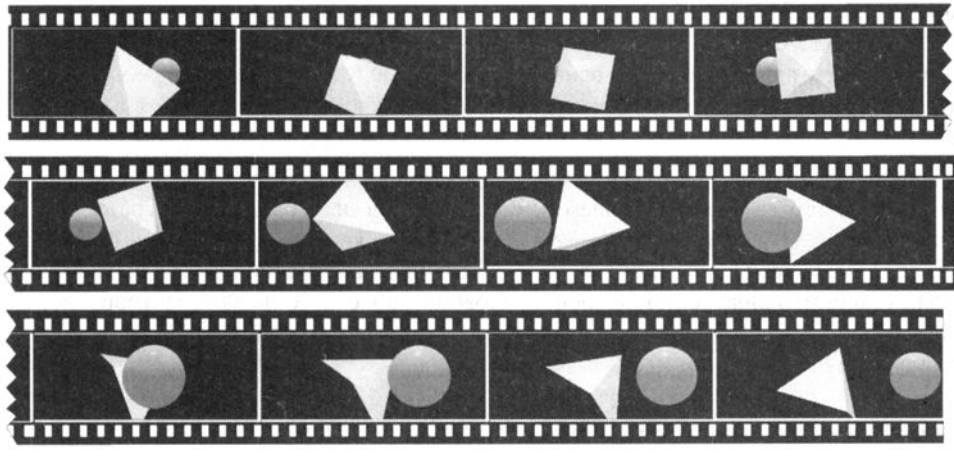


Рис. 61.6. Анимация двух объектов

Рассмотрим листинг 61.8, в котором описана сущность сферы. В этой сущности расположен элемент материала `PhongMaterial`, элемент геометрии сферы `SphereMesh`, а далее следуют элементы трансформации и анимации `Transform` и `NumberAnimation`. В элементе трансформации для управления углом поворота сферы мы создаем свойство `myParam`. А для самой трансформации в свойстве `matrix` формируем трансформационную матрицу (см. *разд. «Трансформационные матрицы» главы 18*). В первом параметре матрицы устанавливаем методом `rotate()` угол поворота. Этот угол поворота будет управляться свойством `myParam`. Вторым параметром при помощи объекта `vector3d` указываем ось, вокруг которой будет осуществляться вращение. Оси указываются в порядке: X, Y и Z — у нас только второй параметр равен 1, все остальные равны 0, следовательно, объект сферы будет вращаться по оси Y. Для того чтобы вращение происходило на удалении, мы производим в трансформационной матрице смещение. Это смещение производится вызовом метода `translate()`, в который передается объект `vector3d` с параметром X, равным 24-м единицам.

В элементе `NumberAnimation` мы указываем `sphereTransform` в качестве уникального идентификатора целевого объекта. В этом элементе мы будем изменять значения свойства `myParam`, поэтому указываем его в виде строки в свойстве `property`. В свойстве `duration` устанавливаем продолжительность цикла анимации, равной 10 секундам. Свойствам `from` и `to` присваиваются начальное и конечное значения угла поворота (0 и 360 градусов), который должен изменяться за каждый цикл. И для того чтобы анимация повторялась бесконечно, мы присваиваем свойству `loops` значение `Animation.Infinite`. Процесс запуска анимации происходит присвоением свойству `running` значения `true`.

Листинг 61.8. Анимация сферы (вращение вокруг оси Y)

```
Entity {
    PhongMaterial { //Материал
        id: phongMaterial
        ambient: Qt.rgba( 0.3, 0.3, 0.3, 1.0 )
        diffuse: Qt.rgba( 1, 1, 1, 1 )
    }
    SphereMesh { //Геометрия объекта сферы
        id: sphereMesh
        radius: 6
    }
    Transform { //Изменения положения объекта сферы
        id: sphereTransform
        property real myParam: 0
        matrix: {
            var mat = Qt.matrix4x4();
            mat.rotate(myParam, Qt.vector3d(0, 1, 0))
            mat.translate(Qt.vector3d(24, 0, 0));
            return mat;
        }
    }
    components: [sphereMesh, phongMaterial, sphereTransform]
    NumberAnimation { //Анимация объекта сферы
        target: sphereTransform
        property: "myParam"
        duration: 10000
        from: 0
        to: 360
        loops: Animation.Infinite
        running: true
    }
}
```

Программа листинга 61.9 реализует сущность для объекта пирамиды. Для нее мы возьмем материал `GoochMaterial`, отличный от материала, использованного нами для объекта сферы. В элемент `Mesh` загружаем файл `pyramid.obj` с данными геометрии пирамиды (см. листинг 61.4). В элементе `Transform` формируем трансформационную матрицу, в которой угол поворота будет управляться при помощи свойства `myRotation`. Значение этого свойства мы передаем первым параметром в метод `rotate()`. А для того чтобы поворот осуществлялся по всем трем осям, передаем вторым параметром объект `vector3d`, в котором все три значения устанавливаем равными 1. Вызовом метода `scale()`устанавливаем пирамиде нужный размер. В элементе анимации `NumberAnimation` поступаем аналогично тому, как показано в листинге 61.8.

Листинг 61.9. Анимация пирамиды (вращение вокруг трех осей)

```
Entity {
    GoochMaterial {//Материал
        id: goochMaterial
        diffuse: Qt.rgba( 1, 1, 1, 1 )
    }
    Mesh { //Геометрия объекта пирамиды
        id: pyramidMesh
        source: "qrc:/pyramid.obj"
    }
    Transform { // Повороты объекта пирамиды
        id: pyramidTransform
        property real myRotation: 0
        matrix: {
            var mat = Qt.matrix4x4();
            mat.rotate(myRotation, Qt.vector3d(1, 1, 1))
            mat.scale(Qt.vector3d(10, 10, 10));
            return mat;
        }
    }
    components: [pyramidMesh, goochMaterial, pyramidTransform]
    NumberAnimation { //Анимация объекта пирамиды
        target: pyramidTransform
        property: "myRotation"
        duration: 10000
        from: 0
        to: 360
        loops: Animation.Infinite
        running: true
    }
}
```

Qt 3D Studio

С недавнего времени арсенал Qt-разработчиков пополнился новым инструментом Qt 3D Studio (рис. 61.7). Он был создан для того, чтобы дизайнеры могли легко создавать трехмерные пользовательские интерфейсы, а разработчики их потом внедрять в свои приложения, написанные на Qt. То есть, этот инструмент задуман как расширение и дополнение уже существующих подходов в разработке программ при помощи виджетов, Qt Quick или Qt 3D.

Очень большой вклад в создание этого инструмента внесла компания NVIDIA, предоставив в распоряжение исходный код своей дизайнерской студии. Программа дизайнерской студии NVIDIA DRIVE Design Studio применяется во многих отраслях и хорошо зарекомендовала себя там. Несмотря на то, что до сих пор еще предстоит много работы по переводу участков кода из MFC на Qt, первая версия Qt 3D Studio уже готова. Вы можете ее загрузить уже сейчас, опробовать (см. *приложение 1*), а также получить больше информации о ней с официальной страницы документации Qt по адресу: **http://doc.qt.io/qt3dstudio/**.

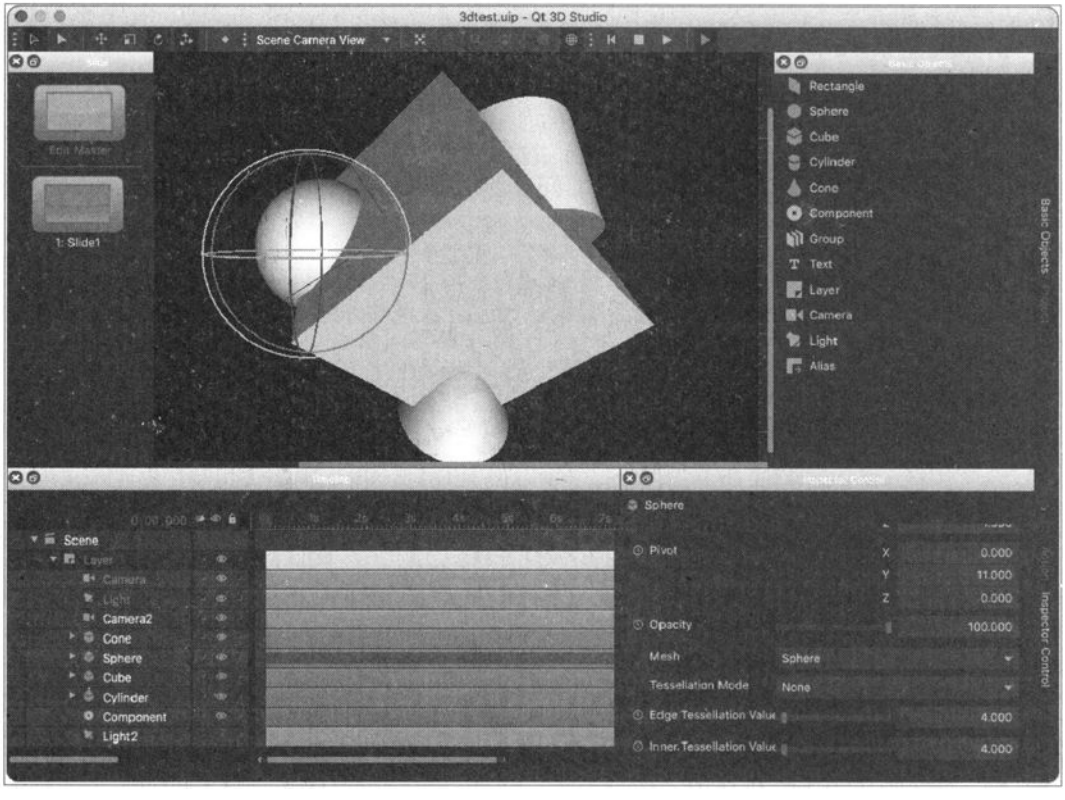


Рис. 61.7. Программа Qt 3D Studio

Резюме

Давайте подытожим то, что мы узнали из этой главы:

- трехмерная графика Qt 3D построена на системе сущностей;
- источники света делятся на три типа: точечный, направленный и прожекторный;
- для того чтобы проецировать трехмерные объекты, нам нужно направить камеру на них;
- благодаря поддержке формата OBJ в Qt 3D можно загружать геометрические данные трехмерных объектов из разных редакторов, используемых для создания трехмерной графики;
- модули Qt 3D предоставляют целый ряд стандартных форм трехмерных объектов — таких как сфера, конус, тор и т. п., которые можно сразу использовать в своих программах;
- геометрические данные трехмерных объектов — это их «физическое присутствие» на трехмерной сцене, но не отображение;
- для отображения объектов необходима поверхность, которая будет отражать свет. За поверхности и их свойства отвечают элементы материалов;
- всем трехмерным объектам, камерам и источникам света можно при помощи трансформаций изменять местоположение, а также угол их поворота;

- для анимации в трехмерном пространстве можно использовать некоторые из элементов, которые применялись для проведения анимации на плоскости;
- в арсенал Qt-разработчиков добавился еще один инструмент, предназначенный для создания трехмерных пользовательских интерфейсов: Qt 3D Studio.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/61-510/** или с помощью следующего QR-кода (рис. 61.8):



Рис. 61.8. QR-код для доступа на страницу комментариев к этой главе



ЧАСТЬ IX

Мобильные приложения и Qt

Сила не в знании, а в его применении!
Бодо Шефер

Глава 62. Введение в мир мобильных приложений

Глава 63. Подготовка к работе над мобильными приложениями

Глава 64. Особенности разработки приложений для мобильных устройств

Глава 65. Пример разработки мобильного приложения

Глава 66. Публикация в магазине мобильных приложений



ГЛАВА 62

Введение в мир мобильных приложений

> Безумцы, уверенные, что способны изменить мир, на самом деле его меняют.
>
> *Стив Джобс*

Все наверняка слышали, знают и уверены в том, что мы уже давно живем в информационную эру. И отсчет начала этого нового времени стартовал с того момента, когда привычная нам всем информация стала продуктом. В настоящее же время мы переживаем вторую волну информационной революции под грифом «Цифровая дистрибуция», потому что сейчас информация становится не просто продуктом, но, благодаря высоким скоростям и безлимитному Интернету, она получила полную независимость от физических носителей, на которых она раньше распространялась и поставлялась для продажи. А это значит, что информация полностью стала самостоятельным продуктом, и теперь уже не нужны компактные диски или DVD для того, чтобы слушать музыку или смотреть фильмы. Нет необходимости и в бумаге для печатания книг. Все это можно приобрести в электронном виде через iTunes, Amazon, Google Play и другие виртуальные магазины. Тенденция, продиктованная нашим новым временем, очевидна — она заключается в устранении всех физических носителей прежде всего как средства для обмена и продажи информации.

Эта революция не обошла стороной и индустрию программного обеспечения. Как вы успели убедиться, с каждым днем нам все труднее встретить или купить программное обеспечение на компакт-дисках и DVD. Все функции поставщика информации приняли на себя компьютерные сети и Интернет. Более того, настольные и мобильные компьютеры с каждым днем все реже и реже оборудуются устройствами для чтения разного рода лазерных дисков. Наступившая вторая волна информационной революции сделала их неактуальными.

Смартфоны меняют все

Так что же послужило толчком к этой информационной революции? Основная причина тех изменений, которые мы сейчас активно переживаем и наблюдаем, связана с появлением миниатюрных устройств, которые именуются *гаджетами* или *смартфонами*.

По своей технической сути — это карманные компьютеры, и именно их появление во многом изменило нашу картину восприятия и поведения в современном мире. Теперь, находясь на улице или в общественном транспорте, нам не стоит удивляться тому, как много людей используют эти миниатюрные устройства: слушают музыку, тренинги и звуковые книги, смотрят фильмы и видеоролики, читают электронные книги, журналы, новости и блог-статьи в пространствах Интернета. Или же сидят в чате или просто убивают время, играя

в какую-нибудь игру. Да, чуть не забыл, эти люди, вдобавок ко всему перечисленному, еще могут с их помощью разговаривать по телефону, хотя это, наверное, и не самое главное, ради чего покупают смартфоны, — прежде всего они ценны тем, какое многообразие возможностей скрыто в этих замечательных устройствах.



Рис. 62.1. Различные модели смартфонов

При этом, чтобы такое многообразие стало возможным, необходимо было отказаться от некоторых технологий, не вписывающихся в современную концепцию. Представьте себе нелепую ситуацию, когда вам, для того чтобы послушать музыку, пришлось бы носить с собой вместе со смартфоном еще и портативное устройство для считывания CD-дисков — а ведь когда-то, и это уже история, плеер с кассетой или компакт-диском был вполне нормальным явлением. Вот так сравнительно недавно появившиеся смартфоны меняют наши привычки и наше взаимодействие с миром и значительно расширяют наши возможности. Благодаря простоте их использования, они внедряются практически во все возрастные категории (от младенцев до людей пенсионного возраста) и проникают во все сферы нашей деятельности (от работы до отдыха).

Неудивительно, что современное человечество стало зависимым от них, ведь все теперь подчиняется принципу «все свое ношу с собой». Действительно, если раньше у вас была необходимость для ориентации на местности брать с собой карту. Или же, когда надо было снять утренник своего ребенка в садике, вы брали фотокамеру, а когда на улице темнело, по привычке доставали из кармана фонарик. Сегодня же, взяв с собой смартфон, вы можете быть спокойны — все упомянутое упаковано в одном небольшом агрегате, который у вас всегда с собой. В нем есть все необходимые вам функции, кроме, пожалуй, открывалки для пива, хотя думаю, что это явление тоже временное.

Так как же все началось? Появление первого в мире смартфона iPhone было анонсировано 9 января 2007 года Стивом Джобсом (в то время руководившим корпорацией Apple). Apple бросила все силы на осуществление этого масштабного проекта, и вот, спустя чуть менее полугода, 29 июля 2007 года iPhone поступил в продажу. Это, бесспорно, был грандиозный успех, и знаменовал он собой начало новой эры в технике и информации.

Но в этой бочке меда была и ложка дегтя — первые смартфоны имели весьма ограниченный набор приложений. И, естественно, этот набор не мог удовлетворить запросы пользовате-

лей, а тем более раскрыть потенциал такого великого творения, как iPhone, на борту которого не было даже механизма для установки сторонних приложений. Как следствие, разработчики-энтузиасты пытались создавать свои собственные дополнительные приложения и с помощью различных ухищрений устанавливать их на iPhone. Понятно, что так долго продолжаться не могло. И вот в августе 2008 года на iPhone появилось приложение виртуального магазина App Store. Это был самый значительный революционный момент для нас, разработчиков, ну и, конечно же, прежде всего для пользователей.

Свершилось! Наконец-то открылся рынок, на котором столкнулись вместе и закрутились в бурном «водовороте» спрос и предложение, и который и по сей день раскручивается все сильнее и сильнее и не собирается сбавлять свои обороты.

В том же 2008 году, а именно 23 сентября, произошло еще одно событие. На свет появился Android — еще одна платформа для смартфонов, которая впоследствии тоже предоставила похожую модель магазина приложений. Эта платформа также стала стремительно набирать своих пользователей, и на сегодняшний день является основным конкурентом платформы iOS от Apple.

Дальнейшее появление новых устройств — таких как планшеты, умные часы и умные телевизионные приставки — еще больше усилило и расширило рынок приложений.

Виртуальные магазины приложений

К настоящему моменту число только пользователей смартфонов уже перевалило за 2,5 миллиарда. Это колоссальный рынок! И если у вас есть идеи, то вам, как разработчику, сейчас даются уникальные шансы воплощать их в жизнь. Для этого нужно только получить возможность распространять свои творения через виртуальный магазин приложений. Клуб разработчиков мобильных приложений — это не эксклюзивный клуб. Приобрести лицензию может практически каждый. И так как эта часть книги и посвящена разработке для мобильных платформ iOS и Android, остается лишь перечислить самые популярные виртуальные магазины, которые доступны для этих платформ:

- App Store — магазин мобильных приложений для платформы iOS от корпорации Apple. Он является единственным для этой платформы;
- Google Play — магазин мобильных приложений для платформы Android от корпорации Google. Имеется на большинстве смартфонов и планшетов с этой операционной системой;
- Amazon Appstore — магазин мобильных приложений от корпорации Amazon. Доступен в основном на планшетах Kindle Fire и смартфонах Fire Phone. Допускается также установка на другой смартфон и планшет вручную;
- Яндекс.Store — магазин мобильных приложений для платформы Android от российской интернет-компании «Яндекс». В основном устанавливается вручную.

Интересный аспект — это стоимость лицензий для разработчиков. Немного проясним ситуацию на сегодняшний день. В Apple ее стоимость составляет $99 в год, а на Google Play плата взимается только лишь разово — $25. За лицензию Amazon Appstore и Яндекс.Store вообще ничего платить не надо.

Очень большое преимущество магазинов приложений — это, конечно же, значительное количество потенциальных клиентов. Но это еще не все. Если вы когда-либо разрабатывали платные приложения для настольных компьютеров, то наверняка сталкивались с целым рядом задач, не связанных с концепцией самого приложения, — таких как, например, защи-

та от взлома, привязывание систем оплаты и др. Это всегда отнимало у вас массу сил и времени. Магазины приложений берут большую часть из этих задач на себя и тем самым позволяют вам концентрироваться на самом продукте, а не над инфраструктурой сбыта. Вот эти преимущества:

- не нужно создавать или использовать отдельную программу установки приложения на устройство;
- не нужно реализовывать механизмы от несанкционированного копирования приложения;
- не нужен отдельный веб-ресурс для загрузки приложений;
- не нужно привязывать системы оплаты;
- не нужно реализовывать отдельный механизм для обновления приложений;
- не нужно заботиться о возвратах денег клиентам — это задача сервиса работы с клиентами магазина.

Для пользователей магазины приложений тоже имеют целый ряд привлекательных сторон. К ним можно отнести:

- большой выбор приложений;
- простоту покупки и загрузки — одним нажатием на кнопку;
- быстрый поиск нужных приложений;
- наличие отзывов пользователей, которые помогают принять объективное решение о покупке или загрузке приложений;
- автоматическое обновление приложений;
- относительно низкая цена ввиду большой конкуренции;
- доверие покупателя к приложениям. Прежде чем выйти в магазин, все приложения проверяются на соответствие требованиям магазина.

На рис. 62.2 показано, как выглядят страницы приложений различных виртуальных магазинов. Мобильные приложения представлены в виртуальных магазинах обычно следующими составляющими:

- *название* — обычно в нескольких словах отражает назначение приложения;
- *значок* — визуальный образ, служащий для узнаваемости приложения;
- *цена* — всегда показана рядом с названием и значком приложения. Приложение может быть платным, бесплатным и иметь внутренние продажи (In-App Purchases);
- *оценки* — показаны в виде звездочек, которые находятся рядом со значком и названием приложения. Это средняя оценка от нескольких до большого числа пользователей. Чем больше звездочек, тем, по мнению пользователей, приложение лучше;
- *описание* — описательный текст, знакомящий с привлекательными особенностями и функциями приложения. Может достигать до 4 тыс. знаков;
- *слайды* — иллюстрации, демонстрирующие привлекательные стороны и возможности приложения. В большинстве случаев их количество ограничено пятью;
- *видео* — в отличие от указанных только что элементов не является обязательным, но если приложение хорошее, и есть что показать, то оно может значительно повысить его загрузки. В App Store видеоролик всегда расположен на первой позиции перед слайдами, в Google Play — на самом верху. В Amazon Appstore видеоролик располагается сразу за значком приложения;



Рис. 62.2. Одно и то же приложение, представленное в четырех различных магазинах (*слева направо*): App Store, Google Play, Amazon Appstore и Яндекс.Store

- *отзывы* — тут пользователи приложений делятся своим мнением и впечатлениями от использования приложений. Пользователи их читают перед тем, как принять решение о покупке или загрузке приложения. С недавнего времени App Store предоставил разработчикам возможность отвечать на отзывы пользователей. Пользуйтесь этим, чтобы не оставлять пользователей, ищущих ответы на свои вопросы по приложению, в информационном вакууме. Но держите марку и будьте деликатны и вежливы.

Распространение приложений вне виртуального магазина

Не всегда и не во всех случаях распространение через виртуальные магазины может оказаться желательным или возможным. Как же реализовывать приложения в обход виртуальных магазинов? И такое желание тоже вполне разрешимо!

Например, Apple предоставляет особый вид корпоративной лицензии Apple Developer Enterprise Program, стоимость которой сейчас составляет $299. Этот вид лицензии позволяет распространять разработанные приложения среди сотрудников внутри своей организации.

С Android дела с независимым от виртуального магазина распространением приложений обстоят и того лучше — нужно только лишь создать отдельный файл архива *.apk и сделать его доступным для скачивания. При умелом подходе такое решение может сослужить хорошую службу в качестве дополнительной возможности распространения ваших Android-приложений.

Qt и разработка мобильных приложений

Очень наболевший вопрос, мучающий и занимающий сознание многих не знакомых с Qt разработчиков, которые желают сделать свое первое мобильное приложение, заключается в том, какую из мобильных платформ выбрать. Ведь все они такие разные, и средства для

разработки тоже очень сильно отличаются друг от друга. На их изучение необходимо затратить массу времени. Например, для создания приложений для Android обычно используется язык Java. Для iOS — это может быть Objective C или Swift. Учить одновременно сразу два языка, а также связанные с ними фреймворки, и программировать на них, да еще и отлаживать код, для одного человека может быть просто неподъемной задачей. А если у вас появится желание увидеть свои приложения на Windows Phone, то можете добавить в свой учебный план C# и умножить время для разработки и отладки на три. Да, да, теперь вам потребуется в три раза больше времени. С каждой добавленной платформой разработки ваша производительность катастрофически падает. Поэтому разработчики-одиночки концентрируются обычно на одной.

Учитывая, что Qt поддерживает три популярные мобильные и три популярные настольные операционные системы, то, имея их в своем арсенале и умело пользуясь Qt, вы можете смело конкурировать с целым отделом разработчиков, состоящим из шести человек, которые разрабатывают свои программы отдельно для каждой платформы. Вас, как Qt-разработчика, никогда не должен интересовать вопрос: какую из платформ выбрать, потому что решение по отбору сводится к знанию некоторых специфических нюансов для целевой платформы и перекомпиляции программы. У вас есть возможность выбирать и использовать все самые лучшие средства для разработки, отладки кода и других средств для улучшения кода программы. Например, если вы не нашли хорошей программы для поиска утечки памяти на Windows, то можете воспользоваться программой Valgrind на Linux или использовать замечательную коллекцию для отладки, входящую в пакет Xcode на Mac OS X.

Разрабатывать пользовательский интерфейс мобильных приложений с Qt можно как при помощи QML (см. *часть VIII* книги), так и с помощью виджетов (см. *часть II*). Использование QML более предпочтительно, так как он специально создавался с расчетом разработки для мобильных устройств. Но это не значит, что если у вас есть проекты для настольных компьютеров, разработанные на виджетах, и ваши клиенты или вы сами вдруг захотите иметь их также и на мобильных устройствах, то вам срочно придется переписывать код пользовательского интерфейса на QML. Гораздо быстрее и проще будет откомпилировать приложение на мобильное устройство, запустить его, испробовать в действии, а уж потом продуманно и взвешенно принимать решение.

Конечно же, это редкий случай, когда настольное приложение будет работать на мобильном устройстве удовлетворительным образом. Это связано с тем, что способы взаимодействия пользователя с мобильными устройствами и настольными компьютерами очень сильно отличаются друг от друга, а также с небольшими размерами экрана некоторых из них. Чем больше разнообразие элементов управления в приложении, тем больше работы по адаптации нужно будет проделать. Ввиду того, что планшеты не обладают недостатком, связанным с маленькими размерами экрана, то адаптацию настольных проектов легче всего будет сделать для них. И если вдруг процесс адаптации на смартфон окажется очень трудоемким или невозможным, то вы при желании все-таки сможете выложить свое приложение в магазине App Store, — но только для iPad, а не для iPhone. Другими словами, переход на популярный шаблон «Функциональная часть на C++, интерфейс пользователя на QML» является желательным, но не обязательным. Из личного опыта добавлю, что среди моих проектов есть также и приложения на виджетах, которые успешно прошли адаптацию на мобильные устройства и теперь прекрасно работают как на настольных компьютерах, так и на смартфонах, и на планшетах.

Теперь немного о лицензии Qt для мобильной разработки. Для экспериментов с разработкой приложений для iOS можно использовать бесплатную версию Qt под лицензией LGPL. Но как только вы соберетесь размещать приложение в магазине App Store, вам будет необ-

ходима коммерческая лицензия Qt. Это связано с тем, что, согласно одному из требований LGPL-лицензии, Qt должна использоваться в виде динамически скомпонованных библиотек, а это, в свою очередь, нарушает требования App Store. Поэтому придется использовать Qt со статически скомпонованными библиотеками, которые доступны только для обладателей коммерческой лицензии. Коммерческую лицензию Qt можно приобрести на веб-странице: **qt.io**.

Резюме

Смартфоны практически вошли во все сферы нашей жизни, и в настоящее время мы уже не представляем себе, как нам обходиться без них. С их распространением и растущими потребностями пользователей сформировалась новая индустрия мобильных приложений.

Разработчикам, использующим Qt, доступны четыре основных магазина для распространения своих приложений: App Store, Google Play, Amazon Appstore и Яндекс.Store. Приложения можно также распространять и вне магазина приложений — внутри собственной компании или используя собственные веб-ресурсы.

Приложения можно разрабатывать как с помощью виджетов, так и средствами Qt Quick.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/62-510/** или с помощью следующего QR-кода (рис. 62.3):



Рис. 62.3. QR-код для доступа на страницу комментариев к этой главе

ГЛАВА 63

Подготовка к работе над мобильными приложениями

> ... разработчики, мы думаем, что у нас есть для вас очень хорошая новость. Вы можете создавать свои приложения для iPhone уже сегодня...
>
> *Стив Джобс*

Для того чтобы начать создание мобильных приложений, потребуются идеи, подкрепленные мотивацией, знания, полученные в предыдущих главах книги, неистовое желание реализовать замыслы и, конечно же, верный друг компьютер. Каким будет сам компьютер — переносным или настольным — большой разницы нет, главное, чтобы он был оснащен Intel-совместимым процессором, имел достаточно свободного места на диске и не менее 8 Гбайт оперативной памяти. Лучше всего для этой цели подойдет Mac. Компьютеры от Apple идеально подходят для разработки платформонезависимых приложений с помощью библиотеки Qt — благодаря им можно покрыть практически все мобильные и настольные платформы. Кроме того, используя для разработки мобильных приложений любой другой компьютер, вы однозначно лишаетесь платформы iOS. Поэтому решайте сами.

Итак, с компьютером мы разобрались, теперь нужно подготовить его к мобильной разработке и проделать небольшую работу по настройки среды. Этим мы и займемся в данной главе.

Подготовка среды для разработки iOS-приложений

Для разработки iOS-приложений, как уже было сказано ранее, без Mac никак не обойтись. И если у вас его еще нет, то совсем необязательно бежать и приобретать самый дорогой, новый и мощный. Для наших задач вполне подойдет Mac Book Pro пятилетней давности с жестким диском 500 Гайт и оперативной памятью не менее 8 Гбайт. В качестве бюджетного варианта можно использовать также и сравнительно недорогой Mac Mini с похожими характеристиками.

Если вы еще не установили Xcode, Qt и Qt Creator, то вам нужно сначала проделать шаги, описанные в *приложении 1*, посвященном настройке среды для работы над Qt-приложениями. Обязательно обратите внимание на то, чтобы при установке Qt на Mac OS X была выбрана опция **iOS** (см. рис. П1.1 в *приложении 1*).

Теперь давайте установим симуляторы — они вам понадобятся для запуска приложений в том случае, если у вас нет iPhone или iPad. Запустите Xcode и выберите пункт меню **Xcode**, затем в открывшемся подменю выберите пункт **Preferences** и в открывшемся окне —

вкладку **Components**. В результате вы увидите окно настроек (рис. 63.1). В этом окне показан большой список симуляторов, и каждый из них занимает не менее 1 Гбайт, поэтому не имеет смысла ставить их все. Установим два — и их нам вполне хватит. Это будут симулятор самой новой на данный момент версии 11.1 и симулятор версии 9.3. Для нас важно убедиться в работоспособности приложений прежде всего на этих версиях iOS. Про новую версию и так все понятно, а версия 9.3 нам нужна потому, что это последняя версия, до которой поддерживаются мобильные устройства iPhone 4s, iPad 2 и iPad mini 1.

> ***Старые устройства хороши для тестов***
>
> Рекомендую также подумать о покупке одного из этих старых устройств — чтобы убедиться в правильной и быстрой работе на них ваших приложений. Это поможет принять решение, стоит ли что-то изменить в приложении, оптимизировать его или вообще не выпускать его для таких устройств.

Если же появится необходимость в других версиях симуляторов, то мы их всегда можем установить из окна **Preferences** и позже.

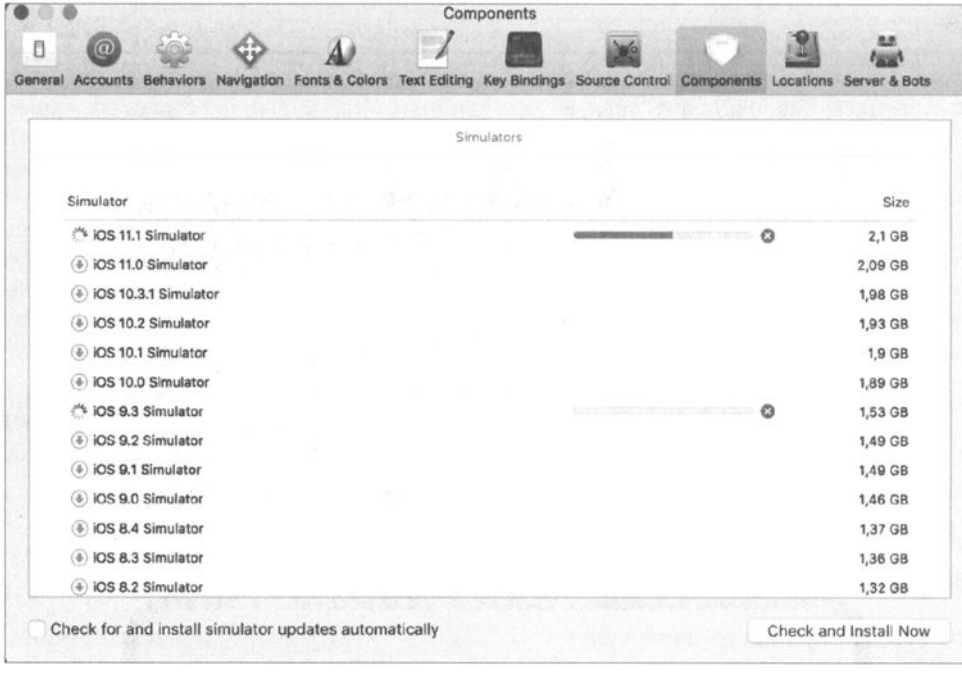


Рис. 63.1. Установка симуляторов iOS

Теперь проверим работоспособность установленных нами симуляторов на практике, для чего запустим Qt Creator и создадим в нем новый проект. Вы можете воспользоваться шагами создания проекта, приведенными в *главе 47*. При создании проекта на шаге **Kit Selection** обязательно обратите внимание, чтобы была активирована опция **Qt 5.10.0 for iOS Simulator** (рис. 63.2).

Теперь осталось только откомпилировать наше приложение и запустить его на симуляторе. Для компиляции и запуска выбираем на симуляторе опцию **Qt 5.10.0 for iOS Simulator** (рис. 63.3), и после нажатия на кнопку запуска видим наше приложение на симуляторе iPhone X (рис. 63.4). Повернуть дисплей симулятора можно комбинацией клавиш <Command>+<→>.

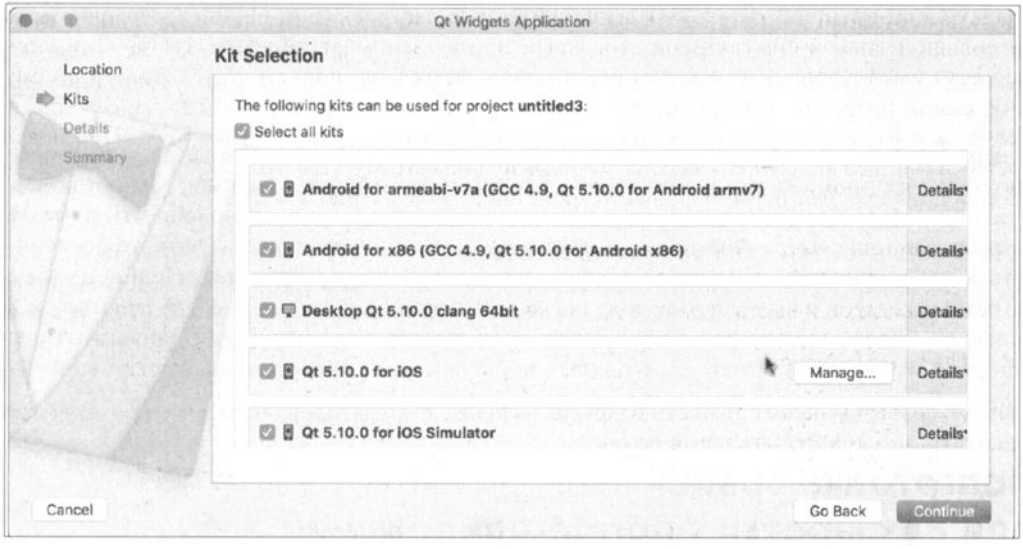


Рис. 63.2. Создание нового проекта: шаг **Kit Selection**

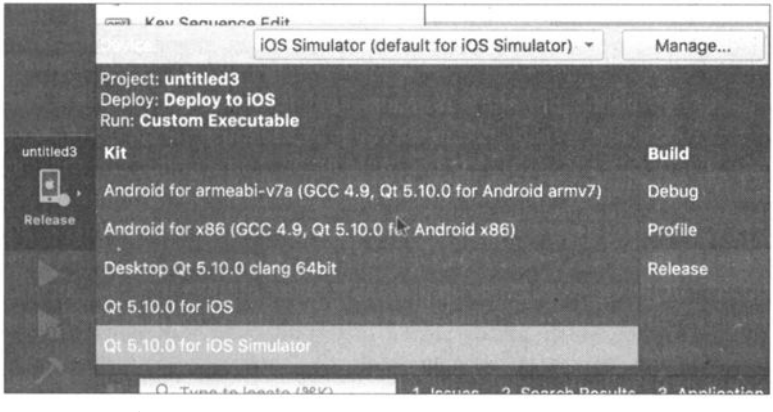


Рис. 63.3. Опция для компиляции и запуска приложения на симуляторе iOS

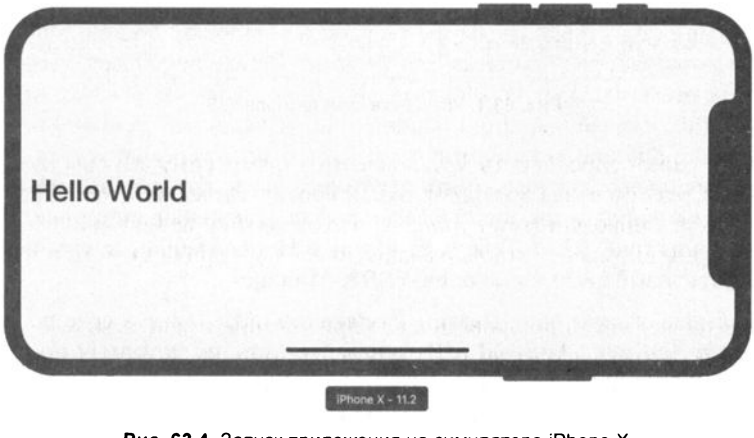


Рис. 63.4. Запуск приложения на симуляторе iPhone X

Удалять ненужные программы с симулятора можно точно так же, как и на iPhone: нужно нажать на значок приложения, подождать, пока появится крестик в левом верхнем углу, после чего нажать на него. Существует еще целый ряд действий, которые можно проводить с симулятором, — их можно найти в его меню.

Теперь было бы, конечно, здорово запустить приложение на реальном устройстве с операционной системой iOS, но, к сожалению, для этого необходима лицензия. Мы обязательно позже (в *главе 66*) рассмотрим, как создать свою учетную запись и приобрести лицензию разработчика для Apple App Store. А пока в качестве альтернативы вы можете воспользоваться преимуществом платформонезависимого разработчика и запускать свои приложения на реальных мобильных устройствах, но на платформе Android — там это работает и без лицензии. О том, как настроить среду разработки программ для Android и запускать приложения на эмуляторе Android и на реальных мобильных устройствах, рассказано в этой главе дальше.

Подготовка среды для разработки Android-приложений

В отличие от создания iOS-приложений, возможности создания Android-приложений не ограничиваются только лишь компьютерами Mac. Будем считать, что вы уже установили Qt на свой компьютер и настроили его для работы с одной из операционных систем: Mac OS X, Windows или Linux. Если нет, то самое время это сделать — воспользуйтесь для этого описанием, приведенным в *приложении 1*, посвященном настройке среды для работы над Qt-приложениями. Обязательно обратите внимание на то, чтобы при установке Qt на Mac OS X, Windows и Linux были выбраны опции **Android ARMv7** и **Android x86** (см. рис. П1.1 в *приложении 1*).

Теперь займемся настройкой среды для создания Android-приложений. Прежде всего, вам следует установить JDK (Java Development Kit, комплект разработчика Java). Для этого перейдите на страницу: **http://www.oracle.com/technetwork/java/javase/downloads/** и найдите там версию JDK 8. В настоящее время самой новой версией JDK является версия 9, но ни в коем случае не загружайте и не устанавливайте ее, потому что Android Studio пока еще с ней не работает. Вместо этого прокрутите страницу немного ниже, найдите версию JDK 8 и загрузите ее для нужной вам операционной системы. Затем запустите и установите ее на свой компьютер, следуя указаниям инсталляционной программы (рис. 63.5).

Не следующем шаге подготовки среды мы скачаем и установим Android Studio — это интегрированная среда разработки программ для Android. Из нее нам будет очень удобно загрузить и настроить все остальные компоненты. Итак, перейдем по ссылке: **https://developer.android.com/studio/**, скачаем Android Studio и произведем ее установку на компьютер. Теперь когда Android Studio установлена на вашем компьютере, запустите ее. При первом запуске программа предложит выполнить некоторые шаги, которые вы можете оставить по умолчанию без изменений, и так пройти весь процесс начальной настройки, пока не увидите окно приглашения (рис. 63.6). Теперь разверните расположенное в этом окне внизу справа меню **Configure** и выберите в нем опцию **SDK Manager**.

В открывшемся диалоговом окне **Default Preferences** перейдите в секцию **Appearance & Behavior | System Settings | Android SDK** и задайте путь, по которому программа Android Studio должна будет сохранить компоненты Android SDK (Android Software Development Kit). Это может быть и локальный диск компьютера — самое главное, чтобы на нем было не менее 40 Гбайт свободного места. Если на вашем диске столько места нет, то нужно либо

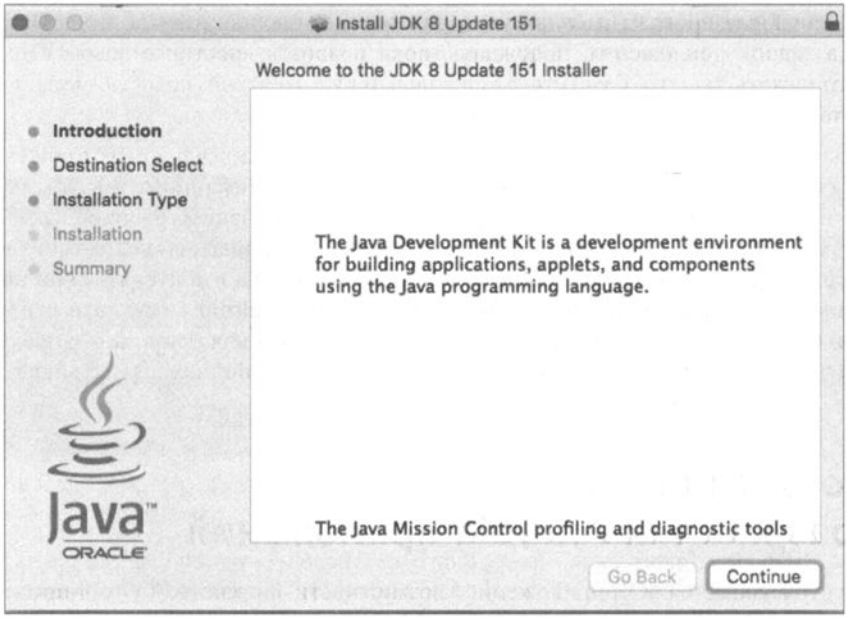


Рис. 63.5. Окно программы установки JDK 8

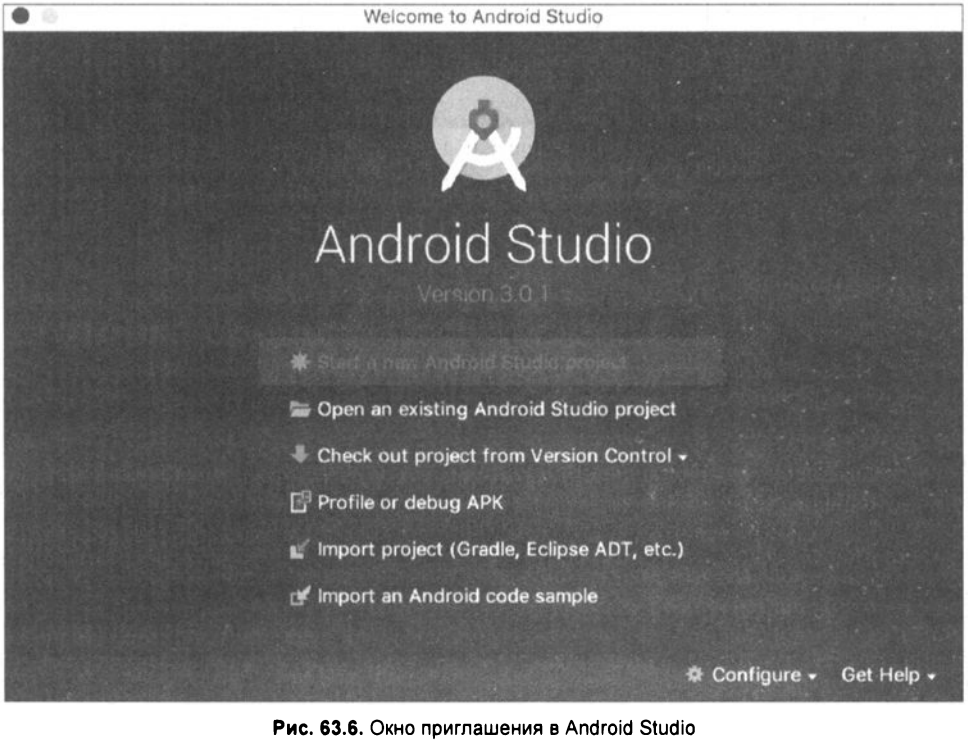


Рис. 63.6. Окно приглашения в Android Studio

освободить требуемое место, либо разместить SDK на отдельном накопителе. Путь для размещения компонентов Android SDK надо вписать в поле ввода **Android SDK Location** (рис. 63.7).

