Юрий Земсков

Qt 4

на примерах

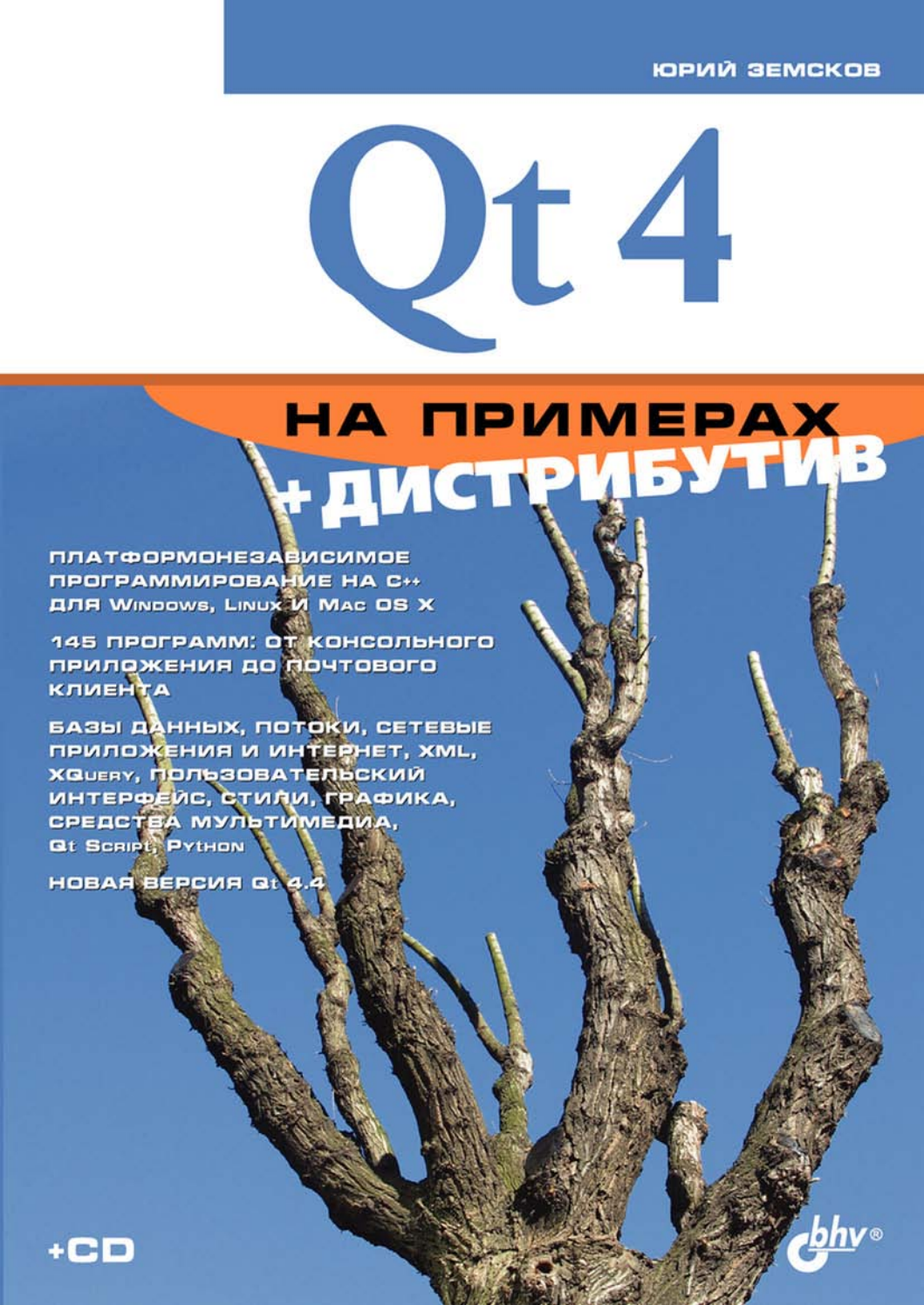
+ дистрибутив

Платформонезависимое программирование на C++ для Windows, Linux и Mac OS X

145 программ: от консольного приложения до почтового клиента

Базы данных, потоки, сетевые приложения и Интернет, XML, XQuery, пользовательский интерфейс, стили, графика, средства мультимедиа, Qt Script, Python

Новая версия Qt 4.4



+CD

bhv®

Юрий Земсков

Qt 4

НА ПРИМЕРАХ

Санкт-Петербург
«БХВ-Петербург»
2008

УДК 681.3.068+800.92Qt4
ББК 32.973.26-018.1
З-55

Земсков Ю. В.

З-55 Qt 4 на примерах. — СПб.: БХВ-Петербург, 2008. — 608 с.: ил. + Дистрибутив (на CD-ROM)

ISBN 978-5-9775-0256-6

Рассмотрена разработка приложений на языке C++ для Windows и/или Linux с применением библиотеки Qt 4, которая широко используется как в мире коммерческого, так и свободного программного обеспечения.

Приведены примеры решения многих задач, с которыми сталкивается разработчик при проектировании пользовательского интерфейса, сетевых взаимодействий, применении средств мультимедиа, языков XML и XQuery, многопоточном программировании. Особое внимание уделено созданию приложений баз данных и программ, которые могут расширяться конечным пользователем с помощью встроенных языков Qt Script или Python. Компакт-диск содержит исходные тексты описанных в книге примеров и библиотеку Qt 4.4 для Windows, Linux и Mac OS X.

Для программистов

УДК 681.3.068+800.92Qt4
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Леонид Кочин*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Игоря Цырульникова*
Оформление обложки	*Елены Беляевой*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 15.07.08.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 49,02.
Тираж 2500 экз. Заказ №
"БХВ-Петербург", 194354, Санкт-Петербург, ул. Есенина, 5Б.

Отпечатано с готовых диапозитивов
в ГУП "Типография "Наука"
199034, Санкт-Петербург, 9 линия, 12

ISBN 978-5-9775-0256-6

© Земсков Ю. В., 2008
© Оформление, издательство "БХВ-Петербург", 2008

Оглавление

Введение 9

Коммерческие и свободные версии Qt 4 12
Благодарности 13

Глава 1. Инструменты программиста 15

1.1. Microsoft Visual C++ 16
1.2. Intel C++ 17
1.3. Borland C++ 17
1.4. GNU C++ и MinGW 17
1.5. QDevelop 17
1.6. Рекомендации по выбору компилятора 19

Глава 2. "Обычный" C++ 21

2.1. Консольная программа на языке C++ 21
2.2. Компиляция и выполнение программы с помощью командной строки Microsoft Visual C++ 23
2.3. Компиляция и выполнение программы с помощью командной строки в системах Linux/FreeBSD/Solaris/HP-UX/Mac OS X 25
2.4. Создание консольного приложения в Microsoft Visual Studio 2005 26
2.5. Шаблоны в C++ 30

Глава 3. Основы Qt 4 35

3.1. Обзор классов библиотеки Qt 4 35
3.2. Компиляция библиотеки Qt 40
3.2.1. Компиляция в системе Microsoft Windows 41
3.2.2. Компиляция в системе Linux/FreeBSD 43
3.3. Простейшее приложение Qt 4 44
3.4. Компиляция приложений Qt 4 из командной строки 45

3.5. Компиляция приложений Qt 4 с помощью интегрированной среды Microsoft Visual Studio ... 47
3.6. Использование кириллицы ... 48
3.7. Консольное приложение Qt ... 52
3.8. Создание новых классов ... 54
3.9. Правила именования объектов в библиотеке Qt ... 57

Глава 4. Текстовые надписи ... 59

4.1. Форматированный текст ... 60
4.2. Ссылки Интернета ... 61
4.3. Редактируемая надпись ... 62
4.4. Надписи с рисунками ... 63
4.5. Несколько элементов на одной форме ... 64
4.6. Диалог с текстовым сообщением ... 66
4.7. Форматирование строк ... 67
4.8. Ввод текста с клавиатуры ... 68

Глава 5. Обработка событий ... 71

5.1. Сигналы и слоты ... 71
5.2. События ... 79
5.3. Обработка событий с помощью виртуальных методов ... 80
5.4. Фильтры для событий ... 84
5.5. Сопоставление сигналов ... 88

Глава 6. Кнопки и диалоги ... 91

6.1. Кнопки нажатия *QPushButton* ... 91
6.2. Программная эмуляция нажатия кнопок ... 97
6.3. Группы переключателей ... 99
6.4. Группа кнопок *QButtonGroup* ... 101
6.5. Кнопки диалогов ... 103
6.6. Модальные диалоги ... 104
6.7. Стандартные диалоги ... 108
6.8. Окно внутри диалога ... 110

Глава 7. Меню, панели инструментов и строка состояния ... 113

7.1. Строка состояния ... 113
7.2. Действия ... 115
7.3. Меню ... 115
7.4. Виджеты в строке состояния ... 116
7.5. Примеры приложений ... 118

Глава 8. Размещение элементов на форме ... 127

8.1. "Ручное" размещение элементов формы ... 127
8.2. Менеджеры размещения ... 134

Глава 9. Автоматизация создания диалогов, отладка программ и предоставление помощи 139

9.1. Создание диалога с помощью Qt Designer 139
9.2. Использование диалога, созданного в Qt Designer 145
9.3. Отладка программ 150
9.4. Система помощи 153

Глава 10. Работа с данными 159

10.1. Числа и строки 159
10.2. Массивы и списки 160
10.3. Контейнеры и итераторы 163
10.4. Класс *QObject* 165
10.5. Неявное совместное использование данных 167
10.6. Явное совместное использование данных 170

Глава 11. Каталоги, файлы, потоки ввода-вывода, ресурсы 173

11.1. Чтение и запись текстового файла 173
11.2. Работа с двоичными данными 176
11.3. Каталоги и свойства файлов 180
11.4. Временные файлы 180
11.5. Каталог приложения 181
11.6. Копирование файлов 181
11.7. Хранение ресурсов в программе 183
11.8. Хранение скомпилированных ресурсов в отдельном двоичном файле 184
11.9. Значок приложения 185
11.10. Диалог выбора файла 185
11.11. Сжатие информации 186

Глава 12. Приложения SDI и MDI 189

12.1. Пример приложения SDI: текстовый редактор 189
12.2. Открытие и сохранение файлов 194
12.3. Команды редактирования 201
12.4. Пример приложения MDI 203
12.5. Плавающие окна 211
12.6. Хранение настроек приложения 214
12.7. Разное 219

Глава 13. Графика и печать 221

13.1. Класс *QPainter* 222
13.2. Пользовательский стиль линий 227
13.3. Форматированный текст и HTML 228
13.4. Отображение форматированного текста 232
13.5. Часы со стрелками 232

13.6. Класс *QPixmap* 237
13.7. Работа с принтером 239
13.8. Печать на нескольких страницах 250
13.9. Предварительный просмотр перед печатью 252
13.10. Графические сцены 254

Глава 14. Списки, таблицы и деревья 263

14.1. Список *QListWidget* 263
14.2. Реакция на действия пользователя 264
14.3. Редактирование элементов списка 267
14.4. Таблица *QTableWidget* 267
14.5. Дерево *QTreeWidget* 278

Глава 15. Технология "Модель — представление" 287

15.1. Табличная модель и ее представления 287
15.2. Выравнивание элементов списка 299
15.3. Галочки для элементов таблицы 300
15.4. Дерево каталогов и файлов 303
15.5. Иерархическая модель 306
15.6. Дерево групп и таблица элементов 313
15.7. Разное 323

Глава 16. Процессы, потоки, синхронизация 333

16.1. Процессы 333
16.2. Потоки 335
16.3. Реентерабельность и потокобезопасность 336
16.4. Методы синхронизации процессов и потоков 338
16.4.1. Мьютексы 338
16.4.2. Читатели и писатели 341
16.4.3. Семафоры 341
16.4.4. Условие ожидания 342
16.5. Производитель — потребитель 343
16.6. Мультипликация 346
16.7. *QtConcurrent:* высокоуровневое API для параллельного программирования 350

Глава 17. Работа с базами данных 359

17.1. Компиляция SQL-драйверов 359
17.2. Подключение к базе данных и выполнение SQL-запросов 362
17.3. Работа с таблицами баз данных 374

Глава 18. Модели таблиц баз данных и их представления 381

18.1. Разработка модели и представления таблицы БД 381
18.2. Делегаты для ячеек таблицы 389

18.3. Связывание элементов управления форм с моделью данных 399
18.4. Консоль запросов 408
18.5. Связывание таблиц 420
18.6. Выполнение запросов в отдельных потоках 422

Глава 19. Средства XML 431

19.1. DOM API 432
19.2. SAX API 437
19.3. Класс *QStreamReader* 440
19.4. Модуль *QtXmlPatterns* 443

Глава 20. Локальные сети и Интернет 447

20.1. Эхо-сервер в блокирующем режиме 449
20.2. Клиент для эхо-сервера 453
20.3. Асинхронный однопоточный эхо-сервер 456
20.4. Многопоточный эхо-сервер 460
20.5. Работа с протоколом FTP 466
20.6. Отправка электронной почты по протоколу SMTP 467
20.7. Использование модуля WebKit: простой Web-браузер 475

Глава 21. Использование модуля QtScript 479

Глава 22. Технология COM. Интеграция с MS Office. Средства мультимедиа 495

22.1. Импорт и экспорт таблиц MS Excel 495
22.2. Работа со звуком 501
22.3. Анимация 502
22.4. Библиотека Phonon 503
22.5. Воспроизведение видео с помощью Media Player 507

Глава 23. Внешний вид элементов управления 509

23.1. Палитры 509
23.2. Стили 511

Глава 24. Интернационализация приложений 519

24.1. Подготовка исходного текста программы 519
24.2. Подготовка файлов описания проекта и ресурсов 522
24.3. Утилиты *lupdate* и *lrelease* 526
24.4. Программа Qt Linguist 526
24.5. Переключение языка во время выполнения программы 528

Глава 25. Использование интерпретатора Python 531

25.1. Зачем это нужно 531
25.2. Основные сведения о языке Python 531

25.3. Встраивание интерпретатора Python в приложения, написанные на C/C++ 534
25.4. Библиотека PythonQt 545
25.5. Библиотека PyQt4 547

Глава 26. Создание библиотек и плагинов для Qt Designer 549

26.1. Создание библиотеки 549
26.2. Статическая компоновка 551
26.3. Динамическая загрузка 553
26.4. Расширение библиотеки виджетов Qt Designer 555

Глава 27. Сравнение библиотеки Qt с другими средствами межплатформенной разработки 565

27.1. Qt и Java 565
27.2. Qt и .NET 566
27.3. Qt и wxWidgets 567

Глава 28. Разное 573

28.1. Борьба с "зависанием" интерфейса пользователя 573
28.2. Перемещение виджетов с помощью мыши 573
28.3. Заголовок окна 575
28.4. Область прокрутки 582
28.5. Сведения о текущей платформе 582
28.6. Бегущая строка 585
28.7. Работа с переменными окружения 587
28.8. Типы MIME 588
28.9. Буфер обмена 589
28.10. Перетаскивание файлов в окно текстового редактора 591

Заключение 595

Приложение. Описание содержимого компакт-диска 597

Список литературы 603

Предметный указатель 605

Ирине и Женьке

Введение

> Если вы возьмете любую работу, любое искусство, любую отрасль знания, любое умение и продвинете их так далеко вперед, как только возможно, — так далеко вперед, как они еще никогда не бывали, до самой границы границ неизведанного, то вы заставите их перейти в царство магии и волшебства.
>
> *Том Роббинс, писатель*

> Мы должны быть благодарны Господу за то, что Он создал мир таким, что все простое в нем истинно, а все сложное — ложно.
>
> *Григорий Сковорода, украинский философ (1722–1794)*

> Когда мне хочется прочесть роман, я сам пишу его.
>
> *Б. Дизраэли, лорд Биконсфилд, британский государственный деятель и писатель*

Библиотека Qt норвежской фирмы Trolltech представляет собой набор классов C++ и инструментов разработки программ для Microsoft Windows, Linux, FreeBSD, Solaris, HP-UX, Mac OS X и встраиваемых систем (*Embedded Linux*). На всех платформах библиотека Qt использует свой собственный набор визуальных элементов, в результате приложения, созданные на ее основе, во всех системах выглядят и работают одинаково.

Поскольку Qt — удобный и мощный инструмент, многие разработчики применяют ее не только для построения платформонезависимых приложений, но также и при создании программ, которые должны работать в конкретной операционной системе (Linux или Windows). Являясь образцом объектно-ориентированной библиотеки, Qt включает в себя классы для работы со структурами данных, файлами и каталогами, сетевыми протоколами, потоками и процессами, 2D- и 3D-графикой, OpenGL, базами данных, форматами HTML и XML, обширной коллекцией виджетов для построения графического интерфейса пользователя и многие другие возможности, облегчающие процесс создания профессиональных приложений на всех его этапах: от про-

граммирования заставки, показываемой при запуске, до разработки системы помощи.

Вместе с библиотекой поставляются средства для быстрой разработки программ: редактор форм Qt Designer, модуль интеграции с Microsoft Visual Studio 6.0, .NET 2003 и .NET 2005, программа для интернационализации приложений Qt Linguist, система помощи Qt Assistant с подробной документацией и примерами программ. Подобно программным продуктам Microsoft Office для Windows, функциональные возможности которых можно расширять с помощью встроенного языка Visual Basic, в приложения Qt тоже может быть встроен свой скриптовый язык Qt Script.

По сравнению с предыдущей версией библиотеки (Qt 3) структура классов Qt 4 существенно изменилась, поэтому для старых приложений Qt 3 необходима переработка исходного текста. Хотя процедура конвертирования в достаточной степени автоматизирована (имеется утилита `qt3to4`), но в серьезных проектах без "ручной" работы не обойтись.

Библиотека Qt — безусловный лидер среди имеющихся средств разработки межплатформенных программ на языке C++. Широко известная и часто используемая в мире Linux, она, благодаря распространению графической оболочки KDE, стала де-факто стандартом проектирования программного обеспечения на этой платформе. Для разработчиков Windows-приложений библиотека Qt долгое время не выходила на передний план, поскольку для Windows существовали более доступные и удобные средства быстрой разработки программ. Но в последнее время расстановка сил в корне изменилась. Новая, 4-я версия Qt, наконец, "дотянулась" по своим возможностям до тех вершин, на которых долгое время господствовали Microsoft и Borland/Inprise (последняя, к слову сказать, когда речь зашла о межплатформенном инструменте, не нашла ничего лучше, как обратиться к библиотеке Qt, но результат, к сожалению, оказался хуже первоисточника). К тому же версия Qt для Windows наконец-то стала свободной, а не только коммерческой, как это было раньше.

Исходные тексты библиотеки открыты, но лицензия GPL требует, чтобы программы, которые разрабатываются на основе Qt, распространялись с открытым исходным кодом. Поэтому если вы не желаете открывать исходный код своей программы, то должны приобрести коммерческую версию Qt.

Библиотека Qt используется многими программистами во всем мире. Среди корпоративных клиентов Trolltech такие известные компании, как Synopsys (автоматизированные системы проектирования электронных схем), Walt Disney Feature Animation (компьютерная анимация), Skype (общение голосом через Интернет), Volvo (применение Qtopia Core для разработки человеко-машинного интерфейса между водителем и бортовым компьютером, рабо-

тающим под управлением Linux), Adobe (программа Adobe Photoshop Album), Google (электронная карта земной поверхности Google Earth), Wolfram (система компьютерной математики Mathematica), Perforce (мощная система управления конфигурированием программного обеспечения — Software Configuration Management, SCM), Samsung и NXP Semiconductors (программное обеспечение смартфонов), Siemens (программное обеспечение для контроля производственной линии), NASA (моделирование условий полета космических кораблей) и еще более 5000 фирм — зарегистрированных пользователей Qt.

Данная книга представляет собой сборник готовых решений многих распространенных проблем, с которыми сталкивается разработчик, использующий библиотеку Qt. Автор надеется, что приведенные рецепты помогут вам освоить большинство возможностей этой замечательной библиотеки и пополнить обширный список программ на сайтах **http://sourceforge.net**, **www.qt-apps.org** и **http://trolltech.com/customers/coolapps**.

Книга состоит из 28 глав. В *главе 1* рассмотрены основные инструменты программиста, *глава 2* поможет читателю вспомнить основы языка C++. *Глава 3* начинает наше знакомство с библиотекой Qt. В *главе 4* обсуждаются основы построения интерфейса, *глава 5* посвящена методам обработки событий, в *главе 6* рассмотрены кнопки и диалоги, а в *главе 7* — меню, панели инструментов и строка состояния. В *главе 8* рассмотрены менеджеры размещения, в *главе 9* — визуальное проектирование интерфейса с помощью программы Qt Designer и разработка системы помощи. В *главе 10* обсуждаются методы работы с данными различного типа, в *главе 11* — каталоги и файлы. В *главе 12* описаны интерфейсы SDI и MDI, в *главе 13* обсуждаются графические инструменты и работа с принтером. *Глава 14* посвящена составным элементам интерфейса: спискам, таблицам и деревьям. В *главе 15* вводится понятие технологии "Модель — представление". В *главе 16* обсуждаются методы использования процессов и потоков. *Главы 17* и *18* посвящены разработке приложений баз данных. В *главе 19* описаны средства XML, а в *главе 20* — работа с сокетами и модулем WebKit. В *главе 21* мы знакомимся со скриптовым языком Qt, в *главе 22* применяем технологию COM для работы с приложениями Microsoft Office, а также обсуждаем мультимедийные возможности библиотеки Qt. В *главе 23* описана работа с палитрами и стилями, определяющими внешний вид визуальных элементов. *Глава 24* посвящена технологии локализации интерфейса приложения. В *главе 25* мы научимся встраивать в Qt-приложения интерпретатор языка Python, а в *главе 26* — создавать динамические библиотеки и плагины для Qt Designer. В *главе 27* обсуждаются и сравниваются известные технологии разработки межплатформенных приложений. В *главе 28* собраны частные вопросы, которые не попали в другие

разделы книги. В приложении приведено описание компакт-диска. Прилагаемый к книге компакт-диск содержит исходные тексты всех программ.

Особое внимание в книге уделено вопросам разработки приложений баз данных, программированию сетевых взаимодействий и многопоточных задач. Рассмотрены новые возможности библиотеки Qt, появившиеся в релизе 4.4 (май 2008 г.): мультимедийная библиотека Phonon, модули WebKit и QtConcurent.

Книга рассчитана на читателей, имеющих хотя бы минимальный опыт разработки программ на языке C++ для Windows или Linux, также будет полезна студентам-программистам и всем, кого интересуют вопросы межплатформенной разработки приложений.

Приводимые решения, если это не оговорено особо, тестировались в системах Microsoft Windows XP и Vista, Mandriva Linux 2008, Ubuntu 7.10 и Knoppix 5.

Коммерческие и свободные версии Qt 4

Коммерческие версии Qt 4: *Qt Console Edition*, *Qt Desktop Light Edition* и *Qt Desktop Edition* различаются составом инструментальных средств разработки, поддерживаемыми функциями и компонентами API, а также, разумеется, ценой. Самый полный набор функциональных возможностей имеет *Qt Desktop Edition*.

Свободная (т. е. некоммерческая, бесплатная) версия *Qt Open Source Edition*, предназначенная только для создания программ с открытым исходным кодом, доступна для платформ Microsoft Windows, UNIX/X11 и Max OS X и включает все средства разработки и компоненты API, имеющиеся в коммерческой *Qt Desktop Edition*, кроме:

- драйверов для доступа к коммерческим СУБД Oracle, DB2 и Sybase;
- модуля `ActiveQt` для работы с COM/ActiveX в Windows;
- поддержки коммерческих компиляторов (например, отсутствует интеграция с Microsoft Visual Studio).

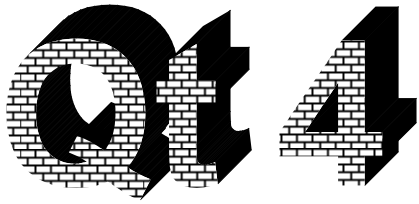
Некоммерческую редакцию библиотеки Qt для Linux и Windows можно скачать с сайта **http://trolltech.com**. Там же имеется подробная документация по последним релизам библиотеки Qt и другим продуктам фирмы Trolltech (к сожалению, пока только на английском, китайском и японском языках).

Заметим, что исходные тексты на языке C++ предоставляются как для коммерческих, так и для свободной версии библиотеки Qt. С помощью коммерческих версий Qt могут разрабатываться программы как с открытым, так и с закрытым исходным кодом.

Благодарности

Автор выражает признательность Игорю Владимировичу Шишигину и Максу Шлее за предложение написать эту книгу, а также Леониду Борисовичу Кочину — за помощь при подготовке книги к печати. Благодарю своих научных руководителей Эдуарда Ивановича Чаплыгина, Валерия Петровича Шевчука и Анатолия Павловича Желтоногова. Большое спасибо моей жене Ирине и дочке Женьке за терпение и понимание.

ГЛАВА 1



Инструменты программиста

Самое важное в политике — следовать своей цели: средства ничего не значат.

Наполеон I

Для редактирования исходного текста программ прежде всего необходим какой-нибудь текстовый редактор. Хорошо, если он выполняет подсветку синтаксиса, умеет форматировать листинги, дает подсказку по текущему объекту и выводит список членов нужного класса. Кроме того, не обойтись без компилятора и отладчика. Все эти инструменты могут быть включены в состав единой, так называемой *интегрированной среды разработки приложений* (IDE, Integrated Development Environment).

Для каждой операционной системы, в которой (или *для* которой) вы пишете приложение, применяются свои инструменты. Но разработчики библиотеки Qt побеспокоились о том, чтобы их продукт был совместим с большинством из них.

Если вы держите эту книгу в руках, то, вероятно, знаете, что для языка C++ существует множество компиляторов. Самые распространенные для системы Microsoft Windows — это компиляторы Microsoft Visual C/C++, Borland C/C++, Intel C/C++, Watcom C/C++, Digital Mars C/C++ и MinGW. В системах Linux/FreeBSD чаще всего используется G++, входящий в состав коллекции компиляторов GCC (GNU Compiler Collection), но также имеется и компилятор от Intel.

Более подробную информацию о совместимости конкретных релизов библиотеки Qt с различными версиями компиляторов можно найти в документации, поставляемой вместе с библиотекой, в разделах *Compiler Notes* (Замечания по отношению компиляторов) и *Platform-Specific Notes* (Замечания по используемым платформам), а также на сайте фирмы Trolltech по адресу **http://trolltech.com/developer/notes/**.

1.1. Microsoft Visual C++

Коммерческие версии Qt работают с компиляторами Microsoft Visual C++ 6.0, .NET 2003, .NET 2005 и свободно распространяемым Visual Studio 2005 Express Edition (в последнем случае необходимо отдельно установить Platform SDK). Для Microsoft Visual C++ 6.0 требуется Service Pack 5. Сообщается, что поддержка Microsoft Visual C++ 6.0 в скором времени, вероятно, будет прекращена.

В коммерческой версии Qt для Windows предусмотрена интеграция с Microsoft Visual Studio (рис. 1.1).

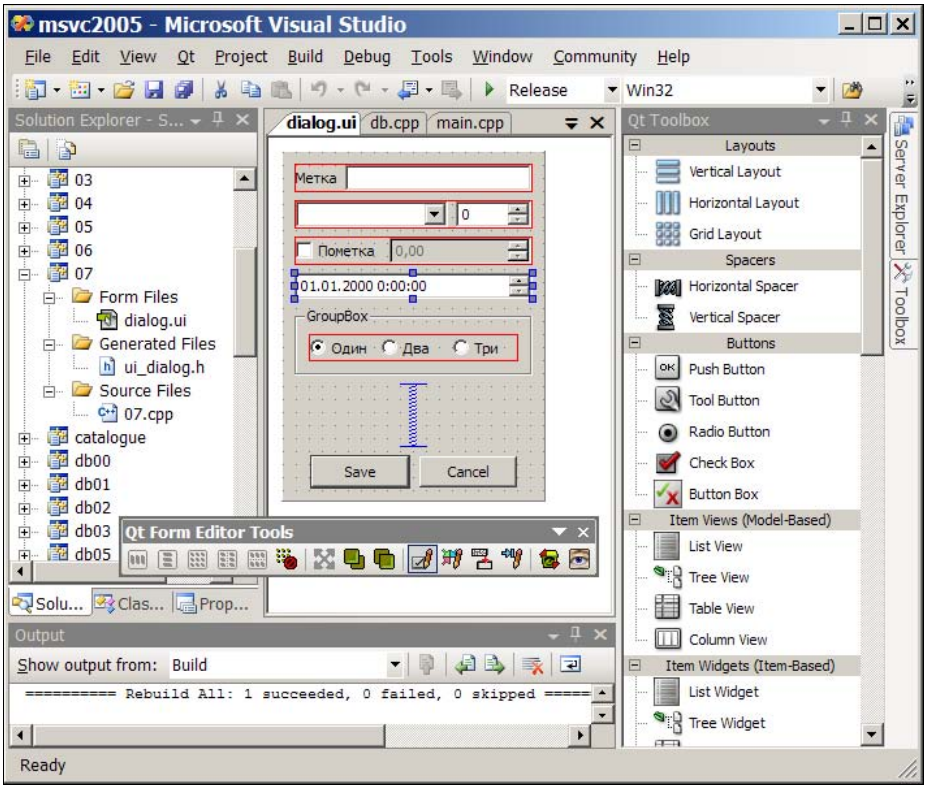


Рис. 1.1. Проект Qt в Microsoft Visual Studio 2005

Начиная с релиза Qt 4.3, компилятор Microsoft Visual C++ наконец-то может использоваться и со свободной версией *Qt Open Source Edition* (но интегратор, по-прежнему, доступен только для коммерческих версий Qt).

1.2. Intel C++

Коммерческие версии библиотеки Qt 4 поддерживают 32- и 64-разрядный Intel C++ 10.0 и 10.1 для Linux и 8.1 для Windows. Однако в Intel C++ 10.0 недопустимы предкомпилированные заголовочные файлы. Также известны проблемы при компиляции в режиме Release с помощью Intel C++ 10.1.

1.3. Borland C++

К сожалению, Qt 4 (в отличие от Qt 3) официально не поддерживает компилятор Borland C/C++, но соответствующие файлы конфигурации в составе библиотеки Qt 4 еще остались, их можно найти в каталоге mkspecs /win32-borland. Хотя утилита `configure` по-прежнему воспринимает ключ `-platform win32-borland`, но требуются дополнительные ухищрения, чтобы собрать библиотеку и свои программы с помощью компилятора Borland.

1.4. GNU C++ и MinGW

Для некоммерческой библиотеки *Qt Open Source Edition* в системе Windows до недавнего времени существовал только свободно распространяемый компилятор MinGW, а в Linux — GCC 3.2 и более поздние версии. Разумеется, GCC и MinGW подходят и для коммерческих версий Qt.

Для начинающих автор рекомендует использовать MinGW Developer Studio — свободно распространяемую интегрированную среду разработки программ на C++ для Windows. Она доступна в Интернете по адресу **www.parinyasoft.com** (вместе с компилятором MinGW32).

1.5. QDevelop

Конечно, с исходным текстом программ можно работать в любом текстовом редакторе, а компиляцию проводить с помощью командной строки. Но удобнее (во всяком случае, так считают те, кто мало знаком с миром UNIX/Linux-систем) разрабатывать программы в какой-нибудь интегрированной среде (IDE). В состав библиотеки Qt входит утилита Qt Designer, с помощью которой можно в диалоговом режиме проектировать графический интерфейс приложений. Но она не является интегрированной средой разработки, т. к. в ней нет текстового редактора и отладчика (как было объявлено, в 4-й версии программисты Trolltech отказались от идеи превратить Qt Designer в полноценную среду разработки, а сосредоточились на вопросах интеграции его со стандартными инструментами).

В Linux, где Qt стала де-факто стандартом разработки программ, с этим особых проблем не возникает, поскольку практически любая IDE (например, KDevelop) поддерживает работу с Qt.

С системой Windows все, как обычно, сложнее. В коммерческой версии Qt для Windows предусмотрена интеграция с Microsoft Visual Studio, но в *Open Source Edition* данная возможность отсутствует. Поэтому в рамках направления Open Source в настоящее время разрабатываются несколько похожих проектов, способных претендовать на звание "IDE для Qt". Среди них стоит отметить QDevelop **http://qdevelop.org** (рис. 1.2).

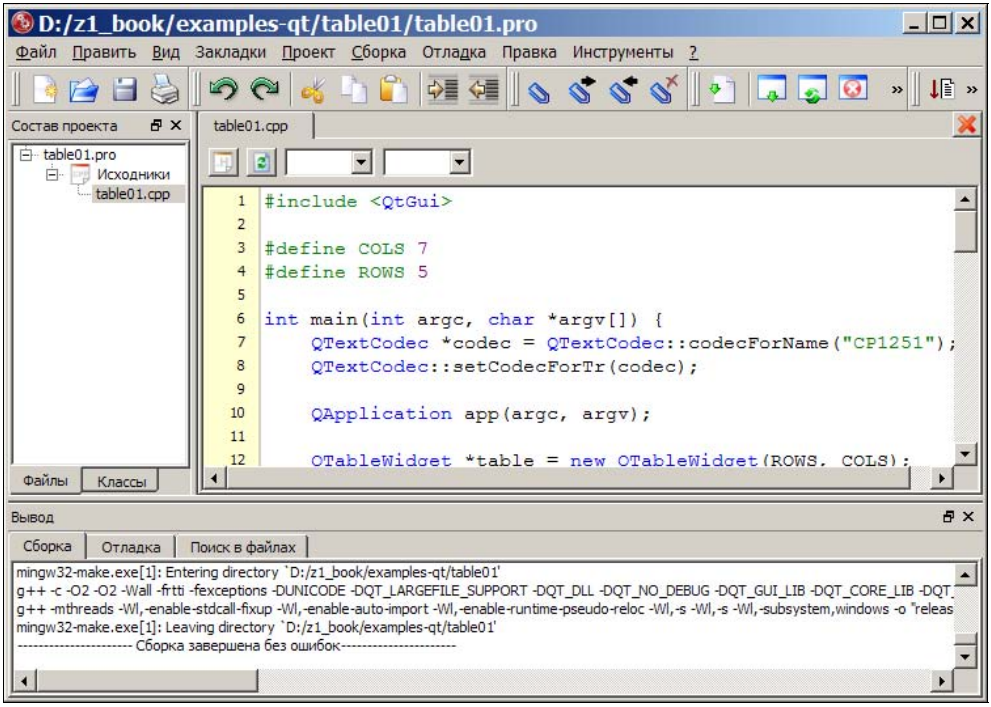


Рис. 1.2. Интегрированная среда разработки QDevelop

Сама программа QDevelop написана с использованием Qt, поэтому ее исходные тексты можно скомпилировать на любой платформе, которая поддерживает библиотеку Qt. Хотя к январю 2008 г. был выпущен только предварительный релиз 0.25, но программа уже вполне работоспособна: имеется русский перевод интерфейса, система контекстной помощи, работает отладчик, подсветка синтаксиса, а также автодополнение кода (для последнего требуется утилита `ctags`, ее можно скачать с сайта **http://ctags.sourceforge.net**).

После компиляции программы QDevelop (или после ее инсталляции, если вы не захотели возиться с исходными текстами) нужно в меню **Инструменты** выбрать пункт **Инструменты** и указать пути к инструментам библиотеки Qt, компилятору и отладчику (рис. 1.3).

В остальном интерфейс программы стандартен и достаточно удобен, поэтому работа с ней не составит трудностей для любого, кто знает какую-либо другую интегрированную среду разработки.

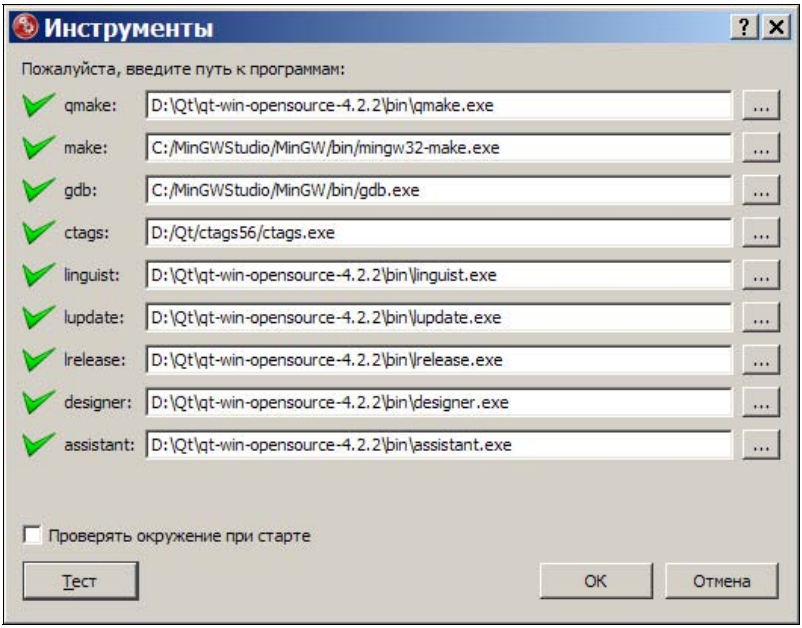


Рис. 1.3. Настройка параметров QDevelop

1.6. Рекомендации по выбору компилятора

> Совет — это теория жизни, а практика жизни обычно совсем другая.
>
> *Пауло Коэльо*

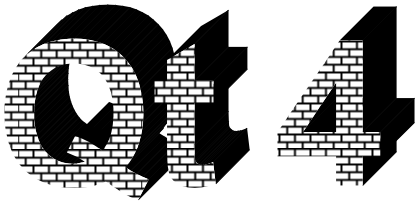
Если читатель собирается компилировать программы только для системы Windows, то, несомненно, лучшим выбором будет Microsoft Visual C++. При отсутствии коммерческого компилятора подойдет компилятор MinGW вместе с отладчиком GDB и интегрированной средой QDevelop.

Для разработки приложений в системе Linux однозначно следует выбрать GNU C++.

Если приложение, разрабатываемое в Windows, предполагается затем компилировать в Linux, то целесообразнее сразу обратиться к MinGW, чтобы затем безболезненно перейти на GNU C++. И наоборот, приложение, разработанное в Linux с помощью GNU C++, в систему Windows лучше всего переносить с помощью MinGW.

Если у вас уже имеется какой-нибудь любимый компилятор, поддерживаемый библиотекой Qt, то можете игнорировать приведенные рассуждения и смело использовать привычные инструменты.

ГЛАВА 2



"Обычный" C++

Всегда можно начать сначала. Не потому ли столько людей ходят по кругу.

Анджей Сток

Прежде чем приступить к обсуждению библиотеки Qt, рассмотрим некоторые концепции C++, которые пригодятся нам в дальнейшем.

2.1. Консольная программа на языке C++

ЗАДАЧА. Написать на языке C++ простую программу, которая запрашивает исходные данные (два целых числа) и выводит результат их сложения.

Решение. В листинге 2.1 приведен текст *консольной* (не имеющей графического интерфейса) программы, которая решает поставленную задачу.

Листинг 2.1. Простейшее консольное приложение

```
 1  // Сложение двух целых чисел (консольное приложение)
 2
 3  #include <iostream>
 4
 5  int main() {
 6
 7      using namespace std;
 8      int a, b;
 9
10      cout << endl << "a = ";
11      cin >> a;
12
13      cout << "b = ";
14      cin >> b;
```

```
15
16      int c = a + b;
17      cout << a << " + " << b << " = " << c << endl;
18
19     return 0;
20  }
```

Пояснения к программе*:

- (1) — строка, начинающаяся двойным слэшем, содержит комментарий;
- (3) — с помощью директивы `#include` подключается так называемый заголовочный файл с определениями внешних классов и функций. В данном случае это библиотека ввода-вывода `iostream` (в старых книгах вы можете встретить `iostream.h`);
- (5) — главная функция, с которой начинается выполнение программы, всегда называется `main`. Она должна возвращать результат типа `int` (целое число), которое обычно считается кодом ошибки. Если при выполнении программы ошибок не возникло, то возвращается число 0. Вообще говоря, могут присутствовать также входные параметры командной строки, но в нашем простом проекте они не нужны. Открывающая фигурная скобка начинает программный блок, относящийся к главной функции приложения;
- (7) — директива `using namespace` указывает, какое *пространство имен* используется по умолчанию. Если ее опустить, то во всей программе вместо идентификаторов `cin` (поток ввода данных), `cout` (поток вывода) и `endl` (конец строки) пришлось бы писать `std::cin`, `std::cout` и `std::endl`;
- (8) — объявили две переменные целого типа;
- (10) — в стандартный поток вывода `std::cout` (как правило, это экран монитора) вывели сначала код перехода на новую строку `endl`, а затем приглашение ввести значение переменной `a`;
- (11) — прочитали значение переменной `a` из стандартного потока ввода `std::cin` (обычно это клавиатура);
- (13, 14) — вывели приглашение на ввод второй переменной `b` и прочитали введенное значение;
- (16) — объявили новую переменную `c` типа `int`, в которую сразу записали результат сложения двух введенных чисел;

* Здесь и далее в круглых скобках указаны номера строк листинга. — *Ред.*

- (17) — вывели значение переменной `a`, символ сложения, значение переменной `b`, символ равенства, результат сложения и выполнили переход на новую строку;
- (19) — вышли из функции `main()`, указав нулевой результат — признак того, что программа завершилась без ошибок;
- (20) — закрывающая фигурная скобка оканчивает программный блок, относящийся к функции `main()`.

Это наша первая программа, поэтому здесь мы не отвлекаемся на анализ корректности вводимых пользователем данных (попробуйте ввести буквенный символ или вещественное число с дробной частью), а также возможное переполнение при сложении очень больших чисел.

2.2. Компиляция и выполнение программы с помощью командной строки Microsoft Visual C++

ЗАДАЧА. С помощью Microsoft Visual C++ выполнить компиляцию и сборку консольной программы, текст которой приведен в листинге 2.1, а также проверить правильность ее выполнения.

Решение.

1. Наберем в любом текстовом редакторе исходный текст программы (см. листинг 2.1). Сохраним его в отдельном каталоге, в файле под именем console1.cpp.
2. Откроем командную строку Visual Studio 2005 Command Prompt, для чего выполним команды меню **Пуск** | **Программы** | **Microsoft Visual Studio 2005** | **Visual Studio Tools** | **Visual Studio 2005 Command Prompt**. В результате на экране появится окно MS-DOS, но предварительно будут настроены переменные окружения, необходимые для работы компилятора и компоновщика.
3. Для удобства запустим в открывшемся консольном окне какой-нибудь текстовый файловый менеджер, например, Far:

   ```
   "C:\Program Files\Far\far.exe"
   ```

   Кавычки необходимы, поскольку полное имя исполняемого файла содержит пробел. Если на вашем компьютере не установлен Far, то данный пункт можно пропустить.
4. Перейдем в каталог, где находится файл с исходным текстом нашей программы. Если вы не используете файловый менеджер, то для этого придется выполнить команду смены текущего каталога `cd`.

5. Выполним компиляцию и сборку программы, для чего введем команду

```
cl -nologo -EHsc -GR -Zc:forScope -Zc:wchar_t -Feconsole1
↳console1.cpp
```

Символ ↳ в самой команде не вводится, тут он обозначает перенос строки. Здесь `cl` — это имя исполняемого файла компилятора. Последним параметром указано имя файла с исходным текстом программы. Ключ `-Fe` задает имя файла-результата сборки программы. Ключ `-nologo` позволяет избежать вывода на экран лишних сообщений (при первом запуске не указывайте его, чтобы узнать название и номер версии установленного у вас компилятора). Остальные ключи пока не обязательны, т. к. соответствующие возможности пока отсутствуют в нашей простой программе, но в дальнейшем без них не обойтись: ключ `-EHsc` разрешает использование исключений; ключ `-GR` разрешает компилятору учитывать информацию о типах во время выполнения программы (RTTI — RunTime Type Information); ключ `-Zc:forScope` устанавливает новое правило видимости параметров оператора цикла `for`; ключ `-Zc:wchar_t` разрешает применение типа `wchar_t` (символов Unicode). Чтобы рассмотреть сообщения, которые будут выведены на экран в результате выполнения указанной команды, при работе в Far нужно нажать комбинацию клавиш <Ctrl>+<O>.

6. Если компиляция и компоновка программы прошла без ошибок, то в текущем каталоге будет создан файл console1.exe. Запустим его на выполнение. Введем два целых числа и получим результат их сложения (рис. 2.1).

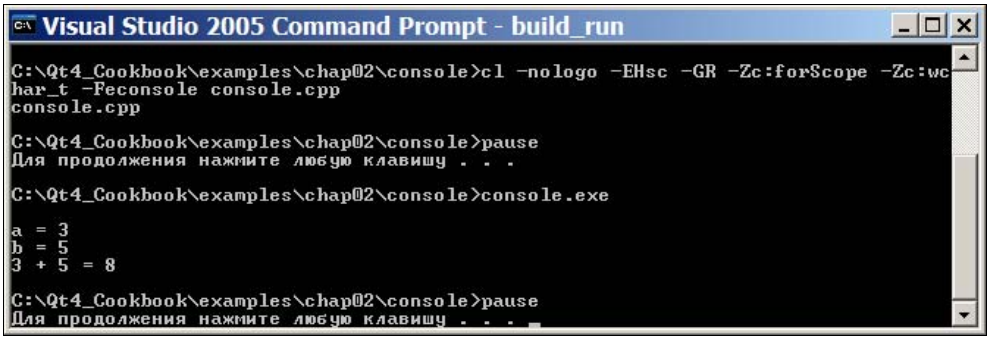


Рис. 2.1. Сборка и выполнение консольной программы

Еще несколько замечаний.

Поскольку при разработке реальных приложений приходится неоднократно вводить в текст программы изменения и снова выполнять компиляцию, то данный процесс лучше автоматизировать с помощью командного файла. Для этого в любом текстовом редакторе нужно набрать текст, приведенный в листинге 2.2, и сохранить его в файле с расширением bat или cmd.

Листинг 2.2. Пример командного файла

```
   cl -nologo -EHsc -GR -Zc:forScope -Zc:wchar_t -Feconsole1
↳console1.cpp
   pause
   console1.exe
   pause
```

Разумеется, запускать данный файл на выполнение необходимо из командной строки Visual Studio.

Следует иметь в виду, что разные компиляторы требуют указания различных ключей. Если вместо Microsoft Visual C++ у вас установлен, например, Borland C++, то команда для сборки программы должна выглядеть следующим образом:

```
bcc32 -q -econsole1 console1.cpp
```

Здесь ключ `q` подавляет приветствие компилятора, а ключ `e` позволяет задать имя выходного файла.

Реальные проекты обычно содержат несколько файлов с исходными текстами, к тому же часто используют предварительно скомпилированные библиотеки. Поэтому этапы *компиляции* (генерации объектных файлов) и *компоновки* (получения исполняемого файла программы) приходится выполнять по отдельности. Вызов компилятора (без компоновщика) в Microsoft Visual C++, Borland C++ и GNU C++ осуществляется с указанием одинакового ключа `c`. Например, для Microsoft Visual C++ компиляция с последующей компоновкой выполняется так:

```
cl -c -nologo -EHsc -GR -Zc:forScope -Zc:wchar_t -Foconsole1
↳ console1.cpp
link -nologo -out:console1.exe console1.obj
```

2.3. Компиляция и выполнение программы с помощью командной строки в системах Linux/FreeBSD/Solaris/HP-UX/Mac OS X

ЗАДАЧА. С помощью GNU C++ выполнить компиляцию и сборку консольной программы, текст которой приведен в листинге 2.1, и проверить правильность ее выполнения.

Решение.

1. Наберем в любом текстовом редакторе исходный текст программы (см. листинг 2.1). Сохраним его в отдельном каталоге, в файле под именем console1.cpp.

2. Для удобства запустим какой-нибудь текстовый файловый менеджер, например, Midnight Commander (команда `mc`).
3. Перейдем в тот каталог, где находится файл с исходным текстом нашей программы. Если не используется файловый менеджер, то для этого придется выполнить команду смены текущего каталога `cd`.
4. Выполним компиляцию и сборку программы, для чего введем команду вызова компилятора и компоновщика GNU C++ (ключ `-o` позволяет задать имя выходного файла):

```
g++ -o console1 console1.cpp
```

Замечание

Для вызова только компилятора (без компоновщика) GNU C++ следует указать ключ `-c`. Для последующей компоновки объектных файлов придется выполнить команду g++ еще раз:

```
g++ -c -o console1.o console1.cpp
g++ -o console1 console1.o
```

2.4. Создание консольного приложения в Microsoft Visual Studio 2005

ЗАДАЧА. Создать группу проектов (*solution* — "решение") и проект консольного приложения Win32, собрать и выполнить программу из листинга 2.1.

Решение.

1. Откроем интегрированную среду разработки программ Microsoft Visual Studio 2005. Закроем окно Qt Visual Studio Integration, если оно появилось на экране.
2. Создадим группу проектов (*solution*), для этого выберем из главного меню пункты **File** (Файл) | **New** (Новый) | **Project** (Проект). Слева, в разделе **Project Types** (Типы проектов) раскроем группу **Other Project Types** (Другие типы проектов) и щелчком мыши укажем пункт **Visual Studio Solutions**, затем справа, в окне **Templates** (шаблоны) щелкнем левой кнопкой мыши по пункту **Blank Solution** (Пустое решение). Внизу в поле **Name** (Имя) введем название, например, `Qt4_Cookbook_Examples`, а в графе **Location** (Расположение) укажем полный путь к каталогу, в котором будут располагаться наши проекты (рис. 2.2). Нажмем кнопку **OK**.
3. Теперь создадим новый консольный проект: **File** (Файл) | **New** (Новый) | **Project** (Проект). В разделе **Project types** (Типы проектов) раскроем груп-

пу **Visual C++** и щелчком мыши укажем пункт **Win32**. В окне **Templates** (Шаблоны) выберем пункт **Win32 Console Application** (Консольное приложение Win32). В поле **Name** введем название проекта (и одновременно имя каталога, в котором будут располагаться все файлы проекта), например, `console1`. В поле **Location** (Расположение) укажем полный путь к каталогу, в котором будет создан новый каталог с указанным именем. В графе **Solution** (Группа проектов) выберем из раскрывающегося списка режим **Add to Solution** (Добавить к имеющейся группе проектов, рис. 2.3). Нажмем кнопку **OK**.

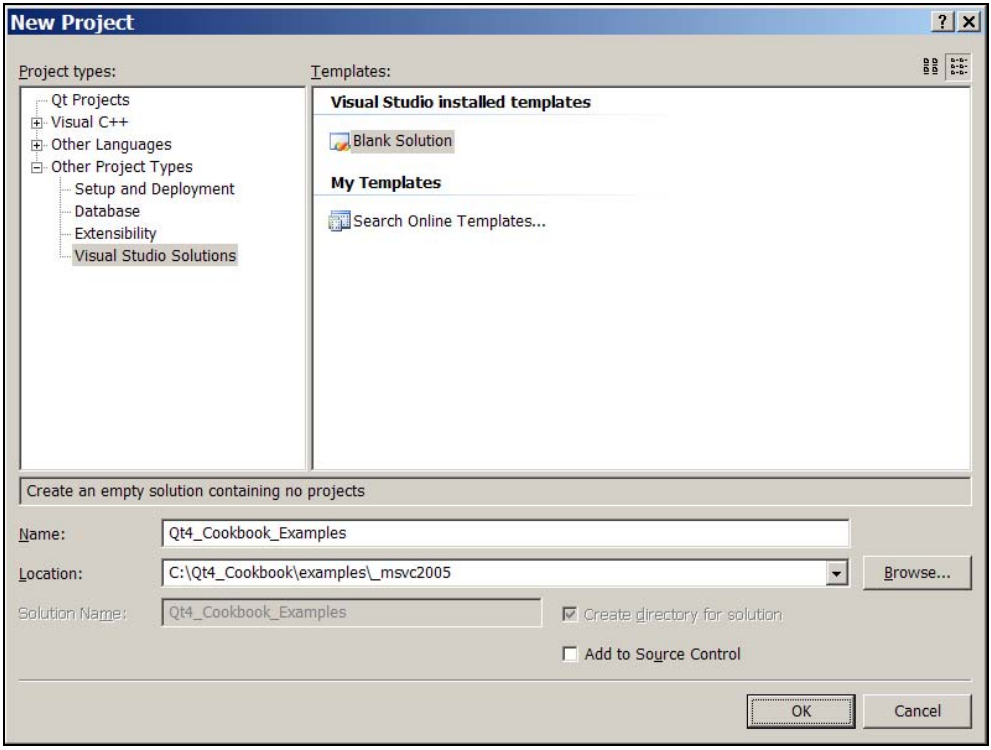


Рис. 2.2. Создание группы проектов в Microsoft Visual Studio 2005

4. На экране появится диалог **Win32 Application Wizard** (Мастер приложения Win32). Перейдем к разделу **Application Settings** (Настройки приложения), выбрав пункт с соответствующим названием слева или просто нажав кнопку **Next** (Далее).
5. Убедимся, что в разделе **Application type** (Тип приложения) выбрана опция **Console application** (Консольное приложение). В разделе **Additional options** (Дополнительные параметры) поставим галочку напротив пункта

Empty project (Пустой проект), т. к. мы не хотим, чтобы мастер добавлял к проекту образец кода (рис. 2.4). Нажмем кнопку **Finish** (Конец).

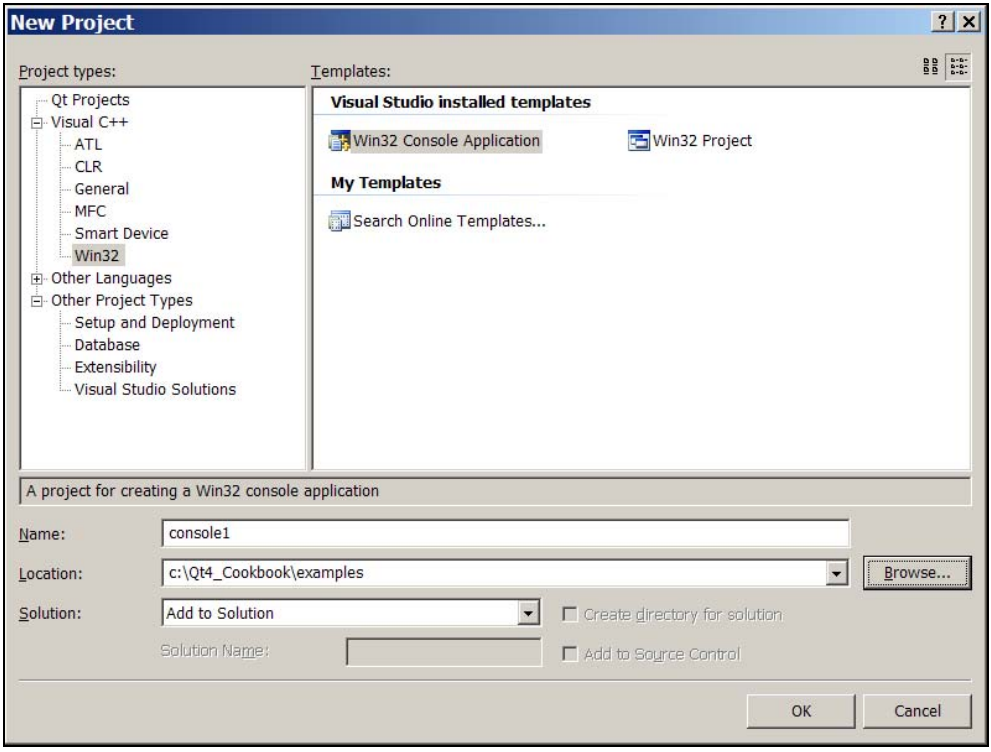


Рис. 2.3. Создание проекта консольного приложения в Microsoft Visual Studio 2005

6. Создадим новый файл для хранения исходного текста нашей программы и сразу добавим его к проекту: **Project** (Проект) | **Add New Item** (Добавить новый элемент). В разделе **Categories** (виды файлов) слева выберем группу **Visual C++**, а справа, в разделе **Templates** (Шаблоны) — пункт **C++ File (.cpp)**. В поле **Name** (Имя) введем название нового файла (т. к. в проекте предполагается всего один файл с исходным текстом, то его лучше назвать так же, как и сам проект), а в поле **Location** (Расположение) укажем каталог, в котором будет сохранен этот файл. Нажмем кнопку **Add** (Добавить).
7. Теперь можно ввести текст программы (см. листинг 2.1).
8. Нажмем кнопку **Save** (Сохранить) на панели инструментов, или воспользуемся главным меню: **File** (Файл) | **Save** (Сохранить), или комбинацией клавиш <Ctrl>+<S>.

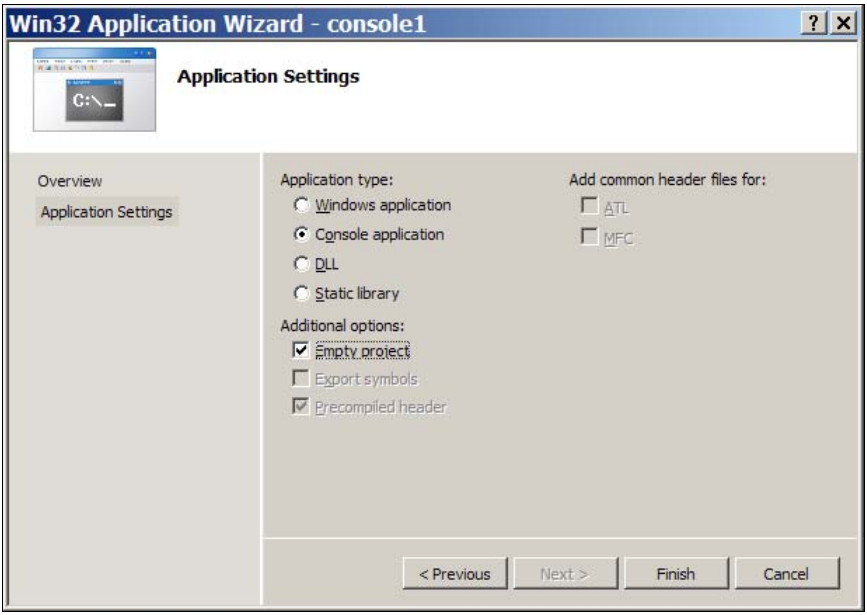


Рис. 2.4. Установка параметров нового проекта

9. Зададим режим компиляции: **Build** (Собрать) | **Configuration Manager** (Менеджер конфигураций). В раскрывающемся списке **Active Solution Configuration** (Текущая конфигурация) выберем пункт **Release**. Это значит, что результат компиляции не будет содержать информацию для отладчика.

10. Выполним компиляцию и сборку программы: **Build** (Собрать) | **Build Solution** (Собрать все проекты) или просто нажмем клавишу <F7>. Если не будет выведено каких-либо сообщений об ошибках, то мы увидим строку:

    ```
    ===== Build: 1 succeeded, 0 failed, 0 up-to-date, 0 skipped =====
    ```

    Это означает, что один проект собран без ошибок (*succeeded* — выполненный успешно).

11. Теперь можно запустить нашу программу на выполнение: **Debug** (Отладка) | **Start Without Debugging** (Выполнить без отладки). На экране появится консольное окно, в котором нас попросят ввести два числа, а затем будет выведен результат их сложения (рис. 2.5). После окончания выполнения программы на экран автоматически выводится приглашение нажать любую клавишу (оно появляется только в том случае, если программа была запущена из интегрированной среды) — иначе бы окно сразу закрылось и мы не успели рассмотреть результат.

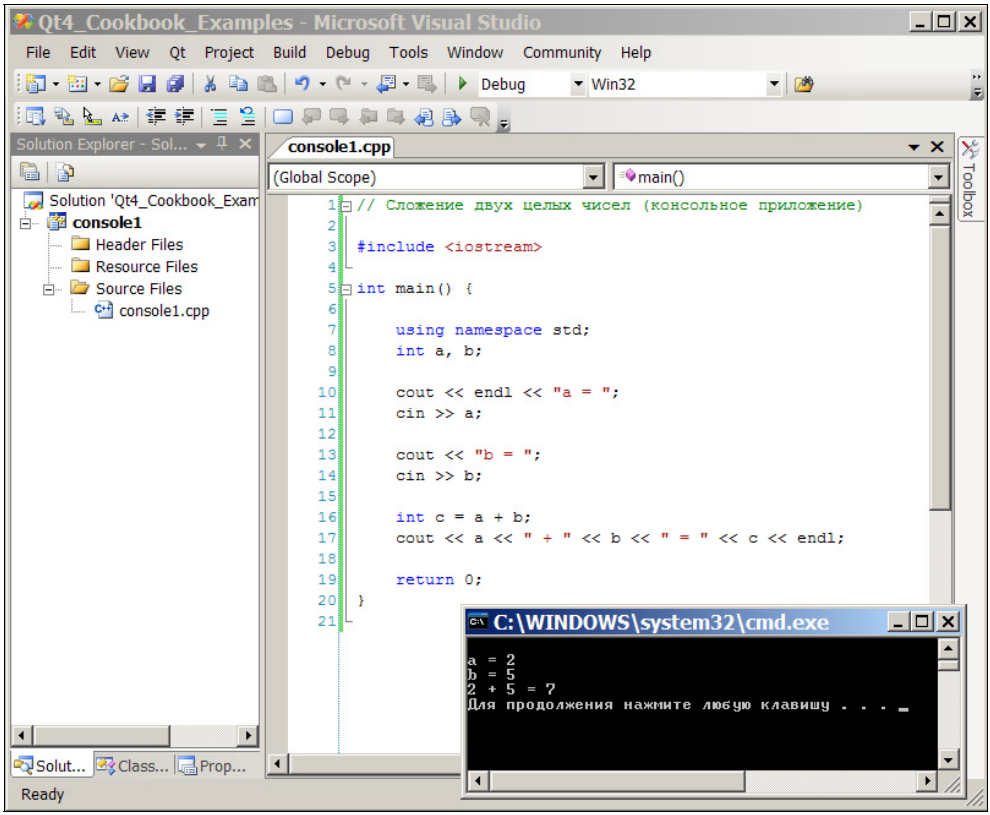


Рис. 2.5. Окно консольного приложения на фоне интегрированной среды Microsoft Visual Studio 2005

2.5. Шаблоны в C++

Напомним основные сведения о шаблонах C++, т. к. они пригодятся нам в дальнейшем. Допустим, имеется функция, которая меняет местами значения своих двух параметров (листинг 2.3).

Листинг 2.3. Пример функции

```
void swap (int& a, int& b) {
    int tmp = a;
    a = b;
    b = tmp;
}
```

Как известно, в языке C++ возможна *перегрузка функций* — определение нескольких функций с одним и тем же именем, но разным набором аргументов. Чтобы использовать функцию `swap` для аргументов другого типа (`long`, `float`, `double`, `char` и т. д.), придется явным образом описывать все возможные варианты перегрузки (листинг 2.4).

Листинг 2.4. Пример перегрузки

```
void swap (double& a, double& b) {
    double tmp = a;
    a = b;
    b = tmp;
}
```

Концепция шаблонов предлагает более элегантное решение (листинг 2.5).

Листинг 2.5. Использование шаблона

```
template <class T>
void swap (T& a, T& b) {
    T tmp = a;
    a = b;
    b = tmp;
}
```

Здесь определен шаблон функции `swap`, оба параметра которой могут иметь произвольный тип `T` (заметим, что вместо традиционного символа `T` можно было указать другой идентификатор, например, `AnyType`). В результате функцию `swap` можно вызывать для перестановки значений переменных любого типа (листинг 2.6).

Листинг 2.6

```
 int a = 2, b = 5;       // Целые числа.
double x = 3.14, y = 2.7;  // Вещественные.
MyClass p1 = MyClass(...); // Любой класс.
MyClass p2 = MyClass(...);

swap (a, b);
swap (x, y);
swap (p1, p2);
```

ЗАДАЧА. Написать на языке C++ простую программу, в которой используются шаблоны (*template*).

Решение. В листинге 2.7 приведен текст простой программы с шаблонами. Здесь определен класс `Coord` (строки 6–18), хранящий координаты точки на плоскости, и шаблоны двух функций. Первая `swap` (строки 21–26) переставляет местами значения двух своих аргументов произвольного типа, а вторая `max` (строки 28–32) возвращает наибольшее значение переданных ей двух аргументов произвольного типа.

Затем работа обоих шаблонов проверяется на значениях различного типа: целого `int`, вещественного `double` и, наконец, `Coord`. Внешний вид консольного окна программы показан на рис. 2.6.

Листинг 2.7. Шаблоны в C++

```
 1  // Шаблоны
 2
 3  #include <iostream>
 4
 5  // Координата точки на плоскости:
 6  class Coord {
 7  public:
 8      int m_x, m_y;
 9
10      // Конструктор:
11      Coord(int x, int y)
12          : m_x(x), m_y(y) {};
13
14      // Сравнение:
15      inline bool operator<(const Coord& c) const {
16          return (m_x*m_x+m_y*m_y)<(c.m_x*c.m_x+c.m_y*c.m_y);
17      }
18  };
19
20  // Поменять местами два значения любого типа:
21  template <class T>
22  void swap (T& a, T& b) {
23      T tmp = a;
24      a = b;
25      b = tmp;
26  }
27
```

```
28  // Наибольшее из двух значений любого типа:
29  template <class T>
30  T max (T a, T b) {
31      return (a < b) ? b : a;
32  }
33
34  int main() {
35      int a = 2, b = 5;
36      double x = 3.14, y = 2.7;
37
38      printf("\n a = %d; b = %d", a, b);
39      swap (a, b);
40      printf("\n a = %d; b = %d", a, b);
41
42      printf("\n max {%d, %d} = %d \n", a, b, max(a, b));
43
44      printf("\n x = %lg; y = %lg", x, y);
45      swap (x, y);
46      printf("\n x = %lg; y = %lg", x, y);
47
48      printf("\n max {%lg, %lg} = %lg \n", x, y, max(x, y));
49
50      Coord p1 = Coord(1, 2);
51      Coord p2 = Coord(3, 4);
52      printf("\n p1 = (%d, %d); p2 = (%d, %d)",
53             p1.m_x, p1.m_y, p2.m_x, p2.m_y);
54      swap (p1, p2);
55      printf("\n p1 = (%d, %d); p2 = (%d, %d)",
56           p1.m_x, p1.m_y, p2.m_x, p2.m_y);
57
58      Coord p3 = max(p1, p2);
59      printf("\n max { (%d, %d), (%d, %d) } = (%d, %d)\n",
60             p1.m_x, p1.m_y, p2.m_x, p2.m_y,
61             p3.m_x, p3.m_y);
62
63      return 0;
64  }
```

Сборку приложения проводят, как обычно (см. *разд. 2.1–2.4*). При работе в интегрированной среде Microsoft Visual Studio 2005 для добавления нового проекта к имеющемуся решению нужно использовать команду меню **File** (Файл) | **New** (Новый) | **Project** (Проект) и не забыть выбрать режим **Add to**

Solution (Добавить к имеющейся группе проектов) вместо установленного по умолчанию **Create new Solution** (Создать новую группу проектов). При компиляции команду **Build** (Собрать) | **Build Solution** (Собрать все проекты) можно заменить командой **Build** (Собрать) | **Project Only** (Только проект) | **Build only...** (Собрать только текущий проект). Основной проект, выполняемый при старте сеанса отладки, выбирают с помощью команды меню **Project** (Проект) | **Set as StartUp Project** (Установить запускаемый проект).

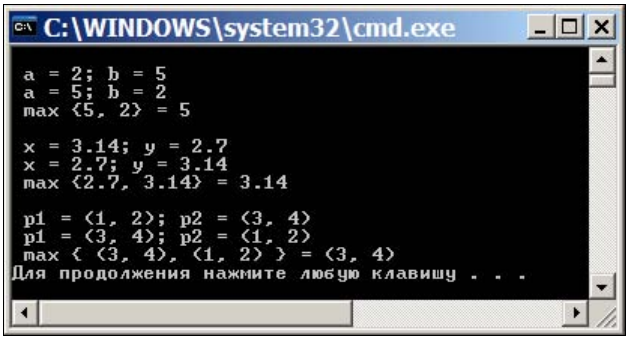
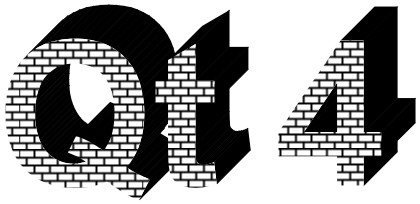


Рис. 2.6. Окно программы "Шаблоны в C++"

ГЛАВА 3



Основы Qt 4

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

В этой главе мы научимся писать и компилировать самые простые приложения с использованием библиотеки Qt.

3.1. Обзор классов библиотеки Qt 4

Кратко перечислим основные классы библиотеки Qt 4.

- Модуль `QtCore` — ядро библиотеки Qt (невизуальные элементы):
  - `QObject` — родитель для большинства других классов библиотеки Qt;
  - `QBitArray` — битовый массив;
  - `QByteArray` — массив байтов;
  - `QBuffer` — класс, позволяющий работать с `QByteArray` как с потоком ввода-вывода;
  - `QChar` — 16-разрядный символ Unicode;
  - `QCoreApplication` — консольное приложение;
  - `QDataStream` — обеспечивает платформонезависимую сериализацию объектов;
  - `QDate` — работа с календарными датами;
  - `QDateTime` — дата и время;
  - `QDir` — работа с каталогами файловой системы;
  - `QEvent` — базовый класс для событий;
  - `QFile` — работа с файлами;

- QFileSystemWatcher — позволяет отслеживать изменения в каталогах и файлах;
- QFlags — набор битовых флагов;
- QIODevice — базовый класс для логических устройств ввода-вывода;
- QLatin1Char — 8-разрядный символ ASCII/Latin1;
- QLatin1String — строка 8-разрядных символов в кодировке Latin1;
- QLibrary — динамически загружаемая библиотека;
- QLine — двумерный вектор, заданный целочисленными координатами своих концов;
- QLineF — двумерный вектор с вещественными координатами;
- QLinkedList — класс-шаблон для работы со связанными списками;
- QList — класс-шаблон для работы со списками элементов;
- QMap и QHash — словарь (массив элементов с доступом по ключу);
- QMutex, QMutexLocker, QSemaphore и QWaitCondition — мьютекс, семафор и условие возобновления потока;
- QReadLocker, QWriteLocker, QReadWriteLock — блокировки чтения/записи;
- QPair — класс-шаблон для работы с парами элементов;
- QPluginLoader — загрузка плагинов во время выполнения приложения;
- QPoint — точка на плоскости, заданная своими целочисленными координатами;
- QPointF — точка на плоскости с вещественными координатами;
- QPointer — класс-шаблон для работы с "безопасными" указателями, автоматически обнуляемыми при уничтожении объекта, на который они указывают;
- QProcess — класс для запуска сторонних процессов и взаимодействия с ними;
- QPointer — класс-шаблон для работы с очередями;
- QRegExp — регулярные выражения;
- QSet — множество элементов;
- QSettings — сохранение и загрузка настроек приложения (в том числе работа с реестром Windows и ini-файлами);

- `QSize` — два целых числа — размер прямоугольника;
- `QSizeF` — два вещественных числа — размер прямоугольника;
- `QSocketNotifier` — слежение за активностью файлового дескриптора;
- `QStack` — стек;
- `QString` — строка Unicode;
- `QStringList` — список строк;
- `QTemporaryFile` — работа с временными файлами;
- `QTextCodec` — текстовый кодек для перекодировки строк;
- `QTextStream` — строковый поток ввода-вывода;
- `QThread`, `QThreadStorage` — многопоточное программирование;
- `QTime` — работа с временем;
- `QTimer` — таймер;
- `QUrl` — унифицированный указатель информационного ресурса (*uniform resource locator*);
- `QUuid` — универсальный уникальный идентификатор (*universally unique identifier*);
- `QVarLengthArray` — массив переменной длины;
- `QVariant` — вариантный тип, позволяющий работать практически со всеми объектами библиотеки Qt;
- `QVector` — класс-шаблон для работы с динамическими массивами.

❒ Модуль `QtGui` — элементы интерфейса пользователя:

- `QWidget` — базовый класс для всех визуальных элементов;
- `QAbstractButton` — базовый класс для представления кнопок;
- `QAbstractScrollArea` — область с полосами прокрутки;
- `QAbstractSlider` — ползунок для ввода значения с помощью мыши;
- `QAbstractSpinBox` — поле с кнопками инкремента/декремента для ввода числового значения;
- `QDialog` — диалог;
- `QFrame` — фрейм;
- `QLabel` — текстовая метка или пиктограмма;
- `QLineEdit` — однострочное поле для ввода строки текста;

- `QCheckBox` — независимый элемент с двумя (или тремя) состояниями и текстовой меткой;
- `QComboBox` — поле ввода с кнопкой выбора и раскрывающимся списком;
- `QDateEdit` — поле для ввода даты;
- `QTimeEdit` — поле для ввода времени;
- `QDateTimeEdit` — поле для ввода даты и времени;
- `QDial` — элемент ввода, по внешнему виду напоминающий поворотный регулятор громкости или спидометр;
- `QSpinBox` — поле для ввода целого числа с кнопками инкремента/декремента;
- `QDoubleSpinBox` — поле ввода вещественных значений с кнопками инкремента/декремента;
- `QFocusFrame` — позволяет получить доступ к области, выходящей за пределы прямоугольника, занимаемого данным элементом;
- `QFontComboBox` — поле ввода с кнопкой и раскрывающимся списком для выбора шрифта;
- `QLCDNumber` — элемент для отображения числового значения, по внешнему виду напоминающий жидкокристаллический индикатор;
- `QMenu` — вертикальное меню;
- `QPushButton` — кнопка;
- `QRadioButton` — зависимый переключатель (радиокнопка);
- `QScrollArea` — элемент, который позволяет отображать свой дочерний виджет внутри области с полосами прокрутки;
- `QScrollBar` — вертикальная или горизонтальная полоса прокрутки;
- `QSizeGrip` — маркер в углу окна, который можно "зацепить" указателем мыши, чтобы изменить размеры окна;
- `QSlider` — вертикальный или горизонтальный ползунок;
- `QToolBar` — панель инструментов;
- `QToolButton` — кнопка на панели инструментов;
- `QProgressBar` — индикатор состояния длительного процесса;
- `QTabBar` — многостраничный диалог с закладками;
- `QTabWidget` — набор элементов, отображаемых на страницах диалога с закладками;

- `QToolBox` — одна колонка элементов, отображаемых на странице диалога с закладками;
- `QDesktopWidget`, `QDesktopServices`, `QSystemTrayIcon` — обеспечивают доступ к рабочему столу и системному лотку на панели задач;
- `QClipboard` — работа с буфером обмена;
- `QFontDatabase` — сведения о всех доступных шрифтах;
- `QSessionManager` — менеджер сеансов.