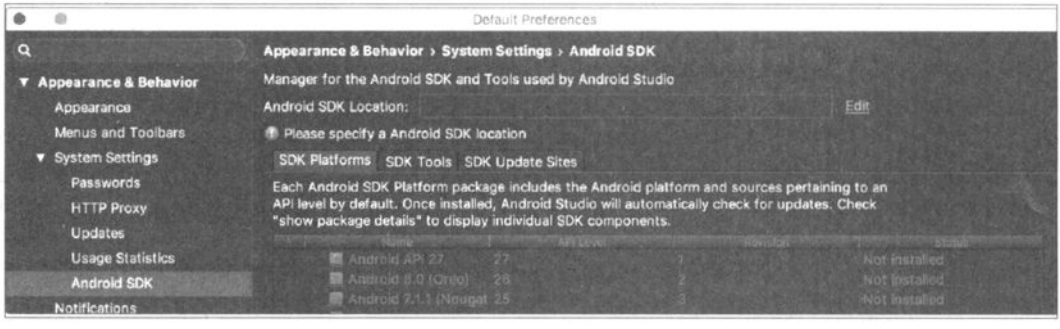


Рис. 63.7. Окно для конфигурации Android SDK

На вкладке **SDK Platforms** окна, показанного на рис. 63.8, мы видим элементы, упорядоченные по убыванию в графе **API Level** (API Уровень). Уровень API (Application Programming Interface, интерфейс прикладного программирования) задает номер интерфейса библиотеки Android SDK, который будут использовать ваши приложения. Нажмите здесь на кнопку флажка **Show Package Default** (она находится внизу окна справа) и выделите в уровнях API 16 и 18 все компоненты. Уровень API 16 — это минимально возможный уро-

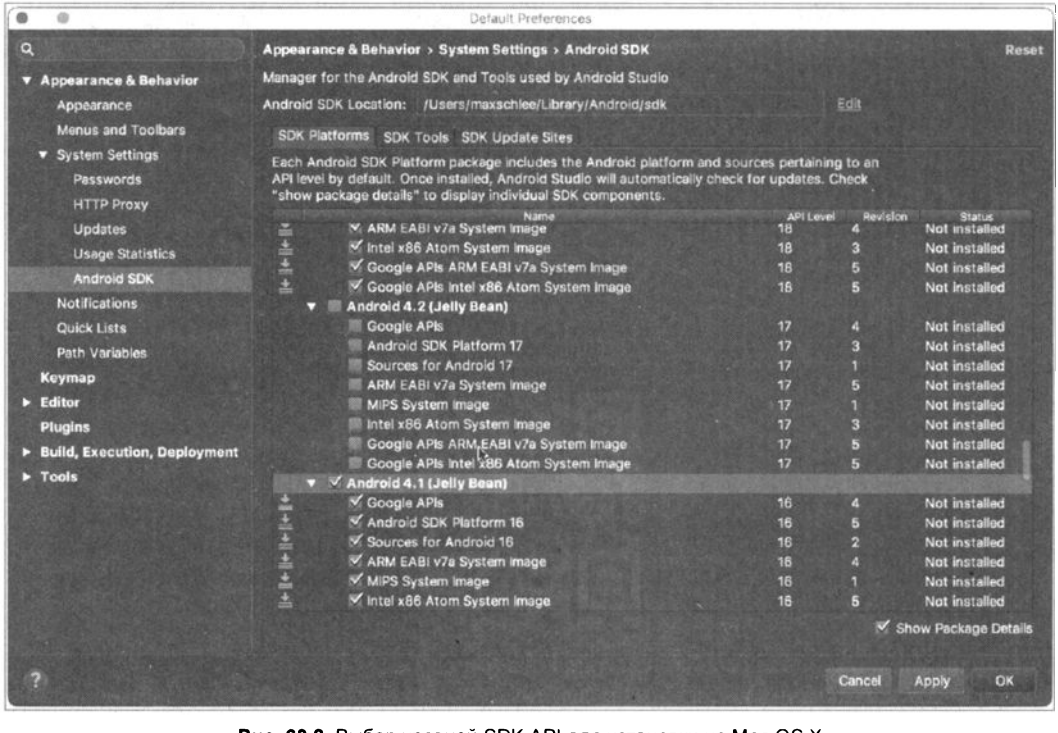


Рис. 63.8. Выбор уровней SDK API для установки на Mac OS X

вень для Qt 5.10. А уровень API 18 мы выделяем потому, что Qt использует его для поддержки устройств Bluetooth, которые нам могут в будущем пригодиться. Этих двух уровней нам будет достаточно.

Затем перейдите на вкладку **SDK Tools** (рис. 63.9) и выделите все имеющиеся на ней опции списка. Обязательно проконтролируйте, чтобы была выделена опция **NDK** (Native Development Kit) — она необходима для того, чтобы Qt-программы можно было компилировать для Android. Проконтролируйте также выделение опции **Google USB Driver** — она устанавливает в Windows драйвер USB, необходимый для запуска приложений на мобильных устройствах. На Mac OS X и Linux надобности в установке USB-драйвера нет, поэтому для этих операционных систем такая опция отсутствует.

Осталось нажать кнопку **OK** и запустить тем самым процесс закачивания и установки выбранных компонентов. Когда он закончится, нажмите кнопку **Finish** для завершения настройки Android Studio.

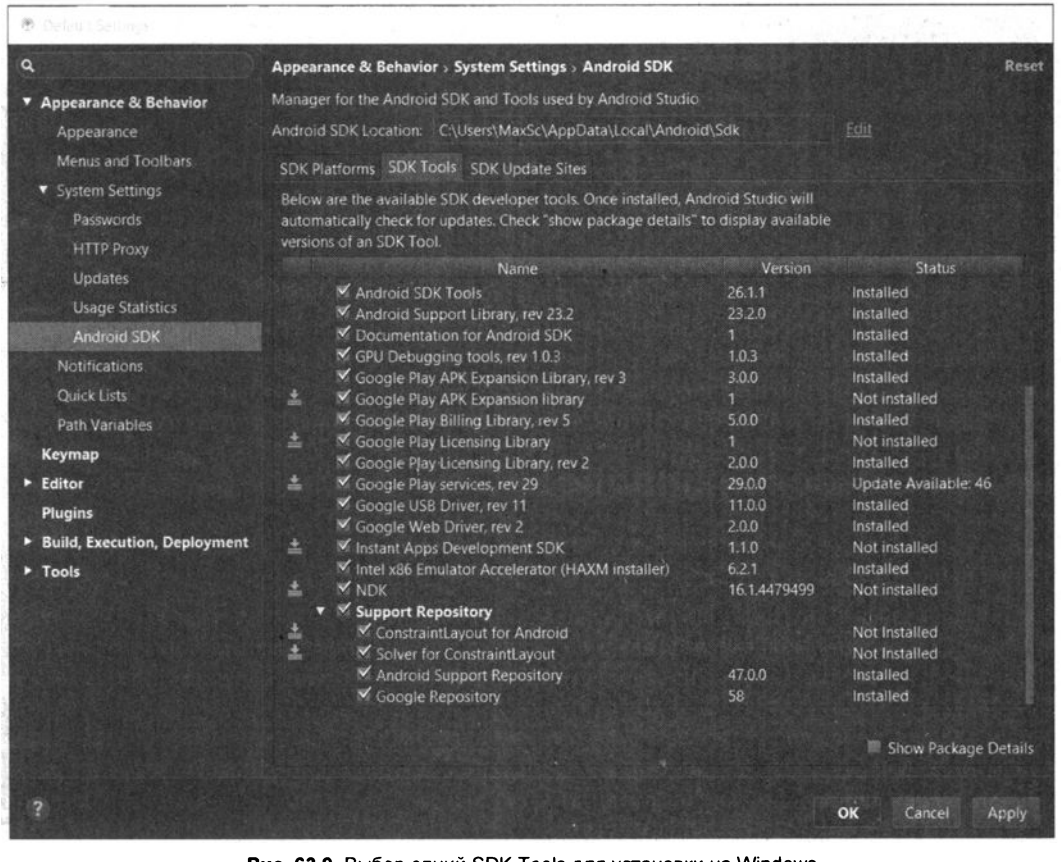


Рис. 63.9. Выбор опций SDK Tools для установки на Windows

Теперь запускаем Qt Creator, выбираем меню **Preferences**, в открывшемся одноименном диалоговом окне выбираем раздел **Devices** (рис. 63.10) и вводим в соответствующие поля пути к установленным нами инструментам и библиотекам JDK, Android SDK и Android NDK. Обратите внимание, что каталог NDK расположен внутри каталога SDK в каталоге ndk-bundle.

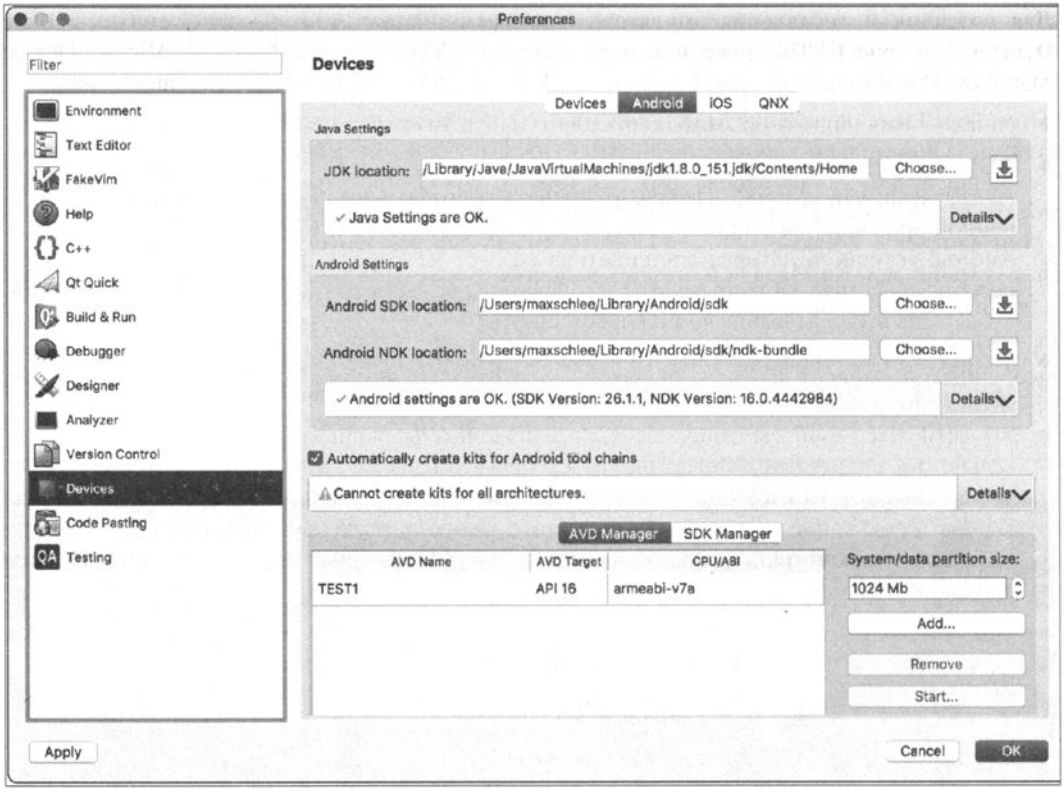


Рис. 63.10. Qt Creator: раздел **Devices**

Теперь нам нужно создать хотя бы один эмулятор/симулятор. Чем он может быть полезен?

- Во-первых, он может избавить вас от бремени иметь при себе мобильные устройства и позволит запускать программы прямо на компьютере под управлением виртуальной машины.
- Во-вторых, вы можете создать много эмуляторов/симуляторов с различными уровнями API, чтобы тестировать приложения на этих уровнях. С реальными устройствами это была бы более труднодостижимая задача.
- В-третьих, вы можете создать эмуляторы/симуляторы с различными разрешениями экранов. Это необходимо, чтобы узнать, как будет себя вести графический интерфейс ваших приложений на различных экранах.
- В-четвертых, вы можете создавать виртуальные устройства с разными архитектурами: armeabi-v7a или x86.

> ***Обязательно используйте для тестов реальные устройства***
>
> Как бы ни было удобно использование эмулятора, не следует забывать, что запуск приложений на нем это не то же самое, что запуск приложений на реальном устройстве. Поэтому перед выпуском приложений в магазин обязательно проверяйте их на реальных устройствах. И чем большее количество реальных устройств при этом будет задействовано, тем лучше. О том, как запустить приложение на реальном устройстве, будет рассмотрено в этой главе позже.

Для создания и добавления эмулятора нажимаем кнопку **Add**, расположенную в окне **Devices** (см. рис. 63.10) ниже и правее вкладки **AVD Manager** (Android Virtual Device Manager). В открывшемся окне **Create new AVD** (рис. 63.11) заполняем следующие поля:

- **Name** — имя, под которым мы сможем отличить эмуляторы/симуляторы друг от друга;
- **ABI** — архитектура процессора. Обычно нам придется выбирать одну из двух альтернатив: armeabi-v7a или x86. Первая альтернатива создает эмулятор для самой распространенной архитектуры Android-устройств — armeabi-v7a. Ее используют более 90% всех Android-устройств. Вторая альтернатива создает не эмулятор, а симулятор, когда процессор задействуется напрямую, и, следовательно, скорость исполнения программ в x86 будет в десятки раз выше, чем в первом случае;
- **Target API** — уровень API-устройства, в нашем случае мы можем выбирать между 16-м и 18-м уровнями, т. к. это те уровни, которые мы установили на компьютер;
- **SD card size** — объем дополнительного накопителя данных. Его указывать не обязательно, но желательно. Может оказаться так, что вашему приложению понадобится сохранять данные на стороннем накопителе, или может появиться необходимость протестировать поведение приложения, после того как пользователь переместит его на дополнительный накопитель данных. Обычно 300 Мбайт для подобных экспериментов должно хватить.

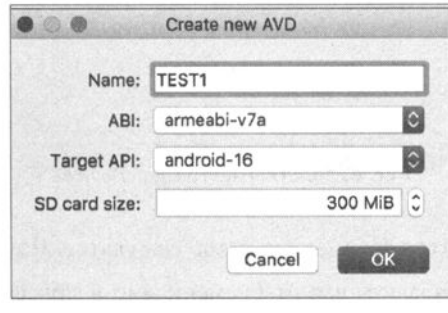


Рис. 63.11. Qt Creator: создание эмулятора/симулятора для Android

Итак, эмулятор для архитектуры armeabi-v7a мы создали, теперь проверим его в действии. Для этого создадим новый проект — вы можете воспользоваться шагами создания проекта, приведенными в *главе 47*. При создании проекта на шаге **Kit Selection** обязательно обратите внимание, чтобы была активирована опция **Android for armeabi-v7a** (см. рис. 63.2).

Теперь осталось только откомпилировать наше приложение и запустить его на эмуляторе — выбираем на эмуляторе опцию **Android for armeabi-v7a** (рис. 63.12) и нажимаем кнопку запуска. Нам будет предложено выбрать из списка только что созданный нами эмулятор — выбираем его, нажимаем на кнопку **OK** и немного погодя видим наше приложение (рис. 63.13).

Поворачивать дисплей эмулятора можно комбинациями клавиш: на Mac OS X — <Command>+ +<→>, а на Windows и Linux — <Ctrl>+<→>. Для того чтобы удалить приложение с эмулятора, нужно нажать на значок приложения и, не отпуская его, перетянуть в появившийся в левом углу пункт **Uninstall**. Есть в эмуляторе и интересная возможность симулировать жесты и множественное нажатие «мультитач». Для этого нужно на Mac OS X нажать клавишу <Command> и, не отпуская ее, нажать на тачпад. В Windows и Linux нужно нажать на клавишу <Ctrl> и, не отпуская ее, нажать левую кнопку мыши.

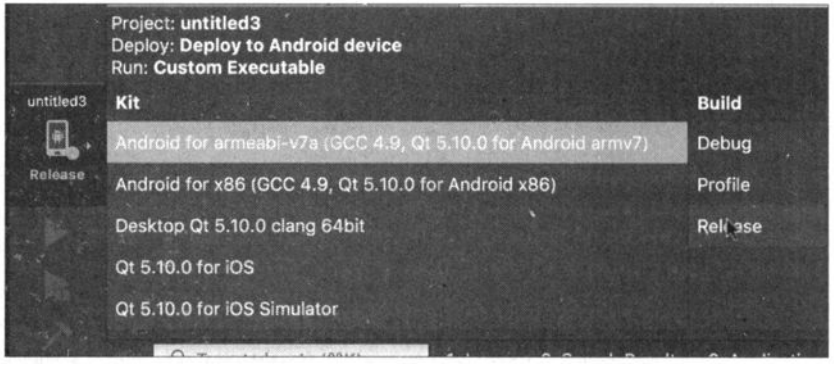


Рис. 63.12. Опция для компиляции и запуска на Android

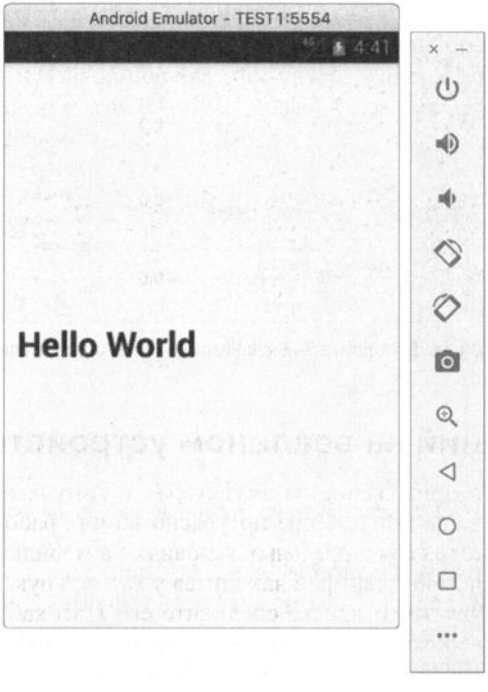


Рис. 63.13. Приложение, запущенное на Android-эмуляторе

Эмулятор предоставляет также дополнительные команды, которые расположены в виде кнопок на полоске, находящейся справа от него (см. рис. 63.13). С их помощью можно выполнить целый ряд действий, например: убавить/увеличить громкость звука, повернуть устройство в портретный или альбомный режим, сделать снимок с экрана и т. д. На этой полосе можно также найти и привычные для каждого Android-устройства три кнопки: **Назад** (треугольник), **Домой** (круг) и **Менеджер задач** (квадратик). Нажатие на самую нижнюю кнопку с тремя точками откроет окно с более «продвинутыми» настройками и действиями (рис. 63.14). В этом окне вы можете изменять показания акселерометра (см. *главу 64*), устанавливать текущее местонахождение в географическом пространстве, симулировать входящие телефонные звонки и СМС, изменять значения заряда аккумулятора и многое другое.

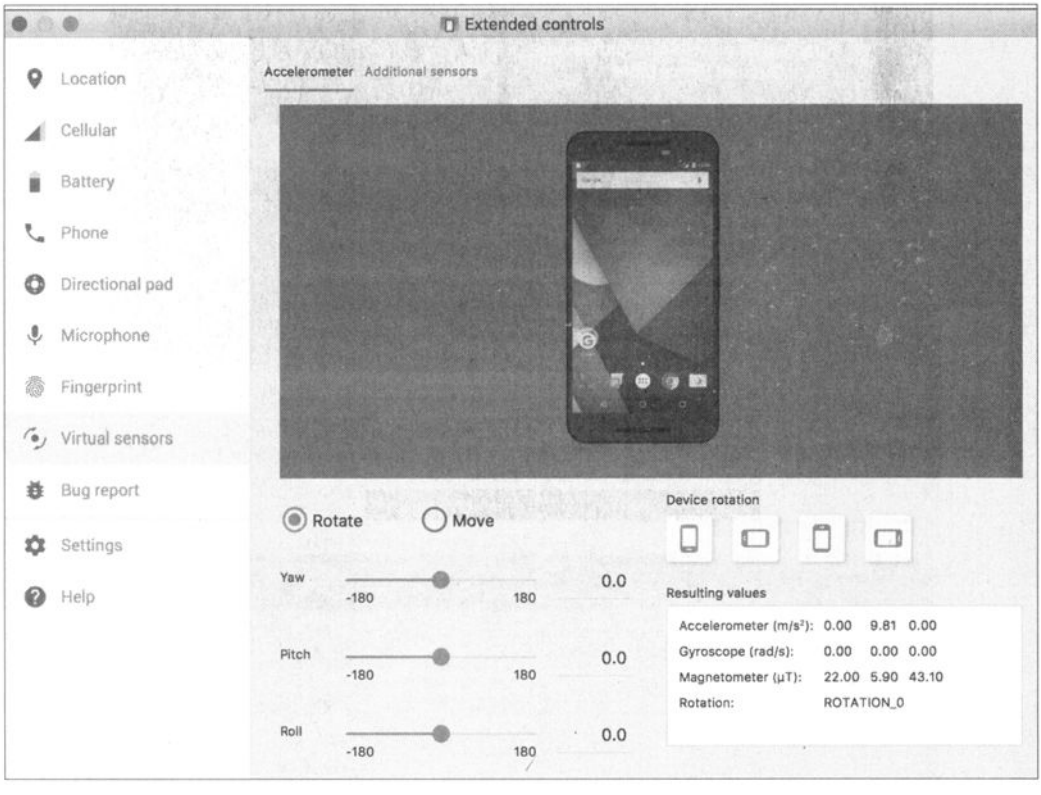


Рис. 63.14. Дополнительные возможности Android-эмулятора

Запуск приложений на реальном устройстве

До сих пор мы запускали приложение на эмуляторах и симуляторах. Но это не реальные устройства, и даже если ваше приложение прекрасно на них работало, вы не можете быть уверены, что оно будет вести себя в реальных условиях на мобильном устройстве точно так же. И теперь, если ваш Android-смартфон находится у вас под рукой, давайте запустим приложение прямо на нем. Для этого просто соедините его USB-кабелем с вашим компьютером. Сначала вам будет необходимо активировать на вашем мобильном устройстве режим отладки — в большинстве случаев для этого достаточно выйти в пункт **Настройки** (Settings), прокрутить в самый низ до пункта **Для разработчиков** (Developer options) и нажать на него, чтобы переключить опцию в самом верху в состояние **ВКЛ** (On) (рис. 63.15). Затем прокрутить до пункта **Отладка по USB** (USB Debugging) и нажать на него, чтобы активировать флажок.

Теперь нажмите в Qt Creator на кнопку запуска проекта, и вы увидите окно (рис. 63.16), в верхней строке которого указан смартфон, а в нижней — эмулятор. Смартфон выделен и, соответственно, является текущим элементом, поэтому нажимаем на кнопку **OK** и ждем запуска приложения на нем. Да, чтобы увидеть запущенное приложение, не забудьте разблокировать экран смартфона.

Если вы не можете найти в этом окне ваше мобильное устройство и пребываете в растерянности, вам придется поискать информацию на странице создателя своего устройства, чтобы узнать, каким образом на нем можно включить режим USB-отладки.



Рис. 63.15. Активация режима отладки по USB на реальном устройстве

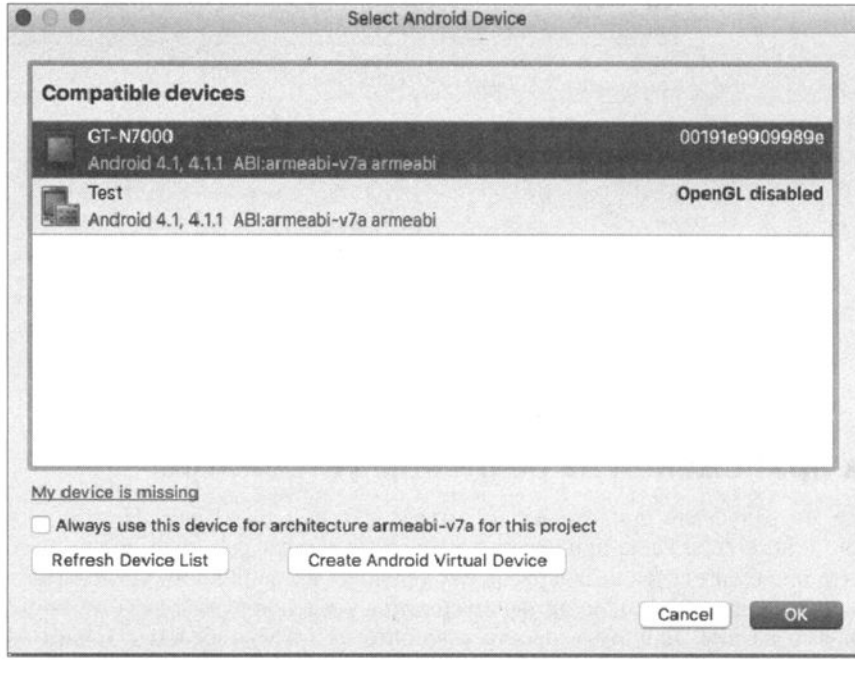


Рис. 63.16. Окно со списком устройств для запуска Android-приложения

Резюме

Мы определились с тем, какой компьютер нам понадобится для создания мобильных приложений.

Настроили среду разработки для iOS и Android. Запустили приложение на симуляторе и эмуляторе, а также рассмотрели их основные возможности.

Для настройки среды разработки Android-приложений нам понадобилось установить целый ряд инструментов и компонентов, таких как JDK, Android Studio, Android SDK и Android NDK.

В завершение мы запустили приложение на реальном мобильном устройстве с операционной системой Android.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/63-510/** или с помощью следующего QR-кода (рис. 63.17):



Рис. 63.17. QR-код для доступа на страницу комментариев к этой главе

ГЛАВА 64



Особенности разработки приложений для мобильных устройств

Мечты созданы для того, чтобы воплощать их в жизнь, а не для того, чтобы сходить по ним с ума.

Неизвестный

Подходы, которым разработчики следуют при создании приложений для настольных и переносных компьютеров, существенно отличаются от подходов, применимых к созданию приложений для мобильных устройств. Вот лишь некоторые из моментов, которые необходимо в большинстве случаев держать в своей голове при мобильной разработке:

- *мобильность* — пользователи мобильных устройств часто находятся в движении, смотрят по сторонам, что-то пытаются на ходу одновременно делать. Вывод из всего этого напрашивается сам собой — их внимание может быть рассеяно. А следовательно, и время использования ими приложений исчисляется короткими промежутками. Поэтому не усложняйте свои приложения большим набором функций и делайте их как можно проще. В идеале, все мобильное приложение должно концентрироваться на одной основной идее. Если вы хотите воплотить сразу несколько идей в одном приложении, то подумайте, не лучше ли для каждой из этих идей реализовать отдельное приложение. Также не принуждайте пользователей к каким-либо действиям, требующим от них дополнительных временных затрат, — это будет вызывать у них агрессию и раздражение;
- *устаревшие модели* — не забывайте, что в обиходе могут оставаться также и старые модели с маленькими сенсорными экранами, медленными процессорами и другими не сильно продвинутыми аппаратными характеристиками. Поэтому спрашивайте себя, до какой версии мобильной операционной системы должно будет запускаться ваше приложение, и обязательно протестируйте его хотя бы на одном из слабых устройств, которое поддерживает эту операционную систему;
- *внутренняя память* — не все смартфоны имеют большой объем внутренней карты, и ваше приложение может испытывать дефицит или отсутствие доступного объема памяти для данных. Поэтому, перед тем как записывать данные, ваше приложение — во избежание аппаратного отказа — должно убедиться в наличии достаточного места для них;
- *сеть* — пропускная способность сети, с которой взаимодействуют мобильные устройства, может быть небольшой и/или ограниченной в объеме трафика. Поэтому старайтесь использовать данные, получаемые по сети, настолько оптимально, насколько это возможно. Запрашивайте только те данные, которые действительно необходимы приложению. В случаях же, когда приложение намеревается получить по сети большой объем данных, то предварительно обязательно спросите у пользователя разрешение на прове-

дение этой операции. И не забывайте также и о том, что абоненты могут находиться вне зоны действия сети, т. е. связь может полностью отсутствовать;

- *аккумулятор* — один из самых страшных грехов, который может совершить разработчик, это написать приложение, после запуска неистово потребляющее заряд аккумулятора. Срок жизни такого приложения на мобильном устройстве будет непродолжителен и обычно ограничен всего лишь одним запуском. А после деинсталляции пользователи покарают его длинной лентой негативных отзывов в магазине приложений. Поэтому не нагружайте аппаратную часть мобильных устройств без особой на то надобности. Подумайте об эффективности использования приложением ресурсов и при необходимости оптимизируйте используемые приложением алгоритмы. Есть еще один момент, связанный с аккумулятором. Дело в том, что в некоторых случаях крайне необходимо отслеживать уровень его заряда, чтобы в критических ситуациях иметь возможность предпринять необходимые действия, например сохранить важные данные пользователя.

Кроме всего прочего, есть и другие особенности, такие как, например, взаимодействие пользователя с сенсорными экранами, переориентация экрана при повороте устройства в вертикальное или горизонтальное положение и т. п. Об этих и других особенностях приложений для мобильных устройств мы и поговорим в этой главе далее. А пока рассмотрим файлы свойств, которые играют очень важную роль для конфигурирования и предоставления информации о самих приложениях.

Анатомия файлов свойств для iOS- и Android-приложений

Существуют файлы с информацией, которые обязательно входят в поставку любого приложения для iOS или Android. Для iOS таким файлом является файл с расширением plist, а для Android — это файл AndroidManifest.xml. Для простых проектов библиотека Qt сама создает эти файлы по умолчанию. Но в большинстве случаев может потребоваться более тонкая настройка, для которой придется изменять и дополнять эти файлы. Поэтому необходимо немного разобраться в их структуре.

Файл свойств iOS-приложений

Расширение plist расшифровывается как Property List (список свойств) — соответственно, этот файл содержит описание свойств приложения. Все данные в нем представлены в XML-формате. В листинге 64.1 приведен пример такого файла.

Листинг 64.1. Пример файла IosInfo.plist для iOS

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
  <key>CFBundleDisplayName</key>
  <string>BassChordsCompass</string>
  <key>CFBundleExecutable</key>
  <string>BassChordsCompass</string>
  <key>CFBundleIcons</key>
```

```
  <dict>
      <key>CFBundlePrimaryIcon</key>
      <dict>
          <key>CFBundleIconFiles</key>
          <array>
              <string>Icon-29x29.png</string>
              <string>Icon-29x29@2x.png</string>
...
          </array>
      </dict>
  </dict>
  <key>CFBundleIcons~ipad</key>
  <dict>
      <key>CFBundlePrimaryIcon</key>
      <dict>
          <key>CFBundleIconFiles</key>
          <array>
              <string>AppIcon29x29.png</string>
              <string>AppIcon29x29@2x.png</string>
...
          </array>
      </dict>
  </dict>
  <key>CFBundleIdentifier</key>
  <string>com.neonway.BassChordsCompassIOS</string>
  <key>CFBundlePackageType</key>
  <string>APPL</string>
  <key>CFBundleShortVersionString</key>
  <string>1.2</string>
  <key>CFBundleVersion</key>
  <string>1.2.1</string>
  <key>LSRequiresIPhoneOS</key>
  <true/>
  <key>UISupportedInterfaceOrientations</key>
  <array>
    <string>UIInterfaceOrientationPortrait</string>
  </array>
  <key>UISupportedInterfaceOrientations~ipad</key>
  <array>
    <string>UIInterfaceOrientationPortrait</string>
  </array>
</dict>
</plist>
```

Давайте разберемся с некоторыми его моментами:

- `CFBundleDisplayName` — имя приложения, которое будет видно пользователю на iOS-устройстве;
- `CFBundleExecutable` — имя двоичного файла для запуска, т. е. то имя, которое указано в секции `TARGET` проектного файла Qt;

- `CFBundleIcons` и `CFBundleIcons~ipad` — списки значков для приложения iOS, расположенных в его ресурсах. Первый — для iPhone, второй — для iPad. Можно также оставить список значков пустым и вместо этого использовать в проектном файле специально созданный каталог с расширением **xcassets**, который будет содержать все необходимые значки и стартовые экраны для приложения. Этот способ мы рассмотрим в *главе 65*;
- `CFBundleIdentifier` — уникальная идентификационная строка приложения, которая создается на портале разработчика **www.developer.apple.com** для каждого из приложений только один раз и больше не подлежит изменению. Вносится она в следующем формате:

  ```
  <домен 1-го уровня>.<домен 2-го уровня>.<строка>
  ```

  Например, идентификационная строка, приведенная в листинге 64.1, выглядит так: `com.neonway.BassChordsCompassIOS`;
- `CFBundleShortVersionString` и `CFBundleVersion` — короткий и полный номера версии приложения. Например, короткий номер версии — 1.2, а полный — 1.2.1;
- `UISupportedInterfaceOrientations` и `UISupportedInterfaceOrientations~ipad` — различные ориентации экрана, поддерживаемые приложением. Первый — для iPhone, второй — для iPad. Более подробно о конфигурации ориентации экрана, а также об отслеживании и обработке событий смены ориентации экрана, читайте в *разд. «Автоматический поворот»* далее в этой главе.

Файл свойств Android-приложений

Свойства Android-приложений описываются в файле манифеста AndroidManifest.xml. Этот файл включает в себя имя приложения, права доступа к спорным ресурсам (permissions), таким как, например, запись звука, доступ к контактным данным и т. п., а также и другие необходимые свойства для конфигурации приложения. Все эти свойства мы рассмотрим далее. Для новых проектов файл манифеста создается автоматически в каталоге сборки приложения, но если вы хотите его изменить, то можно создать отдельный файл и включить его в проектный файл Qt следующим образом:

```
OTHER_FILES += _android/AndroidManifest.xml
```

Тогда этот файл станет доступным к редактированию прямо из Qt Creator. Для того чтобы его отредактировать, нужно сначала найти в проектном менеджере папку Other files, открыть ее, затем открыть папку _android — в ней и будет расположен файл AndoirdManifest.xml (рис. 64.1, поз. 1). Сделайте двойной щелчок мышью на этом файле, и он откроется для редактирования, как это показано в правой части рис. 64.1.

В этом режиме вы можете задать или изменить следующее:

- поз. 2: **Package name** — имя пакета вашего приложения в формате: `<домен 1 уровня>.<домен 2 уровня>.<имя приложения>`. Например, `com.neonway.BassChordsCompass`. Имя пакета приложения является одновременно и уникальным идентификатором вашего приложения в магазине Google Play. К странице приложения в магазине всегда можно будет получить доступ из веб-браузера по ссылке: **https://play.google.com/store/apps/details?id=<имя пакета>**;
- поз. 3 — доступны сразу два места для задания номера версии. Очень важно уяснить разницу между ними:
  - первым идет **Version code** — код версии. Это значение является внутренним, и оно нужно только для механизма управления версиями между вашим приложением и ма-

газином. Вы можете назначить ему любое значение — главное, чтобы оно было уникально для каждой из версий, и каждое последующее значение должно быть больше предыдущего. Вы можете сравнить его со счетчиком, по которому Google Play сличает уже имеющееся в магазине приложение с тем, которое вы намереваетесь отправить туда. Если значение меньше или равно уже имеющемуся в магазине приложению, то магазин исходит из того, что вы отсылаете либо старую, либо ту же самую версию приложения, и может отказаться принять ее. Если же значения счетчика новой версии больше, то тогда все в порядке;

- вторым идет **Version name** — имя версии. Это и есть тот самый номер, с которым мы ассоциируем номер версии приложения. Этот номер видит пользователь в магазине, этот номер обычно показывают и в самом приложении, например, в информационном диалоговом окне. Поэтому задавайте его в соответствии с текущей версией вашего приложения в формате: `<номер версии>.<номер уровня>.[номер обновлений];`

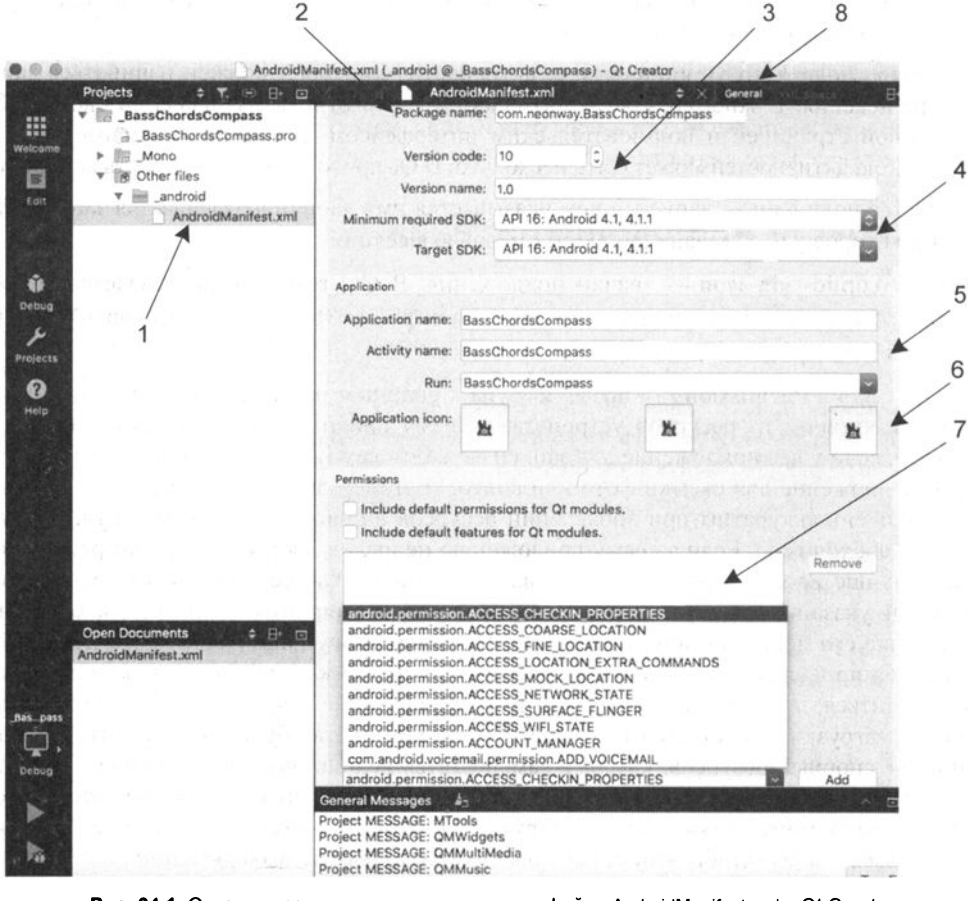


Рис. 64.1. Окно интерактивного редактирования файла AndroidManifest.xml в Qt Creator

♦ поз. 4 — два элемента выпадающих списков:

- первым идет **Minimum required SDK** — минимальный возможный SDK. В нем желательно выбирать API с самым низким номером, при котором ваше приложение способно правильно работать, — чем ниже номер уровня API, тем большее количе-

ство потенциальных клиентов вы способны охватить (см. рис. 64.2, где показано распределение рынка Android-устройств по версиям Android API). Это связано с *обратной совместимостью* системы, которая подразумевает, что приложение, предназначенное для более низкого уровня API, всегда может быть запущено на более высоком API. Например, приложение для API 14 может запускаться на мобильных устройствах с API 21. Для Qt 5.10 самый низкий уровень — это API 16. Обязательно тестируйте ваше приложение на самом низком уровне API, которое оно поддерживает. Это необходимо, чтобы убедиться в его работоспособности. Для этой цели используйте симулятор, на котором установлен этот уровень API;

- вторым идет **Target SDK** — целевой SDK, это обычно самый высокий уровень API, на котором вы проверили и можете гарантировать работоспособность своего приложения;

- поз. 5 — три поля ввода:
  - первое поле **Application name** — имя приложения, которое будет видно, после установки приложения на мобильном устройстве, под его значком;
  - второе поле **Activity name** — имя активности, с которой должен начинаться запуск приложения. В Android каждая активность — это отдельная задача со своей собственной страницей и пользовательским интерфейсом. В обычных приложениях для Android активностей может быть несколько. В Qt-приложениях активность всего одна;
  - третье поле **Run** — запуск, в нем указывается имя двоичного файла для запуска, т. е. то, которое у вас указано в проектном файле в секции `TARGET`;
- поз. 6: **Application icon** — значки приложения. В Qt Creator предоставляется возможность выбора для приложения трех значков: Low dpi (36×36 пикселов), Mid dpi (48×48 пикселов) и High dpi (72×72 пиксела);
- поз. 7: секция **Permissions** — права доступа к спорным ресурсам. В этой секции нужно указать, к каким из ресурсов устройства вашему приложению необходим доступ. Например, если у вас приложение для записи звука, то вам необходим доступ к микрофону, если приложение для съемки фото или видео, то нужен доступ к видеокамере и т. п. Но будьте очень аккуратны при добавлении ресурсов в список — указывать нужно только самое необходимое. Если вашему приложению не нужен доступ к спорным ресурсам, то тогда лучше вообще ничего не указывать. В Google Play сейчас с этим очень строго. Каждый указанный в приложении спорный ресурс придется сопроводить описанием о порядке его использования в документе «Privacy Policy» и поместить его на странице описания приложения в магазине Google Play — чтобы каждый пользователь мог с ним ознакомиться, до того, как он загрузит ваше приложение. Кроме того, после нажатия на кнопку загрузки приложения в Google Play на устройстве будет отображаться окно со списком спорных ресурсов, которые намеревается использовать приложение. Это может снизить количество загрузок вашего приложения, потому что, просмотрев этот список, пользователь может отказаться от загрузки. И хоть это возможно не всегда, но нужно стремиться к тому, чтобы этот список был пустым. Делайте выводы сами;
- поз. 8: как вы могли заметить из расширения файла Mainifest.xml — это файл в формате XML, а значит, его можно редактировать не только с помощью интерактивного редактора Qt Creator, но также и с помощью обычного текстового редактора. Тем не менее, программа Qt Creator предоставляет возможность редактирования файла AndroidMainfest.xml в текстовом режиме. Для этого переключите на верхней панели режим редактирования **General** в режим **XML Source**.

Version	Codename	API	Distribution
2.3.3 - 2.3.7	Gingerbread	10	0.4%
4.0.3 - 4.0.4	Ice Cream Sandwich	15	0.5%
4.1.x	Jelly Bean	16	2.0%
4.2.x		17	3.0%
4.3		18	0.9%
4.4	KitKat	19	13.4%
5.0	Lollipop	21	6.1%
5.1		22	20.2%
6.0	Marshmallow	23	29.7%
7.0	Nougat	24	19.3%
7.1		25	4.0%
8.0	Oreo	26	0.5%

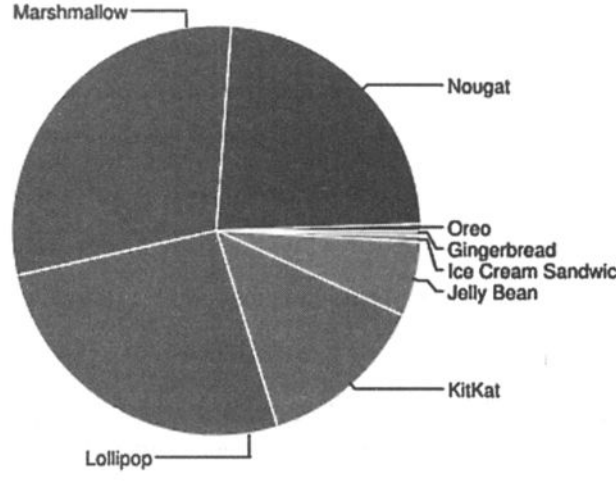


Рис. 64.2. Распределение рынка Android-устройств по версиям Android API по состоянию на декабрь 2017 года (источник: **https://developer.android.com/about/dashboards/index.html**)

В листинге 64.2 показано содержимое файла AndroidManifest.xml. Основу его структуры формируют теги `manifest`, `application` и `activity`. В первом теге вы видите атрибуты: `package`, `versionName`, `versionCode` — для задания соответственно имени пакета приложения, имени версии и кода версии, которые мы уже рассмотрели ранее. Тег `application` определяют атрибуты для задания имени приложения и его значка. В этот тег вложен тег `activity`, который предоставляет дополнительные возможности, обеспечиваемые приложением, — например, какую из ориентаций экрана (атрибут `screenOrientation`) поддерживает приложение (эта опция недоступна из интерактивного режима Qt Creator). В нашем примере установлено значение `portrait`, а это значит, что приложение поддерживает только портретную ориентацию экрана. Внутри тега `activity` расположен тег `intent-filter`. В нем размещенны еще два тега: `action` и `category`. Значение `android.intent.action.MAIN` атрибута `name` в первом теге говорит о том, что это активность, с которой должен начинаться запуск приложения. Значение `android.intent.category.LAUNCHER` атрибута `name` во втором теге говорит о том, что приложение может быть запущено с устройства нажатием на значок. Файл AndroidManifest.xml содержит также тег `uses-sdk` — для указания минимально возможной и целевой SDK, о чем мы уже говорили ранее. Тег `supports-screens` служит для указания поддерживаемых размеров дисплеев (эта опция тоже недоступна из интерактивного режима Qt Creator). В примере, приведенном в листинге 64.2, мы делаем доступными все экраны смартфонов и планшетов. Если ваше приложение не рассчитано на поддержку каких-либо из них, то нужно удалить из этого тега соответствующие атрибуты, либо присвоить им значение `false`.

Листинг 64.2. Пример файла манифеста для Android

```
<?xml version='1.0' encoding='utf-8'?>
<manifest
  xmlns:android="http://schemas.android.com/apk/res/android"
  package="com.neonway.BassChordsCompass"
```

```
  android:versionName="1.0"
  android:versionCode="10"
  android:installLocation="auto">
  <application
    android:name= "org.qtproject.qt5.android.bindings.QtApplication"
    android:label="BassChordsCompass"
    android:icon="@drawable/icon">
    <activity
      android:configChanges= "orientation|uiMode|screenLayout|screenS...
      android:name="org.qtproject.qt5.android.bindings.QtActivity"
      android:label="BassChordsCompass"
      android:screenOrientation="portrait"
      android:launchMode="singleTop">
      <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
      </intent-filter>
    </activity>
  </application>
  <uses-sdk
    android:minSdkVersion="16"
    android:targetSdkVersion="16"/>
  <supports-screens
    android:xlargeScreens="true"
    android:largeScreens="true"
    android:normalScreens="true"
    android:anyDensity="true"
    android:smallScreens="true"/>
</manifest>
```

Полноэкранный режим

Практически все приложения на мобильных устройствах запускаются в полноэкранном режиме. Но не всегда они занимают всю область экрана. Обычно в верхней части экрана остается область уведомлений, в которой пользователь может отслеживать время, заряд аккумулятора и т. д. (рис. 64.3).

Если ваше приложение задумано так, что оно должно оккупировать всю область экрана, то придется внести некоторые изменения в файлы свойств.

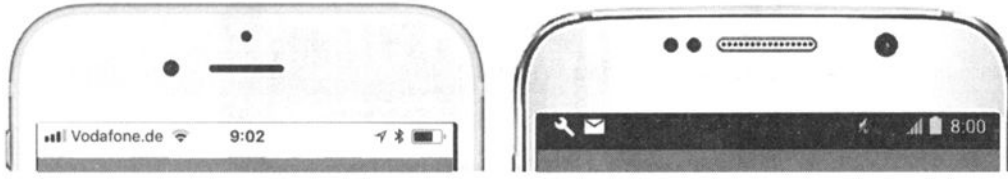


Рис. 64.3. Область уведомления: iPhone (*слева*) и смартфон на Android (*справа*)

iOS-реализация

Для iOS необходимо дополнить plist-файл (см. листинг 64.1) строками, приведенными в листинге 64.3.

Листинг 64.3. Активация полноэкранного режима на iOS

```
<key>UIRequiresFullScreen</key>
<true/>
<key>UIStatusBarHidden</key>
<true/>
<key>UIStatusBarHidden~ipad</key>
<true/>
```

Android-реализация

Для активации полноэкранного режима на Android достаточно добавить в тег `application` файла AndroidManifest.xml (см. листинг 64.2) следующую строку с атрибутом `theme`:

```
android:theme="@android:style/Theme.NoTitleBar.Fullscreen"
```

Автоматический поворот

В мобильных устройствах экран обычно находится в вертикальном положении и начинает работу в портретном режиме, а когда пользователь поворачивает устройство в горизонтальное положение, то оно меняет режим отображения на альбомный (рис. 64.4). Другими словами, происходит изменение разрешения экрана. В такие моменты мобильное приложение может отреагировать на это изменение, чтобы адаптироваться под новую область экрана. Но все не так просто, как на словах, потому что чаще всего для этого недостаточно только изменить размеры и пропорции интерфейса приложения. В подавляющем большинстве случаев потребуется изменять и сам способ размещения элементов в интерфейсе либо реализовать две различные версии интерфейса и сделать так, чтобы одна из них отображалась для портретной ориентации, а другая — для альбомной.

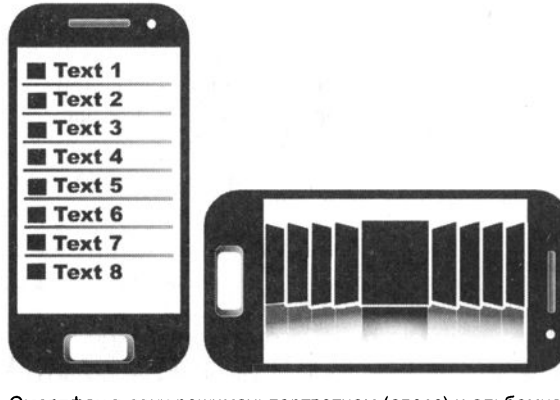


Рис. 64.4. Смартфон в двух режимах: портретном (*слева*) и альбомном (*справа*)

После совершения поворота может получиться так, что поменяется количество элементов, отображаемых на экране, и образуются некрасивые незанятые места, которые необходимо будет с пользой чем-то заполнить. Возьмем, например, приложение для показа фотографий: если в нем в портретной ориентации будет представлен список изображений (рис. 64.4, *слева*), то при повороте в альбомную ориентацию количество элементов этого списка уменьшится, те же элементы, которые останутся в зоне видимости, будут неэффективно использовать пространство экрана. Поэтому более целесообразно для альбомной ориентации использовать другой элемент представления, например, `coverflow`[1] (рис. 64.4, *справа*), или просто запустить показ фотографий.

Хотя режим поворота не всегда является обязательным для мобильных приложений, но, например, видео в большинстве случаев на смартфонах отображают только в альбомной ориентации. При этом выбор — предоставлять ли в приложении возможность автоматического поворота или не предоставлять — есть не всегда, потому что все приложения для iPad, например, обязаны его поддерживать, иначе они не будут пропущены в магазин.

Стратегию реагирования на поворот нужно принимать на самых ранних стадиях разработки приложения. И прежде чем предоставить поддержку для обеих ориентаций, следует продумать, как они будут выглядеть. Это очень важно в плане максимального задействования всех сильных сторон портретного и альбомного предоставлений с целью расширения возможностей приложения.

Также важно сделать выводы о том, чего не стоит делать, чтобы облегчить себе работу для поддержки автоматического поворота. Поэтому постарайтесь придерживаться следующих простых рекомендаций:

- не перегружайте интерфейс приложения большим количеством элементов;
- используйте элементы с простой и понятной логикой — например, кнопки, ползунки и т. п.;
- используйте элементы, которые обладают возможностью изменять свои размеры, — например, `Flickable`, `ListView` и т. п.;
- никогда не задавайте абсолютных позиций для элементов — всегда используйте элементы автоматических размещений и фиксаторы.

Конфигурирование приложений для поддержки поворота

iOS-реализация

Чтобы предоставить приложению для iOS возможность автоматического поворота или убрать ее, необходимо внести в plist-файл (см. листинг 64.1) изменения и дополнения, приведенные в листинге 64.4. Варианты ориентации для iPhoine указываются после тега `key` со значением `UISupportedInterfaceOrientation`. Для iPad это значение тоже указывается после тега `key`, но с той разницей, что оно дополнено фрагментом `~ipad`, т. е. `UISupportedInterfaceOrientations~ipad`.

В нашем примере (см. листинг 64.4) iPhone поддерживает две портретные ориентации: одна — нормальная, другая — вверх ногами, а iPad поддерживает все виды ориентации. Если, для

[1] Cover Flow — трехмерный графический интерфейс пользователя, включенный в iTunes, Finder и другие продукты компании Apple Inc. для наглядного поиска и быстрого просмотра файлов и цифровых медиабиблиотек посредством графических изображений обложек. — *Ред.*

iPad нам, например, не нужна портретная ориентация вверх ногами, то нужно просто убрать нижнюю строчку с тегом:

```
<string>UIInterfaceOrientationPortraitUpsideDown</string>.
```

А чтобы добавить к iPhone две альбомные ориентации: нормальную и вверх ногами, нужно просто добавить строчки:

```
<string>UIInterfaceOrientationLandscapeLeft</string>
<string>UIInterfaceOrientationLandscapeRight</string>
```

Листинг 64.4. Управление режимами поворота на iOS

```
<key>UISupportedInterfaceOrientations</key>
<array>
        <string>UIInterfaceOrientationPortrait</string>
        <string>UIInterfaceOrientationPortraitUpsideDown</string>
</array>
<key>UISupportedInterfaceOrientations~ipad</key>
<array>
        <string>UIInterfaceOrientationLandscapeLeft</string>
        <string>UIInterfaceOrientationLandscapeRight</string>
        <string>UIInterfaceOrientationPortrait</string>
        <string>UIInterfaceOrientationPortraitUpsideDown</string>
</array>
```

Android-реализация

Для Android возможности поворота задаются атрибутом `screenOrientation` в теге `activity` файла AndroidManifest.xml (см. листинг 64.2). Этот атрибут необходимо добавить или заменить в зависимости от необходимости приложения одной из следующих строк:

- портретная ориентация: `android:screenOrientation="portrait"`;
- портретная ориентация вверх ногами: `android:screenOrientation="reversePortrait"`;
- альбомная ориентация: `android:screenOrientation="landscape"`;
- альбомная ориентация вверх ногами: `android:screenOrientation="reverseLandscape"`;
- все виды ориентации: `android:screenOrientation="unspecified"`

Значения ориентации можно объединять друг с другом логической операции ИЛИ. Так, для поддержки приложением двух вариантов ориентации: альбомной и альбомной вверх ногами — можно указать следующее значение атрибута:

```
android:screenOrientation="landscape" | "reverseLandscape".
```

Обработка поворота в приложениях

Отконфигурировав приложение для поворотов, вам необходимо программно отслеживать и обрабатывать их. В C++ для этого можно воспользоваться объектом класса `QScreen`. Указатель на него возвращает метод `QGuiApplication::primaryScreen()`. При изменениях ориентации экрана этот объект высылает сигнал `primaryOrientationChanged()`, который передает в параметре текущую ориентацию экрана. Этот сигнал можно соединить со слотом обработки. Вот как это может выглядеть в коде программы:

```
QScreen* pscreen = qApp::primaryScreen();
connect(pscreen,
        SIGNAL(primaryOrientationChanged(Qt::ScreenOrientation)),
        SLOT(slotOrientationChanged(Qt::ScreenOrientation))
       );
```

Метод обработки принимает в параметре значение текущей ориентации экрана `Qt::ScreenOrientation`, на которое мы можем отреагировать следующим образом — например, так:

```
void MySmartApp::slotOrientationChanged(Qt::ScreenOrientation so)
{
    if (so == Qt::LandscapeOrientation
        || Qt::InvertedLandscapeOrientation) {
        // использовать размещение для альбомной ориентации
    }
    else if (so == Qt::PortraitOrientation
            || Qt::InvertedPortraitOrientation) {
        // использовать размещение для портретной ориентации
    }
}
```

В QML есть обертка для объекта класса `QScreen` — элемент `Screen`. Этот элемент доступен в модуле QtQuick.Window 2.2. Разница с C++ заключается в том, что этот элемент не высылает сигналы. Как быть? Для того чтобы получать уведомления о событиях поворота экрана, мы связываем свойство `primaryOrientation` элемента `Screen` со своим собственным свойством `orientation` (листинг 64.5). Далее мы реализуем свойство для обработки изменений его значений `onOrientationChanged`. Реализация в этом свойстве похожа на ту, которую мы делали только что на C++.

Листинг 64.5. Обработка события поворота экрана в QML

```
import QtQuick 2.8
import QtQuick.Window 2.2
Item {
    width: 320
    height: 480
    Text {id:txt; anchors.centerIn: parent}
    property int orientation: Screen.primaryOrientation
    onOrientationChanged: {
        if (orientation === Qt.LandscapeOrientation
            || orientation === Qt.InvertedLandscapeOrientation) {
            txt.text = "Landscape Orientation"
        }
        else if (orientation === Qt.PortraitOrientation
            || orientation === Qt.InvertedPortraitOrientation) {
            txt.text = "Portrait Orientation"
        }
    }
}
```

Сенсоры

Все мобильные устройства оснащены дополнительными датчиками, их количество и возможности варьируются от одной модели к другой. Эти датчики предоставляют дополнительные интересные возможности, такие как, например, определение местоположения в географическом пространстве, определение освещения в комнате и многое другое. Для того чтобы использовать сенсоры в своих программах, нужно добавить в проектный файл модуль QtSensors:

```
QT += qml quick sensors
```

Все сенсоры имеют три основных свойства:

- `dataRate` — управляет частотой замеров, которая задается в герцах;
- `active` — управляет состоянием активности: значение `false` означает, что сенсор не должен выполнять замеры, значение `true` означает, что замеры должны производиться в соответствии с заданным интервалом времени;
- `onReadingChanged` — свойство обработки события новых данных. Должно содержать код для обработки считанных замеров сенсора.

В табл. 64.1 указаны сенсоры, которые обычно входят в комплектацию большинства смартфонов и планшетов.

Таблица 64.1. *Типы сенсоров*

Имя элемента сенсора	Описание
`QML: Accelerometer` `C++: QAccelerometer`	Предоставляет информацию об ускорении перемещения устройства в трехмерном пространстве. Можно использовать, например, как альтернативу для джойстика
`QML: AmbientLightSensor` `C++: QAmbientLightSensor`	Предоставляет информацию об освещении, попадающем на смартфон или таблет
`QML: AmbientTemperaturSensor` `C++: QAmbientTemperaturSensor`	Предоставляет информацию об окружающей температуре
`QML: Magnetometer` `C++: QMagnetometer`	Предоставляет информацию об угле в направлении на север. Можно использовать, например, для реализации компаса
`QML: Gyroscope` `C++: QGyroscope`	Предоставляет данные о текущем положении устройства в трехмерном пространстве в градусах. Можно использовать, например, для реализации инструмента уровня

Рассмотрим использование сенсоров на примере акселерометра. Это очень популярный сенсор — он может легко заменить клавиши курсора или джойстик для игр. При этом, поворачивая устройство влево или вправо, можно рулить автомобилем, а наклонив устройство вниз, снизить его скорость. На рис. 64.5 показаны различные положения смартфона и соответствующие им значения акселерометра.

В нашем примере (рис. 64.6) мы реализуем приложение, в котором будем перемещать белый шарик в сторону наклона смартфона, используя значение акселерометра (листинг 64.6).

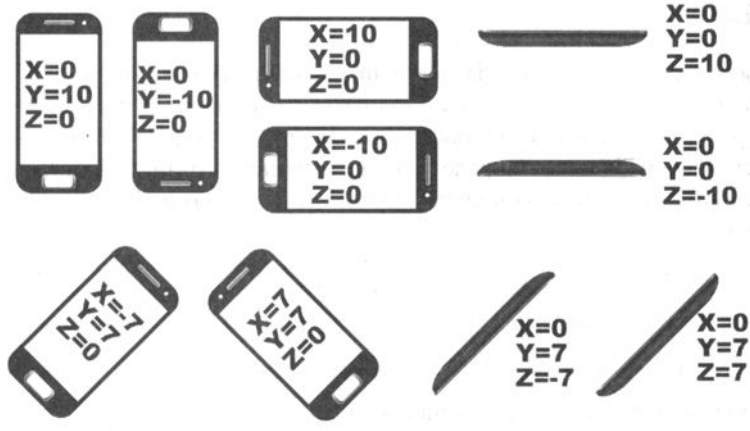


Рис. 64.5. Значения акселерометра в различных положениях смартфона

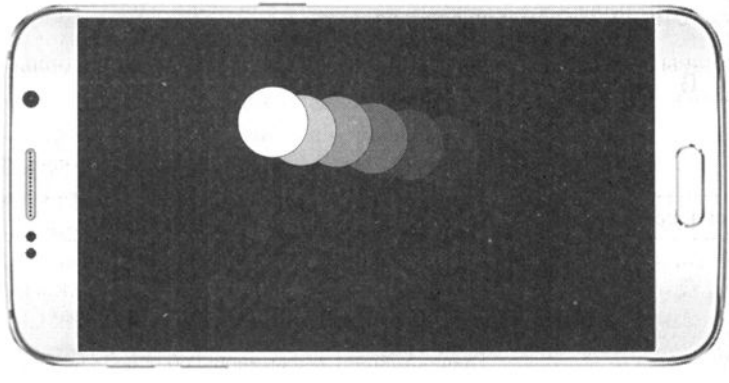


Рис. 64.6. Практический пример использования акселерометра

Листинг 64.6. Использование сенсора `Accelerometer`

```
import QtQuick 2.8
import QtSensors 5.1
Rectangle {
    id: mainRect
    width: 320
    height: 480
    color:  "black"
    Rectangle {
        id: rect
        color:  "white"
        x: parent.width / 2
        y: parent.height / 2
        width: parent.width / 6
        height: width
        radius: width / 2
```

```
        Accelerometer {
            dataRate: 1000
            skipDuplicates: true
            axesOrientationMode: Accelerometer.FixedOrientation
            active: Qt.application.state
            onReadingChanged: {
                rect.x += reading.x
                rect.y += reading.y
                var maxX = mainRect.width - rect.width
                var maxY = mainRect.height - rect.height
                rect.x = rect.x > maxX ? maxX : rect.x < 0 ? 0 : rect.x
                rect.y = rect.y > maxY ? maxY : rect.y < 0 ? 0 : rect.y
            }
        }
    }
}
```

В листинге 64.6 мы размещаем в элементе `Rectangle` еще один элемент `Rectangle`, который будет выступать в роли белого шарика. Для того чтобы сделать из прямоугольника окружность, мы закругляем его углы с помощью свойства `radius`. Начальное расположение шарика устанавливаем свойствами `x` и `y` в районе середины экрана. В качестве диаметра задаем одну шестую часть ширины экрана. Далее присваиваем элементу сенсора `Accelerometer` интервал для получения данных, равный одной тысяче замеров в секунду (свойство `dataRate`). Для того чтобы экономить заряд аккумулятора и не получать одни и те же данные, включаем режим исключения повторных данных (свойство `skipDuplicates`). Устанавливаем направление осей сенсора так, чтобы они оставались всегда неизменны, даже при повороте экрана устройства (свойство `axesOrientaionMode`). В целях экономии аккумулятора мы хотим, чтобы сенсор не работал, если вдруг приложение станет неактивным или невидимым для пользователя. Поэтому делаем активность сенсора (свойство `active`) зависимой от текущего состояния приложения `Qt.application.state`. В свойстве обработки события новых данных `onReadingChanged` мы считываем из свойства `reading` новые значения `x` и `y` и вычисляем новую позицию для нашего шарика.

Библиотека Qt предоставляет сенсоры более высокого уровня, которые базируются на реализации ряда сенсоров из табл. 64.1. Для упрощения и ускорения разработки в некоторых случаях более удобно будет воспользоваться именно ими. Эти сенсоры указаны в табл. 64.2.

Таблица 64.2. *Сенсоры более высокого уровня*

Имя элемента сенсора	Описание
QML: `RotationSensor` C++: `QRotationSensor`	Сообщает о поворотах устройства вокруг осей в трехмерном пространстве
QML: `Compass` C++: `QCompass`	Специальная реализация для компаса
QML: `OrientationSensor` C++: `QOrientationSensor`	Позволяет определить, какой стороной повернуто устройство
QML: `TapSensor` C++: `QTapSensor`	Сообщает о касаниях и двойных касаниях пользователя

Завершим обзор сенсоров особым сенсором, который не входит ни в одну из двух групп табл. 64.1 и 64.2, отличается своими свойствами и располагается в отдельном модуле. Это датчик определения геопозиции. Он использует сразу две составляющие: GPS (Global Positioning System) и сетевые ресурсы. Причем, благодаря сетевым ресурсам, он работает не только на мобильных устройствах, но так же и на обычных настольных и переносных компьютерах. Для его использования в проектный файл необходимо включить модули Qt Network и Qt Positioning:

```
QT += quick qml network positioning
```

В простом примере, показанном на рис. 64.7, мы отображаем значения текущего местоположения, где **Latitude** — значение широты, а **Longitude** — долготы. При отрицательных значениях Latitude устройство расположено в южном от экватора полушарии, а при положительных — в северном. При отрицательных значениях Longitude устройство расположено к западу от нулевого меридиана (Гринвич), а при положительных — к востоку (листинг 64.7).



Рис. 64.7. Значения широты и долготы для города Дармштадт (Германия)

В листинге 64.7 мы импортируем модуль `QtPositioning`. Для отображения значений Latitude и Longitude используем элемент `Text`. Основные действия происходят в свойстве `onPositionChanged` элемента сенсора `PositionSource`. Там мы получаем актуальные значения позиций из свойства `position.coordinate` и присваиваем их при помощи идентификатора `txt` свойству `text` для отображения.

Листинг 64.7. Использование сенсора `PositionSource`

```
import QtQuick 2.8
import QtPositioning 5.2
Item {
    width: 320
    height: 480
    Text {
        id: txt
        anchors.centerIn: parent
        PositionSource {
            updateInterval: 1000
            active: Qt.application.state
```

```
            onPositionChanged: {
                txt.text =
                    "Latitude:" + position.coordinate.latitude
                  + "<br>Longitude:" + position.coordinate.longitude
            }
        }
    }
}
```

Пользовательский ввод при помощи пальцев

Для взаимодействия с сенсорным экраном пользователи мобильных устройств обычно используют пальцы. Очень важно понять, что палец — это не то же самое, что мышь компьютера. С одной стороны, пользователь может одновременно коснуться экрана в четырех разных местах. С другой — у пользователя нет возможности нажать на сенсорном экране на правую кнопку, которая есть у мыши. Все это немного усложняет разработку приложений, которые намерены поддерживать множественное касание «мультитач». О том, как технически реализовать поддержку мультитач в программах на C++, рассказано в *главе 14*, а про поддержку мультитач в QML — в *главе 57*.

На мобильных устройствах очень распространены *жесты*. Жесты включают в себя одно или множественное касание и дополнительное движение, например поворот. Если вы написали приложение с поддержкой жестов, то его лучше всего проверять с помощью настоящего устройства, а не на эмуляторе, — эмулятор не даст вам того же восприятия, как «живой» тест. В табл. 64.3 приводятся основные применяемые пользователями жесты, которые необходимо иметь в виду для реализации приложений и проведения тестов.

Таблица 64.3. *Основные типы жестов*

Название	Описание	Визуальный образ
Tap (Касание)	Касание одним пальцем какого-либо элемента управления. Подобно щелчку мыши	
Double Tap (Двойное касание)	Двойное касание пальцем какого-либо элемента управления. Подобно двойному щелчку мыши	
Swipe (Скользить)	Перемещение пальца, не отрывая его от поверхности сенсорного экрана. Может, например, использоваться для перелистывания изображений или страниц	

Таблица 64.3 *(окончание)*

Название	Описание	Визуальный образ
Flick (Резкое движение)	Палец очень быстро перемещается по поверхности экрана. Применяется для быстрой смены между показами изображений	
Pinch-in, Pinch-out (Сужение, Растяжение)	Задействованы два пальца, которые прикасаются к объекту, затем раздвигаются и увеличивают его либо сдвигаются и уменьшают его. Этот жест можно применять, например, для того, чтобы увеличить на фотографии мелкие детали либо чтобы перейти из просмотра фотографий в режим обозревателя фотоизображений	
Rotate (Поворот)	Задействованы два пальца, которые прикасаются к объекту, затем рука поворачивается и вместе с ней объект поворачивается вокруг своей оси. Этот жест можно применять, например, для того, чтобы сменить расположение фотографий из горизонтального положения в вертикальное	

Виджеты и элементы QML по умолчанию обладают поддержкой необходимых для них жестов. В том же случае, если вы хотите реализовать элемент с собственной реакцией на жесты Pinch-in, Pinch-out и Rotate (см. табл. 64.3), вы можете воспользоваться программой, приведенной в листинге 64.8. Она представляет белый квадрат на черном фоне (рис. 64.8) — прикоснувшись к нему двумя пальцами и раздвинув или сдвинув их, вы произведете изменение размера квадрата, а поворотом пальцев повернете его вокруг собственной оси.

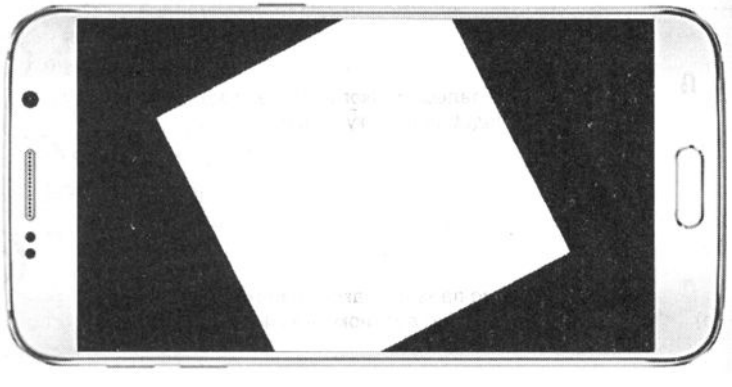


Рис. 64.8. Приложение обработки жестов: сужение/растяжение и поворот

Листинг 64.8. Обработка жестов растяжение/сужение и поворот

```
import QtQuick 2.8
Rectangle {
 id: rectMain
```

```
 width: 320
 height: 480
 color: "black"
 Rectangle {
   id: rect
   color: "white"
   anchors.centerIn: rectMain
   width: Math.min(rectMain.width, rectMain.height) / 2
   height: width
   PinchArea {
     anchors.fill: rect
     pinch.minimumRotation: -360
     pinch.maximumRotation: 360
     pinch.minimumScale: 0.5
     pinch.maximumScale: 2.0
     onPinchStarted: print("PinchStarted")
     onPinchUpdated: {
       rect.rotation = -pinch.angle
       var nW    = rect.width * pinch.scale
       var nMax   = Math.min(rectMain.width, rectMain.height)
       var nMin   = nMax / 2
       rect.width  = nW > nMax ? nMax : nW < nMin ? nMin : nW
       rect.height  = rect.width
     }
     onPinchFinished: print("PinchFinished")
   }
 }
}
```

В листинге 64.8 мы помещаем один элемент `Rectangle` в другой. Первый представляет собой фон главного окна приложения, второй — это белый квадрат, который мы помещаем в центр окна приложения при помощи свойства фиксатора `centerIn`. Для реализации жестов `Pinch` и `Rotate` можно использовать элемент `PinchArea`. Это не визуальный элемент, поэтому при помощи свойства фиксатора `fill` мы размещаем его на всей области белого квадрата. В элементе `PinchArea` при помощи свойств `pinch.maximumRotation` и `pinch.minimumRotation` мы задаем диапазон для вращения как по часовой стрелке на 360 градусов, так и против часовой стрелки на те же 360 градусов. Диапазон изменения фактора сужения/растяжения мы ограничиваем от 0.7 до 1.3 свойствами `pinch.minimumScale` и `pinch.minimumScale`. Далее идут свойства обработки событий жестов:

- свойство `onPinchStarted` сигнализирует о том, что жест распознан, и пользователь начал его выполнять. В этом свойстве мы отображаем информационное сообщение на консоли;
- свойство `onPinchUpdated` сигнализирует о том, что пользователь находится в процессе выполнения жеста, и о том, что значения параметров жеста изменились. В этом свойстве мы изменяем размеры и поворот белого квадрата в соответствии с новыми значениями жеста. Эти значения содержатся в свойствах `pinch.angle` и `pinch.scale`. Мы используем идентификатор `rect` и присваиваем свойствам `rotate`, `width` и `height` новые значения;
- свойство `onPinchEnd` сигнализирует о том, что пользователь завершил выполнение жеста. В этом свойстве мы выводим информационное сообщение на консоль.

Положение рук

Чаще всего смартфоны держат в одной руке (рис. 64.9, *слева*). Планшеты никто с помощью одной руки не держит, и самым распространенным способом является вариант держания его двумя руками снизу (рис. 64.9, *справа*). Оба эти способа имеют одну общую особенность, которая заключается в том, что пальцы находятся внизу, а следовательно, могут закрывать видимость нижней части сенсорного экрана. Из этого наблюдения следуют два вывода:

- всю информацию, предназначенную для отображения, которую должен видеть пользователь, лучше размещать в верхней половине экрана или хотя бы немного выше области, в которой пользователь может орудовать пальцами;
- все элементы управления, которые используются чаще всего, лучше размещать в нижней половине экрана — в зоне досягаемости большого пальца руки, держащей смартфон (или больших пальцев рук — для планшета).

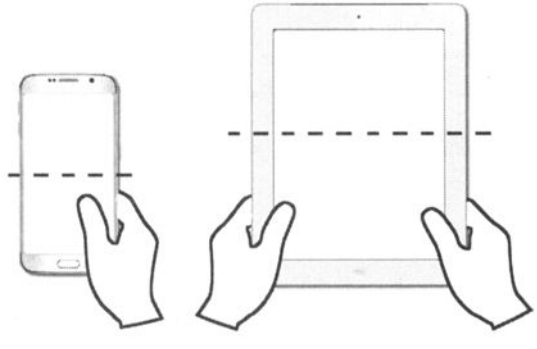


Рис. 64.9. Зоны досягаемости больших пальцев: смартфон (*слева*), планшет (*справа*)

Резюме

Разработка приложений для мобильных устройств требует особого подхода, который отличается от разработки приложений для настольных и переносных компьютеров. Это разница связана с рядом особенностей: от поведенческих факторов самих пользователей до ограничений, связанных с аппаратной частью устройств.

Очень важную часть в разработке приложений играют файлы свойств, структуру которых необходимо знать, чтобы конфигурировать создаваемые приложения в соответствии с их возможностями.

Все мобильные устройства оснащены сенсорами, которые предоставляют дополнительные интересные возможности, — такие как, например, регистрация перемещения устройства, попадающий на устройство свет, определение геопозиции и др.

Мобильное устройство может быть повернуто в портретное или альбомное положение. Эти события необходимо корректно обрабатывать.

Пользователи взаимодействуют с сенсорным экраном устройства с помощью пальцев и привыкли использовать жесты. Тестировать работу жестов необходимо на настоящих устройствах.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/64-510/** или с помощью следующего QR-кода (рис. 64.10):



Рис. 64.10. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 65

Пример разработки мобильного приложения

Идея без реализации это ничто!

Вот и пришло время собрать все полученные знания вместе и реализовать настоящее мобильное приложение, которое будет предназначено для публикации в магазине. Мне хотелось в этой главе «убить сразу двух зайцев», и поэтому в процессе разработки этого приложения мы рассмотрим некоторые аспекты, которые еще не успели затронуть для QML в предыдущих главах этой книги, а именно:

- использование модуля `QtMultimedia`;
- использование элемента таймера;
- использование модуля `QtQuick.Extras`;
- использование сенсора гироскопа;
- распознавание операционной системы, на которой запущено приложение;
- реализация механизма масштабирования, который нужен для реализации независимости приложений от разрешения экранов устройств;
- использование системы *активов приложений* (assets) в качестве альтернативы системе ресурсов Qt. Она рекомендуется к применению для файлов с большим размером и файлов мультимедиа. В активах приложений обычно сохраняются значки приложений.

Обдумывание и планирование приложения

Прежде всего, нужно не создавать что-либо, а представить, обдумать, спланировать.

Никола Тесла

Все начинается с идеи. Это как ЭВРИКА вашего личного осознания. Всего накопленного вами жизненного и практического опыта. И теперь самое главное — как только вы начнете разрабатывать свое первое мобильное приложение, вам вдруг может прийти в голову следующая гениальная идея, и еще одна, а потом еще одна и т. д. Главное — ни в коем случае не останавливайтесь и никогда не бросайте еще не завершенного до конца приложения.

Но также никогда не следует забывать родившиеся новые, пусть и не использованные пока идеи. Поэтому настоятельно рекомендую вам завести тетрадь, куда вы станете их записывать. Эта тетрадь может быть не только в бумажном, но также и в электронном виде, — на смартфоне или на вашем компьютере.

Как только вы отправите подготовленное приложение в магазин и будете готовы для реализации следующего, это будет означать, что пришло время достать вашу заветную тетрадь и найти в ней нужные вам идеи. Почему я об этом говорю? Потому что это будет гарантировать вам воплощение своих интересных идей для разработки приложений. К примеру, когда мне понадобилась идея мобильного приложения для этой книги, то, как всегда, на помощь мне пришла такая тетрадь.

Я был очень требователен к этому приложению, т. к. мне хотелось реализовать не просто очередной пример, а нечто практичное и полезное. Вместе с тем, приложение должно было быть простым в реализации и иметь компактный исходный код. Я листал свою тетрадь в поисках идей. И вот она — приложение-сигнализация!

Идея заключается в том, что бы реализовать для смартфона сигнализацию по принципу действия автомобильной. Устройства такой сигнализации ставятся на автомобили с той целью, чтобы к ним никто не прикасался без ведома их хозяев. Для смартфона реализация подобного механизма могла бы быть интересной для тех, например, случаев, когда вы оставили свой смартфон лежать на столе и не хотите, чтобы его кто-нибудь забрал, или, наоборот, хотите узнать, прикоснется ли кто-нибудь к нему, и кто это будет. Вы могли бы также поставить на сигнализацию смартфон своего ребенка, чтобы он не смог использовать его тайком, а занимался бы полезными делами, — учил уроки, например. Можно будет поставить при помощи смартфона сигнализацию и на дверь, просто подложив его под нее, после чего никто не сможет проникнуть в ваше помещение незамеченным. Да что там говорить, применений такому устройству можно найти множество — нужно только включить воображение.

Итак, идея есть, теперь определимся с названием нашего приложения.

Название приложения

Поскольку в большинстве случаев название приложения — это также и название проекта, то очень важно определиться с ним еще до написания первых строчек кода. Ни в коем случае не торопитесь и подходите к выбору названия очень обдуманно — ведь «как корабль назовешь, так он и поплывет», — говорил капитан Врунгель и был прав. Удачно подобранное название может значительно повысить количество загрузок приложения из магазина. Название должно раскрывать функциональное назначение приложения — чтобы пользователь не тратил своего времени на вопросы о том, что оно может делать. Такой подход способен повысить вероятность того, что на него обратят внимание.

В то же время очень важно, чтобы название приложения было уникальным, т. е. никто до вас его еще не использовал, — это поможет избежать конфликтов с другими разработчиками.

Одним из кандидатов на хорошее описательное название для нашего приложения может стать «Don't touch my phone» (Не трогай мой смартфон) — задаем его в поисковике и видим, что приложение с таким названием уже есть на Google Play. Поэтому попробуем другое название: «Do not touch it» (Не трогай это). Приложений с таким названием в поисковиках пока нет, поэтому останавливаемся на нем.

Значок приложения

Первое, что пользователи видят в магазинах приложений по своим поисковым запросам, — это значки приложений и их названия. Поэтому значок важен не меньше названия. Хорошо подобранный значок способен значительно повысить интерес к приложению. Тут, как нель-

зя кстати, подходит древняя мудрость: «Хорошая картинка стоит тысячи слов». Это значит, что нужно стремиться создавать образы значков так, чтобы они максимально полно передавали информацию о назначении приложения.

Одной из составляющих визуального образа значка для нашего приложения может стать сирена, поскольку она ассоциируется и с сигналом тревоги, и с сигнализацией. Второй составляющей должен стать смартфон — чтобы было понятно, что приложение связано с самим мобильным устройством. Итак, разобьем визуальную область на две половины и соединим обе эти составляющие вместе — у нас получился значок, показанный на рис. 65.1.



Рис. 65.1. Значок приложения

Что будет в первой версии?

Не пытайтесь сделать первую версию приложения совершенной и снабдить ее продвинутыми функциями. Начинайте всегда с малого. Первая версия — это лишь проверка идеи вашего приложения на пригодность. Поэтому отбросьте все лишнее и постарайтесь сделать максимум возможного, чтобы донести эту идею до пользователей. Если идею поймут, и она понравится, то вы всегда сможете выпустить следующую версию приложения и добавить в нее то, чего не хватило вашим пользователям. А чтобы выяснить, чего именно им не хватило, вы можете прочитать ленту откликов на ваше приложение в магазине. Помните, что пользователи — это не вы, и то, что кажется важным вам, совсем не обязательно важно для них. Поэтому всегда развивайте ваше приложение постепенно и идите шаг за шагом от малого к большему в соответствии с пожеланиями и нуждами пользователей. Такой подход сделает приложение лучше и сэкономит массу времени и средств.

Итак, пользовательский интерфейс первой версии нашего приложения мы сделаем максимально простым: две кнопки и одна надпись:

- нажатие на кнопку в левом нижнем углу покажет пользователям панель с информацией о приложении, а надпись вверху окна приложения проинформирует их о том, что мобильное устройство нужно положить на гладкую и неподвижную поверхность (рис. 65.2, *а*);
- центральный элемент приложения — кнопка активации сигнализации. Она содержит поясняющий текст и занимает в окне достаточно места для того, чтобы пользователь сразу ее заметил. Поясняющий текст на кнопке информирует пользователей о том, что для активации сигнализации к этой кнопке нужно притронуться и не отпускать (рис. 65.2, *б*);
- как только пользователь притронется к кнопке активации, начнется движение красной линии вокруг обода кнопки (рис. 65.2, *в*);
- в момент, когда красная линия вокруг обода кнопки сомкнется, сигнализация будет активирована, а сама кнопка сменит свой текст на текст отмены активации (рис. 65.2, *г*);
- теперь, если кто-нибудь сдвинет смартфон с места, сигнализация сработает, и приложение отреагирует звуком сирены и пульсирующим экраном (рис. 65.2, *д*).

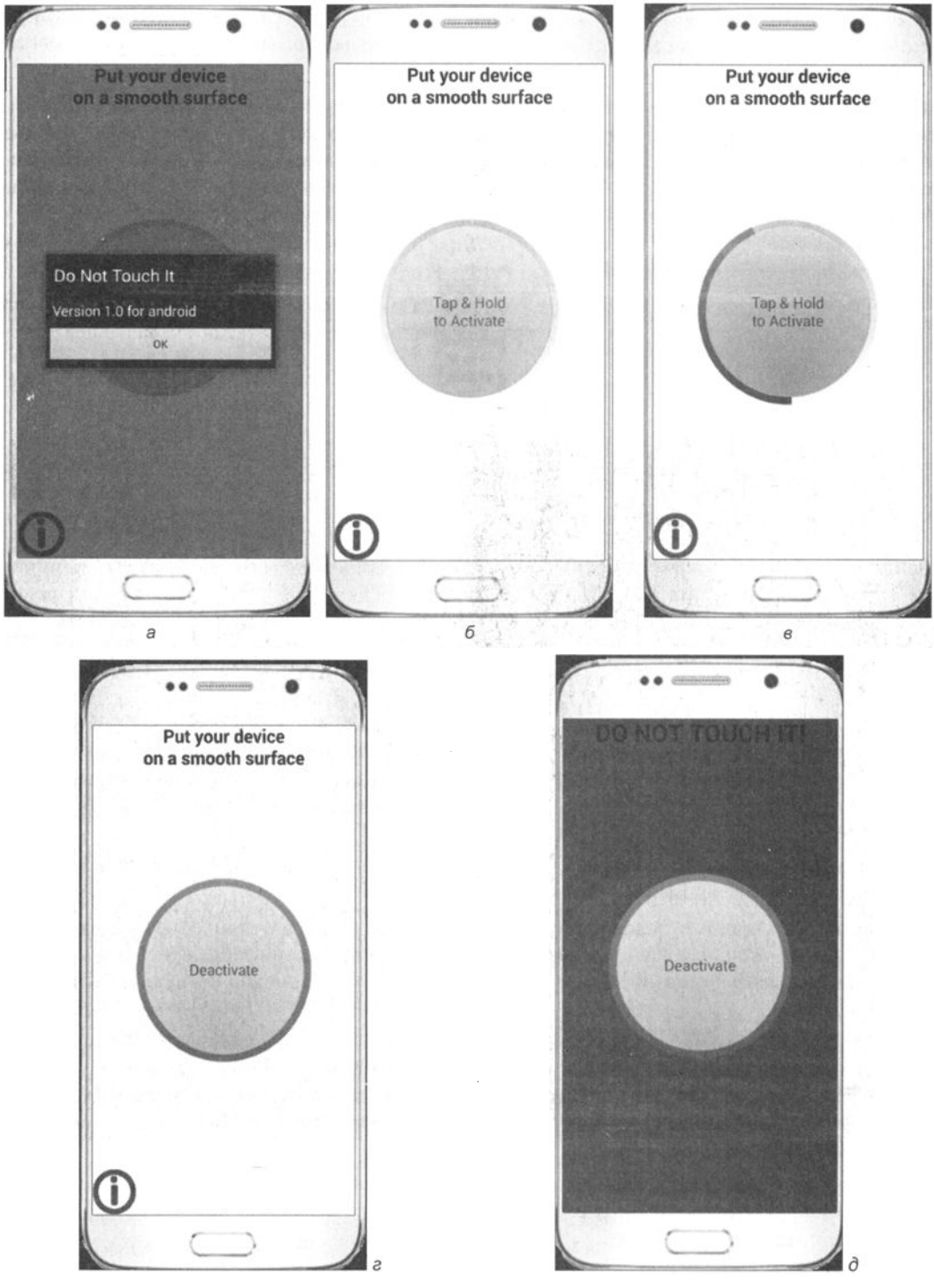


Рис. 65.2. Приложение-сигнализация: *а* — показ информационной панели; *б* — нормальное состояние приложения; *в* — приложение в процессе активирования сигнализации; *г* — приложение с активированной сигнализацией; *д* — срабатывание сигнализации

Пишем код

Наши цели ясны, задачи определены, за работу товарищи!
Никита Сергеевич Хрущев

Теперь, когда наступила полная ясность в том, что должно быть реализовано, можно, наконец, приступить к написанию кода приложения.

Листинг 65.1. Импорт модулей и главный элемент QML-программы (файл main.qml)

```
import QtQuick 2.9
import QtQuick.Dialogs 1.2
import QtQuick.Extras 1.4 as QQE
import QtMultimedia 5.9
import QtSensors 5.9
Rectangle {
    id: main
    width: 320
    height: 480
    property int minDim: Math.min(width, height)
    property bool alarm: false
...
}
```

В первых строках QML-программы (листинг 65.1) мы импортируем модули, которые понадобятся нашему приложению:

- модуль `QtQuick` — необходим для реализации всех базовых визуальных и не визуальных элементов приложения;
- модуль `QtQuick.Dialogs` — нужен для отображения информации о приложении в диалоговом окне (информационной панели);
- модуль `QtQuick.Extras` — нужен для использования элемента кнопки `DelayButton`, которая расположена в середине окна приложения (см. рис. 65.2, *а*). Заметьте, что мы импортируем этот модуль под идентификатором `QQE`. Это сделано для того, чтобы избежать возможных конфликтов в тех случаях, если будет импортироваться модуль `QtQuick.Controls`, который содержит элемент с тем же именем `DelayButton`;
- модуль `QtMultimedia` — нужен приложению для воспроизведения звуковой сигнализации;
- модуль `QtSensors` — с его помощью мы задействуем элемент сенсора гироскопа `Gyroscope`, отслеживающего движения мобильного устройства. По сигналу этого сенсора приложение будет принимать решение о срабатывании сигнализации.

Затем переходим к реализации основной программы и начинаем ее с главного ее элемента, — это центральный элемент приложения, ему мы присваиваем идентификатор `main` и далее будем дополнять его по мере реализации новыми элементами. Здесь мы прежде всего задаем ему размеры 320×480. На мобильном устройстве эти размеры не имеют никакого значения, потому что приложение будет исполняться в полноэкранном режиме и иметь размеры экрана мобильного устройства. Это мы сделали на тот случай, если вдруг приложение будет запущено на компьютере, — чтобы у его окна был заданный размер.

В центральном элементе приложения определены два свойства: `minDim` и `alarm`:

- первое свойство необходимо приложению для реализации масштабирования элементов — оно обеспечит независимость размеров элементов от разрешений экранов мобильных устройств, и они всегда будут занимать заданную часть площади. Это значение вычисляется из разрешения самой малой размерности мобильного устройства.

  Тема масштабирования — это объемная и непростая тема, которая вполне могла бы занять отдельную главу. Но благодаря этому примененному нами несложному приему, можно все упростить и не вдаваться в подробности. Каким образом будет использоваться значение свойства `minDim` для масштабирования, мы покажем в листингах 65.2–65.4;

- свойство `alarm` — это центральная переменная, которая будет информировать нужные элементы о срабатывании сигнализации или устанавливать ее значение.

Теперь мы станем шаг за шагом добавлять в основную программу (см. листинг 65.1) новые элементы, и первыми добавим комбинацию из двух элементов: `Image` и `MessageDialog` (листинг 65.2), основной задачей которых будет предоставление пользователю информации о приложении.

Элемент `Image` мы располагаем при помощи элемента фиксатора (свойство `anchords.bottom`) в нижнем левом углу окна (см. рис. 65.2, *а*). Обратите внимание на самое интересное — на размеры этого элемента. Их мы задаем не конкретным численным значением в пикселах, а в долях от самой малой размерности экрана. Эта размерность хранится в свойстве `minDim` (см. листинг 65.1). В нашем случае мы устанавливаем ширину и высоту (свойства `width` и `height`) равными одной шестой от значения свойства `minDim`. Теперь, каким бы ни было разрешение экранов мобильных устройств, наша информационная кнопка всегда будет занимать на них примерно одинаковую часть площади экрана.

В качестве образа для кнопки мы загружаем растровое изображение InfoButton.png из Qt-ресурсов проекта. Чтобы зафиксировать нажатие пользователем на элемент `Image`, мы заполняем его область элементом `MouseArea`, из которого управляем показом диалогового окна (информационной панели) при помощи идентификатора `aboutDialog`.

Само диалоговое окно по умолчанию является модальным, и при помощи свойства `visible` мы делаем его невидимым. В свойстве `title` мы задаем заголовок окна, и в этом заголовке помимо надписи с названием приложения отображаем строку с операционной системой, на которой исполняется приложение. На первой позиции рис. 65.2, *а* отображается окно (панель), которое содержит строку **android**. В случае с iOS мы бы увидели **ios** и т. д. Если бы требовалось выполнить в QML-коде какие-либо специфические действия для одной из платформ, мы могли бы поступить следующим образом:

```
if (Qt.platform.os === "ios") {/*действия необходимые для iOS*/}
```

В свойстве `text` мы указываем текст, предназначенный для отображения внутри информационного диалогового окна.

Листинг 65.2. Элементы для отображения информации о приложении

```
Image {
    id: infoButton
    anchors.bottom: main.bottom
    width:  main.minDim / 6
```

```
    height:  width
    source: "qrc:/InfoButton.png"
    MouseArea {
        anchors.fill: infoButton
        onClicked: aboutDialog.visible = true
    }
}
MessageDialog {
    id: aboutDialog
    visible: false
    title: "Do Not Touch It for " + Qt.platform.os
    text: "Version 1.0"
}
```

Очень важно, чтобы пользователь понимал, что он должен сделать. Поэтому следующий элемент, который мы внесем в главный элемент листинга 65.1, — это элемент текста `Text` (листинг 65.3). Он будет отображать в верхней части окна приложения небольшое руководство для пользователя (см. свойство `anchors`). Этот текст может изменяться в зависимости от срабатывания сигнализации: после ее срабатывания будет отображаться текст **DO NOT TOUCH IT!** (см. рис. 65.2, *д*), во всех же остальных случаях пользователь будет видеть текст руководства.

Размер шрифта текста мы вычисляем способом, независимым от разрешения экрана — при помощи свойства `minDim`, и присваиваем его свойству `font.pixelSize`.

Листинг 65.3. Элемент с отображением инструкции для пользователя

```
Text {
    anchors.top: main.top
    anchors.horizontalCenter: main.horizontalCenter
    font.pixelSize: main.minDim / 15
    font.bold: true
    horizontalAlignment: Text.AlignHCenter
    text: main.alarm ? "DO NOT TOUCH IT!"
                     : "Put your device\non a smooth surface"
}
```

Следующий кандидат на размещение в основном элементе приложения — это элемент кнопки с задержкой `DelayButton`. Заметьте, что его имени предшествует идентификатор модуля `QQE`, — как было сказано ранее, это сделано, чтобы избежать конфликта при возможном импорте модуля `QtQuick.Controls`, потому что этот модуль содержит элемент с точно таким же названием. Размеры элемента кнопки с задержкой вычисляются независимым от разрешения экрана способом — при помощи свойства `minDim` главного приложения. Этот элемент будет занимать больше половины от самой малой размерности экрана, и мы размещаем его в центре окна приложения при помощи свойства фиксатора `anchors.centerIn`.

В зависимости от состояния кнопки (свойство `checked`) на ней будет отображаться поясняющая надпись с разной информацией. В свойстве обработки изменения состояния кнопки `onCheckedChanged` будет всегда производиться отключение сигнализации.

Листинг 65.4. Размещение элемента кнопки с задержкой