❒ Модуль `QtNetwork` — классы для работы с сетью:

- `QFtp` — клиентская часть протокола FTP;
- `QHttp` — работа с протоколом HTTP;
- `QHostAddress` — работа с IP-адресами;
- `QUdpSocket` — UDP-сокет;
- `QTcpSocket` — TCP-сокет;
- `QTcpServer` — сервер TCP;
- `QUrl` — работа с URL;
- `QSslSocket` — серверный и клиентский SSL-сокет.

❒ Модуль `QtOpenGL` — поддержка OpenGL.

❒ Модуль `QtSql` — работа с базами данных:

- `QSql` — пространство имен, в котором определены различные объекты, используемые при работе с модулем `QtSql`;
- `QSqlDatabase` — соединение с базой данных;
- `QSqlDriver` — абстрактный базовый класс для доступа к базам данных;
- `QSqlDriverCreator` — шаблон класса для доступа к базе данных, с помощью которого создаются классы для работы с конкретной СУБД;
- `QSqlError` — информация об ошибке при работе с базами данных;
- `QSqlField` — поле таблицы БД;
- `QSqlIndex` — индекс БД;
- `QSqlQuery` — запрос на языке SQL;
- `QSqlQueryModel` — модель данных, полученных в результате выполнения запроса;
- `QSqlRecord` — одна запись в таблице БД;

- `QSqlResult` — абстрактный интерфейс для доступа к данным;
- `QSqlTableModel` — модель данных, доступных для редактирования, соответствующая одной таблице БД;
- `QSqlRelationalTableModel` — модель данных, доступных для редактирования, соответствующая одной таблице БД (с поддержкой внешних ключей).

- ❒ Модуль `QtScript` — интерпретатор скриптового языка.
- ❒ Модуль `QtSvg` — поддержка векторной графики SVG.
- ❒ Модуль `QtXml` — работа с XML (DOM и SAX).
- ❒ Модуль `QtXmlPatterns` — для работы с XQuery.
- ❒ Модуль `QtDesigner` — классы для расширения Qt Designer.
- ❒ Модуль `QtUiTools` — работа с формами, разработанными в Qt Designer.
- ❒ Модуль `QtAssistant`, `QtHelp` — система помощи.
- ❒ Модуль `Qt3Support` — поддержка старых программ, разработанных для Qt 3.
- ❒ Модуль `QtTest` — тестирование приложений.
- ❒ Модуль `QtWebKit` — для работы с Web-страницами.
- ❒ Модуль `QtOpenGL` — для работы с трехмерной графикой.
- ❒ Модуль `Phonon` — средства мультимедиа.
- ❒ Модуль `QtTest` — средства тестирования.

Следующие два модуля доступны только в коммерческой версии библиотеки Qt 4 для Windows.

- ❒ Модуль `QAxContainer` — работа с элементами ActiveX.
- ❒ Модуль `QAxServer` — классы для разработки серверных компонентов ActiveX.

Следующий модуль имеется только в библиотеках Qt 4 для UNIX/Linux/FreeBSD.

- ❒ Модуль `QtDBus` — взаимодействие между процессами с использованием протокола D-Bus.

3.2. Компиляция библиотеки Qt

Поскольку библиотека Qt поставляется в исходных кодах, прежде чем компилировать свои программы, необходимо собрать саму библиотеку.

3.2.1. Компиляция в системе Microsoft Windows

Если установлен компилятор MinGW, то путь к его исполняемым файлам, находящимся в папке bin, должен быть указан в системной переменной `PATH` (напоминаем, что после редактирования значений переменных окружения требуется перезагрузить компьютер, чтобы все изменения вступили в силу). При использовании Visual Studio 6.0/2003/2005 все команды следует вводить в командной строке **Visual Studio Command Prompt**.

Порядок установки следующий:

- Если ставится коммерческая версия Qt, то имеющийся файл лицензии скопируйте в каталог Documents and Settings\ИмяПользователя и дайте ему имя .qt-license.
- Распакуйте содержимое архива qt-win-opensource-src-4.x.x.zip или qt-win-commercial-src-4.x.x (в конце имени файла указывается номер версии) в любой каталог, например, c:\qt440 (полное имя каталога не должно содержать пробелов).
- Добавьте путь к каталогу c:\qt440\bin в системную переменную `PATH`. Перезагрузите компьютер.
- Перейдите в каталог c:\qt440.

Замечание

Если библиотека Qt компилируется повторно (например, с новым набором параметров), то предварительно нужно выполнить команду `nmake confclean`.

- Теперь запустите программу `configure.exe`. При этом могут быть указаны дополнительные параметры, например:
  - `-platform x` — платформа установки. Вместо `x` можно указать `win32-g++`, а для коммерческих версий Qt: `win32-icc`, `win32-msvc`, `win32-msvc.net` и `win32-msvc2005`;
  - `-release` — собирать только конфигурацию Release (без информации для отладки на уровне исходного текста);
  - `-debug` — собирать только конфигурацию Debug (с информацией для отладчика);
  - `-debug-and-release` — собирать обе конфигурации;
  - `-shared` — собирать динамические библиотеки;
  - `-static` — собирать статические библиотеки;
  - `-exceptions` — использовать исключения;
  - `-no-exceptions` — не использовать исключения;

- `-no-qt3support` — не компилировать модули поддержки старой версии Qt 3;
- `-fast` — генерировать только те make-файлы, которые необходимы для компиляции библиотеки.

Полный список всех опций можно узнать, запустив `configure` с параметром `-help`. Если никаких параметров не задавать, то по умолчанию будут компилироваться динамические библиотеки. Иногда утилита `configure` не может определить целевую платформу автоматически, в этом случае придется указать опцию `-platform` вручную.

- ❒ В результате работы `configure` будут сгенерированы make-файлы для сборки библиотеки Qt, инструментов разработки и демонстрационных примеров. Не выходя из каталога установки, запустите процесс компиляции с помощью команды `mingw32-make` (или `nmake`, если используется Microsoft Visual C++) и приготовьтесь к длительному ожиданию.
- ❒ Если ставилась коммерческая версия Qt, то установите интегратор с Microsoft Visual Studio, для этого запустите на выполнение файл qt-vsintegration-x.x.x.exe. После этого откройте Microsoft Visual Studio, выполните команду меню **Tools** (Инструменты) | **Optrions** (Опции), в группе **Qt** выберите страницу **Builds** и, нажав кнопку **Add** (Добавить), впишите произвольное имя версии Qt (например, `qt440`) и укажите полное имя каталога, в который установлена библиотека Qt (например, c:\Qt\qt440). После нажатия кнопки **Ok** из раскрывающегося списка выберите значение параметра **Default Qt Version** (Версия Qt по умолчанию).

После компиляции библиотеки Qt 4 в каталоге bin появятся следующие исполняемые файлы:

- `qmake` — утилита для создания файла проекта *.pro и make-файлов;
- `assistant` — программа просмотра документации;
- `designer` — визуальный редактор графического интерфейса пользователя;
- `moc` — метакомпилятор, предназначенный для перевода исходных текстов программ, написанных с использованием библиотеки Qt, на обычный язык C++;
- `uic` — компилятор ui-файлов, создаваемых программой designer;
- `rcc` — компилятор qrc-ресурсов;
- `qtdemo` — оболочка для запуска демонстрационных программ;
- `linguist` — инструмент, облегчающий перевод интерфейса приложений на другие языки;

- `qt3to4` — утилита для конвертации старых программ, разработанных с использованием 3-й версии библиотеки;
- `lupdate`, `lrelease` — утилиты для интернационализации приложений.

3.2.2. Компиляция в системе Linux/FreeBSD

Поскольку библиотека Qt является основой для KDE, то, скорее всего, она уже установлена в вашей системе. Но даже в этом случае могут отсутствовать некоторые средства разработки, т. к. в KDevelop существуют свои собственные инструменты. Кроме того, обычно вместе с дистрибутивом Linux распространяются не самые свежие версии Qt.

Порядок установки следующий:

- ❒ Предварительно должен быть установлен компилятор `g++` или `icc`.
- ❒ Распакуйте содержимое архива qt-x11-opensource-src-4.x.x.tar.gz в какой-нибудь временный каталог:

```
cd /tmp
gunzip qt-x11-opensource-4.x.x.tar.gz
tar xvf qt-x11-opensource-4.x.x.tar
```

- ❒ Перейдите в этот каталог и запустите программу `configure`:

```
cd /tmp/qt-x11-opensource-4.x.x
./configure
```

При необходимости разрешается задать параметры сборки, список которых можно узнать, если при запуске `configure` указать ключ `-help`.

- ❒ Не выходя из каталога установки, запустите процесс компиляции с помощью команды `make` и приготовьтесь к длительному ожиданию.
- ❒ Для копирования всех необходимых файлов в системные каталоги дайте команду

```
su -c "make install"
```

и введите пароль суперпользователя.

- ❒ Добавьте в переменную окружения `PATH` путь к исполняемым файлам и библиотекам Qt. Для этого в случае использования оболочек bash, ksh, zsh или sh требуется в файл *.profile добавить строки:

```
PATH=/usr/local/Qt4/bin:$PATH
export PATH
```

Для csh и tcsh:

```
setenv PATH /usr/local/Qt4/bin:$PATH
```

3.3. Простейшее приложение Qt 4

ЗАДАЧА. Реализовать с помощью библиотеки Qt 4 простейшее приложение: вывести на экран пустое окно с заголовком *"Hello, Qt4"*.

Решение. Программа, решающая поставленную задачу, приведена в листинге 3.1, а внешний вид окна в системе Windows показан на рис. 3.1.

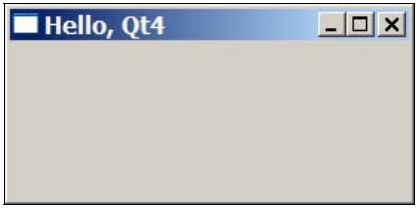


Рис. 3.1. Простейшее приложение Qt 4: пустое окно с заголовком

Листинг 3.1. Простейшее приложение Qt 4 (файл chap03/emptywin/emptywin.cpp)

```
1  // Простейшее приложение Qt 4 (пустое окно)
2
3  #include <QApplication>
4  #include <QMainWindow>
5
6  int main(int argc, char *argv[]) {
7
8      QApplication app(argc, argv);
9
10     QMainWindow mw;
11     mw.setWindowTitle("Hello, Qt4");
12     mw.resize(400, 300);
13     mw.show();
14
15     return app.exec();
16 }
```

Пояснения к программе:

- (3, 4) — подключили заголовочные файлы с определениями классов `QApplication` (приложение) и `QMainWindow` (главное окно);
- (6) — обязательная функция `main()`, как обычно, должна возвращать результат типа `int`. Входные аргументы — это количество параметров входной строки, переданных при запуске программы, и массив строк — значений этих параметров;

- (8) — объявили переменную `app` (от англ. *application* — приложение) типа `QApplication`. Параметры конструктора — те же, что и у функции `main()`;
- (10) — создали главное окно приложения — экземпляр класса `QMainWindow`;
- (11) — установили текст заголовка окна (пока мы не отвлекаемся на проблемы использования кириллицы);
- (12) — установили ширину и высоту окна (400×300 пикселов);
- (13) — вывели окно на экран с помощью метода `show()`;
- (15) — вызовом метода `exec()` запустили цикл обработки событий. Пока в нашей программе никакие события не определены, кроме стандартных реакций на действия пользователя (изменение размеров и положения окна, нажатие на кнопки в строке заголовка и т. д.).

3.4. Компиляция приложений Qt 4 из командной строки

ЗАДАЧА. С помощью командной строки выполнить сборку приложения, использующего библиотеку Qt 4.

Решение. Чтобы облегчить процесс компиляции и компоновки приложений Qt 4 на различных платформах, в этой библиотеке используется собственный формат файла проекта PRO и утилита командной строки `qmake`.

Весь процесс сборки разделяется на три этапа.

Шаг 1. Генерация файла описания проекта

Для этого предусмотрена команда `qmake -project`, которую нужно выполнять в том каталоге, где находятся файлы с исходными текстами проекта. По умолчанию имя файла проекта будет совпадать с названием этого каталога. Для задания другого имени pro-файла при вызове `qmake` следует указать параметр `-o`. Можно явно задать имя cpp-файла:

```
qmake -project -o ИмяФайлаPro ИмяФайлаCPP
```

Для программы из листинга 3.1 в результате выполнения этой команды будет создан файл с расширением pro (листинг 3.2).

Листинг 3.2. Пример файла проекта

```
1    TEMPLATE = app
2    TARGET = emptywin
3    DEPENDPATH += .
4    INCLUDEPATH += .
5
```

```
6    # Input
7    SOURCES += emptywin.cpp
```

Если используется Microsoft Visual C++, то все команды необходимо выполнять в консольном окне **Microsoft Visual Studio Command Prompt**. В некоторых системах Linux вместо `qmake` следует писать `qmake-qt4`. Дело в том, что популярный менеджер рабочего стола KDE пока что построен на основе предыдущей версии библиотеки Qt 3, поэтому в одной системе приходится уживаться и старой Qt 3, и новой Qt 4. Установленную версию библиотеки можно узнать с помощью команд

```
qmake -v
```

или

```
qmake-qt4 -v
```

Шаг 2. Генерация make-файла

Далее необходимо сформировать make-файл, выполнив команду:

```
qmake
```

Шаг 3. Сборка проекта

Теперь нужно запустить компиляцию и компоновку проекта. В Microsoft Visual C++ для этого служит команда

```
nmake
```

Можно явно указать имя make-файла:

```
nmake -f ИмяФайла
```

Если в текущем каталоге имеется файл `Makefile`, то его имя указывать не обязательно. Кроме того, можно задать режим компиляции Debug или Release, например:

```
nmake release
```

По умолчанию действует режим Debug. Для выбора режима, действующего по умолчанию, к тексту pro-файла (перед выполнением шага 2) можно добавить специальную директиву:

```
CONFIG += release
```

или

```
CONFIG += debug
```

Можно указать:

```
CONFIG += debug_and_release
```

Тогда по команде `nmake all` будут собраны обе конфигурации.

Если у вас установлен компилятор MinGW для Windows, то команду `nmake` нужно везде заменить на `mingw32-make`. Для GNU C++ в Linux/FreeBSD вместо `nmake` следует писать просто `make`.

Внимание!

После внесения исправлений в текст файла описания проекта *.pro нельзя снова выполнять команду `qmake` с параметром `-project`, иначе все изменения в этом файле будут потеряны.

Поскольку в процессе разработки программу приходится много раз перекомпилировать, описанный трехшаговый процесс лучше всего проводить с помощью командного файла (листинг 3.3 для MinGW в системе Windows).

Листинг 3.3. Пример командного файла

```
1    qmake -project -o emptywin.pro emptywin.cpp
2    qmake -makefile emptywin.pro
3
4    rem mingw32-make debug
5    mingw32-make release
6    pause
```

Поясним приведенный пример:

- (1) — создаем файл описания проекта `emptywin.pro`;
- (2) — создаем make-файл;
- (4) — выполняем компиляцию в режиме Debug, т. е. с информацией для отладчика (в данном случае строка закомментирована);
- (5) — выполняем компиляцию в режиме Release (без возможности отладки на уровне исходного текста программы);
- (6) — ждем нажатия любой клавиши (если требуется, чтобы консольное окно не закрылось раньше времени и мы успели прочитать все сообщения компилятора).

3.5. Компиляция приложений Qt 4 с помощью интегрированной среды Microsoft Visual Studio

Вместе с библиотекой Qt до выхода 4-го релиза поставлялась интегрированная среда разработки. Но потом фирма Trolltech сосредоточила основные усилия на том, чтобы улучшить совместимость своего продукта со стандарт-

ными средствами разработки программ на языке C++. Поэтому теперь в нашем распоряжении имеется специальная утилита Qt Designer, предназначенная только для редактирования элементов графического интерфейса пользователя. Для работы с исходными текстами программами, компиляции и отладки приложений в среде Windows предлагается использовать Microsoft Visual Studio 6.0, .NET 2003 или 2005, а в Linux — KDevelop.

Чтобы Microsoft Visual Studio "понимала" формат файла PRO, могла работать с ресурсами библиотеки Qt, и выдавала подсказку по классам и функциям, определенным в Qt, необходимо установить специальный интегратор. К сожалению, он поставляется только с коммерческими версиями библиотеки.

ЗАДАЧА. Выполнить сборку приложения, использующего библиотеку Qt 4, в интегрированной среде Microsoft Visual Studio 2005.

Решение.

Вариант 1. Если у нас уже имеется файл *.pro, то в Microsoft Visual Studio достаточно выполнить команду меню **Qt** | **Open Solution from .pro File**, чтобы получить обычный проект, который затем можно компилировать и отлаживать, как и любой другой проект Visual Studio.

Вариант 2. Если мы хотим создать проект Visual Studio "с нуля", то нужно выполнить команду меню **File** (Файл) | **New** (Новый) | **Project** (Проект) и в группе **Qt Projects** выбрать один из возможных типов проекта: **Qt Application** (Приложение Qt), **Qt Console Application** (Консольное приложение Qt), **Qt Library** (Библиотека Qt), **Qt4 Designer Plugin** (Плагин для Qt Designer) или **ActiveQt Server** (Сервер ActiveQt).

3.6. Использование кириллицы

Теперь, как и обещали, обсудим вопрос о символах национальных алфавитов.

ЗАДАЧА. Требуется использовать заданные в исходном тексте программы строковые значения, включающие символы кириллицы.

Решение. Если в тексте программы в строковых значениях встречаются символы, расположенные за пределами первой половины кодовой таблицы, т. е. отличные от стандартной латиницы, то для их правильного отображения при выполнении программы разработчику необходимо указать правильную кодировку. Для этого существуют несколько вариантов.

Вариант 1. Простейший способ работы с национальными алфавитами обеспечивает функция `QString::fromLocal8Bit()`. Дело в том, что для хранения строковых значений в Qt имеется класс `QString`, который оперирует двухбайтовыми символами Unicode. Для преобразования обычных строк с однобай-

товыми символами к типу `QString` как раз и предназначен метод `fromLocal8Bit()`. При этом действует системная кодировка, т. е. для системы Windows считается, что текст программы сохранен в кодировке `cp1251`, а для Linux — обычно в кодировке `koi8-r` или `utf-8`.

Недостаток данного варианта заключается в том, что если программа разрабатывалась, например, в системе Windows, а затем мы захотели скомпилировать ее исходные тексты в Linux, где применяется другая кодовая таблица для символов кириллицы, то предварительно придется осуществить перекодировку исходных текстов, иначе после компиляции в работающей программе все русские сообщения окажутся нечитаемыми.

Измененный текст нашей первой программы приведен в листинге 3.4, а внешний вид окна в системе Linux показан на рис. 3.2.

Листинг 3.4. Символы кириллицы в тексте программы (файл chap03/emptywin-rus1/emptywin-rus1.cpp)

```
// Простейшее приложение Qt4 (пустое окно)
// Вариант с кириллицей в заголовке

#include <QApplication>
#include <QMainWindow>

int main(int argc, char *argv[]) {

    QApplication app(argc, argv);

    QMainWindow mw;
    mw.setWindowTitle(QString::fromLocal8Bit("Пустое окно Qt4"));
    mw.resize(400, 300);
    mw.show();

    return app.exec();
}
```

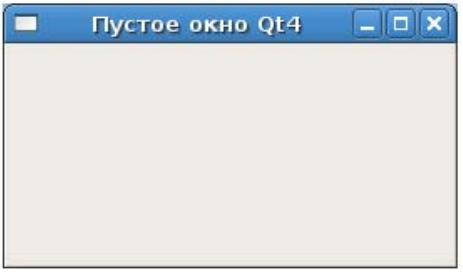


Рис. 3.2. Символы кириллицы в заголовке окна

Вариант 2. Другой вариант работы с символами кириллицы — явное использование *кодеков* (специальных объектов для перекодировки строк). Соответствующий класс `QTextCodec` определен в заголовочном файле с тем же именем (без расширения! Заголовочные файлы с расширением h остались только в старых проектах Qt 3). При создании кодека сначала нужно указать название кодовой таблицы, например:

```
QTextCodec *codec = QTextCodec::codecForName("CP1251");
```

а затем связать этот кодек со всеми строками C++:

```
QTextCodec::setCodecForCStrings(codec);
```

Поскольку при создании кодека указана конкретная кодовая таблица, используемая для хранения текста программы, то исход компиляции не зависит от системной кодировки той платформы, на которой производится сборка программы: результат будет везде одним и тем же. Но файлы с исходными текстами программ в этом случае нельзя подвергать перекодировке (или после перекодировки требуется изменить название кодовой таблицы, заданной при создании кодека). В листинге 3.5 приведен новый вариант нашей программы.

Листинг 3.5. Использование кодеков (файл chap03/emptywin-rus2/emptywin-rus2.cpp)

```
// Простейшее приложение Qt4 (пустое окно)
// Вариант с использованием кодека

#include <QApplication>
#include <QMainWindow>
#include <QTextCodec>

int main(int argc, char *argv[]) {

    QApplication app(argc, argv);

    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForCStrings(codec);

    QMainWindow mw;
    mw.setWindowTitle("Пустое окно Qt4");
    mw.resize(400, 300);
    mw.show();

    return app.exec();
}
```

В строке (12) вместо `CP1251` можно указать любую другую кодовую таблицу, например, `KOI8-R` или `UTF-8`. Главное, чтобы текст программы действительно был сохранен в этой кодировке.

Вариант 3. Наконец, наиболее предпочтительный метод работы с символами национальных алфавитов реализует специальная *функция перевода* `tr()`, с помощью которой осуществляется *интернационализация* приложений. Подробнее данный вопрос мы обсудим позже, а пока договоримся все строковые значения, указанные в тексте программы, передавать в качестве параметра функции `tr()`. Эта статическая функция является членом всех классов библиотеки Qt, порожденных от базового класса `QObject`, но если, как сейчас, мы собираемся вызвать ее в главной программе, а не в каком-либо методе класса, то приходится указывать какой-нибудь подходящий объект, например, `QObject::tr()`. Для задания кодировки, используемой функцией перевода, нужно создать соответствующий кодек и передать его в качестве аргумента методу `setCodecForTr()`. Окончательный вариант нашей программы приведен в листинге 3.6.

Листинг 3.6. Использование функции перевода (файл chap03/emptywin-rus3/emptywin-rus3.cpp)

```
 1  // Простейшее приложение Qt4 (пустое окно)
 2  // Кодеки и функция tr()
 3
 4  #include <QApplication>
 5  #include <QMainWindow>
 6  #include <QTextCodec>
 7
 8  int main(int argc, char *argv[]) {
 9
10      QApplication app(argc, argv);
11
12      QTextCodec *codec = QTextCodec::codecForName("CP1251");
13      QTextCodec::setCodecForTr(codec);
14
15      QMainWindow mw;
16      mw.setWindowTitle(QMainWindow::tr("Пустое окно Qt4"));
17
18      mw.resize(400, 300);
19      mw.show();
20
21     return app.exec();
22  }
```

В строке (12) вместо `CP1251` можно указать любую другую кодовую таблицу, например, `KOI8-R` или `UTF-8`.

3.7. Консольное приложение Qt

ЗАДАЧА. Разработать консольное приложение Qt, которое при старте выводит на экран текущую дату и время.

Решение. В листинге 3.7 приведен текст консольной программы, а на рис. 3.3 показан внешний вид ее окна.

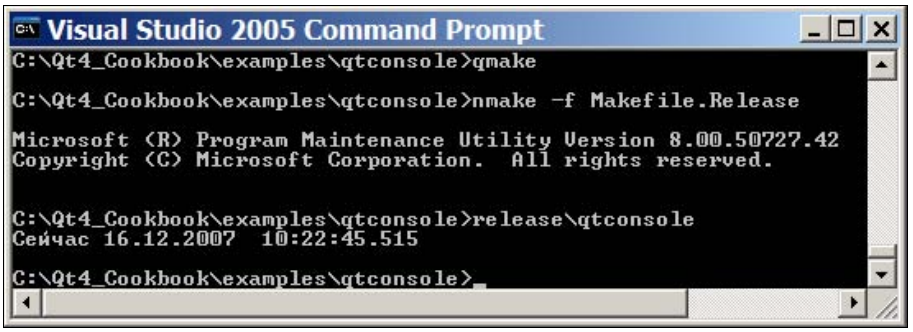


Рис. 3.3. Консольное приложение Qt (точное время)

Листинг 3.7. Консольное приложение Qt (файл chap03/qtconsole/qtconsole.cpp)

```
// Консольное приложение Qt
// Выводит на экран текущую дату и время

#include <QtCore>
#include <iostream>

int main(int argc, char *argv[]) {
    // QCoreApplication app(argc, argv);
    QTextCodec *incodec = QTextCodec::codecForName("CP1251");
    QTextCodec *outcodec = QTextCodec::codecForName("CP866");
    QTextCodec::setCodecForTr(incodec);

    QDateTime now = QDateTime::currentDateTime();
    QString s = QObject::tr("Сейчас ")
              + now.toString("dd.MM.yyyy  hh:mm:ss.zzz");
    std::cout << outcodec->fromUnicode(s).constData()
              << std::endl;
```

```
18
19      return 0; // return app.exec();
20  }
```

Комментарии к программе:

- (4) — подключили заголовочный файл, содержащий определения основных классов ядра библиотеки Qt (это короче, чем указывать по отдельности <QCoreApplication>, <QTextCodec>, <QString> и <QDateTime>);
- (5) — подключили стандартную библиотеку C++ для потокового ввода-вывода;
- (8, 19) — если в программе определить экземпляр класса QCoreApplication (8) и запустить цикл обработки событий (19), то в консольном приложении будет доступен механизм сигналов и слотов, т. е. возможность программировать нужные реакции на внешние события (пока в нашей программе по умолчанию определены только реакции на стандартные события). В результате выход из консольного приложения будет осуществляться не сразу, а при закрытии консольного окна или нажатием комбинации клавиш <Ctrl>+<C>;
- (9–11) — создали кодеки для входного (ввод с клавиатуры) и выходного потоков (вывод на консольный экран) и связали первый из них со строками, переданными в качестве аргумента функции перевода tr(). Два разных кодека приходится использовать только в системе Windows, поскольку в ней системная кодировка CP1251 отличается от кодировки консоли CP866. В Linux оба кодека можно заменить одним: KOI8-R или UTF-8;
- (13) — создали экземпляр класса QDateTime (дата и время) и с помощью функции QDateTime::currentDateTime() записали в него текущую дату и время;
- (14, 15) — преобразовали дату и время из внутреннего формата QDateTime в строковое представление QString с помощью функции QDateTime::toString(). Обратите внимание, что для строки, содержащей символы кириллицы, мы вызвали функцию перевода tr();
- (16, 17) — вывели на консоль получившуюся строку и символ конца строки. Для преобразования строки к нужной локали из формата Unicode, используемого классом QString, мы вызываем функцию QTextCodec::fromUnicode(), а для получения указателя на стандартную строку C++, которую можно передать потоку вывода std::cout, применяем функцию QByteArray::constData();
- (19) — завершили выполнение программы (или запустили цикл обработки событий).

Перед компиляцией консольного приложения, после выполнения команды

```
qmake -project
```

необходимо добавить в сформированный файл описания проекта *.pro строку

```
CONFIG += console
```

после чего можно, как и раньше, выполнить команду `qmake` для генерации make-файла и, наконец, вызвать компилятор и компоновщик с помощью команды `nmake` (или `make`, или `mingw32-make`).

Вместо строки

```
10   QTextCodec *outcodec = QTextCodec::codecForName("CP866");
```

рассчитанной на то, что программу будут компилировать только для системы Windows, можно написать фрагмент, приведенный в листинге 3.8.

Листинг 3.8

```
#ifdef Q_WS_WIN  //Если программа компилируется для системы Windows
QTextCodec *outcodec = QTextCodec::codecForName("CP866");
#else // Иначе
QTextCodec *outcodec = QTextCodec::codecForLocale();
#endif
```

Здесь учитывается тот факт, что в Linux, в отличие от Windows, кодировка консоли совпадает с системной.

3.8. Создание новых классов

Любые классы, разрабатываемые программистом, обычно являются наследниками базового класса `QObject` (или какого-нибудь из его потомков). Только в этом случае для них можно использовать стандартные механизмы библиотеки Qt, к рассмотрению которых мы скоро приступим: сигналы и слоты, сведения о типах во время выполнения программы, динамические свойства объектов. При этом необходимо соблюдать следующие правила:

- в секции `private` при объявлении класса нужно указать макрос `Q_OBJECT`;
- объявления классов, содержащих макрос `Q_OBJECT`, *обязательно должны располагаться в заголовочном файле* *.h, а не в самой программе *.cpp.

Утилита `qmake`, которую мы запускаем перед компиляцией, просматривает заголовочные файлы проекта, и если в описании какого-нибудь класса встречается макрос `Q_OBJECT`, то в make-файл вставляется вызов так называемого *компилятора метаобъектов* `moc`, который автоматически формирует на язы-

ке C++ необходимый код (в папке проекта после компиляции можно найти файл `moc_*.cpp`).

ЗАДАЧА. Разработать класс `MyWin` на основе стандартного класса `QMainWindow` (главное окно приложения).

Решение. Мы снова создаем пустое окно (рис. 3.4, листинги 3.9–3.11). Только теперь это будет наш собственный класс, который полностью готов для расширения своей функциональности.

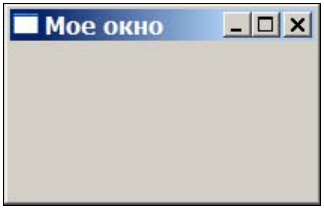


Рис. 3.4. Наследник `QMainWindow`

Листинг 3.9. Файл chap03/mywin/mywin.h

```
#ifndef _MYWIN_H_
#define _MYWIN_H

#include <QtGui>

class MyWin : public QMainWindow {
   Q_OBJECT
public:
   MyWin(const QString& title);
};

#endif // _MYWIN_H_
```

Листинг 3.10. Файл chap03/mywin/mywin.cpp

```
#include "mywin.h"

MyWin::MyWin(const QString& title)
      : QMainWindow(0) {

    setWindowTitle(title);
    resize(200, 120);
```

```
20      move(150, 100);
21  }
```

Листинг 3.11. Файл chap03/mywin/main.cpp

```
22  #include "mywin.h"
23
24  int main(int argc, char *argv[]) {
25     QApplication app(argc, argv);
26     QTextCodec *codec = QTextCodec::codecForName("CP1251");
27     QTextCodec::setCodecForTr(codec);
28
29     MyWin win(QObject::tr("Мое окно"));
30     win.show();
31     return app.exec();
32  }
```

Пояснения к программе:

- (1, 2, 12) — в нашей простой программе это не обязательно, но при разработке сложных проектов, состоящих из большого числа заголовочных файлов, чтобы избежать повторного включения любого из них, применяют стандартный способ: текст заголовочного файла заключают в тело директивы условной компиляции;
- (4) — для ускорения компиляции часто вместо включения всего модуля библиотеки Qt подключают только необходимые заголовочные файлы. В нашем случае вполне можно было обойтись строкой

  `#include <QMainWindow>;`
- (6) — объявляем новый класс на основе `QMainWindow`;
- (7) — не забыли про макрос `Q_OBJECT`;
- (9, 15–21) конструктор с одним параметром (строка заголовка окна);
- (15) — как правило, при передаче объекта в качестве параметра функции указывают не сам объект, а ссылку на него (тогда в стек копируется только адрес экземпляра класса). Если параметр входной, т. е. его значение не должно изменяться после выхода из функции, то указывают модификатор `const`;
- (16) — не забыли вызвать конструктор базового класса. Необязательный параметр — указатель на родительский элемент. В нашем случае родителя нет;

- (18) — установили текст заголовка окна;
- (19) — задали размеры окна по горизонтали и вертикали (в пикселах);
- (20) — координаты левого верхнего угла окна в системе отсчета родителя (левый верхний угол соответствует началу отсчета, горизонтальная ось координат направлена вправо, вертикальная — вниз). В нашем случае родителя нет, поэтому имеются в виду координаты экрана;
- (29) — создали экземпляр разработанного нами класса и указали текст заголовка. Это главное окно, не имеющее родителя, поэтому оператор `new` отсутствует;
- (30) — вывели окно на экран.

3.9. Правила именования объектов в библиотеке Qt

> Многих вещей нет потому, что их не смогли никак назвать.
>
> *Станислав Ежи Лец*

Сформулируем основные правила именования программных объектов, которых придерживаются разработчики библиотеки Qt.

Имена классов начинаются с большой буквы. Классы библиотеки Qt начинаются с символа `Q`, например, `QMainWindow`. Первые символы каждого слова в составных именах заглавные.

Имена функций и любых членов класса начинаются со строчной буквы, но первые буквы всех остальных слов (кроме первого) в составных именах, тем не менее, делаются заглавными, например, `setTabOrder()`.

Перед переменными — членами класса не ставится префикс `m_`.

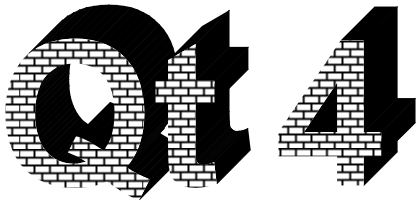
Функции библиотеки Qt, входящие в глобальное пространство имен, начинаются со строчной буквы `q`, после которой пишется заглавная буква, например: `qDebug()`.

Метод чтения для свойства не имеет в начале имени, как это обычно принято, префикса `get`, а просто повторяет название свойства. Метод записи, тем не менее, имеет префикс `set`. Пример: для свойства `minimumWidth` (минимальная ширина) методы доступа называются `minimumWidth()` и `setMinimumWidth()`. Исключение из этого правила допускается для свойств логического типа, имена функций чтения для которых иногда имеют префикс `is` или `has`, например, `isEnabled()` или `hasFocus()`.

Если название класса, функции или свойства содержит аббревиатуру, то входящие в нее символы считаются одним словом и поэтому все буквы, кроме, может быть, первой, пишутся в нижнем регистре. Примеры: `QSslSocket` для названия класса, `sslErrors()` и `setSslConfiguration()` для имен функций.

Объявление каждого класса принято помещать в отдельный заголовочный файл *.h, название которого повторяет имя класса, но записывается строчными буквами. Данное правило не относится к внутренним (служебным) классам, объявления которых обычно помещают в файлы *.cpp или *_p.h.

ГЛАВА 4



Текстовые надписи

Предпочитаю надпись "вход воспрещен" надписи "выхода нет".

Станислав Ежи Лец

Для показа текстовых надписей в библиотеке Qt предусмотрен класс `QLabel`, экземпляры которого создаются, как обычно, в C++:

```
QLabel mylabel;
```

или

```
QLabel *mylabel = new QLabel ( parent, flag );
```

или

```
QLabel *mylabel = new QLabel ( text, parent, flag );
```

Здесь `text` — текст надписи (константа или переменная типа `QString`); `parent` — указатель на родительский элемент (экземпляр класса `QWidget` или его потомка); `flag` — комбинация битовых флагов `Qt::WindowFlags`, задающих внешний вид фрейма для надписи.

Если родитель не указан, то значение параметра `parent` по умолчанию считается равным нулю, т. е. элемент не имеет родителя. Если в качестве родителя указан какой-либо другой элемент графического интерфейса, то создаваемая надпись будет автоматически уничтожена при выполнении деструктора своего родителя. Описанный механизм общий для всех элементов библиотеки Qt и позволяет программисту забыть о необходимости явно вызывать деструкторы для освобождения памяти.

Внимание!

Следует помнить, что для правильного удаления дочерних объектов требуется, чтобы все виджеты — потомки класса `QWidget` — размещались в динамической

памяти (т. е. создавались с помощью оператора `new`). Исключение можно сделать только для тех объектов, при создании которых не был указан родитель `parent`.

Последний параметр конструктора — комбинация флагов — при создании надписей обычно не указывают (по умолчанию считается равным нулю).

Текст надписи, если он не задан в конструкторе или если его требуется изменить для уже созданного элемента, можно задать с помощью метода `QLabel::setText(text)`. При этом можно передать либо обычную текстовую строку, либо фрагмент кода на языке разметки гипертекста HTML (HyperText Markup Language). Для отображения целого или вещественного числа предусмотрен метод `setNum(число)`, для отображения графического изображения — `setPixmap()`. При вызове этих методов предыдущее содержимое надписи очищается.

Выравнивание текста надписи по вертикали и/или горизонтали указывают или внутри фрагмента гипертекста с помощью соответствующих тегов языка HTML, или в методе `QLabel::setAlignment(режим)`. Параметр `режим` задается комбинацией флагов `AlignLeft`, `AlignRight`, `AlignHCenter` и `AlignJustify` для выравнивания по левому/правому краю, по центру или по ширине соответственно, а также `AlignBottom`, `AlignTop` и `AlignVCenter` для выравнивания по нижнему/верхнему краю или по центру. Чтобы центрировать текст надписи по обоим изменениям, нужно указать `AlignHCenter | AlignVCenter` или просто `AlignCenter`. Метод `setWordWrap()` позволяет разрешить или запретить перенос текста на другую строку.

4.1. Форматированный текст

ЗАДАЧА. Вывести с помощью надписи `QLabel` форматированный текст.

Решение. На рис. 4.1 показан внешний вид окна, а в листинге 4.1 приведен текст программы.

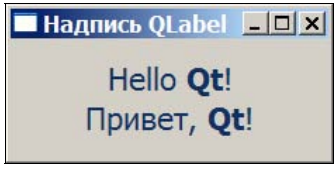


Рис. 4.1. Окно с надписью `QLabel`

Листинг 4.1. Надпись `QLabel` (файл chap04/label/label.cpp)

```
1  #include <QApplication>
2  #include <QLabel>
```

```
 3  #include <QTextCodec>
 4
 5  int main( int argc, char *argv[] ) {
 6    QApplication app(argc, argv);
 7    QTextCodec *codec = QTextCodec::codecForName("CP1251");
 8    QTextCodec::setCodecForTr(codec);
 9
10    QLabel label( QObject::tr(
11        "<font size=\"+3\" color=\"#103060\"><center>"
12        "   Hello <b>Qt</b>!<br />"
13        "   Привет, <b>Qt</b>!"
14        "</center></font>" ) );
15    label.resize(200, 100);
16    label.setWindowTitle(QObject::tr("Надпись QLabel"));
17    label.show();
18    return app.exec();
19  }
```

Поддерживаются и другие теги HTML: форматирование символов и абзацев, выравнивание, списки, таблицы, рисунки и стили.

4.2. Ссылки Интернета

ЗАДАЧА. Разместить внутри надписи `QLabel` ссылку на сайт в Интернете и адрес электронной почты. По щелчку кнопкой мыши ссылка должна открываться в окне браузера. При щелчке на строке адреса электронной почты должна открываться почтовая программа, зарегистрированная в системе.

Решение. Чтобы разрешить работу ссылок, оформленных с помощью тега `<a href>`, используется метод `QLabel::setOpenExternalLinks()`. Пример надписи, содержащей ссылки, показан на рис. 4.2, а текст программы приведен в листинге 4.2.

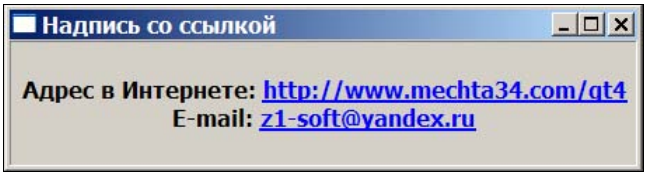


Рис. 4.2. Активные ссылки внутри надписи

Листинг 4.2. Ссылки внутри надписи (файл chap04/clicklabel/clicklabel.cpp)

```
 1  #include <QApplication>
 2  #include <QLabel>
 3  #include <QTextCodec>
 4
 5  int main( int argc, char *argv[] ) {
 6     QApplication app(argc, argv);
 7     QTextCodec *codec = QTextCodec::codecForName("CP1251");
 8     QTextCodec::setCodecForTr(codec);
 9
10     QLabel label( QObject::tr(
11        "Адрес в Интернете: "
12        "<a href=\"http://www.google.com \">"
13        "http://www.google.com</a><br/>"
14        "E-mail: <a href=\"mailto:name@server.ru"
15        "?subject=Книга 'Qt4 Cookbook'&body=Здравствуй, Юрий!\">2
16        "name@server.ru</a>" ) );
17     label.setOpenExternalLinks(true);
18     label.setFrameStyle(QFrame::Panel | QFrame::Sunken);
19     label.setAlignment(Qt::AlignCenter);
20
21     QFont font("Serif", 12, QFont::Bold);
22     label.setFont(font);
23     label.resize(500, 100);
24     label.setWindowTitle(QObject::tr("Надпись со ссылкой"));
25     label.show();
26     return app.exec();
27  }
```

4.3. Редактируемая надпись

ЗАДАЧА. Разрешить пользователю изменять текст надписи `QLabel`.

Решение. Если в предыдущем примере (листинг 4.2) после строки 19 добавить строку

```
label.setTextInteractionFlags(Qt::TextEditorInteraction);
```

то надпись станет редактируемой.

Замечание

Более полно возможности редактирования текста реализованы в классах `QLineEdit` и `QTextEdit`.

4.4. Надписи с рисунками

ЗАДАЧА. Отобразить рисунок внутри надписи `QLabel`.

Решение. Графические изображения можно вставлять двумя способами:

- с помощью тега `<img src=...>`
- методом `QLabel::setPixmap()`:

```
label.setPixmap(QPixmap("images/picture.png"));
```

В первом случае надпись может отображать как текст, так и графику, во втором — только графику без текста. Пример надписи с графическим изображением показан на рис. 4.3, текст программы приведен в листинге 4.3.

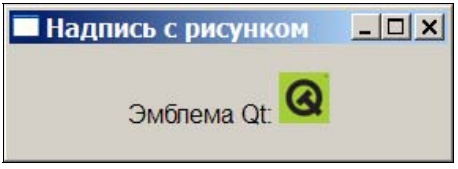


Рис. 4.3. Рисунок внутри надписи `QLabel`

Листинг 4.3. Надпись с рисунком (файл chap04/piclabel/piclabel.cpp)

```
#include <QApplication>
#include <QLabel>
#include <QTextCodec>

int main( int argc, char *argv[] ) {
   QApplication app(argc, argv);
   QTextCodec *codec = QTextCodec::codecForName("CP1251");
   QTextCodec::setCodecForTr(codec);

   QLabel label( QObject::tr(
      "Эмблема Qt: <img src=\"img/qt-logo.png\"/>" ) );
   label.setAlignment(Qt::AlignCenter);
   label.setFont(QFont("Arial", 12));

   label.resize(200, 100);
   label.setWindowTitle(QObject::tr("Надпись с рисунком"));
   label.show();
   return app.exec();
}
```

После компиляции необходимо создать каталог img в том каталоге, *откуда производится запуск программы*, и скопировать в него файл qt-logo.png. Позже мы узнаем, как поместить графические изображения и другие ресурсы внутрь исполняемого файла приложения.

4.5. Несколько элементов на одной форме

ЗАДАЧА. Разместить в окне несколько надписей.

Решение. Для размещения в окне приложения нескольких элементов можно применить прямое позиционирование с помощью методов `QWidget::move()` и `QWidget::resize()` или специальные *менеджеры размещения*. Второй вариант обычно более предпочтителен. Подробнее этот вопрос мы рассмотрим в *разд. 8.2*, а пока приведем простейший пример использования менеджера вертикального размещения `QVBoxLayout` (рис. 4.4, листинг 4.4).

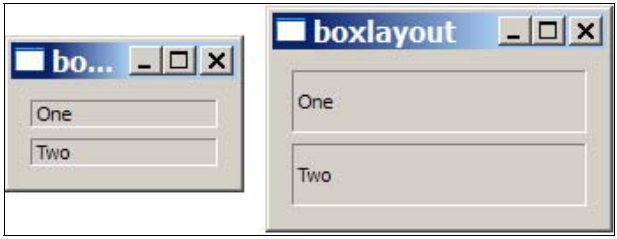


Рис. 4.4. Вертикальный менеджер размещения `QVBoxLayout`

Листинг 4.4. Размещение надписей по вертикали (файл chap04/boxlayout/boxlayout.cpp)

```
#include <QApplication>
#include <QVBoxLayout>
#include <QLabel>

int main(int argc, char *argv[]){
   QApplication app(argc, argv);
   QWidget win;
   QVBoxLayout* layout = new QVBoxLayout(&win);
   QLabel* label1 = new QLabel("One", &win);
   QLabel* label2 = new QLabel("Two", &win);

   label1->setFrameStyle(QFrame::Panel | QFrame::Sunken);
   label2->setFrameStyle(QFrame::Panel | QFrame::Sunken);

```

```
15      layout->addWidget(label1);
16      layout->addWidget(label2);
17
18      win.setLayout(layout);
19      win.show();
20      return app.exec();
21  }
```

Пояснения к программе:

- (6) — создали объект-приложение;
- (7) — виджет, на котором будем размещать надписи;
- (8) — создали менеджер вертикального размещения;
- (9, 10) — две надписи;
- (12, 13) — надписи будут выглядеть "утопленными";
- (15, 16) — добавили надписи к менеджеру размещения;
- (18) — назначили менеджер размещения для главного виджета;
- (19) — показали главный виджет;
- (20) — запустили цикл обработки событий.

Внимание!

Единственный объект (основной виджет), не имеющий родителя, создан без использования оператора `new`. При создании всех остальных виджетов указан родитель. В строках (9 и 10) мы перестраховались, поскольку при добавлении элемента к менеджеру размещения с помощью метода `addWidget()` этот менеджер автоматически становится родителем данного элемента.

Заметим, что надписи автоматически растягиваются при увеличении размеров окна. Если требуется разместить элементы в одной строке по горизонтали, то вертикальный менеджер размещения `QVBoxLayout` заменяют горизонтальным `QHBoxLayout`.

ЗАДАЧА. Разместить в окне четыре надписи в две колонки и два ряда.

Решение. Для размещения элементов в нескольких колонках и рядах предназначен сеточный менеджер `QGridLayout` (рис. 4.5, листинг 4.5).

Рис. 4.5. Сеточный менеджер размещения

Листинг 4.5. Сеточный менеджер размещения (файл chap04/gridlayout/gridlayout.cpp)

```
#include <QApplication>
#include <QGridLayout>
#include <QLabel>

int main(int argc, char *argv[]){
   QApplication app(argc, argv);
   QWidget win;
   QGridLayout* layout = new QGridLayout(&win);

   QLabel* label1 = new QLabel("One");
   QLabel* label2 = new QLabel("Two");
   QLabel* label3 = new QLabel("Three");
   QLabel* label4 = new QLabel("Four");

   layout->addWidget(label1, 0, 0, Qt::AlignCenter);
   layout->addWidget(label2, 0, 1, Qt::AlignCenter);
   layout->addWidget(label3, 1, 0, Qt::AlignCenter);
   layout->addWidget(label4, 1, 1, Qt::AlignCenter);

   win.setLayout(layout);
   win.show();
   return app.exec();
}
```

При добавлении элемента к сеточному менеджеру размещения первым параметром ставится указатель на добавляемый элемент, далее задаются целочисленный номер строки и номер столбца, после чего следует необязательный параметр выравнивания.

4.6. Диалог с текстовым сообщением

ЗАДАЧА. Вывести на экран диалог с текстовым сообщением.

Решение. Для вывода на экран модальных диалогов с текстовыми сообщениями служат статические методы класса `QMessageBox`: функции `information()`, `question()`, `critical()` и `warning()`, что в переводе с английского означает соответственно: "информация", "вопрос", "критическая ошибка" и "предупреждение". При вызове всех четырех функций первым параметром указывается родительский элемент, вторым — текст заголовка окна, третьим — текст сообщения, четвертым — набор битовых флагов, опреде-

ляющий состав кнопок окна диалога (например, для вывода двух кнопок **Да** и **Нет** нужно указать `QMessageBox::Yes | QMessageBox::No`). По умолчанию если четвертый параметр не указан, то выводится только кнопка **OK**. Последний, необязательный параметр задает кнопку по умолчанию, которая срабатывает, если пользователь нажимает клавишу <Enter>. Диалог закрывается при нажатии любой из кнопок или при нажатии клавиши <Esc>. Возвращается код нажатой кнопки. Пример иллюстрирует листинг 4.6.

Листинг 4.6. Пример диалога

```
if ( QMessageBox::question(
        this, tr("Подтвердите"),
        tr("Выйти из программы?"),
        QMessageBox::Ok | QMessageBox::Cancel,
        QMessageBox::Ok )
    ) == QMessageBox::Ok)
 . . . . .
```

Нажатие клавиши <Esc> на клавиатуре воспринимается аналогично нажатию кнопки **Cancel** (Отмена), если эта кнопка задана при создании диалога. Если для диалога определена всего одна кнопка, то именно она сработает при нажатии как клавиши <Enter>, так и <Esc>.

4.7. Форматирование строк

ЗАДАЧА. Необходимо включить в текст сообщения значения переменных различного типа, а для чисел задать формат их вывода (общее количество выводимых символов и число знаков после десятичной точки).

Решение. Для форматирования выводимых сообщений в библиотеке Qt имеется шаблонный символ `%n`, где `n` — число от 1 до 99, и метод `QString::arg()`, который все встречающиеся в строке шаблонные символы `%n` с наименьшим номером заменяет строковым представлением своего аргумента и возвращает получившуюся строку, к которой затем можно повторно применить метод `arg()`, чтобы произвести подстановку для шаблонных символов со следующим номером. Пример:

```
QString str1 = tr("Меня зовут %1, мне %2 лет").arg(name).arg(age);
QString str2 = tr("Из %1 > %2 следует, что %2 < %1").arg(x).arg(y);
```

После выводимого значения в методе `arg()` можно указать необязательные параметры: ширину поля, формат (`E`, `G`, `f`) и точность (число знаков после десятичной точки).

4.8. Ввод текста с клавиатуры

ЗАДАЧА. Разместить в окне приложения поле для ввода/редактирования текстовой строки.

Решение. Для ввода текстовой строки используется класс `QLineEdit`. Соответствующий конструктор:

```
QLineEdit ( QWidget *parent = 0 )
```

или

```
QLineEdit ( const QString& contents, QWidget *parent = 0 )
```

Параметр `contents` задает начальный текст (обычно это пустая строка), а `parent`, как и для надписи, — родительский элемент (если виджет создается с помощью оператора `new`).

Замечание

Обратите внимание, что во всех функциях библиотеки Qt вместо самого экземпляра класса в качестве параметра функции указывается ссылка на него с модификатором `const` (чтобы случайно не испортить значение параметра при выполнении функции).

Узнать текущее текстовое значение, введенное в элемент `QLineEdit`, позволяет метод `text()`. Если поле предназначено для ввода пароля, то с помощью метода `setEchoMode(QLineEdit::Password)` можно установить режим, при котором все символы заменяются звездочками. Метод `setInputMask()` позволяет задать маску ввода.

В листинге 4.7 приведен текст программы, решающей поставленную задачу, на рис. 4.6 показан внешний вид окна приложения в системе Windows.

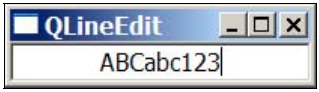


Рис. 4.6. Однострочное поле ввода `QLineEdit`

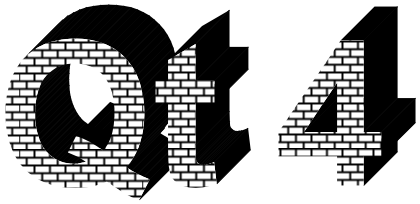
Листинг 4.7. Однострочное поле ввода (файл chap04/lineedit/lineedit.cpp)

```
1  #include <QtGui>
2
3  int main( int argc, char *argv[] ) {
4     QApplication app(argc, argv);
5
```

```
 6     QLineEdit le( "ABCabc123" );
 7     le.setAlignment(Qt::AlignCenter);
 8     le.setFont(QFont("Tahoma", 12));
 9     le.setWindowTitle("QLineEdit");
10     le.show();
11
12     return app.exec();
13  }
```

Здесь мы не стали создавать главное окно явно, поэтому оно было сформировано автоматически. Пользователь может изменять текст внутри элемента `QLineEdit`, но пока мы нигде не используем введенное значение, потому что не знаем главного: как запрограммировать реакции на внешние события (в том числе и действия пользователя). Этому вопросу посвящен следующий раздел.

ГЛАВА 5



Обработка событий

> 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*

Обычно для программирования реакций на действия пользователя и любые другие внешние события в C++ используются *функции обратного вызова* (*callback*), а для связывания событий с этими функциями — специальные макросы (*карты сообщений).* В библиотеке Qt предлагается более удобный подход: механизм *сигналов* и *слотов*.

5.1. Сигналы и слоты

Сигнал — это сообщение о том, что произошло какое-либо событие, например, нажатие на кнопку или выбор пункта меню. У сигнала есть *источник* (например, кнопка) и *приемник* (объект, метод которого будет обрабатывать это событие). *Слот* — это сама функция-обработчик события. Связь между всеми четырьмя перечисленными элементами задается с помощью метода `connect` (соединить), синтаксис которого иллюстрирует листинг 5.1.

Листинг 5.1. Метод `connect`

```
bool QObject::connect (
   const QObject *sender,     // Источник события
   const char *signal,        // Сигнал
   const QObject *receiver,   // Объект-приемник сигнала
   const char *method,        // Функция-обработчик
   Qt::ConnectionType type = Qt::AutoConnection
  ) const
```

Последний параметр определяет режим обработки: `Qt::DirectConnection` — немедленная обработка; `Qt::QueuedConnection` — постановка в общую очередь и обработка после событий, которые уже в ней находятся; `Qt::AutoConnection` — если источник события находится в том же потоке, что и приемник, то будет задан режим `Qt::DirectConnection`, в противном случае — `Qt::QueuedConnection`. Начиная с версии 4.3.0, был введен еще один режим: `Qt::BlockingQueuedConnection` — то же, что и `Qt::QueuedConnection`, но поток, в котором было сгенерировано событие, будет приостановлен до тех пор, пока не наступит очередь для обработки данного события (при этом объект-источник сигнала и объект-приемник должны находиться в разных потоках).

Для определения сигнала и слота предназначены макросы `SIGNAL` и `SLOT`. Например, мы хотим, чтобы текстовая метка `label` (экземпляр класса `QLabel`) отображала позицию полосы прокрутки `scrollBar` (экземпляр класса `QScrollBar`). В документации на библиотеку Qt (открыв doc/html/index.html или программу `assistant`) находим, что в классе `QAbstractSlider`, потомком которого является `QScrollBar`, определен сигнал

```
void QAbstractSlider::valueChanged ( int value ),
```

оповещающий об изменении положения ползунка полосы прокрутки. Далее в описании класса `QLabel` находим, что изменение текста надписи производится с помощью функции `setText(строка)` или `setNum(число)`. Тогда вызов метода `connect` должен выглядеть так, как показано в листинге 5.2.

Листинг 5.2. Пример вызова метода `connect`

```
QObject::connect(
   scrollBar,                 // Источник события
   SIGNAL(valueChanged(int)), // Сигнал
   label,                     // Объект-приемник сигнала
   SLOT( setNum(int) ) );     // Функция-обработчик
```

Другой пример: чтобы связать нажатие кнопки `btn` (экземпляр класса `QPushButton`) с вызовом метода `QApplication::quit()`, который завершает работу всего приложения, нужно написать:

```
connect (btn, SIGNAL(clicked()), qApp, SLOT(quit()));
```

Макрос `qApp` возвращает указатель на экземпляр приложения.

Замечание

Параметрами сигнала и слота являются типы, а не переменные.

Число параметров слота всегда не больше чем сигнала. Соответствие между ними, как обычно, позиционное: при выполнении программы значением *i*-го параметра слота становится значение *i*-го параметра сигнала.

В объявлении класса методы-слоты необходимо указывать в разделе `public slots` или `private slots`. Если поместить их вместе с обычными методами класса, то программа будет скомпилирована без ошибок, но при ее выполнении неправильно объявленные методы-слоты вызываться не будут.

Обычно в программе используются стандартные сигналы, но если требуется определить собственные, то их нужно объявить в разделе `signals` (листинг 5.3).

Листинг 5.3. Пример объявления сигнала

```
class MainWondow : public QMainWindow {
    Q_OBJECT
.............
signals:
    void mySignal(int n);     // Сигнал
private slots:
    void onMySignal(int n);  // Слот
.............
```

Генерацию сигнала в нужном месте программы осуществляет оператор

```
emit mySignal(5);
```

Один сигнал может быть "соединен" с другим, например:

```
connect(myButton,  // Кнопка
        SIGNAL( clicked() ), // Сигнал — нажатие на кнопку
        this,  // Экземпляр класса, в котором определен buttonClicked
        SIGNAL( buttonClicked() ) );
```

Один и тот же сигнал можно связать с несколькими слотами и/или другими сигналами. С одним и тем же слотом можно связать несколько сигналов. Если указать несколько одинаковых соединений, тогда одно событие вызовет генерацию нескольких сигналов и, соответственно, многократное выполнение метода-слота.

Разорвать связь между сигналом и слотом можно с помощью статического метода `disconnect()`: (листинг 5.4).

Листинг 5.4. Пример вызова метода `disconnect`

```
bool QObject::disconnect ( const QObject *sender,
                           const char *signal,
                           const QObject *receiver,
                           const char *method )
```

ЗАДАЧА. Разместить в окне кнопку, при нажатии которой будет выполняться выход из программы.

Решение. В листинге 5.5 приведен текст небольшой программы, иллюстрирующей принцип обработки нажатия на кнопку (экземпляр класса `QPushButton`), а на рис. 5.1 показано, как выглядит окно программы в системе Windows.

Рис. 5.1. Окно с кнопкой

Листинг 5.5. Обработка нажатия на кнопку (файл chap05/btnquit/btnquit.cpp)

```
 1  // Сигналы и слоты: кнопка в окне
 2
 3  #include <QApplication>
 4  #include <QPushButton>
 5
 6  int main(int argc, char *argv[]) {
 7
 8      QApplication app(argc, argv);
 9
10      QPushButton button(QString::fromLocal8Bit("&Выход") );
11      button.setFont(QFont("Times", 16, QFont::Bold));
12      QObject::connect(
13          &button,           // Источник сигнала
14          SIGNAL(clicked()), // Сигнал о нажатии кнопки
15          &app,              // Приемник сигнала
16          SLOT( quit() ) );  // Функция-слот (обработчик события)
17      button.show();
18
19      return app.exec();
20  }
```

Здесь мы разместили в окне обычную кнопку (10) с надписью **Выход** и связали ее нажатие — сигнал `clicked` (14) — с функцией-обработчиком `quit` (16), которая завершает приложение `app`. Заметим, что мы не создаем главное окно для кнопки, это будет сделано автоматически (см. рис. 5.1).

Символ "`&`" перед буквой "`В`" в тексте надписи на кнопке (11) позволяет активировать ее не только по щелчку левой кнопкой мыши или нажатием клавиши <Enter>, но также с помощью комбинации клавиш <Alt>+<в> (к сожалению, только в режиме ввода кириллицы).

ЗАДАЧА. Разместить в окне надпись `QLabel`, поле ввода `QSpinBox` с кнопками инкремента/декремента и ползунок `QSlider`. Текущее значение в поле ввода и позиция ползунка должны быть связаны между собой: при изменении пользователем первого требуется синхронно изменять второе и наоборот. Текст надписи также должен отображать текущее значение и изменяться вместе с ним.

Решение. В листинге 5.6 приведен текст программы, а на рис. 5.2 показан внешний вид окна в системе Windows.

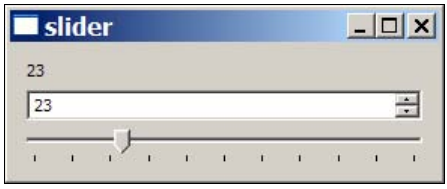


Рис. 5.2. Позиция ползунка `QSlider` и число в поле ввода `QSpinBox` отображается внутри надписи `QLabel`

Листинг 5.6. Ползунок, поле ввода и надпись (файл chap05/slider/slider.cpp)

```
#include <QApplication>
#include <QVBoxLayout>
#include <QLabel>
#include <QSpinBox>
#include <QSlider>

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

   QLabel *label = new QLabel;

   QSpinBox *spinBox = new QSpinBox;
   spinBox->setMinimum(0);
   spinBox->setMaximum(100);
   spinBox->setSingleStep(5);

```

```
17      QSlider *slider = new QSlider(Qt::Horizontal);
18      slider->setRange( spinBox->minimum(),
19                        spinBox->maximum() );
20      slider->setSingleStep(10);
21      slider->setTickPosition(QSlider::TicksBelow);
22      slider->setTickInterval(10);
23
24      QObject::connect( spinBox, SIGNAL(valueChanged(int)),
25                        label, SLOT(setNum(int)) );
26      QObject::connect( spinBox, SIGNAL(valueChanged(int)),
27                        slider, SLOT(setValue(int)) );
28      QObject::connect( slider, SIGNAL(valueChanged(int)),
29                        label, SLOT(setNum(int)) );
30      QObject::connect( slider, SIGNAL(valueChanged(int)),
31                        spinBox, SLOT(setValue(int)) );
32
33      spinBox->setValue(20);
34
35      QWidget window;
36      QVBoxLayout* mainLayout = new QVBoxLayout(&window);
37      mainLayout->addWidget(label);
38      mainLayout->addWidget(spinBox);
39      mainLayout->addWidget(slider);
40      window.show();
41
42      return app.exec();
43  }
```

Пояснения к программе:

- (1–5) — для уменьшения времени компиляции вместо включения всего модуля `<QtGui>` лучше подключить только требуемые заголовочные файлы;
- (10, 12, 17) — создаем надпись `QLabel`, поле `QSpinBox` для ввода целого числа и ползунок `QSlider`;
- (13–15) — устанавливаем диапазон возможных значений, которые можно вводить в поле `QSpinBox`, а также шаг изменения числа при нажатии на кнопки инкремента/декремента (разумеется, остается возможность вручную ввести любое значение, не кратное пяти);
- (18–22) — устанавливаем диапазон чисел для слайдера, а также шаг изменения позиции при нажатии клавиш управления стрелками (остается возможность позиционировать ползунок с помощью мыши произвольно в пределах заданного диапазона);

- (24, 25) — соединяем событие "изменение числа в поле ввода" со слотом "установить число, отображаемое надписью";
- (26, 27) — соединяем событие "изменение числа в поле ввода" со слотом "установить позицию ползунка";
- (28, 29) — соединяем событие "изменение позиции ползунка" со слотом "установить число, отображаемое надписью";
- (30, 31) — соединяем событие "изменение позиции ползунка" со слотом "установить число в поле ввода";
- (33) задаем число, отображаемое полем ввода. При этом автоматически изменятся позиция ползунка и текст надписи (ведь соединения уже установлены);
- (35–40) — создаем окно и с помощью менеджера `QVBoxLayout` размещаем все три элемента.

ЗАДАЧА. Разместить на форме поле для ввода целых чисел с кнопками инкремента/декремента, поворотный переключатель, горизонтальный слайдер и индикатор прогресса. Связать их между собой таким образом, чтобы все они отображали одинаковые значения (рис. 5.3). Соответствующий код приведен в листингах 5.7–5.9.

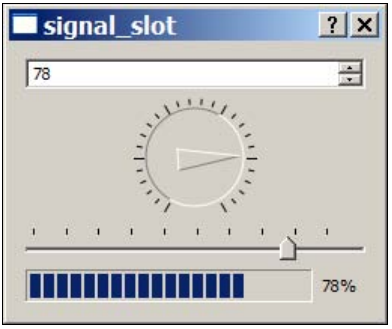


Рис. 5.3. Элементы `QSpinBox`, `QDial`, `QSlider` и `QProgressBar`

Листинг 5.7. Файл mydialog.h

```
#ifndef _MYDIALOG_H_
#define _MYDIALOG_H_

#include <QtGui>

class MyDialog : public QDialog {
    Q_OBJECT
```

```
public:
    MyDialog();
private:
    QSpinBox *m_sb;
    QDial *m_dial;
    QSlider *m_slider;
    QProgressBar *m_pb;
};
#endif // _MYDIALOG_H_
```

Листинг 5.8. Файл mydialog.cpp

```
#include <QtGui>
#include "mydialog.h"

#define MAXVAL 99
#define MINVAL 0

MyDialog::MyDialog()
        : QDialog() {
    m_sb = new QSpinBox(this);
    m_sb->setRange(MINVAL, MAXVAL);

    m_dial = new QDial(this);
    m_dial->setRange(MINVAL, MAXVAL);
    m_dial->setNotchesVisible(true);

    m_slider = new QSlider(Qt::Horizontal, this);
    m_slider->setRange(MINVAL, MAXVAL);
    m_slider->setTickInterval(10);
    m_slider->setTickPosition(QSlider::TicksAbove);

    m_pb = new QProgressBar(this);
    m_pb->setRange(MINVAL, MAXVAL);

    QVBoxLayout *layout = new QVBoxLayout(this);
    layout->addWidget(m_sb);
    layout->addWidget(m_dial);
    layout->addWidget(m_slider);
    layout->addWidget(m_pb);
    setLayout(layout);

    connect(m_sb, SIGNAL(valueChanged(int)),
            m_dial,SLOT(setValue(int)));
```

```
    connect(m_sb, SIGNAL(valueChanged(int)),
            m_slider, SLOT(setValue(int)));
    connect(m_sb, SIGNAL(valueChanged(int)),
            m_pb, SLOT(setValue(int)));

    connect(m_dial, SIGNAL(valueChanged(int)),
            m_sb, SLOT(setValue(int)));
    connect(m_dial, SIGNAL(valueChanged(int)),
            m_slider, SLOT(setValue(int)));
    connect(m_dial, SIGNAL(valueChanged(int)),
            m_pb, SLOT(setValue(int)));

    connect(m_slider, SIGNAL(valueChanged(int)),
            m_sb, SLOT(setValue(int)));
    connect(m_slider, SIGNAL(valueChanged(int)),
            m_dial, SLOT(setValue(int)));
    connect(m_slider, SIGNAL(valueChanged(int)),
            m_pb, SLOT(setValue(int)));
}
```

Листинг 5.9. Файл main.cpp

```
#include <QApplication>
#include "mydialog.h"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    MyDialog dialog;
    Dialog.show();
    return app.exec();
}
```

5.2. События

Когда в приложении что-то происходит, например, пользователь нажимает клавишу на клавиатуре, то мы можем сгенерировать соответствующий сигнал. Но как обработчик события узнает всю необходимую информацию о нем? Для этого используются объекты-события, наследники класса `QEvent`:

- `QCloseEvent` — при закрытии окна;
- `QResizeEvent` — при изменении размеров виджета;
- `QTimerEvent` — при срабатывании таймера;

- `QMouseEvent` — при движении манипулятора "мышь" и/или нажатии ее кнопок;
- `QInputEvent` — события, связанные с пользовательским вводом;
- `QKeyEvent` — при нажатии клавиши на клавиатуре;

и многие другие.

Чтобы узнать тип события, имеется метод `QEvent::type()`. Событие можно игнорировать, вызвав `QEvent::ignore()`, или разрешить для дальнейшей обработки с помощью метода `QEvent::accept()`.

5.3. Обработка событий с помощью виртуальных методов

> Хороши или плохи события жизни, во многом зависит от того, как мы их воспринимаем.
>
> *Мишель де Монтень*

Кроме механизма сигналов и слотов, в Qt применяется также старый метод виртуальных функций. Так, например, базовый класс `QWidget` (элемент интерфейса, *виджет*) и все его многочисленные потомки имеют виртуальный метод `mouseMoveEvent()`:

```
void QWidget::mouseMoveEvent ( QMouseEvent * event ),
```

который вызывается при перемещении указателя мыши над данным элементом. Если требуется определить какую-либо реакцию на это событие, то достаточно в собственном классе переопределить метод `mouseMoveEvent()`. Текущие координаты мыши в системе отсчета данного виджета можно узнать с помощью методов `QMouseEvent::x()` и `QMouseEvent::y()`. Следует иметь в виду, что по умолчанию метод `mouseMoveEvent()` вызывается, только если при перемещении указателя мыши удерживается любая из ее кнопок. Чтобы это происходило независимо от состояния кнопок, нужно для данного виджета вызвать метод `setMouseTracking(true)`.

Аналогично можно обрабатывать событие изменения размеров любого элемента, только в этом случае для него придется переопределить метод `resizeEvent()`.

ЗАДАЧА. Разработать на основе класса `QCheckBox` новый элемент с тремя состояниями, который управлялся бы всеми тремя кнопками мыши. Левая кнопка мыши должна устанавливать галочку (независимо от предыдущего состояния), правая — сбрасывать, средняя — устанавливать третье состояние (затемненная галочка).

Решение. Обратившись к документации, поставляемой вместе с библиотекой Qt, с помощью индекса находим описание класса `QCheckBox`. В разделе *List of all members, including inherited members* (Список всех членов класса, включая унаследованные) находим, что все наследники класса `QWidget` имеют виртуальный метод `mousePressEvent()`, который вызывается при нажатии кнопки мыши в то время, когда ее указатель находится над данным элементом. Итак, наша задача — разработать новый класс на основе `QCheckBox` и определить в нем метод `mousePressEvent()`.

Параметр данного метода — объект типа `QMouseEvent`. Изучая описание класса `QMouseEvent`, находим, что с помощью метода `button()` можно узнать, какая именно клавиша мыши была нажата. В результате, следуя шаблону *разд. 3.8*, получаем простую программу (рис. 5.4, листинги 5.10–5.12).

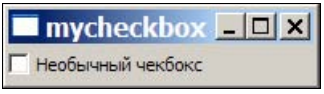


Рис. 5.4. Чтобы поставить галочку, нужно нажать левую кнопку мыши; с помощью правой кнопки галочка убирается

Листинг 5.10. Файл chap05/mycheckbox/mycheckbox.h

```
#ifndef _MYCHECKBOX_H_
#define _MYCHECKBOX_H_

#include <QtGui>

class MyCheckBox : public QCheckBox {
   Q_OBJECT
public:
   MyCheckBox(const QString& text, QWidget *parent=0);
protected:
   virtual void mousePressEvent(QMouseEvent *event);
};

#endif // _MYCHECKBOX_H_
```

Листинг 5.11. Файл chap05/mycheckbox/mycheckbox.cpp

```
#include "mycheckbox.h"

MyCheckBox::MyCheckBox( const QString& text,
                        QWidget *parent /* =0 */ )
```

```
19            : QCheckBox(text, parent) {
20     setTristate(true);
21  }
22
23  void MyCheckBox::mousePressEvent(QMouseEvent *event) {
24     switch (event->button()) {
25     case Qt::LeftButton:   // левая кнопка
26          setCheckState(Qt::Checked);
27          break;
28     case Qt::RightButton:  // правая кнопка
29          setCheckState(Qt::Unchecked);
30          break;
31     case Qt::MidButton:    // средняя кнопка
32          setCheckState(Qt::PartiallyChecked);
33     }
34     event->ignore();
35  }
```

Листинг 5.12. Файл chap05/mycheckbox/mycheckbox.cpp

```
36  #include "mycheckbox.h"
37
38  int main(int argc, char *argv[]) {
39      QApplication app(argc, argv);
40      QTextCodec *codec = QTextCodec::codecForName("CP1251");
41      QTextCodec::setCodecForTr(codec);
42
43      MyCheckBox chb(QObject::tr("Необычный чекбокс"));
44      chb.show();
45      return app.exec();
46  }
```

Пояснения к программе:

- (9, 17–21) — параметры конструктора (текст надписи и указатель на родительский элемент);
- (20) — наш переключатель должен иметь три возможных состояния;
- (24–33) — в зависимости от того, какая кнопка мыши нажата, либо устанавливаем галочку, либо сбрасываем ее, либо устанавливаем третье состояние (затемненная галочка);
- (34) — игнорируем данное событие, т. е. не передаем его родительскому элементу для дальнейшей обработки.

ЗАДАЧА. При закрытии окна приложения необходимо требовать у пользователя подтвердить выход из программы. При отрицательном ответе не закрывать окно.

Решение. Нужно переопределить виртуальный метод `closeEvent()`. Для этого изменим текст листинга 3.9 так, как показано в листинге 5.13.

Листинг 5.13

```
class MyWin : public QMainWindow {
  Q_OBJECT
public:
  MyWin(const QString& title);
protected:
  void closeEvent(QCloseEvent *event);
};
```

К тексту листинга 3.10 добавим реализацию метода `closeEvent()` (листинг 5.14).

Листинг 5.14

```
void MyWin::closeEvent(QCloseEvent *event){
   int r = QMessageBox::question(
       this, tr("Подтвердите"),
       tr("Выйти из программы?"),
       QMessageBox::Yes | QMessageBox::No,
       QMessageBox::Yes);
   if (r == QMessageBox::Yes)
      event->accept();
   else
      event->ignore();
}
```

Замечание

Вывод на экран текстовых сообщений обеспечивают статические методы класса `QMessageBox`: функции `information()`, `question()`, `critical()` и `warning()`, что в переводе с английского означает соответственно: "информация", "вопрос", "критическая ошибка" и "предупреждение".

Первый параметр функции `question()` — указатель на родительский элемент (в нашем случае — на главное окно приложения); второй параметр — текст заголовка диалога; третий — текст вопроса; четвертый — набор флагов, определяющих состав кнопок диалога; последний параметр — кнопка по умолчанию.

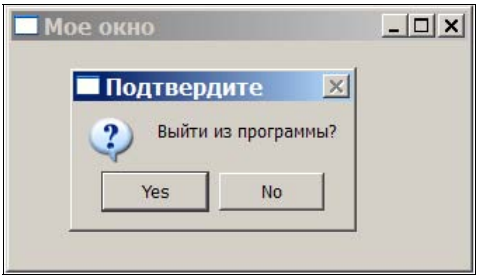


Рис. 5.5. Вопрос при закрытии окна

В результате при попытке закрыть главное окно приложения на экран будет выведен диалог с вопросом (рис. 5.5). Надписи на кнопках стандартных диалогов пока выводятся на английском. Чуть позже мы узнаем, как их русифицировать.

5.4. Фильтры для событий

Чтобы установить фильтр для событий, связанных с некоторым объектом `obj1`, для него нужно вызвать метод `QObject::installEventFilter(obj2)`. В качестве параметра следует задать указатель на тот объект, который должен получать сообщения обо всех событиях, происходящих с объектом `obj1`. Разумеется, объектом-приемником `obj2` может быть и сам исходный объект `obj1`. Обработка событий производится внутри виртуальной функции `QObject::eventFilter()` объекта-приемника `obj2`. Эта функция должна возвращать результат `true` или `false`. В первом случае дальнейшая обработка события будет прекращена, а во втором — событие будет передано для обработки исходному объекту `obj1`.

ЗАДАЧА. Запретить закрытие главного окна приложения только в том случае, если оно вызвано нажатием комбинации клавиш <Alt>+<F4>.