```
QQE.DelayButton {
    id: btn
    width: main.minDim / 1.5
    height: width
    anchors.centerIn: main
    text: checked ? "Deactivate" : "Tap & Hold<br>to Activate"
    onCheckedChanged: main.alarm = false
}
```

Теперь можно дополнить основную программу, представленную в листинге 65.1, и собственно сигнализацией — это будет элемент сенсора `Gyroscope` (листинг 65.5). Только этот элемент получит право включать режим тревоги.

Элемент сенсора в зависимости от состояний кнопки `DelayButton`, рассмотренной в листинге 65.4, включается и выключается свойством `alwaysOn` и приводится в состояние активности свойством `active`.

Свойство `skipDuplicates` с установленным значением `true` позволяет нам экономить заряд аккумулятора за счет того, что мы избегаем получения одинаковых данных.

Для отслеживания движений в свойстве обработки получения данных мы выбираем наибольший поворот по осям X, Y, Z, который присваивается переменной `movement`. Если этот угол при включенном состоянии кнопки `DelayButton` превышает 10 градусов, мы включаем режим тревоги. Режим тревоги означает, что мы присваиваем значение `true` свойству `alarm`, принадлежащему основному элементу приложения.

Листинг 65.5. Сигнализация (элемент гироскоп)

```
Gyroscope {
    active: btn.checked
    alwaysOn: btn.checked
    axesOrientationMode: Gyroscope.FixedOrientation
    skipDuplicates: true
    onReadingChanged: {
        var movement = Math.max(Math.abs(reading.x),
                                Math.abs(reading.y),
                                Math.abs(reading.z)
                               )
        if (movement > 10 && btn.checked) {
            main.alarm = true
        }
    }
}
```

Еще один элемент, который мы добавим в основную программу (см. листинг 65.1), — это элемент таймера (листинг 65.6). Его задачей будет периодически изменять основной фон приложения с белого цвета на красный цвет и наоборот. Действия таймера будут повторяться (см. свойство `repeat`) с момента срабатывания сигнала тревоги и до его отключения.

Листинг 65.6. Изменения цвета фона приложения при помощи таймера

```
Timer {
    id: timer
    interval: 250
    repeat: true
    property bool bBlink: false
    onTriggered:  {
        main.color = bBlink ? "white" : "red"
        bBlink = !bBlink
    }
    onRunningChanged: {
        if (!running) {
            main.color = "white"
        }
    }
}
```

Для того чтобы в момент наступления тревоги приложение могло воспроизводить звук сирены, мы дополним основную программу, приведенную в листинге 65.1, элементом `MediaPlayer` для воспроизведения файлов мультимедиа. Этот элемент будет воспроизводить mp3-файл, расположенный в системных активах (assets) приложения. В свойстве `loops` устанавливаем бесконечный режим воспроизведения. То есть оно будет продолжаться до тех пор, пока не будет отменен сигнал тревоги (см. листинг 65.7). То, каким образом можно организовать системные активы приложения, мы рассмотрим позже в листингах 65.9 и 65.10.

Листинг 65.7. Воспроизведение звука сирены

```
MediaPlayer {
    id: sound
    source: (assetsPath + "alarm.mp3")
    loops: MediaPlayer.Infinite
    volume: 1
}
```

Завершим основную программу, приведенную в листинге 65.1, добавлением в нее свойства `onAlarmChanged` (листинг 65.8). Это свойство реагирует на любое изменение состояния свойства тревоги `alarm` и управляет запуском и остановом воспроизведения звука и таймера, которые мы рассмотрели в листингах 65.6 и 65.7.

Листинг 65.8. Управление воспроизведением звука сирены и таймером

```
onAlarmChanged: {
    if (alarm) {
        sound.play()
        timer.start()
    }
```

```
    else {
        timer.stop()
        sound.stop()
    }
}
```

В листинге 65.9 приведена основная C++ функция нашей программы `main()`. Эта функция реализует стандартный подход для запуска QML-программы при помощи элемента `QQuickView`. Этот класс мы уже использовали в *главе 60* и похожим образом запускали там код QML-программ. Самое интересное место в этом листинге — это то, как мы с помощью объекта класса `QUrl` делаем доступным путь к системным активам из C++ в QML.

Путь к системным активам приложения для каждой операционной системы определяется по-своему, поэтому с помощью директив препроцессора `Q_OS_IOS` и `Q_OS_ANDROID` мы разделяем код на отдельные реализации для Android и iOS. Затем вызовом `setContextProperty()` делаем объект `urlAssetsPath` доступным в QML под именем `"assetsPath"`.

Обратите также внимание на то, что для перекрытия области уведомлений в верхней части экрана и заполнения всей области экрана для iOS необходим вызов `showFullScreen()`.

Листинг 65.9. Основная C++-функция приложения (файл main.cpp)

```
#include <QGuiApplication>
#include <QQuickView>
#include <QQmlContext>
#include <QUrl>

int main(int argc, char** argv)
{
    QGuiApplication app(argc, argv);
    QQuickView view;

    QUrl urlAssetsPath;
#if defined(Q_OS_IOS)
    urlAssetsPath = QUrl("file://" + qApp->applicationDirPath() + "/");
#elif defined(Q_OS_ANDROID)
    urlAssetsPath = QUrl("assets:/Resources/");
#endif
    view.rootContext()->setContextProperty("assetsPath", urlAssetsPath);
    view.setSource(QUrl("qrc:/main.qml"));
    view.setResizeMode(QQuickView::SizeRootObjectToView);

#if defined(Q_OS_IOS)
    view.showFullScreen();
#else
    view.show();
#endif

    return app.exec();
}
```

Теперь перейдем к проектному файлу приложения (листинг 65.10). В секции QT мы указываем четыре модуля. Первые два: quick и qml — предназначены для работы QML-программ, последние два: sensors и multimedia — нужны, чтобы мы могли использовать в приложении сенсор гироскопа и воспроизводить звук.

В секции RESOURCES указываем файл ресурсов. В файле ресурсов resources.qrc помещены исходный файл приложения main.qml и растровое изображение информационной кнопки InfoButton.png.

На обеих платформах мы будем воспроизводить один и тот же звуковой файл сирены alarm.mp3, поэтому определяем его в переменной APP_FILES.files до начала платформных секций.

Далее идет секция ios со следующими переменными:

- QMAKE_INFO_PLIST — задает путь к файлу свойств приложения *.plist;
- QMAKE_BUNDLE_DATA — помещает переданные ей файлы в системные активы приложения. В нашем случае это файл alarm.mp3, на который указывает переменная APP_FILES;
- QMAKE_BUNDLE_DATA — устанавливает для Xcode путь к каталогу с расширением xcassets со значками и стартовыми экранами приложения.

Следом идет секция android со следующими переменными:

- ANDROID_PAKAGE_SOURCE_DIR — задает каталог, где содержится все необходимое для создания пакетных файлов Android (файлов в расширением apk). Например, растровые изображения значков и стартовых экранов, а также файл свойств AndroidManifest.xml;
- OTHER_FILES — эта переменная предполагает прикрепление в проекту любых файлов. С ее помощью мы прикрепляем к нашему проекту собственную версию файла AndroidManifest.xml;
- APP_FILES.path — задает путь для каталога системных активов приложения. В нашем случае мы его задаем с подкаталогом Resources;
- INSTALLS — помещает файлы в каталог системных активов приложения.

Листинг 65.10. Проектный файл DoNotTouchIt.pro