Решение. К сожалению, комбинация клавиш <Alt>+<F4> обрабатывается операционной системой и не может быть перехвачена в программе. Но можно отследить момент нажатия клавиши <Alt>. Если, например, в течение 0,5 с после этого был вызван виртуальный метод `closeEvent()`, то с большой долей уверенности можно считать, что была нажата комбинация клавиш <Alt>+<F4>. Текст программы иллюстрирует листинг 5.15.

Листинг 5.15. Файл chap05/altf4/altf4.cpp

```
01  #include <QtGui>
02
```

```
03 class MainWindow : public QMainWindow{
04     Q_OBJECT
05 public:
06     MainWindow() : QMainWindow(){
07        altPressed = false;
08        installEventFilter(this);
09     }
10     bool eventFilter(QObject * /* o */, QEvent *e) {
11         if (e->type() == QEvent::ShortcutOverride) {
12            QKeyEvent *event = (QKeyEvent *)e;
13
14            if (event->modifiers() == Qt::AltModifier) {
15               altPressed = true;
16               QTimer::singleShot(500, this, SLOT(testSlot()));
17               e->accept();
18               return true;
19            }
20            if (event->key() == Qt::Key_F4){
21               qDebug() << "F4 is pressed";
22            }
23        }
24        return QMainWindow::event(e);
25     }
26     void closeEvent(QCloseEvent *event){
27        if ( altPressed ){
28            qDebug() << "Alt + F4 is ignored";
29            altPressed = false;
30            event->ignore();
31        }
32     }
33 public slots:
34     void testSlot() {
35         if (altPressed){
36            qDebug() << "Alt + F4 was not pressed";
37            altPressed = false;
38         }
39     }
40 private:
41     bool altPressed;
42 };
43
44 #include "altf4.moc"
45 int main(int argc, char **argv){
46     QApplication app(argc, argv);
47
```

```
48      MainWindow main;
49      main.show();
50      return app.exec();
51  }
```

Пояснения к программе:

- (8) — установили фильтр для событий;
- (10–25) — сам фильтр;
- (11) — если событие связано с клавиатурой;
- (14) — если нажата клавиша <Alt>;
- (16) — через 500 мс будет вызван слот `testSlot()`;
- (17) — событие не будет проигнорировано;
- (20–22) — простое нажатие клавиши <F4> без модификатора <Alt> будет обработано;
- (24) — обработать все остальные события обычным образом;
- (26–32) — если перед попыткой закрытия окна была нажата клавиша <Alt>, то запрещаем обработку события;
- (44) — файл *.moc подключают, когда не хотят выносить декларации классов в отдельный заголовочный файл.

Рассмотренный метод не отличается стопроцентной надежностью. Например, если не отпуская клавишу <Alt>, повторно нажать <F4>, то окно все-таки закроется. И наоборот, не увенчается успехом попытка закрыть приложение щелчком мыши по соответствующей кнопке в заголовке окна, если при этом нажата клавиша <Alt>.

ЗАДАЧА. Измерить период неактивности пользователя.

Решение. Можно использовать таймер `QTimer`, перезапуская его при каждом действии пользователя. Для отслеживания событий клавиатуры и мыши (нажатие кнопок, перемещение, вращение колесика мыши) лучше применить фильтр для событий. В листинге 5.16 приведен пример программы, которая выводит предупреждение, если пользователь не задействовал клавиатуру и мышь в течение последних пяти секунд. Для удобства в строке состояния отображаются часы (рис. 5.6).

Листинг 5.16. Файл chap05/idle/idle.cpp

```
#include <QtGui>

#define DELAY 5000 // мс
```

```
class MyWin : public QMainWindow {
    Q_OBJECT
public:
    MyWin() : QMainWindow(0) {
       setMouseTracking(true);

       clock = new QLabel(statusBar()); // часы в строке состояния

       clockTimer = new QTimer(this); // таймер для часов
       connect( clockTimer, SIGNAL(timeout()),
                this, SLOT(showTime()));
       clockTimer->start(100); // каждые 100 мс

       idleTimer = new QTimer(this); // таймер неактивности
       installEventFilter(this); // установили фильтр для событий
       connect( idleTimer, SIGNAL(timeout()),
                this, SLOT(myMessage()) ); // сообщение
       idleTimer->start(DELAY); // запустили таймер неактивности
    }

    bool eventFilter(QObject * /* o */, QEvent *e){
       switch(e->type()) {
          case QEvent::MouseMove:          // перемещение мыши
          case QEvent::MouseButtonPress :  // нажатие кнопки мыши
          case QEvent::MouseButtonRelease: // отпускание кнопки мыши
          case QEvent::Wheel:              // вращение колесика мыши
          case QEvent::KeyPress:           // нажатие клавиши
          case QEvent::KeyRelease:         // отпускание клавиши
             idleTimer->start(DELAY); // запустили таймер неактивности
             break;
          default:
             break;
       }
       return false; // продолжить обработку события
   }
public slots:
   void showTime() {
      QTime time = QTime::currentTime();
      QString text = time.toString("hh:mm:ss.zzz");
      clock->setText(text);
 }
   void myMessage() {
      QMessageBox::information(
         this, tr("Сообщение"),
```

```
            tr("Вы ничего не делали целых %1 секунд!")
              .arg(DELAY/1000) );
        idleTimer->start(DELAY);
    }
private:
    QTimer *idleTimer;  // таймер неактивности
    QTimer *clockTimer; // таймер для часов в строке состояния
    QLabel *clock;      // часы в строке состояния
};

#include "idle.moc"

int main(int argc, char **argv){
    QApplication app(argc, argv);
    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);

    MyWin w;
    w.show();
    return app.exec();
}
```

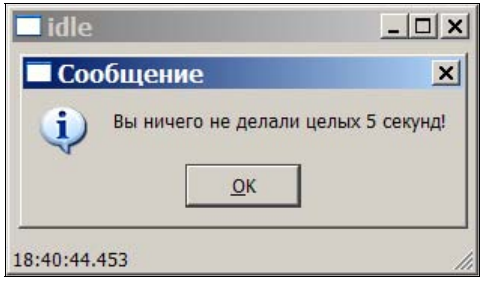


Рис. 5.6. Отслеживание длительного бездействия пользователя

Здесь мы поленились выносить объявление класса в отдельный заголовочный файл, поэтому пришлось явно подключить файл *.moc, генерируемый метакомпилятором Qt.

5.5. Сопоставление сигналов

Обычно мы связываем сигнал, генерируемый определенным объектом, с методом-слотом другого (или того же), также определенного, объекта. Поэтому сам сигнал нигде не хранит ссылку на объект, которым он сгенерирован. На-

пример, если у нас имеется группа кнопок, то мы для каждой кнопки должны связать сигнал `clicked()` с некоторым слотом-обработчиком нажатия, этот обработчик должен быть своим для каждой кнопки. Но что, если мы хотим обрабатывать нажатия нескольких кнопок в одной-единственной функции? Тогда нам нужны сведения о том, какая именно кнопка была нажата. Раз этой информации нет в самом сигнале, значит, нам потребуется дополнительный объект, который будет ее хранить. В библиотеке Qt для этого предусмотрен класс `QSignalMapper`, объекты которого перехватывают сигнал, определяют, каким объектом он сгенерирован, после чего генерируют аналогичный сигнал, но уже включающий информацию об объекте-источнике в виде целочисленного идентификатора, строки или указателя на виджет.

Чтобы добавить новую пару для сопоставления, необходимо вызвать метод `QSignalMapper::setMapping()`, задав в качестве первого параметра указатель на объект-источник сигнала, а в качестве второго — соответствующий ему целочисленный идентификатор, строку `QString`, указатель на некоторый объект `QObject` или виджет `QWidget`. После этого нужно связать любой сигнал этого объекта-источника со слотом `QSignalMapper::map()`. В результате данный слот при получении указанного сигнала будет генерировать новый сигнал `QSignalMapper::mapped()` с параметром типа `int`, `QString&`, `QObject*` или `QWidget*`.

ЗАДАЧА. Обрабатывать нажатие нескольких кнопок в одном слоте.

Решение. Предположим, мы храним текстовые надписи для каждой кнопки в виде списка `QStringList` (листинг 5.17).

Листинг 5.17. Класс "Группа кнопок"

```
class ButtonWidget : public QWidget {
    Q_OBJECT
public:
    ButtonWidget(QStringList texts, QWidget *parent = 0);
signals:
    void clicked(const QString& text);
private:
    QSignalMapper *signalMapper;
};
```

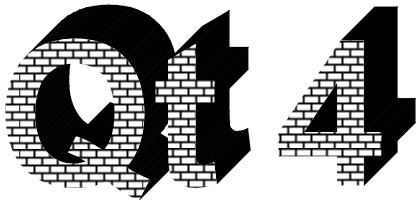
Требуется создать объект-сопоставитель `QSignalMapper`, нужное количество кнопок, сообщить сопоставителю о том, какая текстовая строка какой кнопке соответствует, перенаправить сигнал `clicked()` каждой кнопки на слот `map()` сопоставителя, а сигнал `mapped()` связать с каким-либо своим сигналом, в котором будет храниться текст нажатой кнопки (листинг 5.18).

Листинг 5.18. Реализация конструктора

```
ButtonWidget::ButtonWidget(QStringList texts, QWidget *parent)
            : QWidget(parent) {
    signalMapper = new QSignalMapper(this);
    QGridLayout *gridLayout = new QGridLayout;
    for (int i = 0; i < texts.size(); ++i) {
        QPushButton *button = new QPushButton(texts[i]);
        connect(button, SIGNAL(clicked()), signalMapper, SLOT(map()));
        signalMapper->setMapping(button, texts[i]);
        gridLayout->addWidget(button, i / 3, i % 3);
     }
     connect(signalMapper, SIGNAL(mapped(const QString&)),
             this, SIGNAL(clicked(const QString&)));
     setLayout(gridLayout);
}
```

В результате при нажатии любой кнопки будет генерироваться сигнал `clicked(const QString&)`. Его можно связать с каким-либо обработчиком, который, анализируя переданную текстовую строку (надпись на кнопке), выполнит нужные действия.

ГЛАВА 6



Кнопки и диалоги

В диалоге с жизнью важен не ее вопрос, а наш ответ.

Марина Цветаева

6.1. Кнопки нажатия *QPushButton*

Кнопки нажатия создаются с помощью конструктора класса `QPushButton`:

```
QPushButton( const QString& text,
             QWidget *parent = 0 )
```

или

```
QPushButton( const QIcon& icon,
             const QString& text,
             QWidget *parent = 0)
```

Здесь `text` — это текст надписи на кнопке; `icon` — ссылка на графическое изображение; `parent` — как обычно, указатель на родительский элемент. Знак `&`, поставленный перед любым символом в тексте надписи `text`, определяет клавишу-акселератор. Например, если указать текст надписи `В&ыход`, то буква `ы` будет подчеркнута и кнопка автоматически "сработает" при нажатии комбинации клавиш <Alt>+<ы>.

При нажатии на кнопку генерируется событие `clicked()`. Обработчики нажатия определяются с помощью метода `connect()`.

ЗАДАЧА. Разместить в окне приложения кнопки нажатия четырех разных типов: 1) обычную, 2) плоскую, 3) с пиктограммой и 4) с контекстным меню. Пункты меню должны позволять изменить стиль оформления интерфейса приложения (Windows, Motif, CDE и т. д.). Вид плоской кнопки должен изменяться при наведении на нее указателя мыши.

Решение. В листингах 6.1 и 6.2 приведен текст программы, демонстрирующий принцип работы с кнопками нажатия `QPushButton`, а на рис. 6.1 показан внешний вид окна приложения.

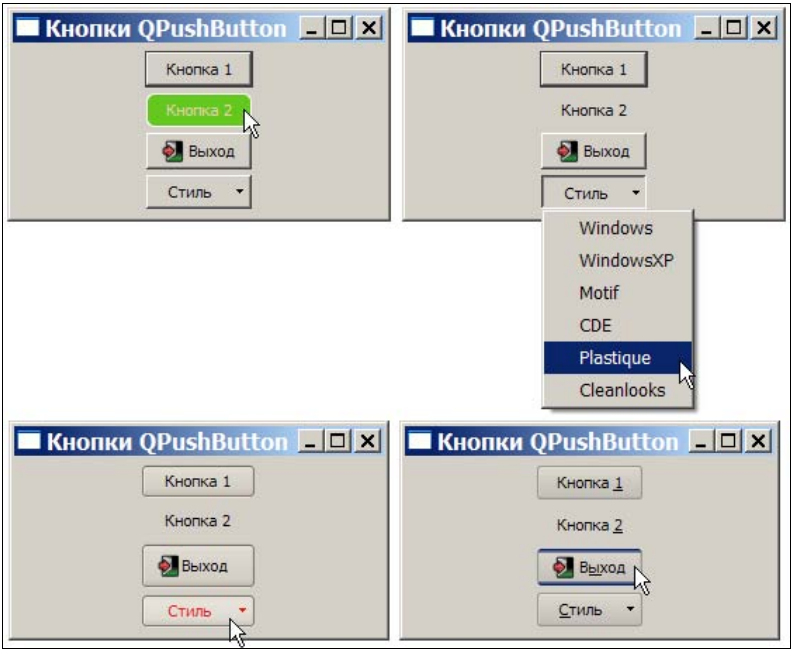


Рис. 6.1. Использование кнопок `QPushButton` и таблицы стилей

Листинг 6.1. Файл chap06/pushbuttons/pushbuttons.h

```
// Кнопки QPushButton

#include <QtGui>

class MyWin : public QMainWindow {
   Q_OBJECT
public:
    MyWin();
private slots:
    void btnClicked();
    void styleChanged(QAction *action);
};
```

Листинг 6.2. Файл chap06/pushbuttons/pushbuttons.cpp

```
14  #include <QtGui>
15
16  #include "pushbuttons.h"
17
18  MyWin::MyWin()
19       : QMainWindow() {
20
21      setWindowTitle(tr("Кнопки QPushButton"));
22      QWidget *cw = new QWidget(this);
23
24      QVBoxLayout *lay = new QVBoxLayout(this);
25      lay->setMargin(5);
26      lay->setSpacing(4);
27
28      // Кнопка с текстовой надписью:
29      QPushButton *b1 = new QPushButton(tr("Кнопка &1"));
30      lay->addWidget(b1, 1, Qt::AlignCenter);
31
32      connect(b1, SIGNAL(clicked()),
33              this, SLOT(btnClicked()) );
34
35      // Плоская кнопка:
36      QPushButton *b2 = new QPushButton(tr("Кнопка &2"), this);
37      b2->setFlat(true);
38      b2->setStyleSheet(
39          QString("QPushButton:hover:!pressed {color:%1; "
40                  "background-color: rgb(100,200,30); "
41                  "border-width: 1px; border-style: solid; "
42                  "border-radius: 6px; border-color: white;}")
43              .arg(QColor(240,210,170).name()));
44      lay->addWidget(b2, 1, Qt::AlignCenter);
45      connect(b2, SIGNAL(clicked()),
46              this, SLOT(btnClicked()) );
47
48      // Кнопка с пиктограммой:
49      QPushButton *b3 = new QPushButton(
50          QIcon(":/img/exit.png"), tr("В&ыход"), this);
51      lay->addWidget(b3, 1, Qt::AlignCenter);
52      connect(b3, SIGNAL(clicked()),
53              this, SLOT(close()) );
54
```

```
55      // Кнопка с всплывающим меню:
56      QActionGroup *ag = new QActionGroup(this);
57      ag->setExclusive(true);
58      QStringList styles = QStyleFactory::keys();
59      foreach ( QString s, styles ) {
60          QAction *a = ag->addAction(s);
61          a->setCheckable(true);
62      }
63      connect(
64          ag,
65          SIGNAL(triggered(QAction*)),
66          this,
67          SLOT(styleChanged(QAction*)) );
68
69      QMenu *menu = new QMenu(tr("Меню"), this);
70      menu->addActions(ag->actions());
71
72      QPushButton *b4 = new QPushButton(tr("&Стиль"), this);
73      b4->setStyleSheet(
74          QString("QPushButton:hover:!pressed{color:%1;}")
75              .arg(QColor(240,20,10).name()));
76      b4->setMenu(menu);
77      lay->addWidget(b4, 1, Qt::AlignCenter);
78
79      cw->setLayout(lay);
80      setCentralWidget(cw);
81  }
82
83  void MyWin::btnClicked() {
84      qApp->beep();
85      QMessageBox::information(
86          this, tr("Внимание"),
87          tr("Вы нажали кнопку!"),
88          QMessageBox::Ok);
89  }
90
91  void MyWin::styleChanged(QAction *action) {
92      qApp->setStyle(QStyleFactory::create(
93          action->actionGroup()->checkedAction()->text()));
94  }
95
96  //-------------------------------------------------
97  int main(int argc, char *argv[]) {
98
```

```
 99      QApplication app(argc, argv);
100
101      QTextCodec *codec = QTextCodec::codecForName("CP1251");
102      QTextCodec::setCodecForTr(codec);
103
104      MyWin mw;
105      mw.show();
106
107      return app.exec();
108  }
```

Пояснения к программе:

- (5–12) — класс окна на основе `QMainWindow`;
- (8) — конструктор окна;
- (10) — слот `btnClicked()` — обработчик нажатия на кнопку;
- (11) — слот `styleChanged()` — обработчик для изменения стиля интерфейса;
- (21) — установили заголовок окна приложения;
- (22) — создали виджет, который будет центральным (79, 80) для главного окна;
- (24–26) — создали менеджер размещения и установили для него поля и расстояние между элементами;
- (29, 30) — создали обычную кнопку, разместили ее с центрированием по горизонтали;
- (32) — задали обработчик нажатия на кнопку;
- (36, 37) — создали плоскую кнопку;
- (38–43) — установили стиль (цвет текста и фона, форму, толщину и цвет границы) для тех случаев, когда указатель мыши находится над кнопкой (состояние `hover`), но сама она не нажата (состояние `!pressed`). Здесь мы в демонстрационных целях смешиваем разные способы указания цвета;
- (44–46) — разместили кнопку с горизонтальным выравниванием по центру и задали обработчик нажатия (тот же, что и для предыдущей кнопки);
- (49–53) — создали кнопку **Выход** с пиктограммой, разместили ее с выравниванием по центру и указали для нее обработчик нажатия (завершение работы приложения);
- (56, 57) — создали группу взаимоисключающих действий `QActionGroup`;
- (58–61) — получили список названий всех доступных на текущей платформе стилей и добавили их в группу действий с пометкой `checkable`. Это

значит, что соответствующие пункты меню можно будет пометить галочками (благодаря установке `exclusive=true` пометка одного пункта автоматически сбрасывает другие);

- (63–67) — установили слот-обработчик для изменения стиля интерфейса;
- (69, 70) — создали контекстное меню и указали, что его пункты будут соответствовать действиям из группы `QActionGroup` (56);
- (72–77) — создали кнопку и связали ее с контекстным меню. Цвет надписи на кнопке будет изменяться при наведении на нее указателя мыши;
- (79, 80) — установили менеджер размещения (24) и центральный виджет (22);
- (83–89) — обработчик нажатия кнопок: выводим диалог с сообщением;
- (91–94) — обработчик выбора пункта стиля: задали стиль, имя которого совпадает с названием выбранного пункта меню;
- (97–108) — главная программа: создали объект-приложение, кодек для правильного отображения русских символов, главное окно и запустили цикл обработки событий.

Замечание

Чтобы не поставлять конечному пользователю вместе с программой графический файл `exit.png` (50), в котором хранится изображение пиктограммы для кнопки **Выход**, необходимо перед компиляцией программы создать в каталоге проекта текстовый файл `ИмяФайла.qrc`:

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
   <file>img/exit.png</file>
</qresource>
</RCC>
```

и включить в файл описания проекта *.pro строку

```
RESOURCES += ИмяФайла.qrc
```

Замечание

Если изображение на кнопке совпадает с какой-либо стандартной пиктограммой, то вместо указания имени графического файла можно использовать метод `QStyle::standardIcon()`. Тогда в строке (50) листинга 6.2 вместо

```
QIcon(":/img/exit.png")
```

следует, например, писать:

```
style()->standardIcon(QStyle::SP_FileDialogBack)
```

Список констант для стандартных изображений можно найти в документации (раздел `QStyle` *Class Reference*).

6.2. Программная эмуляция нажатия кнопок

ЗАДАЧА. Требуется "нажать" кнопку программно, без участия пользователя.

Решение. Можно использовать метод `QAbstractButton::click()` или `QAbstractButton::animateClick(ms)`. Во втором случае указывают задержку в миллисекундах. Текст программы приведен в листингах 6.3 и 6.4, внешний вид окна приложения показан на рис. 6.2.

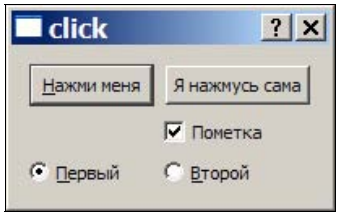


Рис. 6.2. Программная эмуляция нажатия кнопок

Листинг 6.3. Файл chap06/click/click.h

```
#include <QtGui>

class MyDialog : public QDialog {
   Q_OBJECT
public:
   MyDialog();

   QPushButton  *m_btn1;
   QPushButton  *m_btn2;
   QCheckBox    *m_chb;
   QRadioButton *m_rb1;
   QRadioButton *m_rb2;

public slots:
   void onClick();
};
```

Листинг 6.4. Файл chap06/click/click.cpp

```
17  #include "click.h"
18
19  MyDialog::MyDialog()
20          : QDialog() {
21
22      QGridLayout *layout = new QGridLayout(this);
23
24      m_btn1 = new QPushButton(tr("&Нажми меня"), this);
25      connect(m_btn1, SIGNAL(clicked()), this, SLOT(onClick()));
26      layout->addWidget(m_btn1, 0, 0);
27
28      m_btn2 = new QPushButton(tr("Я нажмусь сама"), this);
29      layout->addWidget(m_btn2, 0, 1);
30
31      m_chb = new QCheckBox(tr("Пометка"), this);
32      m_chb->setCheckState(Qt::Unchecked);
33      layout->addWidget(m_chb, 1, 1);
34
35      m_rb1 = new QRadioButton(tr("&Первый"), this);
36      layout->addWidget(m_rb1, 2, 0);
37
38      m_rb2 = new QRadioButton(tr("&Второй"), this);
39      m_rb2->setChecked(true);
40      layout->addWidget(m_rb2, 2, 1);
41
42      setLayout(layout);
43  }
44
45  void MyDialog::onClick() {
46      m_btn2->animateClick(200);
47      m_chb->animateClick(300);
48      if (!m_chb->isChecked())
49          m_rb1->animateClick(400);
50      else
51          m_rb2->animateClick(400);
52  }
53
54  //=============================================
55  int main(int argc, char *argv[]) {
56      QApplication app(argc, argv);
57      QTextCodec *codec = QTextCodec::codecForName("CP1251");
```

```
58      QTextCodec::setCodecForTr(codec);
59
60      MyDialog dlg;
61      dlg.show();
62      return app.exec();
63  }
```

6.3. Группы переключателей

ЗАДАЧА. Необходимо разместить на форме две группы зависимых переключателей (иногда их называют радиокнопками), по три в каждой группе.

Решение. Для каждой группы потребуется создать собственный менеджер размещения. Текст программы приведен в листингах 6.5 и 6.6, внешний вид формы показан на рис. 6.3.

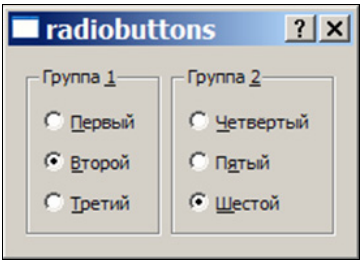


Рис. 6.3. Две независимые группы переключателей

Листинг 6.5. Файл chap06/rdbrn/rbtn.h

```
#include <QtGui>

class MyDialog : public QDialog {
    Q_OBJECT
public:
    MyDialog();

    QRadioButton  *m_rb11;
    QRadioButton  *m_rb12;
    QRadioButton  *m_rb13;
    QRadioButton  *m_rb21;
    QRadioButton  *m_rb22;
    QRadioButton  *m_rb23;
};
```

Листинг 6.6. Файл chap06/rdbrn/rbtn.cpp

```
#include "rbtn.h"

MyDialog::MyDialog() : QDialog() {
    QGroupBox *group1 = new QGroupBox(tr("Группа &1"));
    m_rb11 = new QRadioButton(tr("&Первый"), group1);
    m_rb12 = new QRadioButton(tr("&Второй"), group1);
    m_rb13 = new QRadioButton(tr("&Третий"), group1);
    m_rb12->setChecked(true);

    QVBoxLayout *layout1 = new QVBoxLayout(group1);
    layout1->addWidget(m_rb11);
    layout1->addWidget(m_rb12);
    layout1->addWidget(m_rb13);
    group1->setLayout(layout1);

    QGroupBox *group2 = new QGroupBox(tr("Группа &2"));
    m_rb21 = new QRadioButton(tr("&Четвертый"), group2);
    m_rb22 = new QRadioButton(tr("П&ятый"), group2);
    m_rb23 = new QRadioButton(tr("&Шестой"), group2);
    m_rb23->setChecked(true);

    QVBoxLayout *layout2 = new QVBoxLayout(group2);
    layout2->addWidget(m_rb21);
    layout2->addWidget(m_rb22);
    layout2->addWidget(m_rb23);
    group2->setLayout(layout2);

    QHBoxLayout *layout = new QHBoxLayout(this);
    layout->addWidget(group1);
    layout->addWidget(group2);
    setLayout(layout);
}
//=============================================
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);

    MyDialog dlg;
    dlg.show();
    return app.exec();
}
```

6.4. Группа кнопок *QButtonGroup*

Для логической организации групп кнопок (но не для размещения их на форме!) предназначен класс `QButtonGroup`.

ЗАДАЧА. Разместить на форме группу из трех зависимых и группу из трех независимых переключателей.

Решение. Иногда вместо создания отдельных кнопок диалогов (обычные кнопки нажатия `QPushButton`) удобнее реализовать группу кнопок (класс `QButtonGroup`):

```
QButtonGroup *btns = new QButtonGroup(parent);
```

После создания группы любые кнопки к ней добавляются с помощью метода

```
QButtonGroup::addButton ( QAbstractButton *button, int id )
```

Идентификатор кнопки `id` можно не указывать.

Текст программы приведен в листингах 6.7 и 6.8, внешний вид формы показан на рис. 6.4.

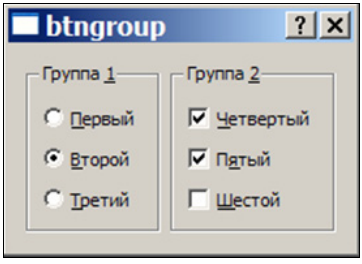


Рис. 6.4. Группа зависимых и независимых переключателей

Листинг 6.7. Файл chap06/btngr/btngr.h

```
#include <QtGui>

class MyDialog : public QDialog {
    Q_OBJECT
public:
    MyDialog();

    QButtonGroup  *m_gr1;
    QButtonGroup  *m_gr2;
};
```

Листинг 6.8. Файл chap06/btngr/btngr.cpp

```
#include "btngr.h"

MyDialog::MyDialog() : QDialog() {

    QGroupBox *group1 = new QGroupBox(tr("Группа &1"));
    QRadioButton *b11 = new QRadioButton(tr("&Первый"));
    QRadioButton *b12 = new QRadioButton(tr("&Второй"));
    QRadioButton *b13 = new QRadioButton(tr("&Третий"));

    m_gr1 = new QButtonGroup;
    m_gr1->setExclusive(true);
    m_gr1->addButton(b11, 1);
    m_gr1->addButton(b12, 2);
    m_gr1->addButton(b13, 3);
    m_gr1->button(2)->setChecked(true);

    QGroupBox *group2 = new QGroupBox(tr("Группа &2"));
    QCheckBox *b21 = new QCheckBox(tr("&Четвертый"));
    QCheckBox *b22 = new QCheckBox(tr("П&ятый"));
    QCheckBox *b23 = new QCheckBox(tr("&Шестой"));

    m_gr2 = new QButtonGroup;
    m_gr2->setExclusive(false);
    m_gr2->addButton(b21, 1);
    m_gr2->addButton(b22, 2);
    m_gr2->addButton(b23, 3);
    m_gr2->button(1)->setChecked(true);
    m_gr2->button(2)->setChecked(true);

    QVBoxLayout *layout1 = new QVBoxLayout;
    layout1->addWidget(b11);
    layout1->addWidget(b12);
    layout1->addWidget(b13);
    group1->setLayout(layout1);

    QVBoxLayout *layout2 = new QVBoxLayout;
    layout2->addWidget(b21);
    layout2->addWidget(b22);
    layout2->addWidget(b23);
    group2->setLayout(layout2);
```

```
    QHBoxLayout *layout = new QHBoxLayout;
    layout->addWidget(group1);
    layout->addWidget(group2);
    setLayout(layout);
}
//=============================================
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);

    MyDialog dlg;
    dlg.show();
    return app.exec();
}
```

6.5. Кнопки диалогов

В библиотеке Qt имеется класс `QDialogButtonBox`, который часто используют для размещения в окне диалога стандартных кнопок **Ok**, **Отмена** (Cancel), **Открыть** (Open), **Найти** (Find) и т. д. При создании бокса для кнопок нужно указать ориентацию `Qt::Horizontal` или `Qt::Vertical` и, как обычно, родителя:

```
QDialogButtonBox ( Qt::Orientation orientation,
                   QWidget *parent = 0 )
```

Для добавления кнопок служат методы

```
QDialogButtonBox::addButton( QAbstractButton *button,
                             ButtonRole role )
QDialogButtonBox::addButton( const QString& text,
                             ButtonRole role )
QDialogButtonBox::addButton(StandardButton button)
```

Первый метод позволяет добавить к боксу произвольную кнопку, созданную ранее; второй — кнопку, имеющую произвольную надпись, но выполняющую стандартную функцию; третий — кнопку со стандартной надписью и ролью.

Здесь `role` — это роль, которую будет выполнять данная кнопка:

- `QDialogButtonBox::AcceptRole` — роль кнопки **Ok**;
- `QDialogButtonBox::RejectRole` — **Отмена** (Cancel);

- `QDialogButtonBox::YesRole` — **Да** (Yes);
- `QDialogButtonBox::NoRole` — **Нет** (No);
- `QDialogButtonBox::ApplyRole` — **Применить** (Apply)

и т. д.

Параметр `button` — предопределенная константа, определенная для стандартных кнопок диалогов:

- `QDialogButtonBox::Ok` — кнопка **Ok**;
- `QDialogButtonBox::Cancel` — **Отмена** (Cancel);
- `QDialogButtonBox::Yes` — **Да** (Yes);
- `QDialogButtonBox::No` — **Нет** (No)

и т. д.

Функции-обработчики нажатия на кнопки определяются с помощью уже описанного механизма сигналов и слотов.

6.6. Модальные диалоги

Модальным называется диалог, который "забирает" на себя фокус ввода, не позволяя переключиться на другие окна до тех пор, пока он не будет закрыт.

Для разработки своих диалогов обычно используют базовый класс `QDialog`. В немодальном режиме диалог выводится на экран, как обычно, с помощью метода `show()`. Для модального режима предназначен метод `exec()`.

Как правило, в модальном диалоге присутствуют по крайней мере две кнопки со стандартными ролями `AcceptRole` и `RejectRole`. Это или кнопки, добавленные в `QDialogButtonBox`, или просто любые кнопки `QPushButton`, сигнал `clicked()` которых связан со слотами диалога `accept()` и `reject()`. Первый слот прячет диалог и возвращает значение `QDialog::Accepted`. Второй — делает то же самое, но возвращает значение `QDialog::Rejected`. Имеется также слот `QDialog::done(результат)`, позволяющий вернуть в метод `QDialog::exec()` произвольное целочисленное значение.

Обычно при выполнении этих слотов, т. е. при закрытии диалога, не требуется его уничтожение, т. к. введенная в диалоге информация еще ждет своей обработки. Но с помощью вызова

```
QDialog::setAttribute(Qt::WA_DeleteOnClose)
```

можно заставить библиотеку Qt уничтожить диалог сразу после его закрытия. Разумеется, в этом случае экземпляр диалога должен быть создан с помощью оператора `new` (иначе возникнет ошибка времени выполнения).

ЗАДАЧА. Разработать диалог, в котором вводятся сведения о работнике: фамилия, имя и дата рождения. При нажатии кнопки **Ok** нужно закрывать диалог и выводить информацию о работнике в главное окно приложения. При нажатии кнопки **Отмена** следует закрывать диалог и ничего не выводить.

Решение. Текст программы приведен в листингах 6.9–6.11, а внешний вид диалога и главного окна приложения показан на рис. 6.5.

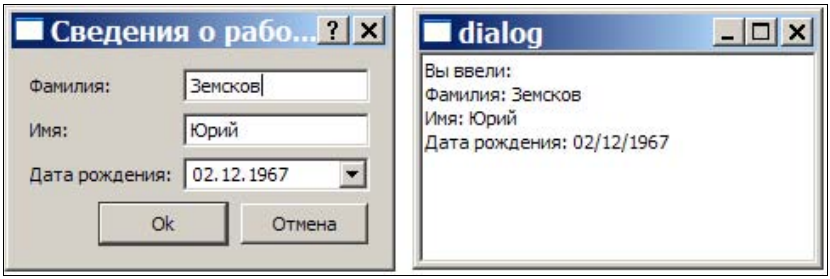


Рис. 6.5. Диалог для ввода данных и главное окно приложения

Листинг 6.9. Файл chap06/dialog/dialog.h

```
#ifndef _DIALOG_H_
#define _DIALOG_H_

#include <QtGui>

class Dialog : public QDialog {
   Q_OBJECT
public:
   Dialog(const QString& surname="",
          const QString& name="",
          QDate born=QDate());
   QString surname();
   QString name();
   QDate born();
private:
   QLineEdit *leSurname;  // Фамилия
   QLineEdit *leName;     // Имя
   QDateEdit *deBorn;     // Дата рождения
};
#endif // _DIALOG_H_
```

Листинг 6.10. Файл chap06/dialog/dialog.cpp

```
22  #include "dialog.h"
23  #include <QtGui>
24
25  Dialog::Dialog(const QString& surname,
26                 const QString& name,
27                 QDate born) {
28
29    QVBoxLayout* mainLayout = new QVBoxLayout(this);
30    QGridLayout* editLayout = new QGridLayout;
31    QDialogButtonBox * btnBox = new QDialogButtonBox(Qt::Horizontal);
32    mainLayout->addLayout(editLayout);
33    mainLayout->addStretch();
34    mainLayout->addWidget(btnBox);
35
36    QLabel *lbSurname = new QLabel(tr("Фамилия:"));
37    QLabel *lbName = new QLabel(tr("Имя:"));
38    QLabel *lbBorn = new QLabel(tr("Дата рождения:"));
39
40    leSurname = new QLineEdit;
41    leSurname->setText(surname);
42
43    leName = new QLineEdit;
44    leName->setText(name);
45
46    deBorn = new QDateEdit;
47    deBorn->setDate(born);
48    deBorn->setCalendarPopup(true);
49
50    editLayout->addWidget(lbSurname, 0, 0);
51    editLayout->addWidget(lbName, 1, 0);
52    editLayout->addWidget(lbBorn, 2, 0);
53
54    editLayout->addWidget(leSurname, 0, 1);
55    editLayout->addWidget(leName, 1, 1);
56    editLayout->addWidget(deBorn, 2, 1);
57
58    btnBox->addButton(tr("Ok"), QDialogButtonBox::AcceptRole);
59    btnBox->addButton(tr("Отмена"), QDialogButtonBox::RejectRole);
60
61    setWindowTitle(tr("Сведения о работнике"));
62
```

```
 63    connect(btnBox, SIGNAL(accepted()), this, SLOT(accept()));
 64    connect(btnBox, SIGNAL(rejected()), this, SLOT(reject()));
 65  }
 66
 67  QString Dialog::surname() {
 68    return leSurname->text();
 69  }
 70
 71  QString Dialog::name() {
 72    return leName->text();
 73  }
 74
 75  QDate Dialog::born() {
 76    return deBorn->date();
 77  }
```

Листинг 6.11. Файл chap06/dialog/main.cpp

```
 78  #include "dialog.h"
 79
 80  #define  tr  QObject::tr
 81
 82  int main(int argc, char *argv[]){
 83    QApplication app(argc, argv);
 84
 85    QTextCodec *codec = QTextCodec::codecForName("CP1251");
 86    QTextCodec::setCodecForTr(codec);
 87
 88    QTextEdit te;
 89    te.show();
 90
 91    Dialog d( "", "", QDate() );
    // или, например: Dialog d("Пупкин", "Вася", QDate(1992, 11, 27));
 92    if (d.exec() == QDialog::Accepted)
 93       te.append(
 94          tr("Вы ввели:\nФамилия: %1\nИмя: %2\nДата рождения: %3")
 95             .arg(d.surname())
 96             .arg(d.name())
 97             .arg(d.born().toString("dd/MM/yyyy")) );
 98    else
 99       te.append(
100          tr("Вы отказались от ввода данных") );
101
102    return app.exec();
103  }
```

6.7. Стандартные диалоги

В библиотеке Qt реализованы стандартные диалоги для открытия/сохранения файла/каталога `QFileDialog`, выбора принтера и параметров печати `QPrintDialog`, выбора шрифта `QFontDialog`, цвета `QColorDialog`, для отображения сообщений `QMessageBox`, вывода информации об ошибках `QErrorMessage`, отображения процента завершенности длительного процесса `QProgressDialog`, а также диалог для ввода строк, чисел или выбора элемента из списка `QInputDialog`.

ЗАДАЧА. Требуется добавить к форме стандартного диалога какой-либо дополнительный виджет.

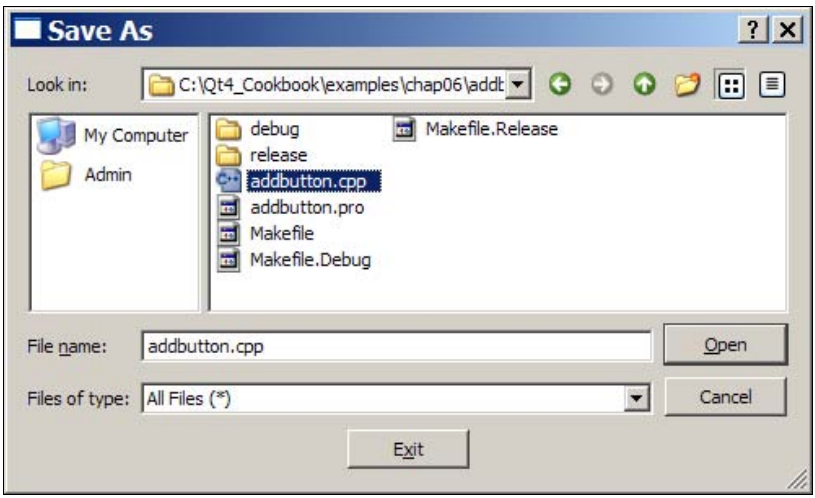


Рис. 6.6. Дополнительная кнопка у стандартного диалога выбора файла

Решение. Метод `layout()` позволяет получить доступ к менеджеру размещения стандартного диалога. Для примера добавим новую кнопку к диалогу выбора файла (рис. 6.6, листинг 6.12).

Листинг 6.12. Файл chap06/addbutton/addbutton.cpp

```
#include <QApplication>
#include <QFileDialog>
#include <QPushButton>
#include <QLayout>
int main(int argc, char **argv){
    QApplication app(argc, argv);
    QFileDialog dlg;
```

```
    QPushButton *button = new QPushButton(&dlg);
    button->setText(" E&xit ");
    QGridLayout *layout = (QGridLayout*)dlg.layout();
    layout->addWidget( button, layout->rowCount(), 0,
                       1, 3, Qt::AlignCenter);
    QObject::connect( button, SIGNAL(clicked()),
                      &app, SLOT(quit()));
    dlg.show();
    return app.exec();
}
```

При нажатии кнопки работа программы завершается.

ЗАДАЧА. В стандартном диалоге `QFileDialog` в режиме `fileMode=QFileDialog::DirectoryOnly` выбрать сразу несколько каталогов.

Решение. С помощью метода `findChild()` можно получить доступ к любому составному элементу диалога, в данном случае нам потребуется список `listView` (для режима `List View` — список) и дерево `treeView` (для режима `Detail View` — таблица). Для них обоих следует установить режим выделения `QAbstractItemView::MultiSelection`. Сказанное иллюстрируют листинг 6.13 и рис. 6.7.

Рис. 6.7. Выбор нескольких каталогов

Листинг 6.13. Файл chap06/selectdirs/selectdirs.cpp

```
#include <QtGui>

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

```
    QFileDialog d;
    d.setFileMode(QFileDialog::DirectoryOnly);
    QListView *v = d.findChild<QListView*>("listView");
    if (v)
       v->setSelectionMode(QAbstractItemView::MultiSelection);

    QTreeView *t = d.findChild<QTreeView*>();
    if (t)
       t->setSelectionMode(QAbstractItemView::MultiSelection);

    d.exec();
}
```

6.8. Окно внутри диалога

Допустим, мы оформили некоторый класс в виде главного окна приложения. А потом возникла необходимость обеспечить ту же функциональность, но внутри диалога `QDialog`.

ЗАДАЧА. Разместить "главное окно" `QMainWindow` внутри диалога.

Решение. Можно воспользоваться методами `QWidget::setParent()` или `QLayout::addWidget()`. Если мы установим для главного окна `QMainWindow` (или любого другого виджета) в качестве родителя диалог `QDialog` (или его менеджер размещения), то данный виджет автоматически разместится внутри этого диалога (рис. 6.8, листинг 6.14).

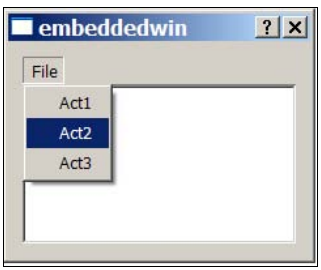


Рис. 6.8. Окно с панелью меню внутри диалога

Листинг 6.14. Файл chap06/embeddedwin/embeddedwin.cpp

```
#include <QtGui>

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

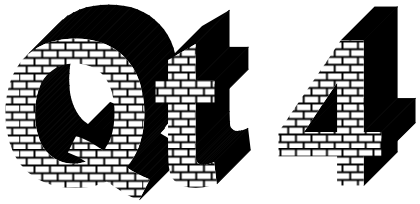
```
    // Главное окно:
    QMainWindow *w = new QMainWindow(0);

    // Виджет — текстовый редактор:
    QTextEdit *te = new QTextEdit(w);
    w->setCentralWidget(te);

    // Менеджер размещения для диалога:
    QVBoxLayout *layout = new QVBoxLayout(&d);
    // Вставляем главное окно внутрь диалога:
    layout->addWidget(w);

    // Меню главного окна:
    QMenu *menu = new QMenu("File", w);
    menu->addAction("Act1");
    menu->addAction("Act2");
    menu->addAction("Act3");
    w->menuBar()->addMenu(menu);

    return d.exec(); // Выводим диалог на экран
}
```

Если добавляемый к диалогу элемент всего один, как в нашем случае, то можно было обойтись без менеджера размещения, а просто заменить родителя для главного окна:

```
w->setParent(&d);
```

Но тогда при изменении размеров диалога размеры текстового редактора `QTextEdit` не изменялись бы.

ГЛАВА 7



Меню, панели инструментов и строка состояния

Званых на обед часто включают в меню.
Юрий Базылев

Разработчик может выбрать один из двух способов описания меню для своего приложения:

- создать в самой программе экземпляры классов `QAction` (действие) и `QMenu` (меню) и добавить новые пункты с помощью метода `QMenu::addAction()`;
- создать действия и пункты меню в диалоговом режиме с помощью программы Qt Designer (автономной или встроенной в интегрированную среду разработки Microsoft Visual Studio или KDevelop).

То же самое верно и для панелей инструментов.

Мы рассмотрим оба варианта по очереди, но сначала поговорим о строке состояния, поскольку именно в ней обычно отображается подсказка по текущему пункту меню или значку панели инструментов.

7.1. Строка состояния

Перегоревшая лампочка является примером того, как днем можно делать вид, что ты в отличном рабочем состоянии.
Алишер Файз

Строка состояния `QStatusBar` создается автоматически в нижней части главного окна приложения, если в программе хоть раз вызывается метод `MainWindow::statusBar()`. При наведении указателя мыши на кнопку панели инструментов или пункт меню в строке состояния на время появляется текст подсказки, если он определен для данной кнопки или пункта меню.

Указатель на экземпляр `QStatusBar` можно получить с помощью метода `QMainWindow::statusBar()`. Вывести в строке состояния произвольный текст (и затереть предыдущий) позволяет функция `showMessage()`:

```
void QStatusBar::showMessage(
    const QString &message,  // Выводимая строка
    int timeout = 0 )        // Кол-во миллисекунд
```

Если `timeout` > 0, то сообщение автоматически исчезает через указанный промежуток времени, а на его место возвращается текст предыдущей надписи.

По умолчанию строка состояния представляется в виде одной панели, располагаемой по всей ширине родительского окна. Но ее можно разбить по ширине на отдельные поля, если вставить в нее другие элементы, например, `QLabel`. Для этого предназначены методы `addWidget()`, `addPermanentWidget()` и `insertWidget()`, синтаксис которых иллюстрирует листинг 7.1.

Листинг 7.1. Методы строки состояния

```
void QStatusBar::addWidget (
    QWidget *widget,  // Вставляемый элемент
    int stretch = 0 ) // Коэффициент растяжимости

void QStatusBar::addPermanentWidget (
    QWidget *widget,
    int stretch = 0 )

int QStatusBar::insertWidget (
    int index,        // Позиция
    QWidget *widget,  // Вставляемый элемент
    int stretch = 0 ) // Коэффициент растяжимости
```

Элементы, добавляемые с помощью метода `addPermanentWidget()`, располагаются в правой части строки состояния и не затираются сообщениями, выводимыми с помощью `showMessage()`.

В нижней правой части строки состояния по умолчанию отображается специальный маркер, который можно "зацепить" указателем мыши для изменения размеров окна. Его показ можно запретить, вызвав метод

```
QStatusBar::setSizeGripEnabled(false)
```

При этом возможность изменять размеры окна по-прежнему остается.

7.2. Действия

> Человек должен иметь возможность действовать так, как он хочет, чтобы действовать так, как он должен.
>
> *Захария*

Пользователь может влиять на процесс выполнения программы с помощью указателя и кнопок мыши, выбирая нужный пункт меню или кнопку на панели инструментов. Кроме того, в его распоряжении имеются клавиши со стрелками, а также различные клавиатурные комбинации. Все эти события объединены в библиотеке Qt под общим названием *действие* (*action*).

При создании действия указывается родительский элемент и, при необходимости, пиктограмма и/или текстовая метка (листинг 7.2).

Листинг 7.2. Создание действия

```
QAction::QAction(QObject *parent)
QAction::QAction(const QString& text, QObject *parent)
QAction::QAction(const QIcon& icon,
                 const QString& text,
                 QObject *parent)
```

Метод `QAction::setStatusTip(текст)` определяет текст подсказки по данному действию, выводимую в строке состояния, а метод

```
QAction::setShortcut(QKeySequence& shortcut)
```

привязывает к действию некоторую комбинацию клавиш.

С каждым действием связывается функция-обработчик, для этого применяется все тот же метод `connect`, например:

```
QAction *exitAction = new QAction(tr("В&ыход"), this);
exitAction->setStatusTip(tr("Выход из программы"));
exitAction->setShortcut(tr("Ctrl+Q"));
connect(exitAction, SIGNAL(triggered()), this, SLOT(close()));
```

7.3. Меню

Горизонтальная панель меню `QMenuBar` создается автоматически, если мы обращаемся к ней для добавления хотя бы одного вертикального меню `QMenu`. Пример иллюстрирует листинг 7.3.

Листинг 7.3. Пример меню

```
// Определяем действия:
QAction *aboutAction = new QAction(tr("&О программе"), this);
QAction *exitAction = new QAction(tr("В&ыход"), this);

// Создаем вертикальное меню и добавляем
// его на автоматически созданную панель QMenuBar:
QMenu *fileMenu = menuBar()->addMenu(tr("&Файл"));
fileMenu->addAction(aboutAction); // Добавили действие 'О программе'
fileMenu->addSeparator();         // Разделитель пунктов меню
fileMenu->addAction(exitAction);  // Добавили действие 'Выход'
```

Здесь мы создали меню **Файл** с двумя пунктами, **О программе** и **Выход**, и горизонтальной полосой-разделителем между ними.

7.4. Виджеты в строке состояния

ЗАДАЧА. Разместить в строке состояния текстовую надпись (в которую будут выводиться подсказки по текущим пунктам меню), поле ввода и поле с раскрывающимся списком.

Решение. Текст программы приведен в листингах 7.4–7.6, а внешний вид окна приложения показан на рис. 7.1.

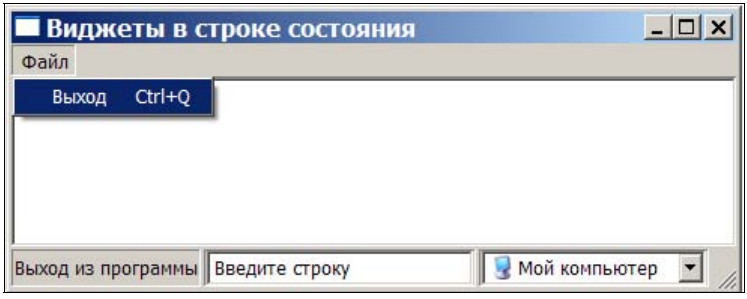


Рис. 7.1. Надпись `QLabel`, строка ввода `QLineEdit` и поле выбора `QComboBox` в строке состояния

Листинг 7.4. Файл chap07/statusbar/mainwin.h

```
1  #ifndef MAINWIN_H
2  #define MAINWIN_H
3
```

```
 4  #include <QtGui>
 5
 6  class MainWin : public QMainWindow {
 7     Q_OBJECT
 8  public:
 9     MainWin();
10  private:
11     QTextEdit *te;          // редактор текста
12     QStatusBar *sb;         // строка состояния
13     QLabel *label;          // надпись
14     QLineEdit *edit;        // поле ввода
15     QComboBox *combo;       // поле со списком
16     QAction *exitAction;    // действие "Выход"
17     QMenu *fileMenu;        // меню "Файл"
18  };
19
20  #endif
```

Листинг 7.5. Файл chap07/statusbar/mainwin.cpp

```
22  #include <QtGui>
23  #include "mainwin.h"
24
25  MainWin::MainWin(){
26     te = new QTextEdit(this);
27     setCentralWidget(te); // редактор текста — основной виджет окна
28
29     QStatusBar *sb = new QStatusBar(this);
30     label = new QLabel(sb); //(tr("Подсказка"), sb);
31     connect(sb, SIGNAL(messageChanged(const QString &)),
32             label, SLOT(setText(const QString &)) );
33
34     edit = new QLineEdit(tr("Введите строку"), sb);
35
36     combo = new QComboBox();
37     combo->addItem(
38        style()->standardIcon(QStyle::SP_ComputerIcon),
39        tr("Мой компьютер"));
40     combo->addItem(
41        style()->standardIcon(QStyle::SP_DriveNetIcon),
42        tr("Сетевое окружение"));
43     combo->setCurrentIndex(0);
44
```

```
45      sb->addPermanentWidget(label, 1);
46      sb->addPermanentWidget(edit, 2);
47      sb->addPermanentWidget(combo, 1);
48      setStatusBar(sb);
49      sb->showMessage(tr("Готов"), 3000);
50
51      exitAction = new QAction(tr("В&ыход"), this);
52      exitAction->setStatusTip(tr("Выход из программы"));
53      exitAction->setShortcut(tr("Ctrl+Q"));
54      connect(exitAction, SIGNAL(triggered()), this, SLOT(close()));
55
56      fileMenu = menuBar()->addMenu(tr("&Файл"));
57      fileMenu->addAction(exitAction);
58
59      setWindowTitle(QMainWindow::tr("Виджеты в строке состояния"));
60  }
```

Листинг 7.6. Файл chap07/statusbar/main.cpp

```
62  #include <QtGui>
63  #include <mainwin.h>
64
65  int main(int argc, char *argv[]) {
66      QApplication app(argc, argv);
67      QTextCodec *codec = QTextCodec::codecForName("CP1251");
68      QTextCodec::setCodecForTr(codec);
69
70      MainWin mw;
71      mw.resize(400, 300);
72      mw.show();
73
74      return app.exec();
75  }
```

7.5. Примеры приложений

ЗАДАЧА. Разработать главное окно приложения с горизонтальным меню, имеющим одну группу пунктов **Файл** и подпункты **О программе** и **Выход**. В строке состояния должна отображаться подсказка по текущему пункту меню, а также текущий размер окна и координаты положения курсора мыши.

Решение. В листингах 7.7 и 7.8 приведен текст программы, в которой иллюстрируются многие из описанных ранее элементов: меню, строка состояния

с несколькими текстовыми полями, обработка перемещения указателя мыши и изменения размеров окна с помощью виртуальных функций, а также определение собственного класса. Здесь мы пока не используем Qt Designer. На рис. 7.2 показано, как выглядит окно программы в системе Windows.

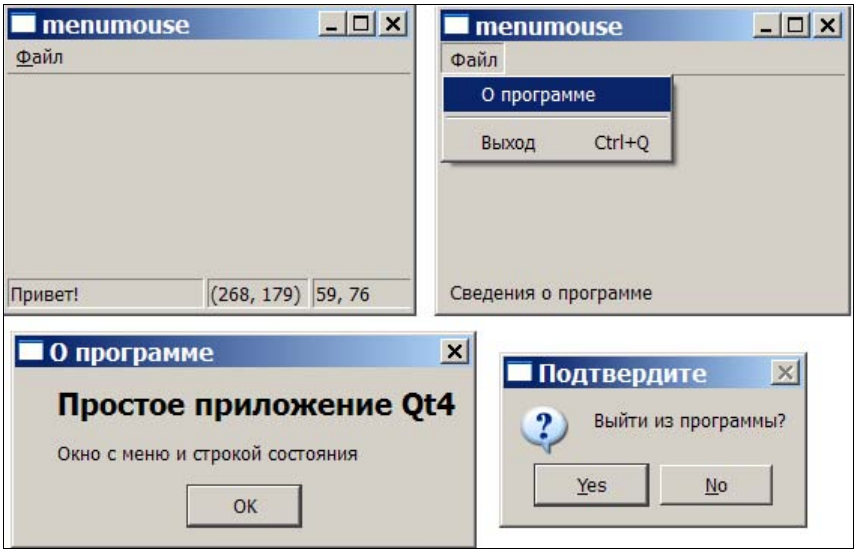


Рис. 7.2. Меню и строка состояния, в которой отображаются размеры окна и координаты указателя мыши

Листинг 7.7. Файл chap07/menumouse/mainwindow.h

```
#include <QtGui>

class MainWindow : public QMainWindow {
    Q_OBJECT

public:
    MainWindow();

protected:
    virtual void resizeEvent(QResizeEvent *event);
    virtual void mouseMoveEvent(QMouseEvent *event);
    virtual void closeEvent(QCloseEvent *event);

private slots:
    void about();

```

```
17  private:
18      QAction *aboutAction;
19      QAction *exitAction;
20      QMenu *fileMenu;
21      QLabel *sb1;
22      QLabel *sb2;
23      QLabel *sb3;
24
25      bool askOnClose();
26  };
```

Пояснения к программе:

- (1) — подключили заголовочный файл `QtGui`, в котором описаны все объекты, относящиеся к элементам графического интерфейса. Это проще, чем отдельно подключать `QMainWindow`, `QLabel`, `QMenu` и т. д.;
- (3) — объявили новый класс `MainWindow`, использовав в качестве базового класс `QMainWindow` (наследника `QObject`);
- (4) — не забыли про обязательный макрос `Q_OBJECT`;
- (7) — конструктор без параметров;
- (10–12) — виртуальные методы, вызываемые автоматически при изменении размеров окна, при перемещении указателя мыши и при закрытии окна;
- (14, 15) — описание слотов (функций-обработчиков). В данном случае здесь упомянута только функция, вызываемая при выборе пункта меню **О программе** (т. к. все остальные обработчики — это виртуальные функции или уже определенные методы базового класса);
- (17–23) — переменные-члены класса: два действия для пунктов меню, само меню и три текстовые метки для строки состояния;
- (25) — служебная функция, вызываемая при закрытии окна (просит подтверждения у пользователя).

Листинг 7.8. Файл chap07/menumouse/mainwindow.cpp

```
1 #include <QtGui>
2 #include "mainwindow.h"
3
4 MainWindow::MainWindow() {
5
6     QTextCodec *codec = QTextCodec::codecForName("CP1251");
7     QTextCodec::setCodecForTr(codec);
```

```
 8
 9     aboutAction = new QAction(tr("&О программе"), this);
10     aboutAction->setStatusTip(tr("Сведения о программе"));
11     connect(aboutAction, SIGNAL(triggered()), this, SLOT(about()));
12
13     exitAction = new QAction(tr("В&ыход"), this);
14     exitAction->setStatusTip(tr("Выход из программы"));
15     exitAction->setShortcut(tr("Ctrl+Q"));
16     connect(exitAction, SIGNAL(triggered()), this, SLOT(close()));
17
18     fileMenu = menuBar()->addMenu(tr("&Файл"));
19     fileMenu->addAction(aboutAction);
20     fileMenu->addSeparator();
21     fileMenu->addAction(exitAction);
22
23     sb1 = new QLabel(statusBar());
24     sb2 = new QLabel(statusBar());
25     sb3 = new QLabel(statusBar());
26
27     statusBar()->setSizeGripEnabled(false);
28     statusBar()->addWidget(sb1, 2);
29     statusBar()->addWidget(sb2, 1);
30     statusBar()->addWidget(sb3, 1);
31
32     sb1->setText(tr("Привет!"));
33
34     setMouseTracking(true);
35 }
36
37 void MainWindow::resizeEvent(QResizeEvent *event) {
38     QSize sz = event->size();
39     sb2->setText( QString( "(%1, %2)" )
40                   .arg( sz.width() )
41                   .arg( sz.height() ) );
42 }
43
44 void MainWindow::mouseMoveEvent(QMouseEvent *event) {
45     QPoint pos = event->pos();
46     sb3->setText(QString( "%1, %2" )
47          .arg( pos.x() ) .arg( pos.y() ) );
48 }
49
```

```
50 void MainWindow::about() {
51     QMessageBox::about(
52         this, tr("О программе"),
53         tr("<h2>Простое приложение Qt4</h2>"
54            "<p>Окно с меню и строкой состояния"));
55 }
56
57 bool MainWindow::askOnClose() {
58     int r = QMessageBox::question(
59         this, tr("Подтвердите"),
60         tr("Выйти из программы?"),
61         QMessageBox::Yes | QMessageBox::No,
62         QMessageBox::Yes);
64     return (r == QMessageBox::Yes);
65 }
66
67 void MainWindow::closeEvent(QCloseEvent *event) {
68     if (askOnClose()) {
69         event->accept();
70     } else {
71         event->ignore();
72     }
73 }
74 //-----------------------------------
75 int main(int argc, char *argv[]) {
76     QApplication app(argc, argv);
77     MainWindow mainWin;
78     mainWin.show();
79     return app.exec();
80 }
```

Пояснения к программе:

- (1, 2) — подключили модуль `QtGui` и свой заголовочный файл;
- (4–35) — конструктор главного окна;
- (6, 7) — определили кодек;
- (9–11) — создали действие "О программе", задали для него текст подсказки и связали с функцией `about()`, реализованной ниже;
- (13–16) — создали действие "Выход", задали для него текст подсказки и клавиатурное сокращение, связали его с функцией `close()`, выполняющей закрытие окна (эта функция определена в базовом классе);
- (18–21) — создали меню **Файл**, состоящее из двух пунктов, с разделителем между ними;

- (23–25) — создали три текстовые метки, в качестве родительского элемента указали строку состояния;
- (27–30) — убрали из строки состояния маркер для изменения размеров окна (иначе правая текстовая метка не доходит до правого края окна, что выглядит не слишком эстетично) и разместили в строке три текстовые метки. Указали для первой коэффициент растяжимости, в два раза больший, чем для остальных;
- (32) — вывели в первое поле строки состояния приветствие;
- (34) — указали, что метод `mouseMoveEvent()` будет вызываться при движении указателя мыши, даже если ее кнопки не нажаты;
- (38–42) — реализовали виртуальный метод, выполняемый при изменении размеров окна. Здесь мы узнаем новые размеры с помощью функции `QResizeEvent::size()` и выводим их во второе поле строки состояния. Для преобразования целых значений к типу `QString` используются метасимволы `%1` и `%2` в строке шаблона (41), а также функция `QString::arg()`, которая замещает их строковым представлением своего аргумента и возвращает результат типа `QString`;
- (44–48) — виртуальный метод, вызываемый при перемещении указателя мыши. Новые координаты указателя возвращает функция `QMouseeEvent::pos()`. Мы выводим их в третье поле строки состояния;
- (50–55) — функция `about()` выполняется при выборе пункта меню **Файл | О программе**. Для вывода на экран информации вызывается метод `about()`, определенный для класса `QMessageBox`. Обратите внимание, что для форматирования текста используется язык разметки HTML;
- (57–73) — при закрытии окна программы (по команде меню **Выход**, нажатием комбинации клавиш <Alt>+<F4> или с помощью крестика в верхнем правом углу окна) исполняется виртуальный метод `closeEvent` (67), в котором мы вызываем функцию `askOnClose` (57). Последняя выводит на экран запрос на подтверждение, используя метод `question` класса `QMessageBox`. Если пользователь отвечает утвердительно, то окно закрывается (69), иначе действие отменяется (71) и выполнение программы продолжается;
- (75) — для экономии места мы не стали размещать главную функцию в отдельном файле main.cpp, как это обычно принято.

В *главе 9* мы узнаем, как использовать программу Qt Designer для определения действий, создания меню и других элементов интерфейса.

ЗАДАЧА. Необходимо, чтобы один из пунктов панели меню располагался справа, отдельно от остальных (рис. 7.3).

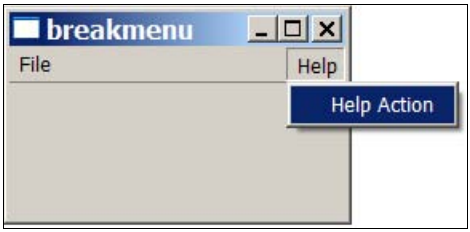


Рис. 7.3. Разрыв между пунктами меню

Решение. Добавить (или вставить в указанную позицию) разделитель между пунктами меню можно с помощью метода `QMenuBar::addSeparator()` или `QMenuBar::addSeparator()`. Но конкретный вид этого разделителя на экране зависит от текущего стиля. Например, применение стиля `Motif`

```
menuBar()->setStyle( new QMotifStyle() );
```

гарантирует, что пункты меню после разделителя будут выровнены по противоположному краю панели. Во всех остальных случаях придется разработать свой класс стиля, переопределив в нем метод `styleHint()`, как показано в листинге 7.9.

Листинг 7.9. Файл chap07/breakmenu/breakmenu.cpp

```
#include <QtGui>
class MyStyle : public QWindowsStyle {
public:
    MyStyle() {}
    int styleHint(StyleHint hint,
                  const QStyleOption *option = 0,
                  const QWidget *widget = 0,
                  QStyleHintReturn *returnData = 0) const {
        if (hint == QStyle::SH_DrawMenuBarSeparator)
            return QStyle::SH_DrawMenuBarSeparator;

        return QWindowsStyle::styleHint(
            hint, option, widget, returnData);
     }
};

class MainWindow : public QMainWindow{
public:
    MainWindow() {
        QMenu *menu1 = new QMenu("File", this);
        menu1->addAction("File Action");
```

```
        QMenu *menu2 = new QMenu("Help", this);
        menu2->addAction("Help Action");
        menuBar()->addMenu(menu1);
        menuBar()->addMenu(menu2);

        menuBar()->setStyle(new MyStyle());
        menuBar()->insertSeparator(menu2->menuAction());
    }
};

int main(int argc, char **argv){
    QApplication app(argc, argv);
    MainWindow w;
    w.show();
    return app.exec();
}
```

Здесь мы просто разрешаем отобразить разрыв, всю "черновую" работу берет на себя библиотека Qt.

ЗАДАЧА. Сначала создаются действия `QAction` (с пиктограммой или без), а затем они добавляются к панели меню или панели инструментов. А как сделать, чтобы одно и то же действие было представлено значком на панели инструментов, но не имело значка в меню (рис. 7.4)?

Решение. Можно создать действия без пиктограммы, а затем добавить нужное графическое изображение непосредственно к каждой кнопке панели инструментов (листинг 7.10).

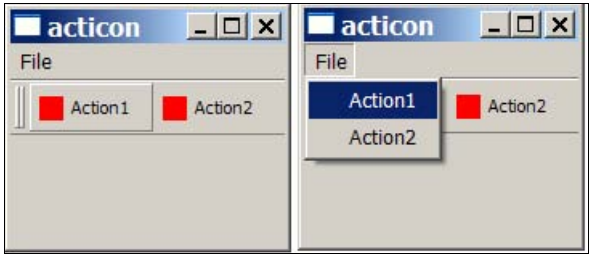


Рис. 7.4. Одно и то же действие со значком на панели инструментов и без значка в меню

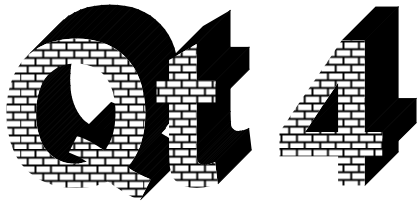
Листинг 7.10. Файл chap07/acticon/acticon.cpp

```
#include <QApplication>
#include <QtGui>
```

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

    QPixmap pix(15,15);
    pix.fill(Qt::red); // Просто красный квадрат вместо пиктограммы

    QMainWindow w; // Главное окно приложения

    QAction *action1 = new QAction("Action1", &w);
    QAction *action2 = new QAction("Action2", &w);

    QToolBar *toolBar = new QToolBar(&w); // Панель инструментов

    // Отображать текст рядом с пиктограммами:
    toolBar->setToolButtonStyle(Qt::ToolButtonTextBesideIcon);

    toolBar->addAction(action1);
    toolBar->addAction(action2);

    // Список кнопок панели инструментов:
    QList<QToolButton*> toolButtons =
        qFindChildren<QToolButton*>(toolBar);

    // Добавляем к каждой кнопке значок (красный квадрат):
    for (int i = 0; i < toolButtons.size(); i++) {
        QToolButton *button = (QToolButton*)toolButtons.at(i);
        button->setIcon(pix);
    }

    // Создаем меню:
    QMenu *menu = new QMenu("File", &w);
    menu->addAction(action1);
    menu->addAction(action2);

    w.menuBar()->addMenu(menu);
    w.addToolBar(toolBar);
    w.show();
    return app.exec();
}
```

Здесь для простоты в качестве значков на панели инструментов используются закрашенные квадратики. Позже мы подробнее остановимся на том, как работать с настоящими графическими изображениями.

ГЛАВА 8



Размещение элементов на форме

В здоровом теле здоровый дух размещается с большим трудом.

А. Подводный

8.1. "Ручное" размещение элементов формы

ЗАДАЧА. Необходимо корректировать размеры элементов диалога при изменении размеров всего окна.

Решение. С помощью метода `QWidget::setGeometry(int x, int y, int w, int h)` или `QWidget::setGeometry(const QRect&)` можно задать положение и размер любого визуального элемента в пикселах. Для установки размеров без изменения положения может использоваться метод `QWidget::resize(int w, int h)` или `QWidget::resize(const QSize&)`. Наоборот, для перемещения элемента в нужную позицию с сохранением прежних размеров служит метод `QWidget::move(int x, int y)` или `QWidget::move(const QPoint&)`.

Недостаток жесткого варианта размещения элементов интерфейса — пользователь не может изменить размер окна диалога (или изменение размеров окна не влияет на взаимное положение и размеры всех его элементов). В результате при низком разрешении монитора все выглядит слишком крупно, а то и вовсе не помещается на экран, при высоком — наоборот, слишком мелко. Кроме того, в различных операционных системах существуют разные шрифты, поэтому надписи и поля ввода, прекрасно смотревшиеся в одной системе, при переносе на другую платформу могут не уместиться в прежних границах. К тому же в солидных программных продуктах принято предоставлять пользователю возможность настраивать интерфейс программы по своему вкусу, в

частности, изменять гарнитуру и размер шрифта. А при локализации (переводе интерфейса программы на другой язык) все еще больше усложняется.

Очевидный (но не лучший) способ решить хотя бы некоторые из перечисленных проблем заключается в том, чтобы в виртуальном методе `QWidget::resizeEvent()`, автоматически выполняемом при изменении размеров окна, пересчитывать размеры и положение всех элементов. Пример программирования такого диалога приведен в листингах 8.1 и 8.2, а внешний вид окна — на рис. 8.1.

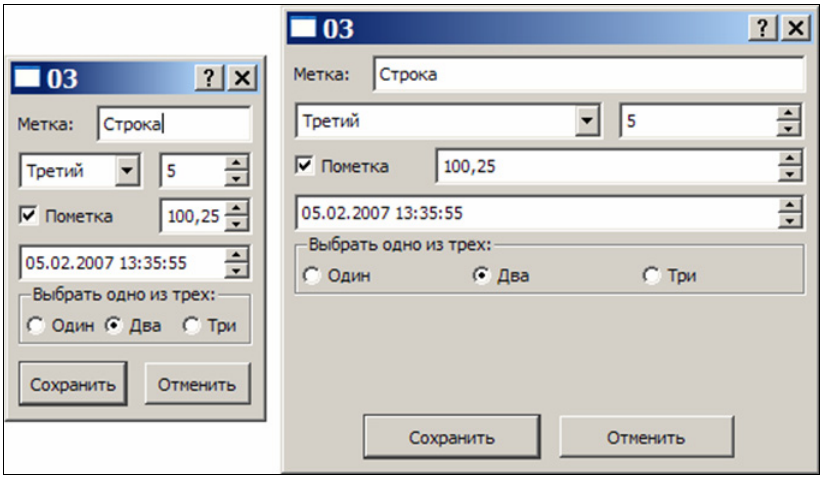


Рис. 8.1. Перерасчет геометрии в методе `resizeEvent()`

Листинг 8.1. Файл chap08/resizeevent/resizeevent.h

```
// Перерасчет геометрии всех элементов при изменении размеров окна

#include <QtGui>

class MyDialog : public QDialog {
    Q_OBJECT

public:
    MyDialog();

protected:
    virtual void resizeEvent(QResizeEvent *event);

```

```
14 private:
15     QLabel *lb;          // Текстовая метка.
16     QLineEdit *le;       // Строковое поле ввода.
17     QComboBox *cb;       // Поле ввода с раскрывающимся списком.
18     QSpinBox *sb;        // Целочисленное поле ввода с кнопками
19                          // инкремента/декремента.
20     QCheckBox *chb;      // Независимый переключатель
21                          // с двумя состояниями.
22     QDoubleSpinBox *dsb; // Поле ввода вещественного значения
23                          // с кнопками инкремента/декремента.
24     QDateTimeEdit *dte;  // Поле ввода даты и времени.
25     QGroupBox  *gb;   // Рамка с надписью вокруг группы элементов.
26     QRadioButton *rb1;   // Три
27     QRadioButton *rb2;   // зависимых
28     QRadioButton *rb3;   // переключателя.
29     QPushButton *btn1;   // Кнопка "Сохранить".
30     QPushButton *btn2;   // Кнопка "Отменить".
31 };
```

Пояснения к программе:

- (5) — объявили класс `MyDialog`, в качестве базового использовали стандартный класс диалога библиотеки Qt;
- (9) — конструктор диалога;
- (15–30) — указатели на все визуальные элементы необходимо сделать членами класса, чтобы иметь к ним доступ из метода `resizeEvent()`.

Листинг 8.2. Файл chap08/resizeevent/resizeevent.cpp

```
 1 // Перерасчет геометрии всех элементов при изменении размеров окна
 2
 3 #include "resizeevent.h"
 4
 5 MyDialog::MyDialog() {
 6
 7     QTextCodec *codec = QTextCodec::codecForName("CP1251");
 8     QTextCodec::setCodecForTr(codec);
 9
10     lb = new QLabel(tr("Метка:"), this);
11
12     le = new QLineEdit(tr("Строка"), this);
13
```

```
14      cb = new QComboBox(this);
15      cb->addItem(tr("Первый"));
16      cb->addItem(tr("Второй"));
17      cb->addItem(tr("Третий"));
18      cb->setCurrentIndex(2);
19      cb->setEditable(true);
20      cb->setInsertPolicy(QComboBox::InsertAtBottom);
21
22      sb = new QSpinBox(this);
23      sb->setValue(5);
24
25      chb = new QCheckBox(tr("Пометка"), this);
26      chb->setCheckState(Qt::Checked);
27
28      dsb = new QDoubleSpinBox(this);
29      dsb->setMaximum(200.0);
30      dsb->setDecimals(2);
31      dsb->setSingleStep(0.25);
32      dsb->setValue(100.25);
33
34      dte = new QDateTimeEdit(
35          QDateTime(QDate(2007, 2, 5), QTime(13, 35, 55, 10)),
36          this);
37
38      gb = new QGroupBox(tr("Выбрать одно из трех:"), this);
39
40      rb1 = new QRadioButton(tr("Один"), gb);
41
42      rb2 = new QRadioButton(tr("Два"), gb);
43
44      rb3 = new QRadioButton(tr("Три"), gb);
45
46      rb2->setChecked(true);
47
48      btn1 = new QPushButton(tr("Сохранить"), this);
49
50      btn2 = new QPushButton(tr("Отменить"), this);
51
52      setMinimumSize(160, 205);
53  }
54
55  void MyDialog::resizeEvent(QResizeEvent* /* event */ ) {
56      int dw = width() - minimumWidth();
```

```
57      int dh = height() - minimumHeight();
58
59      lb->setGeometry(5, 6, 45, 24);
60      le->setGeometry(55, 6, 100+dw, 24);
61
62      cb->setGeometry(5, 36, 80+dw*2/3, 24);
63      sb->setGeometry(95+dw*2/3, 36, 60+dw/3, 24);
64
65      chb->setGeometry(5, 65, 70, 24);
66      dsb->setGeometry(95, 65, 60+dw, 24);
67
68      dte->setGeometry(5, 95, 150+dw, 24);
69
70      gb->setGeometry(5, 120, 150+dw, 40);
71      rb1->setGeometry(5, 15, 45+dw/3, 24);
72      rb2->setGeometry(55+dw/3, 15, 45+dw/3, 24);
73      rb3->setGeometry(105+dw*2/3, 15, 45+dw/3, 24);
74
75      btn1->setGeometry(5+dw/4, 170+dh, 70+dw/4, 29);
76      btn2->setGeometry(85+dw/2, 170+dh, 70+dw/4, 29);
77  }
78
79  int main(int argc, char *argv[]) {
80
81      QApplication app(argc, argv);
82
83      MyDialog dlg;
84      dlg.show();
85
86      return app.exec();
87  }
```

Пояснения к программе:

- (7, 8) — кодек и установка кодировки;
- (10) — текстовая метка. Второй параметр конструктора (указатель на родительский элемент) можно опустить;
- (12) — однострочное поле ввода. То же самое замечание относительно параметра `this`;
- (14–20) — поле ввода с раскрывающимся списком. Текущим сделан второй, считая с нуля, т. е. последний элемент списка. Вызов (19) разрешает вводить в поле любые строковые значения, независимо от того, имеются

ли они в списке выбора. В (20) указано, что новые значения, введенные пользователем, будут автоматически дописываться в конец списка;

- (22, 23) — целочисленное поле ввода с кнопками инкремента/декремента;
- (25, 26) — независимый переключатель с двумя состояниями;
- (28–32) — поле для ввода вещественного значения с кнопками инкремента/декремента. В (29) установлено максимальное допустимое значение, в (30) — число десятичных цифр в дробной части, а в (31) — величина приращения при нажатии кнопки;
- (34–36) — поле для ввода даты и времени. Заданы год, месяц, число, часы, минуты, секунды и миллисекунды;
- (38) — рамка с надписью вокруг группы элементов;
- (40, 42, 44) — три зависимых переключателя с текстовыми метками;
- (46) — установлен второй переключатель, остальные автоматически выключены;
- (48, 50) — две кнопки;
- (52) — заданы минимально возможные размеры окна диалога. Если `setMinimumSize()` заменить на `setFixedSize()`, то пользователь не сможет изменить ни ширину, ни высоту окна;
- (55) — виртуальный метод, автоматически вызываемый при изменении размеров окна;
- (56, 57) — вычисление приращения ширины и высоты окна относительно их минимально разрешенных значений;
- (59) — положение и размеры текстовой метки фиксированы;
- (60) — положение однострочного поля ввода не изменяется, а ширина увеличивается на то же значение, что и ширина всего окна. В результате правый край элемента зафиксирован относительно правой границы окна;
- (62, 63) — ширина поля ввода со списком выбора увеличивается в два раза быстрее, чем ширина поля для ввода целого значения;
- (65) — позиция и размеры независимого переключателя фиксированы;
- (66, 68, 70) — ширина полей для ввода вещественного значения и для ввода даты/времени, а также ширина рамки вокруг группы элементов увеличивается на то же значение, что и ширина всего окна;
- (71–73) — ширина текстовых меток зависимых переключателей растет одинаково, на 1/3 от прироста ширины всего окна;
- (75, 76) — кнопки в нижней части диалога увеличиваются по ширине, а расстояние от них до нижней границы окна остается неизменным.

На рис. 8.2 показано, как изменяется внешний вид элементов диалога, если при запуске программы, текст которой приведен в листингах 8.1 и 8.2, указать в командной строке параметр `-style=ИмяСтиля`. Обратите внимание, что при одних и тех же размерах окна, устанавливаемых сразу после запуска программы, для стилей `WindowsXP`, `Motif`, `CDE` и `Cleanlooks` текстовая метка "**Один**" обрезана на последней букве, а в `Motif` высота всех полей ввода и ширина кнопок оказалась на грани критической: текстовые надписи еле "влезли" в отведенные для них границы из-за увеличенной толщины декоративных элементов. В системе Linux, где размер шрифта по умолчанию выбирается обычно больше, чем в Windows (из-за традиционно худшего качества отображения шрифтов), это различие компоновки элементов диалога для различных стилей еще более заметно.

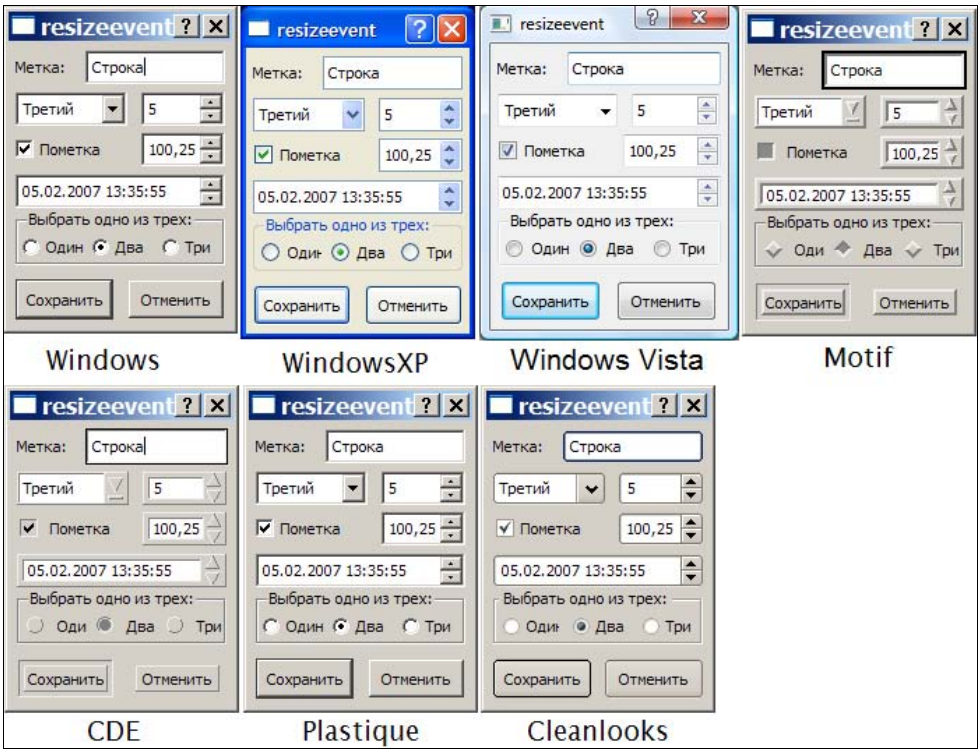


Рис. 8.2. Внешний вид окна и элементов управления в системе Windows при запуске программы с параметром `-style=Windows`, `WindowsXP`, `WindowsVista`, `Motif`, `CDE`, `Plastique` и `Cleanlooks`

8.2. Менеджеры размещения

ЗАДАЧА. Необходимо разместить в окне несколько элементов интерфейса и задать автоматический режим изменения их размеров при изменении размеров всего окна.

Решение. В Qt имеются классы `QHBoxLayout`, `QVBoxLayout` и `QGridLayout`, которые специально предназначены для управления положением и размерами элементов в окне. Первый позволяет располагать элементы друг за другом по горизонтали, второй — по вертикали, а третий размещает виджеты в ячейках воображаемой таблицы, причем каждый элемент может занимать несколько смежных ячеек по вертикали и/или горизонтали.

Менеджеры `QHBoxLayout` и `QVBoxLayout` используются одинаково:

- создаем экземпляр класса `QVBoxLayout` или `QHBoxLayout`

```
QHBoxLayout *layout = new QHBoxLayout(parent);
```

- добавляем к нему произвольное число элементов:

```
layout->addWidget(указательНаВиджет, коэффициент, выравнивание);
```

  Коэффициент (по умолчанию нулевой) задает степень растяжимости добавляемого элемента при изменении размеров всего окна. Последний необязательный параметр — комбинация битовых флагов, определяющих выравнивание по горизонтали: `Qt::AlignLeft`, `Qt::AlignRight`, `Qt::AlignHCenter` или `Qt::AlignJustify` и по вертикали: `Qt::AlignTop`, `Qt::AlignBottom` или `Qt::AlignVCenter`. Отцентрировать по вертикали и горизонтали одновременно позволяет константа `Qt::AlignCenter`.

  Вместо элемента управления можно добавить другой менеджер размещения со всеми своими элементами, для этого `addWidget()` заменяют на `addLayout()`;

- устанавливаем менеджер для данного окна

```
window->setLayout(layout);
```

Менеджер `QGridLayout` используется аналогично, только при вставке виджета в нужную ячейку с помощью метода `addWidget()` можно задать номер строки и столбца (нумерация начинается с нуля), а также, если требуется, число соседних строк и столбцов, которые будет занимать элемент (листинг 8.3).

Листинг 8.3

```
void QGridLayout::addWidget(
    QWidget *widget,                // Виджет
```

```
    int row, int column,            // Строка и столбец
    Qt::Alignment alignment = 0 )   // Выравнивание
```

или

```
void QGridLayout::addWidget(
    QWidget *widget,                // Виджет
    int fromRow, int fromColumn,    // Строка и столбец
    int rowSpan, int columnSpan,    // Число строк и столбцов
    Qt::Alignment alignment = 0 )   // Выравнивание
```

Растяжимостью строк и столбцов сетки управляют методы `setRowStretch()` и `setColumnStretch()`, первый параметр которых указывает номер строки (столбца), а второй — коэффициент растяжения. Чем он больше, тем сильнее будет растягиваться/сжиматься данная строка по вертикали (или столбец по горизонтали) по сравнению с остальными строками (столбцами) при изменении размеров всего окна.

В листинге 8.4 показаны строки, которые потребуется дописать к тексту конструктора диалога (см. листинг 8.2), если мы захотим использовать менеджеры размещения. Здесь все элементы располагаются в сетке `QGridLayout`, а для зависимых переключателей и кнопок создаются отдельные менеджеры `QHBoxLayout`. Разумеется, метод `resizeEvent()` теперь не нужен.

Листинг 8.4. Менеджеры размещения (файл chap08/autoresize/autoresize.cpp)

```
        // Создаем менеджер размещения:
52      QGridLayout *mainlay = new QGridLayout();

        // Размер полей вокруг сетки элементов и
        // интервалы между ячейками сетки:
53      mainlay->setMargin(2);
54      mainlay->setSpacing(3);
55
        // Размещаем элементы:
56      mainlay->addWidget(lb, 0, 0);
57      mainlay->addWidget(le, 0, 1, 1, 2);
58
59      mainlay->addWidget(cb, 1, 0, 1, 2);
60      mainlay->addWidget(sb, 1, 2);
61
62      mainlay->addWidget(chb, 2, 0);
63      mainlay->addWidget(dsb, 2, 1, 1, 2);
64
65      mainlay->addWidget(dte, 3, 0, 1, 3);
66
```

```
        // Менеджер размещения для переключателей:
67      QHBoxLayout *hbl = new QHBoxLayout();
68      hbl->addWidget(rb1, 1);
69      hbl->addWidget(rb2, 1);
70      hbl->addWidget(rb3, 1);
71      gb->setLayout(hbl);
72      mainlay->addWidget(gb, 4, 0, 1, 3);
73
        // Менеджер размещения для кнопок диалога:
74      QHBoxLayout *btns = new QHBoxLayout();
75      btns->addStretch(1);
76      btns->addWidget(btn1, 2);
77      btns->addWidget(btn2, 2);
78      btns->addStretch(1);
79      mainlay->addLayout(btns, 6, 0, 1, 3);
80
        // Растяжимость колонок и строк:
81      mainlay->setColumnStretch(1, 1);
82      mainlay->setColumnStretch(2, 1);
83      mainlay->setRowStretch(5, 1);

84      setLayout(mainlay);
85  }
```

В приведенном примере мы задействовали менеджер `QGridLayout`. Разумеется, можно предложить и другие варианты, например, на верхнем уровне использовать `QVBoxLayout` и заполнять его менеджерами `QHBoxLayout`, в которые вставлять элементы каждой горизонтальной строки диалога, подобно тому, как это сделано с переключателями в нашем примере.

Если нас не удовлетворяет то, как ведут себя какие-либо элементы диалога при изменении размеров окна, то можно попытаться вызвать для них метод `setSizePolicy()`, первый параметр которого задает политику изменения ширины элемента, а второй — его высоты. Каждый параметр может принимать одно из следующих значений:

- `QSizePolicy::Fixed` — размер элемента в данном направлении не изменяется и всегда равен значению, которое возвращает для него метод `sizeHint()`;
- `QSizePolicy::Minimum` — "идеальный" размер элемента, возвращаемый методом `sizeHint()`, считается его минимальным размером. Элемент может растягиваться, но не может сжиматься;
- `QSizePolicy::Maximum` — "идеальный" размер элемента считается его максимальным размером. Элемент может сжиматься, но не может растягиваться;

- `QSizePolicy::Preferred` — элемент старается поддерживать некоторый предпочтительный для него размер, возвращаемый методом `sizeHint()`, но при необходимости может растянуться или сжаться;
- `QSizePolicy::Expanding` — элемент старается принять максимально возможный доступный ему размер, но при необходимости может сжиматься;
- `QSizePolicy::MinimumExpanding` — элемент "старается" принять максимально возможный доступный ему размер, при этом его размер в данном направлении не может стать меньше того значения, что возвращает метод `sizeHint()`;
- `QSizePolicy::Ignored` — размер, возвращаемый методом `sizeHint()`, игнорируется. Элемент старается занять все доступное для него пространство.

ЗАДАЧА. При раскрытии списка `QComboBox` требуется, чтобы строки списка не обрезались по ширине (рис. 8.3).

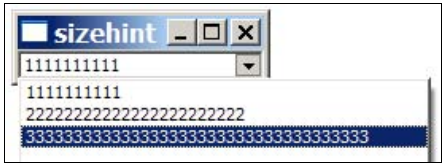


Рис. 8.3. Управление шириной раскрывающегося списка

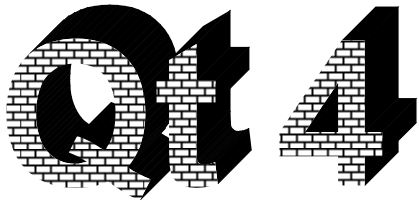
Решение задачи показано в листинге 8.5.

Листинг 8.5. Файл chap08/sizehint/sizehint.cpp

```
#include <QtGui>

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

    combo.view()->setSizePolicy( QSizePolicy::Minimum,
                                 QSizePolicy::Minimum);
    combo.addItem("1111111111");
    combo.addItem("22222222222222222222");
    combo.addItem("33333333333333333333333333333333");

    combo.show();
    return app.exec();
}
```

Метод `view()` позволяет получить раскрывающийся список, связанный с данным элементом `QComboBox`.

ГЛАВА 9



Автоматизация создания диалогов, отладка программ и предоставление помощи

> Математика имеет хороший инструмент. Экономика обладает хорошим материалом. Экономико-математические методы — это совмещение хорошего инструмента с хорошим исходным материалом. Уравнения умнее своих создателей.
>
> *Генрих Герц*

9.1. Создание диалога с помощью Qt Designer

В предыдущих примерах мы строили диалоговые окна в самой программе, вызывая для каждого элемента интерфейса конструктор соответствующего класса. Но есть и другой способ, более удобный для разработки сложных диалоговых форм. В состав библиотеки Qt входит утилита Qt Designer, с помощью которой можно создавать диалоги в интерактивном режиме, как это обычно делается в интегрированных средах, предоставляющих возможности быстрой разработки приложений (*RAD*, *Rapid Application Development*), например, в Delphi. Qt Designer используется либо как самостоятельное приложение, либо как компонент, встроенный в интегрированную среду Microsoft Visual Studio. Во втором случае доступ к его функциям производится с помощью пункта меню **Qt**.

ЗАДАЧА. Решить задачу из *разд. 8.2* с помощью Qt Designer.

Решение. Откроем Qt Designer и (если диалог создания новой формы не появится автоматически) выполним команду меню **File** | **New Form** (Файл | Новая форма). Из представленных вариантов заготовок (диалоги с горизонталь-

ным или вертикальным размещением кнопок, главное окно приложения или произвольный виджет) выберем первый и нажмем кнопку **Create** (Создать). В левом окне дизайнера **Widget Box** (Виджеты) в группе **Display Widget** (Виджеты для отображения информации) найдем элемент **Label** (рис. 9.1) и с помощью левой кнопки мыши перетащим его на форму. В правой части дизайнера, в окне **Property Editor** (Редактор свойств) укажем идентификатор элемента (значение свойства `objectName`), например `lb`, и текст надписи (свойство `text`).

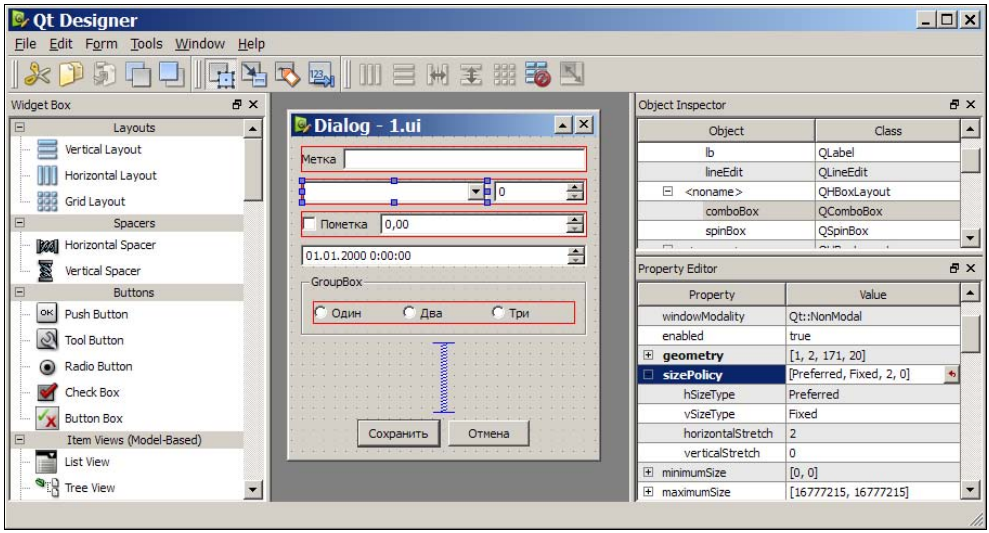


Рис. 9.1. Внешний вид окна Qt Designer

Аналогично разместим на форме однострочное поле ввода **Line Edit** (идентификатор `le`), поле со списком **Combo Box** (идентификатор `cb`), поле для ввода целого числа с кнопками инкремента/декремента **Spin Box** (идентификатор `sb`), независимый переключатель **Check Box** (идентификатор `chb`), поле для ввода вещественного числа с кнопками инкремента/декремента **Double Spin Box** (идентификатор `dsb`), поле для выбора даты/времени **Date/Time Edit** (идентификатор `dte`), контейнер **Group Box** (идентификатор `groupBox`) с тремя зависимыми переключателями **Radio Button** (идентификаторы `rb1`, `rb2` и `rb3`).

Пока не следует сильно заботиться о красивом размещении элементов друг относительно друга и об их точных размерах. Для первого переключателя зададим значение свойства `checked=true`. Между рамкой с зависимыми переключателями и кнопками диалога вставим элемент **Vertical Spacer** (вертикальный промежуток, он находится в группе **Spacers**). Для элемента **Button**

Box проверим значение свойства `standardButtons`: галочки должны быть установлены только для кнопок `Cancel` и `Save`.

Теперь выделим мышью первые два элемента (**Label** и **Line Edit**), находящиеся в первой горизонтальной строке диалога. Для выделения нескольких элементов можно использовать клавишу <Shift>. Щелкнем по выделенной группе правой кнопкой мыши и в появившемся контекстном меню выберем команду **Lay out** | **Lay Out Horizontally** (расположить по горизонтали). В результате автоматически будет создан менеджер размещения `QHBoxLayout` и оба выделенных виджета окажутся внутри него. Останется только задать единичное значение параметра `horizontalStretch` (растяжение по горизонтали) из раздела `sizePolicy` для однострочного поля ввода.

Аналогично создадим два горизонтальных менеджера размещения для двух следующих строк нашего диалога. Поле для ввода даты/времени пропустим, т. к. оно единственное в своей строке. Далее выделим три зависимых переключателя и создадим для них все тот же горизонтальный менеджер размещения. Для самой рамки **GroupBox**, внутри которой расположены три зависимых переключателя, также создадим свой `QHBoxLayout` (хотя эта рамка и единственная в своей строке, но она служит контейнером для других элементов и без менеджера размещения ее высота не может быть вычислена правильно).

Кнопки диалога трогать не будем, потому что при создании формы мастер уже разместил их в элементе **Button Box** — аналоге менеджера размещения специально для кнопок. Можно изменить состав кнопок с помощью свойства `standardButtons` и их выравнивание по горизонтали (свойство `centerButtons=true`).

Наконец, щелкнем правой кнопкой мыши по пустому фону диалога и в появившемся контекстном меню выберем команду **Lay Out Vertically** (расположить по вертикали). В результате будет создан вертикальный менеджер `QVBoxLayout` и внутри него будут размещены все имеющиеся на форме строки `QHBoxLayout` вместе с полем ввода даты/времени и вертикальным промежутком (для которых мы так и не создали горизонтального менеджера размещения).

Можно проверить, как будет выглядеть наш диалог при различном оформлении (команда меню **Form** | **Preview in**).

Теперь надо задать порядок обхода элементов при нажатии клавиши <Tab>. Для этого выполним команду меню **Edit** | **Edit Tab Order**. На редактируемой форме появятся квадратики с числами (рис. 9.2). Будем щелкать по ним левой кнопкой мыши в нужном порядке. После этого можно опять войти в режим предпросмотра **Preview in** и с помощью клавиши <Tab> убедиться, что порядок обхода элементов задан правильно.

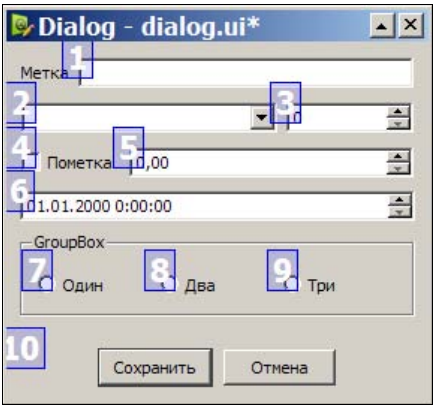


Рис. 9.2. Порядок обхода элементов

Допустим теперь, что нам требуется сделать недоступным поле ввода вещественного числа `dsb` при сбросе переключателя `chb`. Другими словами, нужно связать сигнал `toggled(bool)` элемента `chb` и слот `enabled(bool)` элемента `dsb`. Если в нижнем правом углу окна Qt Designer не видна панель **Signal/Slot Editor**, то выполним пункт меню **Tools | Signal/Slot Editor**. Для добавления нового соединения нажмем кнопку "+" и последовательно укажем все четыре параметра соединения (рис. 9.3). После этого в свойствах элемента `chb` следует задать начальное значение `checked=false`, а для элемента `dsb` — значение `enabled=false`. Теперь можно в режиме предварительного просмотра проверить, как изменяется доступность поля ввода числа при изменении состояния переключателя.

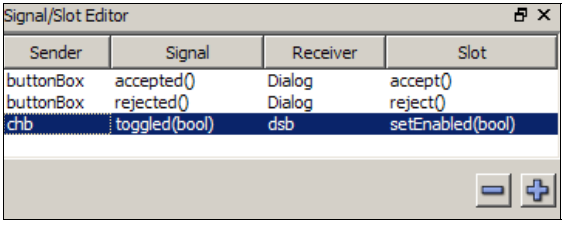


Рис. 9.3. Определение соединений между сигналами и слотами

Аналогично определим еще два соединения, соответствующие нажатию кнопок **Сохранить** и **Отмена** (см. рис. 9.3). Стандартные для модального диалога слоты `accept` и `reject` закрывают диалог и запоминают результат `Accepted` (принят) или `Rejected` (отклонен), который затем может быть прочитан с помощью метода `result()`.

Теперь предположим, что при изменении числа в поле `sb` необходимо тут же выводить это же значение в поле `le`. Воспользуемся еще одним способом оп-

ределения соединений. Войдем в режим визуального редактирования сигналов и слотов, выполнив пункт меню **Edit** | **Edit Signals/Slots** или просто нажав клавишу <F4>. С помощью левой кнопки мыши "зацепим" поле `sb` — источник сигнала и "бросим" его на элемент `le` — приемник сигнала (рис. 9.4).

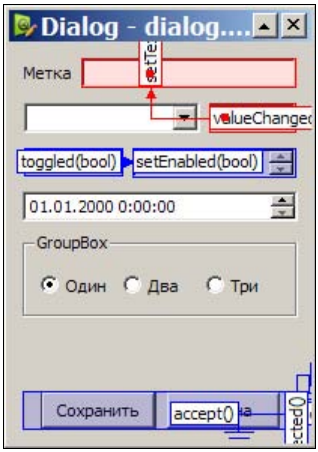


Рис. 9.4. Второй способ задания соединений между сигналами и слотами

В открывшемся диалоге (рис. 9.5) поставим галочку **Show all signals and slots** (показывать все сигналы и слоты), в левом окне выберем сигнал `valueChanged(QString)`, а в правом — слот `setText(QString)`. Нажмем кнопку **OK**. После этого можно снова проверить, как все работает (**Form** | **Preview**).

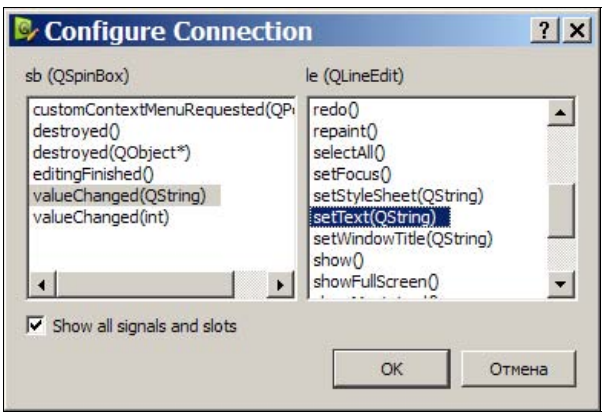


Рис. 9.5. Редактирование свойств соединений

Разумеется, для реализации более сложных действий (например, изменения доступности одних элементов диалога в зависимости от значений, вводимых пользователем в другие элементы) без настоящего программирования не обойтись.

Сохраним описание диалога под любым именем (например, `dialog`) и с расширением ui.

В листинге 9.1 приведен фрагмент получившегося в результате файла. Как видим, описание диалога хранится в формате XML.

Листинг 9.1. Файл *.ui

```
<ui version="4.0" >
 <class>Dialog</class>
 <widget class="QDialog" name="Dialog" >
  <property name="geometry" >
   <rect>
    <x>0</x>
    <y>0</y>
    <width>274</width>
    <height>259</height>
   </rect>
  </property>
  .......
</ui>
```

В листинге 9.2 показано, как использовать в программе диалог, разработанный с помощью Qt Designer.

Листинг 9.2. Файл chap09/designer1/dialog.cpp

```
#include <QApplication>
#include <QDialog>

#include "ui_dialog.h"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    Ui::Dialog ui;
    QDialog *dialog = new QDialog;
    ui.setupUi(dialog);
    dialog->show();
    return app.exec();
}
```

Мы подключили заголовочный файл, имя которого начинается префиксом `ui_`, после которого записывается имя файла, созданного в дизайнере. Откуда берется `ui_dialog.h`? Этот файл формируется с помощью утилиты `uic`, которая автоматически вызывается при компиляции программы.

Затем мы объявили экземпляр класса `Ui::Dialog` (этот класс описан в заголовочном файле) и экземпляр стандартного класса `QDialog`, после чего для первого вызвали метод `setupUi()`, указав второй в качестве параметра.

Компиляция программы проводится в обычном порядке: сначала команда

```
qmake -project
```

затем просто `qmake` и, наконец, `mingw32-make` или `nmake`. Автоматически создаваемый файл проекта *.pro будет содержать строку вида

```
FORMS += ИмяФайла1.ui ИмяФайла2.ui ИмяФайла3.ui
```

в которой перечислены все имеющиеся в текущем каталоге файлы *.ui с описанием диалоговых окон (в нашем случае такой файл всего один). Если мы добавляем новый диалог к уже имеющемуся проекту и не хотим (как это часто бывает) заново генерировать pro-файл, чтобы не потерять имеющиеся в нем настройки, то вместо выполнения команды `qmake -project` следует просто добавить к тексту pro-файла строку

```
FORMS += ИмяФайла.ui
```

Можете запустить программу на выполнение: на экран выводится окно диалога, доступность поля ввода вещественного числа зависит от состояния переключателя, изменяемое значение `sb` тут же отображается в однострочном поле ввода `le`, а при нажатии любой из двух кнопок диалог закрывается.

9.2. Использование диалога, созданного в Qt Designer

Для выполнения более сложных действий придется описать новый класс, например, `MyDialog`, указав в качестве базового стандартный класс `QDialog`. При этом относительно разработанного в дизайнере класса `Ui::Dialog` имеется две возможности: либо определить экземпляр этого класса внутри `MyDialog`, либо сделать класс `Ui::Dialog` вторым родителем класса `MyDialog`.

Первый вариант (с одним базовым классом) приведен в листингах 9.3 и 9.4.

Листинг 9.3. Файл chap09/designer2/dialog.h

```
1  #include <QDialog>
2  #include "ui_dialog.h"
3
```

```
 4  class MyDialog : public QDialog {
 5    Q_OBJECT
 6  public:
 7    MyDialog(QWidget *parent = 0);
 8  private:
 9    Ui::Dialog ui;
10  };
```

Листинг 9.4. Фрагмент файла chap09/designer2/dialog.cpp

```
11  #include <QtGui>
12  #include "dialog.h"
13
14  MyDialog::MyDialog(QWidget *parent)
15          : QDialog(parent) {
16
17     ui.setupUi(this);
18     ui.cb->addItem(tr("Первый"));
19     ui.cb->addItem(tr("Второй"));
20     ui.cb->addItem(tr("Третий"));
21     ui.cb->setCurrentIndex(2);
22
23     QRegExp re("[A-Za-z][1-9][0-9]{0,2}");
24     ui.le->setValidator(new QRegExpValidator(re, this));
25
26     ui.cb->addItem(tr("Первый"));
27     ui.cb->addItem(tr("Второй"));
28     ui.cb->addItem(tr("Третий"));
29     ui.cb->setCurrentIndex(2);
30     ui.cb->setEditable(true);
31     ui.cb->setInsertPolicy(QComboBox::InsertAtBottom);
32
33     ui.sb->setValue(5);
34
35     ui.chb->setCheckState(Qt::Checked);
36
37     ui.dsb->setMaximum(200.0);
38     ui.dsb->setDecimals(2);
39     ui.dsb->setSingleStep(0.25);
40     ui.dsb->setValue(100.25);
41
42     ui.dte->setDate(QDate(2007, 2, 5));
43     ui.dte->setTime(QTime(13, 35, 55, 10));
44
```

```
45      ui.rb2->setChecked(true);
46  }
47  //------------------------------
48  int main(int argc, char *argv[]) {
49      QApplication app(argc, argv);
50      QTextCodec *codec = QTextCodec::codecForName("CP1251");
51      QTextCodec::setCodecForTr(codec);
52
53      MyDialog *dlg = new MyDialog();
54      dlg->show();
55      return app.exec();
56  }
```

В строках (23, 24) мы использовали два новых элемента: *регулярное выражение* и *валидатор*. Подробнее об этих вещах мы поговорим чуть позже, пока же нам достаточно знать, что с их помощью мы ограничили набор возможных символов, которые разрешено вводить с клавиатуры, пока курсор стоит в однострочном поле ввода `le`: на первом месте может быть только строчная или заглавная латинская буква, на втором — число от 1 до 9, на третьем и четвертом местах — либо ничего, либо число от 0 до 9. Обратите внимание, что валидатор начинает работать, только если ввод данных начинается в пустом поле. Поскольку мы установили для него начальное значение (символ "5"), то описанные правила начнут действовать, только если стереть первоначальное содержимое поля.

Второй вариант (с двумя базовыми классами) показан в листингах 9.5 и 9.6.

Листинг 9.5. Файл chap09/designer3/dialog.h

```
1  #include <QDialog>
2  #include "ui_dialog.h"
3
4  class MyDialog : public QDialog, public Ui::Dialog {
5    Q_OBJECT
6  public:
7    MyDialog(QWidget *parent=0);
8  private slots:
9    void on_rb3_toggled(bool s);
10 };
```

Листинг 9.6. Файл chap09/designer3/dialog.cpp

```
11 #include <QtGui>
12 #include "dialog.h"
13
```

```
14  MyDialog::MyDialog(QWidget *parent)
15          : QDialog(parent) {
16     setupUi(this);
17     cb->addItem(tr("Первый"));
18     cb->addItem(tr("Второй"));
19     cb->addItem(tr("Третий"));
20     cb->setCurrentIndex(2);
       // ........
42  }
43
44  void MyDialog::on_rb3_toggled(bool s) {
45    dte->setHidden(s);
46  }
47  //---------------------------------
48  int main(int argc, char *argv[]) {
    ...
56  }
```

Как видим, к элементам диалога можно обращаться по тем именам, которые были указаны для них в Qt Designer.

Соединения между сигналами и слотами для класса `MyDialog` можно определять как обычно, с помощью метода `connect` (например, в конструкторе класса `MyDialog`), при этом пользовательские слоты должны быть перечислены в объявлении класса или унаследованы от базовых классов. Но можно воспользоваться особым правилом именования слотов:

```
private slots:
    on_ИмяВиджета_ИмяСигнала(ПараметрыСигнала);
```

В этом случае соединение между сигналом и слотом будет выполнено автоматически (в методе `setupUi()`, который генерируется утилитой `uic`). Именно этот способ мы применили в строках (8, 9, 44) листинга 9.6. В результате при выборе переключателя `rb3` поле ввода даты/времени будет спрятано, а при выборе любого другого — снова показано.

Оба описанных варианта компоновки внешнего ресурса с основной программой являются статическими: после компиляции программы ui-файл можно удалить. Существует и вариант динамической загрузки во время выполнения программы (листинг 9.7). Здесь используется класс `QUiLoader`. При этом в тексте программы требуется подключить заголовочный файл `QtUiTools`, а в файле проекта `*.pro` — убрать строку `FORMS` и добавить строку

```
CONFIG += uitools
```

Напоминаем, что после редактирования pro-файла не следует выполнять команду `qmake` с параметром `-project`, иначе все изменения в нем будут поте-

ряны. В нашем случае файл dialog.ui должен находиться в том же каталоге, что и исполняемый файл программы.

Листинг 9.7. Динамическая загрузка ui-ресурса (файл chap09/designer4/dialog.cpp)

```
#include <QtGui>
#include <QtUiTools>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);

    QUiLoader uiLoader;
    QFile file("dialog.ui");
    file.open(QFile::ReadOnly);
    QWidget *dlg = uiLoader.load(&file);
    file.close();
    if (dlg) {
        QComboBox *cb = dlg->findChild<QComboBox*>("cb");
        cb->addItem(QObject::tr("Первый"));
        cb->addItem(QObject::tr("Второй"));
        cb->addItem(QObject::tr("Третий"));
        cb->setCurrentIndex(2);
        cb->setEditable(true);
        cb->setInsertPolicy(QComboBox::InsertAtBottom);

        QSpinBox *sb = dlg->findChild<QSpinBox*>("sb");
        sb->setValue(5);

        dlg->findChild<QCheckBox*>("chb")->setCheckState(Qt::Checked);

        // .......

        dlg->findChild<QRadioButton*>("rb2")->setChecked(true);

        dlg->show();
        return app.exec();
    }
    else
        return 1; // Код ошибки
}
```

Для обращения к элементам диалога при динамической загрузке используется метод

```
parent->findChild<Тип *>("ИмяЭлемента");
```

или функция

```
qFindChild<Тип *>(parent, "ИмяЭлемента");
```

Второй вариант предназначен для компиляторов, которые не поддерживают шаблоны методов (например, Microsoft Visual C++ 6.0).

ЗАДАЧА. С помощью Qt Designer разработать форму с нестандартным виджетом.

Решение. Если новый виджет определен на основе стандартного (например, программист разработал свой класс `MyTable`, выбрав в качестве базового класс `QTableWidget` или `QTableView`), тогда в Qt Designer следует задействовать именно этот базовый виджет, но после размещения его на форме нужно с помощью контекстного меню выполнить команду **Promote to** (Перевести в) и указать имя своего класса и название заголовочного файла.

Чуть позже мы узнаем, как создавать совершенно новые визуальные элементы и добавлять их в библиотеку виджетов, чтобы использовать совместно с Qt Designer.

В *разд. 18.4* мы покажем, как с помощью Qt Designer разрабатывать приложения баз данных.

9.3. Отладка программ

> Вреднее всего желание предохранить себя от ошибок.
>
> *Георг Вильгельм Фридрих Гегель*

Наши программы уже стали достаточно сложными, чтобы вспомнить об отладке. Если вы работаете с Microsoft Visual Studio, то достаточно скомпилировать проект в режиме Debug, чтобы потом насладиться всеми возможностями интегрированного отладчика. Если имеется QDevelop или KDevelop, то можно использовать отладчик GDB (встроенный в интегрированную среду или из командной строки).

Также есть возможность включать в текст программы отладочные сообщения, вывод которых после окончания процесса отладки можно запретить.

ЗАДАЧА. При выполнении программы в режиме Debug необходимо выводить на консоль отладочные сообщения, облегчающие процесс отладки. При выполнении программы в обычном режиме требуется запретить вывод отладочных сообщений.

Решение. Вывести отладочные сообщения можно с помощью функции `qDebug()`, которая работает аналогично стандартной `printf()`, например:

```
qDebug("n = %d", n);
```

Сообщения будут выводиться в стандартный поток `stderr` (если он не переопределен). В Windows для неконсольных приложений сообщения будут переданы отладчику. Чтобы они выводились на консольный экран, нужно в файле описания проекта добавить строку:

```
win32: CONFIG += console.
```

Чтобы подавить вывод сообщений, необходимо во время компиляции определить `QT_NO_DEBUG_OUTPUT`.

Если в программе подключить модуль `<QtDebug>`, то становится доступным расширенный синтаксис, поддерживающий вывод не только данных стандартного типа, но и многих классов Qt, например:

```
qDebug() << tr("Кисть myBrush:)" << myBrush << " j =" << j;
```

ЗАДАЧА. Требуется выводить отладочные сообщения не только на консоль, но также в виде надписи `QLabel` и текста `QTextEdit`.

Решение. В листинге 9.8 приведен текст программы, а на рис. 9.6 показан результат ее работы.

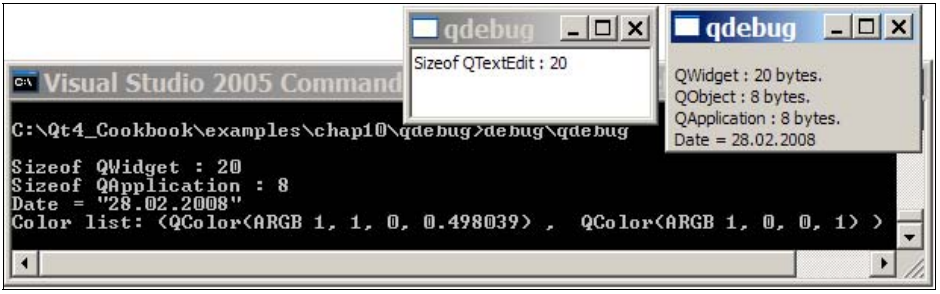


Рис. 9.6. Вывод отладочных сообщений на консоль, в надпись `QLabel` и текстовый редактор `QTextEdit`

Листинг 9.8. Файл chap10/qdebug/qdebug.cpp

```
#include <QApplication>
#include <QString>
#include <QLabel>
#include <QTextEdit>
```

```
#include <QWidget>
#include <QTextStream>
#include <QDate>
#include <QList>
#include <QtDebug>

int main(int argc, char * argv[]) {
    QApplication app(argc, argv);
    QWidget widget;
    qDebug() << "\nSizeof QWidget :" << sizeof(widget)
             << "\nSizeof QApplication :" << sizeof(app);

    QDate date(2008, 02, 28);
    qDebug() << "Date =" << date.toString("dd.MM.yyyy");

    QList<QColor> colorlist;
    colorlist << QColor(255, 0, 127) << QColor("blue");
    qDebug() << "Color list:" << colorlist;

    QString message;
    QTextStream buf(&message);
    buf << "\n QWidget : " << sizeof(widget)
        << " bytes. \n QObject : " << sizeof(QObject)
        << " bytes. \n QApplication : " << sizeof(app)
        << " bytes.\n Date = " << date.toString("dd.MM.yyyy");

    QLabel label(message); // текстовая надпись
    label.show();

    QTextEdit te;          // текстовый редактор
    te.append(QObject::tr("Sizeof QTextEdit : %1").arg(sizeof(te)));
    te.show();

    return app.exec();
};
```

Текст файла описания проекта *.pro:

```
TEMPLATE = app
SOURCES += qdebug.cpp
win32: CONFIG += console
```

Последняя строка обеспечивает вывод на консоль в системе Windows.

9.4. Система помощи

> Лучше самая малая помощь, чем самое большое сочувствие.
>
> *Владислав Лоранц*

Программист может помочь пользователям своей программы, активно используя в ней следующее:

- всплывающие сообщения с подсказкой о назначении каждого элемента интерфейса, которые появляются на экране, если указатель мыши на какое-то время задерживается над данным элементом;
- пояснения по текущим пунктам меню и кнопкам панели инструментов, отображаемые в строке состояния главного окна приложения;
- более развернутые сообщения, выводимые на экран после того, как пользователь активизирует элемент интерфейса и включит режим "Что это?", нажав соответствующую кнопку в заголовке окна диалога или комбинацию клавиш <Shift>+<F1>;
- подробное описание всех возможностей программы и порядка работы с ней, снабженное удобными элементами навигации и поиска.

Существует несколько видов подсказок.

- **Всплывающие сообщения с подсказкой.** Для задания текста всплывающего сообщения предназначен метод `setToolTip(tr("..."))`. Он доступен для всех виджетов `QWidget`, действий `QAction`, графических примитивов `QGraphicsItem`, элементов стандартных моделей `QStandardItem` и т. п. Подсказка будет отображаться автоматически при наведении указателя мыши на данный элемент, но только если он расположен в активном окне (имеющем фокус ввода). Можно сделать так, чтобы подсказка отображалась и в неактивном окне, для этого нужно у окна (а не у самого элемента!) установить флаг `Qt::WA_AlwaysShowToolTips`. Если задать пустую строку, то подсказка не будет отображаться.

Замечание

Можно перехватить событие `QEvent::ToolTip` и изменить, например, положение или внешний вид всплывающих сообщений.

- **Подсказка в строке состояния.** Задать текст пояснения, которое будет появляться в строке состояния при наведении указателя мыши на данный элемент интерфейса, можно с помощью метода `setStatusTip(tr("..."))`.
- **Фраза "Что это?".** Для задания текста более подробной подсказки предназначен метод `setWhatsThis(tr("..."))`. Для форматирования пригодны теги HTML (в том числе `<img>`). Если нас не устраивают системные спосо-

бы входа в режим "Что это?", то можно использовать класс `QWhatsThis`, а также статические методы `QWhatsThis::enterWhatsThisMode()` и `QWhatsThis::leaveWhatsThisMode()` для программного входа в режим "Что это?" и выхода из него.

Чтобы предоставить пользователю систематизированное описание возможностей программы, нужно подготовить его в виде связанных файлов *.html, для показа которых подойдет стандартный браузер:

```
QUrl url("имя файла.html");
url.setScheme("file");
url.setFragment("метка");
QDesktopServices::openUrl(url);
```

Отображение HTML-файлов и навигация по ссылкам возможны также с помощью класса `QTextBrowser`. Но наиболее мощные средства для построения системы помощи предоставляет программа `Qt Assistant` и модуль `QtHelp`.

Для создания системы помощи, кроме набора HTML-страниц, в которых, собственно, и хранится все описание программы, потребуется еще файл *.qhp, содержащий оглавление, индексы и ссылки на страницы HTML. Формат этого файла приведен в листинге 9.9.

Листинг 9.9. Пример файла *.qhp

```
<?xml version="1.0" encoding="UTF-8"?>
    <QtHelpProject version="1.0">
       <namespace>mycompany.ru.myapplication.1_0</namespace>
       <virtualFolder>doc</virtualFolder>
       <customFilter name="My Application 1.0">
          <filterAttribute>myapp</filterAttribute>
          <filterAttribute>1.0</filterAttribute>
       </customFilter>
       <filterSection>
          <filterAttribute>myapp</filterAttribute>
          <filterAttribute>1.0</filterAttribute>
          <toc>
              <section title="Мое приложение" ref="index.html">
                <section title="Глава 1" ref="doc.html#chapter1"/>
                <section title=" Глава 2" ref="doc.html#chapter2"/>
                <section title=" Глава 3" ref="doc.html#chapter3"/>
              </section>
          </toc>
          <keywords>
              <keyword name="foo"
                   id="MyApplication::foo" ref="doc.html#foo"/>
```

```
            <keyword name="bar" ref="doc.html#bar"/>
            <keyword id="MyApplication::foobar"
                 ref="doc.html#foobar"/>
        </keywords>
        <files>
            <file>classic.css</file>
            <file>index.html</file>
            <file>doc.html</file>
            <file>img/fig.png</file>
        </files>
    </filterSection>
</QtHelpProject>
```

После подготовки такого файла нужно запустить компилятор (утилиту командной строки) `qhelpgenerator`:

```
qhelpgenerator doc.qhp -o doc.qch
```

Результатом будет скомпилированный файл помощи doc.qch, в него в сжатом виде войдут все файлы, перечисленные в разделе `<files>`.

Чтобы проверить содержимое скомпилированного файла, следует запустить Qt Assistant и открыть файл, выполнив команды **Edit** (Правка) | **Preferences** (Настройки) | **Documentation** (Документация) | **Add** (Добавить). После этого на странице содержания справочной системы (вкладка **Contents**) появится новая тема, в индексе (вкладка **Index**) отобразятся новые ключевые слова, а содержимое добавленных текстов станет доступно в системе поиска (страница **Search**).

Но это еще не все. Несколько скомпилированных файлов помощи могут группироваться в так называемые коллекции *.qhc. Для формирования такой коллекции надо подготовить файл *.qhcp, в котором должны содержаться ссылки на скомпилированные файлы помощи, которые требуется включить в коллекцию (листинг 9.10).

Листинг 9.10. Пример файла *.qhcp

```
<?xml version="1.0" encoding="utf-8" ?>
  <QHelpCollectionProject version="1.0">
    <docFiles>
        <register>
            <file>doc.qch</file>
            <file>doc2.qch</file>
        </register>
    </docFiles>
  </QHelpCollectionProject>
```

В принципе, коллекция может состоять и из одного файла.

Затем с помощью компилятора коллекций `qcollectiongenerator` формируется файл коллекции *.qhc:

```
qcollectiongenerator mycollection.qhcp -o mycollection.qhc
```

Существует возможность получить скомпилированный файл коллекции сразу, минуя этап формирования файла *.qch, для этого описание коллекции необходимо изменить так, как показано в листинге 9.11.

Листинг 9.11. Пример включения нескомпилированного проекта *.qhp в коллекцию *.qhcp

```
...
<docFiles>
    <generate>
        <file>
            <input>doc.qhp</input>
            <output>doc.qch</output>
        </file>
    </generate>
    <register>
        <file>doc.qch</file>
    </register>
</docFiles>
...
```

Чтобы открыть файл коллекции в Qt Assistant, его можно просто указать в командной строке:

```
assistant -collectionFile file
```

Другая возможность — зарегистрировать файл коллекции в системе помощи, чтобы он открывался автоматически. Для этого вместо ключа `-collectionFile` нужно указать `-register`.

Если требуется вызвать Qt Assistant из приложения Qt, используют `QProcess` (листинг 9.12).

Листинг 9.12. Создание процесса для вызова Qt Assistant

```
QProcess *p = new QProcess;
QStringList args;
args << QLatin1String("-collectionFile")
    << QLatin1String("mycollection.qhc")
    << QLatin1String("-enableRemoteControl");
```

```
p->start(QLatin1String("assistant"), args);
if (!p->waitForStarted())
    return;
```

Благодаря флагу `-enableRemoteControl` появляется возможность управлять поведением Qt Assistant из приложения, например:

```
QTextStream str(p);
str << QLatin1String(
        "setSource qthelp://com.mycompany.1_0_0/doc/index.html\0")
    << endl;
```

Замечание

До релиза 4.4 в библиотеке Qt существовали другие форматы файлов описания проекта системы помощи: DCF и ADP. Для преобразования к новому формату предназначена утилита `qhelpconverter`.

Если по каким-либо причинам нам не хочется показывать файлы документации во внешней программе, мы можем интегрировать их в наше приложение. Это делается с помощью средств, предоставляемых модулем `QtHelp`.

Сначала в приложении создают объект `QHelpEngineCore`:

```
QHelpEngineCore helpEngine("mycollection.qhc");
```

Затем, когда требуется получить ссылки на все файлы подсказки, связанные, например, с ключевым словом `MyDialog::ChangeButton`:

```
QMap<QString, QUrl> links = helpEngine.linksForIdentifier(
                        QLatin1String("MyDialog::ChangeButton"));
```

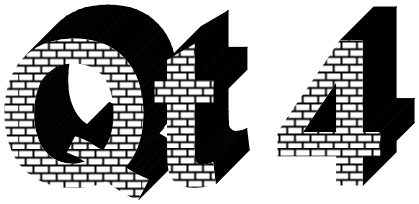
Теперь можно показать, например, содержимое первой ссылки (листинг 9.13).

Листинг 9.13. Формирование текста подсказки

```
if (links.count()) {
    QByteArray helpData = helpEngine->fileData(
                              links.constBegin().value());
    if (!helpData.isEmpty())
       // Выводим в какой-нибудь виджет данные helpData
}
```

Законченный проект находится в каталоге примеров библиотеки Qt (examples \help\contextsensitivehelp).

ГЛАВА 10



Работа с данными

Господи, дай мне спокойствие принять то, чего я не могу изменить, дай мне мужество изменить то, что я могу изменить, и дай мне мудрость отличить одно от другого.

Кристоф Фридрих Ойтингер

10.1. Числа и строки

Вместо обычных для C/C++ элементарных типов данных, которые, тем не менее, можно использовать и в приложениях Qt, в самой библиотеке Qt определены платформонезависимые типы:

- `qint8` — целое со знаком (8 битов), аналог типа `signed char`;
- `quint8` и `uchar` — целое неотрицательное (8 битов, `unsigned char`);
- `qint16` — целое со знаком (16 битов, `short`);
- `quint16` и `ushort` — целое неотрицательное (16 битов, `unsigned short`);
- `qint32` — целое со знаком (32 бита, `int`);
- `quint32` и `uint` — целое неотрицательное (32 бита, `unsigned int`);
- `qint64` — целое со знаком (64 бита, `long`);
- `quint64` и `ulong` — целое неотрицательное (64 бита, `unsigned long`);
- `qlonglong` — эквивалент `qint64`;
- `qulonglong` — эквивалент `quint64`;
- `qreal` — вещественное число, аналог `double`, за исключением платформ с микропроцессорами архитектуры ARM (*Advanced RISC Machines*), в этом случае тип `qreal` определен как `float`;
- `quintptr` — указатель, соответствует `quint32` или `quint64`, в зависимости от разрядности платформы;

- `qptrdiff` — разность двух указателей, соответствует `qint32` или `qint64`, в зависимости от разрядности платформы.

Для литералов типа `quint64` предусмотрен макрос `Q_UINT64_C()`, например:

```
quint64 val = Q_UINT64_C(932838457459459);
```

Аналогично для `qint64`:

```
qint64 value = Q_INT64_C(932838457459459);
```

Узнать, как на данной платформе располагаются байты многобайтовых чисел, можно с помощью класса `QSysInfo` (листинг 10.1).

Листинг 10.1. Класс QSysInfo

```
switch ( QSysInfo::ByteOrder ) {
      case QSysInfo::BigEndian:
          // сначала старший
          break;
      case QSysInfo::LittleEndian:
          // сначала младший
}
```

Для преобразования форматов применяются функции `qFromLittleEndian()`, `qFromBigEndian()`, `qToBigEndian()`, `qToLittleEndian()`.

Для работы с символами Unicode предназначен класс `QChar`, со строками Unicode — класс `QString`.

10.2. Массивы и списки

В библиотеке Qt имеется несколько шаблонных классов для работы с массивами и списками: вектор `QVector<T>`, список `QList<T>`, двусвязный список `QLinkedList<T>` и низкоуровневый класс для работы с массивами переменной длины `QVarLengthArray<T>`. Все они обладают сходной функциональностью. Для большинства случаев рекомендуется `QList`. Для работы со списком строк имеется специальный класс `QStringList`. Если требуется, чтобы время вставки/удаления элемента не зависело от его порядкового номера, то лучше обратиться к `QLinkedList`. Если нужно, чтобы элементы с соседними порядковыми номерами располагались в памяти рядом, то используйте `QVector`.

Общий алгоритм работы со списком `QList`:

- Объявить объект — список элементов нужного типа, например:

```
QList<int> intList;  // список целых чисел
QList<QDate> dateList;   // список дат
QList<QString> strList;  // список строк
```

- Заполнить список значениями. Добавить элемент в конец списка можно методом `append()` или оператором <<:

```
intList << 1 << 10 << -20;
dateList.append(QDate(2008, 03, 31));
strList.append(tr("Строка"));
```

  Для добавления элемента в начало списка существует метод `prepend()`.

- Выполнить необходимые операции над списком, например:

```
strList[3] = "Другая строка"; // заменили существующее значение
strList.at(3) = "Другая строка"; // более быстрый вариант
intList.removeAt(1); // удалили элемент
dateList.swap(0, 2); // поменяли элементы местами
strList.clear(); // очистили список (удалили из него все элементы)
```

Хранить в списках можно только такие объекты, для которых определены конструктор по умолчанию, конструктор копирования и оператор присваивания. Этим требованиям удовлетворяют простые типы данных C++, а также многие невизуальные классы библиотеки Qt. Визуальные объекты — наследники `QWidget` не могут храниться в списках и векторах непосредственно, но вместо самих объектов всегда можно хранить указатели на них.

ЗАДАЧА. Определить класс `Employee` (работник), который можно будет хранить в списке `QList` или массиве `QVector`.

Решение иллюстрирует листинг 10.2.

Листинг 10.2. Класс "Работник"

```
class Employee {
public:
     Employee() {} ;  // Конструктор по умолчанию
     Employee(const Employee &other);  // Конструктор копирования
     Employee &operator=(const Employee &other); // Оператор =
private:
     QString surname, name, patronymic; // ФИО
     QDate dateOfBirth;                 // Дата рождения
 };
```

Замечание

Если конструктор копирования и/или оператор присваивания в классе не определены, то компилятор C++ пытается реализовать их поэлементным копированием переменных — членов класса. Если не определен конструктор по умолчанию, то аналогично делается попытка инициализации всех переменных их значениями по умолчанию. Во всех случаях требуется, чтобы переменные —

члены класса по отдельности удовлетворяли перечисленным требованиям, т. е. имели конструктор по умолчанию, конструктор копирования и оператор присваивания (или могли быть скопированы и проинициализированы по умолчанию).

ЗАДАЧА. Сравнить на равенство/неравенство список `QList` и вектор `QVector`.

Решение. Для поэлементного сравнения структур, даже если формально их типы не совпадают, в библиотеке Qt имеется функция `qEqual()`. В листинге 10.3 приведен пример ее использования, а на рис. 10.1 показан результат работы программы.

Рис. 10.1. Поэлементное сравнение списка и вектора

Листинг 10.3. Сравнение структур разного типа (файл chap10/compare/main.cpp)

```
#include <QStringList>
#include <QVector>
#include <QtDebug>
using namespace std;
int main(){
   QStringList list;
   list << "abc" << "ab" << "cde";

   QVector<QString> vect(3);
   vect[0] = "abc";
   vect[1] = "ab";
   vect[2] = "cde";

   bool x = qEqual( list.begin(),
                    list.end(),
                    vect.begin());
   qDebug() << list << endl << vect << endl << x; // true
   qDebug() << "------------------" << endl;
```

```
19      list.replace(0, "lala"); // или list[0] = "lala";
20      x = qEqual( list.begin(),
21                  list.end(),
22                  vect.begin());
23      qDebug() << list << endl << vect << endl << x; // false
24
25      return 0;
26  }
```

Перед компиляцией необходимо в файл *.pro добавить строку

```
CONFIG += console
```

10.3. Контейнеры и итераторы

В библиотеке Qt имеются два типа итераторов:

- в стиле STL;
- в стиле Java.

Итератор в стиле STL указывает непосредственно на сам элемент; метод `begin()` возвращает указатель на первый элемент; метод `end()` — на "пустое место", следующее за самым последним элементом. Напротив, итератор в стиле Java указывает не на элемент, а на промежуток между двумя соседними элементами; метод `toFront()` возвращает указатель на место "перед первым элементом", метод `toBack()` — после последнего элемента. Итераторы в стиле Java несколько уступают в производительности, но проще в использовании, чем итераторы в стиле STL.

ЗАДАЧА. С помощью итератора в стиле STL перебрать все элементы строкового списка по порядку, преобразуя все символы к нижнему регистру.

Решение иллюстрирует листинг 10.4.

Листинг 10.4. Итератор в стиле STL: перебор по порядку

```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QList<QString>::iterator i;
for (i = list.begin(); i != list.end(); ++i) {
    *i = (*i).toLower();
    qDebug() << i << " : " << *i ;
}
```

ЗАДАЧА. С помощью итератора в стиле STL перебрать все элементы строкового списка в обратном порядке (от конца к началу), преобразуя все символы к нижнему регистру.

Решение иллюстрирует листинг 10.5.

Листинг 10.5. Итератор в стиле STL: перебор в обратном порядке

```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QList<QString>::iterator i = list.end();
while (i != list.begin()) {
    --i;
    *i = (*i).toLower();
    qDebug() << i << " : " << *i ;
}
```

Замечание

Если при переборе элементов контейнера не предполагается изменять их значение, то эффективнее использовать константные итераторы `const_iterator`. В этом случае методы `begin()` и `end()` нужно заменить на `constBegin()` и `constEnd()`.

Константные итераторы можно заменить макросом `foreach` ("для каждого"), например:

```
QList<double> list;
list << 1.0 << 2.5 << 3.1415;
foreach ( double x, list) {
    qDebug() << i << " : " << x ;
}
```

ЗАДАЧА. Используя итератор в стиле Java, перебрать все элементы списка по порядку (не изменяя их).

Решение иллюстрирует листинг 10.6.

Листинг 10.6. Константный итератор в стиле Java: перебор по порядку

```
QList<QString> list;
list << "A" << "B" << "C" << "D";
QListIterator<QString> i(list);
while (i.hasNext())
    qDebug() << i.next();
```

ЗАДАЧА. С помощью итератора в стиле Java перебрать все элементы списка в обратном порядке (не изменяя их).

Решение иллюстрирует листинг 10.7.

Листинг 10.7. Константный итератор в стиле Java: перебор в обратном порядке

```
QListIterator<QString> i(list);
i.toBack();
while (i.hasPrevious())
    qDebug() << i.previous();
```

Замечание

Для списка `QList` и очереди `QQueue` применяется константный итератор `QListIterator`, для двусвязного списка `QLinkedList` — итератор `QLinkedListIterator`, для вектора `QVector` и стека `QStack` — итератор `QVectorIterator`, для множества `QSet` — `QSetIterator`, для `QMap` — `QMapIterator`, для `QHash` — `QHashIterator`. Если требуется изменять элементы, то используются итераторы `QMutableListIterator`, `QMutableLinkedListIterator`, `QMutableVectorIterator`, `QMutableSetIterator`, `QMutableMapIterator`, `QMutableHashIterator`.

ЗАДАЧА. Используя итератор в стиле Java, удалить все нечетные и обнулить все отрицательные элементы списка.

Решение иллюстрирует листинг 10.8.

Листинг 10.8. Итератор для изменения элементов

```
QList<int> list;
QMutableListIterator<int> i(list);
while (i.hasNext()) {
   int y = i.next();
   if (y % 2 != 0)
       i.remove();
   else if ( y < 0 )
       i.setValue(0);
}
```

10.4. Класс *QObject*

Класс `QObject` является базовым для большинства других классов библиотеки Qt.

Особенности класса QObject и всех его прямых или косвенных потомков:

- поддержка *сигналов и слотов* — все классы, в которых используется механизм сигналов и слотов, должны прямо или косвенно наследовать класс QObject;
- поддержка *фильтрации событий* — в любом потомке QObject можно установить фильтр для перехвата событий, что позволяет изменять поведение объекта без переписывания исходного кода класса;
- механизм *иерархии объектов* (экземпляров классов, а не самих классов!) — при создании потомка QObject можно указать родительский объект parent, который перед своим уничтожением автоматически удалит из памяти всех своих потомков. В результате программист освобождается от необходимости следить за освобождением памяти. Все объекты, при создании которых указан родитель parent, могут размещаться только в динамической памяти (т. е. должны создаваться с помощью оператора new);
- механизм *метаобъектной информации* — поддерживаются методы, позволяющие узнать имя класса и имена всех его классов-родителей, а также корректное приведение типов с помощью шаблонной функции qobject_cast<T>();
- *динамические свойства объектов* — дополнительные атрибуты объектов, имеющие имя, значение, методы доступа (чтения/записи/сброса) и параметры управления видимостью. Набор свойств может изменяться во время выполнения программы;
- поддержка *таймера* — любой потомок QObject имеет средства для запуска таймера и перехвата его срабатываний (без предварительного создания отдельного объекта QTimer).

В любом определении нового класса, если он использует механизм сигналов/слотов или метаобъектную информацию, в разделе private обязательно должен быть указан макрос Q_OBJECT. При множественном наследовании необходимо соблюдать два условия:

- класс QObject (или унаследованный от него) должен быть указан первым в списке родителей:

```
class MyClass : public QObject, public AnotherClass {
    Q_OBJECT
   // . . .
}
```

- среди родителей может быть только один класс — наследник QObject. Например, в предыдущем примере класс AnotherClass не может быть наследником QObject.

10.5. Неявное совместное использование данных

Для повышения эффективности работы в библиотеке Qt вместо копирования данных предусмотрены ссылки на эти данные. В результате несколько объектов, имеющих совпадающие значения, могут ссылаться на одну и ту же область памяти. Чтобы учесть количество этих объектов и вовремя освободить область памяти, которую больше никто не использует, применяют счетчики ссылок.

Рассмотрим листинг 10.9.

Листинг 10.9. Иллюстрация совместного использования данных

```
1   QString s1, s2;
2   s1 = "Добрый";
3   s2 = s1;
4   s2 += " день!"
```

Пояснения к листингу:

- (1) — сначала оба объекта (строки `QString`) не содержат данных, т. е. указатель на данные у них обоих равен `null`;
- (2) — первый объект получает некоторое значение. Для хранения данных (в нашем случае это строка текста) отводится некоторая область в динамической памяти. Теперь указатель на данные для объекта `s1` ссылается на эту область, а счетчик ссылок для этой области равен единице;
- (3) — вместо того, чтобы копировать значение первого объекта во второй, мы просто копируем указатель на нашу область памяти. Теперь оба указатели ссылаются на одну и ту же область, а счетчик ссылок для нее становится равным двум;
- (4) — значение второго объекта изменилось, поэтому ничего не остается, как выделить для второй строки новую область данных, скопировать в эту область все символы первой строки и приписать к ней новые. Теперь указатели на данные для обоих объектов ссылаются на разные области памяти, а счетчики ссылок для каждой области равны единице.

Такая модель получила название *неявного совместного использования данных* (*implicit data sharing*). Она широко применяется при разработке следующих классов библиотеки Qt: `QBitArray`, `QBitmap`, `QBrush`, `QByteArray`, `QCache`, `QCursor`, `QDir`, `QFileInfo`, `QFont`, `QFontInfo`, `QFontMetrics`, `QFontMetricsF`, `QGLColormap`, `QGradient`, `QHash`, `QIcon`, `QImage`, `QKeySequence`, `QLinkedList`, `QList`,

QLocale, QMap, QMultiHash, QMultiMap, QPalette, QPen, QPicture, QPixmap, QPolygon, QPolygonF, QQueue, QRegExp, QRegion, QSet, QSqlField, QSqlQuery, QSqlRecord, QStack, QString, QStringList, QTextBoundaryFinder, QTextCursor, QTextDocumentFragment, QTextFormat, QUrl, QVariant, QVector, QX11Info. Термин "неявный" означает, что отсоединение объекта от общей области данных перед изменением его значения происходит автоматически, без явного вызова какой-либо функции. Пример такого подхода при работе с графическими изображениями содержит листинг 10.10.

Листинг 10.10. Два графических объекта

```
1   QPixmap p1, p2;
2   p1.load("image.png");
3   p2 = p1;      // p1 и p2 используют одну область памяти
4
5   QPainter paint;
6   paint.begin(&p2); // теперь для p2 выделена новая область памяти
7   paint.drawText( 10, 30, "..." );
8   paint.end();
```

Разумеется, библиотека Qt предоставляет программисту средства для создания собственных классов, работающих на основе этой модели.

ЗАДАЧА. Разработать класс, в котором применяется модель неявного совместного использования данных.

Решение. Допустим, мы хотим хранить и обрабатывать информацию о работниках предприятия. Соответствующий класс приведен в листинге 10.11.

Листинг 10.11. Исходный класс "Работник"

```
#include <QString>
class Employee {         // Работник
public:
   Employee(ind id, const QString& name) {   // Конструктор
      setId(id);              // Установить табельный номер
      setName(name);          // Установить имя
   }
   ~Employee() {}             // Деструктор
   void setId(int id){ this->id = id; }
   void setName(const QString& name){ this->name = name; }
   int id() const {return this->id; } // Вернуть номер
   const QString& name() const {return this->name;} // Вернуть имя
```

```
private:
   int id;         // Табельный номер
   QString name;   // Имя
};
```

Чтобы класс `Employee` мог неявно использовать общие данные, нужно вынести все переменные — члены класса `Employee` в отдельный класс, например, `EmployeeData`, сделав его наследником специального класса `QSharedData`, а в классе `Employee` оставить только указатель на `EmployeeData`: (листинг 10.12).

Листинг 10.12. Класс "Работник" с неявным использованием общих данных

```
   #include <QSharedData>
   #include <QString>
   class EmployeeData : public QSharedData {
   public:
     EmployeeData() : id(0) { name.clear(); } // Конструктор по умолчанию
     EmployeeData(const EmployeeData& other) // Конструктор копирования
         : QSharedData(other), id(other.id), name(other.name) { }
     ~EmployeeData() { }
     int id;
     QString name;
};

class Employee {
public:
   Employee() { d = new EmployeeData; }
   Employee(int id, QString name) {
       d = new EmployeeData;
       setId(id);
       setName(name);
   }
   void setId(int id) { d->id = id; }
   void setName(QString name) { d->name = name; }
   int id() const { return d->id; }
   QString name() const { return d->name; }
private:
   QSharedDataPointer<EmployeeData> d;
};
```

Доступ ко всем переменным класса осуществляется через указатель `d`. Операция разыменования `->` переопределена в классе `QShareDataPointer`, в ней автоматически вызывается специальная функция `detach()`. Если данная

функция выполняется в теле функции, не помеченной модификатором `const`, то проверяется значение счетчика ссылок, и если оно больше единицы, то объект-наследник класса `QSharedData` копируется в новую область динамической памяти, счетчик ссылок для старого объекта уменьшается на единицу, сам указатель обновляется, а значение счетчика ссылок для нового объекта устанавливается равным единице.

Кроме того, объект `QSharedData` автоматически удаляется, когда его счетчик ссылок становится равным нулю.

Напротив, при выполнении операции присваивания или вызове конструктора копирования реального копирования данных не происходит, присваивается только указатель `d` и увеличивается на единицу значение счетчика ссылок. Причем конструктор копирования достаточно определить только для объекта данных, а копирование интерфейсного объекта `Employee` обеспечивает компилятор.

10.6. Явное совместное использование данных

Существует и модель *явного совместного использования данных* (*explicit data sharing*), но в библиотеке Qt остался только один класс, который ее задействует. Это класс `QWebHistoryItem`, введенный в релизе 4.4 в связи с появлением нового модуля `WebKit`. С помощью класса `QWebHistoryItem` представляется одна запись в списке-истории посещенных Web-страниц `QWebPage`. В каждой такой записи хранится заголовок страницы, ее адрес, дата/время последнего посещения и пиктограмма страницы.

В остальных классах, где ранее применялась явная модель, в новых релизах она была заменена неявной. Но разумеется, никто не запрещает программисту разрабатывать свои классы на основе явной модели.

ЗАДАЧА. Изменить класс `Employee`, чтобы он мог явно использовать общие данные.

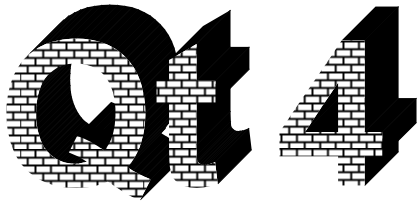
Решение. Для реализации явной модели в библиотеке Qt имеется класс `QExplicitlySharedDataPointer`. Единственное, что придется изменить в листинге 10.12, — это тип указателя `d`. Но в результате разыменование указателя даже в методах, не помеченных модификатором `const`, уже не будет автоматически вызывать функцию `detach()`. Другими словами, указатель `d` станет похож на обычный указатель языка C/C++ с тем исключением, что он все еще обеспечивает подсчет ссылок и автоматическое удаление объекта с нулевым значением счетчика. Функцию `detach()` после этого придется вызывать явно.

Когда явное использование общих данных удобнее неявного? Продолжим пример о работниках предприятия. Рассмотрим фрагмент кода:

```
Employee e1 ( 1, tr("Иванов В.К.") );
Employee e2 = e1;
e1.setName( tr("Петров А.И.") );
```

Если класс `Employee` использует неявную модель, то при изменении имени первого работника в памяти создаются два объекта `EmployeeData` с одинаковыми табельными номерами, но разными именами. Вряд ли мы хотели этого. Скорее всего, нового работника целесообразно создавать не только с новым именем, но и со своим уникальным номером. Поэтому нужно как-то разделить эти два случая: переименование старого работника со старым табельным номером (тогда в случае явной модели переименование `e1` повлечет за собой и изменение `e2`) или создание нового работника со своим именем и номером (тогда перед переименованием `e1` нужно вызвать функцию `detach()`). Таким образом, явная модель в данном случае выглядит более естественно. Но если взять вместо `Employee` класс `QImage`, нетрудно прийти к обратному выводу.

ГЛАВА 11



Каталоги, файлы, потоки ввода-вывода, ресурсы

Кто владеет информацией — тот владеет миром.
Уинстон Черчилль

Компьютер выдает только то, что в него ввели.
Секрет успеха — не информация, но люди.
Л. Якокка

Для работы с файловой системой в библиотеке Qt имеются следующие классы:

- `QFile` — файл;
- `QTemporaryFile` — временный файл;
- `QDir` — каталог файловой системы;
- `QFileInfo` — представление информации о файлах, каталогах и ссылках;
- `QFileSystemWatcher` — слежение за файлами и каталогами на предмет их изменения;
- `QDirIterator` — итератор для последовательного перебора каталогов;
- `QByteArray` — массив байтов (по существу, содержимое файла, хранимое в оперативной памяти);
- `QBuffer` — класс, позволяющий работать с `QByteArray`, как с устройством ввода-вывода.

11.1. Чтение и запись текстового файла

Общий алгоритм работы с текстовыми файлами выглядит следующим образом:

1. Создаем файловую переменную типа `QFile`:

```
QFile file("ПолноеИмяФайла");
```

или

```
QFile file;
file.setName("ПолноеИмяФайла");
```

2. При необходимости проверяем, существует ли файл с указанным именем:

```
if ( ! file.exists() )
   // такой файл не существует
```

3. Открываем файл с помощью метода `open()`, указав режим работы:

```
 if ( ! file.open(режим) )
    // не получилось открыть файл
```

Здесь `режим` — это набор битовых флагов:

- `QIODevice::ReadOnly` — только для чтения;
- `QIODevice::WriteOnly` — только для записи;
- `QIODevice::ReadWrite` — для чтения и записи;
- `QIODevice::Append` — для добавления данных в конец файла;
- `QIODevice::Truncate` — с предварительной очисткой содержимого файла;
- `QIODevice::Text` — текстовый файл (символы конца строки преобразуются в соответствие с текущей платформой);
- `QIODevice::Unbuffered` — без промежуточного буферирования.

4. Выполняем необходимые операции по чтению или записи.
5. При необходимости вызываем принудительную запись всех изменений из промежуточного буфера в файл с помощью метода `flush()`.
6. Закрываем файл с помощью метода `close()`.

ЗАДАЧА. Прочитать содержимое текстового файла по строкам.

Решение иллюстрирует листинг 11.1.

Листинг 11.1. Чтение текстового файла

```
QFile file("name.txt");
if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
    return;
while (!file.atEnd()) {
    QString line = file.readLine();
    // . . . обработка строки line . . .
}
file.close();
```

ЗАДАЧА. Прочитать все содержимое текстового файла, сохраненного в кодировке UTF-8.

Решение иллюстрирует листинг 11.2.

Листинг 11.2. Чтение содержимого файла в кодировке UTF-8

```
QString fopenutf8(QString fullFileName) {
    QString s = "";
    QFile file(fullFileName);
    if (file.exists()) {
        if (file.open(QFile::ReadOnly | QFile::Text)) {
            s = QString::fromUtf8(file.readAll());
            file.close();
        }
    }
    return s;
}
```

Здесь с помощью метода `readAll()` в память читается сразу все содержимое текстового файла, что нецелесообразно для файлов большого размера.

ЗАДАЧА. Прочитать содержимое текстового файла по строкам, используя поток ввода `QTextStream`.

Решение иллюстрирует листинг 11.3.

Листинг 11.3. Чтение текстового файла через `QTextStream`

```
QFile file("name.txt");
if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
    return;
QTextStream in(&file);
in.setCodec("CP866"); // или любая другая кодировка
while (!in.atEnd()) {
    QString line = in.readLine();
    // . . . обработка строки line . . .
}
```

ЗАДАЧА. Записывать строки в текстовый файл, используя поток вывода `QTextStream`.

Решение иллюстрирует листинг 11.4.

Листинг 11.4. Запись строк в текстовый файл через QTextStream

```
QFile file("name.txt");
if (!file.open(QIODevice::WriteOnly | QIODevice::Text))
    return;
QTextStream out(&file);
out.setCodec("UTF-8");
out << tr("Значение x = ") << 12 << "\n";
```

11.2. Работа с двоичными данными

Для хранения в файле данных различных типов, определенных в библиотеке Qt, предназначен платформонезависимый класс потокового ввода-вывода QDataStream. Следует иметь в виду, что по мере развития библиотеки Qt формат хранения данных несколько раз менялся, поэтому перед чтением или записью в объект QDataStream необходимо установить номер версии формата потока, для этого предназначен метод setVersion(). Номер версии при чтении должен совпадать с тем, что был установлен при записи. Если поток QDataStream служит для хранения только базовых типов C++, то номер версии можно не указывать.