```
TEMPLATE = app
TARGET = DoNotTouchIt
QT += quick qml sensors multimedia
SOURCES += main.cpp
RESOURCES += resources.qrc
APP_FILES.files += ../../common/alarm.mp3
ios {
    QMAKE_INFO_PLIST = _ios/IosInfo.plist
    QMAKE_BUNDLE_DATA += APP_FILES
    QMAKE_ASSET_CATALOGS += $$PWD/_ios/Images.xcassets
}
android {
    ANDROID_PACKAGE_SOURCE_DIR = $$PWD/_android
    OTHER_FILES += _android/AndroidManifest.xml
    APP_FILES.path = /assets/Resources
    INSTALLS += APP_FILES
}
```

Добавление к приложению значков и стартовых экранов

С добавлением значков вся сложность заключается в том, что, согласно требованию магазинов, будет нужно создать не один значок, а целую их серию с различными разрешениями.

Кроме того, запуск приложения занимает время, и оставлять пользователя любоваться на черный экран и гадать, что происходит, не хорошо. Поэтому всем приложениям необходимо показывать при запуске стартовый экран.

Наш стартовый экран, показанный на рис. 65.3, создан на базе значка приложения. Надпись с информацией о том, что необходимо немного подождать, мы разместили внизу. Теперь при показе этого экрана будет сразу понятно, что приложение находится в процессе запуска.

Каждая из мобильных платформ имеет свои собственные требования на то, сколько значков и стартовых экранов должно быть, и то, в каком разрешении и где они должны храниться и как называться. Существуют программные решения, позволяющие ускорить и автоматизировать процесс создания значков и стартовых экранов. Они создают их во всех необходимых разрешениях и для целого ряда платформ. К таким программам можно отнести, например, Quick Icons и Icon Helper.



Рис. 65.3. Стартовый экран приложения

Далее приводится описание использования значков и стартовых экранов нашего проекта DoNotTouchIt, исходный код которого вы можете использовать в качестве шаблона для создания собственных приложений.

iOS-реализация

В iOS существуют два способа использования значков и стартовых экранов.

Первый заключается в том, что их помещают в системный актив приложения и описывают в файле свойств (см. *разд. «Файл свойств iOS-приложений» главы 64*).

Второй способ, который принят в проекте DoNotTouchIt, заключается в том, что создается специальный каталог Images.xcassets, в который помещают каталоги *.appiconset — с файлами значков и *.launchimage — с файлами стартовых экранов (рис. 65.4). Помимо значков и стартовых экранов каталоги снабжены сопроводительными файлами Contents.json в формате JSON.

Каталог Images.xcasset необходимо указать в проектном файле в секции `QMAKE_ASSET_CATALOGS` (см. листинг 65.10).



Рис. 65.4. Содержимое каталога Images.xcassets для значков и стартовых экранов

В табл. 65.1 приведены все необходимые для размещения в каталоге *.appiconset значки с их разрешениями и названиями файлов.

Таблица 65.1. *Необходимые значки для iOS-приложения*

Название файла	Разрешение
AppIcon20x20.png	20×20
AppIcon20x20@2x.png	40×40
AppIcon20x20@3x.png	60×60
AppIcon29x29.png	29×29
AppIcon29x29@2x.png	58×58
AppIcon29x29@3x.png	87×87
AppIcon40x40.png	40×40
AppIcon40x40@2x.png	80×80
AppIcon40x40@3x.png	120×120
AppIcon60x60@2x.png	120×120
AppIcon60x60@3x.png	180×180
AppIcon76x76.png	76×76
AppIcon76x76@2x.png	152×152
AppIcon83x83@2x.png	167×167
AppIcon1024x1024.png	1024×1024

В табл. 65.2 приведены все необходимые для размещения в каталоге *.launchimage стартовые экраны с их разрешениями и названиями файлов.

Таблица 65.2. Необходимые стартовые экраны для iOS-приложения

Название файла	Разрешение
iPadLandscape_1x.png	1024×768
iPadLandscape_2x.png	2048×1536
iPadLandscapeWithoutStatusBar_1x.png	1024×748
iPadLandscapeWithoutStatusBar_2x.png	2048×1496
iPadPortrait_1x.png	768×1024
iPadPortrait_2x.png	1536×2048
iPadPortraitWithoutStatusBar_1x.png	768×1004
iPadPortraitWithoutStatusBar_2x.png	1536×2008
iPhoneLandscapeRetinaHD55.png	2208×1242
iPhonePortrait_2x.png	640×960
iPhonePortrait_Retina.png 4	640×1136
iPhonePortraitRetinaHD47.png	750×1334
iPhonePortraitRetinaHD55.png	1242×2208

Android-реализация

Для Android-приложения необходимы шесть значков с одинаковыми именами icon.png, каждый из которых нужно разместить в отдельном каталоге, а также и стартовый экран logo.png (рис. 65.5).

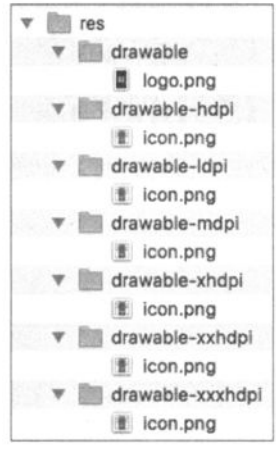


Рис. 65.5. Содержимое каталога res для значков и стартовых экранов

В табл. 65.3 приведены имена каталогов и разрешения значков, которые должны находиться в этих каталогах. Все каталоги со значками должны находиться в каталоге res, который в свою очередь находится в каталоге, который указан в проектном файле в секции ANDROID_PACKAGE_SOURCE_DIR (см. листинг 65.10).

Таблица 65.3. *Необходимые значки для Android-приложения*

Название каталога	Разрешение значка
drawable-ldpi	36×36
drawable-mdpi	48×48
drawable-hdpi	72×72
drawable-xhdpi	96×96
drawable-xxhdpi	144×144
drawable-xxxhdpi	192×192

Стартовый экран для Android-приложений представлен в проекте DoNotTouchIt одним файлом с названием logo.png, который находится в каталоге drawable.

Резюме

Мы спланировали, обдумали и реализовали проект первой версии приложения сигнализации для мобильных устройств. А заодно познакомились с возможностью проигрывания звука, использования таймера, сенсора гироскопа, системных активов приложения и реализации масштабирования.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/65-510/** или с помощью следующего QR-кода (рис. 65.6):



Рис. 65.6. QR-код для доступа на страницу комментариев к этой главе



ГЛАВА 66

Публикация в магазине мобильных приложений

Успех способствует тому, кто занимается тем, что хочет.

Вот мы и подошли к долгожданному моменту — размещению приложения в магазине. И теперь наберитесь терпения, потому что предстоит проделать еще много работы:

- зарегистрироваться и получить лицензию разработчика, если вы это еще не успели сделать;
- создать для приложений электронные подписи — потому что ко всем приложениям, предназначенным для отправки в магазин, должен применяться специальный механизм асимметричного шифрования. И каждому разработчику необходимо создать свою собственную электронную подпись, с помощью которой он будет подписывать свои приложения;
- создать страницу приложения, заполнить все ее обязательные поля и предоставить на ней всю нужную информацию о приложении. В эту информацию входят: тексты описания приложения, снимки с экранов, ссылки на веб-ресурсы и т. д.;
- подготовить пакет приложения, предназначенный для размещения в магазине, и отослать его на страницу приложения.

Кроме этого, специально для App Store еще очень важно зарегистрировать хотя бы одно реальное iOS-устройство, на котором вы сможете проверять все ваши приложения, перед тем как отослать их в магазин.

Этим всем мы сейчас и займемся — сначала для App Store, а потом для Google Play.

Этапы работы для App Store

Итак, вы собираетесь выпускать приложения для iOS? Если ваш ответ «да», то тогда вам необходимо создать учетную запись разработчика для App Store и приобрести лицензию.

Если вы ответили на поставленный вопрос «нет» — перелистывайте страницы этой главы до тех пор, пока не достигнете *разд. «Этапы работы для Google Play»*.

Регистрация

В настоящее время индивидуальная лицензия разработчика приложений стоит 99 долларов США в год. Для прохождения процедуры регистрации учетной записи разработчика и приобретения лицензии откройте в веб-браузере ссылку: **www.developer.apple.com/programs/** или воспользуйтесь QR-кодом, приведенным на рис. 66.1.

Рис. 66.1. Ссылка на страницу регистрации аккаунта разработчика App Store

Если регистрация прошла успешно, то я вас поздравляю — вы только что стали разработчиком приложений для App Store.

Теперь давайте сразу запустим Xcode и добавим в него вашу учетную запись. Для этого выберите пункт меню **Xcode**, затем в открывшемся подменю — пункт **Preferences**, в открывшемся окне — вкладку **Acconts** и нажмите в ее нижнем левом углу кнопку «+». В результате вы увидите окно с надписью **Select the type of account you would like to add** (Выберите тип учетной записи, которую вы хотите добавить), показанное на рис. 66.2. В этом окне выберите самый верхний пункт **Apple ID** (Идентификатор Apple) и нажмите кнопку **Continue** (Продолжить). В следующем окне потребуется заполнить входные данные вашей учетной записи и нажать на кнопку **Sign In** (Войти), после чего ваша учетная запись будет добавлена в Xcode.



Рис. 66.2. Добавление учетной записи в Xcode

Настройки для запуска приложений на реальных устройствах

Перед размещением приложений в магазине необходимо их тщательно протестировать на реальных iOS-устройствах. И теперь, когда у вас появилась лицензия разработчика, вы, наконец, можете это сделать.

Для этого устройство необходимо сначала зарегистрировать. Соедините ваш iPhone или iPad USB-кабелем с Mac и снова запустите Xcode. В открывшемся окне Xcode выберите пункт меню **Window** (Окно), затем в открывшемся подменю — пункт **Devices and**

Simulators (Устройства и симуляторы). В открывшемся диалоговом окне обратите внимание на пункт **Identifier** (Идентификатор) с длинным номером (рис. 66.3) — это и есть идентификационный номер вашего устройства. Каждое устройство, выпущенное Apple, имеет свой собственный уникальный номер. Выделите этот номер и скопируйте в промежуточный буфер обмена с помощью комбинации клавиш <Command>+<C>.

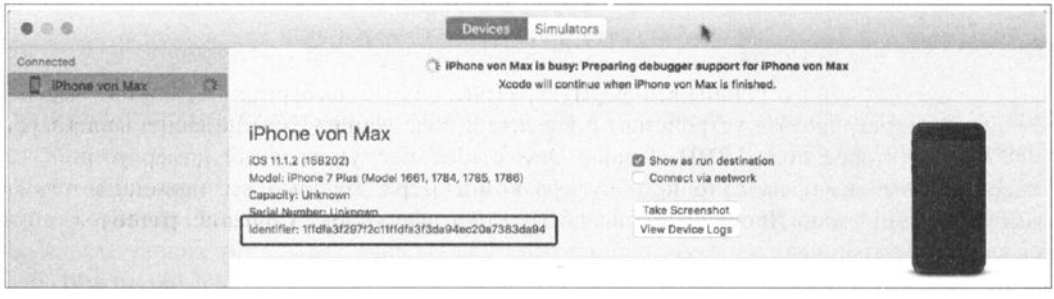


Рис. 66.3. Окно Xcode для управления устройствами разработчика

Затем откройте веб-браузер, перейдите в портал разработчиков Apple по ссылке: **http://developer.apple.com/** и введите там свои входные данные. Когда вас запустят на веб-страницу, выберите пункт **Account** — чтобы попасть в личный кабинет. Затем выберите с левой стороны или в середине окна веб-страницы пункт, который начинается со слова **Certificates** (эти пункты выделены на рис. 66.4 рамками).

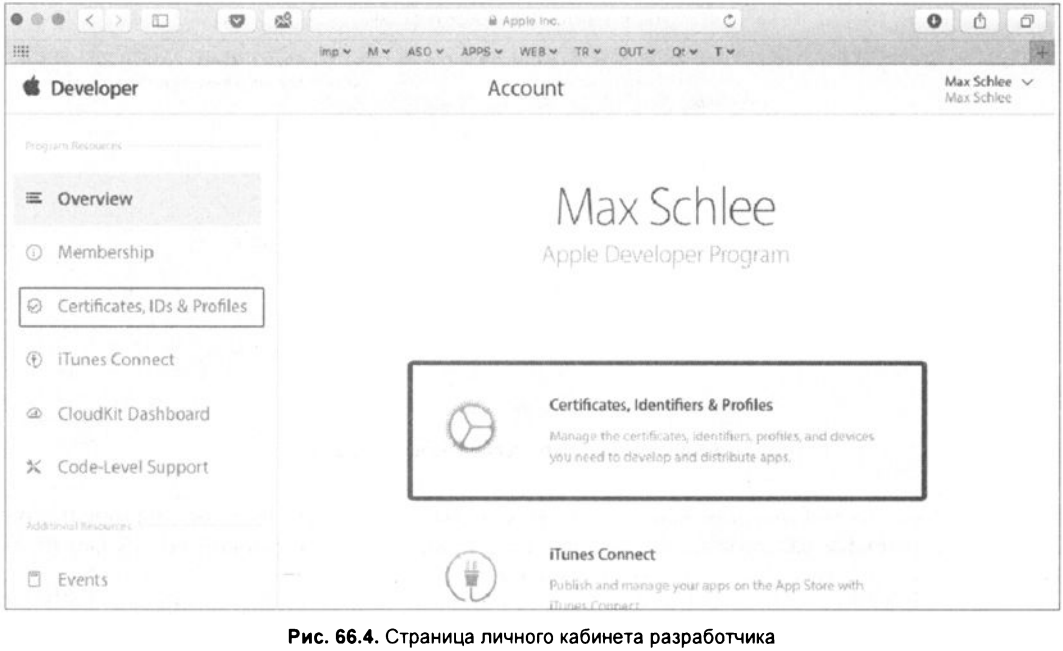


Рис. 66.4. Страница личного кабинета разработчика

Открывшуюся веб-страницу прокрутите вниз, найдите с левой стороны секцию **Devices** (Устройства), выберите в ней пункт **All** (Все) и нажмите с правой стороны страницы кнопку «+», показанную в рамке на рис. 66.5.

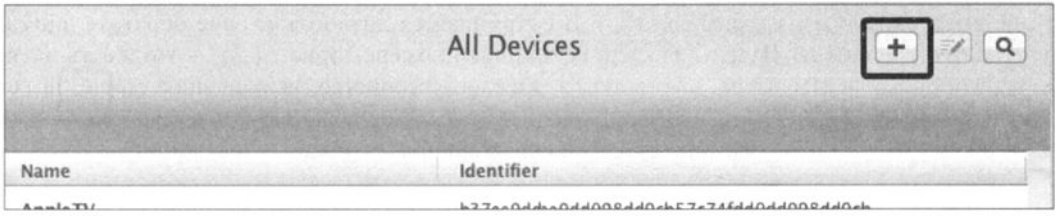


Рис. 66.5. Страница зарегистрированных устройств

Перед вами откроется страница веб-формуляра (рис. 66.6) — выберите в ней пункт **Register Device** (Зарегистрировать устройство) и внесите в поле **Name** (Имя) название вашего устройства. Во второе поле **UDID** (Unique Device Identifier, уникальный номер устройства) вставьте номер из промежуточного буфера компьютера, который вы запомнили там на самом первом шаге настройки (см. рис. 66.3), и нажмите кнопку **Continue**, расположенную в самом низу страницы.

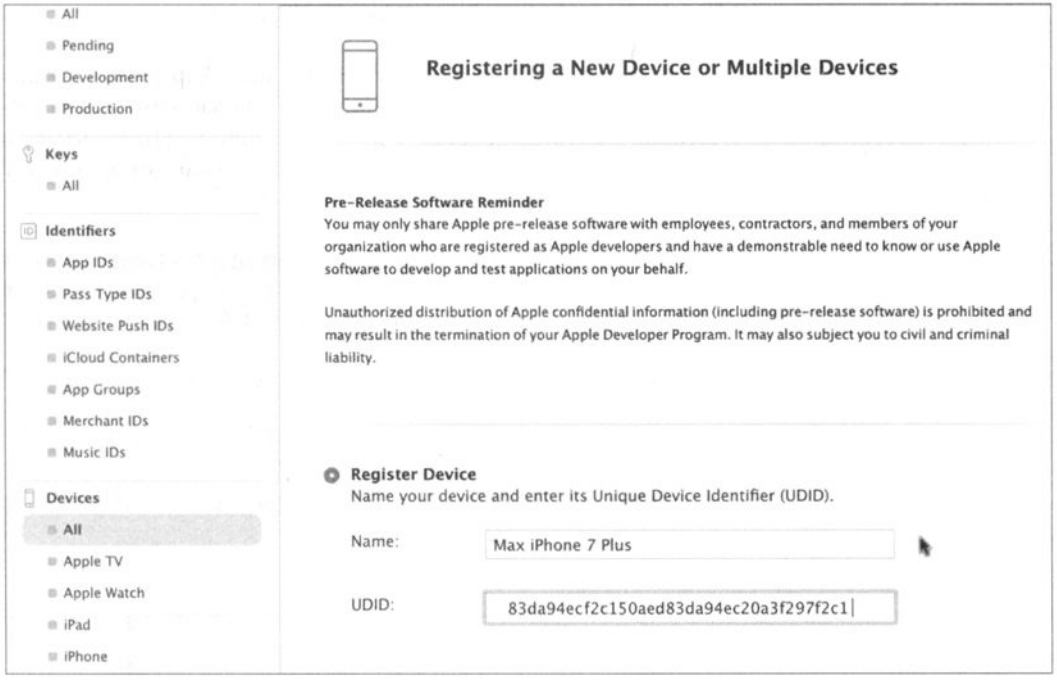


Рис. 66.6. Страница регистрации нового устройства

Итак, устройство зарегистрировано. Но это еще не все, потому что приложения, предназначенные для запуска на устройстве, должны быть подписаны электронной подписью, и для этого нам нужно создать сертификат разработки.

Вернитесь в верх страницы, выберите в секции **Certificates** (Сертификаты) пункт **Development** (Развитие) и нажмите на кнопку «+», расположенную вверху справа и выделенную на рис. 66.7 рамкой.

Откроется страница веб-формуляра (рис. 66.8), на которой выберите опцию **iOS App Development** и нажмите на кнопку **Continue**, расположенную внизу страницы.

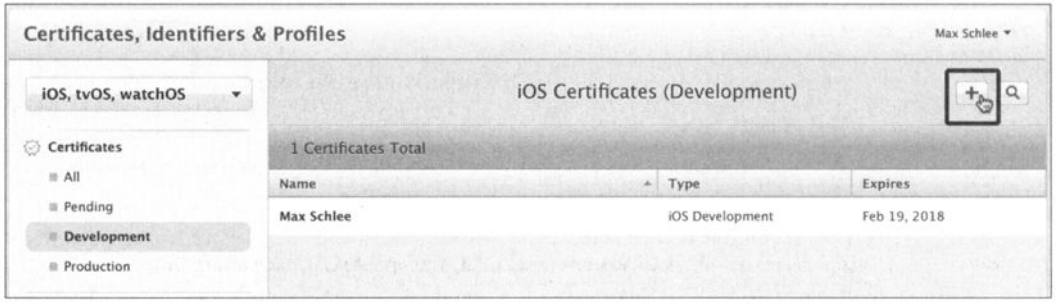


Рис. 66.7. Страница сертификатов разработки

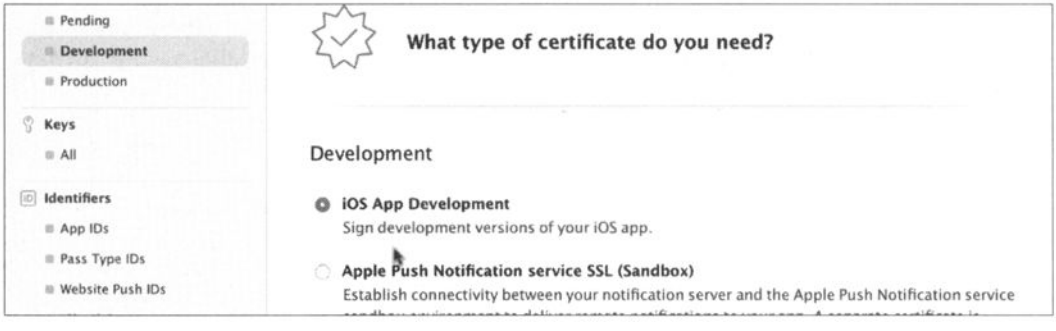


Рис. 66.8. Выбор сертификата разработки для его создания

Теперь перед вами откроется веб-страница создания сертификата с надписью **Generate your certificate** (Создать сертификат). На этой странице есть кнопка для выбора файла **Choose File**, которая выделена на рис. 66.9 рамкой. Этой кнопкой мы воспользуемся позже, т. к. необходимый файл для загрузки на эту страницу необходимо еще создать. Создается этот файл с помощью программы Keychain Access, которая является стандартной утилитой Mac OS X.

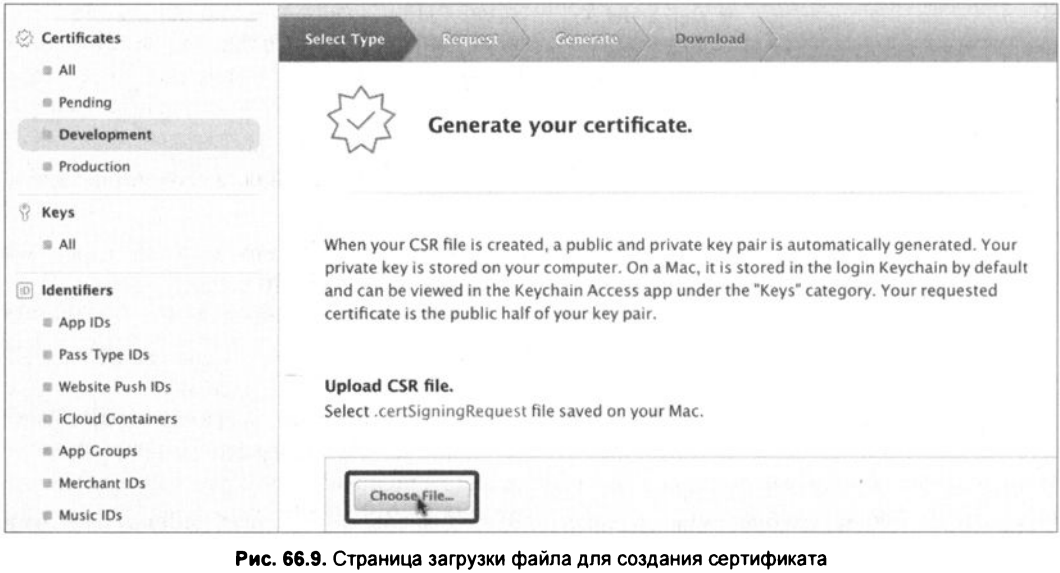


Рис. 66.9. Страница загрузки файла для создания сертификата

Так что запустим утилиту Keychain Access, выберем в ее меню пункт **Keychain Access** (Доступ в Keychain), в открывшемся подменю — пункт **Certificate Assistant** (Помощник по сертификату) и в его подменю — **Request a Certificate From a Certificate Authority** (Запросить сертификат из центра сертификации), как показано на рис. 66.10.

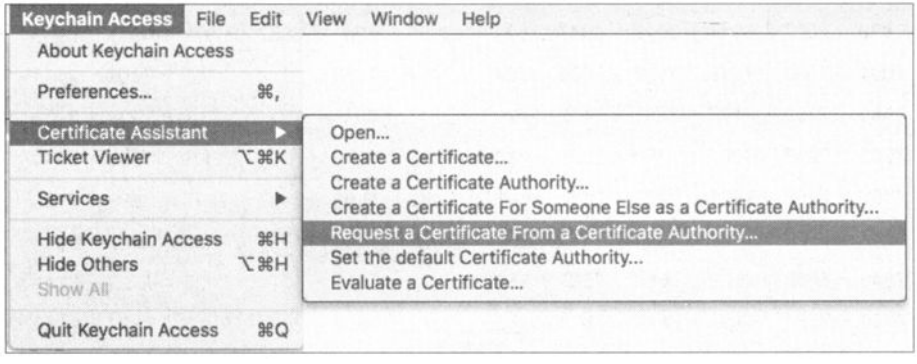


Рис. 66.10. Программа Keychain Access: получение файла для создания сертификата

Поля открывшегося диалогового окна (рис. 66.11) заполните следующим образом: в поле **User Email Address** введите адрес своей электронной почты, а в поле **Common Name** — свои имя и фамилию, следующее поле оставьте пустым, а в пункте ниже выберите опцию **Saved to disk** (Сохранить на диск) и нажмите на кнопку **Continue**. В открывшемся диалоговом окне выберите место, где должен быть сохранен файл, и нажмите кнопку **Save**.

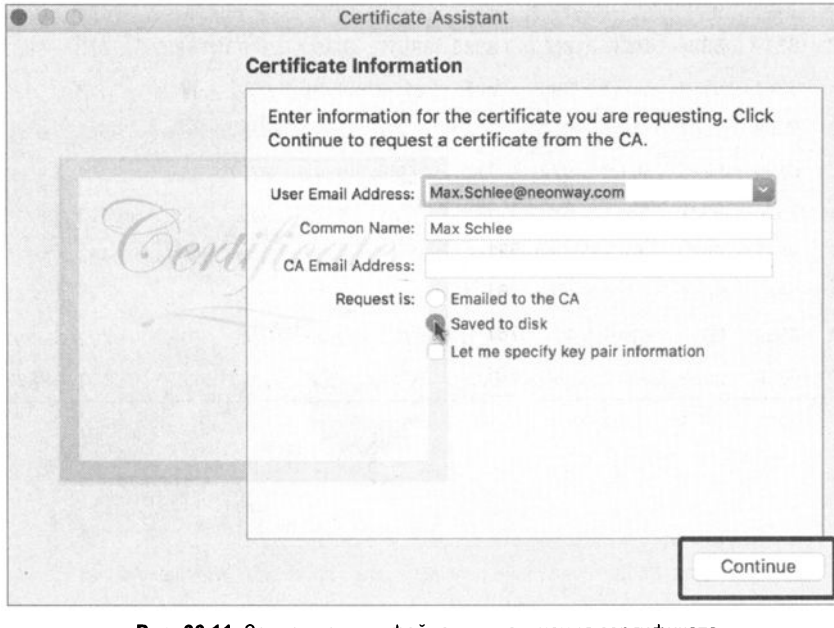


Рис. 66.11. Запись на диск файла для получения сертификата

Теперь вернитесь к веб-странице с кнопкой для выбора файла **Choose File** (см. рис. 66.9) и нажмите на эту кнопку. В открывшемся диалоговом окне найдите и загрузите созданный нами в программе Keychain Access файл. После этого вы увидите страницу с созданным

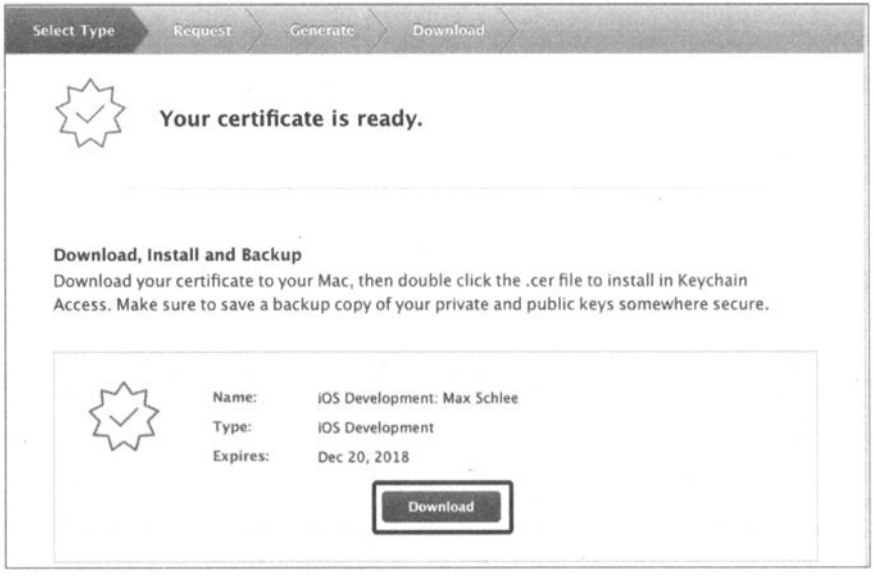


Рис. 66.12. Загрузка файла сертификата

сертификатом. Нажимаем на кнопку **Download** и сгружаем сертификат в каталог загрузок веб-браузера (рис. 66.12).

После загрузки находим файл и щелкаем на нем двойным щелчком — откроется диалоговое окно, в котором у вас попросят разрешения добавить сертификат в программу Keychain Access (рис. 66.13). Нажмите в этом окне на кнопку **Add** (Добавить).

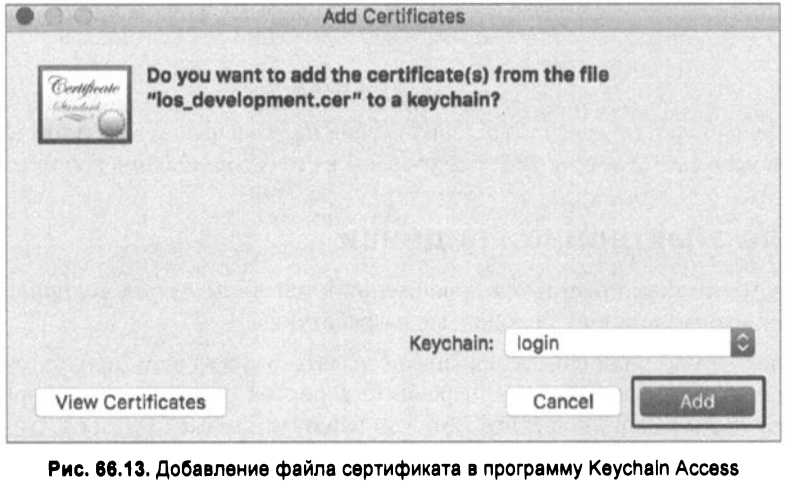


Рис. 66.13. Добавление файла сертификата в программу Keychain Access

Отлично! Вся процедура выполнена до конца! Теперь для быстрой проверки — чтобы убедиться в том, что все прошло удачно, — вы можете запустить Xcode и создать в нем новый iOS-проект на свой выбор. После чего нажать на список устройств — в этом списке теперь, помимо симуляторов, в самом верху должно появиться ваше iOS-устройство (рис. 66.14). Выберите его и нажмите на кнопку компиляции и запуска проекта, расположенную слева.

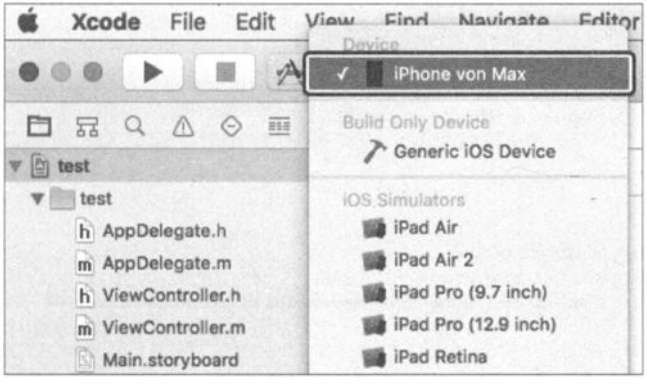


Рис. 66.14. Использование зарегистрированного устройства в Xcode

Если все было сделано правильно, то вы сначала увидите диалоговое окно, спрашивающее у вас разрешение подписать приложение (рис. 66.15). Для того чтобы оно больше вас об этом не спрашивало, нажмите кнопку **Always Allow** (Разрешать всегда). Немного погодя вы увидите приложение, запущенное на вашем iOS-устройстве.



Рис. 66.15. Диалоговое окно подписи приложений

Теперь тщательно протестируйте свое приложение на реальных iOS- и Android-устройствах до тех пор, пока вы не будете полностью уверены в его безошибочной работе.

Создание электронной подписи

Мы уже знаем, что перед отправкой приложений в магазины App Store, приложения должны быть подписаны электронной подписью разработчика.

Для создания этой подписи необходимо создать сертификат продуктов (Production Certificate) — откройте веб-браузер, перейдите в портал разработчиков Apple по ссылке: **http://developer.apple.com/** и введите там свои входные данные. Когда вас запустят на веб-страницу, выберите пункт **Account** — чтобы попасть в личный кабинет. Затем выберите с левой стороны или в середине окна веб-страницы пункт, который начинается со слова **Certificates** (эти пункты выделены на рис. 66.4 рамками). Теперь найдите слева в секции **Certificates** пункт **Production**, выберите его и нажмите на кнопку «+», показанную на рис. 66.16 в рамке.

Откроется страница веб-формуляра (рис. 66.17) — выберите на ней опцию **App Store and Ad Hoc** и нажмите на кнопку **Continue** внизу.

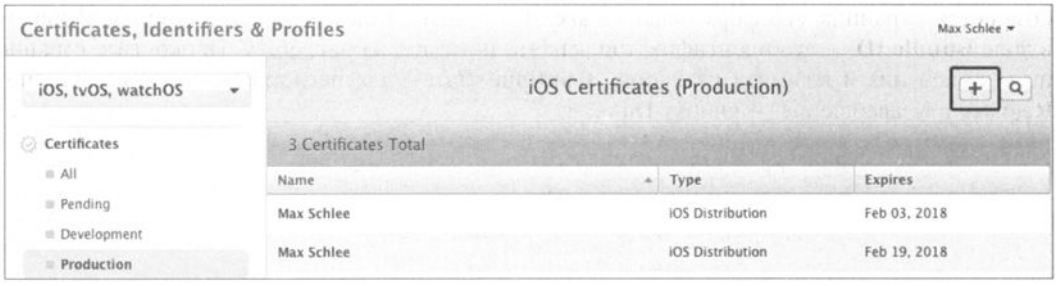


Рис. 66.16. Страница сертификатов для подписи продуктов

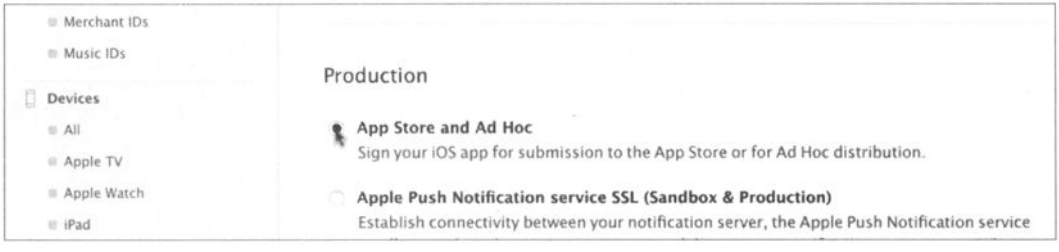


Рис. 66.17. Выбор типа сертификата продуктов

Следующие шаги для создания и установки сертификата продуктов идентичны шагам, которые мы только что проделали для создания и установки сертификата разработки. Выполните их с момента, показанного на рис. 66.10.

Создание страницы приложения

Для создания страницы приложения необходимо сначала создать App ID (идентификатор приложения). Для этого откройте веб-браузер, перейдите в портал разработчиков Apple по ссылке: **http://developer.apple.com/** и войдите в личный кабинет. Затем выберите с левой стороны или в середине окна веб-страницы пункт, который начинается со слова **Certificates** (эти пункты выделены на рис. 66.4 рамками).

Теперь найдите слева в секции **Identifiers** пункт **App IDs**, выберите его и нажмите на кнопку «+», показанную на рис. 66.18 в рамке.

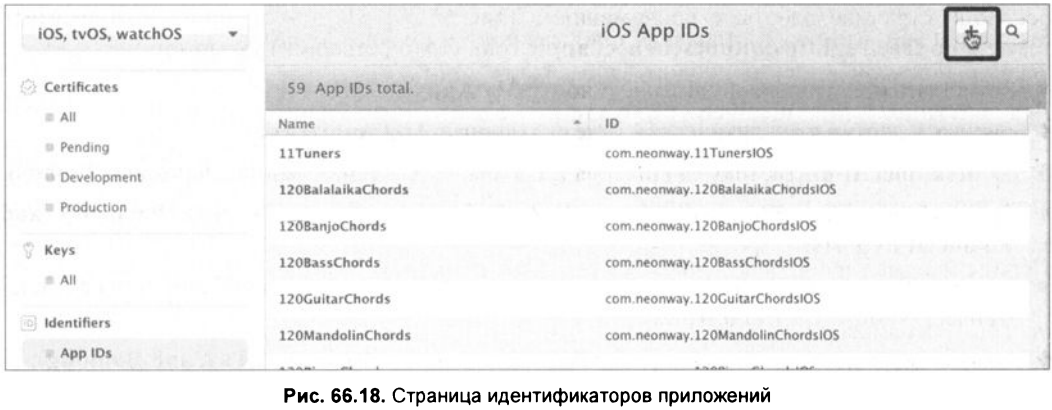


Рис. 66.18. Страница идентификаторов приложений

Откроется страница веб-формуляра — введите в поле **Name** имя вашего приложения и в поле **Bundle ID** — его идентификатор, как это показано на рис. 66.19. Прокрутите страницу в самый низ и нажмите на кнопку **Continue**. Затем для регистрации нажмите кнопку **Register** и в завершение — кнопку **Done**.

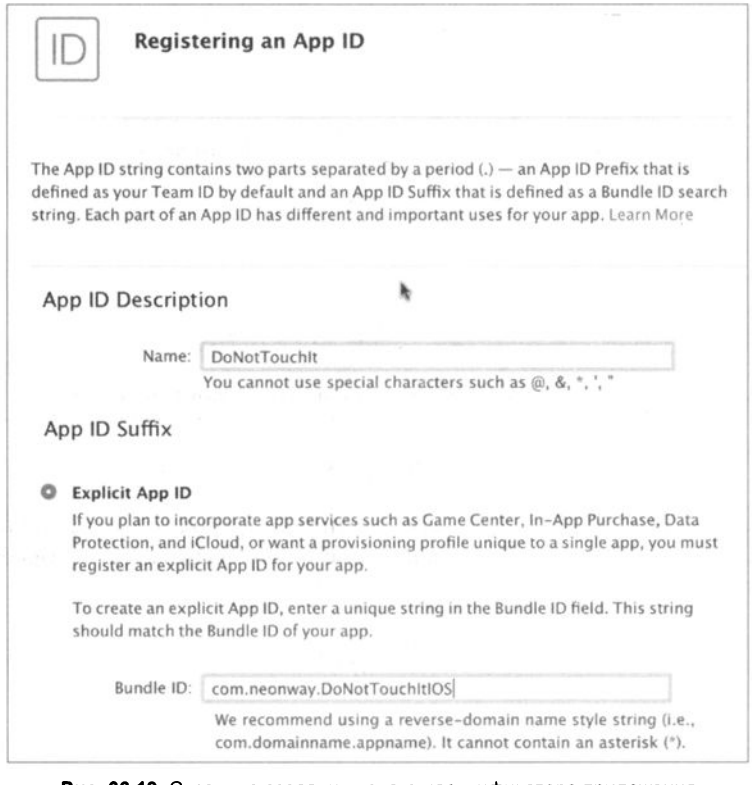


Рис. 66.19. Страница создания нового идентификатора приложения

Только что созданные идентификаторы приложений не сразу доступны, поэтому подождите пять минут. Подождали? Теперь нажмите в личном кабинете разработчика расположенную вверху справа кнопку **Account** — откроется страница, показанная на рис. 66.4, нажмите в середине страницы на большую кнопку с надписью **iTunes Connect**, и вы окажетесь на основной странице работы с приложениями (рис. 66.20). На эту страницу можно выйти также и по ссылке: **https://itunesconnect.apple.com**. С этой страницы вы можете:

- размещать приложения в магазине (кнопка **My Apps**);
- получать доступ к аналитическим отчетам (кнопка **App Analytics**);
- просматривать статистику скачивания и продаж приложений (кнопка **Sales and Trends**);
- просматривать ежемесячные выплаты от Apple на ваш счет (кнопка **Payments and Financial Reports**);
- создавать пользователей и управлять их правами доступа к вашей информации на iTunes Connect (кнопка **User and Roles**);
- подписывать документы соглашений App Store (кнопка **Agreemets Tax, and Banking**);
- получать доступ к помощи (кнопка **Resources and Help**).

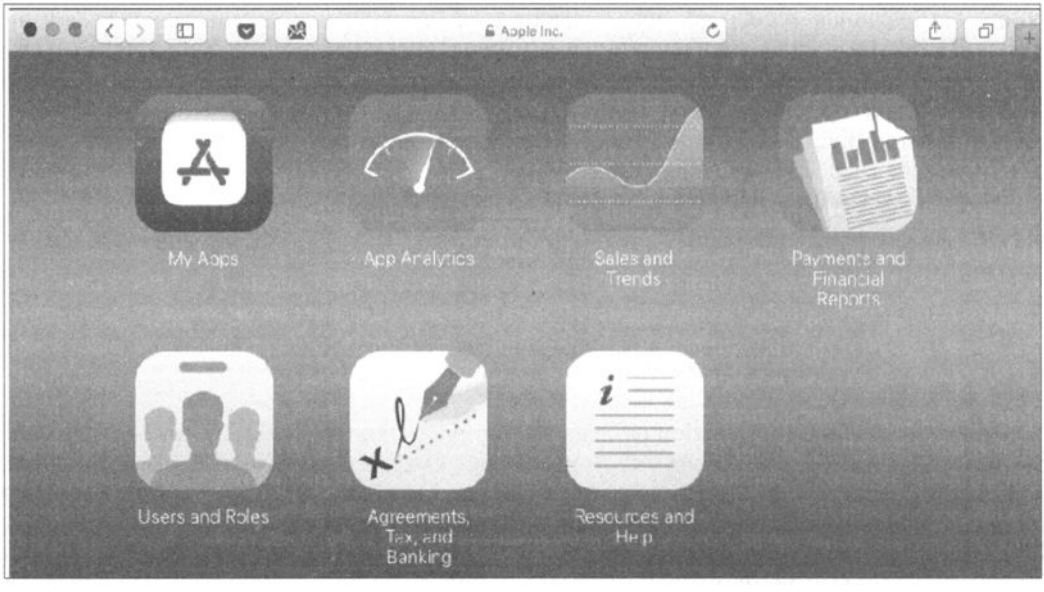


Рис. 66.20. Страница iTunes Connect с опциями управления приложениями и другой информацией

Нам нужно разместить приложение, поэтому нажмите на кнопку **My Apps**. Перед вами откроется веб-страница, в левом верхнем углу которой вы увидите кнопку «+» — нажмите на нее и выберите пункт **New App**, выделенный рамкой на рис. 66.21.

Перед вами откроется страница веб-формуляра, показанная на рис. 66.22. Так как наше приложение рассчитано на iOS, поставьте соответствующий флажок в секции **Platforms**. В поле



Рис. 66.21. Добавление новой страницы приложения

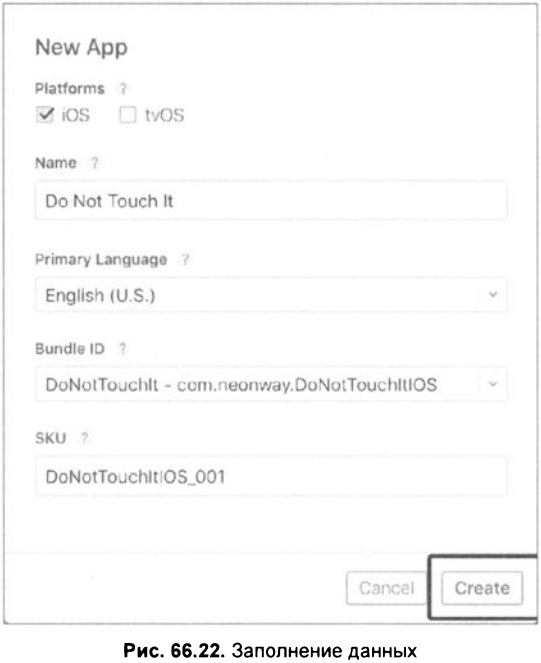


Рис. 66.22. Заполнение данных для страницы приложения

Name внесите имя приложения — это то самое имя, которое будет отображаться в магазине приложений. В качестве основного языка рекомендую выбрать английский: **English (U.S.)**. В пункте **Bundle ID** выбираем из выпадающего списка идентификатор приложения, который мы создали при помощи формуляра, показанного на рис. 66.19. В пункте **SKU** задаем идентификатор приложения для iTunes Connect. Этот идентификатор предназначен только для внутреннего использования — например, для отслеживания финансовой информации приложения. Нажимаем на кнопку **Create** и создаем страницу приложения.

На созданной нами странице приложения имеются три раздела (показаны слева на рис. 66.23):

- **App Information** (информация о приложении);
- **Pricing and Availability** (цена и доступность приложения);
- **Prepare for Submission** (подготовка к отправке приложения).

В секции **Localizable Information** раздела **App Information** (рис. 66.23), справа от названия секции отображается текущий язык — английский **English (U.S.)**. Мы можем все оставить как есть — тогда для всех пользователей будет отображаться информация на английском языке. А можем добавить еще языки, на которые хотим сделать перевод, — например, русский. Для каждого языка мы можем задать собственное имя приложения (поле ввода **Name**), собственный подзаголовок (поле ввода **Subtitle**) и внести собственную ссылку на документ **Privacy Policy** (Политика конфиденциальности).

- В секции **General Information** раздела **App Information** (см. рис. 66.23) в пункте **Category** мы задаем категории, в которых будет размещаться наше приложение: выбираем в качестве первой категории **Utilities** (Утилиты), а в качестве второй — **Entertainment** (Развлечения).

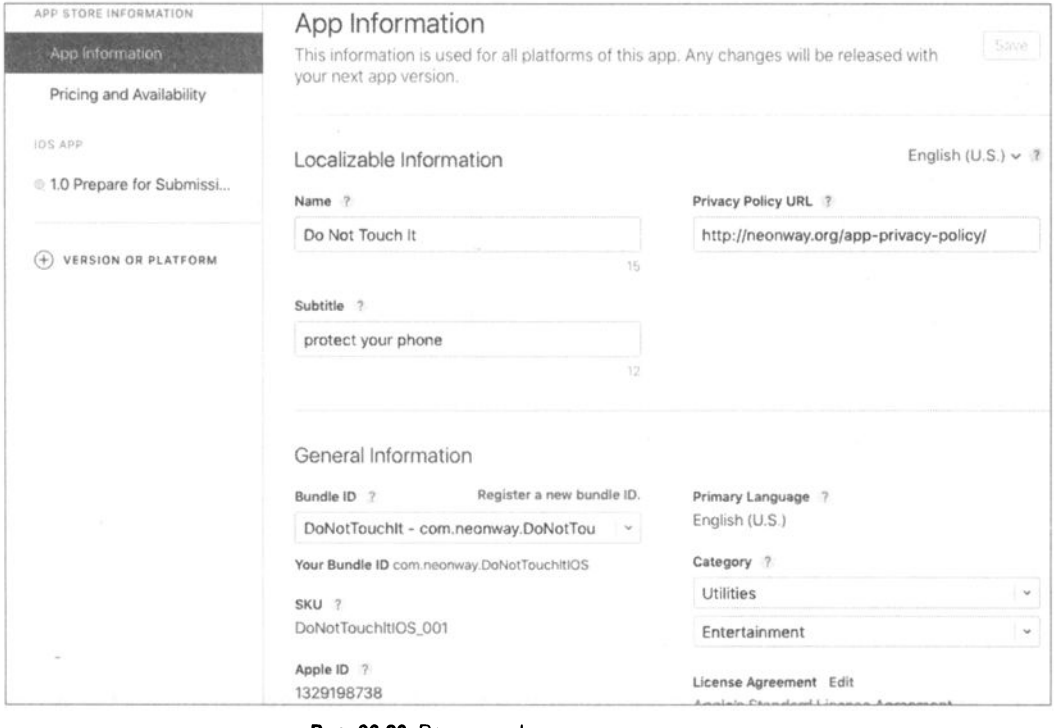


Рис. 66.23. Раздел информации о приложении

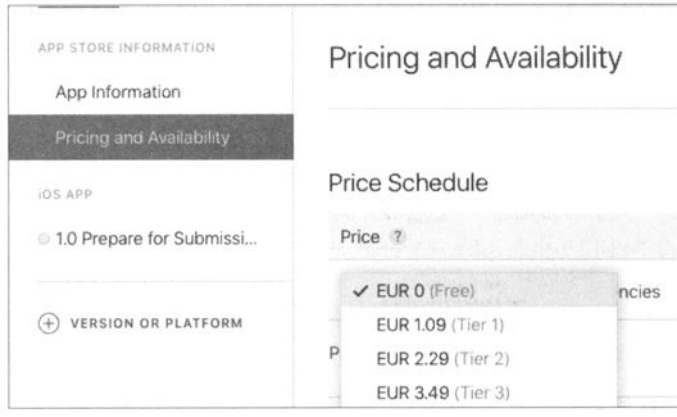


Рис. 66.24. Раздел для назначения цены приложения

В разделе **Pricing and Availability** (рис. 66.24) можно назначить приложению цену. Наше приложение предполагается сделать бесплатным, поэтому выбираем пункт **EUR 0 (Free)**.

> ***Tier (Ярус) — специальная единица валют Apple***
>
> Чтобы абстрагироваться от валют, Apple ввела специальную единицу — Tier (Ярус). Всего этих единиц 95, и каждой из них соответствуют свои собственные значения валют. Назначая цену, разработчик выбирает подходящий ярус. Например, Tier 4 соответствует цене приложения 3,99 долларов, или 4,49 евро, или 299 рублей и т. д. Цену приложений можно назначать для долларов — в диапазоне от 0 до 999, для евро — в диапазоне от 0 до 1099,99 , а для рублей — от 0 до 74990. Apple в зависимости от текущих курсов валют время от времени актуализирует соответствующие ярусам ценовые значения. Информацию об актуальных значениях всех доступных валют можно получить на этой же странице, перейдя по ссылке **All Prices and Currencies** (Все цены и валюты).

Чтобы объективно оценить, сколько стоит ваше приложение, найдите в магазине подобные ему приложения и изучите их функциональные особенности и их стоимость.

Далее мы переходим в третий раздел — **Prepare for Submission** — и попадаем на страницу подготовки и описания приложения. Ввиду большого набора элементов страница на рис. 66.25 показана не полностью. Так же, как и в разделе **App Information**, на этой странице можно добавлять языки для перевода и локализации. Чтобы сделать это, нужно выбрать пункт **English (U.S.)** вверху справа и сменить язык на тот, на который вы хотите сделать перевод.

Рассмотрим некоторые из опций этой страницы:

- **App Previews and Screenshots** — тут размещаются до пяти снимков с экрана и до трех видео. Вместо снимков с экрана можно размещать и красивые постеры, которые могут привлечь внимание потенциальных клиентов. Для iPhone 5.5" эти изображения в портретной ориентации могут иметь разрешение 1242×2208, а для iPad — разрешение 2048×2732. Для альбомной ориентации разрешения остаются теми же, только в обратном порядке. Для iPhone 5.8" размещать снимки с экранов не обязательно. Видео для iPhone 5.5" в портретной ориентации могут иметь разрешение 1080×1920, а для iPad — 1200×1600. Для альбомной ориентации разрешения остаются теми же, только в обратном порядке. Продолжительность видео не должна превышать тридцати секунд;
- **Promotional Text** — тут вносится текст, нужный для проведения маркетинговых акций и отображения важных сообщений;

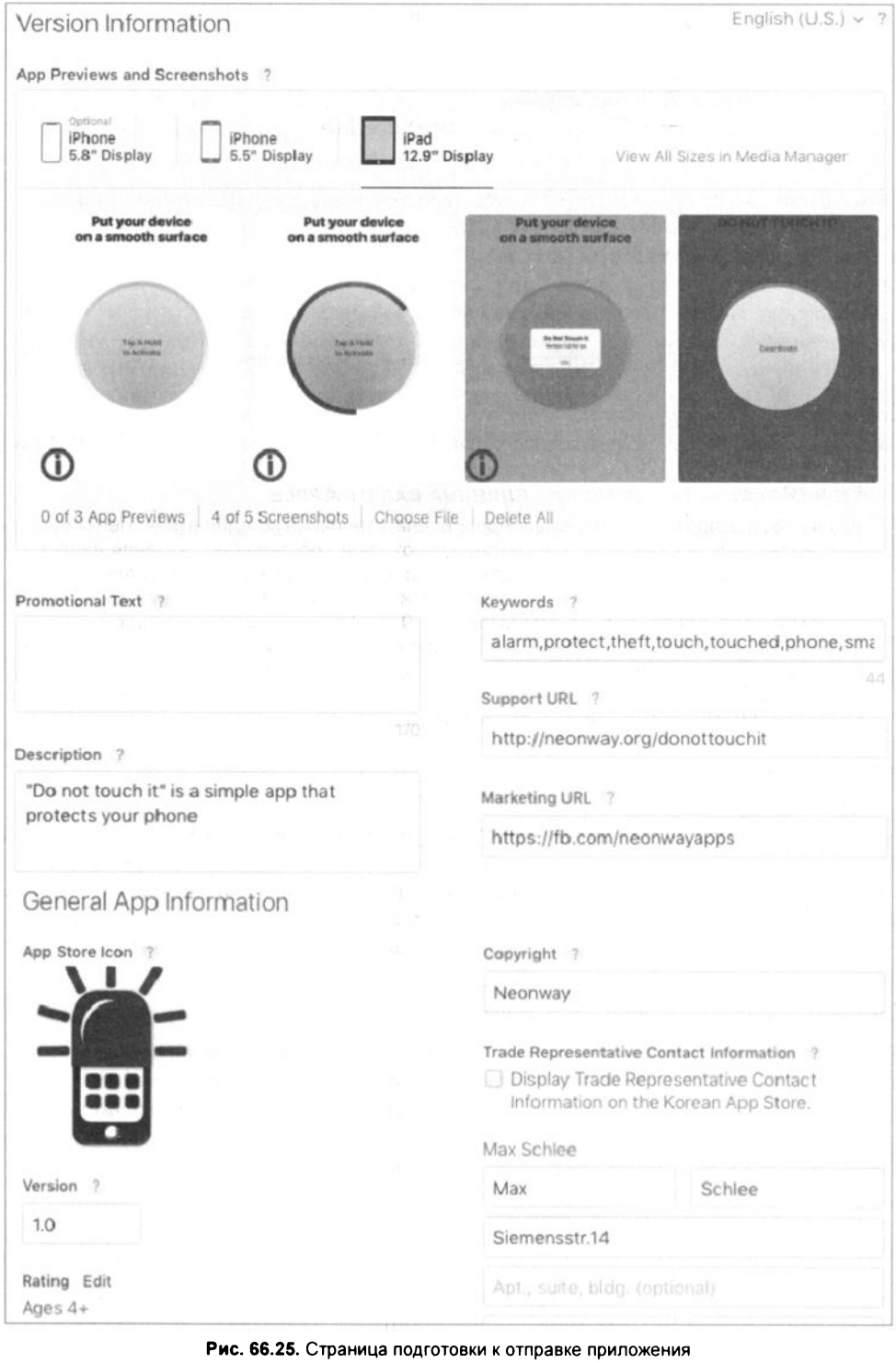


Рис. 66.25. Страница подготовки к отправке приложения

- **Keywords** — каждый магазин обладает своей собственной поисковой системой. И в этом пункте приводятся ключевые слова, по которым пользователи смогут найти ваше приложение. Слова необходимо отделять друг от друга запятыми без пробелов, и в общей сложности вся строка со словами не должна превышать 100 знаков. Будьте избирательны в словах, не спамьте и указывайте только те, которые имеют прямое отношение к тематике вашего приложения. Ни в коем случае не используйте слова, которые зарегистрированы в качестве торговых марок и товарных знаков;