Ошибки чтения-записи можно контролировать с помощью метода status(), который возвращает одно из значений: QDataStream::Ok (ошибок не было), QDataStream::ReadPastEnd (при чтении достигнут конец файла, данные прочитаны не все) или QDataStream::ReadCorruptData (ошибка при чтении, данные испорчены). При любой ошибке оператор >> считывает нулевые данные, поэтому обычно проще всего прочитать весь файл целиком, а в конце проконтролировать, была ли ошибка.

Чтобы использовать поток QDataStream для хранения объектов своего собственного класса, программист должен переопределить в нем операторы >> и <<.

ЗАДАЧА. Записать в двоичный файл QFile целое число, графическое изображение QImage и объект QMap.

Решение.

1. Создаем графический объект QImage и карту QMap:

```
QImage image("рисунок.png");
QMap<QString, QColor> map;
```

2. Для примера заполняем карту некоторыми данными:

```
map.insert("red", Qt::red);
map.insert("green", Qt::green);
map.insert("blue", Qt::blue);
```

3. Создаем файл и открываем его для записи:

```
QFile file("colors.dat");
if (!file.open(QIODevice::WriteOnly)) {
    cerr << "Cannot open file for writing: "
         << qPrintable(file.errorString()) << endl;
    return;
}
```

4. Создаем поток вывода `QDataStream` и устанавливаем номер версии, определяющий формат хранимых данных:

```
QDataStream out(&file);
out.setVersion(QDataStream::Qt_4_3);
```

5. Выполняем запись целого числа, графического изображения и карты:

```
out << quint32(0x12345678) << image << map;
```

6. Закрываем файл:

```
file.close();
```

ЗАДАЧА. Прочитать целое число, графическое изображение `QImage` и карту `QMap` из двоичного файла, созданного в предыдущем примере.

Решение иллюстрирует листинг 11.5.

Листинг 11.5. Чтение из двоичного файла

```
quint32 n;
QImage image;
QMap<QString, QColor> map;
QFile file("facts.dat");
if (!file.open(QIODevice::ReadOnly)) {
    cerr << "Cannot open file for reading: "
         << qPrintable(file.errorString()) << endl;
    return;
}
QDataStream in(&file);
in.setVersion(QDataStream::Qt_4_3);
in >> n >> image >> map;
file.close();
```

ЗАДАЧА. Записать в двоичный файл строку, число, дату и графическое изображение. Контролировать возможные ошибки записи.

Решение иллюстрирует листинг 11.6.

Листинг 11.6. Запись в двоичный файл

```
QFile file("name.dat");
if (file.open(QIODevice::WriteOnly)){
   QDataStream out(&file);
   out.setVersion(QDataStream::Qt_4_3);
   out << tr("Сегодня") << QDate(2007, 12, 30);
   out << tr("До Нового года осталось")
       << qint32(29) << tr("часов");
   out << QImage("image.png");
   if (out.status() != QDataStream::Ok)
      // ошибка при записи в файл
}
file.close();
```

ЗАДАЧА. Прочитать из двоичного файла строку, число, дату и графическое изображение. Контролировать возможные ошибки при чтении.

Решение иллюстрирует листинг 11.7.

Листинг 11.7. Чтение с контролем ошибок

```
QFile file("name.dat");
if (file.open(QIODevice::ReadOnly)) {
   QDataStream in(&file);
   in.setVersion(QDataStream::Qt_4_3);
   QString s1, s2, s3;
   QDate d;
   qint32 n;
   QImage img;
   in >> s1 >> d >> s2 >> n >> s3 >> img;
   if (out.status() != QDataStream::Ok)
      // ошибка при чтении из файла
      // описание ошибки — out.errorString()
}
file.close();
```

ЗАДАЧА. Разработать сериализуемый класс `MyClass`, объекты которого можно записывать в `QDataStream` и считывать из него.

Решение иллюстрирует листинг 11.8.

Листинг 11.8. Пример сериализуемого класса

```
class MyClass {
public:
   QString text;
   QColor color;
};
QDataStream &operator<<( QDataStream &out, const ColorText &data ){
   out << data.text << data.color;
   return stream;
}
QDataStream &operator>>( QDataStream &out, ColorText &data ){
   out >> data.text;
   out >> data.color;
   return stream;
}
```

Теперь можно создать, например, список объектов своего класса и записать их в файл (листинг 11.9).

Листинг 11.9. Пример записи объектов в файл

```
QList<MyClass> list;
MyClass data;
data.text = tr("Красный");
data.color = Qt::red;
list << data;
// .....
QFile file( "filename.dat" );
if( !file.open( QIODevice::WriteOnly ) )
   return;
QDataStream stream( &file );
stream.setVersion( QDataStream::Qt_4_3 );
stream << list;
file.close();
```

А затем прочитать записанные данные (листинг 11.10).

Листинг 11.10. Пример чтения данных

```
QList<MyClass> list;
QFile file( "test.dat" );
```

```
if( !file.open( QIODevice::ReadOnly ) )
return;
QDataStream stream( &file );
stream.setVersion( QDataStream::Qt_4_3 );
stream >> list;
file.close();
QStringList sl;
foreach( ColorText data, list )
   sl  << data.text << "("
       << data.color.red() << ","
       << data.color.green() << ","
       << data.color.blue() << ")";
```

11.3. Каталоги и свойства файлов

ЗАДАЧА. Получить список всех дисковых устройств и каталогов внешнего уровня на каждом из них.

Решение иллюстрирует листинг 11.11.

Листинг 11.11. Получение списка дисковых устройств и каталогов

```
foreach( QFileInfo drive, QDir::drives() ) {
   cout << "Drive: " << drive.absolutePath();
   QDir dir = drive.dir();
   dir.setFilter( QDir::Dirs );
   foreach( QFileInfo rootDir, dir.entryInfoList() )
       cout << " " << rootDir.fileName();
}
```

В системе Windows мы получим названия дисков C:/, D:/ и всех папок, расположенных в корневом каталоге каждого диска. В Linux имеется только одна корневая файловая система `root`, мы увидим названия всех каталогов верхнего уровня.

Ключевое слово `foreach` — это на самом деле макрос, определенный в библиотеке Qt, который позволяет организовать цикл перебора всех элементов заданного контейнера. Первый аргумент — переменная для хранения текущего элемента, второй — сам контейнер.

11.4. Временные файлы

ЗАДАЧА. Создать временный файл для хранения некоторых промежуточных данных.

Решение. Для работы с временными файлами предназначен класс `QTemporaryFile`:

```
QTemporaryFile file;
if ( file.open() ) {
     // . . .
}
```

При создании для временного файла автоматически генерируется уникальное имя, его можно узнать с помощью метода `QTemporaryFile::fileName()`. После уничтожения файловой переменной (например, после выхода из функции) сам файл также уничтожается. Каталог для хранения временных файлов зависит от текущей платформы, его можно выяснить с помощью статического метода `QDir::tempPath()`. Как правило, в системах Linux это каталог `/tmp`, в Windows — значение переменной окружения `TEMP` или `TMP`. Сам каталог для временных файлов, т. е. объект типа `QDir`, можно получить с помощью статического метода `QDir::temp()`.

11.5. Каталог приложения

ЗАДАЧА. Узнать, в каком каталоге находится исполняемый файл приложения.

Решение. Можно использовать статический метод

```
QString QCoreApplication::applicationDirPath()
```

Пример:

```
QString s = qApp->applicationDirPath();
```

При этом следует помнить, что результат может оказаться неверным, если в самом приложении уже был изменен текущий каталог.

Если требуется знать также и имя исполняемого файла, то нужно вызвать метод `QCoreApplication::applicationFilePath()`.

11.6. Копирование файлов

ЗАДАЧА. Выполнить копирование файла.

Решение. Скопировать файлы, расположенные *на локальном компьютере* или *на сетевых ресурсах* проще всего с помощью метода `QFile::copy()` (листинг 11.12).

Листинг 11.12. Копирование файлов

```
 1  #include <QtCore>
 2  #include <QtDebug>
 3
 4  bool copy_file(const QString& inFileName,
 5                 const QString& outFileName) {
 6     QFile inFile(inFileName);
 7     if(!inFile.open(QIODevice::ReadOnly)){
 8        qDebug() << "Can not open source file";
 9        return false;
10     }
11
12     QFile outFile(outFileName);
13     if(QFile::exists(outFileName)){
14         qDebug() << "Target file exists, deleting...";
15         QFile::remove(outFileName);
16     }
17
18     bool f = inFile.copy(outFileName);
19     if ( f )
20         qDebug() << "Ok";
21     else
22         qDebug() << "Failed";
23
24     return f;
25  }
26  //---------------------------------
27  int main(int argc, char *argv[]){
28     QCoreApplication app(argc, argv);
29     if (argc < 3)
30         return -1;
31
32     return copy_file( argv[1], argv[2] ) ? 0 : -2;
33  }
```

Поскольку функция `QFile::copy()` никогда не затирает уже существующие файлы (возвращая в этом случае значение `false`), нам приходится удалить целевой файл вручную (15). Заметьте, после копирования мы не закрываем исходный файл, поскольку это делает сама функция `copy()`.

Перед компиляцией в файл *.pro необходимо добавить строку

```
CONFIG += console
```

Запускают программу из командной строки, указав при этом два параметра: полное имя исходного файла и полное имя файла (нового или существующего), в который будет производиться копирование. Любой из файлов может располагаться на общих ресурсах локальной сети, например, в системе Windows допустимы команды такого вида:

```
ИмяПрограммы  \\server1\public\file1.txt  \\server2\share\file2.txt
```

Читателю предлагается самостоятельно доработать программу, чтобы исходный и целевой файлы выбирались с помощью методов `QFileDialog::getOpenFileName()` и `QFileDialog::getSaveFileName()`. Кроме того, можно реализовать копирование вручную, через чтение/запись отдельных блоков файла и дополнить все это индикатором прогресса. Что касается копирования каталога, то эта задача сводится к копированию всех его файлов с предварительным созданием целевого каталога и, возможно, всех его подкаталогов. Для создания/удаления каталогов и перебора всех их элементов можно использовать класс `QDir`. Другой вариант — вызов соответствующей команды операционной системы.

11.7. Хранение ресурсов в программе

ЗАДАЧА. Обеспечить хранение графических изображений, используемых приложением, и других ресурсов в самом исполняемом файле.

Решение. Графические изображения, в том числе и те, что отображаются на панелях инструментов, можно хранить в отдельных файлах, но будет гораздо удобнее, если их разместить в виде ресурсов в самом исполняемом файле нашего приложения. Для этого потребуется выполнить последовательность действий.

1. Создать текстовый файл описания ресурсов с расширением qrc (листинг 11.13).

Листинг 11.13. Пример qrc-файла

```
<!DOCTYPE RCC><RCC version="1.0">
  <qresource>
    <file alias="create.png">img/new.png</file>
    <file>img/cut.png</file>
    <file>img/copy.png</file>
    <file>img/paste.png</file>
  </qresource>
</RCC>
```

Здесь для примера считается, что мы хотим использовать четыре значка для пунктов меню: **Новый** (New), **Вырезать** (Cut), **Копировать** (Copy) и **Вставить** (Paste) и что файлы с соответствующими изображениями находятся в каталоге img (все пути указываются относительно того каталога, в котором находится сам файл .qrc). Для первой пиктограммы задан псевдоним (alias). В самой программе перечисленные ресурсы будут доступны по следующим именам: `:/create.png`, `:/img/cut.png`, `:/img/copy.png` и т. д. (двоеточие перед именем означает, что мы обращаемся к скомпилированным ресурсам, а не к файлам, из которых они скомпилированы). Если требуется хранить ресурсы в иерархическом виде, отличающемся от структуры каталогов до компиляции, то используются префиксы (листинг 11.14).

Листинг 11.14. Префиксы в именах ресурсов

```
<qresource prefix="/file">
  <file alias="create.png">>img/new.png</file>
</qresource>
<qresource prefix="/edit">
  <file>img/cut.png</file>
  <file>img/copy.png</file>
  <file>img/paste.png</file>
</qresource>
```

Здесь для примера считается, что все пиктограммы до компиляции хранились в одной папке img, но в самой программе мы будем обращаться к ним с учетом префикса: `:/file/create.png`, `:/edit/img/cut.png`, `:/edit/img/copy.png` и т. д.

2. Включить в текст pro-файла строку

```
RESOURCES += ИмяФайла.qrc
```

3. В самой программе использовать нужный ресурс, например, при создании действия:

```
actCopy = new QAction(QIcon(":/img/copy.png"), tr("&Copy"), this);
```

После компиляции во время выполнения программы каталог img с файлами-пиктограммками уже не нужен.

11.8. Хранение скомпилированных ресурсов в отдельном двоичном файле

ЗАДАЧА. Скомпилированные ресурсы требуется хранить в отдельном файле.

Решение. В некоторых случаях разумнее хранить ресурсы в отдельных файлах, а не в самой программе. Для этого соответствующий файл *.qrc упоминать в pro-файле не нужно, но придется скомпилировать его самостоятельно с помощью утилиты `rcc`:

```
rcc -binary ИмяФайла.qrc -o ИмяФайла.rcc
```

Перед обращением к ресурсам соответствующий rcc-файл необходимо загрузить вручную:

```
QResource::registerResource("/Путь/ИмяФайла.rcc", "Префикс");
```

Функция возвращает `true`, если файл ресурсов загружен удачно. Префикс можно не указывать. По окончании работы с ресурсами файл можно выгрузить с помощью функции `unregisterResource()` с теми же параметрами.

11.9. Значок приложения

ЗАДАЧА. Добавить значок приложения, отображаемый в верхнем левом углу каждого окна.

Решение. Последовательность действий в системе Microsoft Windows:

1. Подготовить файл значка в *.ico (например, с помощью утилиты `png2ico`);
2. Создать текстовый файл с расширением rc, содержащий одну строку:

   ```
   IDI_ICON1 ICON DISCARDABLE "ИмяФайла.ico"
   ```

3. К тексту pro-файла добавить строку

   ```
   RC_FILE = ИмяФайла.rc
   ```

Замечание

Значок приложения автоматически будет повторяться в верхнем левом углу всех окон верхнего уровня (т. е. не имеющих родителя). Для отдельных окон его можно задать вручную с помощью метода
`setWindowIcon( QIcon( ":/путь/ИмяФайла.png" ) )`

В Linux решение зависит от установленного менеджера рабочего стола, поэтому обсуждение этого вопроса выходит за рамки данной книги.

11.10. Диалог выбора файла

ЗАДАЧА. Отобразить диалог для выбора существующего или ввода имени нового файла для сохранения некоторой информации.

Решение иллюстрирует листинг 11.15.

Листинг 11.15. Реализация диалога выбора файла

```
QString fileName = QFileDialog::getSaveFileName(
    this,               // виджет-родитель
    tr("Выбор файла"),  // заголовок окна диалога
    QString(),          // начальный каталог — текущий
    // набор масок для файлов (через ';;'):
    tr("Документы (*.txt *.doc);;Все файлы (*.*)) );
if ( ! fileName.isEmpty() ) {
   // сохраняем информацию в файл
}else{
   // пользователь отказался от выбора файла
}
```

Здесь мы опустили два необязательных параметра функции `getSaveFileName()`: указатель на строку `QString`, в которой можно задать текущую маску (она, возможно, будет изменена пользователем), и набор флагов, определяющих режимы отображения информации в диалоге:

- `QFileDialog::ShowDirsOnly` — показывать только каталоги;
- `QFileDialog::DontResolveSymlinks` — не разрешать символические ссылки;
- `QFileDialog::DontConfirmOverwrite` — не требовать дополнительного подтверждения, можно ли затирать существующий файл;
- `QFileDialog::DontUseNativeDialog` — не использовать "родной" для текущей платформы диалог сохранения (он задан по умолчанию в системах Windows и Mac OS).

Если во введенном пользователем имени файла отсутствует расширение, то можно добавить некоторое расширение по умолчанию:

```
QString suf = QFileInfo(fileName).suffix();
if (suf.isEmpty())
    fileName.append(".txt");
```

11.11. Сжатие информации

ЗАДАЧА. Сжать (заархивировать) указанный файл, а затем распаковать его и сравнить результат с исходным файлом.

Решение. Для сжатия информации можно применить глобальную функцию `qCompress()`, для обратного преобразования — функцию `qUncompress()` (листинг 11.16).

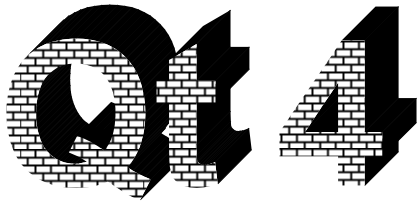
Листинг 11.16. Архивирование и разархивирование одиночного файла

```
#include <iostream>
#include <QtCore>
using namespace std;

int main( int argc, char **argv ) {
    QCoreApplication app(argc, argv);
    if (argc < 1){
        cout << "filename?" << endl;
        return -1;
    }
    QString filename(argv[1]);
    QFile file(filename);
    if (! file.open(QIODevice::ReadOnly) ){
        cout << "Cannot open file " << filename << endl;
        return -2;
    }
    QByteArray arr = file.readAll();
    file.close();

    QString file2name = filename + ".gz";
    file.setFileName(file2name);
    if (!file.open(QIODevice::WriteOnly | QIODevice::Truncate)){
        cout << "Cannot open file for writing" << endl;
        return -3;
    }
    QByteArray arr2 = qCompress(arr); // сжимаем
    file.write(arr2);                 // записываем в файл
    file.close();

    // Теперь откроем сжатый файл и восстановим информацию:
    if (!file.open(QIODevice::ReadOnly)){
        cout << "Cannot open compressed file" << endl;
        return -5;
    }
    arr = file.readAll();  // читаем все содержимое
    file.close();

    file.setFileName("uncompressed_"+filename);
    if (!file.open(QIODevice::WriteOnly | QIODevice::Truncate)){
        cout << "Cannot open file for writing" << endl;
        return -6;
    }
```

```
    arr2 = qUncompress(arr);        // разархивируем
    if ( file.write(arr2) == -1){  // записываем в файл
        cout << "Cannot write uncompressed file" << endl;
        return -7;
    }
    file.close();
    return 0;
}
```

При запуске этой программы укажите в командной строке имя архивируемого (лучше текстового) файла, например, text1.txt, а после ее выполнения сравните содержимое файла text1.txt с результатом разархивирования, которое записывается в файл uncompressed_text1.txt.

ГЛАВА 12



Приложения SDI и MDI

Интерес к внутренней сути и презрение к внешнему виду — таковы должны быть отношения между разумным человеком и его книгами.
Филипп Дормер Стенхоп Честерфилд

Напомним, что приложение SDI (*Single Document Interface* — однодокументный интерфейс) в каждый момент времени может работать только с одним документом (как, например, стандартные для системы Windows графический редактор Paint и текстовый редактор Блокнот). Напротив, приложение MDI (*Multiple Document Interface* — многодокументный интерфейс) предоставляет рабочую область, в которой одновременно могут сосуществовать несколько открытых документов (например, широко распространенная бухгалтерская программа 1С:Предприятие).

12.1. Пример приложения SDI: текстовый редактор

ЗАДАЧА. Разработать прототип простого текстового редактора с интерфейсом SDI.

Решение. В любом приложении Qt можно выделить стандартные шаги, связанные с построением пользовательского интерфейса.

- Разработка класса главного окна приложения:
  - определение действий `QAction`;
  - построение главного меню;
  - формирование панелей инструментов;
  - создание строки состояния (возможно, с вложенными в нее виджетами);

- создание центрального виджета главного окна;
- связывание действий с методами-слотами и реализация слотов.

❒ Создание пиктограмм для действий (с помощью графического редактора).

❒ Создание текстового файла описания ресурсов *.qrc.

В листингах 12.1–12.5 приведено решение поставленной задачи, а на рис. 12.1 показан внешний вид окна приложения. Чуть позже мы используем данный пример для описания метода *локализации* приложений, поэтому все надписи на элементах интерфейса пока пишем по-английски.

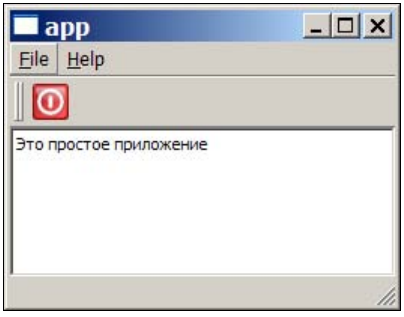


Рис. 12.1. Простое приложение SDI со строкой меню, панелью инструментов, редактором текста `QTextEdit` и строкой состояния

Листинг 12.1. Файл chap12/app1/mainwin.h

```
#ifndef MAINWIN_H
#define MAINWIN_H

#include <QMainWindow>

class QAction;
class QMenu;
class QTextEdit;

class MainWin : public QMainWindow {
    Q_OBJECT
public:
    MainWin();      // Конструктор главного окна
private slots:      // слоты:
    void about();   // слот "О программе"
private:
    void createActions();   // Создать действия
```

```
18     void createMenus();      // Создать меню
19     void createToolBars();   // Создать панели инструментов
20     void createStatusBar(); // Создать строку состояния
21
22     QTextEdit *textEdit;     // Виджет-редактор текста
23
24     QMenu *fileMenu;         // Меню "Файл"
25     QMenu *helpMenu;         // Меню "Помощь"
26     QToolBar *fileToolBar;   // Панель инструментов
27
28     QAction *exitAct;        // Действие "Выход"
29     QAction *aboutAct;       // Действие "О программе"
30     QAction *aboutQtAct;     // Действие "О библиотеке Qt"
31 };
32
33 #endif // MAINWIN_H
```

Листинг 12.2. Файл chap12/app1/mainwin.cpp

```
35 #include <QtGui>
36 #include "mainwin.h"
37
38 MainWin::MainWin(){    // Конструктор главного окна
39     textEdit = new QTextEdit;
40     setCentralWidget(textEdit);
41
42     createActions();
43     createMenus();
44     createToolBars();
45     createStatusBar();
46 }
47
48 void MainWin::about(){   // Слот "О программе"
49    QMessageBox::about(this, tr("About"),
50       tr("The program demonstrates how to "
51          "write GUI applications using Qt, with a menu bar, "
52          "toolbars, and a status bar."));
53 }
54
55 void MainWin::createActions(){   // Создание действий
56     exitAct = new QAction( QIcon(":/images/fileexit.png"),
                              tr("E&xit"), this);
57     exitAct->setShortcut(tr("Ctrl+Q"));
```

```
58      exitAct->setStatusTip(tr("Exit the application"));
59      connect(exitAct, SIGNAL(triggered()), this, SLOT(close()));
60
61      aboutAct = new QAction(tr("&About"), this);
62      aboutAct->setStatusTip(
            tr("Show the application's About box"));
63      connect(aboutAct, SIGNAL(triggered()),
64              this, SLOT(about()));
65      aboutQtAct = new QAction(tr("About &Qt"), this);
66      aboutQtAct->setStatusTip(
            tr("Show the Qt library's About box"));
67      connect( aboutQtAct, SIGNAL(triggered()),
                 qApp, SLOT(aboutQt()));
68  }
69
70  void MainWin::createMenus(){     // Создание главного меню
71      fileMenu = menuBar()->addMenu(tr("&File"));
72      fileMenu->addAction(exitAct);
73
74      menuBar()->addSeparator();
75
76      helpMenu = menuBar()->addMenu(tr("&Help"));
77      helpMenu->addAction(aboutAct);
78      helpMenu->addAction(aboutQtAct);
79  }
80
81  void MainWin::createToolBars(){    //
82      fileToolBar = addToolBar(tr("File"));
83      fileToolBar->addAction(exitAct);
84  }
85
86  void MainWin::createStatusBar(){
87      statusBar()->showMessage(tr("Ready"));
88  }
```

Листинг 12.3. Главная программа (файл chap12/app1/main.cpp)

```
90  #include <QApplication>
91  #include "mainwin.h"
92
93  int main(int argc, char *argv[]){
94      Q_INIT_RESOURCE(app);
95
```

```
 96     QApplication app(argc, argv);
 97     MainWin mainWin;
 98     mainWin.show();
 99     return app.exec();
100 }
```

Листинг 12.4. Файл описания ресурсов chap12/app1/app.qrc

```
101  <!DOCTYPE RCC><RCC version="1.0">
102     <qresource>
103         <file>images/fileexit.png</file>
104     </qresource>
105  </RCC>
```

Листинг 12.5. Файл описания проекта app.pro

```
106   HEADERS       = mainwin.h
107   SOURCES       = main.cpp \
108                   mainwin.cpp
109   RESOURCES     = app.qrc
```

Пояснения к программе:

- (1, 2, 33) — условная директива препроцессора позволяет избежать повторного включения заголовочного файла;
- (4–8) — вместо данных строк можно было бы просто подключить модуль `<QtGui>`, но тогда увеличится время компиляции. Разумеется, в нашем простом примере выигрыш несущественный, но этот же прием можно использовать в больших проектах;
- (10–31, 38–88) — класс главного окна приложения на основе стандартного `QMainWindow`;
- (13, 38–46) — в конструкторе главного окна создаем центральный виджет — область для ввода текста `QTextEdit`, действия `QAction` (42), главное меню (43), панели инструментов (44) и строку состояния (45);
- (15, 48-53, 67) — слот `about()` вызывается при выполнении пункта меню **Help** | **About** (выводит информацию о нашей программе);
- (17, 42, 55–68) — в функции `createActions()` создаются все действия главного окна, которые затем станут пунктами меню и кнопками панелей инструментов, и определяются связи между действиями и слотами (59, 63, 67);

- (18, 43, 70–79) — в функции `createMenus()` формируются пункты меню;
- (19, 44, 81–84) — в функции `createToolBars()` создается панель инструментов с одной кнопкой **Выход**. Чуть позже мы добавим другие кнопки и создадим еще одну панель;
- (20, 45, 86–88) — в функции `createStatusBar()` создается строка состояния, в которую будут выводиться подсказки по текущим пунктам меню и кнопкам панели инструментов. Сразу после запуска программы в строку состояния выводится сообщение `Ready` (готов);
- (90–100) — в главной программе формируется и выводится на экран главное окно, после чего запускается цикл обработки событий (99);
- (94) — макрос `Q_INIT_RESOURCE(Имя_файла_qrc_без_расширения)` инициализирует ресурсы, указанные в файле *.qrc. Обычно инициализация происходит автоматически при запуске программы, но на некоторых платформах, если ресурсы хранятся в статической библиотеке, требуется сделать это явно;
- (101–105) — необходимо создать текстовый файл описания ресурсов *.qrc, пока он содержит путь к единственному используемому в программе графическому файлу fileexit.png, в котором хранится значок для действия **Выход**;
- (106–109) — файл описания проекта *.pro формируется автоматически в результате выполнения команды `qmake -project`, но его можно создать и вручную (или, по крайней мере, дописать строку `RESOURCES`, если файл *.qrc был создан уже после генерации файла *.pro).

Компилируется проект как обычно. Например, при использовании Microsoft Visual C++ указывают последовательность команд:

```
qmake -project
qmake
nmake release
```

Пока наше приложение не умеет ни сохранять, ни открывать файлы, но в следующих главах мы добавим эти и многие другие возможности.

12.2. Открытие и сохранение файлов

ЗАДАЧА. Доработать предыдущую программу так, чтобы появилась возможность сохранять и открывать текстовые файлы.

Решение. В раздел `private slots` класса `MainWin` добавим заголовки методов-слотов (листинг 12.6).

Листинг 12.6

```
private slots:
    void newFile();  // Создать новый файл
    void open();     // Открыть файл
    bool save();     // Сохранить файл
    bool saveAs();   // Сохранить файл под новым именем
    void documentWasModified(); // Установить признак "изменен"
```

В раздел `private` добавим содержимое листинга 12.7.

Листинг 12.7

```
 bool maybeSave();   // Требуется ли сохранение
 void loadFile(const QString &fileName);  // Чтение из файла
 bool saveFile(const QString &fileName);  // Запись в файл
 void setCurrentFile(const QString &fileName); // Запомнить имя файла
 QString strippedName(
     const QString &fullFileName); // Вернуть имя файла без каталогов

 QString curFile;  // Здесь будем хранить текущее имя файла
 QAction *newAct;  // Создать новый файл
 QAction *openAct; // Открыть документ из файла
 QAction *saveAct; // Сохранить документ в файл
 QAction *saveAsAct; // Сохранить под новым именем
```

В раздел `protected` добавим заголовок виртуального метода `closeEvent()`, который автоматически вызывается при закрытии окна:

```
protected:
    void closeEvent(QCloseEvent *event);
```

В конструкторе класса `MainWin` свяжем сигнал `contentsChanged`, автоматически генерируемый при изменении содержимого поля `QTextEdit`, со слотом `documentWasModified()`:

```
connect(textEdit->document(), SIGNAL(contentsChanged()),
        this, SLOT(documentWasModified()));
```

и не забудем установить начальное имя файла (пустую строку):

```
setCurrentFile("");
```

В методе `createActions()` добавим новые действия (листинг 12.8).

Листинг 12.8

```
newAct = new QAction( QIcon(":/images/filenew.png"),
                      tr("&New"), this);
newAct->setShortcut(tr("Ctrl+N"));
newAct->setStatusTip(tr("Create a new file"));
connect(newAct, SIGNAL(triggered()), this, SLOT(newFile()));

openAct = new QAction(QIcon(":/images/fileopen.png"),
                      tr("&Open..."), this);
openAct->setShortcut(tr("Ctrl+O"));
openAct->setStatusTip(tr("Open an existing file"));
connect(openAct, SIGNAL(triggered()), this, SLOT(open()));

saveAct = new QAction(QIcon(":/images/filesave.png"),
                      tr("&Save"), this);
saveAct->setShortcut(tr("Ctrl+S"));
saveAct->setStatusTip(tr("Save the document to disk"));
connect(saveAct, SIGNAL(triggered()), this, SLOT(save()));

saveAsAct = new QAction(tr("Save &As..."), this);
saveAsAct->setStatusTip(tr("Save the document under a new name"));
connect(saveAsAct, SIGNAL(triggered()), this, SLOT(saveAs()));
```

В методе `createMenus()` — новые пункты меню (листинг 12.9), а в методе `createToolBars()` — новые кнопки (листинг 12.10).

Листинг 12.9

```
fileMenu->addAction(newAct);
fileMenu->addAction(openAct);
fileMenu->addAction(saveAct);
fileMenu->addAction(saveAsAct);
```

Листинг 12.10

```
fileToolBar->addAction(newAct);
fileToolBar->addAction(openAct);
fileToolBar->addAction(saveAct);
fileToolBar->addSeparator();
```

Реализацию метода `setCurrentFile()` иллюстрирует листинг 12.11.

Листинг 12.11

```
void MainWin::setCurrentFile(const QString &fileName){
    curFile = fileName;

    // Сбросим признак "Документ изменен":
    textEdit->document()->setModified(false);
    setWindowModified(false);

    QString shownName;  // Имя, отображаемое в заголовке окна
    if (curFile.isEmpty())
        shownName = tr("untitled.txt"); // Файл пока без имени
    else
        shownName = strippedName(curFile);  // Имя без каталогов

    setWindowTitle(tr(   // Установить заголовок окна
         "%1[*] — %2").arg(shownName).arg(tr("Application") ) );
}
```

Все виджеты — наследники класса `QWidget` имеют поле `windowModified`, которое хранит признак изменения содержимого, отображаемого данным виджетом. С помощью метода `setWindowModified()` можно установить нужное значение этого признака (`true` или `false`).

Метод `strippedName()` возвращает собственно имя файла без названий каталогов:

```
QString MainWin::strippedName(const QString &fullFileName){
   return QFileInfo(fullFileName).fileName();
}
```

Реализация слота `documentWasModified()`:

```
void MainWin::documentWasModified(){
    setWindowModified(textEdit->document()->isModified());
}
```

Метод `QTextEdit::document()` возвращает указатель на объект `QTextDocument`, в котором хранится содержимое виджета `QTextEdit`. Метод `QTextDocument::isModified()` возвращает значение признака изменения документа. Данное значение мы просто копируем в аналогичный признак объекта главного окна.

Реализацию метода-слота для сохранения документа иллюстрирует листинг 12.12.

Листинг 12.12

```
bool MainWin::save(){
   if ( curFile.isEmpty() ) {
       return saveAs();
   } else {
      return saveFile(curFile);
   }
}
```

Если имя файла пока не задано, то вызывается слот "Сохранить как" (листинг 12.13).

Листинг 12.13

```
bool MainWin::saveAs(){
   QString fileName = QFileDialog::getSaveFileName(this);
   if ( fileName.isEmpty() )
       return false;

   return saveFile(fileName);
}
```

Статический метод `QFileDialog::getSaveFileName()` выводит на экран диалог выбора файла и возвращает полное имя выбранного файла или пустую строку, если пользователь отказался от выбора. Само сохранение документа в файл происходит в методе `saveFile()` (листинг 12.14).

Листинг 12.14

```
bool MainWin::saveFile(const QString &fileName){
   QFile file(fileName);
   if (!file.open(QFile::WriteOnly | QFile::Text)) {
      QMessageBox::warning(this, tr("Application"),
                           tr("Cannot write file %1:\n%2.")
                           .arg(fileName)
                           .arg(file.errorString()));
      return false;
   }
   QTextStream out(&file);
   QApplication::setOverrideCursor(Qt::WaitCursor);
   out << textEdit->toPlainText();
   QApplication::restoreOverrideCursor();
```

```
    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File saved"), 2000);
    return true;
}
```

В методе `maybeSave()` мы проверяем значение признака изменения документа. Если он установлен, то просим пользователя подтвердить необходимость сохранения документа в файл. При утвердительном ответе вызываем метод `MainWin::save()` и возвращаем признак успешности завершения этой операции (`true` или `false`), при отрицательном — просто возвращаем логическое значение `false`: (листинг 12.15).

Листинг 12.15

```
bool MainWin::maybeSave(){
    if (textEdit->document()->isModified()) {
         QMessageBox::StandardButton ret;
      ret = QMessageBox::warning(this, tr("Application"),
                tr("The document has been modified.\n"
                   "Do you want to save your changes?"),
                   QMessageBox::Save | QMessageBox::Discard |
                   QMessageBox::Cancel);
       if (ret == QMessageBox::Save)
          return save(); // Пытаемся сохранить документ
       else if (ret == QMessageBox::Cancel)
          return false;
    }
    return true;
}
```

Реализацию виртуального метода `closeEvent()`, который автоматически вызывается при попытке закрыть окно, независимо от того, нажал ли пользователь крестик в заголовке окна, или был вызван слот `close()`, иллюстрирует листинг 12.16.

Листинг 12.16

```
void MainWin::closeEvent(QCloseEvent *event){
    if ( maybeSave() ) {
       event->accept();  // Продолжить закрытие окна
    } else {
       event->ignore();  // Игнорировать событие
    }
}
```

Если документ не удалось сохранить или пользователь отказался от его сохранения, то закрытие окна отменяется.

В слоте `newFile()` сначала проверяется необходимость и успешность сохранения текущего документа (листинг 12.17).

Листинг 12.17

```
void MainWin::newFile(){
    if (maybeSave()) {
        textEdit->clear();
        setCurrentFile("");
    }
}
```

Аналогично для метода `open()` (листинг 12.18).

Листинг 12.18

```
void MainWin::open(){
   if ( maybeSave() ) {
      QString fileName = QFileDialog::getOpenFileName(this);
      if ( !fileName.isEmpty() )
          loadFile(fileName);
   }
}
```

Загружает документ из файла метод `loadFile()` (листинг 12.19).

Листинг 12.19

```
void MainWin::loadFile(const QString &fileName){
   QFile file(fileName);
   if (!file.open(QFile::ReadOnly | QFile::Text)) {
       QMessageBox::warning(this, tr("Application"),
                            tr("Cannot read file %1:\n%2.")
                            .arg(fileName)
                            .arg(file.errorString()));
       return;
    }
    QTextStream in(&file);
    QApplication::setOverrideCursor(Qt::WaitCursor);
    textEdit->setPlainText(in.readAll());
    QApplication::restoreOverrideCursor();
```

```
    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File loaded"), 2000);
}
```

После этого нужно подготовить пиктограммы для новых действий, поместить их в каталог images и добавить соответствующие строки в файл описания ресурсов app.qrc:

```
<file>images/filenew.png</file>
<file>images/fileopen.png</file>
<file>images/filesave.png</file>
```

12.3. Команды редактирования

ЗАДАЧА. Добавить в проект из *разд. 12.2* команды меню и кнопки для редактирования текста.

Решение. Хотя виджет `QTextEdit` поддерживает контекстное меню и комбинации клавиш <Ctrl>+<C>, <Ctrl>+<X>, <Ctrl>+<V> для копирования, вырезания и вставки текстовых фрагментов, хорошо бы добавить к главному меню программы новую группу **Edit** (Правка) и создать дополнительную панель инструментов для соответствующих команд.

Для этого в разделе `private` класса `MainWin` допишем определения новых действий:

```
QAction *cutAct;    // Вырезать
QAction *copyAct;   // Копировать
QAction *pasteAct;  // Вставить
```

и объявим еще одну панель инструментов:

```
QToolBar *editToolBar;
```

В функции `createActions()` создадим нужные действия (листинг 12.20).

Листинг 12.20

```
cutAct = new QAction(QIcon(":/images/editcut.png"),
                     tr("Cu&t"), this);
cutAct->setShortcut(tr("Ctrl+X"));
cutAct->setStatusTip(tr(
     "Cut the current selection's contents to the clipboard"));
connect(cutAct, SIGNAL(triggered()), textEdit, SLOT(cut()));

copyAct = new QAction(QIcon(":/images/editcopy.png"),
                      tr("&Copy"), this);
```

```
copyAct->setShortcut(tr("Ctrl+C"));
copyAct->setStatusTip(tr(
     "Copy the current selection's contents to the clipboard"));
connect(copyAct, SIGNAL(triggered()), textEdit, SLOT(copy()));

pasteAct = new QAction(QIcon(":/images/editpaste.png"),
                       tr("&Paste"), this);
pasteAct->setShortcut(tr("Ctrl+V"));
pasteAct->setStatusTip(tr(
     "Paste the clipboard's contents into the current selection"));
connect(pasteAct, SIGNAL(triggered()), textEdit, SLOT(paste()));
```

Действия "Вырезать" и "Копировать" должны быть доступны, только если выделен какой-либо фрагмент текста. К счастью, виджет `QTextEdit` генерирует сигнал `copyAvailable(bool)`, параметр которого равен `true`, если имеется выделенный фрагмент. Этот сигнал мы используем для разрешения действий с помощью метода `QAction::setEnabled()` (листинг 12.21).

Листинг 12.21

```
cutAct->setEnabled(false);
copyAct->setEnabled(false);
connect(textEdit, SIGNAL(copyAvailable(bool)),
        cutAct, SLOT(setEnabled(bool)));
connect(textEdit, SIGNAL(copyAvailable(bool)),
        copyAct, SLOT(setEnabled(bool)));
```

Теперь в методе `createMenus()` можно добавить группу меню **Edit** (Правка) (листинг 12.22), а в методе `createToolBars()` — создать еще одну панель инструментов (листинг 12.23).

Листинг 12.22

```
editMenu = menuBar()->addMenu(tr("&Edit"));
editMenu->addAction(cutAct);
editMenu->addAction(copyAct);
editMenu->addAction(pasteAct);
```

Листинг 12.23

```
editToolBar = addToolBar(tr("Edit"));
editToolBar->addAction(cutAct);
editToolBar->addAction(copyAct);
editToolBar->addAction(pasteAct);
```

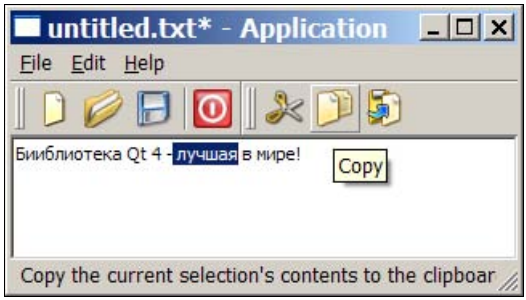


Рис. 12.2. Текстовый редактор

В результате получаем главное окно, показанное на рис. 12.2.

Полные тексты всех файлов проекта можно найти на прилагаемом диске в каталоге `chap12/app2`. В *разд. 13.8* мы добавим к нашему проекту возможность вывода на принтер и в файл PDF.

12.4. Пример приложения MDI

Приложение MDI (*Multiple Document Interface*) позволяет пользователю открывать сразу несколько документов, отображая каждое из них в своем окне. Эти окна управляются общей рабочей областью (рис. 12.3).

ЗАДАЧА. Переработать листинги *разд. 12.1*, реализовав MDI-приложение.

Решение приведено в листингах 12.24–12.30.

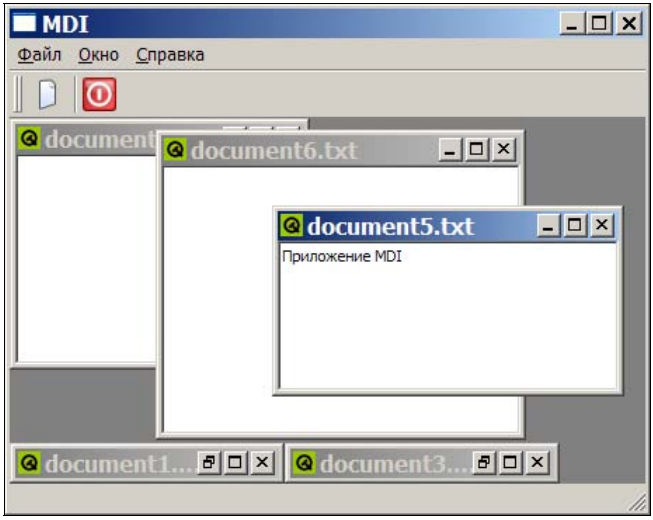


Рис. 12.3. Приложение с интерфейсом MDI

Листинг 12.24. Файл chap12/mdi/mainwin.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class QAction;
class QMenu;
class QMdiArea;
class QMdiSubWindow;
class MdiChild;
class QSignalMapper;

class MainWin : public QMainWindow {
  Q_OBJECT
public:
  MainWin();
protected:
  void closeEvent(QCloseEvent *event);
private slots:
  void newFile();
  void about();
  void updateMenus();
  void updateWindowMenu();
  MdiChild *createMdiChild();
  void setActiveSubWindow(QWidget *window);
private:
  void createActions();
  void createMenus();
  void createToolBars();
  void createStatusBar();

  MdiChild *activeMdiChild();
  QMdiSubWindow *findMdiChild(const QString &fileName);

  QMdiArea *mdiArea;
  QSignalMapper *windowMapper;

  QMenu *fileMenu;
  QMenu *windowMenu;
  QMenu *helpMenu;
  QToolBar *fileToolBar;
```

```
41    QAction *newAct;
42    QAction *exitAct;
43    QAction *closeAct;
44    QAction *closeAllAct;
45    QAction *tileAct;
46    QAction *cascadeAct;
47    QAction *nextAct;
48    QAction *previousAct;
49    QAction *separatorAct;
50    QAction *aboutAct;
51    QAction *aboutQtAct;
52  };
53  #endif
```

Листинг 12.25. Файл chap12/mdi/mainwin.cpp

```
55  #include <QtGui>
56  #include "mainwin.h"
57  #include "mdichild.h"
58
59  MainWin::MainWin() { // Конструктор главного окна
60     mdiArea = new QMdiArea;
61     setCentralWidget(mdiArea);
62     connect(mdiArea, SIGNAL(subWindowActivated(QMdiSubWindow *)),
63             this, SLOT(updateMenus()));
64     windowMapper = new QSignalMapper(this);
65     connect(windowMapper, SIGNAL(mapped(QWidget *)),
66             this, SLOT(setActiveSubWindow(QWidget *)));
67     //mdiArea->setHorizontalScrollBarPolicy(Qt::ScrollBarAsNeeded);
68     //mdiArea->setVerticalScrollBarPolicy(Qt::ScrollBarAsNeeded);
69     createActions();   // Создание действий
70     createMenus();     // Формирование меню
71     createToolBars();  // Создание панелей инструментов
72     createStatusBar(); // Создание строки состояния
73     updateMenus();
74     setWindowTitle(tr("MDI"));
75  }
76
77  void MainWin::closeEvent(QCloseEvent *event){ // При закрытии окна
78     mdiArea->closeAllSubWindows(); // Закрыть все подокна
79     if (activeMdiChild()){ // Если какое-то окно осталось открытым,
80        event->ignore();    // то запретить закрытие главного окна
```

```
 81     } else {
 82        event->accept();     // иначе разрешить
 83     }
 84  }
 85
 86  void MainWin::newFile(){
 87     MdiChild *child = createMdiChild();
 88     child->newFile();
 89     child->show();
 90  }
 91
 92  void MainWin::about(){  // О программе
 93     QMessageBox::about(this, tr("О программе"),
 94           tr("Приложение <b>MDI</b>."));
 95  }
 96
 97  void MainWin::updateMenus(){  // Обновить меню
 98     bool hasMdiChild = (activeMdiChild() != 0);
 99     closeAct->setEnabled(hasMdiChild);
...   // аналогично для остальных действий, работающих с окнами
106  }
107
108  void MainWin::updateWindowMenu(){ // Обновить меню Window
109     windowMenu->clear();
110     windowMenu->addAction(closeAct);
111     windowMenu->addAction(closeAllAct);
112     windowMenu->addSeparator();
...    // аналогично для остальных действий, работающих с окнами
119
120     QList<QMdiSubWindow *> windows = mdiArea->subWindowList();
121     separatorAct->setVisible(!windows.isEmpty());
122
123     for (int i = 0; i < windows.size(); ++i) {
124        MdiChild *child = qobject_cast<MdiChild *>(
                                    windows.at(i)->widget() );
126
127        QString text; // Номер окна и имя файла
128        if (i < 9) {
129           text = tr("&%1 %2")
130                 .arg(i + 1)
131                 .arg(child->userFriendlyCurrentFile());
```

```
132         } else {
133            text = tr("%1 %2")
134                  .arg(i + 1)
135                  .arg(child->userFriendlyCurrentFile());
136         }
137         QAction *action  = windowMenu->addAction(text);
138         action->setCheckable(true);
139         action ->setChecked(child == activeMdiChild());
140         connect( action, SIGNAL(triggered()),
                    windowMapper, SLOT(map()) );
141         windowMapper->setMapping(action, windows.at(i));
142     }
143 }
144
145 MdiChild *MainWin::createMdiChild(){ // Создать новое окно
146    MdiChild *child = new MdiChild;
147    mdiArea->addSubWindow(child);
148
149    return child;
150 }
151
152 void MainWin::createActions(){  // Создать действия
153    newAct = new QAction( QIcon(":/images/filenew.png"),
                           tr("&Новый"), this);
154    newAct->setShortcut(tr("Ctrl+N"));
155    newAct->setStatusTip(tr("Создать новое окно"));
156    connect(newAct, SIGNAL(triggered()), this, SLOT(newFile()));
157
...   // аналогично для всех остальных действий
199 }
200
201 void MainWin::createMenus(){ // Создать пункты меню
202   fileMenu = menuBar()->addMenu(tr("&Файл"));
203   fileMenu->addAction(newAct);
204   fileMenu->addSeparator();
205   fileMenu->addAction(exitAct);
206
207   windowMenu = menuBar()->addMenu(tr("&Окно"));
208   updateWindowMenu();
209   connect( windowMenu, SIGNAL(aboutToShow()),
              this, SLOT(updateWindowMenu()) );
210
211   menuBar()->addSeparator();
212
```

```
213   helpMenu = menuBar()->addMenu(tr("&Справка"));
214   helpMenu->addAction(aboutAct);
215   helpMenu->addAction(aboutQtAct);
216 }
217
218 void MainWin::createToolBars(){  // Создать панели инструментов
219   fileToolBar = addToolBar(tr("Файл"));
220   fileToolBar->addAction(newAct);
221   fileToolBar->addSeparator();
222   fileToolBar->addAction(exitAct);
223 }
224
225 void MainWin::createStatusBar(){ // Создать строку состояния
226    statusBar()->showMessage(tr("Готов"));
227 }
228
229 MdiChild *MainWin::activeMdiChild(){ // Вернуть активное окно
230   if (QMdiSubWindow *activeSubWindow = mdiArea->activeSubWindow())
231       return qobject_cast<MdiChild *>(activeSubWindow->widget());
232   return 0;
233 }
234
235 QMdiSubWindow *MainWin::findMdiChild(const QString &fileName){
236    QString canonicalFilePath =
                    QFileInfo(fileName).canonicalFilePath();
237
238    foreach (QMdiSubWindow *window, mdiArea->subWindowList()){
239       MdiChild *mdiChild = qobject_cast<MdiChild *>(
                                          window->widget());
240       if (mdiChild->currentFile() == canonicalFilePath)
241           return window;
242    }
243    return 0;
244 }
245
246 void MainWin::setActiveSubWindow(QWidget *window){
247    if (!window)
248        return;
249    mdiArea->setActiveSubWindow( // Сделать окно текущим
                       qobject_cast<QMdiSubWindow *>(window));
250 }
```

Листинг 12.26. Файл chap12/mdi/mdichild.h

```
252 #ifndef MDICHILD_H
253 #define MDICHILD_H
254
255 #include <QTextEdit>
256
257 class MdiChild : public QTextEdit {
258    Q_OBJECT
259 public:
260    MdiChild();
261    void newFile();
262    QString userFriendlyCurrentFile();
263    QString currentFile() { return curFile; }
264 private:
265    void setCurrentFile(const QString &fileName);
266    QString strippedName(const QString &fullFileName);
267    QString curFile;
268    bool isUntitled;
269 };
270 #endif
```

Листинг 12.27. Файл chap12/mdi/mdichild.cpp

```
272 #include <QtGui>
273 #include "mdichild.h"
274
275 MdiChild::MdiChild(){
276    setAttribute(Qt::WA_DeleteOnClose);
277    isUntitled = true;
278 }
279
280 void MdiChild::newFile(){
281    static int sequenceNumber = 1;
282
283    isUntitled = true;
284    curFile = tr("document%1.txt").arg(sequenceNumber++);
285    setWindowTitle(curFile + "[*]");
286 }
287
288 QString MdiChild::userFriendlyCurrentFile(){
289    return strippedName(curFile);
290 }
291
```

```
292 void MdiChild::setCurrentFile(const QString &fileName){
293    curFile = QFileInfo(fileName).canonicalFilePath();
294    isUntitled = false;
295    document()->setModified(false);
296    setWindowModified(false);
297    setWindowTitle(userFriendlyCurrentFile() + "[*]");
298 }
299
300 QString MdiChild::strippedName(const QString &fullFileName){
301    return QFileInfo(fullFileName).fileName();
302 }
```

Листинг 12.28. Файл chap12/mdi/main.cpp

```
304 #include <QApplication>
305 #include <QTextCodec>
306 #include "mainwin.h"
307
308 int main(int argc, char *argv[]){
309    Q_INIT_RESOURCE(mdi);
310    QApplication app(argc, argv);
311    QTextCodec *codec = QTextCodec::codecForName("CP1251");
312    QTextCodec::setCodecForTr(codec);
313
314    MainWin mainWin;
315    mainWin.show();
316    return app.exec();
317 }
```

Листинг 12.29. Файл chap12/mdi/mdi.qrc

```
318 <!DOCTYPE RCC><RCC version="1.0">
319 <qresource>
320    <file>images/filenew.png</file>
321    <file>images/fileexit.png</file>
322 </qresource>
323 </RCC>
```

Листинг 12.30. Файл chap12/mdi/mdi.pro

```
324 TEMPLATE = app
325 HEADERS += mainwin.h mdichild.h
326 SOURCES += main.cpp mainwin.cpp mdichild.cpp
327 RESOURCES += mdi.qrc
```

Пояснения к программе:

- (62–63) — обновление пунктов меню при смене текущего окна рабочей области MDI;
- (64–66, 141) создаем сопоставитель сигналов `QSignalMapper` (см. *разд. 5.5*), чтобы знать, в каком именно окне рабочей области был сгенерирован сигнал;
- (67–68) — раскомментируйте эти строки, если требуется снабдить рабочую область полосами прокрутки.

12.5. Плавающие окна

Говоря о многооконном интерфейсе, нельзя не упомянуть еще об одном способе его организации — с помощью, так называемых, "плавающих" (*docking*) окон, которые пользователь может произвольно перемещать, прикреплять к различным областям главного окна и друг к другу и даже образовывать многостраничные закладки.

ЗАДАЧА. Разработать приложение с плавающими окнами (рис. 12.4).

Решение задачи приведено в листингах 12.31 и 12.32.

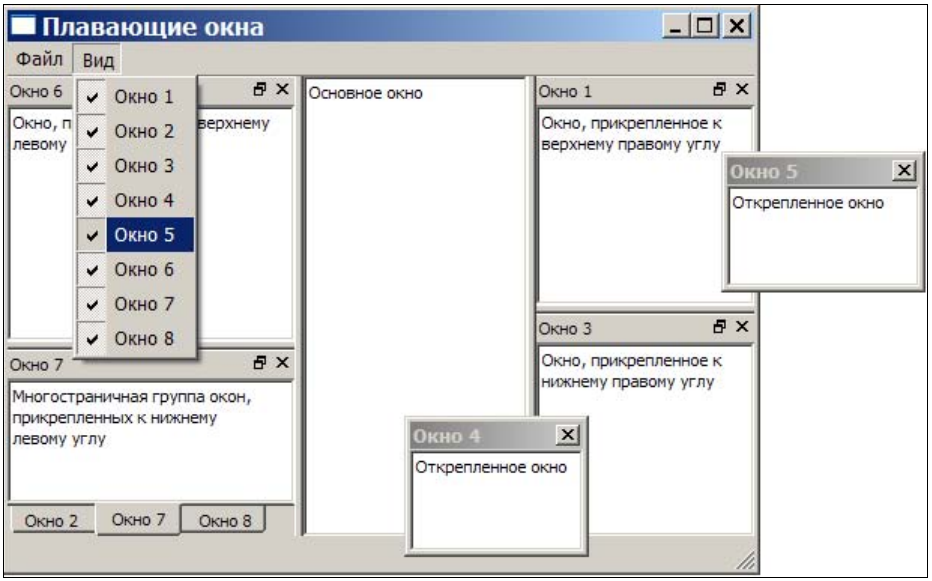


Рис. 12.4. Внешний вид "плавающих" окон

Листинг 12.31. Файл chap12/dock/mainwin.h

```
#include <QtGui>

class MainWin : public QMainWindow{
    Q_OBJECT
public:
    MainWin();
private slots:
    void createDockWin(); // Создать новое плавающее окно
private:
    void createActions();    // Создать все действия
    void createMenus();      // Создать меню
    void createStatusBar();  // Создать строку состояния

    QMenu *fileMenu; // Меню "Файл"
    QMenu *viewMenu; // Меню "Вид"
    QAction *newDockAct; // Действие "Новое окно"
    QAction *quitAct;    // Действие "Выход"
    int count;      // Счетчик окон
};
```

Листинг 12.32. Файл chap12/dock/mainwin.cpp

```
#include <QtGui>
#include "mainwin.h"

MainWin::MainWin() : QMainWindow() {
    count = 0;

    // Текстовый редактор для главного окна:
    QTextEdit *textEdit = new QTextEdit;
    setCentralWidget(textEdit);

    createActions();   // Создать действия
    createMenus();     // Создать меню
    createStatusBar(); // Создать строку состояния
    setWindowTitle(tr("Плавающие окна")); // Заголовок окна

    // Создадим два плавающих окна:
    createDockWin();
    createDockWin();
}
```

```
void MainWin::createDockWin(){ // Создать новое плавающее окно
    QDockWidget *dock = new QDockWidget(
        tr("Окно %1").arg(++count),
        this);

    // Зададим области, к которым можно будет прикрепляться:
    dock->setAllowedAreas(
        Qt::LeftDockWidgetArea | Qt::RightDockWidgetArea );

    // Текстовый редактор для плавающего окна:
    QTextEdit *te = new QTextEdit(dock);
    dock->setWidget(te);

    // Добавим плавающее окно к главному окну:
    addDockWidget(Qt::RightDockWidgetArea, dock);

    // Добавим новый пункт в меню "Вид"
    // для показа/скрытия плавающего окна:
    viewMenu->addAction(dock->toggleViewAction());
}

void MainWin::createActions(){ // Создать действия
    newDockAct = new QAction(
        tr("&Создать"), this);
    newDockAct->setShortcut(tr("Ctrl+N"));
    newDockAct->setStatusTip(tr("Создать новое плавающее окно"));
    connect(newDockAct, SIGNAL(triggered()),
            this, SLOT(createDockWin()));

    quitAct = new QAction(tr("В&ыход"), this);
    quitAct->setShortcut(tr("Ctrl+Q"));
    quitAct->setStatusTip(tr("Завершение работы"));
    connect(quitAct, SIGNAL(triggered()), this, SLOT(close()));
}

void MainWin::createMenus(){ // Создать меню
    fileMenu = menuBar()->addMenu(tr("&Файл"));
    fileMenu->addAction(newDockAct);
    fileMenu->addSeparator();
    fileMenu->addAction(quitAct);

    viewMenu = menuBar()->addMenu(tr("&Вид"));
}
```

```
void MainWin::createStatusBar(){ // Создать строку состояния
    statusBar()->showMessage(tr("Готов"));
}

//-------------------------------------
int main(int argc, char *argv[]){
    QApplication app(argc, argv);
    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);

    MainWin w; // Главное окно
    w.show();
    return app.exec();
}
```

Замечание

При "закрытии" плавающего окна с помощью крестика в правом верхнем углу оно не уничтожается, как обычное окно, а просто скрывается с экрана. Если вернуть галочку в соответствующем пункте меню **Вид**, то "закрытое" окно снова отобразится на своем прежнем месте. Чтобы действительно удалить плавающее окно, его нужно сначала "открепить" от главного окна с помощью метода `QMainWindow::removeDockWidget()`.

Замечание

Чтобы запомнить положение всех плавающих окон и панелей инструментов относительно главного окна, используется метод `QMainWindow::saveState()`. Он формирует объект `QByteArray`, который можно сохранить, например, в файле. Для восстановления состояния предназначен метод `QMainWindow::restoreState()`.

12.6. Хранение настроек приложения

Сохранение различных параметров между сеансами выполнения программы обычно обеспечивают ini-файлы или реестр. В библиотеке Qt имеется класс `QSettings`, который умеет работать с текстовыми файлами на всех платформах, а в Windows — также и с реестром.

ЗАДАЧА. Сохранить в текстовом файле *.ini положение и размер главного окна приложения.

Решение. Перед показом главного окна необходимо прочитать настройки из файла, а если такой файл пока отсутствует, то принять некоторые настройки по умолчанию. Перед закрытием приложения требуется сохранить текущие настройки в файл (листинги 12.33–12.35).

Листинг 12.33. Файл chap12/settings/settings.h

```
#include <QtGui>

class MainWin : public QMainWindow {
    Q_OBJECT
public:
    MainWin();          // Конструктор главного окна приложения
    void createActions();   // Создать действия
    void createMenus();     // Создать меню
    void createToolBars();  // Создать панели инструментов
    void createStatusBar(); // Создать строку состояния
    void readSettings();    // Прочитать и принять настройки
    void writeSettings();   // Записать настройки в файл
    void closeEvent(QCloseEvent *event); // Перед закрытием окна
public slots:
    // ...
protected:
    QSettings *m_settings;  // Настройки
    QAction *m_actNew;      // Действие "Новый"
    QAction *m_actExit;     // Действие "Выход"
    QMenu *m_menuFile;      // Меню "Файл"
    QToolBar *m_tb;         // Панель инструментов
};
```

Листинг 12.34. Файл chap12/settings/settings.cpp

```
#include "settings.h"

MainWin::MainWin()
       : QMainWindow(0) {

    // Текстовый редактор — центральный виджет:
    QTextEdit *te = new QTextEdit(this);
    setCentralWidget(te);

    m_settings = new QSettings(
        tr("z1.ini"), // Имя файла для хранения настроек
        QSettings::IniFormat, // Формат — файл ini
        this );
    readSettings(); // Прочитать и принять настройки

    createActions();  // Создать действия
    createMenus();    // Создать меню
    createToolBars(); // Создать панели инструментов
```

```
    createStatusBar(); // Создать строку состояния
    setWindowIcon(qApp->style()->standardIcon(
        QStyle::SP_ComputerIcon ) ); // Значок главного окна
}

void MainWin::createActions() {  // Создать действия
    m_actNew = new QAction(
        qApp->style()->standardIcon(QStyle::SP_FileDialogNewFolder),
        tr("&Новый документ"), this);
    m_actNew->setShortcut(tr("Ctrl+N"));
    m_actNew->setStatusTip(tr("Создать новый документ"));
    //connect(m_actNew, SIGNAL(triggered()), this, SLOT((...)));

    m_actExit = new QAction(
        qApp->style()->standardIcon(QStyle::SP_DialogCancelButton),
//SP_DialogCloseButton),
        tr("В&ыход"), this);
    m_actExit->setShortcut(tr("Ctrl+Q"));
    m_actExit->setStatusTip(tr("Завершить работу"));
    connect(m_actExit, SIGNAL(triggered()), this, SLOT(close()));
}

void MainWin::createMenus() {  // Создать меню
    m_menuFile = menuBar()->addMenu(tr("&Файл"));
    m_menuFile->addAction(m_actNew);
    m_menuFile->addSeparator();
    m_menuFile->addAction(m_actExit);
}

void MainWin::createToolBars() {  // Создать панели инструментов
    m_tb = addToolBar(tr("General"));
    m_tb->addAction(m_actNew);
    m_tb->addSeparator();
    m_tb->addAction(m_actExit);
}

void MainWin::createStatusBar() { // Создать строку состояния
    statusBar()->showMessage(tr("Готов"));
}

void MainWin::readSettings(){  // Прочитать и принять настройки
    // Узнаем размеры экрана:
    QDesktopWidget desk;
    QRect rect = desk.availableGeometry();
```

```
    m_settings->beginGroup("MainWin"); // в группе [MainWin]
    QPoint pos = m_settings->value("pos", QPoint(0, 0)).toPoint();
    QSize size = m_settings->value("size", rect.size()).toSize();
    if (pos.x() < -4)
       pos.setX(-4);
    if (pos.y() < -4)
        pos.setY(-4);
    if (size.width() > rect.width() )
        size.setWidth(rect.width());
    if (size.height() > rect.height() )
        size.setHeight(rect.height());
    m_settings->endGroup();

    // Изменить размеры и положение главного окна:
    resize(size);
    move(pos);
}

void MainWin::writeSettings(){ // Записать настройки
    // Работать в группе [MainWin]
    m_settings->beginGroup("MainWin");

    m_settings->setValue("pos", pos());
    m_settings->setValue("size", size());
    m_settings->endGroup();

    m_settings->sync(); // Сбросить изменения на диск
}

void MainWin::closeEvent(QCloseEvent *event) { // Перед закрытием окна
    if ( QMessageBox::question(
            this,
            tr("Подтвердите"),
            tr("Завершить работу?"),
            QMessageBox::Yes | QMessageBox::No,
            QMessageBox::Yes) == QMessageBox::Yes ) {
        writeSettings();  // Записать состояние окна
        event->accept();  // Разрешить выход из программы
     }else{
         event->ignore(); // Запретить выход из программы
     }
}
```

Листинг 12.35. Файл chap12/settings/main.cpp

```
#include <QtGui>
#include "settings.h"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QTextCodec *incodec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(incodec);

    QTranslator qtTranslator; // Перевод для библиотеки Qt
    qtTranslator.load("translations/qt_ru");
    app.installTranslator(&qtTranslator);

    MainWin mw;
    mw.show();

    return app.exec();
}
```

Чтобы в стандартном диалоге `QMessageBox` увидеть русские надписи на кнопках, мы загрузили файл перевода qt_ru.qm. Перед запуском программы в каталоге settings нужно создать подкаталог translations и поместить туда этот файл (после установки библиотеки Qt он находится в каталоге QtXXX/translations). Чтобы исполняемый файл "увидел" каталог с файлом перевода, он должен располагаться там же, где и остальные файлы проекта. Поскольку по умолчанию в системе Windows исполняемый файл формируется в подкаталогах debug или release, мы добавили в последней строке файла *.pro указание компоновщику поместить исполняемый файл на один уровень выше (листинг 12.36).

Листинг 12.36. Файл chap12/settings/settings.pro

```
TEMPLATE = app
DEPENDPATH += .
INCLUDEPATH += .
HEADERS += settings.h
SOURCES += settings.cpp \
           main.cpp
win32:TARGET  = ../settings
```

В результате работы программы формируется следующий текстовый файл, в котором хранится позиция левого верхнего угла главного окна и его размеры, какими они были на момент последнего закрытия (листинг 12.37).

Листинг 12.37. Файл chap12/settings/z1.ini

```
[MainWin]
pos=@Point(386 396)
size=@Size(290 190)
```

Разумеется, в реальной программе придется создать больше групп и параметров, а также разделить системный файл *.ini (общий для всех пользователей) и файл настроек текущего пользователя, хранящийся в каталоге пользователя. Для этого при создании объекта `QSettings` указываются параметры `QSettings::SystemScope` или `QSettings::UserScope`.

12.7. Разное

ЗАДАЧА. Как "открепить" дочернее окно приложения MDI от своей рабочей области, сделав его самостоятельным?

Решение. Чтобы удалить окно из рабочей области MDI, используется метод `QMdiArea::removeSubWindow()`. При этом объект `QMdiSubWindow` остается без родителя. Если он, в свою очередь, как обычно бывает, является контейнером для другого виджета, то последний может быть откреплен от объекта `QMdiSubWindow` с помощью метода `QWidget::setParent(0)`. В листинге 12.38 приведен пример решения задачи.

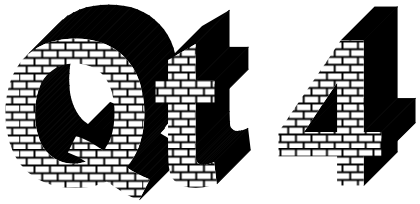
Листинг 12.38. Файл chap12/child2top/child2top.cpp

```
#include <QtGui>

class Widget : public QWidget {
    Q_OBJECT
public:
    Widget(QMdiArea *parent) : QWidget(parent) {
        QTextEdit *edit = new QTextEdit(this);
        QVBoxLayout *layout = new QVBoxLayout(this);
        layout->addWidget(edit);
        mdi = parent;
        mdiMode = false;
        QTimer *timer = new QTimer();
        connect(timer, SIGNAL(timeout() ),
                this, SLOT(mySlot() ) );
        timer->start(2000);
    }
```

```
public slots:
    void mySlot() {
        if (mdiMode){
            setParent(mdi);
            mdi->addSubWindow(this);
            show();
            mdiMode = !mdiMode;
        } else {
            QWidget *w = qobject_cast<QWidget*>(this->parent());
            mdi->removeSubWindow(w);
            setParent(0);
            if (w)  w->close();
            move(500, 400);
            show();
            mdiMode = !mdiMode;
        }
    }
private:
    bool mdiMode;
    QMdiArea *mdi;
};

#include "child2top.moc"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QMainWindow mw;
    QMdiArea *mdi = new QMdiArea;
    mw.setCentralWidget(mdi);
    Widget *widget = new Widget(mdi);
    mdi->addSubWindow(widget);
    mw.resize(400, 300);
    mw.show();

    return app.exec();
}
```

Здесь виджет `QTextEditor` каждые две секунды изменяет свое состояние: то "прикрепляется" к рабочей области окна MDI, то "открепляется" от нее, становясь самостоятельным окном верхнего уровня.

ГЛАВА 13



Графика и печать

> Неважно, насколько плохо вы рисуете, до тех пор, пока вы рисуете плохо не так, как другие.
> *Джордж Мур*

В библиотеке Qt 4 имеется несколько классов для работы с графическими изображениями. Самый примитивный из них — `QBitmap`, представляющий монохромное (т. е. черно-белое) прямоугольное изображение с помощью нулевых и единичных битов. В основном класс `QBitmap` используется при создании пользовательских курсоров `QCursor`, кистей `QBrush`, областей для рисования `QRegion`, а также для хранения битовых масок, применяемых при работе с полноцветными изображениями.

Для представления полноцветных растровых изображений предназначен платформонезависимый класс `QImage`. Он оптимизирован для случаев, когда требуется получить быстрый доступ к отдельным пикселам изображения. Для быстрого вывода на экран растровых графических изображений имеется класс `QPixmap`. В отличие от `QImage`, при отрисовке `QPixmap` задействует возможности видеокарты быстро отображать графические примитивы. Масштабируемые значки с несколькими состояниями (`Normal`, `Disabled`, `Active` и `Selected`) представлены классом `QIcon`. Более сложный класс `QPicture` применяется для рисования произвольных объектов двумерной графики.

Для визуализации большого числа двумерных элементов (линий, прямоугольников, текстовых надписей, произвольных примитивов) с возможностью масштабирования и вращения предназначены классы, объединенные общим названием `Graphics View`. Они заменяют устаревший класс `QCanvas`.

Но основой всех классов для работы с двумерной графикой является класс `QPainter`, в том числе отвечающий за прорисовку самих виджетов. Для отображения графических файлов SVG (*Scalable Vector Graphics* — масштабируемая векторная графика) есть специальный модуль `QtSvg`. Кроме того, для

работы с двумерными и трехмерными изображениями в библиотеке Qt имеется модуль QtOpenGL.

13.1. Класс *QPainter*

С помощью методов класса QPainter мы можем отображать на плоскости различные геометрические объекты: точки, линии, прямоугольники, эллипсы, дуги и окружности, секторы и сегменты круга, многоугольники, кривые Безье, наборы элементов векторной графики, растровые изображения и текстовые строки. Для лучшего качества отображения шрифтов и геометрических фигур можно использовать *антиалиасинг* (сглаживание). Поддерживаются такие возможности, как трансформация изображения, взвешенное наложение (альфа-блендинг) и градиентная заливка. С помощью класса QPainter можно сгенерировать страницу для вывода на принтер или в файл формата PDF.

Отрисовка любого изображения, например виджета MyWidget, должна происходить внутри виртуального метода QWidget::paintEvent(), который вызывается всякий раз, когда виджет требует полной или частичной перерисовки (например, при перемещении окна) (листинг 13.1).

Листинг 13.1

```
void MyWidget::paintEvent(QPaintEvent *event){
    QPainter painter(this);
    painter.setPen(Qt::blue);                // Цвет пера
    painter.setFont(QFont("Arial", 30));     // Шрифт
    painter.drawText(rect(),             // Прямоугольник виджета
                     Qt::AlignCenter,    // Выравнивание по центру
                     tr("Привет!") );    // Отображаемый текст
}
```

Параметр конструктора — это адрес контекста рисования — любого объекта, унаследованного от QPaintDevice. Класс QPaintDevice является одним из родителей для любого виджета (прямого или косвенного потомка класса QWidget).

Для рисования используются следующие основные методы:

- точка:

```
void QPainter::drawPoint ( const QPointF& position );
void QPainter::drawPoint ( const QPoint& position );
void QPainter::drawPoint ( int x, int y );
```

- набор точек, заданный массивом или многоугольником:

```
void QPainter::drawPoints ( const QPointF *points, int pointCount );
void QPainter::drawPoints ( const QPoint *points, int pointCount );
```

- линия:

```
void QPainter::drawLine ( const QLineF& line );
void QPainter::drawLine ( const QPoint& p1, const QPoint& p2 );
void QPainter::drawLine ( int x1, int y1, int x2, int y2 );
```

- набор линий:

```
void QPainter::drawLines ( const QLineF *lines, int lineCount );
void QPainter::drawLines ( const QPointF *pointPairs, int lineCount);
void QPainter::drawLines ( const QVector<QPointF>& pointPairs );
void QPainter::drawLines ( const QVector<QLineF>& lines );
```

- прямоугольник:

```
void QPainter::drawRect ( const QRectF& rectangle );
void QPainter::drawRect ( int x, int y, int width, int height );
```

- набор прямоугольников:

```
void QPainter::drawRects ( const QRectF *rectangles, int rectCount );
```

- прямоугольник с закругленными углами:

```
void QPainter::drawRoundRect (
    const QRectF& r,
    int xRnd = 25, int yRnd = 25 );
```

- эллипс:

```
void QPainter::drawEllipse ( const QRectF& rectangle );
void QPainter::drawEllipse ( int x, int y, int width, int height );
```

- дуга эллипса/окружности (целое число 16 при задании угла соответствует одному градусу) или весь эллипс/окружность (при `startAngle = 0`, `spanAngle = 360 x 16 = 5760`):

```
void QPainter::drawArc (
    const QRectF& rectangle, int startAngle, int spanAngle );
```

- сектор круга (область, ограниченная двумя радиусами и дугой окружности):

```
void QPainter::drawPie (
    const QRectF& rectangle, int startAngle, int spanAngle );
```

- сегмент круга (область, ограниченная хордой и дугой окружности):

```
void QPainter::drawChord (
    const QRectF& rectangle, int startAngle, int spanAngle );
void QPainter::drawChord (
    int x, int y, int width, int height,
    int startAngle, int spanAngle );
```

- ломаная линия:

```
void QPainter::drawPolyline (
    const QPointF *points, int pointCount );
```

- многоугольник:

```
void QPainter::drawPolygon (
    const QPointF *points, int pointCount,
    Qt::FillRule fillRule = Qt::OddEvenFill );
    Qt::FillRule fillRule = Qt::OddEvenFill );
```

- выпуклый многоугольник:

```
void QPainter::drawConvexPolygon (
    const QPointF *points, int pointCount );
```

- кривая Безье, заданная четырьмя подряд идущими точками многоугольника:

```
void QPainter::drawCubicBezier (
    const QPolygon& controlPoints, int index = 0 );
```

- заливка прямоугольника:

```
void QPainter::fillRect (
    const QRectF& rectangle, const QBrush& brush );
void QPainter::fillRect (
    int x, int y, int width, int height,
    const QBrush & brush );
```

- очистка внутренней области прямоугольника (заливка цветом фона):

```
void QPainter::eraseRect ( const QRectF& rectangle );
```

- растровое изображение:

```
void QPainter::drawPixmap (
    const QRectF& target,
    const QPixmap& pixmap,
    const QRectF & source );
```

```
void QPainter::drawPixmap (
    int x, int y, const QPixmap& pixmap,
    int sx, int sy, int sw, int sh );
```

- набор элементов векторной графики:

```
void QPainter::drawPath ( const QPainterPath& path );
```

- рисунок ("записанная" последовательность команд рисования):

```
void QPainter::drawPicture (
    const QPointF& point, const QPicture& picture );
```

- текстовая строка:

```
void QPainter::drawText (
     const QPointF& position, const QString& text );
void QPainter::drawText (
     int x, int y, const QString & text );
void QPainter::drawText (
     int x, int y, int width, int height,
     int flags, const QString& text,
     QRect *boundingRect = 0 );
void QPainter::drawText (
     const QRectF& rectangle, const QString& text,
     const QTextOption& option = QTextOption() );
```

Для рисования объектов используются следующие настройки:

- перо `QPen` — объект для рисования контурных линий. Определяется набором параметров:
  - цвет `QColor`;
  - толщина (целое неотрицательное число);
  - стиль `Qt::PenStyle`:
    - `Qt::NoPen` — невидимое перо;
    - `Qt::SolidLine` — сплошная линия;
    - и т. д.;
  - форма пера `Qt::PenCapStyle`, влияющая на форму концов толстых линий:
    - `Qt::FlatCap` — квадратное перо;
    - и другие;
  - форма сочленения соседних отрезков линий `Qt::PenJoinStyle`:

- Qt::MiterJoin, Qt:: SvgMiterJoin — соединением углом;
- и т. д.

- кисть QBrush — объект для заполнения внутренней области двумерных фигур. Кисть определяется набором параметров:
  - цвет QColor;
  - стиль заливки:
    - Qt::NoBrush — нет заливки;
    - Qt::SolidPattern — сплошная заливка однотонным цветом;
    - Qt::Dense1Pattern — несплошная заливка большой плотности;
    - ...
    - Qt::TexturePattern — произвольная заливка;
- шрифт текста QFont;
- режим отрисовки фона:
  - Qt::TransparentMode — прозрачный фон;
  - Qt::OpaqueMode — непрозрачный фон;
- цвет и стиль фона QBrush (если он не прозрачный);

и многие другие.

В одном и том же контексте можно рисовать с помощью нескольких объектов QPainter, что иллюстрирует листинг 13.2.

Листинг 13.2

```
QPainter p1;
QPainter p2;
p1.begin(this);
// ...
p1.end();
p2.begin(this);
// ...
p2.end();
```

Наоборот, один объект QPainter может рисовать в разных контекстах (листинг 13.3).

Листинг 13.3

```
   QPainter p;
// сначала рисуем в контексте нашего виджета:
   p.begin(this);
```

```
   // ...
   p.end();
// создаем пустой рисунок того же размера, что и наш виджет:
   QPixmap img( rect() );
// рисуем в его контексте:
   p.begin(&img);
   // ...
   p.end();
```

13.2. Пользовательский стиль линий

ЗАДАЧА. Требуется нарисовать сплошную и штрихпунктирную линию, а также линию, представляющую собой периодическую последовательность штрихов увеличивающейся длины.

Решение. Стиль линий выбирают с помощью метода `QPen::setStyle()`. Пользовательский стиль можно задать методом

```
QPen::setDashPattern(const QVector<qreal>& pattern)
```

где вектор-параметр содержит длину каждой линии и промежутка после нее (листинг 13.4).

Листинг 13.4

```
QPixmap pixmap( 200, 100 );
pixmap.fill( Qt::white );
QPainter painter( &pixmap );
QPen pen( Qt::black );

pen.setStyle( Qt::SolidLine );
painter.setPen( pen );
painter.drawLine( QPoint( 10, 10 ), QPoint( 190, 10 ) );

pen.setStyle( Qt::DashDotLine );
painter.setPen( pen );
painter.drawLine( QPoint( 10, 50 ), QPoint( 190, 50 ) );

pen.setDashPattern(
    QVector<qreal>() << 1 << 1 << 2 << 2 << 4 << 4
                     << 6 << 6 << 8 << 8 );
pen.setStyle( Qt::CustomDashLine );
painter.setPen( pen );
painter.drawLine( QPoint( 10, 90 ), QPoint( 190, 90 ) );
```

13.3. Форматированный текст и HTML

Красивая ложь? Внимание! Это уже творчество.
Станислав Ежи Лец

ЗАДАЧА. Отобразить с помощью элемента `QTextEdit` текст, содержащий сложное форматирование: заголовок, маркированный список, таблицу.

Решение. В листинге 13.5 приведен текст программы, а на рис. 13.1 показан результат ее работы.

Рис. 13.1. Форматированный текст

Листинг 13.5. Файл chap13/html/html.cpp

```
#include <QtGui>

int main(int argc, char *argv[]) {
   QTextCodec *codec = QTextCodec::codecForName("CP1251");
   QTextCodec::setCodecForTr(codec);

   QApplication app(argc, argv);

   QTextEdit *te = new QTextEdit();
   te->setFont(QFont("Times", 14, QFont::Normal));
   te->setAutoFormatting(QTextEdit::AutoAll);
   te->setText(QObject::tr(
       "<html> <head>"
       "<meta http-equiv=\"Content-Type\" "
       "   content=\"text/html; charset=cp1251\" />"
```

```
16          "</head>"
17          "<body bgcolor=#c0d8ff>"
18          "<h3>Пример форматированного текста</h3>"
19          "<ul type=square>"
20          "  <li>Первый <font color=#a04010>пункт</font> списка"
21          "  <li>Второй <b>пункт</b> <i>списка</i>"
22          "</ul>"
23          "<p align=center>Таблица</p>"
24          "<table border=1 cellspacing=0 cellpadding=5 width=90%>"
25          "  <tr>"
26          "    <th bgcolor=#f0d0e0 width=50%>Колонка 1</th>"
27          "    <th bgcolor=#c0f0e0 width=50%>Колонка 2</th>"
28          "  </tr><tr>"
29          "    <td>Ячейка 1,1</td>"
30          "    <td align=center>Ячейка 1,2</td>"
31          "  </tr><tr>"
32          "    <td align=right>Ячейка 2,1</td>"
33          "    <td>Ячейка 2,2</td>"
34          "  </tr><tr>"
35          "    <td colspan=2 align=center>Объединенная ячейка</td>"
36          "  </tr>"
37          "</table>"
38          "</body></html>" ) );
39      te->resize(420,300);
40      te->show();
41
42      QString html = te->toHtml();
43      QFile file("f1.html");
44      file.open(QIODevice::WriteOnly | QIODevice::Text);
45      file.write(html.toLocal8Bit());
46      file.close();
47      // system("f1.html");
48
49      return app.exec();
50  }
```

Пояснения к программе:

- (9–11) — создали объект `QTextEdit`, установили шрифт и режим автоматического форматирования (строки с символом `*` в первой позиции будут переформатированы при вводе как строки маркированного списка);
- (13–38) — задали текст на языке HTML;
- (39, 40) — установили размер и вывели виджет на экран;

- (42) — получили текст на языке HTML;
- (43–46) — записали текст в файл, используя системную кодировку;
- (47) — можно открыть этот файл в браузере (если в системе зарегистрировано расширение html).

ЗАДАЧА. Изменить ширину полей вокруг отдельного блока и для всего текстового документа.

Решение. В листинге 13.6 приведено решение задачи, а на рис. 13.2 показан внешний вид окна приложения.

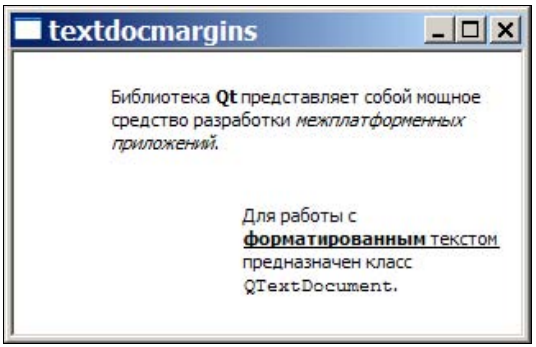


Рис. 13.2. Изменение ширины полей всего документа и отдельного блока

Листинг 13.6. Файл chap13/textdocmargins/textdocmargins.cpp

```
#include <QtGui>

int main(int argc, char** argv){
  QApplication app(argc, argv);
  QTextCodec *codec = QTextCodec::codecForName("CP1251");
  QTextCodec::setCodecForTr(codec);

  QTextEdit edit;
  edit.setText( QObject::tr(
      "Библиотека <b>Qt</b> представляет собой мощное средство "
      "разработки <i>межплатформенных приложений</i>.\n") );

   edit.append( QObject::tr(
      "Для отображения <u><b>форматированного</b> текста</u> "
      "предназначен класс <tt>QTextDocument</tt>.") );

```

```
17     QTextDocument *doc = edit.document();
18     QTextCursor cursor(doc);
19     cursor.movePosition(QTextCursor::End, QTextCursor::MoveAnchor);
20    //cursor.movePosition(QTextCursor::Start, QTextCursor::KeepAnchor);
21
22    QTextBlockFormat tbf = cursor.blockFormat();
23    tbf.setLeftMargin(80);
24    tbf.setTopMargin(30);
25    cursor.setBlockFormat(tbf);
26
27    QTextFrame *tf = doc->rootFrame();
28    QTextFrameFormat tff = tf->frameFormat();
29    // tff.setMargin(50);
30    tff.setLeftMargin(60);
31    tff.setTopMargin(20);
32    tff.setRightMargin(10);
33    tff.setBottomMargin(10);
34    tf->setFrameFormat(tff);
35
36    edit.show();
37    return app.exec();
38  }
```

Пояснения к программе:

- (8) — для отображения форматированного текста используется виджет `QTextEdit`;
- (9–11) — задали текстовое содержимое на языке HTML;
- (13) — добавили еще один абзац текста;
- (17) — получили содержимое виджета `QTextEdit`;
- (18) — создали курсор;
- (19) — переместили курсор и выделение в конец текста;
- (20) — другой вариант — переместить курсор в начало текста, оставив выделение в его конце, тем самым выделив весь текст;
- (22–25) — создали объект "Формат выделенного блока текста" и задали высоту верхнего и ширину левого поля;
- (27–34) — создали объект "Фрейм документа" и задали размеры всех его полей;
- (29) — если все четыре размера совпадают, то их можно определить с помощью одного вызова `setMargin()`.

13.4. Отображение форматированного текста

ЗАДАЧА. Отобразить внутри объекта `QPainter` форматированный текст, заданный на языке HTML, и прямоугольную рамку вокруг него.

Решение иллюстрирует листинг 13.7.

Листинг 13.7. Текст в рамке

```
QTextDocument doc;
doc.setHtml(
   "<p align=center>Для работы с <b><u>форматированным</u> "
   "текстом</b> используется класс <tt>QTextDocument</tt>"
"in a nice way.</p>" );
QRect rect = QRect(10, 30, 200, 40);
rect.translate( 0, rect.height()+10 );
rect.setHeight( 160 );
doc.setTextWidth( rect.width() );
painter.save();
painter.translate( rect.topLeft() );
doc.drawContents( &painter, rect.translated( -rect.topLeft() ) );
painter.restore();
painter.drawRect( rect );
```

13.5. Часы со стрелками

ЗАДАЧА. Реализовать новый виджет: часы со стрелками. Дублировать текущее время в цифровом виде.

Решение. Текст программы приведен в листингах 13.8–13.10, а внешний вид виджета показан на рис. 13.3.

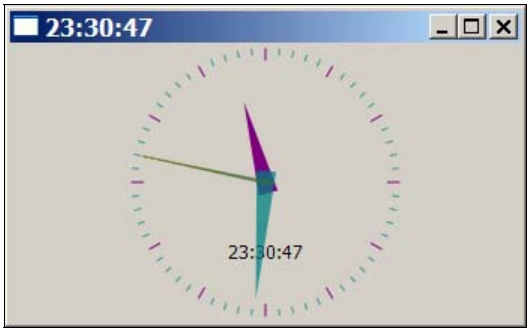


Рис. 13.3. Часы со стрелками

Листинг 13.8. Файл chap13/clock/clock.h

```
#ifndef _CLOCK_H_
#define _CLOCK_H_

#include <QWidget>
#include <QTime>
#include <QString>

class Clock : public QWidget {
    Q_OBJECT
public:
    Clock(QWidget *parent = 0);
protected:
    void paintEvent(QPaintEvent *event);
    QTime time;
    QString strTime;
signals:
    void timeChanged(const QString& s);
public slots:
    void update();
};
#endif // _CLOCK_H_
```

Листинг 13.9. Файл chap13/clock/clock.cpp

```
#include <QtGui>
#include "clock.h"

Clock::Clock(QWidget *parent) : QWidget(parent){
   QTimer *timer = new QTimer(this);
   connect(timer, SIGNAL(timeout()), this, SLOT(update()));
   timer->start(1000);
   setWindowTitle(tr("Clock"));
   resize(150, 150);
}

void Clock::update(){
    time = QTime::currentTime();
    strTime = time.toString("hh:mm:ss");
    setWindowTitle(strTime);
    emit timeChanged(strTime);
    QWidget::update();
}
```

```
40
41 void Clock::paintEvent(QPaintEvent*){
42    static const QPoint hourHand[3] = {
43         QPoint(7, 8),
44         QPoint(-7, 8),
45         QPoint(0, -60)
46    };
47    static const QPoint minuteHand[3] = {
48       QPoint(7, 8),
49       QPoint(-7, 8),
50       QPoint(0, -85)
51    };
52    static const QPoint secondHand[3] = {
53       QPoint(1, 4),
54       QPoint(-1, 4),
55       QPoint(0, -90)
56    };
57    QColor hourColor(127, 0, 127);
58    QColor minuteColor(0, 127, 127, 190);
59    QColor secondColor(100, 100, 0, 150);
60
61    int side = qMin(width(), height());
62
63    QPainter painter(this);
64    painter.setRenderHint(QPainter::Antialiasing);
65
66    QFont font = painter.font();
67    font.setPointSizeF(side/20);
68    painter.setFont(font);
69    painter.drawText( QRectF(0, height()/2, width(), side/2),
70                      Qt::AlignCenter, strTime );
71
72    painter.translate(width() / 2, height() / 2);
73    painter.scale(side / 200.0, side / 200.0);
74
75    painter.setPen(Qt::NoPen);
76    painter.setBrush(hourColor);
77
78    painter.save();
79    painter.rotate(30.0 * ((time.hour() + time.minute() / 60.0)));
80    painter.drawConvexPolygon(hourHand, 3);
81    painter.restore();
82
```

```
 83     painter.setPen(hourColor);
 84     for (int i = 0; i < 12; ++i) {
 85         painter.drawLine(88, 0, 96, 0);
 86         painter.rotate(30.0);
 87     }
 88
 89     painter.setPen(Qt::NoPen);
 90     painter.setBrush(minuteColor);
 91     painter.save();
 92     painter.save();
 93
 94     painter.rotate(6.0 * (time.minute() + time.second() / 60.0));
 95     painter.drawConvexPolygon(minuteHand, 3);
 96
 97     painter.restore();
 98     painter.setPen(secondColor);
 99     painter.rotate( 6.0 * time.second() );
100     painter.drawConvexPolygon(secondHand, 3);
101     painter.restore();
102
103     painter.setPen(minuteColor);
104     for (int j = 0; j < 60; ++j) {
105         if ((j % 5) != 0)
106             painter.drawLine(92, 0, 96, 0);
107         painter.rotate(6.0);
108     }
109 }
```

Листинг 13.10. Файл chap13/clock/main.cpp

```
110 #include <QApplication>
111 #include "clock.h"
112
113 int main( int argc, char *argv[] ) {
114     QApplication app(argc, argv);
115     Clock clock;
116     clock.show();
117     return app.exec();
118 }
```

Пояснения к программе:

- (13, 41–109) — метод `paintEvent()` отвечает за внешний вид нашего виджета;

- (17) — мы хотим генерировать сигнал `timeChanged()` каждую секунду;
- (26–28) — создали таймер, который будет срабатывать каждую секунду, и связали сигнал о его срабатывании с методом-слотом `update()`;
- (33–39) — в методе `update()` узнаем текущее время, формируем его строковое представление, генерируем сигнал `timeChanged()` и вызываем перерисовку нашего виджета;
- (42–56) — изображения часовой, минутной и секундной стрелок будут формироваться в виде треугольников по трем заданным здесь точкам;
- (57–59) — задали цвет каждой стрелки;
- (61) — диаметр окружности для рисования циферблата;
- (63, 64) — создали объект рисования `QPainter` и установили режим антиалиасинга;
- (66–70) — написали текущее время цифрами в нижней части будущего циферблата;
- (72, 73) — перенесли начало отсчета в середину будущего циферблата и установили масштаб;
- (75, 76) — сбросили перо и выбрали кисть (треугольник часовой стрелки не будет иметь выделенной границы);
- (78) — сохранили текущий режим рисования в стеке;
- (79, 80) — повернули объект рисования под нужным углом и отобразили часовую стрелку, задав фиксированные координаты (можно было, наоборот, пересчитать координаты самой стрелки);
- (81) — восстановили режим, сохраненный в стеке ранее (т. е. повернули объект рисования назад);
- (83–87) — нарисовали двенадцать делений, каждый раз поворачивая объект рисования на 30 градусов;
- (89–92) — выбрали цвет кисти для рисования минутной стрелки и дважды сохранили в стек текущий режим;
- (94–97) — повернули холст, нарисовали минутную стрелку и вернули сохраненный режим;
- (98–101) — повернули холст, нарисовали секундную стрелку и вернули сохраненный режим;
- (103–108) — нарисовали 60 делений.

13.6. Класс *QPixmap*

Как мы уже говорили, класс `QPixmap` предназначен для быстрого вывода на экран графических изображений. Для создания объекта `QPixmap` используется конструктор

```
QPixmap::QPixmap( const QString& fileName, // Имя файла
                  const char *format = 0,  // Формат изображения
                  Qt::ImageConversionFlags flags = Qt::AutoColor )
```

В качестве первого параметра может быть указано имя файла на диске или имя ресурса. Второй параметр необязателен, он задает формат изображения. Если файл не существует или имеет незнакомый формат, то создается пустое изображение. Допускается указывать следующие форматы:

- BMP — формат Windows Bitmap;
- GIF — Graphic Interchange Format;
- JPG, JPEG — Joint Photographic Experts Group;
- PNG — Portable Network Graphics;
- PBM — Portable Bitmap;
- PGM — Portable Graymap;
- PPM — Portable Pixmap;
- XBM — X11 Bitmap;
- XPM — X11 Pixmap.

Для изображений, сохраненных в формате GIF, PBM и PGM, библиотека Qt поддерживает только чтение, для остальных — и чтение, и запись. Если формат не указан, то делается попытка определить его автоматически, исходя из содержимого файла.

Замечание

Если предполагается работать с графическими изображениями, представленными в формате GIF, то перед компиляцией библиотеки Qt при запуске `configure` следует указать опцию `-qt-gif`.

Последний, также необязательный параметр конструктора — это комбинация флагов, задающая параметры преобразования, которое необходимо выполнить после загрузки изображения из файла:

- `Qt::AutoColor` — если в файле содержится монохромное изображение, то создаваемый в памяти объект также будет монохромным;

- `Qt::ColorOnly` — рисунок подвергается преобразованию, чтобы глубина цвета стала равна значению `QPixmap::defaultDepth()`, используемому по умолчанию;
- `Qt::MonoOnly` — создается монохромное изображение, при этом допускается указывать дополнительные флаги, определяющие алгоритм преобразования.

Имеется вариант конструктора `QPixmap` без параметров, в этом случае создается пустое изображение, которое затем может быть заполнено заданным рисунком из файла с помощью метода

```
bool QPixmap::load(
   const QString & fileName,        // Имя файла
   const char * format = 0,         // Формат
   Qt::ImageConversionFlags flags = Qt::AutoColor )
```

или заданным потоком двоичных данных (листинг 13.11).

Листинг 13.11

```
bool QPixmap::loadFromData(
   const uchar * data,       // Данные
   uint len,                 // Длина
   const char * format = 0, // Формат
   Qt::ImageConversionFlags flags = Qt::AutoColor )
bool QPixmap::loadFromData(
   const QByteArray & data, // Данные
   const char * format = 0,
   Qt::ImageConversionFlags flags = Qt::AutoColor )
```

Третий вариант конструктора

```
QPixmap::QPixmap( int width,     // Ширина
                  int height )   // Высота
```

позволяет создать пустой рисунок заданного размера (в пикселах), который затем можно начинать "рисовать" (листинг 13.12), а потом при необходимости сохранить, вызвав метод `save()`.

Листинг 13.12

```
QPixmap pixmap( 200, 100 );
pixmap.fill( Qt::blue );
QPainter painter( &pixmap );
```

```
painter.setPen( Qt::red );
painter.setBrush( Qt::yellow );
QPolygon polyPoints;
polyPoints << QPoint( 10, 20 )
           << QPoint( 30, 40 )
           << QPoint( 50, 60 )
           << QPoint( 70, 80 );
QVector<QPoint> linePoints;
foreach( QPoint point, polyPoints )
   linePoints << point + QPoint( 80, 0 );
painter.drawLine( QPoint( 10, 10 ), QPoint( 30, 90 ) );
painter.drawPolyline( polyPoints );
painter.drawLines( linePoints );
```

13.7. Работа с принтером

Для работы с принтером в Qt используется класс `QPrinter`, созданный на основе класса `QPaintDevice`, поэтому вывод на печать производится так же, как и вывод на экран. Для управления режимами печати имеются следующие параметры:

- цветность `ColorMode`:
  - `QPrinter::Color` — режим цветной печати;
  - `QPrinter::GrayScale` — режим печати в оттенках серого;
- ориентация страницы `Orientation`:
  - `QPrinter::Portrait` — вертикальная;
  - `QPrinter::Landscape` — горизонтальная;
- выходной формат `OutputFormat`:
  - `QPrinter::NativeFormat` — используется "родной" метод вывода на печать, зависящий от текущей платформы: в системах Windows и Mac OS — системный драйвер принтера, а в Linux — либо CUPS (*Common Unix Printing System*), либо сначала создается файл PostScript, а затем запускается стандартная утилита lpr или любая другая, установленная с помощью метода `QPrinter::setPrintProgram()`;
  - `QPrinter::PdfFormat` — генерируется файл PDF;
  - `QPrinter::PostScriptFormat` — генерируется файл PostScript.

  Если печать перенаправляется в файл, то нужно задать имя файла с помощью метода `setOutputFileName()`;

- порядок печати страниц `PageOrder`:
  - `QPrinter::FirstPageFirst` — по порядку;
  - `QPrinter::LastPageFirst` — в обратном порядке;
- размер страниц `PageSize`:
  - `QPrinter::A0` — формат A0 (841×1189 мм);
  - ...
  - `QPrinter::B0` — формат B0 (1030×1456 мм);
  - `QPrinter::Custom` — произвольный размер;
- источник бумаги `PaperSource`: `QPrinter::Auto`, `QPrinter::Manual`, `QPrinter::Cassette` и т. д.
- диапазон печатаемых страниц `PrintRange`: `QPrinter::AllPages`, `QPrinter::Selection`, `QPrinter::PageRange`;
- разрешение `PrinterMode`: `QPrinter::ScreenResolution`, `QPrinter::HighResolution`;
- состояние принтера `PrinterState`: `QPrinter::Idle`, `QPrinter::Active`, `QPrinter::Aborted`, `QPrinter::Error`.

Для установки параметров страницы имеется класс `QPageSetupDialog`:

```
QPageSetupDialog dlg(&printer, parent);
bool f = (dlg.exec() == QDialog::Accepted);
```

Если пользователь отказался от выбора параметров, то возвращается значение `false`. В системах Windows и Mac OS используется "родной" диалог установки параметров страницы, но он не подходит при печати в файл PDF или PostScript.

ЗАДАЧА. Напечатать простое изображение с текстом. Предусмотреть выбор принтера и разрешение, а также возможность печати в файл PDF или PostScript.

Решение. На рис. 13.4 показаны диалоги, которые выводит на экран наша программа. При запуске сначала выводится окно с единственной кнопкой **Печать**, при нажатии на которую пользователю предоставляют возможность выбрать один из трех режимов печати: на принтер, в файл PDF или файл PostScript. В первом случае на экран выводится диалог выбора принтера (рис. 13.5), а затем — список разрешений, которые поддерживает указанный пользователем принтер (см. рис. 13.4). В остальных двух случаях разрешение выбирается произвольно. В результате мы получаем страницу, показанную на рис. 13.6. Листинги 13.13 и 13.14 содержат текст программы.

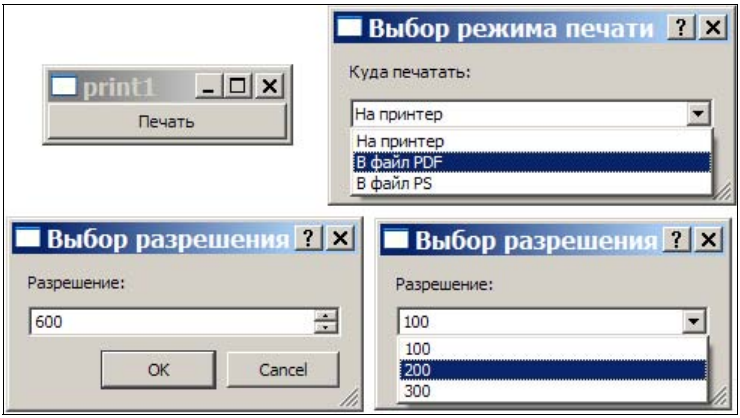


Рис. 13.4. Внешний вид окна приложения и его диалогов

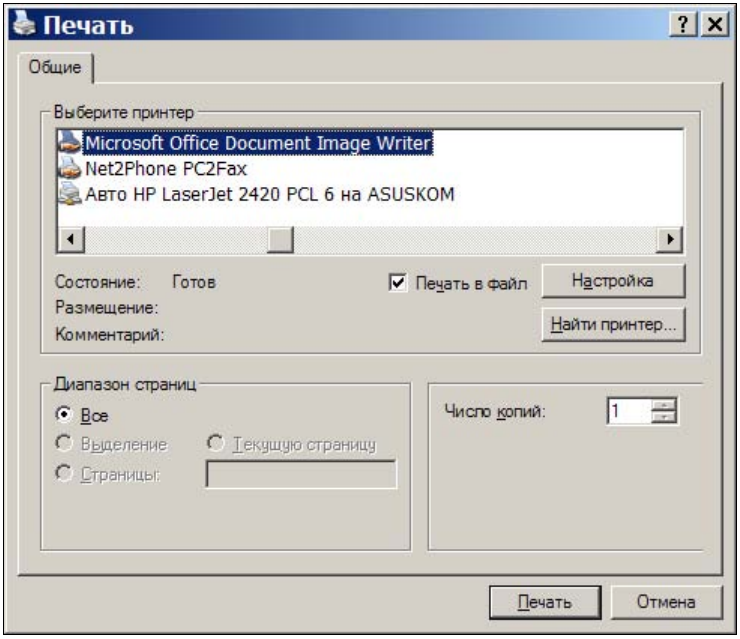


Рис. 13.5. Диалог выбора принтера: можно печатать в файл

Листинг 13.13. Файл chap13/print1/print1.h

```
1  #include <QtGui>
2
3  class MyWin : public QMainWindow {
4     Q_OBJECT
```

```
5  public:
6     MyWin();
7  public slots:
8     void onPrint();
9  };
```

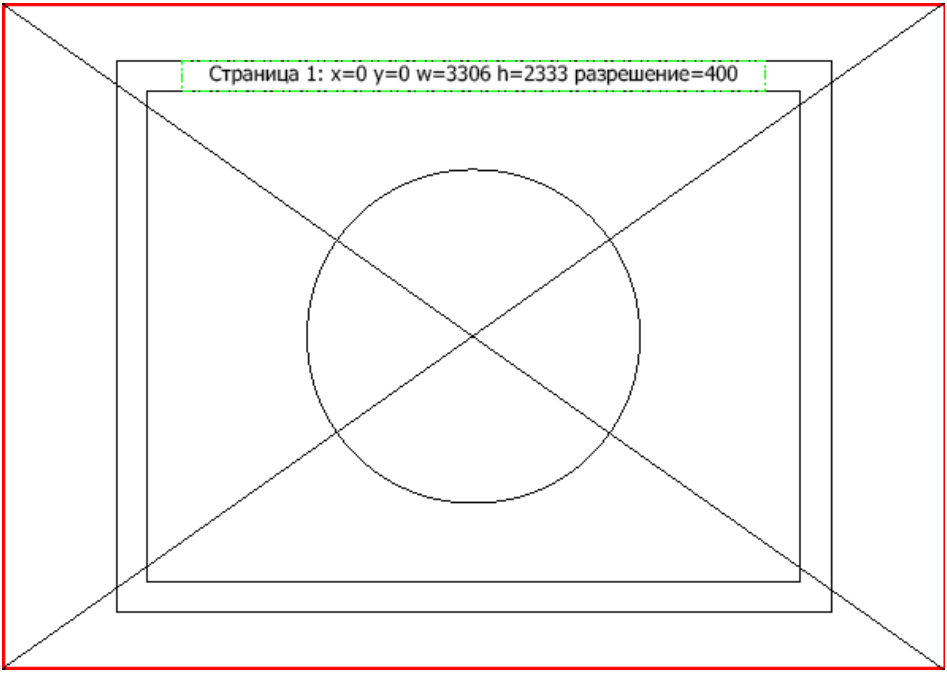


Рис. 13.6. Печатаемая страница

Листинг 13.14. Файл chap13/print1/print1.cpp

```
10 #include "print1.h"
11 #define N  3  // Кол-во страниц
12
13 MyWin::MyWin() : QMainWindow() {
14   QPushButton *btn = new QPushButton(tr("&Печать"), this);
15   setCentralWidget(btn);
16   resize(btn->sizeHint());
17   connect(btn,  SIGNAL(clicked()),
18           this, SLOT(onPrint()));
19 }
20
```

```
21  void MyWin::onPrint(){
22    QPrinter printer(QPrinter::HighResolution);
23    printer.setPageSize(QPrinter::A5);
24    printer.setColorMode(QPrinter::Color);
25    printer.setOrientation(QPrinter::Landscape);
26    printer.setFullPage(true);
27
28    QString modePrinter = tr("На принтер");
29    QString modePDF = tr("В файл PDF");
30    QString modePS = tr("В файл PS");
31    QStringList items;
32    items << modePrinter << modePDF << modePS;
33    bool ok;
34    QString mode = QInputDialog::getItem(
35       this, tr("Выбор режима печати"),
36       tr("Куда печатать:"), items, 0, false, &ok);
37    if (!ok || mode.isEmpty())
38       return;
39
40    QString fileName = "";
41    bool f;
42    if (mode == modePDF){
43       printer.setOutputFormat(QPrinter::PdfFormat);
44       fileName = QFileDialog::getSaveFileName(
45           this, tr("Выбор файла"),
46           QString(),
47           tr("Adobe PDF (*.pdf);;Все файлы (*.*)"));
48       f = fileName.isEmpty();
49    }else if (mode == modePS){
50       printer.setOutputFormat(QPrinter::PostScriptFormat);
51       fileName = QFileDialog::getSaveFileName(
52           this, tr("Выбор файла"),
53           QString(),
54           tr("Файлы PostScript (*.ps);;Все файлы (*.*)"));
55       f = fileName.isEmpty();
56   }else{
57       printer.setOutputFormat(QPrinter::NativeFormat);
58       QPrintDialog dlg( &printer, this );
59       f = (dlg.exec() != QDialog::Accepted);
60   }
61   if ( f )
62       return;
63
```

```
64  if (!fileName.isEmpty()){
65      printer.setOutputFileName(fileName);
66      int dpi = QInputDialog::getInteger(
67      this, tr("Выбор разрешения"),
68      tr("Разрешение:"), 600, 75, 999999, 1, &ok);
69      if ( ok )
70          printer.setResolution(dpi);
71  }else{
72      QList<int> res = printer.supportedResolutions();
73      items.clear();
74      foreach (int i, res)
75          items.append(QString("%1").arg(i));
76
77      mode = QInputDialog::getItem(
78         this, tr("Выбор разрешения"),
79         tr("Разрешение:"), items, 0, false, &ok);
80      if ( ok ) {
81         bool flag;
82         int dpi = mode.toInt(&flag);
83         if ( flag )
84         printer.setResolution(dpi);
85      }
86   }
87   QPainter painter;
88   painter.begin(&printer);
89   painter.setPen( Qt::black );
90   painter.save();
91
92   for( int page=0; page<N; page++ ){
93       QPen pen(Qt::red, 14);
94       painter.setPen(pen);
95       QRectF rect = printer.pageRect();
96       painter.drawRect( rect );
97
98       painter.restore();
99       painter.drawLine(
100          printer.pageRect().topLeft(),
101          printer.pageRect().bottomRight() );
102      painter.drawLine(
103          printer.pageRect().topRight(),
104          printer.pageRect().bottomLeft() );
105      int dpi = printer.resolution();
106      QRectF textArea(
107          rect.left() + dpi * 1.0,
```

```
108             rect.top() + dpi * 0.5,
109             rect.width() — dpi * 2.0,
110             rect.height()- dpi * 1.0 );
111         painter.drawRect( textArea );
112         painter.setFont(QFont("Serif", 13));
113         QString s = tr(
114            "Страница %1: x=%2 y=%3 w=%4 h=%5 разрешение=%6" );
115         painter.drawText(
116            textArea, Qt::AlignHCenter | Qt::AlignTop,
117            s.arg( page+1 )
118             .arg(rect.left())
119             .arg(rect.top())
120             .arg(rect.width())
121             .arg(rect.height())
122             .arg(dpi) );
123         qreal x0 = (rect.left() + rect.right()) / 2.0;
124         qreal y0 = (rect.top() + rect.bottom()) / 2.0;
125         qreal d = qMin(x0, y0);
126         QRectF cirect(x0 — d / 2.0, y0 — d / 2.0,  d, d);
127         painter.drawArc(cirect, 0, 16*360);
128
129         QFontMetrics fm = painter.fontMetrics();
130         int h = fm.height() + qRound(dpi*0.05);
131         int w = fm.width(s) + qRound(dpi*0.2);
132         painter.drawRect( textArea.adjusted(h,h,-h,-h) );
133
134         painter.save();
135         pen = QPen(Qt::green, 7, Qt::DashDotLine,
136                    Qt::RoundCap, Qt::RoundJoin);
137         painter.setPen(pen);
138         painter.drawRect(
139             QRectF ( x0 — w / 2.0, textArea.top(),
140                      w,  h ) );
141         if( page < N-1 )
142             printer.newPage();
143     }
144     painter.restore();
145     painter.end();
146 }
147
148 //-----------------------------------------------------
149 int main( int argc, char **argv ){
150     QApplication app( argc, argv );
```

```
151     QTextCodec *codec = QTextCodec::codecForName("CP1251");
152     QTextCodec::setCodecForTr(codec);
153
154     MyWin mw;
155     mw.show();
156     return app.exec();
157 }
```

Пояснения к программе:

- (3–9) — в нашем классе будет определен конструктор и слот, выполняемый при нажатии на кнопку **Печать**;
- (13–19) — в конструкторе создаем главное окно с кнопкой **Печать**, устанавливаем его размеры равными стандартным размерам кнопки и связываем кнопку со слотом `onPrint()`;
- (21–146) — слот `onPrint()` выполняет всю работу по формированию изображения для принтера;
- (22–26) — создаем объект `QPrinter` и задаем параметры печати;
- (28–38) — формируем диалог для выбора способа печати: на принтер, в файл PDF или PostScript. Если пользователь отказался от выбора, то больше ничего не делаем;
- (40–62) — выводим диалог для выбора файла или принтера. Если пользователь отказался от выбора, то больше ничего не делаем;
- (64–86) — выводим диалог для выбора разрешения. При печати на принтер показываем только те режимы, которые поддерживаются принтером, при печати в файл PDF или PostScript разрешение выбирается произвольно;

Замечание

В системе Linux при печати на принтер на самом деле сначала формируется файл PostScript, а список допустимых разрешений всегда состоит только из одного пункта, соответствующего разрешению экрана.

- (87, 88) — создаем объект `QPainter` и устанавливаем контекст для рисования. Эти две строчки можно было бы заменить одной: `QPainter painter(&printer);`
- (89, 90) — устанавливаем черный цвет пера и запоминаем этот режим;
- (92) — тело цикла выполнится столько раз, сколько страниц мы хотим напечатать;
- (93–96) — выбираем красное широкое перо и рисуем прямоугольник, показывающий область, доступную для формирования изображения;

- (98) — восстанавливаем запомненный режим рисования (черное перо обычной толщины);
- (99–104) — по всей области страницы, доступной для рисования, проводим крест-накрест две диагональные линии;
- (105–111) — отступаем от границ области рисования 1 дюйм слева и справа, и полдюйма сверху и снизу; рисуем черный прямоугольник;
- (112–122) — выбираем гарнитуру и размер шрифта и формируем строку, в которой содержится информация о номере страницы, используемом разрешении (в пикселах на дюйм) и размерах области рисования (в пикселах);
- (123–125) — вычисляем координаты центра области рисования. Определяем минимальное значение, чтобы определить диаметр окружности, которую мы хотим поместить в центр страницы;
- (126, 127) — задаем квадратную область (координаты левого верхнего угла, ширину и высоту) и рисуем окружность. Значение 16×360 соответствует 360 градусам;
- (129–132) — вычисляем высоту и ширину прямоугольной области, которую занимает текстовая строка (114). Отступаем вниз на высоту текстовой строки и рисуем второй черный прямоугольник внутри первого;
- (134–140) — запоминаем текущие настройки, снова меняем перо и рисуем толстыми синими штрихпунктирными линиями прямоугольник вокруг текстовой строки;
- (141, 142) — вставляем разрыв страницы;
- (145) — завершаем формирование страниц для печати (метод `end()` будет вызван автоматически при уничтожении объекта `QPainter`, но мы решили перестраховаться).

ЗАДАЧА. Распечатать внешний вид виджета, как он показан на экране.

Решение данной задачи показано в листинге 13.15. Для примера мы вывели на принтер табличную часть виджета `QTableWidget` (рис. 13.7), но аналогично можно было распечатать всю форму вместе с кнопкой или таблицу с заголовками строк и столбцов.

Листинг 13.15. Файл chap13/printwidget/printwidget.cpp

```
#include <QtGui>

class Widget : public QWidget {
    Q_OBJECT
```

```
public:
    Widget(){
        QPushButton *button = new QPushButton(this);
        button->setText("Print me");
        button->setIcon(style()->
            standardIcon(QStyle::SP_DialogApplyButton));

        m_table = new QTableWidget(4, 3, this);
        m_table->setItem( 0, 1,
                          new QTableWidgetItem("All right!"));
        m_table->setItem( 2, 2,
                          new QTableWidgetItem("Ok!"));

        QVBoxLayout *layout = new QVBoxLayout(this);
        layout->addWidget(m_table);
        layout->addWidget(button);
        adjustSize();

        connect(button, SIGNAL(clicked()), this, SLOT(print()));
    }

    QTableWidget *m_table;

public slots:
    void print() {
        QPrinter *printer = new QPrinter;
        QPrintDialog *printDialog = new QPrintDialog( printer,
                                                      this );
        if (printDialog->exec() == QDialog::Accepted) {
            QPainter p(printer);
            QPixmap pm = QPixmap::grabWidget(
                m_table->viewport() );
            p.drawPixmap(0, 0, pm);
        }
    }
};

#include "printwidget.moc"

int main(int argc, char **argv){
    QApplication app(argc, argv);
    Widget win;
    win.show();
    return app.exec();
}
```

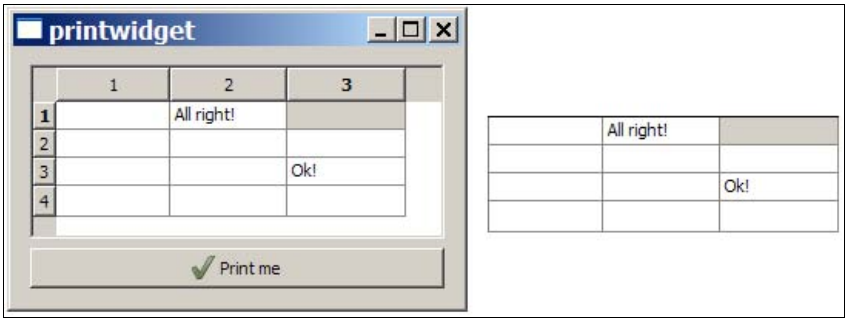


Рис. 13.7. Внешний вид окна приложения (слева) и распечатка (справа)

Виджет печатается именно так, как он в данный момент выглядит на экране. Например, если объект `QTableWidget` не помещается внутри области прокрутки, то при печати мы также увидим лишь часть таблицы. В большинстве случаев это не совсем то, что хочет пользователь при выводе таблицы на печать, поэтому мы вернемся к данному вопросу при рассмотрении методов построения отчетов для баз данных.

ЗАДАЧА. Вывести на принтер графическое изображение `QImage`.

Решение иллюстрирует листинг 13.16.

Листинг 13.16

```
void MyWin::printImage(const QImage &image) {
   QPrinter printer;
   QPrintDialog printDialog = QPrintDialog(printer);
   if ( printDialog.exec() ) {
      QPainter painter(printer);
      QRect rect = painter.viewport();
      QSize size = image.size();
      size.scale( rect.size(),  // масштаб по размерам страницы
                  Qt::KeepAspectRatio ); // сохранять пропорции
      painter.setViewport( rect.x(), rect.y(),
                           size.width(), size.height() );
      painter.setWindow(image.rect());
      painter.drawImage(0, 0, image);
   }
}
```

13.8. Печать на нескольких страницах

Общий алгоритм вывода на печать нескольких страниц:

1. Создать объект `QPrinter`.
2. При необходимости вывести на экран диалог `QPrintDialog`, позволяющий пользователю выбрать принтер и установить параметры печати.
3. Создать объект `QPainter`.
4. С его помощью нарисовать содержимое страницы.
5. Вызвать `QPrinter::newPage()` для перехода к следующей странице.
6. Повторять пункты 4–5 для всех страниц.

Если выводимый на печать документ является обычным неформатированным текстом или представлен в формате HTML, то необходимая разбивка на страницы обеспечивается автоматически методами `QTextDocument::print()`, `QTextEdit::print()`, `QWebFrame::print()` и `QWebView::print()`. В качестве единственного параметра все эти методы принимают указатель на объект `QPrinter`. Если необходимо распечатать что-то более сложное, то придется заниматься разбивкой материала и отрисовкой каждой страницы вручную.

ЗАДАЧА. Доработать программу "Простой текстовый редактор", которую мы рассматривали в *разд. 12.1–12.3*, добавив возможность вывода текстового документа на печать.

Решение. К счастью, в данном случае нам не придется вручную вычислять, влезет ли очередная строка на текущую страницу или уже пора сделать перенос на следующую: метод `QTextDocument::print()` выполнит эту работу самостоятельно. Нам останется только:

- объявить новые действия в разделе `private` класса `MainWin`:

```
QAction *printAct;
QAction *printPdfAct;
```

- в методе `MainWin::createActions()` создать эти действия и связать их с какими-нибудь слотами (листинг 13.17).

Листинг 13.17

```
printAct = new QAction(tr("&Print"), this);
printAct->setStatusTip(tr("Print the document"));
connect(printAct, SIGNAL(triggered()), this, SLOT(print()));
```

```
printPdfAct = new QAction(tr("Print to PDF &file"), this);
printPdfAct->setStatusTip(tr("Print the document to PDF file"));
connect(printPdfAct, SIGNAL(triggered()), this, SLOT(printPdf()));
```

- в методе `MainWin::createMenus()` создать соответствующие пункты меню:

```
fileMenu->addSeparator();
fileMenu->addAction(printAct);
fileMenu->addAction(printPdfAct);
```

- в разделе `private slots` класса `MainWin` дописать заголовки двух слотов:

```
void print();
void printPdf();
```

- и, наконец, реализовать эти слоты (листинг 13.18).

Листинг 13.18

```
void MainWin::print(){
   QPrinter printer(QPrinter::HighResolution);
   QPrintDialog *dlg = new QPrintDialog(&printer, this);
   if (textEdit->textCursor().hasSelection())
      dlg->addEnabledOption(QAbstractPrintDialog::PrintSelection);

   dlg->setWindowTitle(tr("Print document"));
   if (dlg->exec() == QDialog::Accepted)
      textEdit->print(&printer);

   delete dlg;
}

void MainWin::printPdf(){
    QString fileName = QFileDialog::getSaveFileName(
        this, "Export PDF", QString(), "*.pdf");
    if (!fileName.isEmpty()) {
        if (QFileInfo(fileName).suffix().isEmpty())
            fileName.append(".pdf");

        QPrinter printer(QPrinter::HighResolution);
        printer.setOutputFormat(QPrinter::PdfFormat);
        printer.setOutputFileName(fileName);
        textEdit->document()->print(&printer);
    }
}
```

Полные исходные тексты этого проекта можно найти в каталоге chap13/printtext.

13.9. Предварительный просмотр перед печатью

Начиная с релиза 4.4, в библиотеке Qt появились новые классы `QPrintPreviewDialog` и `QPrintPreviewWidget`, обеспечивающие предварительный просмотр на экране выводимых на принтер страниц с возможностью изменения параметров печати. Программисту обычно приходится напрямую работать с первым классом, который, в свою очередь, задействует второй для непосредственного отображения просматриваемых страниц.

Общий алгоритм использования класса `QPrintPreviewDialog`:

- создать экземпляр этого класса:
  - с указанием существующего объекта `QPrinter`:

```
QPrintPreviewDialog ( QPrinter *printer,
                      QWidget * parent = 0,
                      Qt::WindowFlags flags = 0 )
```

  - или безотносительно к объекту печати:

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

- связать сигнал

```
void QPrintPreviewDialog::paintRequested ( QPrinter * printer )
```

  со слотом, который будет формировать выводимое изображение или текст;

- вызвать метод `QPrintPreviewDialog::exec()`.

Разумеется, слот, используемый для диалога просмотра, должен формировать точно такое же изображение, что и для настоящего вывода на принтер. Поэтому лучше всего, если это будет один и тот же фрагмент кода.

ЗАДАЧА. Реализовать возможность предварительного просмотра страниц перед их выводом на принтер для текстового редактора, рассмотренного в *разд. 12.1–12.3* и *13.8* (рис. 13.8).

Решение. Нам потребуется описать два новых слота:

```
void previewDialog();
void paintPreview(QPrinter *printer);
```

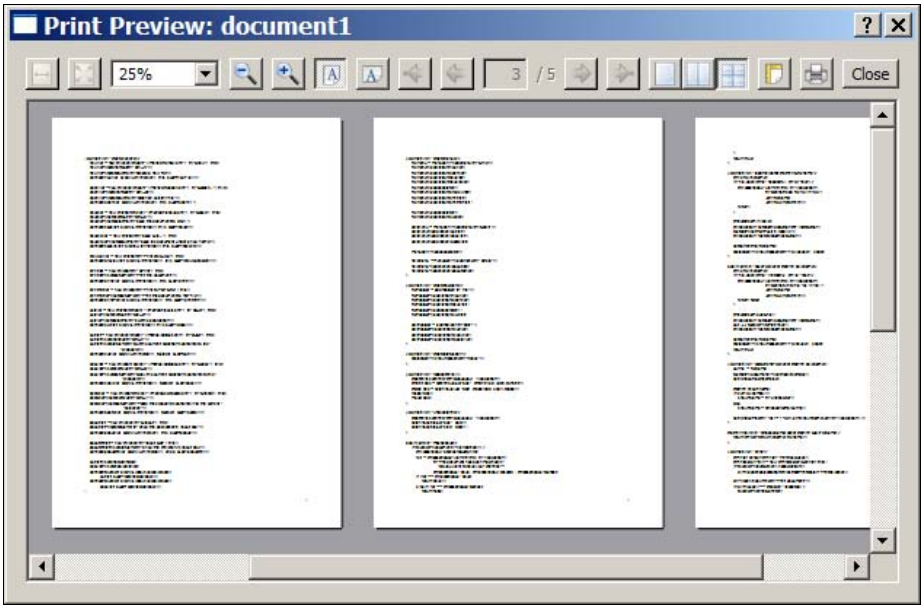


Рис. 13.8. Диалог предварительного просмотра перед печатью документа

пункт меню:

```
fileMenu->addAction(previewAct);
```

и одно действие:

```
QAction *previewAct;
```

Затем связать это действие с первым слотом:

```
previewAct = new QAction(tr("Print pre&view"), this);
connect( previewAct, SIGNAL(triggered()),
         this, SLOT(previewDialog()) );
```

и, наконец, реализовать оба слота, не забыв выполнить три пункта нашего алгоритма (листинг 13.19).

Листинг 13.19

```
 void MainWin::previewDialog(){
    QPrinter printer(QPrinter::HighResolution);
    QPrintPreviewDialog preview(&printer, this); // создали диалог
    connect(&preview, SIGNAL(paintRequested(QPrinter*)),
            SLOT(paintPreview(QPrinter*))); // указали слот
    preview.exec();                    // вывели диалог на экран
}
```

```
void MainWin::paintPreview(QPrinter *printer){
    textEdit->print(printer);
}
```

Файлы проекта для этой задачи находятся в каталоге `chap13/previewtext`, внешний вид диалога предварительного просмотра показан на рис. 13.8.

Как видим, отрисовкой страниц как при просмотре, так и при печати (см. *разд. 13.8*) занимается один и тот же метод `QTextEdit::print()`, что гарантирует идентичность изображений. Правда, мы не учли одно обстоятельство: объект печати `QPrinter` у диалога `QPrintPreviewDialog` в нашей программе локальный, поэтому пользователь, установив параметры страницы в диалоге просмотра и насладившись качественным изображением документа, должен дать команду на печать именно в этом же диалоге. После его закрытия все настройки удалятся вместе с временным объектом `preview`. Поэтому, вернувшись к основному окну программы, чтобы выполнить распечатку текста из главного меню, пользователь обнаружит, что размеры и ориентация страницы, а также ширина всех полей оказались совсем не теми, что он задал в окне предварительного просмотра. Доработка, исправляющая описанный недостаток, тривиальна, и мы оставляем ее читателю в качестве небольшого упражнения. Один из возможных вариантов решения можно найти в каталоге chap13/previewtext2.

13.10. Графические сцены

Для построения сложных двумерных изображений в библиотеке Qt используются классы `QGraphicsScene` (графическая сцена), `QGraphicsItem` (графический элемент) и `QGraphicsView` (вид графической сцены). В качестве элемента графической сцены могут выступать любые графические примитивы (линии, эллипсы, прямоугольники), текстовые надписи и даже виджеты.

ЗАДАЧА. Построить графическую сцену, состоящую из текстовой надписи, а также виджетов `QLabel` и `QTableWidget`. Отобразить сцену в перспективе (рис. 13.9).

Решение иллюстрирует листинг 13.20.

Листинг 13.20. Файл chap13/scene/scene.cpp

```
#include <QtGui>
int main(int argc, char **argv){
    QApplication app(argc, argv);
    QTextCodec *incodec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(incodec);
```

```
    QGraphicsScene scene;
    QGraphicsTextItem *t1 = scene.addText(
        QObject::tr("Библиотека Qt"),
        QFont("Helvetica", 24) );
    t1->moveBy(50, 30);

    QLabel lb(QObject::tr(
        "<img src=icon.png width=30 />"
        "<font face=Times size=+5 color=#103090>"
        " <u>Библиотека</u> <b>Qt</b> &mdash;<br/>"
        "<i>лучшая</i> в мире!"
        "</font>" ) );
    lb.setStyleSheet("QLabel{background-color: white}");
    QGraphicsProxyWidget *t2 = scene.addWidget(&lb);
    t2->moveBy(30, 100);

    QTableWidget table(2, 3);
    QTableWidgetItem *it = new QTableWidgetItem(
        QObject::tr("Введите свое имя:" ) );
    table.setItem(0, 0, it);
    table.setSpan(0, 0, 1, 3);
    table.resize(270, 120);

    QGraphicsProxyWidget *t3 = scene.addWidget(&table);
    t3->moveBy(10, 180);

    QGraphicsView view(&scene);
    view.setRenderHints( QPainter::Antialiasing |
                         QPainter::TextAntialiasing |
                         QPainter::SmoothPixmapTransform);

    QTransform transform; // Трансформация сцены
    transform.rotate(40, Qt::YAxis); // Поворот вдоль оси Y
    transform.rotate(50, Qt::XAxis); // Поворот вдоль оси X
    transform.rotate(-30, Qt::ZAxis); // Поворот вдоль оси Z
    view.setTransform(transform);    // Применить преобразование
    view.show();
    return app.exec();
}
```

Самое удивительное, что пользователь может редактировать текст в ячейках таблицы (правда, текстовый курсор во время редактирования не показывается).

Рис. 13.9. Виджеты внутри графической сцены

Чтобы внутри надписи отображалась картинка, необходимо подготовить графический файл icon.png и поместить его в каталог проекта. Поскольку в системе Windows исполняемый файл формируется к подкаталоге release или debug, следует или скопировать картинку также и в эти подкаталоги, или сделать ее ресурсом, или сказать компоновщику, что исполняемый файл нужно поместить в основном каталоге проекта. Мы выбрали последний вариант (листинг 13.21).

Листинг 13.21. Файл chap13/scene/scene.pro

```
TEMPLATE = app
win32: TARGET = ../scene
DEPENDPATH += .
INCLUDEPATH += .
SOURCES += scene.cpp
```

ЗАДАЧА. Разработать свой элемент графической сцены (рис. 13.10).

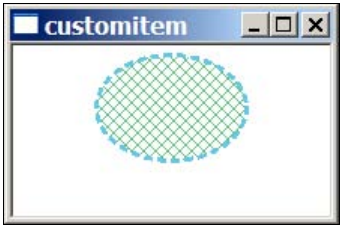


Рис. 13.10. Разработка нового элемента графической сцены

Решение иллюстрирует листинг 13.22.

Листинг 13.22. Файл chap13/customitem1/customitem.cpp

```
#include <QtGui>

class Item : public QGraphicsItem {
public:
    // Конструктор:
    Item(QGraphicsItem *parent = 0)
    : QGraphicsItem(parent) { }

    // Охватывающий прямоугольник:
    QRectF boundingRect() const {
        return QRectF(0, 0, 100, 100);
    }

    // Прорисовка элемента:
    void paint( QPainter *painter,
                const QStyleOptionGraphicsItem * /* option */ ,
                QWidget * /* viewport */ ) {
        QPen pen( QBrush(QColor(100,200,230)), 3, Qt::DotLine );
        painter->setPen(pen); // Установили перо (граница фигуры)
        QBrush brush(QColor(100,200,130), Qt::DiagCrossPattern);
        painter->setBrush(brush); // Установили кисть (фон фигуры)
        painter->drawEllipse(0, 0, 100, 70);
    }
};

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

    QGraphicsScene scene;     // Графическая сцена
    scene.addItem(new Item);  // Добавили свой элемент

    QGraphicsView view(&scene); // Вид сцены
    view.show();                // Показали вид

    return app.exec();
}
```

ЗАДАЧА. Изменять вид элемента графической сцены при нажатии кнопок мыши.

Решение. Добавим поле `down`, которое будет хранить значение `true`, пока над элементом находится указатель мыши с нажатой кнопкой (любой). В методе `paint()` в зависимости от значения `down` выбираем различные кисти и перо (листинг 13.23).

Листинг 13.23. Файл chap13/customitem2/customitem.cpp

```
#include <QtGui>

class Item : public QGraphicsItem {
public:
    Item(QGraphicsItem *parent = 0)
    : QGraphicsItem(parent), down(false) { }

    QRectF boundingRect() const {
        return QRectF(0, 0, 100, 70);
    }

    void paint(QPainter *painter,
               const QStyleOptionGraphicsItem * /* option */ ,
               QWidget * /* viewport */ ) {
        painter->setBrush(down ? Qt::blue : Qt::green);
        QPen pen(
            QBrush( down ? Qt::red : Qt::blue ),
            3, Qt::DotLine );
        painter->setPen(pen);
        painter->drawEllipse(boundingRect());
    }

protected:
    void mousePressEvent(QGraphicsSceneMouseEvent * /* event */ ) {
        if (!down) {
            down = true;
            update();
        }
    }
    void mouseReleaseEvent(QGraphicsSceneMouseEvent *event) {
        if (event->buttons() == 0) {
            down = false;
            update();
        }
    }
```

```
private:
    bool down;
};
```

Текст главной программы `main()` останется прежним.

ЗАДАЧА. В предыдущем решении имеется небольшой изъян: сам элемент имеет овальную форму, но его активная область, воспринимающая щелчки мыши, — прямоугольная. Требуется исправить этот недостаток.

Решение. Переопределим виртуальный метод `Item::shape()`, который используется при определении формы элемента:

```
QPainterPath shape() const {
    QPainterPath path;
    path.addEllipse(boundingRect());
    return path;
}
```

Доработанный с учетом данного изменения текст программы можно найти в файле `customitem3/customitem.cpp`.

ЗАДАЧА. Отобразить множество графических элементов в перспективе, при наведении указателя мыши на элемент — изменять его цвет (рис. 13.11). Разрешить перемещение элементов с помощью мыши.

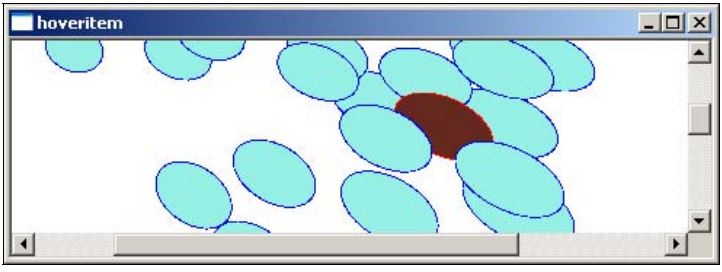


Рис. 13.11. Элементы изменяют свой цвет при наведении на них указателя мыши

Решение иллюстрирует листинг 13.24.

Листинг 13.24. Файл chap13/hoveritem/hoveritem.cpp

```
#include <QtGui>

class Item : public QGraphicsItem {
public:
```

```
    // Конструктор:
    Item(QGraphicsItem *parent = 0)
        : QGraphicsItem(parent) {
        // Реагировать на событие "Указатель мыши над элементом":
        setAcceptsHoverEvents(true);
    }

    // Охватывающий прямоугольник:
    QRectF boundingRect() const {
        return QRectF(0, 0, 70, 50);
    }

    // Активная поверхность элемента:
    QPainterPath shape() const {
        QPainterPath path;
        path.addEllipse(boundingRect());
        return path;
    }

    // Прорисовка элемента:
    void paint(QPainter *painter,
               const QStyleOptionGraphicsItem *option,
               QWidget * /* viewport */ ) {
        if (option->state & QStyle::State_MouseOver){
            painter->setBrush(QColor(100, 40, 30));
            painter->setPen(Qt::red);
        }else{
            painter->setBrush(QColor(150, 240, 230));
            painter->setPen(Qt::blue);
        }
        painter->drawEllipse(boundingRect());
    }
};

int main(int argc, char **argv) {
    QApplication app(argc, argv);
    QGraphicsScene scene;
    for (int i = 0; i < 30; ++i) {
       Item *item = new Item;
       item->setPos(-200 + qrand() % 400,
                    -200 + qrand() % 400);
       item->setFlag(QGraphicsItem::ItemIsMovable);
       scene.addItem(item);
    }
```

```
    QGraphicsView view(&scene);
    QTransform transform; // Трансформация сцены
    transform.rotate(40, Qt::YAxis); // Поворот вдоль оси Y
    transform.rotate(50, Qt::XAxis); // Поворот вдоль оси X
    transform.rotate(30, Qt::ZAxis); // Поворот вдоль оси Z
    view.setTransform(transform);    // Применить преобразование
    view.show();

    return app.exec();
}
```

ЗАДАЧА. Вывести на принтер графическую сцену `QGraphicsScene`.

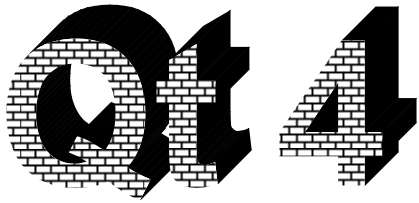
Решение иллюстрирует листинг 13.25.

Листинг 13.25

```
QGraphicsScene scene( QRect( -100, -50, 400, 200 ) );
scene.addText("Hello, world!");
...
QPrinter printer;
QPrintDialog dlg( &printer );
if( dlg.exec() ) {
   QPainter painter( &printer );
   scene.render( &painter, printer.pageRect(),
                 scene.sceneRect(),
                 Qt::KeepAspectRatio );
}
```

Предлагаем читателю самостоятельно на основе данного примера написать программу для предварительного просмотра и вывода на печать содержимого таблицы `QTableWidget` (см. также листинг 13.15).

ГЛАВА 14



Списки, таблицы и деревья

Нужно иметь в голове великое множество разнообразнейших идей, чтобы родить одну хорошую.
Луи Себастьян Мерсье

Как и в любой другой GUI-библиотеке, в Qt имеются средства для отображения списков (одномерных массивов данных), таблиц (двумерных массивов) и деревьев (иерархических таблиц).

14.1. Список *QListWidget*

Для отображения одномерного набора массива элементов существуют классы `QListWidget` и `QListView`. Выглядят они одинаково, но первый сам хранит все свои элементы, а второй использует технологию *"модель — представление"*, которую мы будем рассматривать в *главе 15*.

Каждый элемент списка `QListWidget` — это объект типа `QListWidgetItem`. Имеется два режима отображения списков: `IconMode` (слева направо, с увеличенным размером значков) и `ListMode` (сверху вниз, с уменьшенным размером значков, рис. 14.1).

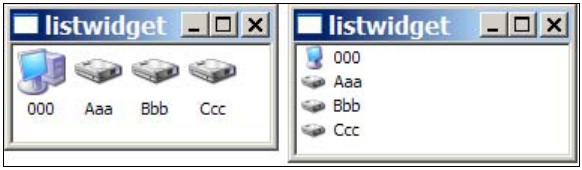


Рис. 14.1. Два режима отображения списка: `QListwidget::IconMode` и `QListwidget::ListMode`

ЗАДАЧА. Вывести на экран список, показанный на рис. 14.1.

Решение. Текст программы приведен в листинге 14.1.

Листинг 14.1. Файл chap14/listwidget/listwidget.cpp

```
 1  #include <QtGui>
 2
 3  int main(int argc, char *argv[]){
 4    QApplication app(argc, argv);
 5
 6    QIcon icon = app.style()->standardIcon(QStyle::SP_DriveHDIcon);
 7
 8    QListWidget listWidget;
 9    new QListWidgetItem(icon, "Aaa", &listWidget);
10    new QListWidgetItem(icon, "Bbb", &listWidget);
11    new QListWidgetItem(icon, "Ccc", &listWidget);
12
13    QListWidgetItem *newItem = new QListWidgetItem;
14    newItem->setText("000");
15    newItem->setIcon(
            app.style()->standardIcon(QStyle::SP_ComputerIcon));
16    listWidget.insertItem(0, newItem);
17
18    listWidget.setViewMode(QListView::ListMode);
19    // или QListView::IconMode
20    listWidget.show();
21    return app.exec();
22  }
```

Пояснения к программе:

- (6) — стандартный значок для будущих элементов списка;
- (8) — создали список;
- (9–11) — добавили в него три новых элемента (список становится родителем для добавляемых элементов);
- (13–16) — другой способ вставки элемента в список;
- (18, 19) — выбрали один из двух режимов отображения списка;
- (20) — вывели список на экран.

14.2. Реакция на действия пользователя

ЗАДАЧА. При двойном щелчке мышью по любому элементу списка требуется выводить на экран дополнительную информацию о данном элементе.

Решение. Для элементов списка определены следующие сигналы:

- `activated(const QModelIndex& index)` — элемент активизирован однократным или двойным (в зависимости от настроек интерфейса) щелчком левой кнопкой мыши или с помощью клавиши <Enter>;
- `clicked(const QModelIndex& index)` — пользователь выполнил однократный щелчок мышью по данному элементу;
- `doubleClicked(const QModelIndex& index)` — пользователь выполнил двукратный щелчок мышью по данному элементу;
- `entered(const QModelIndex& index)` — курсор мыши находится над данным элементом;
- `pressed(const QModelIndex& index)` — кнопка мыши нажата над данным элементом;

и другие.

Во всех случаях параметр `QModelIndex` позволяет определить, какой именно элемент списка сгенерировал данный сигнал. Чтобы выяснить номер активизированного элемента, нужно вызвать метод `QModelIndex::row()`. Это номер строки, начиная с нуля (даже если используется режим `IconView`). Указатель на выбранный элемент возвращает метод `QListWidget::currentItem()`. Если выделено несколько элементов, то их список можно получить с помощью метода `QListWidget::selectedItems()`.

Решение задачи приведено в листингах 14.2 и 14.3, а результат ее работы показан на рис. 14.2.

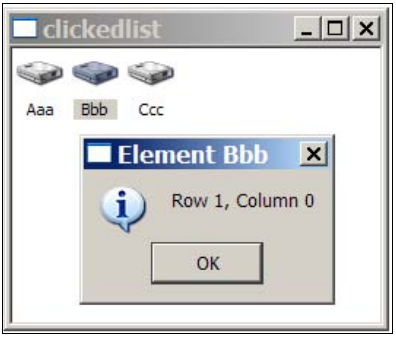


Рис. 14.2. При активизации элемента списка мы получаем сообщение о его координатах

Листинг 14.2. Файл chap14/clickedlist/clickedlist.h

```
#include <QtGui>

class ClickedList : public QListWidget {
    Q_OBJECT
```

```
public:
    ClickedList(QWidget *parent = 0);
public slots:
    void onActivate(const QModelIndex& index);
};
```

Листинг 14.3. Файл chap14/clickedlist/clickedlist.cpp

```
#include "clickedlist.h"

ClickedList::ClickedList(QWidget *parent)
           : QListWidget(parent) {
    connect( this, SIGNAL(activated(const QModelIndex&)),
             this, SLOT(onActivate(const QModelIndex&)) );
}

void ClickedList::onActivate(const QModelIndex& index) {
    QString title = currentItem()->text();
    int row = index.row();
    int col = index.column();
    QMessageBox::information(
        this, tr("Element %1").arg(title),
        tr("Row %1, Column %2").arg(row).arg(col));
}

//-------------------------------------
int main(int argc, char *argv[]){
    QApplication app(argc, argv);

    QIcon icon = app.style()->standardIcon(QStyle::SP_DriveHDIcon);

    ClickedList list;
    new QListWidgetItem(icon, "Aaa", &list);
    new QListWidgetItem(icon, "Bbb", &list);
    new QListWidgetItem(icon, "Ccc", &list);

    list.setViewMode(QListView::IconMode); // QListView::ListMode
    list.show();
    return app.exec();
}
```

При активизации любого из трех элементов списка на экран выводится сообщение, из которого можно узнать название элемента, а также номер строки и

столбца, в котором он находится. Номер столбца всегда равен нулю, независимо от режима отображения списка.

14.3. Редактирование элементов списка

Дополнительные параметры, определяющие поведение элемента списка, задаются с помощью битовых флагов. Текущую комбинацию этих флагов можно получить, вызвав метод `QListWidgetItem::flags()`. Изменение состояния всех флагов производится методом `QListWidgetItem::setFlags()` или путем переопределения самого метода `flags()`. Например, чтобы разрешить пользователю редактировать название элемента списка, нужно установить бит `Qt::ItemIsEditable`. Чтобы сделать элемент неактивным, следует сбросить бит `Qt::ItemIsEnabled`. Чтобы отобразить возле каждого элемента виджет `QCheckBox`, необходимо установить флаг `Qt::ItemIsUserCheckable`. Если кроме двух обычных состояний (выбран/не выбран) разрешено третье состояние (затемненная галочка, частично выбран), то нужно установить бит `Qt::ItemIsTristate`.

Вход в режим редактирования выполняется при активизации элемента или при нажатии клавиши <F2>. По окончании редактирования генерируется сигнал `itemChanged(QListWidgetItem*)`. Новое название элемента доступно с помощью метода `text()`.

14.4. Таблица *QTableWidget*

Для отображения таблиц в Qt используется класс `QTableWidget`. Каждая ячейка таблицы является экземпляром класса `QTableWidgetItem`.

ЗАДАЧА. Разработать таблицу, в ячейках которой размещаются текстовые метки, элементы `QComboBox` и `QCheckBox`. Объединить несколько соседних ячеек в одну (рис. 14.3).

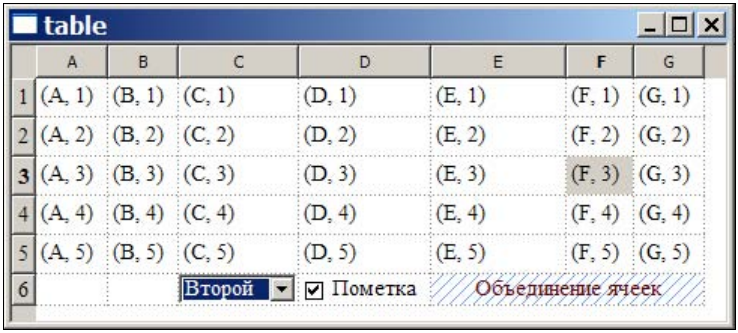


Рис. 14.3. Таблица `QTableWidget`

Решение данной задачи показано в листинге 14.4.

Листинг 14.4. Файл chap14/table/table.cpp

```
#include <QtGui>

#define COLS 7 // количество столбцов таблицы
#define ROWS 5 // количество строк

int main(int argc, char *argv[]) {
  QTextCodec *codec = QTextCodec::codecForName("CP1251");
  QTextCodec::setCodecForTr(codec);

  QApplication app(argc, argv); // приложение

  QTableWidget table(ROWS, COLS);
  table.setGridStyle(Qt::DotLine);
  table.setFont(QFont("Times", 11, QFont::Normal));
  table.resize(550,200);

  for (int i=0; i<COLS; i++) {
     QTableWidgetItem *item = new QTableWidgetItem;
     item->setText(QString(QChar('A' + i)));
     table.setHorizontalHeaderItem(i, item);
     for (int j=0; j<ROWS; j++) {
        QTableWidgetItem *item = new QTableWidgetItem;
        item->setText(QString("(%1, %2)")
                 .arg(QChar('A'+i)) .arg(j+1));
        table.setItem(j, i, item);
     }
  }

  table.insertRow(ROWS);

  QTableWidgetItem *item1 = new QTableWidgetItem;
  item1->setText(QObject::tr("Пометка"));
  item1->setCheckState(Qt::Checked);
  table.setItem(ROWS, 3, item1);

  QComboBox *cb = new QComboBox(&table);
  cb->addItem(QObject::tr("Первый"));
  cb->addItem(QObject::tr("Второй"));
  cb->addItem(QObject::tr("Третий"));
```

```
40    cb->setCurrentIndex(1);
41    table.setCellWidget(ROWS, 2, cb);
42
43    QTableWidgetItem *item2 = new QTableWidgetItem();
44    item2->setText(QObject::tr("Объединение ячеек"));
45    item2->setForeground(QBrush(QColor(100, 10, 10)));
46    item2->setBackground(QBrush(QColor("#70a0ff"),
47                         Qt::BDiagPattern));
48    item2->setTextAlignment(Qt::AlignCenter);
49    table.setSpan(ROWS, 4, 1, 3);
50    table.setItem(ROWS, 4, item2);
51
52    table.setCurrentCell(2, 5);
53
54    table.resizeRowsToContents();
55    table.resizeColumnsToContents();
56
57    table.show();
58    table.setColumnWidth(2, 80);
59    table.setColumnWidth(4, 90);
60    table.setRowHeight(ROWS, 21);
61
62    return app.exec();
63  }
```

Пояснения к программе:

- (12) — создаем таблицу;
- (13) — задаем стиль линий;
- (14) — параметры шрифта;
- (15) — ширина и высота таблицы в пикселах;
- (17) — для каждого столбца таблицы;
- (18) — создаем новую ячейку (ниже она станет заголовком столбца);
- (19) — определяем текст заголовка столбца (буквы латинского алфавита);
- (20) — задаем ячейку-заголовок для `i`-го столбца;
- (21) — для каждой строки таблицы;
- (22) — создаем ячейку, которая будет находиться на пересечении `i`-го столбца и `j`-й строки (сами строки будут нумероваться по умолчанию);
- (23) — текстовая надпись для каждой ячейки;
- (25) — назначаем элемент для ячейки таблицы;

- (29) — добавляем новую строку в конец таблицы;
- (31–34) — вставляем в таблицу ячейку с пометкой `QCheckBox`;
- (36–41) — добавляем в таблицу ячейку с полем выбора `QComboBox`;
- (43–50) — вставляем в таблицу "широкую" ячейку, которая занимает по горизонтали три соседние ячейки, закрашиваем ее фон диагональной штриховкой, определяем цвет и выравнивание надписи;
- (52) — устанавливаем текущую ячейку, в которую будет помещен прямоугольник-курсор при запуске программы;
- (54, 55) — задаем автоматическую установку ширины и высоты ячеек;
- (57) — выводим таблицу на экран;
- (58–60) — изменяем ширину столбцов 2-го и 4-го (считая с нуля), а также высоту последней строки.

ЗАДАЧА. Изменить вид и положение индикатора сортировки в заголовке столбца таблицы `QHeaderView`.

Решение.

Вариант 1. Можно разработать свой класс стиля, в котором переопределить метод `drawPrimitive()`, чтобы указать нужную иконку для индикатора сортировки столбца и его положение (рис. 14.4, листинг 14.5).

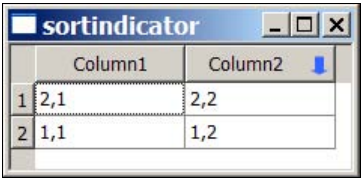


Рис. 14.4. Измененный индикатор сортировки столбца

Листинг 14.5. Файл chap14/sortindicator1/sortindicator.cpp

```
#include <QtGui>

class MyStyle: public QWindowsStyle {
public:
    MyStyle(){ }
    void drawPrimitive(PrimitiveElement element,
                       const QStyleOption *option,
                       QPainter *painter,
                       const QWidget *widget = 0) const {
```

```
        if (element == PE_IndicatorHeaderArrow ) {
            const QStyleOptionHeader *headerOption =
                qstyleoption_cast<const QStyleOptionHeader*>(option);
            int x, y, width, height;
            option->rect.getRect(&x, &y, &width, &height);
            QPixmap pixmap;
            if (headerOption->sortIndicator ==
                                    QStyleOptionHeader::SortUp)
                pixmap.load("up.png");
            else
                pixmap.load("down.png");

            painter->save();
            painter->drawPixmap(x+width-10, y+height-10, pixmap);
            painter->restore();
        }else
            QWindowsStyle::drawPrimitive( element, option,
                                          painter, widget);
    }
};

#define ROWS 3
#define COLS 2

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

    QTableWidget table(2, 2);
    QStringList titles;
    titles << "Column1" << "Column2";
    table.setHorizontalHeaderLabels(titles);
    for (int r=0; r<ROWS; r++){
        for (int c=0; c<COLS; c++){
            QTableWidgetItem *item = new QTableWidgetItem(
                QObject::tr("%1,%2").arg(r+1).arg(c+1) );
            table.setItem(r, c, item);
        }
    }
    table.setSortingEnabled(true);
    app.setStyle(new MyStyle());
    table.resizeRowsToContents();
    table.show();

    return app.exec();
}
```

Вариант 2. Обычно удобнее использовать каскадные таблицы стилей (рис. 14.5, листинг 14.6).

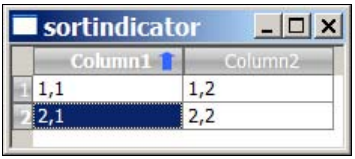


Рис. 14.5. Применение каскадных таблиц стилей к заголовкам столбцов и строк таблицы

Листинг 14.6. Файл chap14/sortindicator2/sortindicator.cpp

```
#include <QtGui>

#define ROWS 2  // количество строк таблицы
#define COLS 2  // количество столбцов

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

    QTableWidget table(ROWS, COLS);
    QStringList titles;
    titles << "Column1" << "Column2";
    table.setHorizontalHeaderLabels(titles);
    for (int r=0; r<ROWS; r++){
        for (int c=0; c<COLS; c++){
            QTableWidgetItem *item = new QTableWidgetItem(
                QObject::tr("%1,%2").arg(r+1).arg(c+1) );
            table.setItem(r, c, item);
        }
    }
    table.setSortingEnabled(true);
    QString s =              // строка с описанием стилей
        " QHeaderView::section {"  // секция заголовка
        "    background-color: qlineargradient("
        "    x1:0, y1:0, x2:0, y2:1,"
        "    stop: 0 #e4e4e4, stop: 0.4 #c0c0c0,"
        "    stop: 0.5 #a8a8a8, stop: 1.0 #d3d3d3);"
        "    color: white;"
        "    padding-left: 4px;"
        "    border: 1px solid #707070;"
        " }"
```

```
        " QHeaderView::down-arrow {"  // индикатор "вниз"
        "    image: url(down.png); width: 18px"
        " }"
        " QHeaderView::up-arrow {"  // индикатор "вверх"
        "    image: url(up.png); width: 18px"
        " }";
    app.setStyleSheet(s);  // применили стили
    table.resizeRowsToContents(); // автоподгонка высоты строк
    table.show();

    return app.exec();
}
```

В обоих случаях графические файлы с изображением стрелок должны находиться в том каталоге, откуда запускается программа. Еще лучше добавить их в виде ресурсов.

ЗАДАЧА. При двойном щелчке кнопкой мыши по ячейке таблицы `QTableWidget` сначала генерируется сигнал `cellClicked()`, а только после этого — сигнал `cellDoubleClicked()`. Как избавиться от первого сигнала, оставив его только для действительно одиночных щелчков кнопки мыши?

Решение. Можно использовать таймер `QTimer`, запуская его каждый раз при появлении сигнала `cellClicked()` и выключая — при срабатывании сигнала `cellDoubleClicked()`. Если в течение некоторого времени второй сигнал так и не получен, то можно считать, что мы имеем дело с одиночным щелчком (рис. 14.6, листинг 14.7).

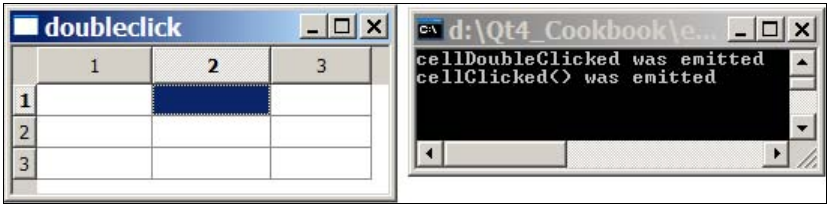


Рис. 14.6. Теперь перед двойным щелчком мыши не сообщается об одиночном щелчке

Листинг 14.7. Файл chap14/doubleclick/doubleclick.cpp

```
#include <QApplication>
#include <QtGui>

class MyTableWidget : public QTableWidget {
    Q_OBJECT
```

```
public:
    MyTableWidget() {
        ignoreNextRelease = false;
        timer = new QTimer(this);
        setColumnCount(3);
        setRowCount(3);
        connect(this, SIGNAL(cellClicked (int, int)),
                this, SLOT(testSlot()));
        connect(this, SIGNAL(cellDoubleClicked (int, int)),
                this, SLOT(testSlot2()));
        connect(timer, SIGNAL(timeout()), this, SLOT(emitClicked()));
    }

    void mouseReleaseEvent(QMouseEvent *event) {
        if (!ignoreNextRelease) {
            timer->start(QApplication::doubleClickInterval());
            blockSignals(true);
            QTableWidget::mouseReleaseEvent(event);
            blockSignals(false);
        }
        ignoreNextRelease = false;
    }

    void mouseDoubleClickEvent(QMouseEvent *event) {
        ignoreNextRelease = true;
        timer->stop();
        QTableWidget::mouseDoubleClickEvent(event);
    }

public slots:
    void testSlot() {
        qDebug("cellClicked() was emitted");   // одиночный щелчок
    }

    void testSlot2() {
        qDebug("cellDoubleClicked was emitted"); // двойной щелчок
    }

    void emitClicked() { // Сгенерировать сигнал об одиночном щелчке:
        emit cellClicked(currentRow(), currentColumn());
        timer->stop();
    }
```

```
private:
    QTimer *timer;
    bool ignoreNextRelease;
};

#include "doubleclick.moc"

int main(int argc, char **argv) {
    QApplication app(argc, argv);
    MyTableWidget table;
    table.show();
    return app.exec();
}
```

Здесь мы не стали выносить объявление класса в отдельный заголовочный файл, поэтому пришлось явно подключить файл *.moc.

Чтобы в системе Windows увидеть консольное окно с сообщениями отладки, выводимыми функцией `qDebug()`, нужно в файл *.pro перед выполнением команды `qmake` добавить строку

```
win32: CONFIG += console
```

ЗАДАЧА. Необходимо дать пользователю возможность сортировать строки таблицы `QTableWidget` по нескольким колонкам одновременно.

Решение задачи показано в листинге 14.8 и на рис. 14.7.

Рис. 14.7. Составное условие сортировки столбцов таблицы

Листинг 14.8. Файл chap14/multisort/multisort.cpp

```
#include <QtGui>

// Макрос для заполнения таблицы данными:
#define SETPERSON(index, first, second, salary) \
```

```
    setItem(index, 0, new QTableWidgetItem(tr(first))); \
    setItem(index, 1, new QTableWidgetItem(tr(second))); \
    setItem(index, 2, new QTableWidgetItem); \
    item(index, 2)->setData(Qt::DisplayRole, salary);

class MyTable : public QTableWidget {
    Q_OBJECT
public:
    MyTable(QWidget *parent = 0)
        : QTableWidget(6, 3, parent) {
        SETPERSON(0, "Иванов", "Иван", 1000000);
        SETPERSON(1, "Иванов", "Петр", 1100000);
        SETPERSON(2, "Иванов", "Сергей", 1300000);
        SETPERSON(3, "Петров", "Иван", 1000000);
        SETPERSON(4, "Сергеев", "Антон", 10000);
        SETPERSON(5, "Сергеев", "Петр", 1000000);
        updateSortOrder();
        connect( horizontalHeader(), SIGNAL(sectionClicked(int)),
                 this, SLOT(onHeaderClicked(int)));
        disconnect( horizontalHeader(), SIGNAL(sectionPressed(int)),
                    this, SLOT(selectColumn(int)));

    }

    void updateSortOrder() {
        QStringList list;
        list << tr("Фамилия") << tr("Имя") << tr("Оклад");
        for (int i=0; i<sortOrder.size(); ++i)
            list[sortOrder.at(i).column].append("("
                  + QString::number(i + 1) + ")");
        setHorizontalHeaderLabels(list);
        for (int i=sortOrder.size() - 1; i >= 0; --i) {
            sortItems(sortOrder.at(i).column,
                      sortOrder.at(i).ascending ?
                         Qt::AscendingOrder : Qt::DescendingOrder);
        }
    }
public slots:
    void onHeaderClicked(int section) {
        bool ascending = true;
        if (!(QApplication::keyboardModifiers()
              & Qt::ControlModifier)
              || sortOrder.isEmpty()) {
```

```
            if (!sortOrder.isEmpty()
                  && sortOrder.first().column == section) {
                ascending = !sortOrder.first().ascending;
            }
            sortOrder.clear();
        } else {
            const int index = findSection(section);
            if (index != -1) {
                if (index == sortOrder.size() - 1) {
                    ascending = !sortOrder.last().ascending;
                }
                sortOrder.removeAt(index);
            }
        }
        sortOrder.append(SortData(section, ascending));
        updateSortOrder();
    }
private:
    int findSection(int section) const {
        for (int i=0; i<sortOrder.size(); ++i) {
            if (sortOrder.at(i).column == section)
                return i;
        }
        return -1;
    }
    struct SortData {
        SortData(int sec = -1, bool asc = true)
            : column(sec), ascending(asc) {}

        int column;
        bool ascending;
    };
    QList<SortData> sortOrder;
};

#include "multisort.moc"

int main(int argc, char **argv){
    QApplication app(argc, argv);
    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);
```

```
    MyTable table;
    table.resizeRowsToContents();
    table.show();

    return app.exec();
}
```

Если щелкать мышью по колонке при нажатой клавише <Ctrl>, то данная колонка добавляется к уже имеющемуся условию сортировки. Повторный щелчок по той же колонке меняет порядок на обратный. Щелчок без нажатия клавиши <Ctrl> начинает новое условие сортировки, сбрасывая все предыдущие.

14.5. Дерево *QTreeWidget*

Для отображения иерархических таблиц (деревьев) в библиотеке Qt используется класс `QTreeWidget`. Каждый элемент дерева — это объект типа `QTreeWidgetItem`.

ЗАДАЧА. Построить дерево, показанное на рис. 14.8.

Решение. Текст программы приведен в листинге 14.9.

Рис. 14.8. Дерево `QTreeWidget`

Листинг 14.9. Файл chap14/tree/tree.cpp

```
#include <QtGui>

int main(int argc, char *argv[]) {

  QTextCodec *codec = QTextCodec::codecForName("CP1251");
  QTextCodec::setCodecForTr(codec);
  QTextCodec::setCodecForCStrings(codec);
```

```
 8
 9    QApplication app(argc, argv);
10
11    QTreeWidget treeWidget;
12
13    //treeWidget.setColumnCount(3);
14     QStringList columns;
15     columns << "Наименование" << "Цена" << "Дата";
16     treeWidget.setHeaderLabels(columns);
17
18     QList<QTreeWidgetItem *> items;
19     QTreeWidgetItem *item;
20     QTreeWidgetItem *child;
21     for (int i = 1; i <= 5; ++i) {
22         columns.clear();
23         columns
24             << QString("Товар %1").arg(i)
25             << QString("%2 руб.").arg(2*i*i-7*i+12);
26         item = new QTreeWidgetItem(columns);
27         item->setFlags(item->flags() | Qt::ItemIsEditable);
28
29         for (int j = 1; j <= 3; ++j) {
30             columns.clear();
31             columns
32                 << QString("Сорт %1").arg(j)
33                 << QString("%2 руб.").arg(2*i*i-3*j+5)
34                 << QDate(2007, 1, 30-i).toString("dd.MM.yy");
35             child = new QTreeWidgetItem(columns);
36             item->addChild(child);
37         }
38
39         items.append(item);
40         item->setExpanded(true);
41     }
42
43     treeWidget.insertTopLevelItems(0, items);
44     treeWidget.setSortingEnabled(true);
45     treeWidget.sortByColumn(0, Qt::AscendingOrder);
46     treeWidget.setSelectionBehavior(
47         QAbstractItemView::SelectItems);
48     treeWidget.setEditTriggers(
49         QAbstractItemView::AllEditTriggers);
50     treeWidget.show();
```

```
51
52      return app.exec();
53  }
```

В (45) задана сортировка групп и подгрупп по нулевому столбцу. Если пользователь щелкнет мышью по заголовку другого столбца, то элементы дерева будут переупорядочены сначала по возрастанию, а после повторного щелчка — по убыванию значений в данном столбце. К сожалению, по умолчанию доступна только сортировка по алфавиту, поскольку все значения хранятся в виде строк.

ЗАДАЧА. Изменить метод сортировки элементов дерева `QTreeWidgetItem`, например, обеспечить сортировку по числовому представлению или по дате.

Решение. Возможны два способа:

- переопределить для элементов дерева оператор <:

```
bool QTreeWidgetItem::operator< (const QTreeWidgetItem& other) const
```

- переопределить для дерева функцию `sortByColumn()`.

Первый вариант показан в листингах 14.10 и 14.11.

Листинг 14.10. Файл chap14/sortedtreewidget.h

```
#include <QtGui>

class TreeWidgetItem : public QTreeWidgetItem {
public:
    TreeWidgetItem(int type=Type)
        : QTreeWidgetItem(type) {}
    TreeWidgetItem(QTreeWidget *tree)
        : QTreeWidgetItem(tree) {}
    TreeWidgetItem(const QStringList& strings)
        : QTreeWidgetItem (strings) {}
    bool operator< (const QTreeWidgetItem& other ) const {
        int sortCol = treeWidget()->sortColumn();
        int pos, num, num2;
        QString s, t, rub(" руб.");
        switch (sortCol) {
            case 0:
                return text(sortCol).toUpper()
                     < other.text(sortCol).toUpper();
            case 1: {
                t = text(sortCol).simplified();
```

```
                pos = t.indexOf(rub);
                if ( pos > 0){
                    s = t.left(pos);
                    num = s.toInt();
                }else
                    num = 0;

                t = other.text(sortCol).simplified();
                pos = t.indexOf(rub);
                if ( pos >= 0){
                    s = t.left(pos);
                    num2 = s.toInt();
                }else
                    num2 = 0;

                return num < num2;
            }
            case 3: {
                QString f("dd.MM.yy");
                return QDate::fromString(text(sortCol), f)
                     < QDate::fromString(other.text(sortCol), f);
            }
            default:
                return false;
        }
    }
};
```

Листинг 14.11. Файл chap14/sortedtreewidget.cpp

```
#include "sortedtreewidget.h"

int main(int argc, char *argv[]) {

    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);
    QTextCodec::setCodecForCStrings(codec);

    QApplication app(argc, argv);
    QTreeWidget treeWidget;

    QStringList columns;
    columns << "Наименование" << "Цена" << "Дата";
    treeWidget.setHeaderLabels(columns);
```

```
    TreeWidgetItem *item;
    TreeWidgetItem *child;
    for (int i = 1; i <= 5; ++i) {
        columns.clear();
        columns
            << QString("Товар %1").arg(i)
            << "" << "";
        item = new TreeWidgetItem(columns);
        item->setFlags(item->flags() | Qt::ItemIsEditable);

        for (int j = 1; j <= 3; ++j) {
            columns.clear();
            columns
                << QString("Сорт %1").arg(j)
                << QString("%2 руб.").arg(4*i*i-2*j+8)
                << QDate(2008, 12-j, 30-i-j).toString("dd.MM.yy");
            child = new TreeWidgetItem(columns);
            item->addChild(child);
            child->setFlags(child->flags() | Qt::ItemIsEditable);
        }

        item->setExpanded(true);
        treeWidget.addTopLevelItem(item);
    }

    treeWidget.setSortingEnabled(true);
    treeWidget.sortByColumn(0, Qt::AscendingOrder);
    treeWidget.setSelectionBehavior(
        QAbstractItemView::SelectItems);
    treeWidget.setEditTriggers(
        QAbstractItemView::AllEditTriggers);
    treeWidget.show();

    return app.exec();
}
```

Следует помнить, что для увеличения скорости вставки элементов разрешать сортировку с помощью метода `setSortingEnabled()` следует после того, как все элементы дерева уже вставлены в него (или добавлять элементы не по одному, а сразу списком).

ЗАДАЧА. Создать дерево, элементы которого имеют пометки `QCheckBox` (рис. 14.9).

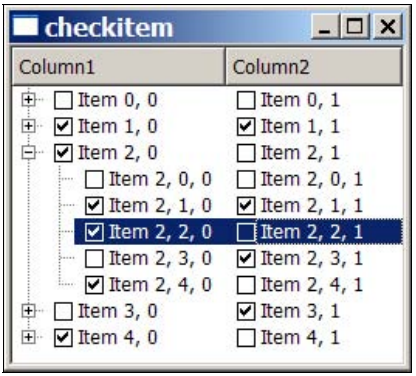


Рис. 14.9. Помечаемые элементы дерева

Решение. Дополнительные опции для элементов дерева задаются с помощью метода `QTreeWidgetItem::setFlags()`. Чтобы отобразить элемент `QCheckBox`, нужно установить флаг `Qt::ItemIsUserCheckable`. Для установки или сброса пометки используется метод `QTreeWidgetItem::setCheckState()`. Листинг 14.12 содержит текст программы.

Листинг 14.12. Файл chap14/checkitem/checkitem.cpp

```
#include <QApplication>
#include <QTreeWidget>

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

    QTreeWidget tree;
    tree.setColumnCount(2); // Число столбцов

    QStringList titles;     // Заголовки столбцов
    titles << "Column1" << "Column2";
    tree.setHeaderLabels(titles);

    QList<QTreeWidgetItem *> items;    // Список пунктов
    QList<QTreeWidgetItem *> subitems; // Список подпунктов
    for (int i = 0; i < 5; ++i){
        QTreeWidgetItem *item = new QTreeWidgetItem(&tree);
        item->setText(0, QObject::tr("Item %1, 0").arg(i));
        item->setText(1, QObject::tr("Item %1, 1").arg(i));
        item->setFlags(item->flags()|Qt::ItemIsUserCheckable);
        item->setCheckState(0,
                            (i % 3) ? Qt::Checked : Qt::Unchecked);
```

```
        item->setCheckState(1,
                            (i % 2) ? Qt::Checked : Qt::Unchecked);

        for (int j = 0; j < 5; ++j){
            QTreeWidgetItem *subitem = new QTreeWidgetItem(item);
            subitem->setText(0, QObject::tr("Item %1, %2, 0")
                                .arg(i).arg(j));
            subitem->setText(1, QObject::tr("Item %1, %2, 1")
                                .arg(i).arg(j));
            subitem->setFlags(item->flags()|Qt::ItemIsUserCheckable);
            subitem->setCheckState(
                0, (j % 3) ? Qt::Checked : Qt::Unchecked);
            subitem->setCheckState(
                1, (j % 2) ? Qt::Checked : Qt::Unchecked);
            subitems.append(subitem);
        }
        item->addChildren(subitems);
        items.append(item);
        tree.insertTopLevelItems(0, items);
    }
    tree.show();
    return app.exec();
}
```

Если для элементов второго столбца не вызывать метод `setCheckState()`, то пометки будут показаны только в первом столбце.

ЗАДАЧА. Как перехватывать событие, связанное со щелчком мыши по знаку разворачивания группы дерева `QTreeView`/`QTreeWidget`?

Решение. Необходимо переопределить метод `mousePressEvent()`, в котором можно выполнить требуемые действия, если координаты события примерно совпадают с положением знака разворачивания (рис. 14.10, листинг 14.13).

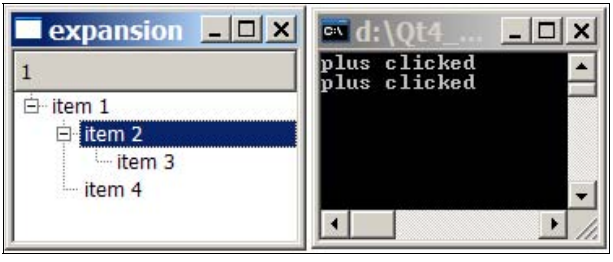


Рис. 14.10. Обработка события "щелчок мыши по знаку сворачивания/разворачивания группы"

Листинг 14.13. Файл chap14/expansion/expansion.cpp

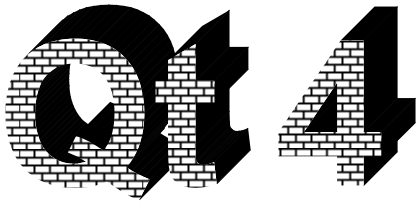
```
#include <QtGui>

class MyTreeWidget : public QTreeWidget {
public:
    MyTreeWidget() {
        setColumnCount(1);
        item1 = new QTreeWidgetItem(this);
        item1->setText(0, "item 1");
        item1->setExpanded(true);

        item2 = new QTreeWidgetItem(item1);
        item2->setText(0, "item 2");
        item2->setExpanded(true);

        item3 = new QTreeWidgetItem(item2);
        item3->setText(0, "item 3");
        item4 = new QTreeWidgetItem(item1);
        item4->setText(0, "item 4");
    }

    void mousePressEvent(QMouseEvent *event) {
        QModelIndex indexClicked = indexAt(event->pos());
        if(indexClicked.isValid()) {
            QRect vrect = visualRect(indexClicked);
            int itemIndentation =
                vrect.x() - visualRect(rootIndex()).x();

            QRect rect = QRect(header()->sectionViewportPosition(0)
                        + itemIndentation - indentation(),
                        vrect.y(), indentation(), vrect.height());

            if(rect.contains(event->pos()) && model()->
                                hasChildren(indexClicked)) {
                qDebug() << "plus clicked"; // Вместо обработки события
                QTreeWidget::mousePressEvent(event);
                return;
            } else {
                QTreeWidget::mousePressEvent(event);
            }
        }
    }
```

```
private:
    QTreeWidgetItem *item1;
    QTreeWidgetItem *item2;
    QTreeWidgetItem *item3;
    QTreeWidgetItem *item4;
};

int main(int argc, char **argv) {
    QApplication app(argc, argv);
    MyTreeWidget tree;
    tree.show();
    return app.exec();
}
```

Чтобы в системе Windows увидеть консольное окно с сообщениями отладки, выводимыми функцией `qDebug()`, нужно в pro-файл перед выполнением команды `qmake` добавить строку

```
win32: CONFIG += console
```

ГЛАВА 15



Технология "Модель — представление"

> Я полагаю, что тот, у кого в голове сложилось о чем-либо живое и ясное представление, сумеет передать его на любом, хоть на тарабарском наречии.
>
> *Мишель Монтень*

В библиотеке Qt разработаны классы для представления массивов данных: список `QListView` (одномерный массив), таблица `QTableView` (двумерный массив) и дерево `QTreeView` (иерархическая таблица). Обычно принято разделять саму информацию и ее представление на экране, поэтому для хранения и преобразования данных предназначены классы `QAbstractItemModel`, `QAbstractListModel`, `QStringListModel`, `QAbstractProxyModel`, `QAbstractTableModel`, `QDirModel`, `QProxyModel` и `QStandardItemModel`.

Если разработчик не хочет разделять *модель* (Model) и ее *представление* (View, Вид), то он может использовать уже знакомые нам по предыдущей главе классы-виджеты `QListWidget`, `QTableWidget` и `QTreeWidget`, которые сами хранят свои данные, не обращаясь к классам-моделям.

15.1. Табличная модель и ее представления

> После смерти нас встречают два вопроса. Злой ангел спрашивает: "Как он жил?", милосердный Создатель спрашивает: "Как он умер?" Творец берет последнюю слезу умирающего и погашает ею все грехи, записанные на его жизненной таблице.
>
> *Сафир Мориц-Готлиб*

ЗАДАЧА. Разработать табличную модель, которая хранит данные в текстовом файле. Вывести на экран редактируемые представления этой модели в виде списка, таблицы и дерева (рис. 15.1).

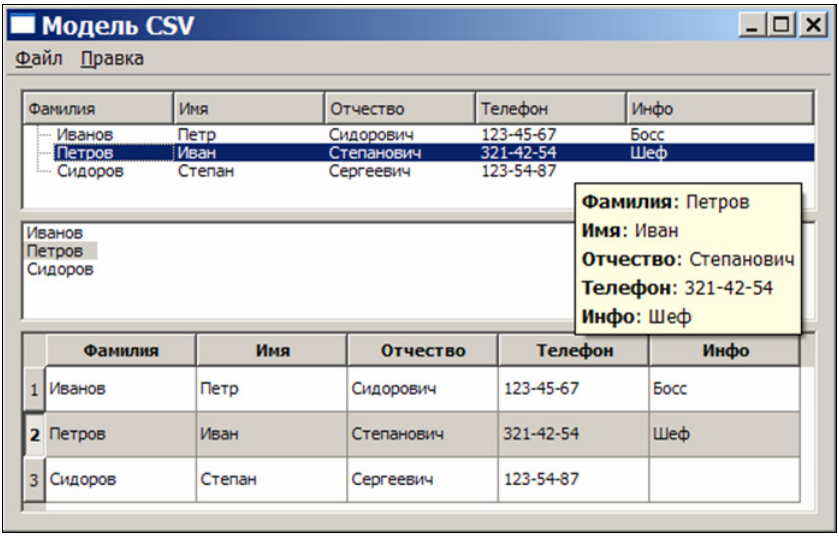


Рис. 15.1. Три представления одной модели

Решение. Предположим, что данные записаны в текстовом файле в следующем виде:

```
"Фамилия","Имя","Отчество","Телефон","Инфо"
"Иванов","Петр","Сидорович","123-45-67","Босс"
"Петров","Иван","Степанович","321-42-54","Шеф"
"Сидоров","Степан","Сергеевич","123-54-87",""
```

В первой строке перечислены заголовки столбцов, в остальных строках хранятся сами данные. Текстовое содержимое каждой ячейки таблицы заключено в кавычки. Разделителем значений служит запятая. Такой формат хранения табличных данных сокращенно называется *CSV* (*Comma-Separated Values* — значения, разделенные запятыми).

Для своей модели возьмем за основу класс `QAbstractTableModel`, как раз предназначенный для работы с данными, представленными в виде таблицы (листинги 15.1–15.5). Нам необходимо реализовать следующие виртуальные методы класса `QAbstractTableModel`:

- `rowCount()` — данный метод возвращает число строк в таблице;
- `columnCount()` — число столбцов таблицы;
- `headerData()` — заголовок столбца или его форматирование;
- `data()` — содержимое элемента (ячейки таблицы) или его форматирование (цвет, шрифт, выравнивание по вертикали и горизонтали, имеется ли элемент `QCheckBox`);

- `flags()` — параметры элемента (можно ли его редактировать, включать в выделение, перетаскивать);
- `hasChildren()` — имеет ли данный элемент дочерние элементы (для деревьев);
- `setData()` — метод, который необходим только в том случае, если пользователю разрешено редактировать элемент;
- `removeRows()` — метод необходимо реализовать, если пользователю разрешено удалять строки;
- `insertRows()` — метод необходимо реализовать, если пользователю разрешено добавлять строки.

Листинг 15.1. Файл chap15/csvmodel/csvmodel.h

```
#ifndef _CSVMODEL_H_
#define _CSVMODEL_H_

#include <QtGui>

class CsvModel : public QAbstractTableModel {
  Q_OBJECT
public:
  CsvModel(QObject *parent=0);
  ~CsvModel();
  int rowCount(const QModelIndex& parent=QModelIndex()) const;
  int columnCount(const QModelIndex& parent=QModelIndex()) const;
  QVariant headerData(int section,
                      Qt::Orientation orientation,
                      int role=Qt::DisplayRole) const;
  Qt::ItemFlags flags(const QModelIndex &index) const;
  QVariant data(const QModelIndex& index,
                int role=Qt::DisplayRole) const;
  bool hasChildren(const QModelIndex& parent=QModelIndex()) const;
  bool setData(const QModelIndex& index,
               const QVariant& value,
               int role=Qt::EditRole);
  bool insertRows(int row, int count,
                  const QModelIndex& parent=QModelIndex());
  bool removeRows(int row, int count,
                  const QModelIndex& parent=QModelIndex());
  bool loadData(const QString& fileName);
  bool saveData(const QString& fileName);
```

```
29 private:
30   QList<QStringList> *m_records;
31 };
32
33 #endif // _CSVMODEL_H_
```

Листинг 15.2. Файл chap15/csvmodel/csvmodel.cpp

```
34 #include <QtGui>
35 #include "csvmodel.h"
36
37 QStringList splitCSVLine(const QString& line) {
38    bool inItem = false;
39    QStringList items;
40    QString item;
41    for (int pos = 0; pos < line.length(); pos++){
42       QChar c = line.at(pos);
43       if ( c == '\"') {
44          if (inItem) {
45              items.append(item);
46              item = "";
47          }
48          inItem = !inItem;
49       }else
50          if (inItem)
51              item += c;
52    }
53    return items;
54 }
55
56 //----------------------------------------
57 CsvModel::CsvModel(QObject *parent)
58         : QAbstractTableModel(parent) {
59    m_records = new QList<QStringList>;
60 }
61
62 CsvModel::~CsvModel(){
63    m_records->clear();
64    delete m_records;
65 }
66
67 int CsvModel::rowCount(const QModelIndex& parent) const {
68   if ( parent.isValid() || (m_records->count() == 0) )
69      return 0;
```

```
  else
     return m_records->count() - 1;
}

int CsvModel::columnCount(const QModelIndex& /* parent */ ) const{
   if (m_records->count())
      return m_records->at(0).count();
   else
      return 0;
}

QVariant CsvModel::headerData( int section,
                               Qt::Orientation orientation,
                               int role) const {
   if (!m_records->count())
      return QVariant();

   if (orientation == Qt::Horizontal) {
      if (role == Qt::DisplayRole)
         return m_records->at(0).at(section);
   }
   else if (orientation == Qt::Vertical) {
      if (role == Qt::DisplayRole)
         return section+1;
   }

   return QAbstractTableModel::headerData(
                     section, orientation, role);
}

Qt::ItemFlags CsvModel::flags(const QModelIndex &index) const{
   if (!index.isValid() || !m_records->count())
      return 0;
   return QAbstractTableModel::flags(index) | Qt::ItemIsEditable;
}

QVariant CsvModel::data( const QModelIndex &index,
                         int role) const {
   if (!index.isValid() || !m_records->count())
       return QVariant(); // пусто

   if (role == Qt::SizeHintRole)
       return QSize(30, 12); // оптимальный размер

```

```
114     QStringList record = m_records->at(index.row()+1);
115
116     if (role == Qt::DisplayRole || role == Qt::EditRole)
117         return record.at(index.column());
118
119     if (role == Qt::ToolTipRole) {
120         QString tip, key, value;
121         tip = "<table>";
122         int maxLines = record.count();
123         for (int i = 0; i < maxLines; i++) {
124             key = headerData(
125                 i, Qt::Horizontal, Qt::DisplayRole).toString();
126             value = record.at(i);
127             if (!value.isEmpty())
128                 tip += QString("<tr><td><b>%1</b>: %2</td></tr>")
129                        .arg(key, value);
130         }
131         tip += "</table>";
132         return tip;
133     }
134     return QVariant(); // пусто
135 }
136
137 bool CsvModel::hasChildren(const QModelIndex& parent ) const {
138     return !parent.isValid();
139 }
140
141 bool CsvModel::setData(
142             const QModelIndex & index,
143             const QVariant& value,
144             int role) {
145
146     if (index.isValid() &&
147       (role == Qt::EditRole || role == Qt::DisplayRole)){
148          (*m_records)[index.row()+1][index.column()] =
                                value.toString();
149        emit dataChanged(index, index);
150        return true;
151     }
152     return false;
153 }
154
155 bool CsvModel::insertRows( int row, int count,
156                            const QModelIndex& parent) {
```

```
157     Q_UNUSED(parent);
158     QStringList emptyRecord;
159     for (int i=0; i<columnCount(QModelIndex()); i++)
160         emptyRecord.append(QString());
161     beginInsertRows(QModelIndex(), row, row+count-1);
162     for (int i=0; i<count; i++)
163        m_records->insert(row+1, emptyRecord);
164
165     endInsertRows();
166     return true;
167  }
168
169  bool CsvModel::removeRows( int row, int count,
170                             const QModelIndex& parent){
171     Q_UNUSED(parent);
172     if (row + count > m_records->count() — 1)
173        return false;
174
175     beginRemoveRows(QModelIndex(), row, row+count-1);
176     for(int i=0; i<count; i++)
177        m_records->removeAt(row+1);
178
179     endRemoveRows();
180     return true;
181  }
182
183  bool CsvModel::saveData(const QString& fileName) {
184     QFile file(fileName);
185     if ( !file.open(QIODevice::WriteOnly |
186                     QIODevice::Text |
187                     QIODevice::Truncate) )
188        return false;
189
190     QTextStream out(&file);
191     foreach(QStringList record, *m_records){
192         QString s = "\"";
193         s += record.join("\",\"");
194         s += "\"\n";
195         out << s;
196     }
197     file.close();
198     return out.status() == QDataStream::Ok;
199  }
```

```
200
201 bool CsvModel::loadData(const QString& fileName) {
202    QFile file(fileName);
203    if ( !file.open(QIODevice::ReadOnly|QIODevice::Text) )
204        return false;
205
206    m_records->clear();
207    QTextStream in(&file);
208    in.setCodec("CP1251");
209    while (!in.atEnd()) {
210       QString line = in.readLine();
211       m_records->append(splitCSVLine(line));  // (37-53)
212    }
213    file.close();
214    reset();
215    return in.status() == QDataStream::Ok;
216 }
```

Листинг 15.3. Файл chap15/csvmodel/mainwin.h

```
217 #ifndef _MAINWIN_H_
218 #define _MAINWIN_H_
219
220 #include <QtGui>
221 #include "csvmodel.h"
222
223 class MainWin : public QMainWindow {
224    Q_OBJECT
225 public:
226    MainWin();
227 public slots:
228    bool loadData();
229    bool saveData();
230    void removeRow();
231    void insertRowBefore();
232    void insertRowAfter();
233 private:
234    CsvModel *m_model;
235    QListView *m_listView;
236    QTableView *m_tableView;
237    QTreeView *m_treeView;
238
```

```
239     QAction *m_loadDataAct;
240     QAction *m_saveDataAct;
241     QAction *m_insertRowBeforeAct;
242     QAction *m_insertRowAfterAct;
243     QAction *m_removeRowAct;
244     QAction *m_exitAct;
245
246     QMenu *m_fileMenu;
247     QMenu *m_editMenu;
248 };
249
250 #endif // _MAINWIN_H_
```

Листинг 15.4. Файл chap15/csvmodel/mainwin.cpp

```
251 #include <QtGui>
252 #include "mainwin.h"
253 #include "csvmodel.h"
254
255 MainWin::MainWin()
256         : QMainWindow() {
257     QWidget *widget = new QWidget(this);
258     m_treeView = new QTreeView(widget);
259     m_listView = new QListView(widget);
260     m_tableView = new QTableView(widget);
261
262     m_treeView->setSelectionMode(
            QAbstractItemView::SingleSelection);
263     m_listView->setSelectionMode(
            QAbstractItemView::SingleSelection);
264     m_tableView->setSelectionMode(
            QAbstractItemView::SingleSelection);
265
266     m_treeView->setSelectionBehavior(
            QAbstractItemView::SelectRows);
267     m_listView->setSelectionBehavior(
            QAbstractItemView::SelectRows);
268     m_tableView->setSelectionBehavior(
            QAbstractItemView::SelectRows);
270
271     m_model = new CsvModel(this);
272     m_treeView->setModel(m_model);
273     m_listView->setModel(m_model);
274     m_tableView->setModel(m_model);
```

```
275
276     m_listView->setSelectionModel(m_treeView->selectionModel());
277     m_tableView->setSelectionModel(m_treeView->selectionModel());
278
279     QVBoxLayout *layout = new QVBoxLayout(widget);
280     layout->addWidget(m_treeView, 2);
281     layout->addWidget(m_listView, 1);
282     layout->addWidget(m_tableView, 3);
283     widget->setLayout(layout);
284     setCentralWidget(widget);
285     setWindowTitle(tr("Модель CSV"));
286
287     m_loadDataAct = new QAction(tr("&Открыть"), this);
288     connect(m_loadDataAct, SIGNAL(triggered()),
289             this, SLOT(loadData()));
290
291     m_saveDataAct = new QAction(tr("&Сохранить"), this);
292     connect(m_saveDataAct, SIGNAL(triggered()),
293             this, SLOT(saveData()));
294
295     m_insertRowBeforeAct = new QAction(tr("Вставить пере&д"),
                                        this);
296     connect(m_insertRowBeforeAct, SIGNAL(triggered()),
297             this, SLOT(insertRowBefore()));
298     m_insertRowAfterAct = new QAction(tr("Вставить посл&е"),
                                      this);
299     connect(m_insertRowAfterAct, SIGNAL(triggered()),
300             this, SLOT(insertRowAfter()));
301
302     m_removeRowAct = new QAction(tr("&Удалить запись"), this);
303     connect(m_removeRowAct, SIGNAL(triggered()),
304             this, SLOT(removeRow()));
305
306     m_exitAct = new QAction(tr("В&ыход"), this);
307     connect(m_exitAct, SIGNAL(triggered()),
308             this, SLOT(close()));
309
310     m_fileMenu = menuBar()->addMenu(tr("&Файл"));
311     m_fileMenu->addAction(m_loadDataAct);
312     m_fileMenu->addAction(m_saveDataAct);
313     m_fileMenu->addSeparator();
314     m_fileMenu->addAction(m_exitAct);
315
```

```
316     m_editMenu = menuBar()->addMenu(tr("&Правка"));
317     m_editMenu->addAction(m_insertRowBeforeAct);
318     m_editMenu->addAction(m_insertRowAfterAct);
319     m_editMenu->addAction(m_removeRowAct);
320  }
321
322  bool MainWin::loadData() {
323     return m_model->loadData("data.csv");
324  }
325
326  bool MainWin::saveData() {
327     return m_model->saveData("data.csv");
328  }
329
330  void MainWin::insertRowBefore() {
331     m_model->insertRows( m_tableView->selectionModel()->
                                    currentIndex().row(), 1);
332  }
333
334  void MainWin::insertRowAfter() {
335      m_model->insertRows(m_tableView->selectionModel()->
                                  currentIndex().row()+1, 1);
336  }
337
338  void MainWin::removeRow() {
339      m_model->removeRows(m_tableView->selectionModel()->
                                    currentIndex().row(), 1);
340  }
```

Листинг 15.5. Файл chap15/csvmodel/main.cpp

```
341  #include <QtGui>
342  #include "mainwin.h"
343  #include "csvmodel.h"
344
345  int main( int argc, char* argv[] ) {
346     QApplication app( argc, argv );
347     QTextCodec *codec = QTextCodec::codecForName("CP1251");
348     QTextCodec::setCodecForTr(codec);
349
350     MainWin win;
351     win.show();
352     return app.exec();
353  }
```

Пояснения к программе:

- (37–53) — функция для преобразования строки значений, разделенных запятыми и заключенных в кавычки, в список строк;
- (67–72) — если список записей `m_records` пуст или задан правильный родительский элемент, то число строк равно нулю (ведь у нас не иерархический список); иначе возвращаем число элементов списка за вычетом единицы (т. к. первая строка файла хранит заголовки столбцов, а не сами данные);
- (74–79) — число столбцов таблицы — это число элементов в первой строке файла;
- (81–83) — при значении параметра `role=Qt::DisplayRole` метод должен возвращать заголовок колонки (если `orientation=Qt::Horizontal`) или столбца (если `orientation=Qt::Vertical`) с номером `section` (считая с нуля);
- (84, 85) — если данные пока не загружены или файл оказался пустым, то вместо заголовка возвращаем пустое значение;
- (96, 97) — для всех остальных комбинаций входных параметров возвращаем значение, определенное в базовом классе;
- (100–104) — если индекс элемента верный, то устанавливаем флаг "редактируемый";
- (106–135) — данные, возвращаемые для каждого элемента таблицы, в зависимости от требуемой роли: `Qt::SizeHintRole` (оптимальный размер элемента), `Qt::DisplayRole` (отображаемая строка), `Qt::EditRole` (строка, отображаемая при редактировании элемента), `Qt::ToolTipRole` (текст всплывающей подсказки);
- (137–139) — если индекс элемента верный, то в нашем случае он не имеет дочерних элементов (у нас не иерархический список);
- (141–153) — этот метод устанавливает новые значения для только что отредактированного элемента;
- (155–167) — вставка новых строк;
- (169–181) — удаление строк;
- (183–199) — сохранение данных в файл;
- (201–216) — загрузка данных из файла;
- (223–248) — класс главного окна приложения.

15.2. Выравнивание элементов списка

Поразительна мудрость природы, которая при таком бесконечном разнообразии сумела всех уравнять!
Эразм Роттердамский

ЗАДАЧА. Изменить выравнивание текста или пиктограммы в элементах вида или виджета (рис. 15.2).

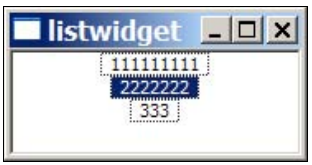


Рис. 15.2. Выравнивание элементов списка по центру

Решение. Необходимо на основе `Q(List/Table/Tree)View/Widget` создать свой класс, в котором переопределить метод `viewOptions()` (листинги 15.6 и 15.7).

Листинг 15.6. Файл chap15/listwidget/listwidget.h

```
#include <QtGui>

class ListWidget : public QListWidget {
public:
    ListWidget();
    QStyleOptionViewItem viewOptions() const;
};
```

Листинг 15.7. Файл chap15/listwidget/listwidget.cpp

```
#include "listwidget.h"

ListWidget::ListWidget() {
    QListWidgetItem *item1 = new QListWidgetItem("111111111");
    QListWidgetItem *item2 = new QListWidgetItem("2222222");
    QListWidgetItem *item3 = new QListWidgetItem("333");
    addItem(item1);
    addItem(item2);
    addItem(item3);
}

QStyleOptionViewItem ListWidget::viewOptions() const {
    QStyleOptionViewItem item;
```

```
    item.init(this);
    item.displayAlignment = Qt::AlignHCenter;
    return item;
}

//--------------------------------
int main(int argc, char** argv){
    QApplication app(argc, argv);
    ListWidget listWidget;
    listWidget.show();
    return app.exec();
}
```

15.3. Галочки для элементов таблицы

ЗАДАЧА. Изменить положение галочек `QCheckBox` внутри элементов представления `QTableView` или виджета `QTableWidget` (рис. 15.3).

Решение. Для тонкой настройки внешнего вида и поведения каждого элемента представления (в обычном режиме и/или во время его редактирования) используются так называемые *делегаты*. Свои классы-делегаты удобно разрабатывать на основе классов `QItemDelegate` или `QAbstractItemDelegate` (листинги 15.8 и 15.9).

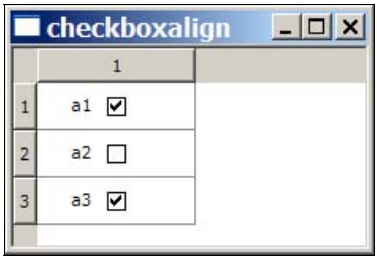


Рис. 15.3. Галочки не слева, как обычно, а справа от текста

Листинг 15.8. Файл chap15/checkboxalign/checkboxalign.h

```
#include <QtGui>

class ItemDelegate : public QItemDelegate { // Делегат
public:
    ItemDelegate(QObject *parent = 0);
    virtual void drawCheck(  // рисовать элемент QCheckBox
```

```
        QPainter *painter,
        const QStyleOptionViewItem &option,
        const QRect &,
        Qt::CheckState state) const;
    virtual bool editorEvent( // рисовать редактор для элемента
        QEvent *event,
        QAbstractItemModel *model,
        const QStyleOptionViewItem &option,
        const QModelIndex &index);
    virtual void drawFocus(  // отобразить текущий элемент
        QPainter *painter,
        const QStyleOptionViewItem &option,
        const QRect &rect) const;
};

//----------------------------------------
class Table : public QTableWidget { // Представление
    Q_OBJECT
public:
    Table(QWidget *parent = 0);
};
```

Листинг 15.9. Файл chap15/checkboxalign/checkboxalign.cpp

```
#include "checkboxalign.h"

ItemDelegate::ItemDelegate(QObject *parent)
            : QItemDelegate(parent) { }

void ItemDelegate::drawCheck( QPainter *painter,
                              const QStyleOptionViewItem &option,
                              const QRect &,
                              Qt::CheckState state) const {

    const int textMargin = QApplication::style()->pixelMetric(
        QStyle::PM_FocusFrameHMargin) + 1;

    QRect checkRect = QStyle::alignedRect(
        option.direction, Qt::AlignCenter,
        check(option, option.rect, Qt::Checked).size(),
        QRect(option.rect.x() + textMargin, option.rect.y(),
        option.rect.width() — (textMargin * 2), option.rect.height()));
```

```
    QItemDelegate::drawCheck(painter, option, checkRect, state);
}

bool ItemDelegate::editorEvent( QEvent *event,
                                QAbstractItemModel *model,
                                const QStyleOptionViewItem &option,
                                const QModelIndex &index) {
    Qt::ItemFlags flags = model->flags(index);
    if ( !(flags & Qt::ItemIsUserCheckable) ||
         !(flags & Qt::ItemIsEnabled))
        return false;

    QVariant value = index.data(Qt::CheckStateRole);
    if (!value.isValid())
        return false;

    if (event->type() == QEvent::MouseButtonRelease) {
        const int textMargin =
            QApplication::style()->pixelMetric(
                QStyle::PM_FocusFrameHMargin) + 1;
        QRect checkRect = QStyle::alignedRect(
            option.direction, Qt::AlignCenter,
            check(option, option.rect, Qt::Checked).size(),
            QRect(option.rect.x() + textMargin, option.rect.y(),
            option.rect.width() - (2 * textMargin),
            option.rect.height()));
        if (!checkRect.contains(
                static_cast<QMouseEvent*>(event)->pos()))
            return false;

    } else if (event->type() == QEvent::KeyPress) {
        if ( static_cast<QKeyEvent*>(event)->key() != Qt::Key_Space &&
             static_cast<QKeyEvent*>(event)->key() != Qt::Key_Select)
            return false;
    } else {
        return false;
    }
    Qt::CheckState state = (static_cast<Qt::CheckState>(value.toInt())
           == Qt::Checked ? Qt::Unchecked : Qt::Checked);

    return model->setData(index, state, Qt::CheckStateRole);
}
```

```
void ItemDelegate::drawFocus( QPainter *painter,
                              const QStyleOptionViewItem &option,
                              const QRect & /* rect */) const {
    QItemDelegate::drawFocus(painter, option, option.rect);
}

//--------------------------------
static int ROWS = 3;
static int COLS = 1;

Table::Table(QWidget *parent)
     : QTableWidget(ROWS, COLS, parent) {

    setItemDelegate(new ItemDelegate(this));
    QTableWidgetItem *item = 0;
    for (int i=0; i<rowCount(); ++i) {
        for (int j=0; j<columnCount(); ++j) {
            setItem( i, j, item = new QTableWidgetItem(
                tr("a%1").arg(i+1)) );
            item->setTextAlignment(Qt::AlignLeft | Qt::AlignVCenter);
            item->setFlags(Qt::ItemIsEnabled|Qt::ItemIsUserCheckable);
            item->setCheckState((i+j) % 2
                 == 0 ? Qt::Checked : Qt::Unchecked);
        }
    }
}
//----------------------------------------
int main(int argc, char **argv){
    QApplication app(argc, argv);
    Table table;
    table.show();
    return app.exec();
}
```

15.4. Дерево каталогов и файлов

Если мне скажут, что завтра наступит конец света, то еще сегодня я посадил бы дерево.
Мартин Лютер

ЗАДАЧА. Требуется отобразить иерархическую структуру каталогов и файлов на диске (рис. 15.4).

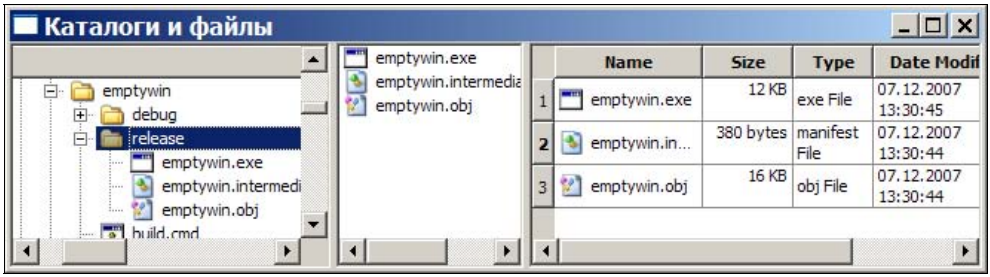


Рис. 15.4. Дерево каталогов, список и таблица файлов для текущего каталога

Решение. Для получения сведений о каталогах и файлах в библиотеке Qt имеется готовая модель `QDirModel`, поэтому нам остается только разобраться с представлениями и методами-слотами, вызываемыми при смене текущего каталога (листинг 15.10).

Листинг 15.10. Файл chap15/dirview/dirview.cpp

```
// Просмотр дерева каталогов на дисках

#include <QtGui>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);

    QDirModel model;  // Модель дерева каталогов
    model.setSorting( // Сортировка
        QDir::DirsFirst | QDir::IgnoreCase | QDir::Name);

    QSplitter splitter; // Разделитель между виджетами
    splitter.setOrientation(Qt::Horizontal);

    QTreeView tree;  // Вид дерева
    tree.setModel(&model);
    //tree.setRootIndex(model->index(QDir::currentPath()));
    tree.header()->setSortIndicator(0, Qt::AscendingOrder);
    tree.header()->setSortIndicatorShown(true);
    //tree.header()->setClickable(true);
    splitter.addWidget(&tree);

    QListView list; // Вид списка
    list.setModel(&model);
```

```
    list.setRootIndex(model.index(QDir::currentPath()));
    list.setViewMode(QListView::ListMode);
    splitter.addWidget(&list);

    QTableView table; // Вид таблицы
    table.setModel(&model);
    table.setRootIndex(model.index(QDir::currentPath()));
    table.resizeRowsToContents();
    splitter.addWidget(&table);

    list.setSelectionModel(tree.selectionModel());
    table.setSelectionModel(tree.selectionModel());

    QObject::connect(
        tree.header(),
        SIGNAL(activated(QModelIndex)),
        &model,
        SLOT(setRootIndex(QModelIndex)));
    QObject::connect(
        &tree,
        SIGNAL(activated(QModelIndex)),
        &list,
        SLOT(setRootIndex(QModelIndex)));
    QObject::connect(
        &tree,
        SIGNAL(activated(QModelIndex)),
        &table,
        SLOT(setRootIndex(QModelIndex)));
    QObject::connect(
        &tree,
        SIGNAL(activated(QModelIndex)),
        &table,
        SLOT(resizeRowsToContents()));

    splitter.setWindowTitle(QObject::tr("Каталоги и файлы"));

    QModelIndex index = model.index(QDir::currentPath());
    tree.expand(index);
    tree.scrollTo(index, QAbstractItemView::PositionAtTop);
    tree.resizeColumnToContents(0);
    splitter.show();

    return app.exec();
}
```

15.5. Иерархическая модель

Не терпеть нужды и не иметь излишка, не командовать другими и не быть в подчинении — вот моя цель.

Франческо Петрарка

ЗАДАЧА. Реализовать иерархическую структуру из *разд. 14.5* с помощью технологии "Модель — представление" (рис. 15.5).

Решение данной задачи показано в листингах 15.11 и 15.12.

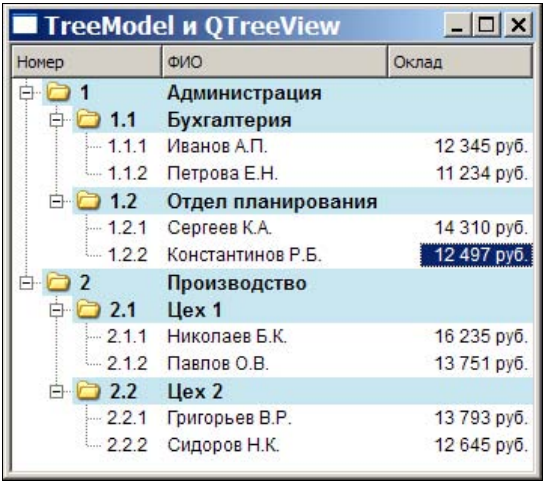


Рис. 15.5. Модель и представление дерева

Листинг 15.11. Файл chap15/treemodel/treemodel.h

```
#include <QtGui>
#include <QFileIconProvider>

class Node {
public:
  Node(Node *parentNode = 0,
     const QString& s1 = "",
     const QString& s2 = "",
     const QString& s3 = "");
  ~Node();
  QString str1, str2, str3;
  Node *parent;
  QList<Node*> children;
};
```

```
15
16  Node* InitTree();
17
18  class TreeModel : public QAbstractItemModel {
19      Q_OBJECT
20  public:
21     TreeModel(QObject *parent = 0);
22     ~TreeModel();
23     QModelIndex index(
24         int row, int column,
25         const QModelIndex& parent = QModelIndex()) const;
26     bool hasChildren(const QModelIndex& parent=QModelIndex()) const;
27     QModelIndex parent(const QModelIndex& index) const;
28     int rowCount(const QModelIndex& index = QModelIndex()) const;
29     int columnCount(const QModelIndex& index = QModelIndex()) const;
30     QVariant data( const QModelIndex& index,
                      int role = Qt::DisplayRole) const;
31     void setRootNode(Node *node);
32     Qt::ItemFlags flags(const QModelIndex& index) const;
33     QVariant headerData(
           int section, Qt::Orientation orientation, int role) const;
34     bool setData(const QModelIndex& index,
                    const QVariant& value, int role = Qt::EditRole);
35  private:
36     Node *nodeFromIndex(const QModelIndex& index) const;
37     Node *rootNode;
38     QFileIconProvider ip;
39  };
```

Пояснения к программе:

- (4–14) — класс — элемент дерева;
- (12) — указатель на родительский элемент;
- (13) — список дочерних элементов;
- (16) — функция для инициализации дерева;
- (18–38) — класс — модель дерева.

Листинг 15.12. Файл chap15/treemodel/treemodel.cpp

```
#include <QtGui>

#include "treemodel.h"
```

```
Node::Node(Node *parentNode, const QString& s1,
           const QString& s2, const QString& s3 )
    : str1(s1), str2(s2), str3(s3),
      parent(parentNode) { }

Node::~Node(){
    qDeleteAll(children); // Удалить все дочерние узлы
}

//-----------------------------------------------
TreeModel::TreeModel(QObject *parent)
         : QAbstractItemModel(parent){
    rootNode = 0;
};

TreeModel::~TreeModel(){
    delete rootNode;
}

// Получить индекс элемента по номеру строки, столбца и индексу родителя
QModelIndex TreeModel::index(int row, int column,
                             const QModelIndex& parent) const {
    if (rootNode) { // Если это корневой узел
        Node *parentNode = nodeFromIndex(parent);
        if (parentNode) // Если имеется родительский узел
            return createIndex(row, column,
                               parentNode->children[row]);
    }
    return QModelIndex(); // нет такого
};

// Есть ли у данного элемента дочерние элементы?
bool TreeModel::hasChildren(const QModelIndex& parent) const {
    Node *parentNode = nodeFromIndex(parent);
    if (!parentNode) return false;                    // нет
    else return (parentNode->children.count() > 0);   // есть
}

// Получить родительский узел
QModelIndex TreeModel::parent(const QModelIndex& index) const {
    Node *node = nodeFromIndex(index);
    if (!node)
        return QModelIndex();  // несуществующий элемент
```

```
    Node *parentNode = node->parent;
    if (!parentNode)
        return QModelIndex();  // нет родительского узла

    Node *grandparentNode = parentNode->parent;
    if (!grandparentNode)
        return QModelIndex(); // родительский узел не имеет родителя

    int row = grandparentNode->children.indexOf(parentNode);
    // Нашли родителя:
    return createIndex(row, index.column(), parentNode);
}

// Количество строк
int TreeModel::rowCount(const QModelIndex& index) const {
    Node *node = nodeFromIndex(index);
    if ( node )
        return node->children.count();
    else
        return 0;
}

// Количество столбцов
int TreeModel::columnCount(const QModelIndex& /* index */ ) const {
    return 3;
}

// Данные:
QVariant TreeModel::data(const QModelIndex& index, int role) const {
    int col = index.column();
    Node *node = nodeFromIndex(index);
    switch (role) {  // В зависимости от требуемой роли
        case Qt::DisplayRole: // Данные для отображения
        case Qt::EditRole: {   // Данные для редактирования
            if (!node) return QVariant();
            else if (col == 0) return node->str1;
            else if (col == 1) return node->str2;
            else if (col == 2) return node->str3;
            else return QVariant();
        }
        case Qt::TextColorRole: // Цвет текста
            return qVariantFromValue(QColor(0, 0, 0));
```

```
        case Qt::TextAlignmentRole: // Выравнивание
            if (col == 0)
                return int(Qt::AlignLeft | Qt::AlignVCenter);
            else if (col == 1)
                return int(Qt::AlignLeft | Qt::AlignVCenter);
            else if (col == 2)
                return int(Qt::AlignRight | Qt::AlignVCenter);
            else return QVariant();

        case Qt::FontRole: { // Шрифт
            QFont font = QFont("Helvetica", 9);
            if (node->children.count() > 0)
                return qVariantFromValue(
                    QFont("Helvetica", 10, QFont::Bold));
            else
                return qVariantFromValue(QFont("Helvetica", 9));
        }
        case Qt::BackgroundColorRole:  { // Цвет фона
            if (node->children.count() > 0)
                return qVariantFromValue(QColor(200, 230, 240));
            else
                return qVariantFromValue(QColor(255, 255, 255));
        }
        case Qt::CheckStateRole:  // Галочка
            return QVariant(); // В модели нет столбцов с пометками

        case Qt::SizeHintRole:  // Размер ячейки по умолчанию
            return QSize(150, 18);

        case Qt::DecorationRole: { // Значок
            if (col == 0 && node->children.count() > 0)
                return ip.icon(QFileIconProvider::Folder);
            else
                return QVariant(); // Нет значка
        }
    }
    return QVariant(); // Для всех остальных случаев данных нет
}

// Установить корневой узел дерева
void TreeModel::setRootNode(Node *node){
    delete rootNode;   // Удалить прежний корневой узел
    rootNode = node;
```

```
    reset();            // Обновить модель
}

// Получить сам элемент по его индексу в модели
Node* TreeModel::nodeFromIndex(const QModelIndex& index) const {
    if (index.isValid()) {  // Если индекс верный
        return static_cast<Node*>(index.internalPointer());
    }else{
        return rootNode;    // Иначе вернуть корневой узел
    }
}

// Флаги дополнительных свойств
Qt::ItemFlags TreeModel::flags(const QModelIndex& index) const {
    if ( index.isValid() )
        return Qt::ItemIsEnabled | Qt::ItemIsEditable |
               Qt::ItemIsSelectable;
    else
        return 0;
}

// Заголовок
QVariant TreeModel::headerData(int section,
                               Qt::Orientation orientation,
                               int role) const {
    if (orientation == Qt::Horizontal &&
            role == Qt::DisplayRole) {
        if (section == 0)
            return tr("Номер");
        else if (section == 1)
            return tr("ФИО");
        else if (section == 2)
            return tr("Оклад");
    }
    return QVariant();
}

// Записать данные после редактирования
bool TreeModel::setData(
            const QModelIndex& index,
            const QVariant& value,
            int role) {
    if ( role != Qt::EditRole )
        return false;
```

```
    int col = index.column();
    Node *node = nodeFromIndex(index);
    if (col == 0)
        node->str1 = value.toString();
    else if (col == 1)
        node->str2 = value.toString();
    else if (col == 2)
        node->str3 = value.toString();

    return true;
}

//-----------------------------------------------------------
Node* InitTree(){  // Данные для дерева
    Node *root = new Node();

    Node *group1 = new Node(root, QObject::tr("1"),
        QObject::tr("Администрация"), "" );
    root->children.append(group1);

    Node *group11 = new Node(group1, QObject::tr("1.1"),
        QObject::tr("Бухгалтерия"), "" );
    group1->children.append(group11);

    // ...

    group11->children.append( new Node(
                    group11, QObject::tr("1.1.1"),
                    QObject::tr("Иванов А.П."),
                    QObject::tr("12 345 руб.") ) );
    // ...
    return root;
}
//----------------------------------------
#include <QtGui>
#include "treemodel.h"

int main(int argc, char *argv[]) {

    QApplication app(argc, argv);
    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);
```

```
    TreeModel model;
    model.setRootNode(InitTree());

    QTreeView view;
    view.setSelectionMode(QAbstractItemView::SingleSelection);
    view.setSelectionBehavior(QAbstractItemView::SelectItems);
    view.setModel(&model); // Задали модель
    view.expandAll();      // Развернули все группы
    view.header()->setStretchLastSection(true);
    view.resizeColumnToContents(1);

    view.setWindowTitle(QObject::tr("TreeModel и QTreeView"));
    view.resize(350, 300);
    view.show();

    return app.exec();
}
```

Разумеется, для организации иерархической модели можно было использовать стандартный класс `QStandardItemModel`, но мы хотели показать принцип реализации пользовательской модели данных.

15.6. Дерево групп и таблица элементов

ЗАДАЧА. Требуется разработать модель и представление для работы с иерархической структурой подразделений фирмы (рис. 15.6).

Решение данной задачи показано в листингах 15.13 и 15.14.

Листинг 15.13. Файл chap15/treemodelview/treemodelview.h

```
#include <QtGui>
#include <QFileIconProvider>  // Для значков групп

enum NodeType {  // Тип узла:
    ELEMENT = 0, // элемент
    GROUP,       // группа.
};

class Node {
public:
    Node( Node *parentNode = 0, const QString& s1 = "",
          const QString& s2 = "", const QString& s3 = "",
```

```
            NodeType nt = ELEMENT );
    ~Node();

    QString str1, str2, str3;
    NodeType nodeType;
    Node *parent;
    QList<Node*> children;
};

// Модель для хранения данных
class TreeModel : public QAbstractItemModel {
    Q_OBJECT
public:
    TreeModel(QObject *parent = 0);
    ~TreeModel();
    QModelIndex index(
        int row, int column,
        const QModelIndex& parent = QModelIndex()) const;
    bool hasChildren(const QModelIndex& parent=QModelIndex()) const;
    QModelIndex parent(const QModelIndex& index) const;
    int rowCount(const QModelIndex& index=QModelIndex()) const;
    int columnCount(const QModelIndex& index=QModelIndex()) const;
    QVariant data(
        const QModelIndex& index,
        int role = Qt::DisplayRole) const;
    void setRootNode(Node *node);
    Qt::ItemFlags flags(const QModelIndex& index) const;
    QVariant headerData(
        int section, Qt::Orientation orientation, int role) const;
    bool setData(
        const QModelIndex& index,
        const QVariant& value, int role = Qt::EditRole);
    // Qt::DropActions supportedDropActions() const;
private:
    Node *nodeFromIndex(const QModelIndex& index) const;
    Node *rootNode;
    QFileIconProvider ip;
};

// Делегат для ячеек таблицы
class MyDelegate : public QItemDelegate {
    Q_OBJECT
```

```
public:
    MyDelegate(QObject *parent = 0);
    QWidget *createEditor(
                QWidget *parent,
                const QStyleOptionViewItem &option,
                const QModelIndex &index) const;
    void setEditorData(QWidget *editor,
                       const QModelIndex &index) const;
    void setModelData(QWidget *editor,
                      QAbstractItemModel *model,
                      const QModelIndex &index) const;
    void updateEditorGeometry(
            QWidget *editor,
            const QStyleOptionViewItem &option,
            const QModelIndex &index) const;
};

// Вид таблицы
class MyTableView : public QTableView {
    Q_OBJECT
public:
    MyTableView(QWidget *parent=0);
};

// Главное окно
class MyWin : public QWidget {
    Q_OBJECT
public:
    MyWin(QWidget *parent=0);
public slots:
    void groupChanged(const QModelIndex& index);
private:
    TreeModel *model;
    QTreeView *treeview;
    MyTableView *tableview;
    QSortFilterProxyModel *treeProxyModel;
    QSortFilterProxyModel *tableProxyModel;
};
```

Листинг 15.14. Файл chap15/treemodelview/treemodelview.cpp

```
#include <QtGui>

#include "treemodelview.h"
```

```
Node::Node(Node *parentNode, const QString& s1,
           const QString& s2, const QString& s3,
           NodeType nt )
    : str1(s1), str2(s2), str3(s3),
      nodeType(nt), parent(parentNode) {}

Node::~Node(){
    qDeleteAll(children);  // Удалить все дочерние элементы
}

//--------------------------------------
TreeModel::TreeModel(QObject *parent)
         : QAbstractItemModel(parent){
    rootNode = 0; // Корневой узел пока отсутствует
};

TreeModel::~TreeModel(){
    delete rootNode; // Удалить корневой узел
}

  // Методы index(), hasChildren(), parent(), rowCount(), columnCount()
  // setRootNode(), nodeFromIndex(),
  // headerData() и setData() — как в листинге 15.12

QVariant TreeModel::data(const QModelIndex& index, int role) const {
    int col = index.column();
    Node *node = nodeFromIndex(index);
    switch (role) { // В зависимости от требуемой роли
        case Qt::DisplayRole: // Данные для отображения
        case Qt::EditRole: {   // Данные для редактирования
            if (!node) return QVariant();
            else if (col == 0) return node->str1; // Столбец 1
            else if (col == 1) return node->str2; // Столбец 2
            else if (col == 2) return node->str3; // Столбец 3
            else if (col == 3) return node->nodeType; // Тип узла
            else if (col == 4) return node->parent;   // Родитель
            else return QVariant(); // Для других столбцов нет данных
        }
        case Qt::TextColorRole: // Цвет текста
            return qVariantFromValue(QColor(0, 0, 0));

        case Qt::TextAlignmentRole: // Выравнивание
            if (col == 0)
                return int(Qt::AlignLeft | Qt::AlignVCenter);
```

```
            // Аналогично для остальных столбцов
            else
                return QVariant();

        case Qt::FontRole: { // Шрифт
            // ...
        }
        case Qt::BackgroundColorRole:  { // Цвет фона
            // ...
        }
        case Qt::CheckStateRole:  // Галочка
            return QVariant();

        case Qt::SizeHintRole:  // Размер ячейки по умолчанию
            return QSize(150, 18);

        case Qt::DecorationRole: { // Значок
            // ...
        }
    }
    return QVariant();
}

Qt::ItemFlags TreeModel::flags(const QModelIndex& index) const {
    if ( index.isValid() )
        return Qt::ItemIsEnabled | Qt::ItemIsEditable |
               Qt::ItemIsSelectable;
               // | Qt::ItemIsDragEnabled | Qt::ItemIsDropEnabled;
    else
        return 0; //Qt::ItemIsDropEnabled;
}

// Если требуется реализовать "перетаскивание" элементов
// Qt::DropActions TreeModel::supportedDropActions() const {
//     return Qt::CopyAction | Qt::MoveAction;
// }

//--------------------------------------------------------
Node* InitTree(){ // Инициализация данных
    Node *root = new Node();

    Node *group1 = new Node(
        root, QObject::tr("1"),
```

```
        QObject::tr("Администрация"),
        "", GROUP);
    root->children.append(group1);

    Node *group11 = new Node(
        group1, QObject::tr("1.1"),
        QObject::tr("Бухгалтерия"),
        "", GROUP);
    group1->children.append(group11);

    // ...

    group11->children.append( new Node(
                    group11, QObject::tr("1.1.1"),
                    QObject::tr("Иванов А.П."),
                    QObject::tr("12 345 руб."), ELEMENT ) );
    // ...
    return root;
}

//-----------------------------------------
MyTableView::MyTableView(QWidget *parent)
           : QTableView(parent) { }

//-----------------------------------------
MyDelegate::MyDelegate(QObject *parent)
             : QItemDelegate(parent) { }

// Создать элемент редактирования ячейки таблицы:
QWidget *MyDelegate::createEditor(
            QWidget *parent,
            const QStyleOptionViewItem& /* option */,
            const QModelIndex& /* index */) const {
    QLineEdit *editor = new QLineEdit(parent);
    editor->installEventFilter(const_cast<MyDelegate*>(this));
    editor->setStyleSheet(QString(
        "QLineEdit{ background-color: %1 }")
           .arg(QColor(230,250,255).name()));
    return editor;
}

// Записать данные из модели в элемент редактирования:
void MyDelegate::setEditorData(
                QWidget *editor,
                const QModelIndex& index) const {
```

```
    QString t = index.model()->data(
            index, Qt::EditRole).toString();
    QLineEdit *le = static_cast<QLineEdit*>(editor);
    le->setText(t);
}

// Записать данные в модель после редактирования:
void MyDelegate::setModelData(
            QWidget *editor,
            QAbstractItemModel *model,
            const QModelIndex& index) const {
    QLineEdit *le = static_cast<QLineEdit*>(editor);
    QString t = le->text();
    model->setData(index, t);
}

// Задать размеры и положение элемента редактирования
void MyDelegate::updateEditorGeometry(
            QWidget *editor,
            const QStyleOptionViewItem &option,
            const QModelIndex& /* index */) const {
    editor->setGeometry(option.rect.adjusted(-1,-1,1,1));
}

//----------------------------------------
// Главное окно приложения
MyWin::MyWin(QWidget *parent)
     : QWidget(parent) {

    QVBoxLayout *lay = new QVBoxLayout(this);
    QSplitter *splitter = new QSplitter(this); // Разделитель

    model = new TreeModel(this);
    model->setRootNode(InitTree());

    // Прокси-модель дерева (для фильтрации):
    treeProxyModel = new QSortFilterProxyModel(this);
    treeProxyModel->setSourceModel(model);
    treeProxyModel->setFilterRole(Qt::DisplayRole);
    treeProxyModel->setFilterKeyColumn(3);
    treeProxyModel->setFilterFixedString("1"); // группы без элементов
```

```
// Прокси-модель таблицы:
tableProxyModel = new QSortFilterProxyModel(this);
tableProxyModel->setSourceModel(model);
//tableProxyModel->setFilterRole(Qt::DisplayRole);
//tableProxyModel->setFilterKeyColumn(4);

treeview = new QTreeView(this); // Вид для дерева
treeview->setSelectionMode(QAbstractItemView::SingleSelection);
treeview->setSelectionBehavior(QAbstractItemView::SelectRows);
treeview->setModel(treeProxyModel);
treeview->hideColumn(2);
treeview->hideColumn(3);
treeview->hideColumn(4);
treeview->expandAll();
treeview->header()->setStretchLastSection(true);
// treeview->setAcceptDrops(true);
// treeview->setDropIndicatorShown(true);

QColor color = QColor(200, 230, 240);
treeview->setStyleSheet(QString(
    "QTreeView{ background-color: %1 }").arg(color.name()));

tableview = new MyTableView(this);  // Вид для таблицы
tableview->setModel(tableProxyModel);
tableview->hideColumn(3);
tableview->hideColumn(4);
tableview->resizeRowsToContents();
//tableview->resizeColumnsToContents();
tableview->horizontalHeader()->setStretchLastSection(true);
// tableview->setSelectionMode(QAbstractItemView::SingleSelection);
// tableview->setDragEnabled(true); ///////
// tableview->setDropIndicatorShown(true);

MyDelegate *d = new MyDelegate(tableview); // Делегат
tableview->setItemDelegate(d);

splitter->addWidget(treeview);
splitter->addWidget(tableview);
splitter->setStretchFactor(0, 2);
splitter->setStretchFactor(1, 3);

// Смена текущей группы:
connect( treeview,
         SIGNAL(clicked(const QModelIndex &)),
```

```
            this,
            SLOT(groupChanged(const QModelIndex &)) );

    setWindowTitle(tr( // Заголовок окна
        "TreeModel, QTreeView и QTableView"));
    lay->addWidget(splitter);
    setLayout(lay);
}

// Обновить таблицу при смене группы в дереве
void MyWin::groupChanged(const QModelIndex& index){
    // Из прокси-модели дерева к исходной:
    QModelIndex i = treeProxyModel->mapToSource(index);

    // От исходной модели к прокси-модели таблицы:
    QModelIndex j = tableProxyModel->mapFromSource(i);

    // Установить текущий корневой узел для вида таблицы:
    tableview->setRootIndex(j);
}

//-------------------------------------
int main(int argc, char *argv[]) {

    QApplication app(argc, argv);

    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);

    MyWin w;
    w.show();

    return app.exec();
}
```

Здесь таблица `tableview`, предназначенная для отображения элементов иерархического справочника, и дерево групп `treeview` отделены друг от друга с помощью подвижного разделителя `splitter`. Чтобы в дереве и в таблице отображались не все элементы исходной модели `model`, мы используем прокси-модели `treeProxyModel` и `tableProxyModel`. При щелчке мышью по элементам дерева `treeview` (группам иерархического справочника) мы пересчитываем индекс активизированного элемента сначала из прокси-модели, установленной для дерева, к индексу исходной модели `model`, а его, в свою очередь, к индексу прокси-модели таблицы. Узнав, таким образом, индекс нужной

группы "с точки зрения" таблицы, мы устанавливаем его в качестве текущего для вида таблицы `tableview`. В результате в таблице отображаются только те элементы, которые принадлежат выделенной группе.

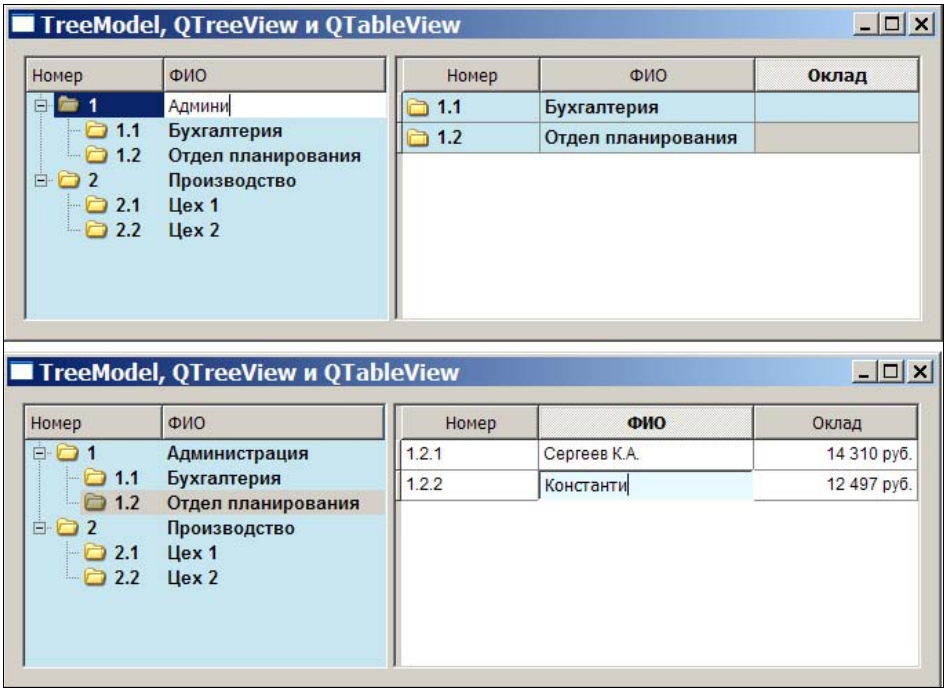


Рис. 15.6. Дерево групп и таблица элементов текущей группы

И в дереве, и в таблице все элементы доступны для редактирования. Для отображения элемента редактирования ячеек таблицы мы используем класс-делегат `MyDelegate`, который отличается от стандартного (действующего по умолчанию) только измененным цветом фона и немного расширенными размерами поля редактирования (выступающими за пределы своей ячейки на два пиксела с каждой стороны).

Приведенный пример иллюстрирует главную идею и требует доработки:

- заголовок второго столбца в дереве групп логично было бы заменить: не "ФИО", как в соответствующем столбце таблицы, а, например, "Наименование". Но для этого придется отделить в модели столбцы для дерева групп от столбцов для таблицы элементов, реализовав их по отдельности и спрятав лишние столбцы в `treeview` и `tableview`;
- в заголовке таблицы элементов желательно показывать названия текущих групп верхнего уровня (как, например, в справочниках программы 1С:Предприятие);

- навигацию по группам хорошо бы обеспечить не только с помощью дерева, но и в самой таблице;
- кроме редактирования элементов дерева и ячеек таблицы нужно реализовать возможность удаления элементов и целых групп, а также перетаскивание и тех, и других в другие группы.

15.7. Разное

ЗАДАЧА. Выполнить перебор всех элементов двумерной и/или иерархической модели `QStandardItemModel`.

Решение. Можно использовать метод `QStandardItemModel::findItems()` с шаблоном для поиска `*` и флагами `Qt::MatchWildcard | Qt::MatchRecursive`. В результате мы получаем одномерный список, перебор элементов которого выполняется, например, с помощью макроса `foreach` (рис. 15.7, листинг 5.15).

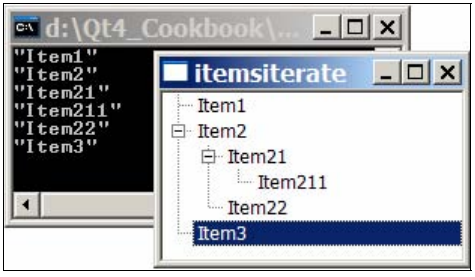


Рис. 15.7. Перебор всех элементов иерархической модели

Листинг 15.15. Файл chap15/itemsiterate/itemsiterate.cpp