- **Description** — тут приводится описательный текст вашего приложения. Максимальная длина этого текста — четыре тысячи знаков. Посетители магазина текст читают обычно редко и в основном только те, кто посмотрел сначала снимки с экранов и/или видео и, следовательно, заинтересовался приложением. В тексте самые важные — первые три предложения, потому что остальные могут быть в магазине скрыты от посетителей, пока те не нажмут кнопку для полного показа текста описания. Поэтому постарайтесь вложить в эти три предложения всю основную информацию или призыв к действию, словом, все, что может убедить посетителя загрузить или купить ваше приложение;
- в пункте **Support URL** указывается ссылка на веб-страницу поддержки вашего приложения. Желательно, чтобы такая была. В крайнем случае можно указать основной домен вашей страницы;
- в пункте **Marketing URL** указывается ссылка на страницу продвижения — это может быть, например, фан-сайт вашего приложения на Фейсбуке;
- в пункте **App Store Icon** размещается значок, который будет отображаться в магазине. Файл значка должен быть в формате PNG без слоев прозрачности и иметь разрешение 1024×1024. Для установки значка просто перетащите его из программы Finder (проводника файлов) в эту область;
- в пункте **Version** вы указываете номер версии вашего приложения. Проследите, чтобы он совпадал с номером версии, указанным в файле свойств *.plist (см. *разд. «Файл свойств iOS-приложений» главы 64*);
- в пункте **Rating**, нажав на кнопку **Edit**, вы отвечаете на вопросы, например, о том, содержит ли ваше приложение пропаганду алкоголя, элементы насилия и т. п. Это нужно, чтобы магазин мог лучше оценить проблемы, связанные с возможной публикацией вашего приложения. Ответить нужно на все вопросы, иначе приложение будет нельзя отправить на проверку в магазин.

И еще один очень важный момент, который нужно всегда иметь в виду, — после публикации приложения здесь практически ничего нельзя будет изменить: ни видео, ни снимки с экранов, ни полное описание и пр. Исключения составляют только **Promotional Text**, цена, ссылка на документ Privacy Policy, ссылка на страницу поддержки и ссылка на страницу продвижения. Все прочие изменения будут возможны только при выпуске обновлений приложения.

Загрузка и публикация приложения

Итак, мы подготовили страницу приложения — осталось теперь только загрузить на нее архив приложения. Архив приложения создается в интерактивной среде разработки Xcode. Поэтому нужно сначала создать для него проектный файл. Для этого просто выберите в Qt Creator опцию **Qt 5.10 for iOS** и нажмите на кнопку компиляции с молотком (рис. 66.26).

После компиляции нам нужно найти в каталоге теневой iOS-сборки файл проекта Xcode — он имеет вид *.xcodeproj. В нижней части рис. 66.27 показано его местонахождение.

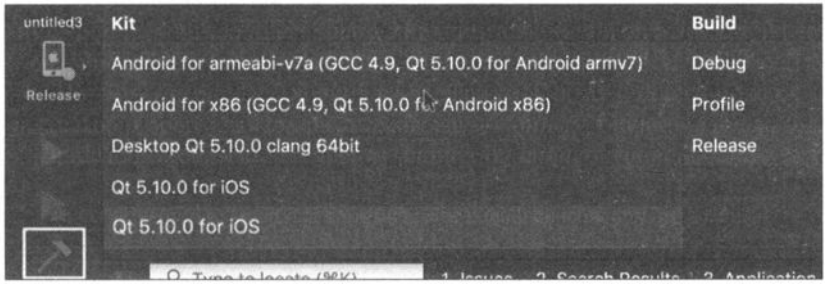


Рис. 66.26. Создание проектного файла для Xcode

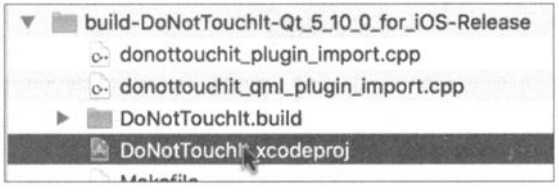


Рис. 66.27. Расположение проектного файла Xcode в каталоге теневой сборки Qt Creator

Щелкаем на файле DoNotTouch.xcodeproj двойным щелчком, и он откроется в Xcode. Выбираем в нем из меню **Product** (Продукт) пункт **Archive** (Архив), как показано на рис. 66.28.

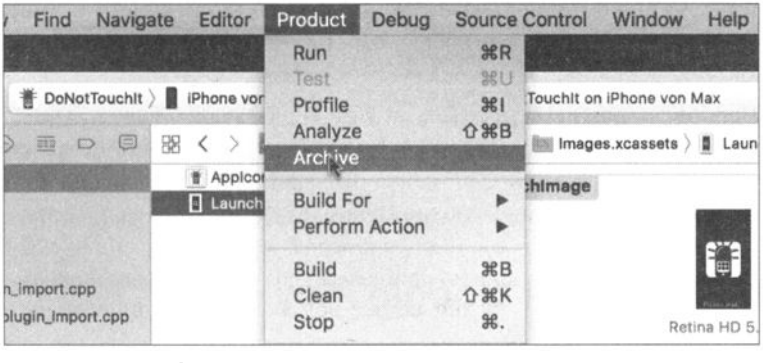


Рис. 66.28. Создание архива для отправки на страницу приложения

Как только процесс создания архива приложения будет завершен, перед вами откроется окно **Organizer** (рис. 66.29). Нажмем в нем на кнопку **Upload to App Store** (Загрузка на App Store) и дождемся окончания процесса загрузки.

Подождем пять минут и вернемся на страницу приложения, показанную на рис. 66.25. Найдем на ней поле **Build**, и если справа от него ничего не стоит, то подождите еще пару минут и обновите страницу. Как только там появится символ «+» (рис. 66.30, *слева*), нажмите на него и выберите из диалогового окна загруженный архив приложения. После этого поле **Build** примет вид, показаный на рис. 66.30, *справа*.

Теперь надо только лишь нажать в верхней части страницы приложения на кнопку **Submit for Review** (Отправить на проверку), ответить на пару вопросов, связанных с шифрованием и законным использованием авторских прав, и отправить приложение на проверку. После этого остается набраться терпения и ждать результатов. Можно также в это время заняться и вопросами продвижения приложения.

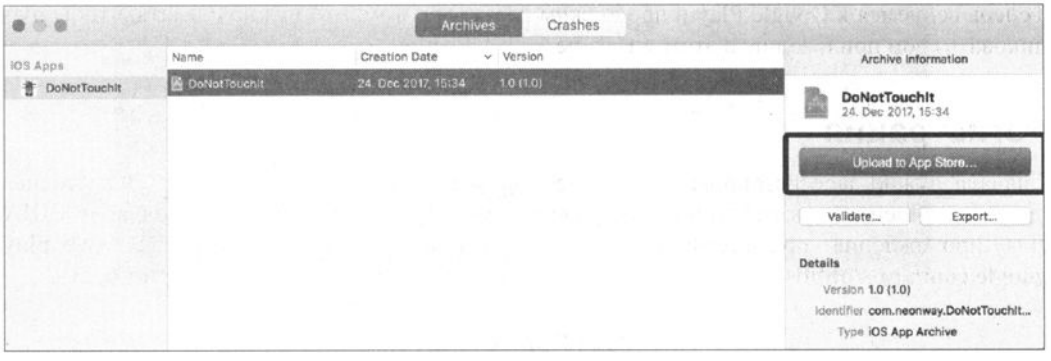


Рис. 66.29. Отправка архива на страницу приложения

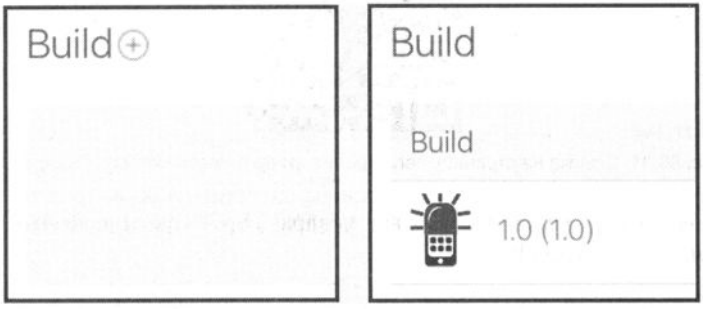


Рис. 66.30. Добавление архива на страницу приложения

Если вы не установили конкретную дату выпуска, то, в случае положительных результатов проверки, ваше приложение будет сразу же отправлено в магазин, а по электронной почте вы получите уведомление об этом. С момента отправки приложения и до его фактического появления в магазине может пройти от одного часа до суток.

Этапы работы для Google Play

Бесспорно, Google Play является самым популярным и доходным магазином для Android-приложений, но наряду с ним существует и целый ряд альтернативных магазинов, в которых вы можете тоже публиковать свои Android-творения. Об этом полезно знать. Некоторые из них представлены в табл. 66.1 со ссылками на порталы для разработчиков.

Таблица 66.1. *Альтернативные магазины Android-приложений*

Магазин	Веб-ссылка
Amazon Appstore	**https://developer.amazon.com**
Яндекс.Store	**https://developer.store.yandex.com**
SlideME	**https://slideme.org/user/register**
GetJar	**http://developer.getjar.mobi**
BlackBerry World	**https://appworld.blackberry.com**
Samsung Apps	**https://samsungapps.com**

Теперь вернемся к Google Play и продолжим разговор о том, что нужно сделать, чтобы публиковать свои приложения в этом магазине.

Регистрация

Сначала нужно зарегистрировать учетную запись разработчика Google Play. В настоящее время за лицензию разработки одноразово взимается плата в размере 25 долларов США. Для прохождения процедуры регистрации откройте в браузере ссылку: **www.play.google.com/apps/publish/** или воспользуйтесь QR-кодом, представленным на рис. 63.31.

Рис. 66.31. Ссылка на страницу регистрации аккаунта разработчика Google Play

Если регистрация прошла успешно, то я вас поздравляю — вы только что стали разработчиком приложений для Google Play.

Создание страницы приложения

Чтобы создать страницу приложения на Google Play, перейдите по веб-ссылке: **https://play.google.com/apps/publish/** — перед вами откроется страница, показанная на рис. 66.32.

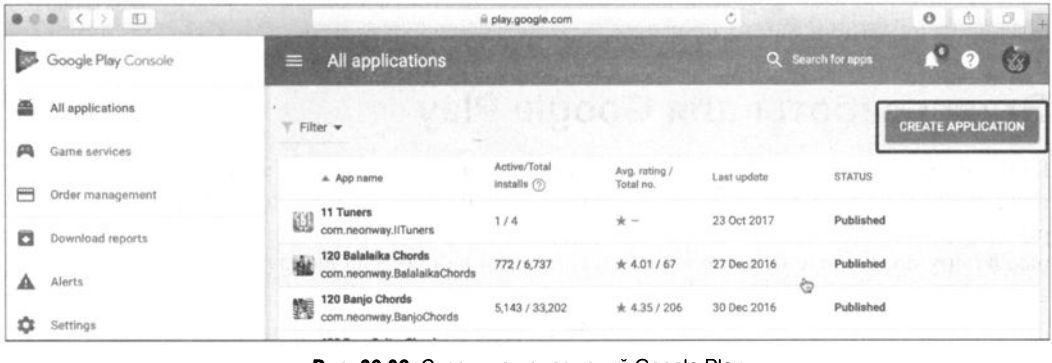


Рис. 66.32. Страница приложений Google Play

Нажмите на кнопку **CREATE APPLICATION** (Создать приложение), которая находится вверху справа, — откроется страница формуляра (рис. 66.33). Укажите в ней язык, который будет установлен по умолчанию: **English (United States)**, и внесите имя приложения в поле **Title**. Завершите создание страницы приложения, нажав на кнопку **CREATE**.

Итак, страница приложения создана (рис. 66.34). С левой ее стороны можно видеть большое количество разделов. Разделы, помеченные значком треугольника с восклицательным знаком, должны быть заполнены в обязательном порядке.

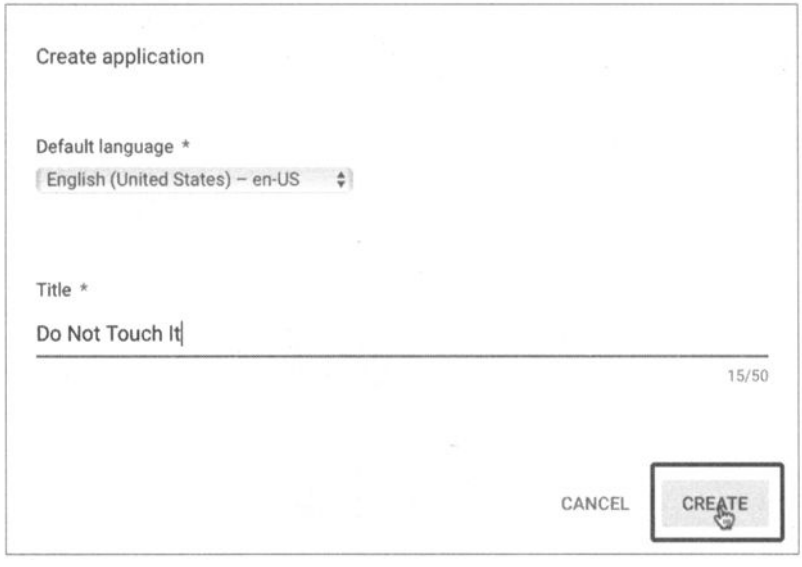


Рис. 66.33. Формуляр для создания новой страницы приложения

На рис. 66.34 открыт для заполнения раздел **Store listing** (Список для магазина) — вносим в него короткое (**Short description**) и полное (**Full description**) описание приложения. Максимальная длина текста для полного описания — четыре тысячи знаков. Страницы приложения можно снабжать переводами на другие языки. Для этого нужно их добавить нажатием на кнопку **Manage Translations** в верхнем правом углу окна.

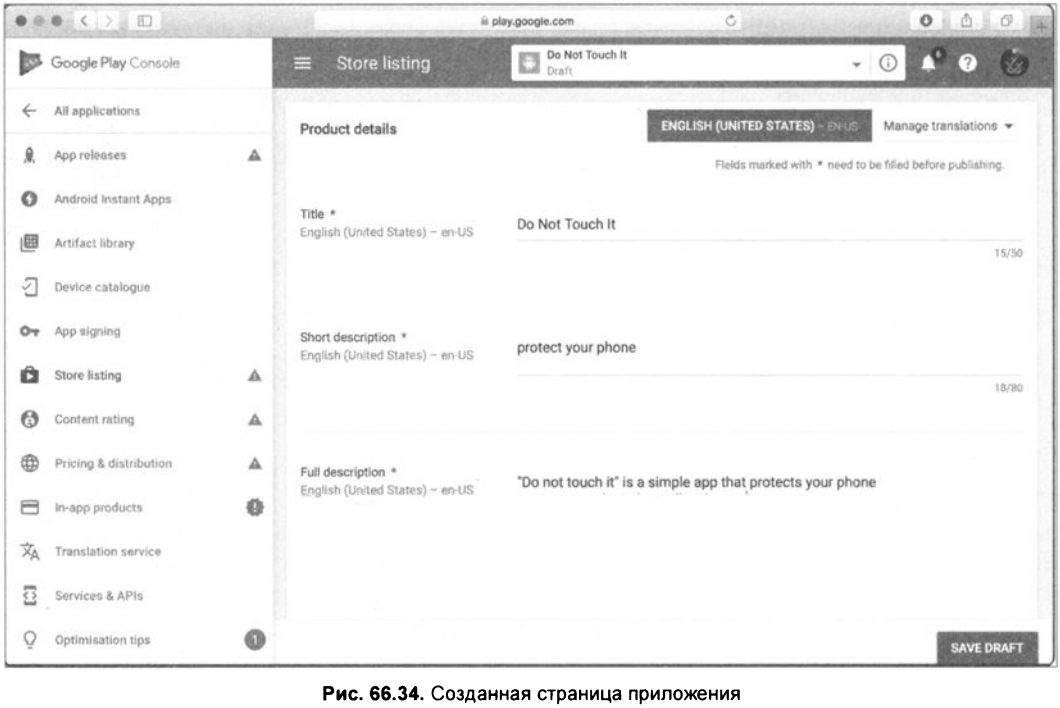


Рис. 66.34. Созданная страница приложения

Прокручиваем страницу вниз и вносим снимки с экранов (рис. 66.35). Для смартфонов (**PHONE**) можно назначить те же самые разрешения изображений, что мы назначили в App Store для iPhone 5.5", а для планшетов (**TABLET**) — разрешения изображений для iPad. Желательно загрузить хотя бы одно изображение для 10-дюймового устройства — это нужно, чтобы сделать приложение доступным и для этих устройств.

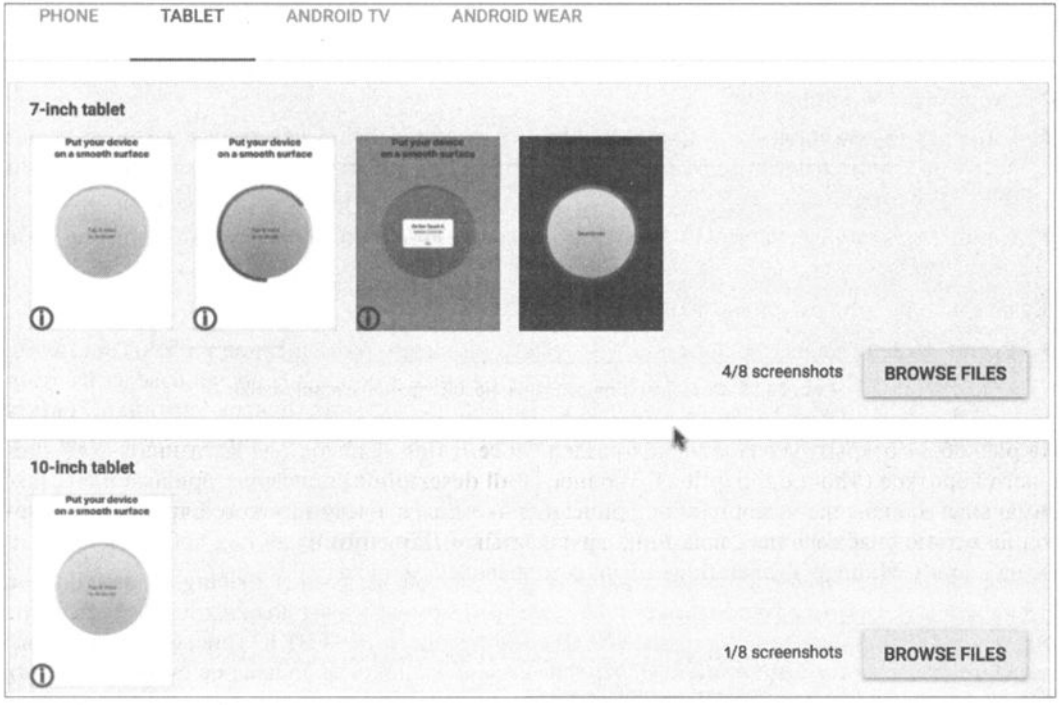


Рис. 66.35. Область загрузки снимков с экрана

Прокручиваем страницу вниз еще дальше. В области **Hi-res icon** (рис. 66.36, *слева*) нужно поместить значок приложения. Файл значка должен быть в формате PNG, иметь разрешение 512×512 пикселов и может содержать слои прозрачности. Просто перетащите его из файлового обозревателя в эту область.



Рис. 66.36. Области загрузки значка приложения (*слева*) и картинки «шапки» страницы (*справа*)

Создайте также в области **Feature graphic** (рис. 66.36, *справа*) картинку, которая будет отображаться в «шапке» страницы приложения. Файл картинки может быть в формате PNG либо JPG и должен иметь разрешение 1024×500 пикселов. Так как эту картинку будет очень хорошо видно, то к ней полезно добавить какой-нибудь короткий поясняющий текст о приложении либо призыв к действию.

Далее на странице следуют пункты, которые нужно обязательно заполнить:

- **Application Type** (Тип приложения) — выбор предоставляется из двух альтернатив: **Game** (Игра) или **Application** (Приложение), и нам нужно выбрать одну. В нашем случае мы выбираем **Application**;
- **Category** (Категория) — из большого списка нужно выбрать категорию, которая лучше всего подходит вашему приложению. В нашем случае мы выбираем категорию **Tools** (Инструменты);
- **Email** (электронная почта) — укажите адрес электронной почты, по которому с вами могут связаться по вопросам, относящимся к приложению.

Желательно на этой странице также указать:

- **Promo Video** (Видеопродвижение) — ссылка на видео, размещенное на YouTube, которое демонстрирует возможности вашего приложения. Кнопка для воспроизведения этого видео всегда отображается в «шапке» страницы приложения (поверх картинки **Feature graphic**). Их посетители магазина смотрят очень часто, поэтому пользуйтесь этой возможностью;
- **Website** (Веб-страница приложения) — ссылка на страницу приложения. Это поможет наладить лучшую связь с вашими потенциальными клиентами.

Выберем теперь в левой стороне страницы (см. рис. 66.34) раздел **Pricing & distribution** (Цена и сбыт). И тут сразу наступает очень важный момент — вы должны для себя решить, будет ли ваше приложение платным (**PAID**) или бесплатным (**FREE**) (рис. 66.37). Если вы сделаете приложение бесплатным, то это навсегда, и вы никогда больше не сможете сделать его платным. Это отличительная черта магазина Google Play, в других магазинах подобного ограничения нет.

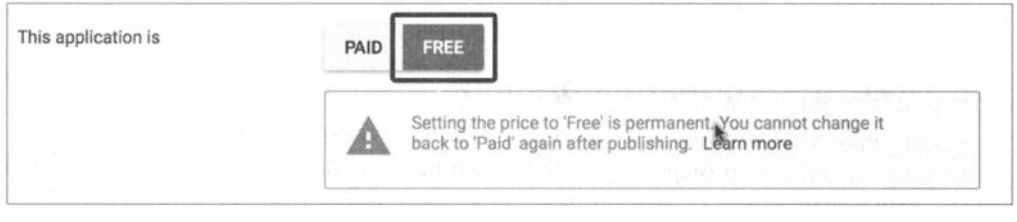


Рис. 66.37. Установка цены приложения

Теперь прокручиваем страницу дальше вниз до пункта **Countries** (Страны). Здесь в окне показан большой список из 142 стран, для которых мы можем сделать наше приложение доступным либо недоступным (рис. 66.38). В «шапке» этого списка выбираем опцию **Available** (Доступно) и тем самым делаем наше приложение доступным для всех стран, указанных в списке.

Далее на этой странице следуют пункты, которые нужно тоже обязательно заполнить:

- **Primary Child-Directed** (Предназначено для детей) — с разработкой приложений для детей до 13 лет связана масса нюансов. Например, приложения не должны содержать веб-ссылок, запрашивать персональные данные и многое другое. Поэтому, если вы не

разрабатываете приложения специально для детей и далеки от этой темы, то, во избежание неприятных сюрпризов, поставьте флажок **No** (Нет);

- **Contains ads** (Содержит рекламу) — если ваше приложение содержит рекламу, поставьте флажок **Yes** (Да), в противном случае — **No** (Нет);
- **Content guidelines** (Рекомендации к содержанию) — содержание страницы должно соответствовать требованиям магазина, и если это действительно так, поставьте флажок у этого пункта. Если вы в чем-то на этот счет сомневаетесь, перейдите по имеющейся рядом с этим пунктом ссылке **Android Content Guidelines**, чтобы ознакомиться с этими требованиями;
- **US export laws** (Закон экспорта США) — в основном это касается сильных методов шифрования данных, которые могут содержаться или использоваться в вашем приложении. Для того чтобы получить более подробную информацию об этом пункте, нажмите ссылку **Learn More** (Узнать больше). Если никаких противоречий не найдено, то поставьте флажок слева от этого пункта.

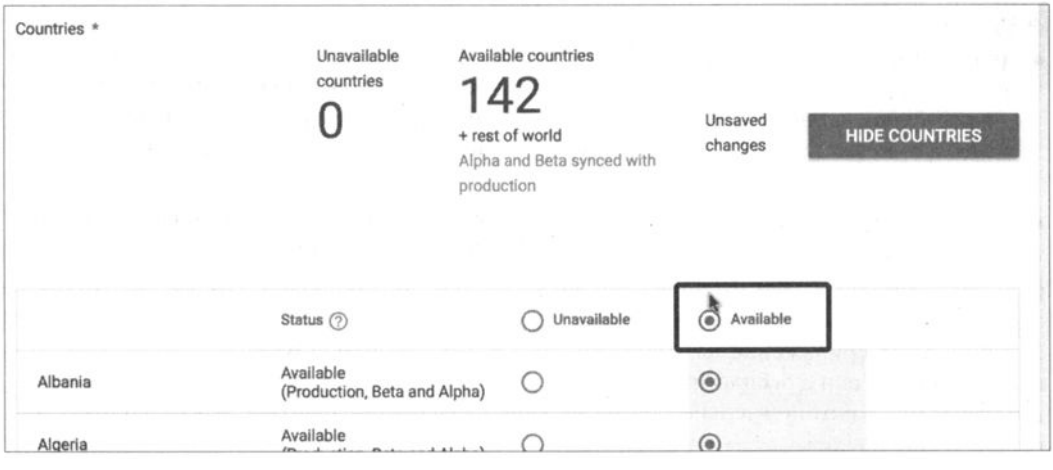


Рис. 66.38. Выбор доступных для приложения стран

Создание электронной подписи

Для Google Play электронную подпись можно создать прямо из Qt Creator. Для этого откройте в нем проект, например наш Do Not Touch It, и перейдите в левом столбце опций на вкладку **Projects** (рис. 66.39). Выберите слева в разделе **Build & Run** опцию **Build**. Затем найдите в области **Build Steps** в правой стороне окна строку **Build Android APK** и нажмите справа от нее кнопку **Details**.

В открывшемся формуляре найдите текстовое поле **Keystore** и нажмите справа от него кнопку **Create** — это приведет к открытию диалогового окна создания подписи (рис. 66.40). Как можно видеть, это окно состоит из двух разделов: **Keystore** (Ключ) — слева и **Certificate** (Сертификат) — справа.

Задайте в полях **Password** и **Retype Password** раздела **Keystore** пароль для ключа. В области **Certificate Distinguished Names** укажите свои данные и/или данные своей организации и поставьте в разделе **Certificate** флажок **Use Keystore password**, чтобы не создавать отдельного пароля для сертификата. Все остальные данные можете оставить по умолчанию, а поля, в которых сомневаетесь, — оставьте пустыми.

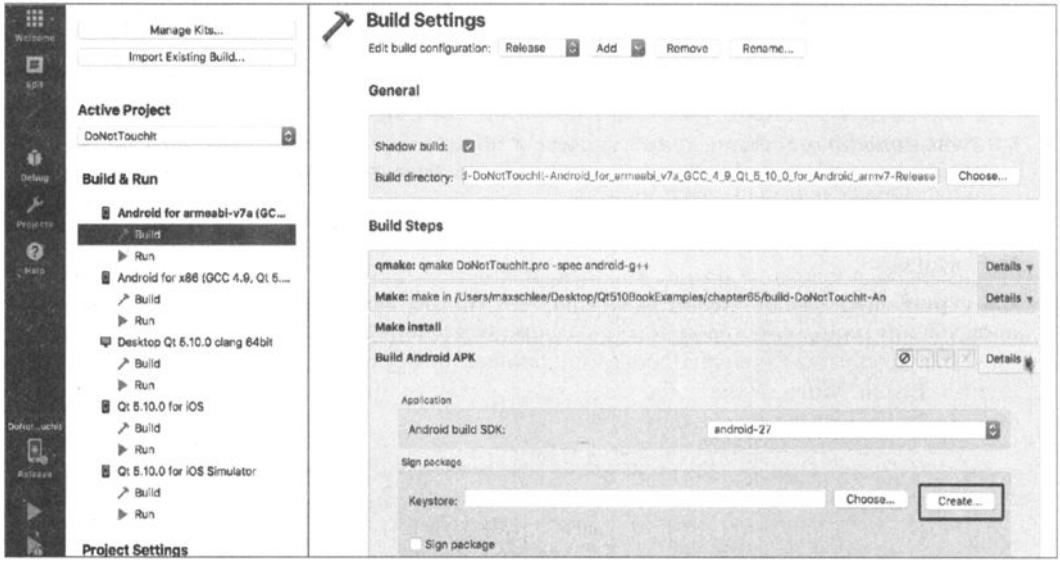


Рис. 66.39. Кнопка создания электронной подписи в Qt Creator

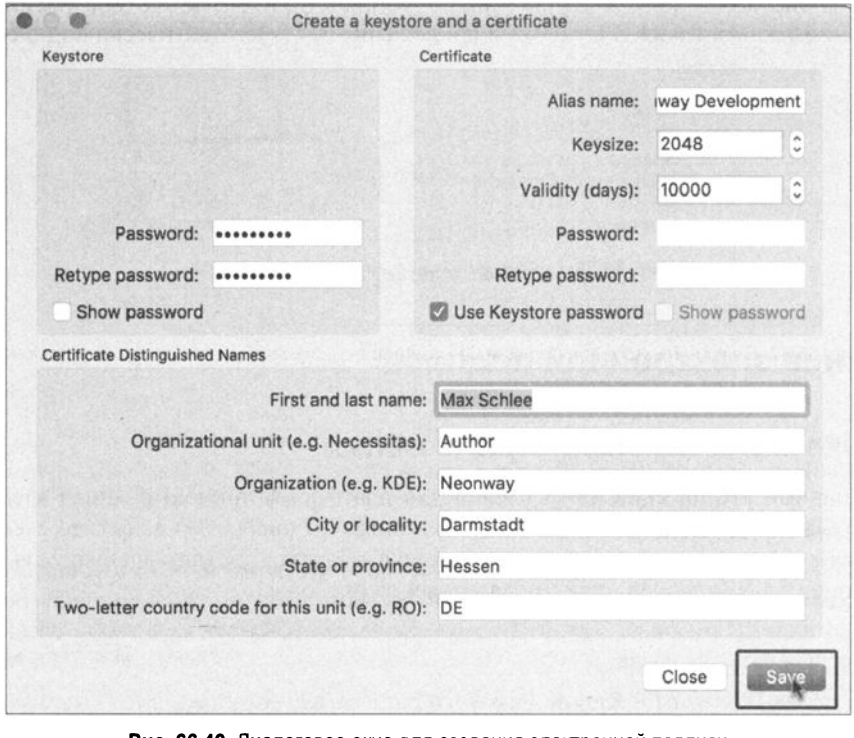


Рис. 66.40. Диалоговое окно для создания электронной подписи

В завершение нажмем на кнопку **Save** и сохраним файл подписи в надежном месте — так, что бы никто не мог у вас его потом скопировать. Хранение электронной подписи в строгой секретности является лучшей гарантией безопасности для ваших приложений.

> ***Будьте бдительны с файлом подписи и хорошо запомните его пароль!***
> Очень важно не потерять файл подписи и не забыть его пароль, иначе вы не сможете больше выпускать обновления к приложениям, которые были им подписаны. Создание нового файла подписи вам не поможет.

Теперь перейдем к загрузке файла подписи. На рис. 66.39 слева от кнопки **Create**, которую мы нажимали для создания файла подписи, находится кнопка **Choose**. Нажмите на нее и откройте из появившегося диалогового окна созданный нами файл. В процессе открытия этого файла вам потребуется ввести пароль (рис. 66.41) и нажать на кнопку **OK**.



Рис. 66.41. Окно ввода пароля для активации электронной подписи

После удачного ввода пароля область **Sign package** в Qt Creator станет выглядеть так, как показано на рис. 66.42. Обратите внимание на установленный флажок **Sign package** — теперь создаваемые для Google Play пакеты приложения будут автоматически подписываться.

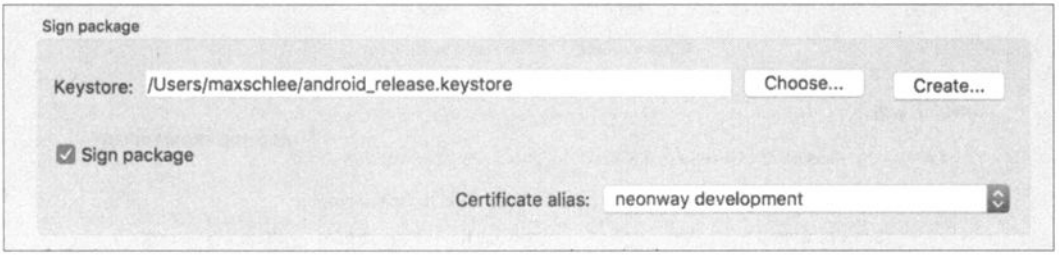


Рис. 66.42. Секция электронной подписи Qt Creator с активированной электронной подписью

Загрузка и публикация приложения

Откомпилируйте приложение из Qt Creator для платформы Android с вашей электронной подписью. Затем на странице создания приложения (см. рис. 66.34) выберите слева раздел **App releases** (Выпуск приложений) — откроется область, предназначенная для выпуска приложений. Выберите в ней пункт, показанный на рис. 66.43.

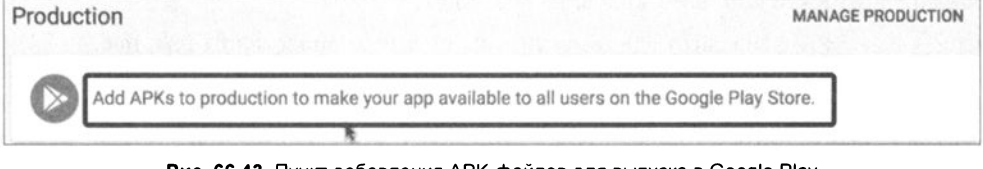


Рис. 66.43. Пункт добавления APK-файлов для выпуска в Google Play

После этого откроется страница с кнопкой **CREATE RELEASE** (Создать выпуск) — нажмите на нее. В результате откроется окно (рис. 66.44), предлагающее воспользоваться новой системой подписи приложений. Чтобы использовать электронную подпись Qt Creator, нажмите на кнопку **OPT OUT** (Отказаться) и подтвердите в диалоговом окне, которое появится следом, ваше решение кнопкой **CONFIRM** (Подтвердить).

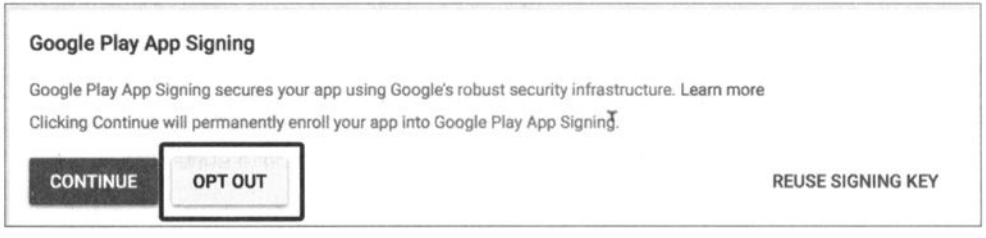


Рис. 66.44. Использование электронной подписи, созданной в Qt Creator

Теперь перед вами откроется страница с областью, куда можно «забросить» подписанный APK-файл с приложением (рис. 66.45). Можно также воспользоваться кнопкой **BROWSE FILES** (Просмотр файлов), чтобы найти этот файл на диске.

APK-файлы — это ZIP-архивы

Файлы в формате APK (Android Package, Android-пакет) являются архивами формата ZIP и могут быть распакованы программой 7-Zip.

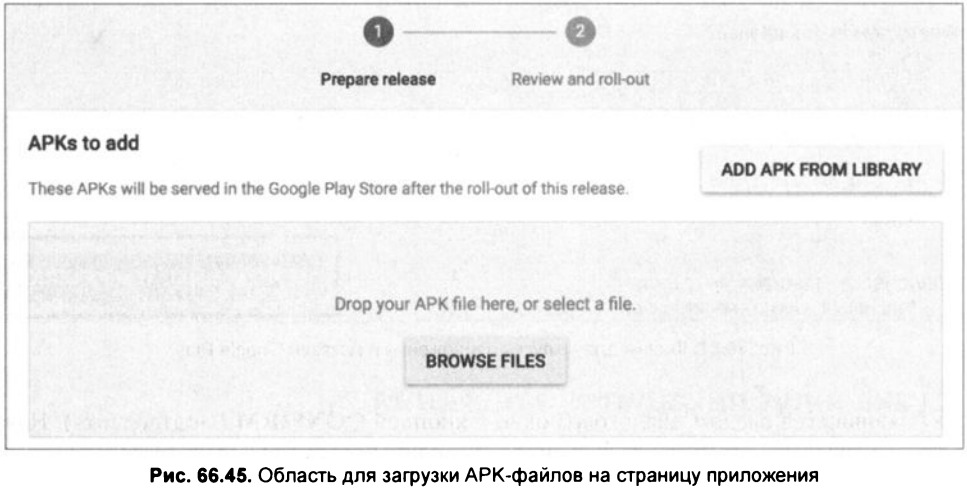


Рис. 66.45. Область для загрузки APK-файлов на страницу приложения

Созданный и подписанный APK-файл можно найти в каталоге теневой Android-сборки проекта — на рис. 66.46 в самом низу показано его местонахождение. Перетащим его в область APK-файлов **APKs to add**, показанную на рис. 66.45.

Теперь, когда APK-файл загружен, выберите на странице приложения (см. рис. 66.34) раздел **Content rating** и пройдите в нем процедуру оценки приложения. Для чего ответьте на все поставленные магазином вопросы.

И в завершение нажмите внизу кнопку **START ROLL-OUT TO PRODUCTION** (Выход на стадию производства) (рис. 66.47) — теперь приложение от появления в магазине отделяет

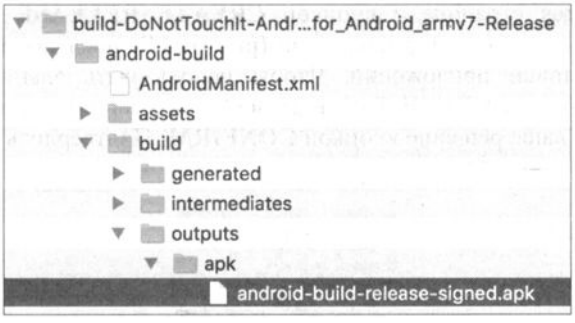


Рис. 66.46. Расположение APK-файла в каталоге теневой сборки Qt Creator

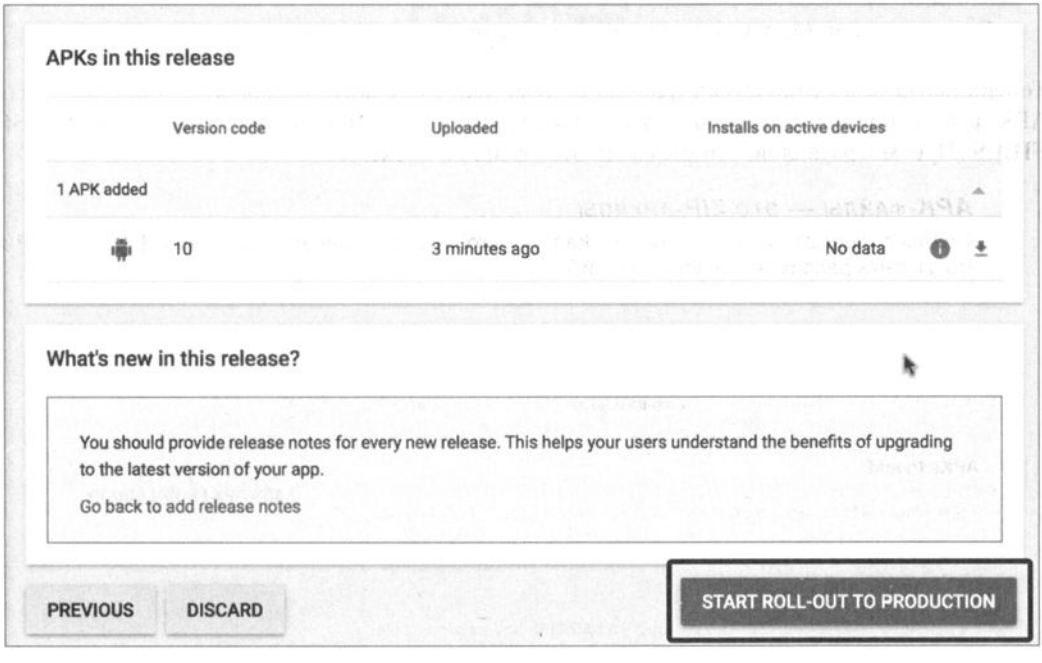


Рис. 66.47. Кнопка для выпуска приложения в магазин Google Play

только появившееся следом диалоговое окно с кнопкой **CONFIRM** (Подтвердить). Нажмите на нее, и через некоторое время вы увидите свое приложение в Google Play.

Резюме

Для того чтобы публиковать приложения в магазинах App Store и Google Play, нужна лицензия разработчика.

Мы научились настраивать реальные iOS-устройства для запуска на них своих приложений. Освоили навыки создания сертификатов, необходимых для iOS-разработки.

Разобрались с процессом отправки приложений в магазины App Store и Google Play. Для этого нам понадобилось научиться создавать электронные подписи, создавать страницы

приложений и наполнять их необходимым содержанием. Мы сформировали файлы архивов и отправили их на страницу приложения. С этой страницы мы затем опубликовали приложение в магазинах.

Свои отзывы и замечания по этой главе вы можете оставить по ссылке: **http://qt-book.com/ru/66-510/** или с помощью следующего QR-кода (рис. 66.48):



Рис. 66.48. QR-код для доступа на страницу комментариев к этой главе



ПРИЛОЖЕНИЯ

Приложение 1. Настройка среды для работы над Qt-приложениями

Приложение 2. Таблица простых чисел

Приложение 3. Таблицы семибитной кодировки ASCII

Приложение 4. Описание архива с примерами

ПРИЛОЖЕНИЕ 1

Настройка среды для работы над Qt-приложениями

Это приложение содержит описание необходимых действий для настройки Qt на операционных системах Mac OS X, Windows и Linux (Ubuntu).

Настройка среды для Mac OS X

1. Загрузите Xcode из магазина приложений Mac App Store.
2. Запустите Xcode и догрузите запрошенные им компоненты.
3. Загрузите со страницы: **http://developer.apple.com/download/more/** установочный пакет Command Line Tools for Xcode.
4. Установите загруженный пакет Command Line Tools for Xcode.
5. Запустите терминал и подайте команду:

   ```
   sudo xcode-select -switch /Applications/Xcode.app"
   ```

6. Загрузите Qt для Mac OS X со страницы: **http://qt.io/download-open-source/** или **http://qt.io/download/**, нажав на кнопку **Download Now** или **Buy Qt**. Загруженный установочный пакет должен иметь расширение dmg. Если что-то не получилось, используйте в качестве запасной ссылку: **http://download.qt.io/archive/qt/**. На этой странице вы всегда сможете выбрать из архива нужную вам версию Qt.
7. Запустите процесс установки Qt и нажмите кнопку **Continue**, чтобы перейти к следующему шагу.
8. Если у вас уже есть учетная запись на портале **http://qt.io**, внесите в форму ваши данные и нажмите кнопку **Next**, если нет, то создайте ее. Или пропустите этот шаг нажатием кнопки **Skip**. Вы увидите окно приветствия — нажмите кнопку **Continue**, чтобы перейти к следующему шагу.
9. Выберите каталог, в который хотите установить Qt, и нажмите кнопку **Continue**, чтобы перейти к следующему шагу.
10. Выберите из предложенного списка (рис. П1.1) компоненты Qt, которые вы хотите установить на свой компьютер. Для разработки программ на Mac OS X и компиляции примеров этой книги должна быть обязательно выбрана опция **macOS**. Для разработки мобильных приложений для iOS и Android (см. *часть IX «Мобильные приложения и Qt»*) необходимо выбрать опции **iOS** и **Android ARMv7**. Рекомендуется также выбрать опцию **Android x86** — это может пригодиться для быстрого запуска приложений на Android-симуляторе. Выделите необходимые дополнительные модули (можно воспользоваться выбором, показанным на рис. П1.1).

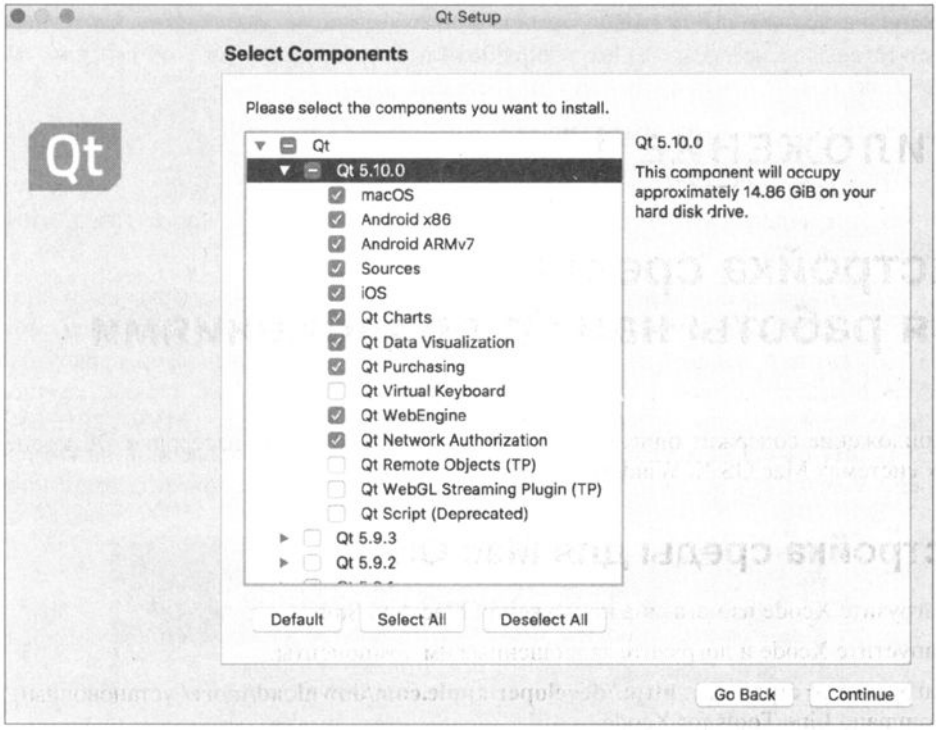


Рис. П1.1. Окно выбора компонентов Qt для установки на Mac OS X

Если прокрутить список в самый конец, то можно в секции **Tools** увидеть **Qt Creator** (см. *главу 47*) — эта интерактивная среда разработки IDE всегда устанавливается в обязательном порядке, а также дополнительные инструменты **Qt 3D Studio** (см. *главу 61*) и **Qt Install Framework**, которые можно установить по желанию.

Теперь убедитесь в том, что у вас на диске компьютера достаточно места для проведения установки (в нашем случае понадобится около 15 Гбайт), и нажмите кнопку **Continue**, чтобы перейти к следующему шагу.

11. Ответьте утвердительно на вопрос о лицензионном соглашении и нажмите кнопку **Continue**.
12. Нажмите на кнопку **Install** и дождитесь окончания процесса установки.

Настройка среды для Windows

1. В процессе инсталляции под Windows установочная программа Qt предложит компилятор MinGW C++, который содержится в пакете установки. Вы можете поставить и использовать его — для примеров этой книги и дальнейшей работы с Qt этого компилятора вам будет вполне достаточно. Если же вы привыкли использовать Visual C++ или хотите использовать его дополнительно к MinGW C++, то позаботьтесь о том, чтобы Visual Studio была установлена на компьютере до установки Qt. Получить Visual Studio можно по этой ссылке: **https://www.visualstudio.com/downloads/**.
2. Теперь загрузите Qt для Windows со страницы: **http://qt.io/download-open-source/** или **http://qt.io/download/**, нажав на кнопку **Download Now** или **Buy Qt**. Загруженный файл

установки должен иметь расширение exe. Если что-то не получилось, то используйте в качестве запасной ссылку: **http://download.qt.io/archive/qt/**. На этой странице вы всегда сможете выбрать из архива нужную вам версию Qt.

3. Запустите загруженный файл установки Qt и нажмите кнопку **Next**.
4. Выполните действия шага 8, описывающего установку для Mac OS X.
5. Выберите каталог, в который хотите установить Qt, и нажмите кнопку **Next**, чтобы перейти к следующему шагу.
6. Выберите из списка, показанного на рис. П1.2, компоненты Qt, которые вы хотите установить на свой компьютер. Для разработки программ на Windows и компиляции примеров этой книги должна быть выбрана опция **MinGW 5.3.0 32 bit**. Затем прокрутите список и обязательно выберите в разделе **Tools** опцию для установки компилятора **MinGW 5.3.0**. Опции **MSVC 2013 64-bit**, **MSVC 2015 32-bit**, **MSVC 2015 64-bit** и **MSVC 2017 64-bit** вы можете выбрать в зависимости от установленной на компьютере версии Visual C++, если его на компьютере нет, то тогда не выбирайте эти опции. Для

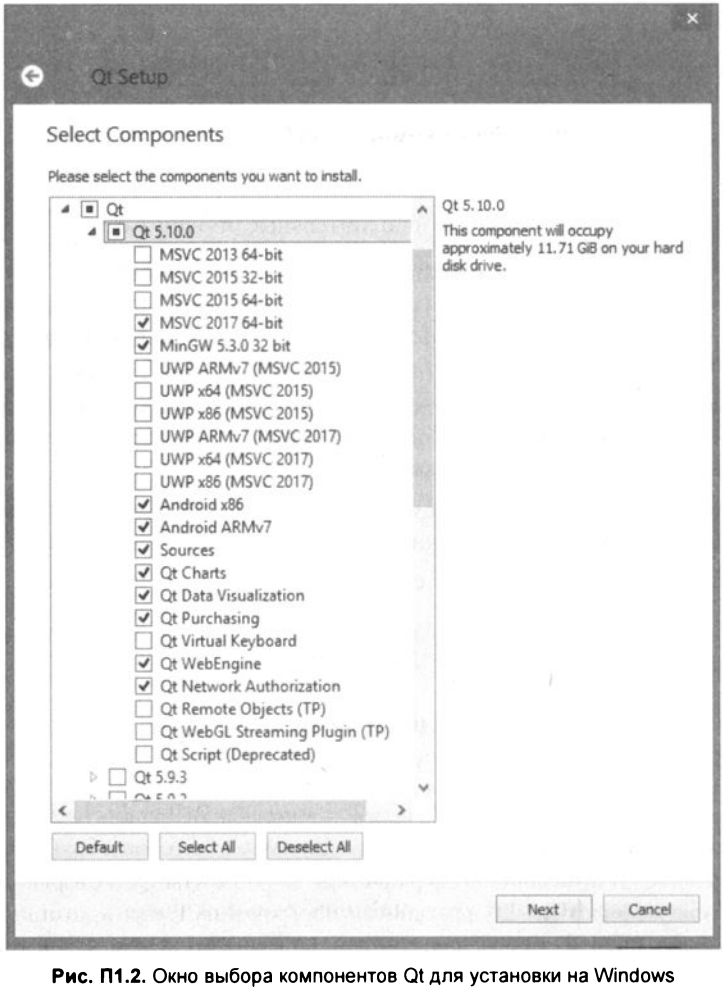


Рис. П1.2. Окно выбора компонентов Qt для установки на Windows

разработки мобильных приложений для Android (см. *часть IX «Мобильные приложения и Qt»*) необходимо выбрать опцию **Android ARMv7**. Можно также дополнительно выбрать и опцию **Android x86** — это может пригодиться для быстрого запуска приложений на Android-симуляторе. Выделите необходимые вам дополнительные модули (можно воспользоваться выбором, показанным на рис. П1.2).

Если прокрутить список в самый конец, то можно в секции **Tools** увидеть **Qt Creator** (см. *главу 47*) — эта интерактивная среда разработки IDE всегда устанавливается в обязательном порядке, а также дополнительные инструменты **Qt 3D Studio** (см. *главу 61*) и **Qt Install Framework**, которые можно установить по желанию.

Теперь убедитесь в том, что у вас на диске компьютера достаточно места для проведения установки (в нашем случае понадобится около 12 Гбайт), и нажмите кнопку **Next**.

7. Ответьте утвердительно на вопрос о лицензионном соглашении и нажмите кнопку **Next**.
8. Выберите название группы ярлыков Qt для стартового меню или оставьте предложенный программой вариант. Нажмите кнопку **Next**.
9. Нажмите на кнопку **Install** и дождитесь окончания процесса установки.

Настройка среды для Ubuntu Linux

1. Установите из командной оболочки компилятор GNU C++:

```
sudo apt-get install g++
```

2. Установите из командной оболочки дополнительные библиотеки:

```
sudo apt-get install libgl1-mesa-dev libglu1-mesa-dev libpulse-dev build-
essential libfontconfig1
```

3. Теперь загрузите Qt для Linux со страницы: **http://qt.io/download-open-source/** или **http://qt.io/download/**, нажав на кнопку **Download Now** или **Buy Qt**. Загруженный файл установки должен иметь расширение run. Если что-то не получилось, то используйте в качестве запасной ссылку: **http://download.qt.io/archive/qt/**. На этой странице вы всегда сможете выбрать из архива нужную вам версию Qt.
4. Сделайте из командной оболочки загруженный установочный файл запускаемым. Для этого перейдите в каталог с загруженным инсталляционным файлом и подайте команду:

```
chmod u+x <qtinstaller>.run
```

5. Запустите из командной оболочки процесс установки Qt:

```
sudo ./<qtinstaller>.run
```

В открывшемся окне нажмите на кнопку **Next**.

6. Выполните действия шага 8, описывающего установку для Mac OS X.
7. Выберите каталог, в который хотите установить Qt, и нажмите кнопку **Next**, чтобы перейти к следующему шагу.
8. Выберите из списка, показанного на рис. П1.3, компоненты Qt, которые вы хотите установить на свой компьютер. Для разработки программ на Linux и компиляции примеров этой книги должна быть обязательно выбрана опция **Desktop gcc 64-bit**. Для разработки мобильных приложений Android (см. *часть IX «Мобильные приложения и Qt»*) необ-

ходимо выбрать опцию **Android ARMv7**. Рекомендуется также выбрать опцию **Android x86** — это поможет быстро запускать приложения на Android-симуляторе. Выделите необходимые дополнительные модули (можно воспользоваться выбором, показанным на рис. П1.3).

Если прокрутить список в самый конец, то можно в секции **Tools** увидеть **Qt Creator** (см. *главу 47*) — эта интерактивная среда разработки IDE устанавливается в обязательном порядке, а также дополнительный инструмент **Qt Install Framework**, предназначенный для создания установочных файлов, который можно установить по желанию.

Теперь убедитесь в том, что у вас на диске компьютера достаточно места для проведения установки (в нашем случае понадобится около 4 Гбайт), и нажмите кнопку **Next**, чтобы перейти к следующему шагу.

9. Ответьте утвердительно на вопрос о лицензионном соглашении и нажмите кнопку **Next**.

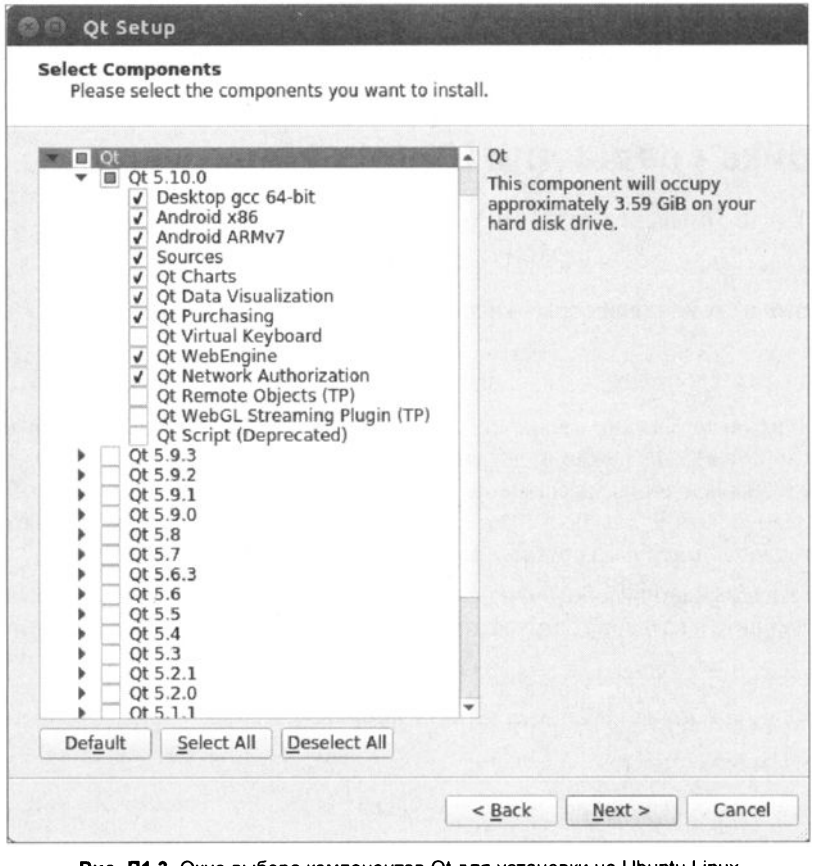


Рис. П1.3. Окно выбора компонентов Qt для установки на Ubuntu Linux

10. Нажмите на кнопку **Install** и дождитесь окончания процесса установки.

11. Теперь, если вы использовали для установки Qt путь по умолчанию, вы можете запустить Qt Creator из командной оболочки следующим образом:

```
sudo /opt/Qt/Tools/QtCreator/bin/qtcreator
```

Свои отзывы и замечания по этому приложению книги вы можете оставить по ссылке: **http://qt-book.com/ru/p1-510/** или с помощью следующего QR-кода (рис. П1.4):



Рис. П1.4. QR-код для доступа на страницу комментариев к этому приложению книги

ПРИЛОЖЕНИЕ 2

Таблица простых чисел

Далее в табл. П2.1 представлена таблица простых чисел.

Таблица П2.1. *Простые числа*

2	3	5	7	11	13	17	19	23	29	31	37	41	43
47	53	59	61	71	73	79	83	89	97	101	103	107	109
113	127	131	137	139	149	151	157	163	167	173	179	181	191
193	197	199	211	223	227	229	233	239	241	251	257	263	269
271	277	281	283	293	307	311	313	317	331	337	347	349	353
359	367	373	379	383	389	397	401	409	419	421	431	433	439
443	449	457	461	463	467	479	487	491	499	503	509	521	523
541	547	557	563	569	571	577	587	593	599	601	607	613	617
619	631	641	643	647	653	659	661	673	677	683	691	701	709
719	727	733	739	743	751	757	761	769	773	787	797	809	811
821	823	827	829	839	853	857	859	863	877	881	883	887	907
911	919	929	937	941	947	953	967	971	977	983	991	997	1009
1013	1019	1021	1031	1033	1039	1049	1051	1061	1063	1069	1087	1091	1093
1097	1103	1109	1117	1123	1129	1151	1153	1163	1171	1181	1187	1193	1201
1213	1217	1223	1229	1231	1237	1249	1259	1277	1279	1283	1289	1291	1297
1301	1303	1307	1319	1321	1327	1361	1367	1373	1381	1399	1409	1423	1427
1429	1433	1439	1447	1451	1453	1459	1471	1481	1483	1487	1489	1493	1499
1511	1523	1531	1543	1549	1553	1559	1567	1571	1579	1583	1597	1601	1607
1609	1613	1619	1621	1627	1637	1657	1663	1667	1669	1693	1697	1699	1709
1721	1723	1733	1741	1747	1753	1759	1777	1783	1787	1789	1801	1811	1823
1831	1847	1861	1867	1871	1873	1877	1879	1889	1901	1907	1913	1931	1933
1949	1951	1973	1979	1987	1993	1997	1999	2003	2011	2017	2027	2029	2039
2053	2063	2069	2081	2083	2087	2089	2099	2111	2113	2129	2131	2137	2141
2143	2153	2161	2179	2203	2207	2213	2221	2237	2239	2243	2251	2267	2269

Таблица П2.1 *(продолжение)*

2273	2281	2287	2293	2297	2309	2311	2333	2339	2341	2347	2351	2357	2371
2377	2381	2383	2389	2393	2399	2411	2417	2423	2437	2441	2447	2459	2467
2473	2477	2503	2521	2531	2539	2543	2549	2551	2557	2579	2591	2593	2609
2617	2621	2633	2647	2657	2659	2663	2671	2677	2683	2687	2689	2693	2699
2707	2711	2713	2719	2729	2731	2741	2749	2753	2767	2777	2789	2791	2797
2801	2803	2819	2833	2837	2843	2851	2857	2861	2879	2887	2897	2903	2909
2917	2927	2939	2953	2957	2963	2969	2971	2999	3001	3011	3019	3023	3037
3041	3049	3061	3067	3079	3083	3089	3109	3119	3121	3137	3163	3167	3169
3181	3187	3191	3203	3209	3217	3221	3229	3251	3253	3257	3259	3271	3299
3301	3307	3313	3319	3323	3329	3331	3343	3347	3359	3361	3371	3373	3389
3391	3407	3413	3433	3449	3457	3461	3463	3467	3469	3491	3499	3511	3517
3527	3529	3533	3539	3541	3547	3557	3559	3571	3581	3583	3593	3607	3613
3617	3623	3631	3637	3643	3659	3671	3673	3677	3691	3697	3701	3709	3719
3727	3733	3739	3761	3767	3769	3779	3793	3797	3803	3821	3823	3833	3847
3851	3853	3863	3877	3881	3889	3907	3911	3917	3919	3923	3929	3931	3943
3947	3967	3989	4001	4003	4007	4013	4019	4021	4027	4049	4051	4057	4073
4079	4091	4093	4099	4111	4127	4129	4133	4139	4153	4157	4159	4177	4201
4211	4217	4219	4229	4231	4241	4243	4253	4259	4261	4271	4273	4283	4289
4297	4327	4337	4339	4349	4357	4363	4373	4391	4397	4409	4421	4423	4441
4447	4451	4457	4463	4481	4483	4493	4507	4517	4523	4547	4549	4561	4567
4583	4591	4597	4603	4621	4637	4639	4643	4649	4651	4657	4663	4673	4679
4691	4703	4721	4723	4729	4733	4751	4759	4783	4787	4789	4793	4799	4801
4813	4817	4831	4861	4871	4877	4889	4903	4909	4919	4931	4933	4937	4943
4951	4957	4967	4969	4973	4987	4993	4999	5003	5009	5011	5021	5023	5039
5051	5059	5077	5081	5087	5099	5101	5107	5113	5119	5147	5153	5167	5171
5179	5189	5197	5209	5227	5231	5233	5237	5261	5273	5279	5281	5297	5303
5309	5323	5333	5347	5351	5381	5387	5393	5399	5407	5413	5417	5419	5431
5437	5441	5443	5449	5471	5477	5479	5483	5501	5503	5507	5519	5521	5527
5531	5557	5563	5569	5573	5581	5591	5623	5639	5641	5647	5651	5653	5657
5659	5669	5683	5689	5693	5701	5711	5717	5737	5741	5743	5749	5779	5783
5791	5801	5807	5813	5821	5827	5839	5843	5849	5851	5857	5861	5867	5869
5879	5881	5897	5903	5923	5927	5939	5953	5981	5987	6007	6011	6029	6037
6043	6047	6053	6067	6073	6079	6089	6091	6101	6113	6121	6131	6133	6143
6151	6163	6173	6197	6199	6203	6211	6217	6221	6229	6247	6257	6263	6269
6271	6277	6287	6299	6301	6311	6317	6323	6329	6337	6343	6353	6359	6361
6367	6373	6379	6389	6397	6421	6427	6449	6451	6469	6473	6481	6491	6521

Таблица П2.1 *(окончание)*

6529	6547	6551	6553	6563	6569	6571	6577	6581	6599	6607	6619	6637	6653
6659	6661	6673	6679	6689	6691	6701	6703	6709	6719	6733	6737	6761	6763
6779	6781	6791	6793	6803	6823	6827	6829	6833	6841	6857	6863	6869	6871
6883	6899	6907	6911	6917	6947	6949	6959	6961	6967	6971	6977	6983	6991
6997	7001	7013	7019	7027	7039	7043	7057	7069	7079	7103	7109	7121	7127
7129	7151	7159	7177	7187	7193	7207	7211	7213	7219	7229	7237	7243	7247
7253	7283	7297	7307	7309	7321	7331	7333	7349	7351	7369	7393	7411	7417
7433	7451	7457	7459	7477	7481	7487	7489	7499	7507	7517	7523	7529	7537
7541	7547	7549	7559	7561	7573	7577	7583	7589	7591	7603	7607	7621	7639
7643	7649	7669	7673	7681	7687	7691	7699	7703	7717	7723	7727	7741	7753
7757	7759	7789	7793	7817	7823	7829	7841	7853	7867	7873	7877	7879	7883
7901	7907	7919	7927	7933	7937	7949	7951	7963	7993	8009	8011	8017	8039
8053	8059	8069	8081	8087	8089	8093	8101	8111	8117	8123	8147	8161	8167
8171	8179	8191	8209	8219	8221	8231	8233	8237	8243	8263	8269	8273	8287
8291	8293	8297	8311	8317	8329	8353	8363	8369	8377	8387	8389	8419	8423
8429	8431	8443	8447	8461	8467	8501	8513	8521	8527	8537	8539	8543	8563
8573	8581	8597	8599	8609	8623	8627	8629	8641	8647	8663	8669	8677	8681
8689	8693	8699	8707	8713	8719	8731	8737	8741	8747	8753	8761	8779	8783
8803	8807	8819	8821	8831	8837	8839	8849	8861	8863	8867	8887	8893	8923
8929	8933	8941	8951	8963	8969	8971	8999	9001	9007	9011	9013	9029	9041
9043	9049	9059	9067	9091	9103	9109	9127	9133	9137	9151	9157	9161	9173
9181	9187	9199	9203	9209	9221	9227	9239	9241	9257	9277	9281	9283	9293
9311	9319	9323	9337	9341	9343	9349	9371	9377	9391	9397	9403	9413	9419
9421	9431	9433	9437	9439	9461	9463	9467	9473	9479	9491	9497	9511	9521
9533	9539	9547	9551	9587	9601	9613	9619	9623	9629	9631	9643	9649	9661
9677	9679	9689	9697	9719	9721	9733	9739	9743	9749	9767	9769	9781	9787
9811	9817	9829	9833	9839	9851	9857	9791	9803	9859	9871	9883	9887	9901

ПРИЛОЖЕНИЕ 3

Таблицы семибитной кодировки ASCII

Кодировка *ASCII* (American Standard Code for Information Interchange, американский стандартный код для обмена информацией) является наиболее распространенной кодировкой для представления десятичных цифр, символов латинского алфавита, знаков препинания и управляющих символов. Первые 128 символов (для представления которых достаточно 7 битов) являются фиксированными и приводятся в табл. П3.1 (управляющие символы) и табл. П3.2 (алфавитно-цифровые и прочие символы). Также стандартизированы и старшие 128 кодов, которые представляют символы национальных алфавитов, но их значения зависят от выбранной кодовой страницы и здесь не приводятся, поскольку в Qt они не используются из-за поддержки стандарта Unicode.

Таблица П3.1. *Управляющие символы*

OCT	DEC	HEX	Символ	Описание
00	00	00	NUL	Пустой
01	01	01	SOH	Начало заголовка
02	02	02	STX	Начало текста
03	03	03	ETX	Конец текста
04	04	04	EOT	Конец передачи
05	05	05	ENQ	Запрос о подтверждении
06	06	06	ACK	Подтверждение
07	07	07	BEL	Звонок
10	08	08	BS	Возврат на один символ
11	09	09	TAB	Горизонтальная табуляция
12	10	0A	LF	Перевод строки
13	11	0B	VT	Вертикальная табуляция
14	12	0C	FF	Новая страница
15	13	0D	CR	Возврат каретки
16	14	0E	SO	Переключение на стандартный код
17	15	0F	SI	Переключение на другие таблицы кодировок
20	16	10	DLE	Отмена соединения с данными

Таблица П3.1 (окончание)

OCT	DEC	HEX	Символ	Описание
21	17	11	DC1	Управление устройством 1
22	18	12	DC2	Управление устройством 2
23	19	13	DC3	Управление устройством 3
24	20	14	DC4	Управление устройством 4
25	21	15	NAK	Не подтверждаю
26	22	16	SYN	Синхронизация передачи
27	23	17	ETB	Конец блока текста
30	24	18	CAN	Отмена
31	25	19	EM	Конец ленты
32	26	1A	SUB	Подставить
33	27	1B	ESC	Отмена
34	28	1C	FS	Разделитель файлов
35	29	1D	GS	Разделитель групп
36	30	1E	RS	Разделитель записей
37	31	1F	US	Разделитель модулей

Таблица П3.2. Символы

OCT	DEC	HEX	Символ	OCT	DEC	HEX	Символ
40	32	20	(пробел)	60	48	30	0
41	33	21	!	61	49	31	1
42	34	22	"	62	50	32	2
43	35	23	#	63	51	33	3
44	36	24	$	64	52	34	4
45	37	25	%	65	53	35	5
46	38	26	&	66	54	36	6
47	39	27	'	67	55	37	7
50	40	28	(	70	56	38	8
51	41	29	)	71	57	39	9
52	42	2A	*	72	58	3A	:
53	43	2B	+	73	59	3B	;
54	44	2C	,	74	60	3C	<
55	45	2D	-	75	61	3D	=
56	46	2E	.	76	62	3E	>
57	47	2F	/	77	63	3F	?

Таблица П3.2 *(окончание)*

OCT	DEC	HEX	Символ	OCT	DEC	HEX	Символ
80	64	40	@	120	96	60	`
81	65	41	A	121	97	61	a
82	66	42	B	122	98	62	b
83	67	43	C	123	99	63	c
84	68	44	D	124	100	64	d
85	69	45	E	125	101	65	e
86	70	46	F	126	102	66	f
87	71	47	G	127	103	67	g
90	72	48	H	130	104	68	h
91	73	49	I	131	105	69	I
92	74	4A	J	132	106	6A	j
93	75	4B	K	133	107	6B	k
94	76	4C	L	134	108	6C	l
95	77	4D	M	135	109	6D	m
96	78	4E	N	136	110	6E	n
97	79	4F	O	137	111	6F	o
100	80	50	P	140	112	70	p
101	81	51	Q	141	113	71	q
102	82	52	R	142	114	72	r
103	83	53	S	143	115	73	s
104	84	54	T	144	116	74	t
105	85	55	U	145	117	75	u
106	86	56	V	146	118	76	v
107	87	57	W	147	119	77	w
110	88	58	X	150	120	78	x
111	89	59	Y	151	121	79	y
112	90	5A	Z	152	122	7A	z
113	91	5B	[	153	123	7B	{
114	92	5C	\	154	124	7C	|
115	93	5D	]	155	125	7D	}
116	94	5E	^	156	126	7E	~
117	95	5F	_	157	127	7F	DEL

ПРИЛОЖЕНИЕ 4

Описание архива с примерами

Примеры к книге выложены на FTP-сервер издательства «БХВ-Петербург» по адресу: **ftp://ftp.bhv.ru/9785977536783.zip**. Ссылка доступна и со страницы книги на сайте **www.bhv.ru**.

Исходные коды примеров книги сгруппированы в отдельные каталоги (папки) глав. Каждый из примеров хранится в отдельном каталоге. Чтобы собрать эти примеры для Windows, Mac OS X и Linux, вы можете использовать Qt Creator или следующие далее команды:

- для Windows (MinGW C++):

```
qmake
make
```

- для Mac OS X:

```
qmake -spec macx-clang; make
```

- для Linux:

```
qmake; make
```

> ***Важно!!!***
>
> Для выполнения примеров вам потребуются настроить среду разработки Qt (см. *приложение 1*). Примеры компилируются только под Qt версии 5.10 и выше.

В табл. П4.1 приведено описание содержимого каталога примеров.

Таблица П4.1. *Содержимое архива*

Папка	Описание
chapter01	Пример *главы 1*: • Hello — программа, отображающая надпись **Hello, World**
chapter02	Пример *главы 2*: • Counter — приложение, демонстрирующее механизм сигналов и слотов
chapter03	Пример *главы 3*: • LibraryInfo — информация об используемой библиотеке Qt

Таблица П4.1 *(продолжение)*

Папка	Описание
chapter04	Примеры *главы 4*: • Background — демонстрация установки фона виджета; • MouseCursor — пример изменения указателя мыши; • ScrollArea — иллюстрация виджета видовой прокрутки
chapter05	Примеры *главы 5*: • AddStretch — приложение, где вместо одной из кнопок выполняется добавление фактора растяжения; • Calculator — калькулятор, демонстрирующий табличное размещение (`QGridLayout`); • GridLayout — приложение, демонстрирующее использование табличной компановки (`QGridLayout`); • HBoxLayout — приложение, демонстрирующее использование горизонтального размещения (`QHBoxLayout`); • Layout — иллюстрация совместного использования горизонтального и вертикального размещений; • Splitter — демонстрация виджета разделителя (`QSplitter`); • Stretch — пример использования фактора растяжения; • VBoxLayout — приложение, демонстрирующее использование вертикального размещения
chapter07	Примеры *главы 7*: • Label — пример использования виджета надписи (`QLabel`); • LabelBuddy — демонстрация возможности ассоциации виджета надписи с другими виджетами; • LabelPixmap — иллюстрация использования растровых изображений в виджете надписи; • LCD — приложение, демонстрирующее виджет электронного индикатора (`QLCDNumber`); • LinkLabel — приложение, демонстрирующее виджет надписи с веб-ссылкой (`QLabel`); • Progress — демонстрация работы виджета индикатора выполнения (`QProgressBar`)
chapter08	Примеры *главы 8*: • ButtonGroup — иллюстрация группировки кнопок; • ButtonPopup — пример кнопки со всплывающим меню; • Buttons — демонстрация различных режимов работы кнопки (`QPushButton`); • CheckBox — иллюстрация флажков (`QCheckBox`); • RadioButton — демонстрация виджета переключателей (`QRadioButton`)

***Таблица П4.1** (продолжение)*

Папка	Описание
chapter09	Примеры *главы 9*: • Dial — пример использования виджета установщика (`QDial`); • ScrollBar — демонстрация виджета полосы прокрутки (`QScrollBar`); • Slider — пример использования виджета ползунка (`QSlider`)
chapter10	Примеры *главы 10*: • DateTimeEdit — пример работы с виджетами отображения даты и времени (`QDateTimeEdit`); • LineEdit — демонстрация виджета однострочного текстового поля (`QLineEdit`); • SpinBox — иллюстрация использования виджета счетчика (`QSpinBox`); • SyntaxHighlighter — возможности реализации расцветки синтаксиса (`QSyntaxHighlighter`); • TextEdit — приложение, использующее виджет многострочного текстового поля (`QTextEdit`); • Validator — приложение, демонстрирующее проверку пользовательского ввода (`QValidator`)
chapter11	Примеры *главы 11*: • ComboBox — демонстрация виджета выпадающего списка (`QComboBox`); • IconMode — иллюстрация режимов показа значков приложений; • ListWidget — пример использования виджета простого списка (`QListBox`); • TableWidget — демонстрация возможностей виджета таблицы (`QTableWidget`); • TabWidget — пример использования виджета вкладок (`QTabWidget`); • ToolBox — иллюстрация работы с виджетом инструментов (`QToolBox`); • TreeWidget — демонстрация возможностей виджета иерархического списка (`QTreeWidget`)
chapter12	Примеры *главы 12*: • FileSystemModel — иллюстрация использования готовой модели `QFileSystemModel`; • Explorer — приложение, имитирующее проводник на базе использования модели `QFileSystemModel`; • HierarchicalModel — приложение, использующее модель `QStandartItemModel` для создания иерархии; • IntListModel — реализация собственной модели данных для списка целых чисел; • ProxyModel — пример использования промежуточной модели для осуществления отбора данных; • Roles — демонстрация использования ролей для отображения данных; • SelectionSharing — иллюстрация разделения выделений элементов между представлениями; • SimpleDelegate — реализация делегата, производящего выделение элементов при попадании на них указателя мыши; • TableModel — реализация табличной модели; • WidgetAndView — демонстрация использования моделей элементно-ориентированных классов

Таблица П4.1 *(продолжение)*

Папка	Описание
chapter13	Пример *главы 13*: • WidgetPalette — демонстрация изменения палитры виджета
chapter14	Примеры *главы 14*: • MouseEvent — пример обработки событий мыши; • MultiTouch — иллюстрация обработки события множественного касания «мультитач»; • ResizeEvent — иллюстрация обработки события изменения размеров
chapter15	Пример *главы 15*: • EventFilter — демонстрация механизма фильтрации событий
chapter16	Примеры *главы 16*: • EventChange — приложение, демонстрирующее подмену событий; • EventSimulation — пример искусственного создания событий
chapter18	Примеры *главы 18*: • CompositionModes — демонстрация режимов совмещения пикселов; • ConicalGradient — отображение конического градиента; • LinearGradient — отображение линейного градиента; • PainterPath — пример отображения графической траектории; • RadialGradient — отображение лучевого градиента; • GraphicsEffect — демонстрация применения эффектов к растровому изображению
chapter19	Примеры *главы 19*: • ImageDraw — рисование в контексте растрового изображения (`QImage`) с его последующим отображением; • Window — приложение, демонстрирующее применение прозрачности к виджету верхнего уровня; • ScanLine — операции с пикселами растрового изображения
chapter20	Примеры *главы 20*: • DrawText — отображение текстовой строки; • GradientText — отображение текстовой строки, заполненной градиентом; • ElidedText — отображение строки в режиме разрыва
chapter21	Примеры *главы 21*: • GraphicsView — отображение элементов с изменяемым местоположением на сцене графического представления; • GraphicsView — отображение виджетов на сцене графического представления; • CustomGraphicsView — демонстрация реализации собственного класса графического представления и собственного класса элемента с возможностью обработки событий и группировки

Таблица П4.1 *(продолжение)*

Папка	Описание
chapter22	Примеры *главы 22*: • Movie — программа, отображающая анимацию (`QMovie`); • ColorAnimation — анимация цвета; • EasingCurves — применение смягчающих линий для анимации; • SvgAnimation — показ анимации в формате SVG; • States — использование состояний и переходов с анимацией
chapter23	Примеры *главы 23*: • OGLDraw — приложение, демонстрирующее эффект сглаживания цветов вершин четырехугольника; • OGLPyramid — вращение пирамиды, демонстрирующее трехмерную графику OpenGL; • OGLQuad — пример вывода графических примитивов OpenGL
chapter24	Пример *главы 24*: • Printer — программа, выводящая картинку на печать с использованием класса `QPrinter`
chapter25	Пример *главы 25*: • CustomWidget — приложение, демонстрирующее создание и использование собственных виджетов
chapter26	Примеры *главы 26*: • AppStyle — иллюстрация использования различных стилей; • CSSStyle — пример использования каскадных стилей; • CustomStyle — приложение, иллюстрирующее создание и использование своих собственных стилей; • Styles — демонстрация интегрированных в Qt стилей
chapter27	Примеры *главы 27*: • SoundPlayer — демонстрация возможности воспроизведения звука (`QMediaPlayer`); • VideoPlayer — пример, показывающий возможности использования модуля того же класса `QMediaPlayer` для воспроизведения видео
chapter28	Пример *главы 28*: • Settings — приложение, сохраняющее свои настройки (`QSettings`)
chapter29	Примеры *главы 29*: • Drag — приложение, реализующее сторону источника для перетаскивания; • Drop — приложение, реализующее принимающую сторону для перетаскивания; • WidgetDragAndDrop — перетаскивание объектов внутри одного приложения

Таблица П4.1 *(продолжение)*

Папка	Описание
Chapter30	Пример *главы 30*: • Hello — пример интернационализации приложения
chapter31	Примеры *главы 31*: • ContextMenu — иллюстрация применения контекстного меню; • Menu — пример встраивания меню в приложение
chapter32	Примеры *главы 32*: • InputDialog — приложение, демонстрирующее реализацию собственного диалогового окна; • MessageBoxes — пример использования окон сообщений; • Wizard — пример использования окна ассистента; • StandardDialogs — демонстрация стандартных диалоговых окон
chapter33	Примеры *главы 33*: • HelpBrowser — приложение, предоставляющее систему помощи; • CustomToolTip — создание своего собственного окна подсказки
chapter34	Примеры *главы 34*: • StatusBar — приложение со строкой состояния; • ToolBar — демонстрация использования панелей инструментов (`QToolBar`); • MDI — пример MDI-приложения (редактора); • SDI — простой редактор для одного документа; • SplashScreen — приложение, отображающее окно заставки (`QSplashScreen`)
chapter35	Примеры *главы 35*: • SystemTray — пример использования области оповещений; • ScreenShot — демонстрация использования рабочего стола
chapter36	Примеры *главы 36*: • FileFinder — приложение для нахождения файлов, демонстрирующее использование класса `QDir`; • FileSystemWatcher — мониторинг изменения файлов и каталогов
chapter37	Примеры *главы 37*: • BlinkLabel — приложение, демонстрирующее работу таймера (`QTimer`); • Clock — приложение электронных часов, иллюстрирующее использование таймера и классов даты и времени (`QDateTime`)
chapter38	Примеры *главы 38*: • Process — приложение командной оболочки, демонстрирующее создание процессов (`QProcess`); • ThreadEvent — демонстрация отправки событий из потока; • ThreadSignal — иллюстрация отправки сигналов из потока; • QtConcurent и QtConcurent2 — использование фреймворка высокого уровня для работы с потоками; • ThreadTimer — пример использования сигнально-слотовых соединений в потоках

***Таблица П4.1** (продолжение)*

Папка	Описание
chapter39	Примеры *главы 39*: • Client и Server — приложения, иллюстрирующие возможности классов `QTcpServer` и `QTcpSocket`; • Downloader — реализации класса загрузки файлов на базе класса `QNetworkAccessManager`; • TcpClient и TcpServer — простой TCP-клиент и TCP-сервер с блокирующим подходом; • UdpClient и UdpServer — UDP-клиент и UDP-сервер на базе класса `QUdpSocket`
chapter40	Примеры *главы 40*: • XmlDOMRead — приложение, читающее XML-документ при помощи DOM; • XmlDOMWrite — приложение, демонстрирующее создание XML-документа при помощи DOM и его запись; • XmlSAXRead — пример чтения XML-документа при помощи SAX; • XmlSreamReader — простой синтаксический XML-анализатор на базе класса `QXmlStreamReader`; • XQuery — демонстрация возможностей класса `QXmlQuery` модуля `QtXmlPatterns`
chapter41	Примеры *главы 41*: • SQL — приложение, осуществляющее чтение и запись в базу данных; • SQLQueryModel — демонстрация проведения отбора данных; • SQLTableModel — использование класса модели `QSqlTableModel`; • SqlRelationTableModel — демонстрация использования модели класса `QSqlRelationTableModel`
chapter42	Примеры *главы 42*: • DynLib — демонстрация создания и загрузки динамических библиотек; • PlugIn — демонстрация создания и загрузки расширений
chapter43	Примеры *главы 43*: • WinAPI — использование платформозависимых функций Windows; • MacButton — использование фреймворка Cocoa
chapter44	Примеры *главы 44*: • MyForm — приложение, созданное с помощью программы Qt Designer; • LoadMyForm — приложение, показывающее динамическую загрузку ui-файла с пользовательским интерфейсом
chapter45	Примеры *главы 45*: • DataDrivenTest — проведение теста с передачей данных; • GuiTest — тест графического интерфейса; • TestLib — программа проведения теста

***Таблица П4.1** (продолжение)*

Папка	Описание
chapter46	Пример *главы 46*: • SimpleView и WebBrowser — демонстрация использования модуля `WebKit`
chapter49	Примеры, используемые в *главе 49*: • ZweiMalZwei и HelloScript — простые примеры для интерпретатора Qt Script
chapter52	Примеры *главы 52*, демонстрирующие использование языка сценария в Qt-приложениях, а также и их отладку: • JSTools — демонстрирует возможность расширения языка JavaScript дополнительными функциями; • SignalAndSlots — показывает возможности использования механизма сигналов слотов в языке сценария JavaScript; • Turtle — пример «черепашьей» графики, управляемой языком сценария JavaScript
chapter53	Пример *главы 53*: • HelloWorld — первый Qt Quick-проект, созданный Qt Creator
chapter54	Примеры *главы 54*, демонстрирующие использование и создание элементов: • CustomElement — пример создания собственного элемента; • Controls — демонстрация элемента окна приложения других элементов управления модуля `QtQuick.Controls`; • Dialogs — использование диалоговых окон выбора цвета, шрифта, файлов и окна сообщений; • Flickable — базовый элемент для представлений; • OneControl — использование кнопки нажатия из модуля элементов управления QML; • IdRefence — использование идентификатора элемента; • OnWidthAndHeight — реагирование на изменение значений свойств элемента; • Properties — расширение элемента свойствами различных типов; • Button — демонстрация различных видов кнопок модуля `QtQuick.Controls`; • RectangleWithFrame — элемент прямоугольника с закругленной рамкой и контуром; • Rectangle — элемент прямоугольника
chapter55	Примеры *главы 55*, демонстрирующие приемы для размещения элементов при помощи фиксации: • Anchors и Ancors2 — использование фиксаторов; • AnchorsOver — использование фиксации с наложением элементов; • AnchorsStretch — использование фиксации с растяжением среднего элемента; • Margins — использование отступов при фиксации; • Grid и Row — табличное и горизонтальное размещения с помощью элементов `Grid` и `Row`; • Flow — размещение элементов в виде змейки; • RowLayout — горизонтальное размещение с помощью элемента `RowLayout`;

Таблица П4.1 *(продолжение)*

Папка	Описание
chapter56	Примеры *главы 56*, демонстрирующие элементы для графики: • BorderImage — отображение растрового изображения для изменяющейся по размерам кнопки без искажений; • Canvas — рисование на элемента холста; • CanvasGradient — отображение градиента на элемента холста; • CanvasText — отображение текста со свечением на элемента холста; • Gradients — отображение линейного градиента; • Effects — показ встроенных элементов шейдера; • Shaders — реализация и использование элемента шейдера; • WebImage — загрузка и показ растрового изображения из веб; • Image — отображение растрового изображения с трансформацией
chapter57	Примеры *главы 57*, работа с пользовательским вводом: • Button и ButtonAlternative — действующая кнопка, использование элемента `MouseArea`; • EnterAndExit и HoverEvent — реакция на перекрытие курсором мыши элемента. Использование элемента `MouseArea`; • KeysInput — ввод с клавиатуры на уровне событий; • MouseArea — использование элемента `MouseArea`; • MultiPointTouchArea — реализация обработки событий множественных касаний «мультитач»; • Navigation — демонстрация смены фокуса между двумя элементами прямоугольников; • Signals — создание сигналов; • TextInput — элемент однострочного ввода текста; • TwoTextEdits — демонстрация смены фокуса между двумя элементами текстового ввода
chapter58	Примеры *главы 58*, демонстрирующие возможности использования анимации: • BehaviorAnimation — анимация поведения; • ColorAnimation — анимация цвета; • NumberAnimation — анимация числовых свойств; • ParallelAnimation, SequentialAnimation — параллельная и последовательная анимации; • RotationAnimation — анимация поворота; • PropertyAnimation — анимация свойств; • LetItSnow — демонстрация использования системы частиц для анимации множества объектов; • State — анимация с использованием состояний; • Transition — анимация с использованием состояний с переходами

Таблица П4.1 *(окончание)*

Папка	Описание
chapter59	Примеры *главы 59*, демонстрирующие возможности использования концепции «модель-представление»: • GridView — табличное представление модели списка ListModel; • GridViewXML — табличное представление XML-модели XmlListModel; • ListView — представление в виде списка JSON-модели данных; • PathViewLine — представление в виде замкнутой ленты (в одну линию); • PathViewQuad — представление в виде замкнутой ленты (3D-карусель); • VisualItemModel — демонстрация визуальной модели данных; • XMLModel — использование XML-модели данных
chapter60	Примеры *главы 60*, показывающие возможности совместного использования QML с C++: • QMLCPPUsage — интеграция QML-программ в Qt; • QMLCPPConnect — соединение сигналов QML со слотами C++; • Calculation — использование алгоритмов C++ в QML; • ImageProvider — создание растровых изображений с C++ и использование их в QML; • PaintElement — реализация визуального элемента эллипса на C++ и использование его в QML; • CPPExtension — расширение QML посредством C++
chapter61	Примеры *главы 61*, показывающие возможности трехмерной графики Qt 3D: • Animation — показ анимации из двух объектов сферы и пирамиды; • Material — отображение объекта сферы с материалом `PhongMaterial`; • Snowman — применение трансформаций для построения из простых трехмерных объектов фигуры снеговика
chapter64	Примеры *главы 64*, показывающие особенность программирования мобильных приложений: • Accelerometer — демонстрация использования сенсора акселерометра; • Gestures — обработка жеста растяжения/сужения и поворота; • Orientation — обработка событий поворота мобильных устройств; • Position — демонстрация определения текущей геопозиции
chapter65	Пример *главы 65*, реализация приложения для магазина App Store и Google Play: • DoNotTouchIt — приложение-сигнализация, реагирующее на подвижность мобильного устройства
common	Каталог с графическими ресурсами
Exaples.pro	Основной проектный файл для компиляции всех примеров книги
readme.txt	Инструкции на английском языке

Предметный указатель

A

Antialiasing 267, 274, 345
Apple 633
Arthur 267
ASCII 225, 1021

B

Backing Store 108
begin() 76
bi-level 306
Bitmap 293
BMP 293
Break points 65

C

C++ 632
Call Stack 701
Callback functions 38
Chromium 666
clear() 76
Clipboard 414
CMYK 263
Cocoa 635
Collision detection 316
Color roles 217
Connection 46
constBegin() 76
constEnd() 76
count() 76
Critical section 547
CSS 8, 383, 670
◊ селектор: определение 383

D

Data Compression 511
datagram 555
Deadlock 551
desktop 31
Disabled 107
DisplayRole 201
DNS 557
DOM 9, 584
double buffering 227, 268
Drag & Drop 159, 233, 415, 416

E

emit 42, 46
empty() 76
Enabled 107
end() 76

F

Firing interval 528
fixqt4headers.pl 704
Flat Button 143
foreach 80, 170
forever 579
Form UI
◊ direct approach 648
◊ inheritance approach 649
◊ multiple inheritance approach 650
FTP 557

G

GCC 633
GDB 63, 694
◊ attach 66
◊ break 66
◊ clear 66
◊ continue 66
◊ delete 66
◊ detach 66
◊ disable 66
◊ enable 66
◊ help 65
◊ next 66
◊ quit 65
◊ run 65
◊ step 66
◊ tbreak 66
◊ until 66
GDI 629
gestrures 240
GIF 293, 294, 330
GL_COLOR_BUFFER_BIT 349
GL_COMPILE 357
GL_DEPTH_BUFFER_BIT 349
GL_FLAT 349, 355
GL_LINE_LOOP 352
GL_LINE_STRIP 352
GL_LINES 352
GL_POINTS 352
GL_POLYGON 353
GL_QUADS 349, 353, 357
GL_SMOOTH 349
GL_TRIANGLE_FAN 357
GL_TRIANGLE_STRIP 352
GL_TRIANGLES 353
glBegin() 349
GLbyte 346
glCallList() 356
glClear() 349
glColor() 349
GLdouble 346
glEnd() 349
glEndList() 357
GLenum 346
GLfloat 346
glFrustum() 356
glGenLists() 357
GLint 346
glLoadIdentity() 349
glMatrixMode() 349, 356
glNewList() 357
glOrtho() 349
glPointSize() 353
glShadeMode() 355
glShadeModel() 349
GLshort 346
GLSL 833
GLubyte 346
GLuint 346
GLushort 346
glVertex() 349
glViewPort() 349, 356
Graphics Interchange Format 294
GraphicsItem: dragLeaveEvent() 322
GraphicsLineItem 319
GUI 13, 104

H

HSL 260
HSV 260, 262
HTML 164, 665
HTTP 557
HTTPS 557

I

IDE 675
IMAP 557
insert() 76
instaceof 740
iPad 633
iPhone 633
iPodtouch 633
IRC 557
isEmpty() 76
iTunes 665

J

JavaScript 712, 720
◊ Array 746
◊ arseFloat() 735
◊ eval() 735
◊ parseFloat() 724
◊ parseInt() 724, 735
◊ sNaN() 735
◊ ключевое слово 720
◊ комментарии 721
◊ массив 746
◊ объект Date 749
◊ объект Global 744
◊ объект Number 744
◊ объект String 745
◊ переменная 721
◊ подстрока 745
◊ преобразование типа 724
◊ регулярные выражения 745
◊ строка 723
◊ тип данных 722
◊ условный оператор 728
◊ цикл 730
Joint Photographic Experts Group (JPEG) 293, 294
JSON 742, 876

K

KDE 497
Key 406
Key Value 406

L

Layout 116
Layout managers 116
lconvert 438
LIBS 633
link 62
Linux 638
Look&Feel 31, 371
lrelease 428, 438
lupdate 428, 430, 438
LZW 294

M

Mac OS 637
Mac OS X 57, 497, 633, 665
macdeployqt 293, 603, 614, 619
makefile 55
make-файл 57
MDI 9, 484, 496
Memory leak 119
Memory mapped files 512
Menu Extras 497
MFC 38
Microsoft Visual Studio 62
MIME 415, 667
MinGW 633, 675
MNG 293, 294, 330
MOC 40, 41, 60
Modelview matrix 349
Motif 38
Multiple Document Window Interface 484, 496
Multi-touch 239

N

Netbios-SSN 557
Normal Button 143
NSButton 635
NSString 636
NSSwitchButton 635
NTP 557

O

Objective C 632
Objective C++ 628, 632
ODF 164
OpenDocument Format 164
OpenGL 345
◊ вершина 349
◊ графические примитивы 350
OpenOffice.org 164

P

PBM 293
PDF 360
PGM 293
Phonon 703, 706
Pixmap Button 143
PlainText 164
PNG 293, 294
PNM 293
POP3 557
Portable Network Graphics 294
PostScript 164
PowerPC 706
PPM 293
progress bar 136
Projection matrix 349

Q

Q_ARG 890
Q_ASSERT() 66
Q_CHECK_PTR() 66
Q_DECLARE_INTERFACE() 620
Q_EXPORT_PLUGIN 705
Q_EXPORT_PLUGIN2 705
Q_INT16 70
Q_INT32 70
Q_INT64 70
Q_INT8 70
Q_INTERFACES() 624
Q_INVOKABLE 897
Q_INVOKABLE 713, 894
Q_OBJECT 40, 60
Q_OS_AIX 628
Q_OS_ANDROID 628
Q_OS_FREEBSD 628
Q_OS_IOS 628
Q_OS_IRIX 628
Q_OS_LINUX 628
Q_OS_OSX 628
Q_OS_SOLARIS 628
Q_OS_TVOS 628
Q_OS_WATCHOS 628
Q_OS_WIN32 628
Q_OS_WIN64 628
Q_OS_WINRT 628
Q_PROPERTY 897
Q_PROPERTY 36
Q_RETURN_ARG 890
Q_UINT16 70
Q_UINT32 70
Q_UINT64 70
Q_UINT8 70
Q_WS_MACX 705
Q_WS_WIN 705
Q_WS_X11 705
QABS() 70
QAbstactScrollArea 192
QAbstractAnimation 333
◊ pause() 333
◊ setLoopCount() 334
◊ start() 333, 334, 336
◊ stateChanged() 333
◊ stop() 333
QAbstractButton 142
◊ clicked() 142
◊ icon() 142
◊ iconSize() 142
◊ isChecked() 143, 150
◊ isDown() 143
◊ isEnabled() 143
◊ pressed() 142
◊ released() 143
◊ setChecked() 143, 147
◊ setDown() 143

QAbstractButton *(прод.)*
◊ setEnabled() 143
◊ setIcon() 142
◊ setIconSize() 142
◊ setText() 142
◊ text() 142
◊ toggled() 143
QAbstractGraphicsShapeItem 319
QAbstractItem
◊ SingleSelection 178
QAbstractItemDelegate 195
QAbstractItemModel 191
◊ data() 197, 201
◊ index() 197
QAbstractItemView 178, 192
◊ DoubleClicked 192
◊ model() 214
◊ MultiSelection 178
◊ NoEditTriggers 192
◊ NoSelection 178
◊ SelectedClicked 193
◊ selectionModel() 193
◊ setEditTriggers() 192
◊ setItemDelegate() 195
◊ setRootIndex() 200
◊ setSelectionMode() 178
◊ setSelectionModel() 193
QAbstractListItem
◊ beginInsertRows() 208
◊ beginRemoveRows() 208
◊ endInsertRows() 208
◊ endRemoveRows() 208
QAbstractListModel 191
QAbstractPrintDialog
◊ fromPage() 363
◊ toPage() 363
QAbstractProxyModel 192
QAbstractScrollArea
◊ viewPort() 240
◊ viewportEvent() 240
QAbstractScrollArea 113
◊ cornerWidget() 113
◊ horizontalScrollBar() 113
◊ setViewPort() 318
◊ verticalScrollBar() 113
◊ viewport() 113
QAbstractSlider 152
◊ setMaximum() 152
◊ setMinimum() 152
◊ setOrientation() 152, 396
◊ setPageSteps() 153
◊ setRange() 152, 396
◊ setSingleStep() 153
◊ setTracking() 153
◊ setValue() 153, 155, 400
◊ sliderChange() 152
◊ sliderMoved() 396
◊ sliderPressed() 153
◊ sliderReleased() 153
◊ value() 153
QAbstractSocket 555
◊ ConnectedState 579
◊ waitForBytesWritten() 577
◊ waitForConnected() 577
◊ waitForDisconnected() 577
◊ waitForReadyRead() 577
QAbstractSpinBox 171
QAbstractTableModel 191
QAbstractTransition
◊ addAnimation() 342
QAction 477
◊ addAction() 491
◊ setEnabled() 442
◊ triggered() 491
qAlpha() 262
QAndroidStyle 373
QAnimationGroup 333
◊ addAnimation() 335
QApplication
◊ alert() 31
◊ applicationDirPath() 513
◊ applicationFilePath() 513
◊ beep() 393
◊ clipboard() 414
◊ closeAllWindows() 491
◊ desktop() 502
◊ installFilter() 249
◊ processEvent() 515
◊ processEvents() 527
◊ restoreOverrideCursor() 111
◊ setDoubleClickInterval() 229
◊ setFont() 310
◊ setPalette() 219
◊ setQuitOnLastWindow() 498
◊ setStyle() 373, 376
◊ setStyleSheet() 383
◊ startDragDistance() 417
QAudioFormat
◊ setChannels() 706
◊ setFrequency() 706
QBasicTimer 532
◊ isActive() 532
◊ start() 532
◊ stop() 532
◊ timerId() 532
qBinaryFind() 92
QBitArray 83
◊ operator[ ] 83
◊ setBit() 83
◊ testBit() 83
QBitmap 306
qBlue() 262, 298
qBound() 70
QBoxLayout 117
◊ addLayout() 122, 123
◊ addSpacing() 118
◊ addStretch() 119
◊ BottomToTop 118
◊ insertLayout() 118
◊ insertSpacing() 118
◊ insertStretch() 118
◊ insertWidget() 118
◊ LeftToRight 118
◊ RightToLeft 118
◊ TopToBottom 118
QBrush 217, 271
QBuffer 421, 508, 512
◊ buffer() 512
◊ setBuffer() 512
QByteArray 83, 416, 421, 561
◊ fromBase64() 83
◊ toBase64() 83
QCanvasRectangle
◊ setRect() 319
QCheckBox 145
◊ setNoChange() 146
◊ setTristate() 146
QChildEvent 234
◊ added() 234
◊ removed() 234
QClipboard 414
◊ dataChanged() 414
◊ image() 415
◊ mimeData() 415
◊ pixmap() 415
◊ setImage() 414
◊ setMimeData () 414
◊ setPixmap() 414
◊ setText() 414
◊ text() 415
QCloseEvent 234
QColor 217, 260
◊ alpha() 261
◊ alphaF() 261
◊ blue() 261
◊ blueF() 261
◊ color0 111
◊ color1 111
◊ darker() 266
◊ getHsv() 263
◊ getRgb() 262
◊ green() 261
◊ greenF() 261
◊ isValid() 456
◊ lighter() 266
◊ red() 261
◊ redF() 261
◊ rgb() 262

◊ setHsv() 263
◊ setRgb() 262
QColorDialog 455
◊ getColor() 455
QComboBox 185, 192
◊ activated() 185
◊ addItem() 185
◊ addItems() 185
◊ currentIndex() 185
◊ editTextChanged() 185
◊ setDuplicatesEnabled() 185
◊ setEditable() 185, 186
◊ setModel() 192
◊ setValidator() 173
QCommonStyle 372, 380
QCOMPARE() 656, 660, 662
qCompress() 83, 511
QConicalGradient 273
QContentHandler
◊ characters() 590
◊ endDocument() 590
◊ endElement() 590
◊ startDocument() 590
◊ startElement() 590
QContextMenuEvent
◊ globalPos() 443
qCopy() 92
qCopyBackward() 92
QCoreApplication 30, 68, 224, 244, 250, 537
◊ applicationDirPath() 513
◊ arguments() 520
◊ exec() 27, 224, 236
◊ exit() 27
◊ installFilter() 249
◊ installTranslator() 434, 435
◊ notify() 249
◊ postEvent() 236, 250, 252, 544
◊ processEvents() 244, 537
◊ quit() 308
◊ sendEvent() 236, 250, 544
◊ setApplicationName() 406, 413
◊ setOrganisationName() 406
◊ setOrganizationName() 413
qCount() 92
QCursor 109
QDataStream 561
QDate 183, 524
◊ addDays() 525
◊ currentDate() 525
◊ day() 524
◊ dayOfWeek() 525
◊ dayOfYear() 525
◊ daysInMonth() 524
◊ daysInYear() 524
◊ daysTo() 525
◊ fromString() 525
◊ month() 524
◊ setDate() 524
◊ weekNumber() 525
◊ year() 524
QDateStream 522
QDateTime 527
◊ currentDateTime() 172
QDateTimeEdit
◊ dateTimeChanged() 172
QDB2 602
QDebug
◊ hex 69
◊ nospace() 69
◊ uppercasedigits 69
qDebug() 67–69, 697
QDeclarativeView 888
qDeleteAll() 92
QDesktopServices 506
QDesktopWidget 502
◊ isVirtualDesktop() 503
◊ primaryScreen() 503
◊ resized() 503
◊ screen() 505
◊ screenCount() 503, 505
◊ screenCountChanged() 503
◊ screenGeometry() 503
◊ screenNumber() 503
QDial 152, 156
◊ setNotchesVisible() 156
◊ setNotchTarget() 156
◊ setWrapping() 156
◊ valueChanged() 158
QDialog 446
◊ accept() 450
◊ Accepted 363, 447, 450
◊ exec() 448, 450
◊ hide() 447
◊ isModal() 446
◊ Rejected 447, 450
◊ rejected() 450
◊ setModal() 446
◊ show() 447
QDir 508, 512
◊ absolutePath() 513
◊ cd() 513
◊ cdUp() 513, 621
◊ count() 513
◊ current() 512
◊ currentPath() 200
◊ drives() 513
◊ entryInfoList() 513
◊ entryList() 513, 516
◊ exists() 513
◊ home() 513
◊ makeAbsolute() 513
◊ mkdir() 513
◊ rename() 513
◊ rmdir() 513
◊ root() 512
◊ setFilter() 516
◊ setSorting() 516
◊ tempPath() 512
QDockWidget
◊ setAllowedAreas() 480
◊ setFeatures() 480
QDomAttr 585
◊ setValue() 588
QDomDocument
◊ appendChild() 587
◊ createAttribute() 587, 588
◊ createElement() 587, 588
◊ createTextNode() 587, 588
◊ documentElement() 586
◊ setContext() 586
◊ toString() 587
QDomElement 585
◊ attribute() 586
◊ tagName() 586
◊ text() 586
QDomNode 585
◊ firstChild() 586
◊ nextSibling() 586
◊ toElement() 585
◊ traverseNode() 586
QDomText 585
QDoubleValidator 173
QDrag 417
◊ exec() 417
◊ setMimeData() 423
◊ setPixmap() 417
QDragEnterEvent 233
◊ acceptProposedAction() 420
QDragLeaveEvent 233
QDragMoveEvent 233
◊ accept() 420
◊ ignore() 420
QDropEvent 234
QEasingCurve 337
◊ InBack 339
◊ InBounce 340
◊ InCirc 338
◊ InCubic 338
◊ InElastic 339
◊ InExpo 341
◊ InOutBack 340
◊ InOutBounce 340
◊ InOutCirc 338
◊ InOutCubic 339
◊ InOutElastic 339
◊ InOutExpo 336, 338
◊ InOutQuad 338

QEasingCurve *(прод.)*
◊ InOutQuart 339
◊ InOutSine 341
◊ InQuad 338
◊ InQuart 339
◊ InQuint 340
◊ InSine 340
◊ Linear 337
◊ OutBack 339
◊ OutBounce 336, 340
◊ OutCirc 338
◊ OutCubic 338
◊ OutElastic 339
◊ OutExpo 337
◊ OutInBack 340
◊ OutInBounce 340
◊ OutInCirc 338
◊ OutInCubic 339
◊ OutInElastic 339
◊ OutInExpo 338
◊ OutInQuad 338
◊ OutInQuart 339
◊ OutInQuint 340
◊ OutInSine 341
◊ OutQuad 338
◊ OutQuart 339
◊ OutQuint 340
◊ OutSine 340
qEqual() 92
QErrorMessage 467
◊ message() 467
QEvent 223, 237
◊ accept() 224, 234
◊ ActionAdded 238
◊ ActionChanged 238
◊ ActionRemoved 238
◊ ActivationChange 238
◊ ApplicationActivated 238
◊ ApplicationDectivate 238
◊ ApplicationFontChange 238
◊ ApplicationLayoutDirection Change 238
◊ ApplicationPaletteChange 238
◊ ApplicationWindowIconChange 238
◊ ChildAdded 238
◊ ChildPolished 238
◊ ChildRemoved 238
◊ Clipboard 238
◊ Close 237
◊ ContextMenu 238
◊ DeferredDelete 238
◊ Destroy 237
◊ DragEnter 238
◊ DragLeave 238
◊ DragMove 238
◊ Drop 238
◊ EnabledChange 238
◊ Enter 237
◊ EnterWhatsThisMode 238
◊ FileOpen 238
◊ FocusIn 237
◊ FocusOut 237
◊ FontChange 238
◊ Hide 237
◊ HideToParent 238
◊ HoverEnter 238
◊ HoverLeave 238
◊ HoverMove 238
◊ IconDrag 238
◊ IconTextChange 238
◊ ignore() 224, 234
◊ InputMethod 238
◊ KeyPress 237, 250, 253
◊ KeyRelease 237, 250
◊ LanguageChange 238, 435
◊ LayoutDirectionChange 238
◊ LayoutRequest 238
◊ Leave 237
◊ LeaveWhatsThisMode 238
◊ LocaleChange 238
◊ ModifiedChange 238
◊ MouseButtonDblClick 237
◊ MouseButtonPress 237, 248
◊ MouseButtonRelease 237
◊ MouseMove 237
◊ MouseTrackingChange 238
◊ Move 237
◊ None 237
◊ Paint 237
◊ PaletteChange 238
◊ ParentAboutToChange 238
◊ ParentChange 237
◊ Polish 238
◊ PolishRequest 238
◊ QueryWhatsThis 238
◊ Resize 237
◊ Shortcut 238
◊ ShortcutOverride 238
◊ Show 237
◊ ShowToParent 238
◊ SockAct 238
◊ StatusTip 238
◊ StyleChange 238
◊ TabletMove 238
◊ TabletPress 238
◊ TabletRelease 238
◊ ThreadChange 237
◊ Timer 237
◊ ToolBarChange 238
◊ ToolTip 238, 470
◊ TouchBegin 238, 240
◊ TouchCancel 238
◊ TouchEnd 238, 240
◊ TouchUpdate 238, 240
◊ type() 224, 237
◊ UpdateLater 238
◊ UpdateRequest 238
◊ User 236, 238
◊ WhatsThis 238
◊ WhatsThisClicked 238
◊ Wheel 238
◊ WindowActivate 237
◊ WindowBlocked 238
◊ WindowDeactivate 238
◊ WindowIconChange 238
◊ WindowStateChange 238
◊ WindowTitleChange 238
◊ WindowUnblocked 238
◊ WinEventAct 238
◊ ZOrderChange 238
QEventLoop 531
◊ exec() 531
◊ quit() 531
qFatal() 67, 68, 69
QFETCH() 660
QFile 508, 510
◊ close() 486
◊ exists() 510
◊ fileName() 510
◊ flush() 510
◊ open() 485
◊ setName() 510
QFileDialog 451
◊ getExistingDirectory() 452, 454
◊ getOpenFileName() 452, 485
◊ getOpenFileNames() 452
◊ getSaveFileName() 452, 486
QFileInfo 508, 516
◊ absoluteFilePath() 517
◊ baseName() 517
◊ completeSuffix() 517
◊ created() 517
◊ fileName() 517
◊ filePath() 517
◊ isDir() 516
◊ isExecutable() 517
◊ isFile() 516
◊ isHidden() 517
◊ isReadable() 517
◊ isSymLink() 516
◊ isWriteable() 517
◊ lastModified() 517
◊ lastRead() 517

QFileSystemModel 192, 199
QFileSystemWatcher 518
◊ addPath() 518
◊ addPaths() 520
◊ directories() 520
◊ directoryChanged() 518, 520
◊ files() 520
◊ fileChanged() 518, 520
◊ removePath() 518
qFill() 92
qFind() 92
qFindChildren<T>() 705
QFocusEvent 227
QFont 310
QFontDatabase 310
◊ families() 310
QFontDialog 457
◊ getFont() 458
QFontInfo 310
◊ bold() 311
◊ family() 311
◊ italic() 311
QFontMetrics 310, 311, 314
◊ ascent() 311
◊ boundingRect() 311
◊ charWidth() 311
◊ descent() 311
◊ elidedText() 314
◊ height() 311
◊ leftBearing() 311
◊ lineSpacing() 311
◊ rightBearing() 311
◊ width() 311
QFormLayout 123
QFrame 112
◊ Box 113
◊ drawFrame() 369
◊ HLine 113
◊ NoFrame 113
◊ Panel 113
◊ Plain 112
◊ Raised 112
◊ setFrame() 160
◊ setFrameStyle() 112, 369
◊ setLineWidth 160
◊ setLineWidth() 113, 369
◊ setMidLineWidth() 113
◊ Sunken 112
◊ VLine 113
◊ WinPanel 113
QFtpl 705
QFusionStyle 373
QFuture 552
◊ isFinished() 552
◊ isPaused() 553
◊ isRunning() 552
◊ isStarted() 553
◊ pause() 553
qFuzzyCompare() 70
QGradient
◊ setColorAt() 313
QGraphicsEffect 289
◊ draw() 289
QGraphicsEllipseItem 319
QGraphicsItem 316
◊ dragEnterEvent() 322
◊ dragMoveEvent() 322
◊ dropEvent() 322
◊ hide() 319
◊ paint() 319
◊ setAcceptDrops() 322
◊ setEnable() 319
◊ setPos() 319
◊ setTransform() 327
◊ setTransformation() 319
◊ show() 319
QGraphicsItemAnimation 333
QGraphicsItemGroup 319
QGraphicsPixmapItem 319
QGraphicsPolygonItem 319
QGraphicsProxyWidget 327
QGraphicsRectItem 319
◊ setBrush() 319
◊ setPen() 319
QGraphicsScene 316
◊ addItem() 318
◊ itemAt() 318
◊ items() 318
QGraphicsSceneEvent
◊ widget() 322
QGraphicsSimpleTextItem 319
QGraphicsView 319, 888
◊ centerOn() 318
◊ setMatrix() 319
◊ setScene() 319
◊ show() 320
qGreen() 262, 298
QGridLayout 117, 123
◊ addWidget() 126
◊ setColStretch() 123
◊ setRowStretch() 123
◊ setSpacing() 123, 124
QGroupBox 148
◊ setCheckable() 149
◊ setChecked() 149
QGtkStyle 373
QGuiApplication
◊ primaryScreen() 958
◊ setOverrideCursor() 110
qHash() 89
QHash<K,T> 89
QHashIterator 77
QHBoxLayout 118, 120, 122
QHeaderView 193
QHelpEvent 470
QHideEvent 234
QHostAddress
◊ LocalHost 579
QHttp 705
QIBASE 602
QImage 296
◊ converToFormat() 296
◊ format() 296
◊ Format_ARGB32 296
◊ Format_ARGB32_Premultiplied 296
◊ Format_Index8 296
◊ Format_Invalid 296
◊ Format_Mono 296
◊ Format_MonoLSB 296
◊ Format_RGB32 296
◊ invertPixels() 301
◊ load() 297
◊ mirrored() 302
◊ pixel() 297
◊ save() 297
◊ scaled() 301
◊ scanLine() 298
◊ setColor() 297
◊ setPixel() 297
QImageIOPlugin 617
QInputDialog 458
◊ getDouble() 458
◊ getInt () 458
◊ getItem() 458
◊ getText() 458
QInputEvent
◊ modifiers() 224, 225, 230
qInstallMsgHandler() 67
QIntValidator 173
QIODevice 508
◊ Append 509
◊ atEnd() 509
◊ close() 509
◊ flush() 579
◊ getChar() 509
◊ isOpen() 510
◊ isReadable() 510
◊ isWriteable() 510
◊ NotOpen 509
◊ open() 509
◊ openMode() 509
◊ pos() 509
◊ read() 509, 510
◊ readAll() 509, 511
◊ readData() 509
◊ readLine() 509
◊ ReadOnly 509

QIODevice *(прод.)*
◊ ReadWrite 509
◊ seek() 509
◊ size() 509
◊ Text 509
◊ Truncate 509
◊ Unbuffered 509
◊ write() 509, 510, 579
◊ writeData() 509
◊ WriteOnly 509
QItemDelegate 195
◊ createEditor() 196
◊ setEditorData() 196
◊ setModelData() 196
QItemSelectionModel 193
◊ currentChanged() 194
◊ currentColumnChanged() 194
◊ currentRowChanged() 194
◊ select() 194
◊ selectedIndexes() 194
◊ selectionChanged() 194
QJSEngine 718, 754
◊ evaluate() 717, 718, 754
◊ globalObject() 755
◊ newQObject() 718, 755, 759
QJSValue 718, 754
◊ isError() 754
◊ setProperty() 718
QKeyEvent 225, 253
◊ key() 225
◊ text() 225
QLabel 132, 162
◊ linkActivated() 136
◊ setAlignment() 132, 230
◊ setBuddy() 135, 160, 174
◊ setMovie() 132, 331
◊ setNum() 155
◊ setOpenExternalLinks() 136
◊ setPixmap() 132, 135, 334, 505
◊ setText() 132, 134, 230, 235, 400, 420, 529
QLayout 117
◊ addLayout() 117
◊ addWidget() 117
◊ removeWidget() 117
◊ setContentsMargins() 117, 123
◊ setSpacing() 117
QLayoutItem 117
QLCDNumber 139
◊ Bin 140
◊ Dec 140
◊ display() 140
◊ Filled 139
◊ Flat 139
◊ Hex 140
◊ Oct 140
◊ Outline 139
◊ overflow() 139
◊ setBinMode() 140
◊ setDecMode() 140
◊ setHexMode() 140
◊ setMode() 140
◊ setNumDigits() 139
◊ setOctMode() 140
◊ setSegmentStyle() 126, 139
◊ setSmallDecimalPoint() 139
◊ valueChanged() 140
QLibrary 615
◊ resolve() 616
QLibraryInfo 71
QLine 259
QLinearGradient 272
QLineEdit 159
◊ clear() 566
◊ copy() 161
◊ cut() 161
◊ isRedoAvailable() 161
◊ isUndoAvailable() 161
◊ maxLength() 161
◊ Password 159
◊ paste() 161
◊ redo() 161
◊ returnPressed() 159
◊ setEchoMode() 159
◊ setMaxLength() 161
◊ setReadOnly() 159
◊ setText() 159, 566
◊ setValidator() 173
◊ text() 159
◊ textChanged() 159
◊ undo() 161
QLineF 259
QLinkedList<T> 85
QLinkedListIterator 77
QList<T> 84
◊ at() 85
◊ move() 84
◊ removeFirst() 84
◊ removeLast() 84
◊ swap() 84
◊ takeAt() 84
◊ takeFirst() 84
◊ takeLast() 84
◊ toSet() 84
◊ toStdList() 84
◊ toVector() 85
QListIterator 77
◊ findNext() 77
◊ findPrevious() 77
◊ hasNext() 77
◊ hasPrevious() 77
◊ next() 77
◊ peekNext() 77
◊ peekPrevious() 77
◊ previous() 77
◊ toBack() 77
◊ toFront() 77
QListView 193
▫ TopToBottom 179
◊ setFlow() 179
◊ setSelectionMode() 182
QListWidget 176, 213
◊ addItem() 176
◊ clear() 176
◊ currentItem() 178
◊ insertItem() 176
◊ insertItems() 176
◊ itemChanged() 178
◊ itemClicked() 178
◊ itemDoubleClicked() 178
◊ itemSelectionChanged() 178
◊ itemWidget() 177
◊ selectedItems() 178
◊ setItemWidget() 177
◊ sortItems() 179, 183
QListWidgetItem 176
◊ clone() 176
◊ operator<() 179
◊ setFlags() 178
◊ setIcon() 177
QLocate
◊ name() 436
◊ system() 436
qLowerBound() 92
QMacCocoaView
◊ setCocoaView() 635
QMacCocoaViewContainer 634
QMacStyle 373
QMainWindow 475
◊ addDockWidget() 480
◊ addToolBar() 478
◊ centralWidget() 476
◊ menuBar() 475
◊ setCentralWidget() 476, 487, 491
◊ statusBar() 476
qmake 57, 59, 62
◊ OBJECTIVE_SOURCES 633
QMap<K,T> 87, 88
QMapIterator 77
QMax() 69
QMdiArea 488
◊ cascadeSubWindows() 488
◊ setHorizontalScrollBarPolicy() 491
◊ setVerticalScrollBarPolicy() 491
◊ subWindowList() 488
◊ titleSubWindows() 488

QMediaPlayer 394
◊ duration() 397, 401
◊ durationChanged() 397
◊ fromAscii() 704
◊ pause() 399
◊ PausedState 399
◊ play() 399
◊ PlayingState 399
◊ positionChanged() 397
◊ setMedia() 394, 397
◊ setPosition() 400
◊ setVideoOutput() 403
◊ setVolume() 396
◊ state() 399
◊ stateChanged() 397
◊ stop() 396
◊ StoppedState 399
◊ toAscii() 704
QMenu 439, 499
◊ addAction() 145, 442
◊ addSeparator() 442
◊ clear() 495
◊ exec() 443
◊ setChecked() 442
◊ triggered() 443
QMessageBox 462
◊ Abort 464
◊ aboutQt() 466
◊ AcceptRole 464
◊ ActionRole 464
◊ addButton() 463
◊ Cancel 464
◊ Critical 463
◊ critical() 465
◊ DestructiveRole 464
◊ exec() 463
◊ Help 464
◊ HelpRole 464
◊ Ignore 464
◊ Information 463
◊ information() 464
◊ No 464
◊ NoButton 464
◊ NoIcon 463
◊ NoRole 464
◊ NoToAll 464
◊ Ok 464
◊ Question 463
◊ RejectRole 464
◊ ResetRole 464
◊ Retry 464
◊ SaveAll 464
◊ setEscapeButton() 463
◊ setIcon() 463
◊ setIconPixmap() 463
◊ setText() 463
◊ StandardButton 463
◊ Warning 463
◊ warning() 464
◊ Yes 464
◊ YesToAll 464
QMetaObject 53
◊ invokeMethod() 890
QMimeData
◊ hasFormat() 419, 426
◊ setColorData() 415
◊ setData() 416, 421
◊ setHtml() 416
◊ setImageData() 415, 421
◊ setText() 416
◊ setUrls() 416
◊ urls() 420
QMimeSource 415
QMin() 69
QML 772, 774
◊ anchors 806, 808, 809
◊ anchors.bottom 810
◊ anchors.bottomMargin 812
◊ anchors.centerIn 809
◊ anchors.fill 808, 809, 838
◊ anchors.horizontalCenter 807, 810
◊ anchors.left 810, 811
◊ anchors.leftMargin 812
◊ anchors.right 810, 811
◊ anchors.rightMargin 812
◊ anchors.top 810
◊ anchors.topMargin 812
◊ anchors.verticalCenter 807, 810
◊ Animation.Infinite 854, 860
◊ Behavior 858
◊ bool 791
◊ BorderImage 785, 825, 843
◊ BorderImage.bottom 825
◊ BorderImage.left 825
◊ BorderImage.right 825
◊ BorderImage.top 825
◊ color 791
◊ ColorAnimation 855, 856
◊ ColorAnimation.duration 856
◊ ColorAnimation.from 856
◊ ColorAnimation.running 856
◊ ColorAnimation.to 856
◊ Column 813, 877, 882
◊ Component 877
◊ date 791
◊ Easing.InBounce 868
◊ Easing.InCirc 868
◊ Flickable 796
◊ Flickable.contentHeight 796
◊ Flickable.contentWidth 796
◊ font 828
◊ font.bold 828
◊ font.pixelSize 828
◊ Gradient 826, 878
◊ GradientStop 826
◊ GradientStop.color 826
◊ GradientStop.position 826
◊ Grid 813
◊ Grid.columns 816
◊ Grid.rows 816
◊ GridView 786, 877, 880, 882
◊ Image 785, 821, 823, 858, 877
◊ Image.rotation 822, 823
◊ Image.scale 822
◊ Image.smooth 823
◊ Image.source 821
◊ Image.transform 823
◊ Image.x 823
◊ Image.y 823
◊ int 791
◊ Item 785, 786, 792, 877
◊ Item.anchors 788
◊ Item.antialiasing 788
◊ Item.children 788
◊ Item.childrenRect 788
◊ Item.clip 788
◊ Item.enable 788
◊ Item.height 786, 788, 789
◊ Item.id 788
◊ Item.implicitHeight 788
◊ Item.implicitWidth 788
◊ Item.onHeightChanged 790
◊ Item.onWidthChanged 790
◊ Item.opacity 788
◊ Item.parent 788, 789, 806
◊ Item.position 788
◊ Item.rotation 788
◊ Item.scale 788
◊ Item.smooth 788
◊ Item.state 788, 864
◊ Item.states 788
◊ Item.transform 788
◊ Item.transformOrgin 788
◊ Item.transitions 788, 867
◊ Item.visible 788
◊ Item.visibleChildren 788
◊ Item.width 788, 789
◊ Item.x 786, 788, 789
◊ Item.y 786, 788, 789
◊ Item.z 788
◊ JavaScript 863
◊ KeyNavigation.down 848
◊ KeyNavigation.left 848

QML *(прод.)*
◊ KeyNavigation.right 848
◊ KeyNavigation.tab 847
◊ KeyNavigation.up 848
◊ Keys.onDownPressed 849
◊ Keys.onLeftPressed 849
◊ Keys.onPressed 849
◊ Keys.onRightPressed 849
◊ Keys.onUpPressed 849
◊ list 791
◊ ListElement 873
◊ ListModel 873, 874
◊ ListModel.insert() 874
◊ ListModel.move() 874
◊ ListModel.remove() 874
◊ ListView 786, 877
◊ ListView.delegate 877
◊ ListView.footer 878
◊ ListView.header 878
◊ ListView.highlight 878
◊ ListView.model 878
◊ MouseArea 780, 838, 858, 862
◊ MouseArea.acceptedButtons 839
◊ MouseArea.hoverEnabled 839
◊ MouseArea.mouse 839
◊ MouseArea.mouseX 842
◊ MouseArea.mouseY 842
◊ MouseArea.onClicked 843, 862, 864
◊ MouseArea.onEntered 840
◊ MouseArea.onMousePosition Changed 842, 858
◊ MouseArea.onPressed 838, 843, 844
◊ MouseArea.onReleased 838, 843, 844
◊ NumberAnimation 855, 858, 862
◊ OnExited 840
◊ origin.x 823
◊ origin.y 823
◊ ParallelAnimation 860
◊ ParallelAnimation.loop 860
◊ Path 882
◊ Path.startY 882
◊ PathAttribute 884
◊ PathQuad 884
◊ PathView 786, 877, 882
◊ PathView.pathItemCount 882
◊ property 790
◊ PropertyAnimation 854
◊ PropertyAnimation.duration 854
◊ PropertyAnimation.from 854
◊ PropertyAnimation.loop 854
◊ PropertyAnimation.properties 854
◊ PropertyAnimation.target 854
◊ PropertyAnimation.to 854
◊ PropertyChanges 864
◊ Qt Creator 780
◊ real 791
◊ Rectangle 785, 786, 793, 806, 822, 877, 889
◊ Rectangle.border.color 787
◊ Rectangle.border.width 787
◊ Rectangle.color 786, 788, 847
◊ Rectangle.gradient 826
◊ Rectangle.opacity 810
◊ Rectangle.radius 787
◊ Rectangle.rotation 826
◊ Rectangle.scale 826
◊ Rectangle.smoth 787
◊ Rotation 823
◊ Rotation.angle 823
◊ RotationAnimation 855, 857, 862
◊ RotationAnimation.Clockwise 857, 862
◊ RotationAnimation. Counterclockwise 857
◊ RotationAnimation.direction 862
◊ RotationAnimation.Shortest 857
◊ Row 813, 814, 877
◊ Scale 823
◊ Scale.xScale 823
◊ Scale.yScale 823
◊ SequentialAnimation 860, 862
◊ signal 841
◊ string 791
◊ Text 785, 791, 793, 806, 828, 843, 877, 889
◊ Text.text 793
◊ TextEdit 786, 845
◊ TextInput 786, 845
◊ TextInput.focus 845, 846
◊ time 791
◊ Transition 867
◊ Transition.to 867
◊ Transtion.from 867
◊ url 791
◊ var 791
◊ vector2d 791
◊ vector3d 791
◊ vector4d 791
◊ WebView 786
◊ Window 779, 786
◊ XmlListModel 874, 875
◊ XmlListModel.query 876
◊ XmlListModel.source 875
◊ XmlRole 876
qmlRegisterType<T>() 897
QModelIndex 197
◊ isValid() 197
QMouseEvent 222, 248, 252, 308
◊ button() 228, 248
◊ globalPos() 228
◊ globalX() 228
◊ globalY() 228
◊ pos() 228
◊ x() 228, 481
◊ y() 228, 481
QMoveEvent 234
◊ oldPos() 234
◊ pos() 234
QMovie 330
◊ frameRect() 331
◊ NoRunning 330
◊ Paused 330
◊ Running 330
◊ setMovie() 331
◊ setPaused() 330
◊ setSpeed() 330
◊ start() 330
◊ state() 330
◊ stop() 330
QMultiHash<K,T> 89
QMultiMap<K,T> 89
QMutex 548
◊ lock() 548
◊ tryLock() 548
◊ unlock() 548
QMutexLocker 549
QMYSQL 602
QM-файл 433
QNetworkAccessManager 570, 580, 705
QNetworkConfigurationManager 581
◊ isOnline() 581
◊ onlineStateChanged() 581
QNetworkProxy 580
◊ setApplicationProxy() 580
QNetworkReply 570
◊ downloadProgress() 570
◊ error() 570
◊ finished() 570
◊ readyRead() 570
◊ uploadProgress() 570
QNetworkRequest 570
◊ setPriority() 573
QObject 35, 44, 51, 246
◊ blockSignals() 46
◊ child() 234
◊ childEvent() 234

◊ children() 52
◊ connect() 45, 49, 127, 158, 448, 472
◊ customEvent() 236
◊ deleteLater() 560
◊ disconnect() 49
◊ dumpObjectInfo() 53, 68
◊ dumpObjectTree() 53
◊ event() 236
◊ eventFilter() 246, 247, 253
◊ findChild() 52, 890
◊ findChildren() 52
◊ inherits() 53
◊ installEventFilter() 246, 247, 252
◊ killTimer() 528
◊ moveToThread() 540
◊ objectName() 51, 68
◊ parent() 52
◊ sender() 44
◊ setObjectName() 51
◊ setParent() 51
◊ setProperty() 890
◊ startTimer() 528
◊ thread() 540
◊ timerEvent() 233, 528
◊ timerId() 528
◊ tr() 429, 430
qobject_cast<T> 622
qobject_cast<T>() 36
QObjectList 52
QOCI 602
QODBC 602
QOpenGLContext
◊ currentContext() 348
◊ functions() 348
QOpenGLWidget
◊ glClearColor() 348
◊ initializeGL() 347
◊ paintGL() 347
◊ resizeGL() 347
QPageLayout
◊ Landscape 359
◊ Portrait 359
QpageSize 360
◊ A0 360
◊ A1 360
◊ A2 360
◊ A3 360
◊ A4 360
◊ A5 360
◊ A6 360
◊ A7 360
◊ A8 360
◊ A9 360
◊ B0 360
◊ B1 360
◊ B10 360
◊ B2 360
◊ B3 360
◊ B4 360
◊ B5 360
◊ B6 360
◊ B7 360
◊ B8 360
◊ B9 360
◊ C5E 360
◊ Comm10E 360
◊ DLE 360
◊ Executive 360
◊ Folio 360
◊ Ledger 360
◊ Tabloid 360
QPaintDevice 267, 268, 359
QPaintEngine 267
QPainter 267, 268, 902
◊ begin() 268, 304
◊ CompositionMode_SourceOver 287
◊ drawEllipse() 304, 364
◊ drawImage() 300, 304
◊ drawLine() 364
◊ drawPath() 284
◊ drawPicture() 281
◊ drawPixmap() 756
◊ drawPolyLine() 276
◊ drawRect() 313, 364
◊ drawText() 312, 315, 364, 369
◊ end() 268, 304
◊ resetTransform () 757
◊ restore() 270, 282
◊ rotate() 282
◊ save() 270, 282
◊ scale() 282
◊ setBrush() 271, 364
◊ setClipPath() 286
◊ setClipRect() 285
◊ setClipRegion() 286
◊ setCompositionMode() 287
◊ setFont() 312, 364
◊ setPen() 270, 271, 364
◊ setRenderHint() 274, 304
◊ setTransform() 283
◊ shear() 282
◊ translate() 282, 757
◊ viewport() 363
QPainterPath 284
QPaintEvent 224, 227, 267
◊ rect() 267
◊ region() 227, 267
QPalette 217
◊ Active 217
◊ Base 218
◊ BrightText 218
◊ Button 218
◊ ButtonText 218
◊ Dark 218
◊ Disabled 217
◊ Highlight 218
◊ HighlightedText 218
◊ Inactive 217
◊ Light 218
◊ Link 218
◊ LinkVisited 218
◊ Mid 218
◊ Midlight 218
◊ setBrush() 219
◊ setColor() 151, 219
◊ Shadow 218
◊ Text 218
◊ Window 218
◊ WindowText 218
QParallelAnimationGroup 335
QPauseAnimation 333
QPen 270
◊ setCapStyle() 271
◊ setColor() 271
◊ setJoinStyle() 271
◊ setWidth() 270
QPicture 281
QPixmap 305
◊ defaultDepth() 305
◊ drawPixmap() 305
◊ fill() 757
◊ grabWindow() 505
◊ load() 135, 305
◊ rotate() 757
◊ save() 305, 453
◊ scaled() 505
QPixmapCache 306
◊ find() 306
◊ insert() 306
QPlainTextEdit 161
QPluginLoader 619, 622
◊ instance() 619, 622
◊ unload() 619
QPoint 256, 257
◊ isNull() 257
◊ manhattanLength() 257
◊ rx() 257
◊ setX() 257
◊ setY() 257
◊ x() 257
◊ y() 257
QPointF 256, 257
QPolygon 260

QPolygonF 260
QPrintDialog 359, 454
◊ exec() 363
QPrinter 164, 359
◊ abort() 360
◊ Color 359
◊ Grayscale 359
◊ HighResolution 164
◊ Legal 360
◊ Letter 360
◊ NativeFormat 164
◊ newPage() 363
◊ PdfFormat 360
◊ PostScriptFormat 164
◊ setColorMode() 359
◊ setCopyCount() 359
◊ setDocName() 360
◊ setFromTo() 359
◊ setMinMax() 363
◊ setOutputFileName() 164, 360
◊ setOutputFormat() 360
◊ setPageOrientation() 359
◊ setPageSize() 360
QProcess 533
◊ CrashExit 534
◊ errorOccurred() 534
◊ exitStatis() 534
◊ finished() 534
◊ NormalExit 534
◊ readAllStandardError() 534
◊ readAllStandardOutput() 534
◊ readyRead() 534
◊ readyReadStandardError() 534
◊ readyReadStandardOutput() 534
◊ start() 534, 535
◊ started() 534
QProgressBar 137
◊ reset() 138
◊ setOrientation() 137
◊ setProgress() 158
◊ setRange() 138
◊ setValue() 138
QProgressDialog 459
◊ canceled() 459
◊ reset() 460
◊ setAutoClose() 460
◊ setAutoReset() 460
◊ setMinimumDuration() 459
◊ setProgess() 459
◊ setTotalSteps() 459
◊ setWindowTitle() 460
◊ wasCanceled() 460
QPropertyAnimation 333
QPSQL 602
QPushButton 127, 143
◊ clicked() 47, 127, 308, 450, 472
◊ setFlat() 144
◊ setMenu() 145
QQmlApplicationEngine 893
QQmlComponent
◊ create() 893
QQmlComponent 893
QQmlContext 894, 895
◊ setContextProperty() 895
QQmlEngine 894
QQueue<T> 86
QQuickImageProvider 904
QQuickItem 890, 901
QQuickPaintedItem 901
QQuickRectangle 901
QQuickText 901
QQuickWidget 894
◊ rootContext() 894
◊ rootObject() 890
QRadialGradient 273
QReadWriteLock
◊ lockForRead() 550
◊ lockForWrite() 550
QReadWriteLock 550
QRect 107, 259
◊ height() 259
◊ setHeight() 259
◊ setSize() 259
◊ setWidth() 259
◊ setX() 259
◊ setY() 259
◊ size() 259
◊ width() 259
◊ x() 259
◊ y() 259
QRectF 259
qRed() 262, 298
QRegExp 97, 173
QRegion
◊ intersected() 286
◊ subtracted() 286
◊ united() 286
◊ xored() 286
QResizeEvent 234
◊ oldSize() 234
◊ size() 234
QResource 62
◊ data() 62
QRgb 262, 298
qRgb() 262
qRgba() 262, 298
qRound() 70
QScreen 958
◊ primaryOrientationChanged() 958
QScrollArea
◊ removeChild() 114
◊ setWidget() 114
◊ widget() 114
QScrollBar 152, 155
◊ sliderMoved() 153
◊ valueChanged() 153, 367
QSemaphore 549
◊ acquire() 549
◊ release() 549
◊ tryAcquire() 550
QSequentialAnimationGroup 335
QSet<T> 90
QSettings
◊ beginGroup() 407, 411
◊ endGroup() 407, 410
◊ readEntry() 410
◊ remove() 407
◊ setValue() 406
◊ value() 407
◊ writeEntry() 411
QShowEvent 234
QSignalMapper 490
QSignalTransition 342
QSize 257
◊ height() 258
◊ isNull() 257, 259
◊ rheight() 258
◊ rwidth() 258
◊ scale() 258
◊ setHeight() 258
◊ setWidth() 258
◊ width() 258
QSizeF 257
QSizeGrip 476
QSizePolicy
◊ Expanding 367
◊ Fixed 366
◊ Ignored 367
◊ Maximum 366
◊ Minimum 366
◊ MinimumExpanding 366
◊ Preferred 366
QSlider 152, 153
◊ NoTicks 153
◊ setTickInterval() 154
◊ setTickmarks() 155
◊ setTickPosition() 153
◊ TicksAbove 153
◊ TicksBelow 153
◊ TicksBothSides 153
QSocket
◊ connectToHost() 563
qSort() 92
QSortFilterProxyModel 212
◊ setFilterRegExp() 212

QSound 393
◊ isFinished() 393
◊ loopsRemaining() 393
◊ play() 393
◊ setLoops() 393
◊ stop() 393
QSpinBox
◊ setPrefix() 171
◊ setRange() 171
◊ setSpecialValueText() 171
◊ setSuffix() 171
◊ setValue() 171
◊ setWrapping() 171
◊ stepDown() 171
◊ stepUp() 171
◊ value() 171
◊ valueChanged() 171
QSplashScreen 482
◊ finish() 482
◊ show() 482
◊ showMessage() 482
QSplitter 129
QSqlDatabase 602
◊ addDatabase() 603
◊ lastError() 603
◊ open() 603
◊ setDatabaseName() 603
◊ setHostName() 603
◊ setUserName() 603
◊ tables() 603
QSqlDriver 601
QSqlDriverCreator<T*> 601
QSqlDriverCreatorBase 601
QSqlDriverPlugin 601, 617
QSqlError 602, 603
◊ isValid() 607
◊ text() 603
QSqlField 602
◊ length() 607
◊ name() 607
◊ type() 607
◊ value() 607
QSqlIndex 602
QSQLITE 602
QSQLITE2 602
QSqlQuery 602, 604
◊ bindValue() 604
◊ exec() 604
◊ first() 604
◊ last() 604
◊ next() 604
◊ prepare() 604
◊ previous() 604
◊ record() 606
◊ seek() 604
◊ setQuery() 607
◊ size() 604
QSqlQueryModel 602
◊ lastError() 607
QSqlRecord 602, 606
◊ contains() 607
◊ count() 607
◊ field() 607
◊ fieldName() 607
◊ indexOf() 606
QSqlRelation 611
QSqlRelationalTableModel 602, 610
QSqlRelationTableModel
◊ setRelation() 611
QSqlResult 601
QSqlTableModel 602, 608
◊ insertRow() 610
◊ OnFieldChange 609
◊ OnManualSubmit 609
◊ OnRowChange 609
◊ removeColumn() 608
◊ removeRows() 610
◊ revertAll() 609
◊ rowCount() 609
◊ setData() 610
◊ setEditStrategy() 609
◊ setFilter() 609
◊ setSort() 609
◊ submitAll() 609
QSslSocket
◊ waitForEncrypted() 577
qStableSort() 92
QStack<T> 86
◊ pop() 86
◊ push() 86
◊ top() 86
QStackedWidget 112
◊ addWidget() 112
◊ indexOf() 112
◊ setCurrentIndex() 112
◊ setCurrentWidget() 112
QStandardItemModel 191
QStandardPaths 506
◊ MoviesLocation 506
◊ PicturesLocation 506
◊ writableLocation() 506
QState 342
◊ addTransition() 342
◊ assignProperty() 342
QStateMachine 342
◊ setInitialState() 342
◊ start() 342
QStatusBar 480
◊ addPermanentWidget() 480
◊ addWidget() 480, 481
◊ cleaMessage() 480
◊ removeWidget() 480
◊ showMessage() 480, 487, 491
QString 95
◊ append() 95
◊ arg() 604
◊ contains() 173, 453
◊ isEmpty() 95, 485
◊ length() 95
◊ number() 96, 235
◊ remove() 443
◊ replace() 96
◊ setNum() 96
◊ split() 515
◊ toLower() 96
◊ toUpper() 96
◊ конвертирование 632, 636
QStringList
◊ filter() 99
◊ join() 96
◊ split() 96
QStringListModel 191, 895
QStyle 372
◊ drawComplexControl() 377
◊ drawControl() 377
◊ drawPrimitive() 377
◊ polish() 380
◊ State_MouseOver 196
◊ unpolish() 381
QStyleFactory 618
◊ create() 376
QStyleItemDelegate
◊ sizeHint() 196
QStyleOptionViewItem 195
QStylePlugin 617, 618
◊ create() 618
QSvgRenderer 332, 333
◊ repaintNeeded() 332
QSvgWidget 332
◊ load() 332
qSwap() 92
QSyntaxHighlighter 162, 165, 167
◊ highlightBlock() 165
◊ previousBlockState() 168
◊ setCurrentState() 169
QSysInfo 637
◊ MacintoshVersion 637
◊ MV_SNOWLEOPARD 637
◊ WindowsVersion 637
◊ WV_VISTA 637
◊ WV_WINDOWS10 637
◊ WV_WINDOWS7 637
◊ WV_WINDOWS8_1 637
◊ WV_XP 637
QSystemTrayIcon 498, 499
◊ setContextMenu() 498
◊ setIcon() 498, 502
◊ setToolTip() 498, 500
◊ showMessage() 498, 501

Qt 29
◊ AlignBottom 133
◊ AlignCenter 133, 230
◊ AlignHCenter 132
◊ AlignJustify 132
◊ AlignLeft 132
◊ AlignRight 132
◊ AlignTop 132
◊ AlignVCenter 133
◊ AltModifier 224
◊ ArrowCursor 109
◊ AscendingOrder 179
◊ AutoConnection 44, 540
◊ BackgroundColorRole 202
◊ BDiagPattern 271
◊ black 265
◊ BlanckCursor 110
◊ blue 265
◊ Checked 180
◊ ClosedHandCursor 110
◊ color0 265, 306
◊ color1 265, 306
◊ ConicalGradientPattern 271
◊ ControlModifier 224
◊ CrossCursor 110
◊ CrossPattern 271
◊ cyan 265
◊ darkBlue 265
◊ darkCyan 265
◊ darkGray 265
◊ darkGreen 265
◊ darkMagenta 266
◊ darkRed 265
◊ darkYellow 266
◊ DashDotDotLine 270
◊ DashDotLine 270
◊ DashLine 270
◊ DecorationRole 202
◊ Dense1Pattern 271
◊ Dense2Pattern 271
◊ Dense3Pattern 271
◊ Dense4Pattern 271
◊ Dense5Pattern 271
◊ Dense6Pattern 271
◊ Dense7Pattern 271
◊ DescendingOrder 179
◊ DiagCrossPattern 271
◊ DirectConnection 44
◊ DisplayRole 202
◊ DotLine 270
◊ ElideLeft 315
◊ ElideMiddle 315
◊ ElideNone 315
◊ ElideRight 315
◊ FDiagPattern 271
◊ FlatCap 271
◊ FontRole 202
◊ ForbiddenCursor 110
◊ gray 265
◊ green 265
◊ Horizontal 152
◊ HorPattern 271
◊ IbeamCursor 110
◊ IgnoreAspectRatio 258, 301
◊ ISODate 525
◊ ItemIsUserCheckable 180
◊ KeepAspectRatio 258, 301
◊ KeepAspectRatioByExpanding 258
◊ LeftButton 229
◊ lightGray 265
◊ LinearGradientPattern 271
◊ magenta 265
◊ MidButton 229
◊ NoBrush 271
◊ NoButton 229
◊ NoModifier 224, 250
◊ NoPen 270
◊ OpenHandCursor 110
◊ PointingHandCursor 110
◊ QueuedConnection 44
◊ RadialGradientPattern 271
◊ red 265
◊ RightButton 229
◊ RoundCap 271
◊ ScreenOrientation 959
◊ ShiftModifier 224
◊ SizeAllCursor 110
◊ SizeBDiagCursor 110
◊ SizeFDiagCursor 110
◊ SizeHorCursor 110
◊ SizeVerCursor 110
◊ SolidLine 270
◊ SolidPattern 271
◊ SplitHCursor 110
◊ SplitVCursor 110
◊ SquareCap 271
◊ SystemLocaleLongDate 525
◊ SystemLocaleShortDate 525
◊ TextColorRole 202
◊ TextDate 525
◊ TextDontClip 313
◊ TextExpandTabs 313
◊ TextShowMnemonic 313
◊ TextSingleLine 313
◊ TexturePattern 271
◊ TextWordWrap 313
◊ ToolTip 470
◊ ToolTipRole 202
◊ TouchPointMoved 240
◊ TouchPointPressed 240
◊ TouchPointReleased 240
◊ TouchPointStationary 240
◊ UpArrowCursor 110
◊ VerPattern 271
◊ Vertical 137, 152
◊ WA_AcceptsTouchEvents 239
◊ WA_DeleteOnClose 493
◊ WA_Hover 195, 380
◊ WA_TranslucentBackground 307
◊ WaitCursor 110
◊ WhatsThisCursor 110
◊ WhatThisRole 202
◊ white 265
◊ WindowStaysOnTopHint 107
◊ yellow 265
Qt 3 703
Qt 4 703
Qt Assistant 55, 703
Qt Creator 675, 677, 683, 694
◊ Locator 688
◊ автоматическое дополнение кода 686
◊ автоформатирование 691
◊ отладчик 697
◊ подсветка синтаксиса 685
◊ поиск и замена 687
◊ пользовательский интерфейс 681
◊ сокрытие и отображение кода 686
◊ типы проектов 677
Qt Designer 383, 640, 676, 678, 683
Qt Help 675
Qt Linguist 428, 432
Qt Quick 772
Qt WebEngine 669
Qt WebEngineWidgets 667
Qt3Support 703
QTableView 193, 608
QTableWidget 183, 213
◊ setCellWidget() 183
◊ setItem() 184
QTableWidgetItem
◊ clone() 183
◊ setIcon() 183
◊ setText() 183
QTabWidget
◊ addTab() 187
◊ setCurrentIndex() 186
◊ setTabEnabled() 186
QTarnsform
◊ rotate() 327

QtConcurrent 551
◊ mapped() 553
◊ run() 552
QTcpServer 557
QTcpSocket
◊ connected() 563
◊ connectToHost() 579
◊ disconnectFromHost() 579
◊ error() 564
◊ nextPendingConnection() 579
◊ readyRead() 563
◊ waitForDisconnected() 579
QtDebugMsg 67
QTDS 602
QTemporaryFile 512
QTest
◊ addColumn() 659
◊ keyClick() 661
◊ keyClicks() 661
◊ keyPress() 661
◊ keyRelease() 661
◊ mouseClick() 662
◊ mouseDClick() 662
◊ mouseMove() 663
◊ mousePress() 662
◊ mouseRelease() 662
◊ newRow() 659
QTEST_MAIN() 657, 661
QTestEventList 663
◊ simulate() 663
qtestlib 706
QTextBrowser 471
◊ backward() 472
◊ backwardAvailable() 472
◊ forward() 472
◊ forwardAvailable() 472
◊ home() 472
◊ setSearchPath() 472
◊ setSource() 472
QTextCodecPlugin 617
QTextCursor 161
QTextDocument 164
◊ redo() 162
◊ undo() 162
QTextDocumentWriter
◊ setFileName() 164
◊ setFormat() 164
QTextEdit 161
◊ clear() 161
◊ copy() 161
◊ cut() 161
◊ deselect() 161
◊ document() 162
◊ find() 162
◊ insertHtml() 162
◊ insertPlainText() 162
◊ paste() 161
◊ redoAvailable() 162
◊ selectAll() 161
◊ selectionChanged() 161
◊ setDocument() 162
◊ setHtml() 161–163, 443
◊ setPlainText() 130, 161, 162, 485
◊ setReadOnly() 161
◊ text() 161
◊ textChanged() 161
◊ textCursor() 162
◊ toPlainText() 486
◊ undoAvailable() 162
◊ zoomIn() 162
◊ zoomOut() 162
QTextStream 521
◊ readAll() 485, 521
◊ readLine() 521
◊ setRealNumberPrecision() 521
QtFatalMsg 67
QtGraphicalEffect
◊ Blend 834
◊ BrightnessContrast 834
◊ Colorize 834
◊ ColorOverlay 834
◊ ConicalGradient 835
◊ Desaturate 835
◊ DirectionalBlur 835
◊ Displace 835
◊ DropShadow 835
◊ FastBlur 835
◊ GammaAdjust 835
◊ GaussianBlur 835
◊ Glow 835
◊ HueSaturation 835
◊ InnerShadow 835
◊ LevelAdjust 835
◊ LinearGradient 835
◊ MaskedBlur 835
◊ OpacityMask 835
◊ RadialBlur 835
◊ RadialGradient 835
◊ RecangularGlow 835
◊ ResursiveBlur 835
◊ ThresholdMask 835
◊ ZoomBlur 835
QtGui 704
QThread 540
◊ exec() 528, 539
◊ exit() 539
◊ priority() 538
◊ quit() 539, 542
◊ run() 538
◊ setPriority() 538
◊ start() 538, 543
◊ usleep() 543
◊ wait() 542
QTime 526
◊ addMSecs() 526
◊ addSecs() 526
◊ currentTime() 526
◊ elapsed() 526
◊ fromString() 526
◊ hour() 526
◊ minute() 526
◊ msec() 526
◊ second() 526
◊ setHMS() 526
◊ start() 526
◊ toString() 526, 560
QTimeLine 333
QTimer 333, 530
◊ isActive() 530
◊ setInterval() 530
◊ singleShot() 530
◊ stop() 530
◊ timeout() 530
QTimerEvent 233
QtMsgType 67
QtMultimedia 392
QToolBar 478
◊ addAction() 478
◊ addSeparator() 478
◊ addWidget() 478
QToolBox 187
◊ addItem() 187
◊ count() 187
◊ currentChanged() 187
◊ insertItem() 187
◊ removeItem() 187
QTouchDevice 240
◊ TouchPad 240
◊ TouchScreen 240
QTouchEvent 239, 240
◊ device() 240
◊ TouchPoint 240
QtQuick
◊ BusyIndicator 799
◊ CircularGauge 801
◊ ColumnLayout 813
◊ ComboBox 799
◊ DelayButton 801
◊ Dial 802
◊ Dial 799
◊ Gauge 802
◊ GridLayout 813
◊ Label 799
◊ PageIndicator 799
◊ PieMenu 802
◊ ProgressBar 799
◊ RangeSlider 799

QtQuick *(прод.)*
◊ RowLayout 813
◊ Slider 799
◊ SpinBox 799
◊ StackLayout 813
◊ StatusIndicator 802
◊ ToggleButton 802
◊ ToolBar 799
◊ Tumbler 799, 802
QtQuickParticles
◊ Age 871
◊ Friction 871
◊ Turbulence 872
◊ Wander 872
QTransform 327
◊ scale() 327
◊ translate() 327
QTranslator 428, 433
◊ load() 433
QTreeView 193
QTreeWidget 180, 213
◊ itemClicked() 182
◊ itemDoubleClicked() 182
◊ itemSelectionChanged() 182
◊ setHeaderLabels() 181
◊ setSortingEnabled() 182
QTreeWidgetItem 180
◊ clone() 180
◊ operator<() 183
◊ setDragEnabled() 182
◊ setExpanded() 181
◊ setIcon() 180
◊ setText() 180
QtSvg 332
QtUiTools 651
QtWarningMsg 67
QtXmlPatterns 594
QUdpSocket 566
◊ bind() 567
◊ readDatagram() 567
◊ readyRead() 567
◊ writeDatagram() 566
QUiLoader 651
qUncompress() 83, 511
qUpperBound() 92
QUrl 667
◊ fromLocalFile() 394
◊ toString() 420
QValidator 173
◊ Acceptable 173
◊ Intermediate 173
◊ Invalid 173
◊ validate() 173
QVariant 100, 890
◊ toInt() 206, 606
◊ toString() 606
◊ type() 100
◊ typeName() 100
◊ typeToName() 100
◊ value<T>() 206
QVariantAnimation
◊ setDuration() 334
◊ setEndValue() 334
◊ setKeyValueAt() 334
◊ setStartValue() 334
QVBoxLayout 118, 121, 122
QVector<T>
◊ data() 82
◊ fill() 82
◊ insert() 82
◊ push_back() 82
◊ reserve() 82
◊ resize() 82
◊ toList() 82
◊ toStdVector() 82
QVectorIterator 77
QVERIFY() 662
QVideoWidget 402
QWaitCondition 550
◊ wait() 550
◊ wakeAll() 550
◊ wakeOne() 550
qWarning() 67, 68, 69
QWebEngineHistory 669
◊ back() 669
◊ canGoBack() 674
◊ canGoForward() 674
◊ clear() 669
◊ forward() 669
QWebEngineView 667, 669, 670
◊ load() 667, 673
◊ loadFinished() 670
◊ loadProgress() 670, 672
◊ stop() 672
QWebHistoryItem 669
QWheelEvent 232
◊ angleDelta() 232
◊ buttons() 232
◊ globalPos() 232
◊ pixelDelta() 232
◊ pos() 232
QWidget 104, 115, 151, 225
▫ setTabOrder() 129
◊ activateWindow() 447
◊ backgroundRole() 150
◊ close() 234
◊ closeEvent() 234, 411, 499
◊ contextMenuEvent() 443
◊ dragEnterEvent() 233, 419
◊ dragLeaveEvent() 233
◊ dragMoveEvent() 233
◊ dropEvent() 234, 419, 420
◊ enterEvent() 233
◊ event() 225
◊ focusInEvent() 227
◊ focusOutEvent() 227
◊ geometry() 107
◊ height() 107, 502
◊ hide() 106, 234
◊ hideEvent() 234
◊ isEnabled() 107
◊ keyPressEvent() 224, 225
◊ keyReleaseEvent() 224, 225
◊ leaveEvent() 233
◊ mouseDoubleClickEvent() 107, 228, 229
◊ mouseMoveEvent() 107, 228, 230, 417, 425, 481
◊ mousePressEvent() 107, 222, 228, 230, 236, 417
◊ mouseReleaseEvent() 107, 228, 230
◊ move() 107
◊ moveEvent() 234
◊ nativeEvent() 632
◊ paintEvent() 227, 267, 369, 756
◊ palette() 217
◊ pos() 107
◊ raise() 447
◊ repaint() 227, 370, 757
◊ resize() 108, 125, 229, 235
◊ resizeEvent() 234
◊ setAcceptDrops() 419, 424
◊ setAttribute() 195, 307
◊ setAutoFillBackground 108
◊ setCursor() 112
◊ setEnabled() 107, 396, 472
◊ setFixedSize() 342, 756
◊ setFont() 310
◊ setGeometry() 104, 107
◊ setGraphicsEffect() 289, 334
◊ setMargin() 113
◊ setMinimumWidth() 138
◊ setMouseTracking() 228
◊ setObjectName() 422
◊ setPalette() 108, 151, 219
◊ setStyle() 373, 374
◊ setStyleSheet() 383
◊ setToolTip() 469
◊ setWindowFlags() 106
◊ setWindowIcon() 493
◊ setWindowTitle() 107, 422, 463, 493
◊ show() 106, 125, 227, 229, 234, 350
◊ showEvent() 234
◊ showFullScreen() 350

◊ size() 107
◊ sizeHint() 366, 636
◊ sizePolicy() 366
◊ style() 377
◊ update() 227
◊ wheelEvent() 232
◊ width() 107, 502
◊ winId() 629
◊ x() 107
◊ y() 107
◊ changeEvent() 436
QWindowsCEStyle 373
QWindowsMobileStyle 373
QWindowsStyle 373
QWindowsVistaStyle 373
QWindowsXPStyle 373
QWizard 460
QWizardPage 461
◊ setTitle() 461
QWorkspace 704
QXmlContentHandler 589
QXmlDefaultHandler 590
QXmlQuery 594
◊ bindVariable() 594
◊ evaluateTo() 594
◊ isValid() 594
◊ setQuery() 594
QXmlSimpleReader 589
◊ setContenHandler() 590
QXmlStreamReader 592
◊ atEnd() 593
◊ errorString() 593
◊ hasError() 593
◊ name() 593
◊ readNext() 593
◊ text() 593
◊ tokenString() 593

R

RAD 640
Radio Button 147
rcc 61
remove() 76
Rendering context 345
RGB 260, 261
RGBA 260

S

SAX 9, 589
Scalable Vector Graphics 332
SDI 9, 484
SFTP 557
shared data 101
Single Document Window Interface 484, 496
singleshot 530
size() 76
Skype 497
SMTP 557
sortColumn() 183
Spacer 644
Splash Screen 482
SQL 599
SQL-модель
◊ запроса 607
◊ реляционная 610
◊ табличная 608
SSH 557
Ssync 557
States 341
Step Into 698
Step Out 699
Step Over 698
STL 74, 79
SVG 332
systat 557
System Tray 497

T

TCP 555
◊ клиент 562
◊ сервер 557
Telnet 557
Thread safety 548
Toggle Button 143
Tool Tip 469
Top-level widget 104
Touchpad 232
TouchPoint
◊ id() 240
◊ lastPos() 240
◊ pos() 240
◊ pressure() 240
◊ startPos() 240
◊ state() 240
Transitions 341
Tristate Button 146
TS-файл 430, 432
Tulip 74
typeof 740
typeof() 723

U

UDP 556
◊ сервер 566
UML 865
Unicode 521
URL 416

V

value() 76
Viber 497
Viewport 349
VNC 557
VoIP 14

W

W3C 582
Watches 700
Web 557, 665
WebEngine 667
WebKit 665
WhatsApp 497
WHOIS 557
Widgets 104
windeployqt 619
Windows 637, 665
Windows API 38, 629
Windows Notepad 484
Workaround 712
Wrapper 715
WYSIWYG 640

X

XBM 293
XCode 57
XML 9, 24, 582, 597
XML Query Language 594
XML-документ 583, 584
◊ комментарии 583
◊ теги 583
◊ чтение 585, 589
◊ элементы 583
XPath 874
XPixmap 294
XPM 293, 294, 305
XQuery 594
◊ concat() 596
◊ count() 597
◊ empty() 597
◊ for 596
◊ order by 596
◊ return 596
◊ where 596

А

Алгоритм сжатия LZW 294
Алгоритмы 91
Анимация 330
Атрибут файла 517

Б

Баг 62
База данных 599
◊ запись 599
◊ первичный ключ 599
◊ поля 599
◊ создание таблицы 600
Буфер обмена 414

В

Веб-браузер 665, 667, 669, 670
Веб-клиент 665, 667
Вектор 82
Взаимная блокировка 551
Вид и поведение 31, 371
Виджет 104, 115
◊ абстрактного счетчика 171
◊ активное состояние 217
◊ верхнего уровня 104
◊ группировки кнопок 148
◊ кнопки 143
◊ компоновка 116
◊ неактивное состояние 217
◊ недоступное состояние 217
◊ переключателя 147
◊ состояние 386
◊ списка 176
◊ установщика 156
◊ флажка 145
◊ элемента настройки 152
◊ экрана 502
Видовое окно 349
Вкладки 186
Вложенное подменю 441
Внешние прерывания 532
Время 526
Всплывающая подсказка 469
Всплывающее меню 145
Выпадающий список 185

Г

Гарнитура шрифта 310
Главное окно приложения 475
Горячие клавиши 414, 429, 440
Градиент 272
◊ конический 273
◊ линейный 272
◊ радиальный 273
Градиенты 826
Графическое представление 316
◊ представление 318
◊ сцена 317
◊ элемент 319

Д

Дата 524
Двойная буферизация 227, 268
Действие 476
Дейтаграмма 556
Диалоговое окно 445
◊ About Qt 466
◊ ввода данных 458
◊ выбора файлов 451
◊ выбора цвета 455
◊ выбора шрифта 457
◊ критического сообщения 465
◊ мастера 460
◊ модальное 446
◊ настройки принтера 454
◊ немодальное 447
◊ правила создания 445
◊ предупреждающего сообщения 464
◊ процесса выполнения операции 459
◊ создание 447
◊ сообщения 462
◊ сообщения о программе 466
◊ сообщения об ошибке 467
Динамическая библиотека 613
Док 479

Ж

Жест
◊ масштабирования 244
◊ поворота 243
Жесты множественных касаний 240

З

Завершение программы 498
Закрытие окна виджета 411
Закулисное хранение 108

И

Иерархический список 180
Изображение двухуровневое 306
Индикатор выполнения 136
Инкапсуляция 38
Интегрированная среда разработки 675
Интерактивные действия 244
Интерактивный отладчик 694
Интернационализация 428
Интернет 665
Интерфейс 620
Итератор 76
◊ в стиле Java 77
◊ в стиле STL 78
◊ константный 77, 79

К

Кисть 271
Клавиатура 225
Клавиши
◊ быстрого вызова 440
◊ горячие 440
Класс обертки 715
Клиент-сервер 556
Ключ 406, 413
◊ значение 406, 413
◊ удаление 407
Кнопка 125, 142, 143, 151
◊ выключатель 143, 144
◊ группировка 148
◊ нажата 142
◊ нажимающаяся 143
◊ обычная 143
◊ отжата 142
◊ плоская 143, 144
◊ с изображением 143, 144
◊ флажок 145
Компоновка виджетов 116
Компоновщик link 62
Конический градиент 273
Контейнер
◊ ассоциативный 75, 86
◊ последовательный 75, 80
Контейнерные классы 75
Контекст
◊ воспроизведения 345
◊ рисования 267
Контекстное меню 443
Контролер 173
Контрольная точка 65, 66, 699
Критическая секция 547
Крэш 696
Кэш изображений 306

Л

Линейный градиент 272
Логические ошибки 697

M

MDI 488
Маска прозрачности 306
Масштабирование 283
Масштабируемая векторная графика 332
Масштабируемая гарнитура 310
Матрица
◊ моделирования 349
◊ проектирования 349
Машина состояний 341
Менеджеры компоновки 116
Меню 439
◊ верхнего уровня 439
◊ всплывающее 439, 442
◊ выпадающее 439
◊ контекстное 439, 443
◊ недоступные команды 441
Метаобъектная информация 36, 53
Метаобъектный компилятор 40, 60
Метафайл 281
Методы отладки 62
Механизм неявных общих данных 217
Многодокументный оконный интерфейс 484, 496
Многопоточность 537
Множественное касание 239
Множество 90
Модель 190, 873
◊ JSON 876
◊ XML 874
◊ индекс 197
◊ промежуточная 211
◊ списка 873
Модули Qt 28
Мультимедиа 392
Мультитач 239
Мультитач-события 239
Мышь 109, 228, 306
Мьютекс 547

H

Надпись 132
Настройки приложения 406

O

Область уведомлений 497
Обнаружение столкновений 316
Обработчики сигналов 841
Общее использование данных 101
Однодокументный оконный интерфейс 484, 496
Однострочное текстовое поле 159
Окно
◊ видовое 349
◊ заставки 482
◊ переменных 700
Открытый формат документов для офисных приложений 164
Отладчик 62
◊ GDB 62, 63, 65, 66
◊ интерактивный 694
Отсечение 285
Очередь 86
Ошибки 694
◊ времени исполнения 696
◊ компоновки 696
◊ логические 697
◊ синтаксические 695

П

Палитра 264, 266
Панель
◊ задач 497
◊ инструментов 187, 477
Перевод 430
Переключатель 147, 151
Перемещение 282
Перетаскивание 159, 415, 416
Переходы 341
Перо 270
Пиксел 293
Плоская кнопка 143
Поворот 283
Подсказка всплывающая 469
Подэлемент 385
◊ стили 385
Поиск 93
Ползунок 153
Полигон 260
Полоса прокрутки 155
Поток 537
◊ основной 537
Пределы 98
Представление 192, 318
Программа
◊ Qt Assistant 55, 703
◊ Qt Designer 58, 383, 640
◊ Qt Linguist 432
◊ процесс 533
Проект 677
Процесс 533
Прямая линия (отрезок) 259
Прямоугольник 259

P

Рабочий стол 31
Радиальный градиент 273
Разделитель 129
Размер 257
Рамка 112
Регулярное выражение 97, 173
Редактор текста 161
Режим сглаживания 267, 274
Режим совмещения 286
Ресурс 61, 670
Рисование 227, 275
◊ линий 275
◊ прямоугольников 277
◊ точек 275
◊ фигур 277

C

Свойства 36
Сглаживание 267, 274, 345
Селектор 383
Семафор 549
Сенсорный экран 239
Сервер TCP 557
Сжатие данных 83, 511
Сигнал 36, 40, 41, 222
Синтаксические ошибки 695
Синхронизация 547
Система помощи 471
Системный реестр 406
Скос 283
Словари 87
Слот 36, 40, 43, 222
Событие 222, 246, 321, 528, 544
Сокет 555
◊ дейтаграммный 555
◊ поточный 555
Сокетное соединение 555
Сопоставление сигналов 490
Сортировка 92
Состояния 386
Список 84, 176
◊ выпадающий 185
◊ иерархический 180
Сравнение 94
Стек 85, 125
◊ виджетов 112
Стиль 372
◊ встроенный 372, 373
◊ собственный 377
◊ создание 387
Строка 95
◊ состояния 480
Сцена 317
Счетчик 171

Т

Таблица 183
Таймер 36, 528, 530, 532
◊ интервал запуска 528
◊ событие 528
Тачпад 232, 240
Текст 95
Тест:
◊ графического интерфейса 661
◊ модульный 656
◊ передача данных 659
◊ системный 656
◊ создание 656
Тестирование 655
Точка 256
Трассировка программы 697

У

Указатель мыши 109
Установщик 156
Утечка памяти 545
Утилита
◊ fixqt4headers.pl 704
◊ lconvert 438
◊ lrelease 428, 438
◊ lupdate 428, 430, 438
◊ macdeployqt 293, 603, 614, 619
◊ qmake 57, 59, 62
◊ rcc 61

Ф

Файл
◊ make 57
◊ атрибуты 517
◊ время изменения 517
◊ время создания 517
◊ компоненты пути 517
◊ наблюдение за изменениями 518
◊ проекта 57, 58, 62
▫ опции 58
◊ размер 517
◊ расширение 628
▫ cpp 58, 59
▫ h 59
▫ po 438
▫ pro 57, 62
▫ qrc 61
▫ qss 383
▫ ts 438
▫ ui 58
▫ xlf 438
Фиксатор 806
Фильтр событий 36, 246
Флажок 145, 151, 441
Фокус 227
Фон 108
Формат
◊ BMP 293
◊ GIF 294
◊ JPEG 294
◊ PDF 360
◊ PNG 294
◊ XML 582, 597
◊ XPM 294
Функции обратного вызова 38
Функция 733
◊ встроенная 735
◊ объект активации 734

Х

Хэш 89

Ц

Цвет 260
◊ палитра 264
Цветовая модель 260
◊ CMYK 260, 263
◊ HSL 260
◊ HSV 260, 262
◊ RGB 260, 261
◊ RGBA 260
◊ субтрактивная 264
Цепочки вызовов 701

Ш

Шаблон 677
Шейдеры 833
Шрифт: гарнитура 310
◊ масштабируемая 310

Э

Электронный индикатор 125, 139
Элемент 319, 785
◊ визуальный 785
◊ свойства 788

Я

Язык
◊ UML 865
◊ запросов XML 594

bhv®

Qt 5.10

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ НА C++



Макс Шлее (Max Schlee) – закончил Университет прикладных наук в городе Кайзерлаутерн (Германия). Сооснователь компании NEONWAY по разработке программного обеспечения на Qt. Работал разработчиком программного обеспечения в фирмах THOMSON, Grass Valley, DigitalFilmTechnology Weiterstadt, Goober Ltd. и Advancis. Эксперт в области объектно-ориентированного проектирования, специализирующийся на C++ и Qt. Создатель более 50 программ и приложений для Windows, Mac OS X, iOS и Android. Увлеченно занимается проектами в области программирования графики, звука и анализа финансовых рынков. Является автором ряда статей, научных докладов на международных конференциях по генеративному программированию пользовательского интерфейса. Автор книг «Qt 3/4/4.5/4.8/5.3. Профессиональное программирование на C++» и др. Связаться с автором можно по адресу электронной почты Max.Schlee@neonway.com или через страницу информационной поддержки книги www.qt-book.com.

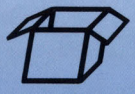
Платформно-независимая реализация приложений — это уже сегодняшний и завтрашний день программной индустрии. И эта книга станет вашим путеводителем в будущее.

Книга подробно знакомит с библиотекой Qt 5.10, являющейся не только средством для создания пользовательских интерфейсов, но и позволяющей разрабатывать приложения практически любой сложности. Недаром Qt широко используется многими организациями и компаниями, такими как ADOBE, AMAZON, AMD, BOSCH, BLACKBERRY, BMW, CANON, CISCO SYSTEMS, DISNEY, INTEL, IBM, PANASONIC, PARALLELS, PIONEER, PHILIPS, ORACLE, HP, GOOBER, GOOGLE, MERCEDES, NASA, NEC, NEONWAY, RAKUTEN, SAMSUNG, SIEMENS, SONY, SUN, TESLA, XEROX, XILINX, YAMAHA и др.

Если вы хотите идти в ногу со временем, то вам без этой книги просто не обойтись, поскольку она является исчерпывающим пособием по написанию Qt 5-программ на C++ и QML.

Библиотека Qt, не ограничивая ваш выбор и творческие возможности, делает кроссплатформенное программирование действительностью. Она предоставляет пользователям то, что им нужно — быстрые программы, которые выглядят и работают должным образом. А разработчикам программ — конструкции, позволяющие создавать меньший код и при этом получать лучший результат. Благодаря этому программирование становится интереснее вне зависимости, создаете вы коммерческий продукт или проект с открытым исходным кодом (Open Source).

Маттиас Эттрих, основатель KDE

+ 

Проекты примеров из книги можно скачать по ссылке ftp://ftp.bhv.ru/9785977536783.zip, а также со страницы книги на сайте www.bhv.ru.

191036, Санкт-Петербург,
Гончарная ул., 20
Тел.: (812) 717-10-50,
339-54-17, 339-54-28
E-mail: mail@bhv.ru
Internet: www.bhv.ru

ISBN 978-5-9775-3678-3



В ПОДЛИННИКЕ®