```
#include <QtGui>

class TreeView : public QTreeView {
public:
    TreeView();
    void iterateOverItems();
    QList<QStandardItem*> returnTheItems();
private:
    QStandardItemModel *model;
};

TreeView::TreeView() : QTreeView() {
    model = new QStandardItemModel(3, 1);
```

```
    QStandardItem *item1 = new QStandardItem("Item1");
    QStandardItem *item2 = new QStandardItem("Item2");
    QStandardItem *item3 = new QStandardItem("Item3");

    model->setItem(0, 0, item1);
    model->setItem(1, 0, item2);
    model->setItem(2, 0, item3);

    QStandardItem *item21 = new QStandardItem("Item21");
    item2->appendRow(item21);

    QStandardItem *item211 = new QStandardItem("Item211");
    item21->appendRow(item211);

    QStandardItem *item22 = new QStandardItem("Item22");
    item2->appendRow(item22);

    setModel(model);
    header()->hide(); // заголовок нам не нужен
}

QList<QStandardItem*> TreeView::returnTheItems(){
    return model->findItems(   // "найдем" все элементы
        "*", Qt::MatchWildcard | Qt::MatchRecursive);
}

void TreeView::iterateOverItems(){ // обход всех элементов модели
    QList<QStandardItem*> list = returnTheItems();
    foreach (QStandardItem* item, list) {
        qDebug() << item->text(); // вывести отладочную информацию
    }
}

int main(int argc, char **argv){
    QApplication app(argc, argv);
    TreeView view;
    view.iterateOverItems();
    view.show();
    return app.exec();
}
```

Чтобы в системе Windows увидеть консольное окно с сообщениями отладки, выводимыми функцией `qDebug()`, нужно в файл *.pro перед выполнением команды `qmake` добавить строку

```
win32: CONFIG += console
```

ЗАДАЧА. Если одна и та же модель используется одновременно несколькими видами, как сделать, чтобы данные для некоторых ролей `Qt::ItemDataRole` были различными?

Решение. Поскольку модели ничего не известно о том, какие виды ее задействуют для получения данных, для их модификации придется применить классы-делегаты (рис. 15.8, листинг 15.16).

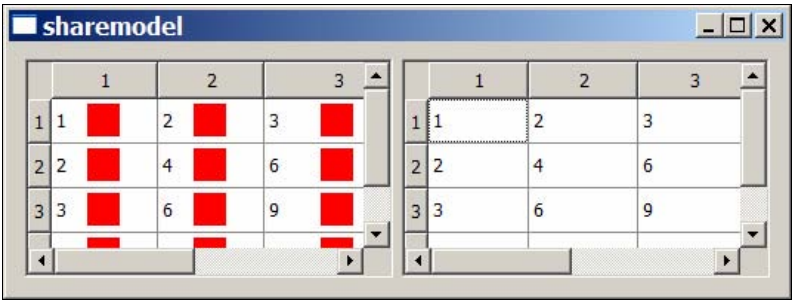


Рис. 15.8. Одна модель, но разные виды

Листинг 15.16. Файл chap15/sharemodel/sharemodel.cpp

```
#include <QtGui>
class ItemDelegate : public QItemDelegate {
public:
    ItemDelegate() : QItemDelegate() {}
    void drawDecoration (   // отобразить значок
            QPainter *painter,
            const QStyleOptionViewItem& option,
            const QRect& rect,
            const QPixmap& pixmap ) const {
        QPixmap pix(22, 22);
        pix.fill(Qt::red); // просто закрашенный квадратик
        QItemDelegate::drawDecoration(painter, option,
                                      option.rect, pix);
    }
};

int main(int argc, char **argv){
    QApplication app(argc, argv);
    QWidget widget;
    QHBoxLayout *layout = new QHBoxLayout(&widget);
    QTableView *tableView = new QTableView(&widget);
```

```
    tableView->setItemDelegate(new ItemDelegate());
    QTableView *secondtableView = new QTableView(&widget);
    QStandardItemModel *model = new QStandardItemModel;
    model->insertRows(0, 4);
    model->insertColumns(0, 4);
    for (int row = 0; row < 4; row++) {
        for (int col = 0; col < 4; col++) {
            QModelIndex index = model->index(row, col);
            model->setData(
                index,
                QVariant((row+1) * (col+1)).toString());
        }
    }
    secondtableView->setModel(model);
    tableView->setModel(model);
    layout->addWidget(tableView);
    layout->addWidget(secondtableView);
    widget.show();
    return app.exec();
}
```

Другой способ решения задачи — переопределить метод `QItemDelegate::paint()`.

ЗАДАЧА. Модифицировать элемент `QComboBox`, чтобы раскрывающийся список состоял из строк, снабженных пометками `QCheckBox` (рис. 15.9).

Решение. На самом деле раскрывающийся список в элементе `QComboBox` разработан с использованием технологии "Модель — представление", поэтому для изменения его функциональности достаточно разработать свою модель (например, на основе `QStandardItemModel`) и "назначить" ее элементу `QComboBox` с помощью метода `QComboBox::setModel()` (листинги 15.17 и 15.18).

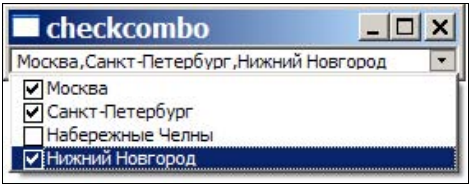


Рис. 15.9. Строки с пометками в раскрывающемся списке

Листинг 15.17. Файл chap15/checkcombo/checkcombo.h

```
01 #include <QtGui>
02
03 class CheckComboModel : public QStandardItemModel{
04    Q_OBJECT
05 public:
06    CheckComboModel(QObject* parent = 0);
07    virtual Qt::ItemFlags flags(const QModelIndex& index) const;
08    virtual QVariant data( const QModelIndex& index,
09                           int role = Qt::DisplayRole) const;
10    virtual bool setData( const QModelIndex& index,
11                          const QVariant& value,
12                          int role = Qt::EditRole);
13 signals:
14    void checkStateChanged();
15 };
16
17 //-------------------------------------
18 class CheckCombo : public QComboBox {
19    Q_OBJECT
20 public:
21    CheckCombo(QWidget *parent=0);
22    bool eventFilter(QObject *receiver, QEvent *event);
23
24    Qt::CheckState itemCheckState(int index) const;
25    void setItemCheckState(int index, Qt::CheckState state);
26
27    QStringList checkedItems() const;
28    void setCheckedItems(const QStringList& items);
29
30    void hidePopup() { }
31
32 public slots:
33    void updateCheckedItems();
34    void toggleCheckState(int index);
35 signals:
36    void checkedItemsChanged(const QStringList& items);
37 };
```

Пояснения к программе:

- (7) — метод `QStandardItemModel::flags()` должен возвращать комбинацию флагов, описывающих набор параметров элементов модели. Если пункты

требуется пометить с помощью QCheckBox, то следует установить флаг Qt::ItemIsUserCheckable;

- (8) — метод QStandardItemModel::data(index, role) должен возвращать данные, соответствующие заданному индексу и роли;
- (10) — метод QStandardItemModel::setData() используется для изменения данных модели;
- (14) — сигнал QStandardItemModel::checkStateChanged() будет генерироваться при изменении состояния какой-либо пометки;
- (22) — фильтр для событий;
- (24) — метод itemCheckState() будет возвращать состояние пометки заданного элемента модели;
- (25) — с помощью метода setItemCheckState() можно будет изменить состояние пометки указанного элемента;
- (27) — метод checkedItems() будет возвращать список помеченных элементов;
- (28) — с помощью метода setCheckedItems() можно будет задать состояния всех пометок;
- (30) — метод hidePopup() закрывает список элементов для выбора;
- (33) — метод updateCheckedItems() обновляет состояния всех пометок;
- (34) — метод toggleCheckState() изменяет состояние пометки для заданного элемента на противоположное;
- (36) — сигнал checkedItemsChanged() генерируется при изменении состояния какой-либо пометки. Его параметр хранит список всех помеченных элементов.

Листинг 15.18. Файл chap15/checkcombo/checkcombo.cpp

```
#include <QtGui>
#include "checkcombo.h"

#define DEFAULT_TEXT ""
#define SEPARATOR ","

// Реализация модели:
CheckComboModel::CheckComboModel(QObject *parent)
          : QStandardItemModel(0, 1, parent) { }

Qt::ItemFlags CheckComboModel::flags(
             const QModelIndex& index) const{
```

```
    return QStandardItemModel::flags(index) |
           Qt::ItemIsUserCheckable;
}

QVariant CheckComboModel::data(  // Данные, хранящиеся в модели
            const QModelIndex& index, int role) const{
    QVariant value = QStandardItemModel::data(index, role);
    if (  index.isValid() && role == Qt::CheckStateRole &&
          !value.isValid() )
        value = Qt::Unchecked;

    return value;
}

bool CheckComboModel::setData(  // Записать новые данные в модель
                    const QModelIndex& index,
                    const QVariant& value, int role){
    bool ok = QStandardItemModel::setData(index, value, role);
    if (ok && role == Qt::CheckStateRole) {
        emit dataChanged(index, index);
        emit checkStateChanged();
    }
    return ok;
}

//--------------------------------------------------------------
// Реализация самого виджета:
CheckCombo::CheckCombo(QWidget* parent)
          : QComboBox(parent) {
    setModel(new CheckComboModel(this));
    connect(this, SIGNAL(activated(int)),
            this, SLOT(toggleCheckState(int)));
    connect(model(), SIGNAL(checkStateChanged()),
            this, SLOT(updateCheckedItems()));

    QLineEdit* lineEdit = new QLineEdit(this);
    lineEdit->setReadOnly(true); // Нельзя вводить произвольный текст
    setLineEdit(lineEdit);

    view()->installEventFilter(this); // Фильтр для событий
    view()->viewport()->installEventFilter(this);
    this->installEventFilter(this);
}
```

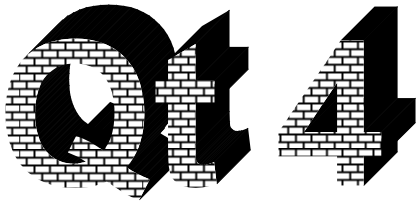
```
bool CheckCombo::eventFilter(QObject *receiver, QEvent *event){
    switch (event->type()) {
        case QEvent::KeyPress:       // Нажатие клавиши
        case QEvent::KeyRelease: {   // Отпускание клавиши
            QKeyEvent* keyEvent = static_cast<QKeyEvent*>(event);
            if ( receiver == this &&
                 (keyEvent->key() == Qt::Key_Up ||     // Стрелка вверх
                  keyEvent->key() == Qt::Key_Down) ) { // Стрелка вниз
                showPopup();  // Показать раскрывающийся список
                return true;
            }
            else if (keyEvent->key() == Qt::Key_Escape) {
                QComboBox::hidePopup(); // Скрыть список
            }
            return false;
        }
        default:
            return false;
    }
}

void CheckCombo::updateCheckedItems(){ // Обновить с учетом пометок
    QStringList items = checkedItems();
    if (items.isEmpty())           // Если пометок нет
        setEditText(DEFAULT_TEXT); // Текст по умолчанию
    else
        setEditText(items.join(SEPARATOR));

    emit checkedItemsChanged(items);
}

void CheckCombo::toggleCheckState(int index){ // Переключить пометку
    QVariant value = itemData(index, Qt::CheckStateRole);
    if (value.isValid()){
        Qt::CheckState state =
            static_cast<Qt::CheckState>(value.toInt());
        setItemData(
            index,
            state == Qt::Unchecked ? Qt::Checked : Qt::Unchecked,
            Qt::CheckStateRole );
    }
}
```

```
Qt::CheckState CheckCombo::itemCheckState(int index) const{
    return static_cast<Qt::CheckState>(
        itemData(index, Qt::CheckStateRole) .toInt() );
}

void CheckCombo::setItemCheckState(
            int index, Qt::CheckState state) {
    setItemData(index, state, Qt::CheckStateRole);
}

// Вернуть список помеченных элементов:
QStringList CheckCombo::checkedItems() const {
    QStringList items;
    if (model()) {
        QModelIndex index = model()->index(
            0, modelColumn(), rootModelIndex());
        QModelIndexList indexes = model()->match(
            index, Qt::CheckStateRole, Qt::Checked,
            -1, Qt::MatchExactly);
        foreach (const QModelIndex& index, indexes)
            items += index.data().toString();
    }
    return items;
}

// Установить пометки:
void CheckCombo::setCheckedItems(const QStringList& items){
    foreach (const QString& text, items){
        const int index = findText(text);
        setItemCheckState(
            index, index != -1 ?
            Qt::Checked : Qt::Unchecked );
    }
}

//-------------------------------------------------
int main(int argc, char *argv[]) {

    QApplication app(argc, argv);

    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);
```

```
    QStringList items;
    items << QObject::tr("Москва")
          << QObject::tr("Санкт-Петербург")
          << QObject::tr("Набережные Челны")
          << QObject::tr("Нижний Новгород");
    CheckCombo *c = new CheckCombo();
    c->addItems(items);
    c->show();

    return app.exec();
}
```

Мы еще вернемся к технологии "Модель — представление" при обсуждении таблиц баз данных.

ГЛАВА 16



Процессы, потоки, синхронизация

Это ужасно, что вечность состоит из отчетных периодов.

Станислав Ежи Лец

Напомним, что *процесс* выполнения любого приложения может разбиваться на несколько отдельных *потоков*, которые выполняются в общем адресном пространстве своего процесса. При этом каждый *поток* (*thread*, *тред*, *облегченный процесс*) использует свой собственный стек и при необходимости может работать со своей собственной очередью событий. На однопроцессорных системах потоки выполняются по очереди, последовательно, но малыми квантами времени, что создает иллюзию их параллельного выполнения. На многопроцессорных системах потоки могут работать по-настоящему параллельно, одновременно друг с другом. Обычно приходится прибегать к многопоточному программированию в тех случаях, когда требуется, чтобы длительные операции по обработке данных (или ожидание прихода этих данных от сервера) не блокировали работу пользовательского интерфейса.

16.1. Процессы

Даже ничтожный процесс требует долгих лет мучительного вызревания.

Эмиль Золя

Для работы с отдельно выполняемыми процессами в библиотеке Qt используется класс `QProcess`. При создании экземпляра процесса, как обычно, указывается родительский элемент. На выполнение процесс запускается с помощью метода `start()`, параметры которого — имя исполняемого файла и массив строк — параметры командной строки. В системе Windows параметры, содержащие символ пробела, заключаются в кавычки.

Выход из метода `start()` и продолжение функционирования основной программы не означает, что вызванный процесс завершил свою работу или что он хотя бы запущен на выполнение. С помощью этого метода мы всего лишь разрешаем запуск процесса, а сам запуск произойдет позже, когда операционная система выполнит необходимые действия с исполняемым файлом программы, а планировщик задач выделит для ее выполнения очередной квант процессорного времени. В момент фактического старта процесса будет сгенерирован сигнал `started()`. Если окажется, что выполнение программы невозможно (например, отсутствует указанный файл или файл не является исполняемым), вместо этого будет сгенерирован сигнал

```
error(QProcess::ProcessError error)
```

По окончании выполнения программы генерируется сигнал

```
finished(int exitCode, QProcess::ExitStatus exitStatus)
```

ЗАДАЧА. Открыть HTML-файл в браузере Internet Explorer. Отследить момент закрытия окна браузера пользователем.

Решение. Текст программы, решающей поставленную задачу, приведен в листинге 16.1.

Листинг 16.1. Файл chap16/process/process.cpp

```
 1  #include <QApplication>
 2  #include <QProcess>
 3  #include <QLabel>
 4
 5  int main (int argc, char * argv[]){
 6    QApplication app(argc, argv);
 7    QLabel label("Starting Internet Explorer...");
 8    label.show();
 9
10    QProcess *process = new QProcess(&label);
11    QStringList arg;
12    arg << QString(
          "c:\\Qt4_Cookbook\\examples\\process1\\index.html");
13    QObject::connect(
14        process, SIGNAL(finished(int, QProcess::ExitStatus)),
15        &label, SLOT(setNum(int)) );
16    process->start(
17        QString(
              "c:\\program files\\internet explorer\\iexplore.exe"),
18        arg );
```

```
19
20     return app.exec();
21  }
```

Пояснения к программе:

- (7, 8) — создаем надпись `QLabel` и выводим ее на экран;
- (10) — создаем объект `QProcess`;
- (11, 12) — параметр всего один — имя файла HTML;
- (13–15) — когда процесс завершится (пользователь закроет окно браузера), текст надписи отобразит код завершения (как и любая программа, Internet Explorer возвращает нулевой код при обычном закрытии);
- (16–18) — запускаем процесс.

16.2. Потоки

Для запуска отдельного потока выполнения средствами библиотеки Qt нужно выполнить следующее:

- определить свой класс, например, с именем `MyThread`, на основе базового класса `QThread`;
- переопределить в нем виртуальный метод `run()`, который будет выполнять необходимые действия в каждом потоке. Если поток должен использовать отдельную очередь событий, то в методе `run()` требуется вызвать метод `exec()`;
- создать экземпляр класса `MyThread` и запустить поток с помощью метода `start()`.

Если поток в данный момент выполняется, то функция `QTread::isRunning()` возвращает `true`. Для выполняющегося потока с помощью метода `setPriority()` можно задать приоритет. Данные, необходимые для работы каждого потока, хранятся в переменных — членах класса `MyThread` или специально предназначенном для этого классе `QThreadStorage`. Поскольку все потоки выполняются в адресном пространстве своего процесса, то передача данных между ними может осуществляться с помощью глобальных переменных или статических переменных — членов класса `MyThread`. Для организации межпоточных взаимодействий предназначены сообщения, сигналы и слоты.

Замечание

Соединение с базой данных нельзя передать в другой поток. Приходится после запуска потока создавать для него свое собственное соединение.

ЗАДАЧА. Узнать, к какому потоку принадлежит данный объект-потомок класса `QObject` и при необходимости "перевести" его в другой поток.

Решение. Метод `QObject::thread()` возвращает указатель на поток, в котором выполняется данный объект. Перевести объект в другой поток можно с помощью метода

```
void QObject::moveToThread ( QThread *targetThread )
```

Например, для перевода объекта `myObject` в основной поток:

```
myObject->moveToThread(QApplication::instance()->thread());
```

Следует иметь в виду, что объект, имеющий родителя `parent`, нельзя перевести в другой поток. Сначала его нужно "оставить без родителя", выполнив метод `setParent(0)`.

Поскольку сам метод `moveToThread()` не является потокобезопасным, его следует применять только для перевода объекта, в настоящее время принадлежащего данному потоку, в какой-нибудь другой поток, но не наоборот.

16.3. Реентерабельность и потокобезопасность

> Все давно уже сказано, но т. к. никто не слушает, приходится постоянно возвращаться назад и повторять все сначала.
>
> *Андре Жид*

В документации, поставляемой вместе с библиотекой Qt, в описании многих функций и классов встречаются понятия *реентерабельный* и *потокобезопасный*.

Реентерабельной (*reentrant*, *повторно входимой*) называется такая функция, которая может быть вызвана из другого потока до завершения ее исполнения в предшествующем вызове. Если внутри функции читаются и/или изменяются значения некоторых глобальных переменных, то, разумеется, такая функция не будет реентерабельной, поскольку второй ее вызов перезапишет значения глобальных переменных, установленные в предыдущем вызове, и когда тот возобновит свою работу, то значения глобальных переменных окажутся для него неверными. Поэтому реентерабельная функция при каждом своем вызове должна ссылаться только на свой собственный, уникальный объект данных.

Класс называется *реентерабельным*, если сразу несколько потоков, работающих каждый со своим экземпляром этого класса, могут одновременно вызывать любой из его методов. Если в классе изменяются и используются значения статических переменных, то он не будет реентерабельным.

Часто требуется, чтобы несколько потоков работали с одним и тем же объектом данных. Функция называется *потокобезопасной* (*thread-safe*), если она может быть вызвана параллельно из нескольких потоков, работающих с общим объектом данных. При этом результат работы предсказуем и не зависит от того, в каком порядке планировщик задач операционной системы инициировал выполнение этих потоков.

Аналогично, класс называют *потокобезопасным*, если сразу несколько потоков, работающих с одним и тем же экземпляром этого класса, могут одновременно вызывать любой из его методов.

Как видим, потокобезопасность — более сильное условие. Потокобезопасный класс всегда реентерабелен, реентерабельный класс может оказаться не потокобезопасным. Большинство классов библиотеки Qt реентерабельные, но потокобезопасны только те, что напрямую связаны с использованием потоков. Поэтому при многопоточном программировании приходится прибегать к некоторым ухищрениям.

Рассмотрим простейший пример из документации (листинг 16.2).

Листинг 16.2

```
class Counter {
public:
   Counter() { n = 0; }
   void increment() { ++n; }
   void decrement() { --n; }
   int value() const { return n; }
private:
   int n;
};
```

Данный класс реентерабелен, т. к. его методы не используют ни глобальных, ни статических переменных. Экземпляры класса `Counter` могут спокойно работать в разных потоках, не мешая друг другу. Но если сразу несколько потоков захотят обратиться к одному и тому же экземпляру этого класса? Дело в том, что операции инкремента и декремента в C++ не являются атомарными, обычно они транслируются в три машинные команды: 1) загрузить значение переменной из памяти в регистр процессора; 2) выполнить инкремент или декремент; 3) сохранить результат из регистра обратно в память.

Допустим, два потока захотят выполнить инкремент переменной `n`. Если они будут действовать последовательно, друг за другом, то значение переменной `n`, разумеется, увеличится на два. А если действия потоков несогласованны? Первый поток начнет операцию, но планировщик задач приостановит его

выполнение, например, сразу после первой или второй машинной команды, после чего сохранит в служебной области памяти контекст прерываемого потока (в том числе и значения регистров) и передаст управление второму потоку. Тот, ничего не зная о действиях первого потока, прочитает не успевшее измениться значение переменной `n`, увеличит его на единицу и запишет в память. Когда первый поток возобновит свое выполнение, он запишет в ячейку памяти, отведенную для переменной `n`, свое значение, которое окажется тем же самым, что уже хранится там после действий второго потока. Другими словами, вместо того, чтобы увеличиться на два, значение переменной `n` увеличится только на единицу. Наш класс оказался не потокобезопасным. Результат деятельности нескольких потоков непредсказуем, если он зависит от того, как соотносятся между собой моменты обращений потоков к общей переменной и действия планировщика задач операционной системы.

Чтобы выполнить условия потокобезопасности, придется усложнить процедуру доступа к общим переменным, сделав ее атомарной: если уж один поток начал работать с общей переменной, то все остальные потоки должны подождать, пока он не сохранит результат своей работы. Другими словами, работу потоков требуется каким-то образом упорядочить или, как говорят, синхронизировать.

16.4. Методы синхронизации процессов и потоков

То сказано глупцом и признано глупцами,
Что будто смерть для нас творит ужасным свет!
Пока на свете мы, она еще не с нами;
Когда ж пришла она, то нас на свете нет!
Василий Андреевич Жуковский

Ситуация, описанная в эпиграфе, — пожалуй, одна из немногих, когда выполняемые задачи лучше так и оставить несинхронизированными. В большинстве других случаев небрежное многопоточное программирование приводит к случайно проявляющимся и трудно обнаруживаемым ошибкам выполнения программ.

Для синхронизации потоков в библиотеке Qt, как и при обычном программировании на C++, используют семафоры и мьютексы.

16.4.1. Мьютексы

Мьютекс (*mutual exclusion* — взаимное исключение) позволяет одному потоку заблокировать доступ к общему объекту данных. В библиотеке Qt для выполнения блокировки имеются классы `QMutex`, `QMutexLocker` и `QReadWriteLock`.

Общий порядок работы с классом `QMutex`:

- создать общий для всех потоков мьютекс:

```
QMutex mutex();
```

- в нужный момент (перед началом доступа потока к общему объекту данных) заблокировать мьютекс:

```
mutex.lock();
```

  После того, как мьютекс заблокирован ("захвачен") каким-либо потоком (говорят, что поток становится "владельцем" мьютекса), выполнение данной операции с этим же мьютексом в других потоках приводит к циклу ожидания до тех пор, пока поток, выполнивший блокирование мьютекса, не разблокирует его. Все запросы на захват мьютекса от других потоков ставятся в очередь к данному мьютексу. В этой очереди потоки упорядочены по своему приоритету. Чтобы не терять время на ожидание, можно использовать "неблокирующий" вариант блокирования (приведенное словосочетание, хоть и несколько режет слух, тем не менее, как нельзя точно отражает существо дела):

```
mutex.tryLock();
```

  Метод `tryLock()` пытается выполнить блокирование мьютекса. Если тот уже заблокирован другим потоком, то возвращается значение `false`, а выполнение данного потока продолжается, чтобы при необходимости сделать какую-либо другую работу, пока общий объект данных недоступен. При вызове `tryLock()` можно указать длительность промежутка времени (в миллисекундах), в течение которого будет выполняться попытка блокирования мьютекса и по истечении которого, если попытка так и осталась безрезультатной, произойдет выход из функции `tryLock()` с возвращением результата `false`;

- после окончания работы с общим объектом данных разблокировать мьютекс:

```
mutex.unlock();
```

Мьютекс можно разблокировать только в том потоке, в котором он был заблокирован.

При создании мьютекса можно указать параметр `QMutex::Recursive`. Тогда поток, заблокировавший мьютекс (и только он!), сможет выполнять повторные блокировки неограниченное число раз. Но для разблокирования рекурсивного мьютекса потребуется соответствующее количество вызовов метода `unlock()`. По умолчанию мьютекс является нерекурсивным (действует параметр `QMutex::NonRecursive`). Попытка повторного блокирования нерекурсив-

ного мьютекса из этого же потока приведет к взаимной блокировке (*dead lock*).

ЗАДАЧА. С помощью мьютекса сделать класс `Counter`, описанный в предыдущем разделе, потокобезопасным.

Решение иллюстрирует листинг 16.3.

Листинг 16.3

```
class Counter {
public:
   Counter() { n = 0; }
   void increment() { mutex.lock(); ++n; mutex.unlock(); }
   void decrement() { mutex.lock(); --n; mutex.unlock(); }
   int value() const { mutex.lock(); int r = n; mutex.unlock();
                       return r; }
private:
   mutable QMutex mutex;
   int n;
};
```

Класс `QMutexLocker` позволяет упростить работу с мьютексами. Чтобы заблокировать мьютекс, нужно создать для него объект `QMutexLocker`:

```
QMutexLocker locker(&mutex);
```

После этого мьютекс можно разблокировать с помощью вызова

```
locker.unlock();
```

и при необходимости попытаться блокировать заново:

```
locker.relock();
```

При уничтожении объекта `QMutexLocker` (например, при выходе из функции, в которой он был создан как локальная переменная) его мьютекс разблокируется автоматически.

ЗАДАЧА. С помощью объекта `QMutexLocker` сделать класс `Counter`, описанный в предыдущем разделе, потокобезопасным.

Решение иллюстрирует листинг 16.4.

Листинг 16.4

```
class Counter {
public:
   Counter() { n = 0; }
```

```
    void increment() { QMutexLocker locker(&mutex); ++n; }
    void decrement() { QMutexLocker locker(&mutex); --n; }
    int value() const { QMutexLocker locker(&mutex); return n; }
private:
    mutable QMutex mutex;
    int n;
};
```

16.4.2. Читатели и писатели

Иногда для повышения производительности желательно разрешить сразу нескольким потокам получать доступ к общему объекту данных, если они не собираются изменять эти данные, а хотят только прочитать их. В этом случае можно применить класс `QReadWriteLock`. Работает он аналогично `QMutex`, но имеет два вида блокирования:

- "Блокирование для чтения" — метод `QReadWriteLock::lockForRead()`, выполненный в одном потоке, не запрещает другим потокам заблокировать мьютекс для чтения еще раз, тоже получив доступ к общему объекту данных.
- "Блокирование для записи" — метод `QReadWriteLock::lockForWrite()`, выполненный в одном потоке, безусловно запрещает другим потокам блокировать объект `QReadWriteLock` как для чтения, так и для записи (иначе как мы можем быть уверенными, что читаемые данные окажутся верными?).

Блокирование для записи сможет успешно выполниться только после того, как все другие блокировки (в том числе и для чтения) будут сняты. Блокированию для чтения может помешать только действующая или запрошенная блокировка для записи (если бы потоки-читатели имели приоритет, то поток-писатель мог бы никогда не дождаться своей очереди).

Как и в случае с `QMutex`, при создании объекта `QReadWriteLock` можно указать параметр `QReadWriteLock::Recursive`, если требуется, чтобы один и тот же поток имел возможность блокировать его произвольное число раз. Оба вида блокирования имеют, как и `QMutex`, "неблокирующие" варианты `tryX()` с параметром и без.

16.4.3. Семафоры

Мьютекс позволяет упорядочить доступ нескольких потоков к одному общему ресурсу. *Семафор* `QSemaphore` ("мьютекс со счетчиком") — это обобщение мьютекса на произвольное количество однотипных ресурсов.

Порядок работы с семафором:

- создать объект `QSemaphore`, указав начальное значение его счетчика, например:

```
QSemaphore sem(5);
```

- запросить доступ с помощью метода `acquire(n)`, где `n` — число объектов (по умолчанию 1), к которым надо получить доступ. Если значение счетчика данного семафора меньше `n`, то поток, запросивший доступ, входит в цикл ожидания, пока нужные ресурсы не освободятся. Если значение счетчика больше или равно `n`, то оно уменьшается на `n`, а поток получает доступ к запрошенным ресурсам;
- после выполнения необходимых действий вызвать метод `release(m)`, в результате указанное количество ресурсов будет освобождено для использования в других потоках.

Как и другие примитивы синхронизации, семафоры имеют "неблокирующий" вариант запроса на доступ к ресурсам `tryAcquire(n)`, который возвращает `false`, если в наличии нет нужного числа свободных ресурсов.

Семафор с единичным начальным значением счетчика по логике работы эквивалентен мьютексу, за тем исключением, что любой поток, вообще говоря, имеет право освободить произвольное количество ресурсов. В результате значение счетчика семафора может произвольно возрастать, даже превысив свое начальное значение, но всегда оставаясь неотрицательным. Более того, освобождать семафор могут не обязательно те потоки, которые его захватывали.

Рассмотрим, например, следующую модель: один или несколько потоков генерируют данные ("производители"), а другие потоки эти данные обрабатывают ("потребители"). Тогда поток первого типа освобождает семафор, как только он подготовил очередную порцию информации для обработки, а поток второго типа захватывает семафор, перед тем как начать обработку очередной порции информации. Если в начале работы нет данных, которые нуждаются в обработке, то начальное значение счетчика семафора должно быть нулевым. Мы еще вернемся к этой модели в *разд. 16.5*.

16.4.4. Условие ожидания

Другим обобщением мьютекса является *условие ожидания* (условная переменная) `QWaitCondition`. Мьютекс позволяет потоку получить монопольный доступ к общему ресурсу, если тот не занят другим потоком. Условная переменная разрешает такой доступ лишь при выполнении некоторого условия.

Порядок использования условий ожидания:

- создать мьютекс `QMutex` или объект `QReadWriteLock`;
- создать условную переменную:

```
QWaitCondition cond;
```

- заблокировать мьютекс (или объект `QReadWriteLock`);
- начать ожидание выполнения условия, вызвав метод `wait()`:

```
cond.wait(&mutex);
```

  Мьютекс при этом освобождается. Вторым необязательным параметром можно указать максимальное время ожидания в миллисекундах. Если мьютекс не был предварительно заблокирован или является рекурсивным, то цикл ожидания не начинается. Если потоку удается дождаться выполнения условия до истечения указанного времени, метод `wait()` возвращает `true`. Если время не указано, то ожидание продолжается неопределенно долго;

- из некоторого другого потока дать знать заблокированному потоку о выполнении ожидаемого им условия, вызвав для данной условной переменной метод `wakeAll()` или `wakeOne()`. В первом случае будут "разбужены" в некотором порядке (зависящем от алгоритма работы планировщика задач операционной системы) все процессы, вызвавшие метод `wait()` для данной условной переменной. Во втором случае закончится цикл ожидания только одного из таких потоков (какого именно — зависит, опять-таки, от планировщика задач).

16.5. Производитель — потребитель

ЗАДАЧА. Выполнить моделирование классической задачи "производитель — потребитель".

Решение. Процесс-производитель генерирует некоторые данные и записывает их в очередь. Процесс-потребитель считывает данные из очереди. Если очередь пуста, то потребитель ждет прихода данных. Если очередь переполняется, то производитель ждет, когда она освободится. Код программы приведен в листинге 16.5.

Листинг 16.5. Файл chap16/consprod/consprod.cpp

```
1  #include <QtCore>
2  #include <QDebug>
3
```

```
// Глобальные переменные:
QQueue<int> queue;  // Очередь целых чисел
int maxlen;         // Макс.кол-во чисел в очереди
QMutex mutex;       // Мьютекс
QWaitCondition queueNotFull;  // Условие "очередь не полна"
QWaitCondition queueNotEmpty; // Условие "очередь не пуста"

// Производитель:
class Producer : public QThread {
public:
   Producer(QObject *parent=0)
      : QThread(parent) {}
protected:
   void produce() {
      qDebug() << "Producing...";
      mutex.lock();
      if (queue.size() == maxlen) {
          qDebug() << "queue is full, waiting!";
          queueNotFull.wait(&mutex);
      }
      queue.enqueue( ( rand() % 100 ) + 1 );
      queueNotEmpty.wakeAll();
      mutex.unlock();
   }

   void Producer::run() {
      forever {
          produce();
          msleep( ( rand() % 3000 ) + 1);
      };
   }
};

// Потребитель:
class Consumer : public QThread {
public:
   Consumer(QObject *parent=0)
      : QThread(parent) {}
protected:
   int consume() {
      qDebug() << "Consuming...";
      QMutexLocker locker(&mutex);
```

```
46        if (queue.isEmpty()) {
47            qDebug() << "queue empty, waiting!";
48            queueNotEmpty.wait(&mutex);
49        }
50        int val = queue.dequeue();
51        queueNotFull.wakeAll();
52        return val;
53    }
54
55    void run(){
56        forever {
57           qDebug() << consume();
58           msleep( ( rand() % 4000 ) + 1);
59        }
60    }
61 };
62
63 //---------------------------------
64 int main() {
65    maxlen = 10;
66    Producer producer;
67    Consumer consumer;
68    producer.start();
69    consumer.start();
70    producer.wait();
71    consumer.wait();
72    return 0;
73 }
```

Пояснения к программе:

- (4–9) — объявляем глобальные переменные, которые будут использоваться как производителем, так и потребителем;
- (12–35) — класс-производитель;
- (17–27) — метод, генерирующий данные;
- (19) — блокируем мьютекс;
- (20–23) — проверяем, не переполнилась ли очередь. Если да, то ждем, когда появится свободное место (22). На время ожидания мьютекс освобождается (первый параметр метода `wait()`);
- (24) — помещаем в конец очереди случайное число в диапазоне от 1 до 100;
- (25) — "будим" все потоки, которые ожидали выполнения условия "очередь не пуста";

- (26) — освобождаем мьютекс;
- (29–34) — цикл работы производителя: сгенерировать очередное число и ждать от 1 мс до 3 с;
- (38–61) — класс-потребитель;
- (43–53) — метод, получающий данные из очереди;
- (45) — блокируем мьютекс;
- (46–49) — если очередь пуста, ждем, пока не появятся данные. На время ожидания мьютекс освобождается;
- (50) — извлекаем из начала очереди порцию данных (целое число, помещенное туда производителем);
- (51) — "будим" все потоки, которые ожидали выполнения условия "очередь не переполнена";
- (52) — возвращаем прочитанное из очереди число;
- (55–60) — цикл работы потребителя: прочитать очередное число из очереди и ждать от 1 мс до 4 с;
- (64–73) — главная программа: создаем одного потребителя и одного производителя, запускаем оба потока и ждем их завершения (70, 71). Поскольку выход из метода `QThread::start()` происходит сразу (запущенный поток начинает выполняться параллельно с основным), без вызова метода `QThread::wait()` наша программа сразу бы закончилась, при этом основной поток автоматически бы завершил все остальные. Из-за бесконечных циклов `forever` потоки производителя и потребителя сами по себе никогда не завершатся, поэтому выход из программы произойдет, только если пользователь нажмет комбинацию клавиш <Ctrl>+<C>.

Перед компиляцией программы после создания файла *.pro необходимо добавить в него строку

```
CONFIG += console
```

16.6. Мультипликация

ЗАДАЧА. Реализовать приложение, в котором отдельный вспомогательный поток через заданные промежутки времени инициирует смену графических изображений, отображаемых основным потоком (рис. 16.1).

Решение. Самый простой способ обеспечить взаимодействие дочерних потоков с пользовательским интерфейсом приложения — принимать от них сигналы и вызывать соответствующие слоты (листинги 16.6–16.8).

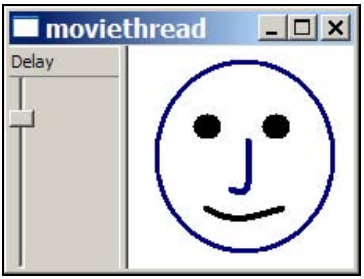


Рис. 16.1. Положение ползунка задает скорость смены картинок

Листинг 16.6. Файл chap16/moviethread/moviethread.h

```
#include <QtGui>

class MovieThread : public QThread {
    Q_OBJECT
public:
    MovieThread();
    ~MovieThread(){}
    void run();
    void loadPics();
signals:
    void show(const QPixmap *image);
public slots:
    void stop();
    void setInterval(int delay);
private:
    QVector<QPixmap> pics;
    int delay;
};

//-----------------------------------
class MovieWin : public QMainWindow {
    Q_OBJECT
public:
    MovieWin();
    void loadPics();
public slots:
    void setDelay(int delay);
    void showPix(const QPixmap* pix);
signals:
    void intervalChanged(int newInterval);
```

```
private:
    QLabel *label;
    QSlider *slider;
};
```

Листинг 16.7. Файл chap16/moviethread/moviethread.cpp

```
#include "moviethread.h"

MovieThread::MovieThread()
           : QThread(),
             delay(100) {
    loadPics();
}

void MovieThread::run() {
    int current = 0;
    int count = pics.size();
    forever {
        msleep(delay);
        emit show(&pics[current]);
        current = (current + 1) % count;
    }
}

void MovieThread::stop() {
    terminate();
    wait(5000);
}

void MovieThread::loadPics() {
    pics << QPixmap("img/1.png")
         << QPixmap("img/2.png")
         << QPixmap("img/3.png")
         << QPixmap("img/4.png")
         << QPixmap("img/5.png")
         << QPixmap("img/6.png");
}

void MovieThread::setInterval(int delay) {
    this->delay = delay;
}
```

```
//-----------------------------------------------
MovieWin::MovieWin() {
   resize(200, 200);
   slider = new QSlider(Qt::Vertical, this);
   slider->setRange(1, 1000);
   slider->setTickInterval(50);

   connect(slider, SIGNAL(valueChanged(int)),
           this,   SLOT(setDelay(int)));
   QDockWidget *dw = new QDockWidget("Delay");
   dw->setWidget(slider);
   addDockWidget(Qt::LeftDockWidgetArea, dw);
   label = new QLabel(tr("Movie"), this);
   setCentralWidget(label);
}

void MovieWin::showPix(const QPixmap* pic) {
    label->setPixmap(*pic);
}

void MovieWin::setDelay(int delay) {
    QString str;
    str = QString("%1 ms").arg(delay);
    slider->setToolTip(str);
    emit intervalChanged(delay);
    if (slider->value() != delay)
        slider->setValue(delay);
}
```

Листинг 16.8. Файл chap16/moviethread/main.cpp

```
#include <QApplication>
#include "moviethread.h"

int main(int argc, char** argv) {
    QApplication app(argc, argv);
    MovieWin win;
    MovieThread movie;
    app.connect(&movie, SIGNAL(show(const QPixmap*)),
                &win, SLOT(showPix(const QPixmap*)),
                Qt::QueuedConnection);
    app.connect(&win, SIGNAL(intervalChanged(int)),
                &movie, SLOT(setInterval(int)),
                Qt::QueuedConnection);
```

```
    app.connect(&app, SIGNAL(aboutToQuit()),
                &movie, SLOT(stop()));
    win.setDelay(500);
    movie.start();
    win.show();
    return app.exec();
}
```

Нам еще предстоит вернуться к многопоточному программированию при обсуждении работы с базами данных.

16.7. *QtConcurrent:* высокоуровневое API для параллельного программирования

Начиная с релиза 4.4, в библиотеке Qt в составе модуля `QtCore` появилось новое пространство имен: `QtConcurrent`. Оно включает в себя набор функций для поддержки параллельных потоков и реализует интерфейс более высокого уровня, чем `QThread`.

Чтобы запустить свою функцию в отдельном потоке, достаточно передать ее в качестве аргумента специальной функции `QtConcurrent::run()`. Если программа выполняется в многопроцессорной системе, то для выполнения заданной функции будет создан отдельный поток.

ЗАДАЧА. Написать многопоточный вариант программы "Hello" с использованием `QtConcurrent` (рис. 16.2).

Решение иллюстрируют листинги 16.9–16.10.

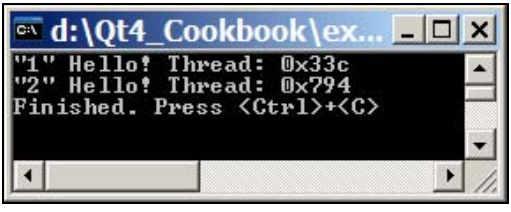


Рис. 16.2. Приветствия из разных потоков

Листинг 16.9. Файл chap16/run/run.cpp

```
#include <QtCore>

void hello(QString s){
    qDebug() << s << "Hello!"
```

```
            << "Thread:" << QThread::currentThreadId();
}

int main(int argc, char *argv[]) {
    QCoreApplication app(argc, argv);

    QFuture<void> f1 = QtConcurrent::run(hello, QString("1"));
    QFuture<void> f2 = QtConcurrent::run(hello, QString("2"));
    f1.waitForFinished();
    f2.waitForFinished();

    qDebug() << "Finished. Press <Ctrl>+<C>";
    return app.exec();
}
```

Листинг 16.10. Файл chap16/run/run.pro

```
TEMPLATE = app
QT -= gui
TARGET =
DEPENDPATH += .
INCLUDEPATH += .
SOURCES += run.cpp
CONFIG += console
```

Здесь функция, предназначенная для выполнения в отдельных потоках, выводит на экран приветствие, свой аргумент (строку текста) и идентификатор потока. Чтобы не отвлекаться на "возню" с кодировками консоли, мы пишем по-английски. Метод `QFuture::waitForFinished()` ожидает, пока не закончится выполнение заданной функции.

На рис. 16.2 показан результат работы программы в системе с двухъядерным процессором (для однопроцессорной системы идентификаторы потоков будут совпадать).

Если требуется, чтобы функция имела несколько аргументов и/или возвращала какое-либо значение, то текст программы изменится не слишком сильно (листинг 16.11).

Листинг 16.11

```
QString func(int a1, double a2, const QString &a3) {
   . . .
}
```

```
. . .
int i = ...;
double f = ...;
QString s = ...;
QFuture<QString> future = QtConcurrent::run(func, i, f, s);
```

Вызов `QtConcurrent::run()` только дает задание на асинхронное выполнение функции, но не ожидает, пока будет создан и запущен новый поток (тем более что в однопроцессорной системе новый поток даже не будет создан, выполнение задания произойдет позже в основном потоке). Вызов `QtConcurrent::run()` возвращает объект типа `QFuture<T>`, по свойствам которого мы сможем проследить за дальнейшей судьбой нашего задания. Например, следующие методы возвращают `true` если:

- `QFuture::isStarted()` — задание уже начало выполняться;
- `QFuture::isRunning()` — оно выполняется сейчас;
- `QFuture::isPaused()` — оно приостановлено с помощью метода `QFuture::pause()`;
- `QFuture::isFinished()` — задание уже выполнилось.

Метод `QFuture::result()` возвращает результат выполнения. Если он еще не готов, то данный метод ожидает, пока выполнение нашей функции не будет закончено.

Часто бывает так, что у нас имеется множество некоторых данных и мы хотим выполнить одну и ту же процедуру обработки каждого элемента этих данных. Для таких случаев предназначены функции

```
QFuture<void> QtConcurrent::map (
    Sequence & sequence,
    MapFunction function )
```

и

```
QFuture<T> QtConcurrent::mapped (
    const Sequence & sequence,
    MapFunction function )
```

Первая позволяет вызвать функцию вида

```
T function(Type &t);
```

для каждого элемента последовательности `sequence` (каждый элемент имеет тип `Type` и может изменяться в результате работы функции), а вторая — функцию вида

```
T function(const Type &t);
```

Функция `mapped()` возвращает набор результатов работы каждого вызова, доступный через методы

```
QList<T> QFuture::results()
```

или

```
T QFuture::resultAt( int index ) const
```

Чтобы не блокировать основной поток ожиданием готовности всех результатов, используют специальный объект `QFutureWatcher`, который следит за объектом `QFuture` и формирует сигнал `finished()`, когда все задания будут выполнены.

ЗАДАЧА. Выполнить в отдельных потоках функцию обработки для каждого элемента массива данных, а в основном потоке следить за ходом процесса с помощью индикатора `QProgressDialog`.

Решение иллюстрируют листинги 16.12, 16.13 и рис. 16.3.

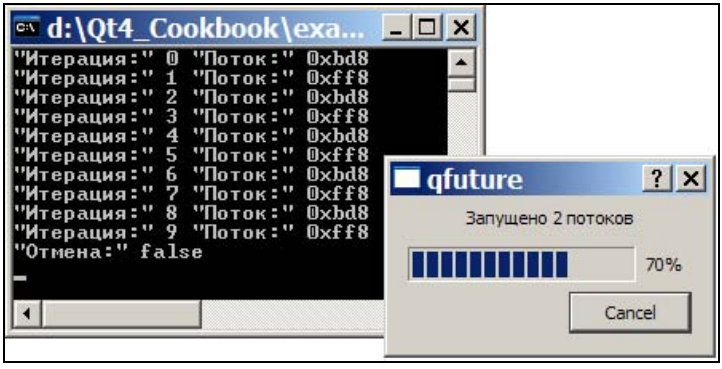


Рис. 16.3. Использование объекта `QFuture` и функции `QtConcurrent::map()`

Листинг 16.12. Файл chap16/qfuture/qfuture.cpp

```
#include <QtGui>
#include <QDebug>

#define tr  QObject::tr

//using namespace QtConcurrent; // можно использовать при желании
const int iterations = 10;

void foo(int &iteration) { // имитируем сложные вычисления
    const int work = 100000000;
```

```
    volatile int v = 0;
    for (int j = 0; j < work; ++j)
        ++v;

    qDebug() << tr("Итерация:") << iteration
             << tr("Поток:") << QThread::currentThreadId();
}

int main(int argc, char **argv){
    QApplication app(argc, argv);
    QTextCodec *incodec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(incodec);

    // Т. к. кодировка консоли MS Windows отличается от системной:
    #ifdef Q_WS_WIN
    QTextCodec *outcodec = QTextCodec::codecForName("CP866");
    QTextCodec::setCodecForLocale(outcodec);
    #endif

    QVector<int> vector;
    for (int i = 0; i < iterations; ++i)
        vector.append(i); // Вектор: 0, 1, ..., 9

    QProgressDialog dialog; // Индикатор прогресса
    dialog.setLabelText(
        tr("Запущено %1 потоков")
        .arg(QThread::idealThreadCount() ) );

    // "Наблюдатель" за потоками:
    QFutureWatcher<void> futureWatcher;

    // Когда все вычисления закончатся:
    QObject::connect(
        &futureWatcher, SIGNAL(finished()),
        &dialog, SLOT(reset()));

    // Если пользователь нажмет кнопку Отмена:
    QObject::connect(
        &dialog, SIGNAL(canceled()),
        &futureWatcher, SLOT(cancel()));

    // При изменении "объема работ":
    QObject::connect(
```

```
        &futureWatcher, SIGNAL(progressRangeChanged(int, int)),
        &dialog, SLOT(setRange(int, int)));

    // По мере завершения очередной части работы:
    QObject::connect(
        &futureWatcher, SIGNAL(progressValueChanged(int)),
        &dialog, SLOT(setValue(int)));

    // Запускаем вычисления:
    futureWatcher.setFuture(QtConcurrent::map(vector, foo));

    dialog.exec();
    futureWatcher.waitForFinished(); // Ждем завершения

    qDebug() << tr("Отмена:") // Была ли нажата кнопка Отмена?
             << futureWatcher.future().isCanceled();
    return app.exec();
}
```

Листинг 16.13. Файл chap16/qfuture/qfuture.pro

```
TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .
win32: CONFIG += console
SOURCES += qfuture.cpp
```

Мы обещали не блокировать основной поток и сделали все, что требовалось для этого, но поскольку в нашем примере основному потоку просто нечего больше делать, кроме как ждать готовности всех вызовов, то нам пришлось прибегнуть к методу `QFutureWatcher::waitForFinished()`.

Замечание

Чтобы очередь сообщений не переполнилась, сигнал `progressValueChanged(int)` генерируется не постоянно, а при завершении некоторой части намеченных вычислений. API гарантирует, что сигнал о стопроцентном выполнении всей работы не будет пропущен.

Наконец, имеются и блокирующие варианты функций `map()` и `mapped()`, которые так и называются: `blockingMap()` и `blockingMapped()`. Выход из этих функций происходит только по окончании всех вычислений.

ЗАДАЧА. Выполнить масштабирование заданного набора изображений в блокирующем режиме.

Решение иллюстрирует рис. 16.4 и листинг 16.14.

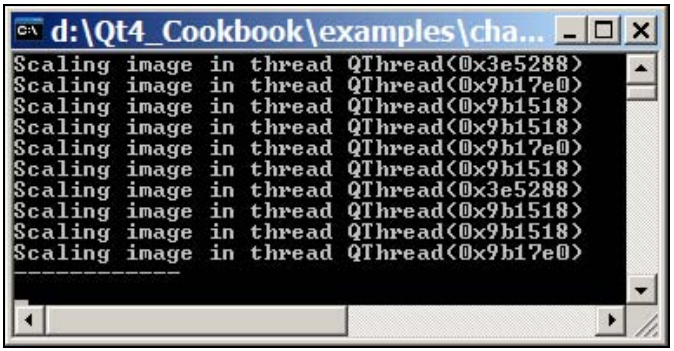


Рис. 16.4. Масштабирование набора изображений в отдельных потоках

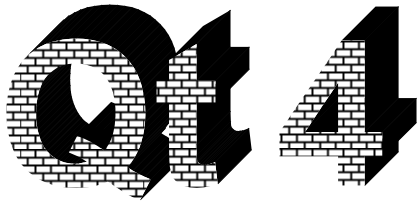
Листинг 16.14. Файл chap16/images/images.cpp

```
#include <QtGui>
#include <QDebug>

QImage scale(const QImage &image){
    qDebug() << "Scaling image in thread"
             << QThread::currentThread();
    return image.scaled(  // Масштабировать картинку
        QSize(100, 100),
        Qt::IgnoreAspectRatio,
        Qt::SmoothTransformation);
}

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

    const int imageCount = 10; // Количество картинок
    QList<QImage> images;
    for (int i = 0; i < imageCount; ++i)
        images.append(
           QImage(1600, 1200,
              QImage::Format_ARGB32_Premultiplied ) );
```

```
    QList<QImage> thumbnails = QtConcurrent::blockingMapped(
        images, scale );
    qDebug() << "------------";
    return app.exec();
}
```

Технология на основе `QtConcurrent` только начала разрабатываться, поэтому может претерпеть сильные изменения при выходе новых релизов библиотеки Qt.

ГЛАВА 17



Работа с базами данных

> Что пользы в том, что ты многое знал, раз ты не умел применять твои знания к твоим нуждам.
> *Франческо Петрарка*

В библиотеке Qt 4 имеются драйверы для работы со следующими СУБД:

- `QDB2` — IBM DB2;
- `QIBASE` — Borland InterBase;
- `QMYSQL` — MySQL;
- `QOCI` — Oracle;
- `QODBC` — ODBC (в том числе Microsoft SQL Server);
- `QPSQL` — PostgreSQL;
- `QSQLITE` — SQLite версии не ниже 3;
- `QSQLITE2` — SQLite версии 2;
- `QTDS` — Sybase Adaptive Server.

В *Qt Open Source Edition* отсутствует поддержка коммерческих СУБД Oracle, Sybase и DB2, т. к. драйверы для них распространяются под лицензией, не совместимой с GPL.

Для работы со всеми остальными СУБД приходится использовать ODBC или библиотеки для C/C++ от сторонних производителей.

17.1. Компиляция SQL-драйверов

ЗАДАЧА. Скомпилировать библиотеки для работы с СУБД.

Решение. При компиляции Qt 4 по умолчанию устанавливается только драйвер `QSQLITE` (если компилируются динамические библиотеки). Чтобы подключить поддержку остальных СУБД, нужно перед компиляцией библиотеки

Qt при запуске `configure` указать параметры `-qt-sql-mysql`, `-qt-sql-psql`, `-qt-sql-odbc`, `-qt-sql-ibase` для соответствующих статических библиотек или `-plugin-sql-mysql`, `-plugin-sql-psql`, `-plugin-sql-odbc`, `-plugin-sql-ibase` для динамических библиотек.

Компилятору и компоновщику потребуются заголовочные файлы и библиотеки, поставляемые вместе с соответствующими СУБД. Поэтому при запуске `configure` обычно требуется указать параметр `-Iпуть_к_h-файлам` и `-Lпуть_к_lib`. В результате компиляции в каталоге lib появится lib- или dll-библиотека для работы с соответствующей СУБД. Чтобы узнать, драйверы каких СУБД уже установлены, можно воспользоваться демонстрационной программой `demos/sqlbrowser`.

Если библиотека Qt уже скомпилирована, то можно просто зайти в каталог

src/plugins/sqldrivers/НужныйДрайвер

и выполнить сборку находящегося там проекта *.pro. Результат компиляции (динамические библиотеки) помещается в каталог plugins/sqldrivers.

ПРИМЕРЫ.

- для сборки драйвера ODBC в Windows:

```
cd %QTDIR%\src\plugins\sqldrivers\odbc
qmake -o Makefile odbc.pro
mingw32-make
```

  При использовании Visual C++ последняя команда, разумеется, `nmake` ;

- для сборки драйвера ODBC в Linux:

```
cd $QTDIR/src/plugins/sqldrivers/odbc
qmake "INCLUDEPATH+=/usr/local/unixODBC/include"  \
    "LIBS+=-L/usr/local/unixODBC/lib -lodbc"
make
```

- для сборки драйвера PostgreSQL в Windows:

```
cd %QTDIR%\src\plugins\sqldrivers\psql
qmake -o Makefile "INCLUDEPATH+=C:\psql\include"  \
      "LIBS+=C:\psql\lib\libpq.a" psql.pro
mingw32-make
```

  Все пути не должны содержать пробелов. Для компилятора Microsoft Visual C++ указывается библиотека C:\psql\lib\ms\libpq.lib;

- для сборки драйвера PostgreSQL в Linux:

```
cd $QTDIR/src/plugins/sqldrivers/psql
qmake -o Makefile "INCLUDEPATH+=/usr/include/pgsql"  \
```

```
        "LIBS+=-L/usr/lib -lpq" psql.pro
make
```

Замечание

Драйвер `QPSQL` может работать только с кодировкой `utf-8`.

- для сборки драйвера MySQL в Windows:

```
  cd %QTDIR%\src\plugins\sqldrivers\mysql
  qmake -o Makefile "INCLUDEPATH+=C:\MySQL\include"  \
        "LIBS+=C:\MySQL\lib\opt\libmysql.a" mysql.pro
mingw32-make
```

  Все пути не должны содержать пробелов. Для компиляции с помощью Microsoft Visual C++ вместо `libmysql.a` нужно указать `libmysql.lib`;

- для сборки драйвера MySQL в Linux:

```
 cd $QTDIR/src/plugins/sqldrivers/mysql
qmake -o Makefile "INCLUDEPATH+=/usr/local/include" \
   "LIBS+=-L/usr/local/lib -lmysqlclient_r" mysql.pro
make
```

Внимание!

При отдельной сборке плагинов следует использовать тот же компилятор, что и при сборке самой библиотеки Qt.

Обычно для Windows поставляются библиотеки *.lib, рассчитанные на применение компилятора Microsoft. Поскольку свободная версия Qt 4 для Windows до недавнего времени поддерживала только компилятор MinGW, то перед компиляцией SQL-драйвера придется сначала сгенерировать def-файл, например:

```
cd c:\mysql\lib\opt
reimp -d libmysql.lib
```

(утилита `reimp` поставляется вместе с MSYS), а затем сформировать библиотеку *.a:

```
dlltool -d libmysql.def -l libmysql.a libmysql.dll -k
```

Замечание

Если установлены SQL-драйверы, скомпилированные в виде динамических библиотек, то перед выполнением приложений требуется позаботиться о том, чтобы сами эти библиотеки, а также библиотеки, поставляемые с соответствующими СУБД, были доступны по стандартным путям поиска.

Если пути к библиотекам были добавлены в системную переменную PATH уже после того, как любое приложение Qt попыталось загрузить некоторый SQL-драйвер, то в Windows придется очистить ветку реестра

```
HKEY_CURRENT_USER\Software\Trolltech\OrganizationDefaults
```

В окончательных релизах, поставляемых пользователю, Qt-драйверы qsql* лучше всего размещать в подкаталоге ./sqldrivers, а используемые ими библиотеки — либо в каталоге самой программы, либо в стандартном каталоге Windows\system32 (для системы Windows) или usr/local/lib (для Linux).

17.2. Подключение к базе данных и выполнение SQL-запросов

Для подключения к базе данных нужно указать название SQL-драйвера, например:

```
QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL", "MyDB1");
```

Второй необязательный параметр позволяет задать имя соединения.

Затем указывают имя сервера, название базы данных, имя пользователя и пароль:

```
db.setHostName("localhost"); // или, например, "my1.server.ru"
db.setDatabaseName("mydb1");
db.setUserName("root");
db.setPassword("mypassword");
```

Если сервер задействует нестандартный порт, то придется задать и его:

```
db.setPort(НомерПорта);
```

При использовании QODBC имя сервера не требуется, а вместо названия базы данных указывают ODBC-псевдоним (алиас).

SQLite не поддерживает авторизацию пользователей, поэтому ему требуется указать только имя файла данных. Предопределенное имя ":memory:" позволяет размещать временную базу данных в оперативной памяти.

После того как все параметры подключения заданы, можно открыть соединение:

```
bool connected = db.open();
```

Если подключение установить не удалось, то неплохо бы узнать описание ошибки и сообщить его пользователю (листинг 17.1).

Листинг 17.1

```
if (!connected) {
   QMessageBox::critical( // Диалог с сообщением об ошибке
       parent,                      // Родительский виджет
       QObject::tr("Database Error"),   // Заголовок
       db.lastError().text());          // Текст сообщения
   return false; // Вернуть признак неудачного подключения
}
```

Если подключение установлено, то можно выполнить любой SQL-запрос, например:

```
QSqlQuery query;
query.exec("SELECT id, name, salary FROM empl WHERE salary>=1000");
```

или

```
QSqlQuery query(
    "SELECT id, name, salary FROM empl WHERE salary>=1000");
```

Здесь запрашиваются номера `id`, имена `name` и оклады `salary` всех работников из таблицы `empl`, у которых оклад не ниже 1000.

Внимание!

Если при создании объекта `QSqlQuery` указан текст запроса на языке SQL, то этот запрос сразу выполняется.

В обоих случаях в конструкторе запроса можно указать базу данных `QSqlDatabase`, с которой он будет работать. Но, как правило, приложение открывает только одно соединение с единственной базой данных, поэтому этот параметр принимает значение по умолчанию.

Замечание

Объект соединения с базой данных нельзя передать в другой поток (и тем более, процесс), поэтому при запуске нового потока/процесса приходится открывать еще одно соединение.

Если при выполнении запроса возникла ошибка, то метод `lastError()` позволяет вывести на экран ее описание:

```
if ( ! query.isActive() )
   QMessageBox::warning( this, tr("Database Error"),
                         query.lastError().text() );
```

Иначе можно получить данные, которые сервер вернул в качестве результата (листинг 17.2).

Листинг 17.2

```
while ( query.next() ) {
   qint64 id = query.value(0).toLongLong();
   QString name = query.value(1).toString();
   double salary = query.value(2).toDouble();
   // .......
}
```

Метод `QSqlQuery::next()` переводит курсор на очередную запись результирующего набора данных или возвращает `false`, если достигнут его конец. Метод `value(номер_столбца)` возвращает значение типа `QVariant`, которое необходимо преобразовать к нужному типу с помощью методов `QVariant::toInt`, `QVariant::toLongLong`, `QVariant::toString`, `QVariant::toDouble`, `QVariant::toDate`, `QVariant::toDateTime` и т. д.

Кроме `next()`, для навигации по набору данных предназначены методы `first()`, `last()`, `previous()`, `seek(int index, bool relative=false)`. Для увеличения быстродействия набор данных лучше сделать однонаправленным, вызвав метод `QSqlQuery::setForwardOnly(true)` до выполнения запроса, после этого можно использовать только `next()`.

Метод `QSqlQuery::size()` возвращает число записей, полученных в результате выполнения запроса `SELECT` (–1, если была ошибка или если драйвер данной СУБД не поддерживает эту функцию). При выполнении SQL-запросов `INSERT`, `UPDATE` или `DELETE` метод `size()` следует заменить на `QSqlQuery::numRowsAffected()`. Метод `QSqlQuery::isSelect()` возвращает `true`, если запрос является запросом `SELECT`. Метод `QSqlQuery::lastError()` позволяет узнать, возникла ли ошибка при последнем выполнении запроса. Аналогичный метод имеет и класс `QSqlDatabase`. В обоих случаях возвращается экземпляр класса `QSqlError`. Тип ошибки можно выяснить, вызвав метод `QSqlError::type()`. Возможные типы ошибок: `QSqlError::NoError` (ошибок не было), `QSqlError::ConnectionError` (ошибка соединения), `QSqlError::StatementError` (синтаксическая ошибка в SQL-запросе), `QSqlError::TransactionError` (ошибка транзакции) и `QSqlError::UnknownError` (неизвестная ошибка).

Если требуется выполнить большое количество однотипных SQL-операторов, то эффективнее указать запрос с поименованными (листинг 17.3) или безымянными параметрами (листинг 17.4).

Листинг 17.3

```
QSqlQuery query;
query.prepare("INSERT INTO empl (id, name, salary) "
              "VALUES (:id, :name, :salary)");
```

```
for (int i=0; i<N; i++) {
   query.bindValue(":id", arr_id[i]);
   query.bindValue(":name", arr_name[i]);
   query.bindValue(":salary", arr_salary[i]);
   query.exec();
}
```

Листинг 17.4

```
 QSqlQuery query;
query.prepare("INSERT INTO empl (id, name, salary) "
       "VALUES (?, ?, ?)");
for (int i=0; i<N; i++) {
  query.addBindValue(arr_id[i]);
  query.addBindValue(arr_name[i]);
  query.addBindValue(arr_salary[i]);
  query.exec();
}
```

Здесь выполняется вставка `N` записей, данные берутся из массивов `arr_id`, `arr_name` и `arr_salary`.

Если СУБД поддерживает механизм транзакций, то для начала новой транзакции используется метод

```
bool QSqlDatabase::transaction()
```

для ее подтверждения нужно вызвать метод

```
bool QSqlDatabase::commit()
```

а для отмены:

```
bool QSqlDatabase::rollback()
```

Если СУБД не поддерживает транзакций, то вызовы `transaction`, `commit` и `rollback` ничего не делают. С помощью метода `QSqlDriver::hasFeature()` можно узнать, поддерживается ли данным драйвером и СУБД та или иная функция, в том числе и транзакции:

```
QSqlDriver *driver = QSqlDatabase::database().driver();
if (driver->hasFeature(QSqlDriver::Transactions))
  .......
```

Каждое соединение с базой данных в любой момент времени может иметь только одну активную транзакцию. Если этого недостаточно, всегда можно открыть еще несколько соединений с той же базой данных.

Ссылку на соединение с базой данных можно получить, вызвав функцию `QSqlDatabase::database(connectionName)`. Необязательный параметр

connectionName — это имя соединения, которое было задано при его создании с помощью QSqlDatabase::addDatabase().

По окончании работы с базой данных соединение нужно закрыть: QSqlDatabase::close(). Затем можно либо открыть его снова с помощью метода open(), либо удалить из списка соединений, вызвав статический метод QSqlDatabase::removeDatabase(connectionName).

ЗАДАЧА. Подключиться к базе данных; создать таблицу employee для хранения сведений о табельном номере, ФИО, дате рождения, окладе и семейном положении работников предприятия; вставить в таблицу три записи и для контроля прочитать их и вывести результат на экран, после чего отключиться от базы данных.

Решение. Текст программы приведен в листингах 17.5 и 17.6, а результат ее работы показан на рис. 17.1.

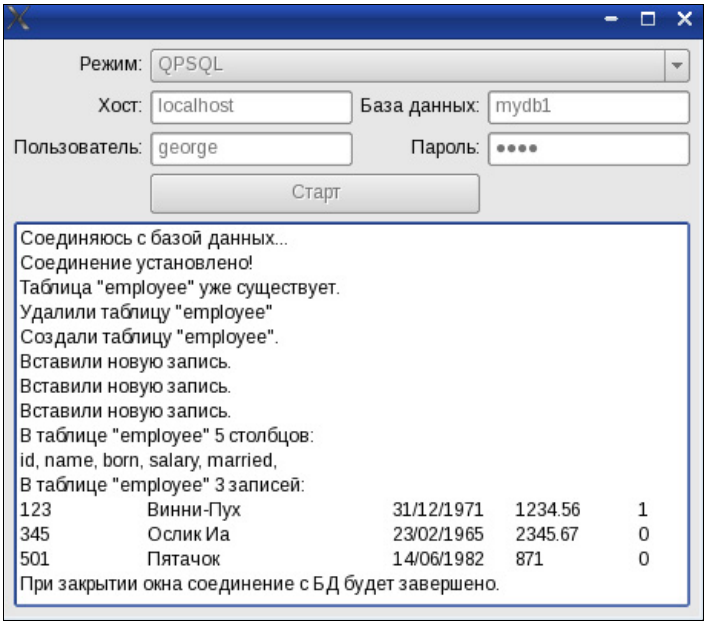


Рис. 17.1. Выполнение SQL-запросов

Листинг 17.5. Файл chap17/db00/db00.h

```
1 #include <QtGui>
2
3 class MyDialog : public QDialog {
4     Q_OBJECT
```

```
 5 public:
 6     MyDialog(QWidget *parent=0);
 7 protected:
 8     virtual void closeEvent(QCloseEvent *event);
 9 private slots:
10     bool start();
11 private:
12     QComboBox *mode;   // Режим (драйвер)
13     QLineEdit *host;   // Хост
14     QLineEdit *dbname; // Имя БД
15     QLineEdit *user;   // Пользователь
16     QLineEdit *password; // Пароль
17     QTextEdit *scr;      // Для вывода сообщений
18     QPushButton *btnStart; // Кнопка 'Старт'
19 ;
```

Листинг 17.6. Файл chap17/db00/db00.cpp

```
 1 #include <QtGui>
 2 #include <QtSql>
 3 #include <QtTest/QtTest> // Для qWait().
 4
 5 #include "db00.h"
 6
 7 MyDialog::MyDialog(QWidget *parent)
 8         : QDialog(parent) {
 9     QTextCodec *codec = QTextCodec::codecForName("CP1251");
10     QTextCodec::setCodecForTr(codec);
11     QTextCodec::setCodecForCStrings(codec);
12     QTextCodec::setCodecForLocale(codec);
13
14     setWindowFlags(Qt::Window);
15
16     mode = new QComboBox(this);
17     QStringList drivers = QSqlDatabase::drivers();
18     drivers.removeAll("QMYSQL3");
19     drivers.removeAll("QOCI8");
20     drivers.removeAll("QODBC3");
21     drivers.removeAll("QPSQL7");
22     drivers.removeAll("QTDS7");
23     mode->addItems(drivers);
24
```

```
25      host = new QLineEdit(tr("localhost"), this);
26      dbname = new QLineEdit(this);
27      user = new QLineEdit(this);
28      password = new QLineEdit(this);
29      password->setEchoMode(QLineEdit::Password);
30
31      btnStart = new QPushButton(tr("Старт"), this);
32
33      scr = new QTextEdit(this);
34      scr->setReadOnly(true);
35
36      QGridLayout *layout = new QGridLayout(this);
37      layout->addWidget(new QLabel(tr("Режим:"), this),
38                        0, 0, Qt::AlignRight);
39      layout->addWidget(mode, 0, 1, 1, 3);
40
41      layout->addWidget(new QLabel(tr("Хост:"), this),
42                        1, 0, Qt::AlignRight);
43      layout->addWidget(host, 1, 1);
44
45      layout->addWidget(new QLabel(tr("База данных:"), this),
46                        1, 2, Qt::AlignRight);
47      layout->addWidget(dbname, 1, 3);
48
49      layout->addWidget(new QLabel(tr("Пользователь:"), this),
50                        2, 0, Qt::AlignRight);
51      layout->addWidget(user, 2, 1);
52
53      layout->addWidget(new QLabel(tr("Пароль:"), this),
54                        2, 2, Qt::AlignRight);
55      layout->addWidget(password, 2, 3);
56
57      layout->addWidget(btnStart, 3, 1, 1, 2);
58      layout->addWidget(scr, 4, 0, 1, 4);
59
60      layout->setMargin(6);
61      layout->setSpacing(5);
62      layout->setColumnStretch(1, 1);
63      layout->setColumnStretch(3, 1);
64      layout->setRowStretch(4, 1);
65      setLayout(layout);
66
67      connect(btnStart, SIGNAL(clicked()), this, SLOT(start()));
68  }
```

```
 69
 70 bool MyDialog::start() {
 71     scr->append(tr("Соединяюсь с базой данных..."));
 72     QSqlDatabase db = QSqlDatabase::addDatabase(
 73         mode->currentText() );
 74     db.setHostName(host->text());
 75     db.setDatabaseName(dbname->text());
 76     db.setUserName(user->text());
 77     db.setPassword(password->text());
 78     if (db.open()) {
 79         mode->setEnabled(false);
 80         host->setEnabled(false);
 81         dbname->setEnabled(false);
 82         user->setEnabled(false);
 83         password->setEnabled(false);
 84         btnStart->setEnabled(false);
 85         scr->append(tr("Соединение установлено!"));
 86     }else{
 87         scr->append(tr("Не могу соединиться: "));
 88         scr->append(db.lastError().text());
 89         return false;
 90     }
 91
 92     QSqlQuery sql = QSqlQuery();
 93     //sql.exec(tr("SET NAMES 'cp1251'"));
 94     QStringList dbtables = db.tables(QSql::Tables);
 95     if (dbtables.contains( tr("employee"),
 96                            Qt::CaseInsensitive)) {
 97         scr->append( tr(
 98                 "Таблица \"employee\" уже существует."));
 99         sql.exec(tr("DROP TABLE employee"));
100         if ( sql.lastError().type() == QSqlError::NoError ) {
101             scr->append( tr(
102                 "Удалили таблицу \"employee\" "));
103         }else{
104             scr->append( tr(
105                 "Не могу удалить таблицу \"employee\":"));
106             scr->append(sql.lastError().text());
107             return false;
108         }
109     }
110
```

```
111     sql.exec( tr(
112         "create table employee ( "
113         "  id integer PRIMARY KEY, "
114         "  name char(30) not null, "
115         "  born date null, "
116         "  salary numeric(12,2), "
117         "  married boolean NULL ) " ) );
118     if ( sql.lastError().type() == QSqlError::NoError ) {
119         scr->append( tr(
120             "Создали таблицу \"employee\"."));
121     }else{
122         scr->append( tr(
123                 "Не могу создать таблицу \"employee\":"));
124         scr->append(sql.lastError().text());
125         return false;
126     }
127
128     if (sql.prepare( tr(
129             "INSERT INTO employee "
130             "  VALUES (?, ?, ?, ?, ?)") ) ) {
131         int arr_id[] = {123, 345, 501};
132         QString arr_name[] = {tr("Винни-Пух"),
133                               tr("Ослик Иа"),
134                               tr("Пятачок")};
135         QDate arr_born[] = {QDate(1971, 12, 31),
136                             QDate(1965, 2, 23),
137                             QDate(1982, 6, 14)};
138         float arr_salary[] = {1234.56f, 2345.67f, 871};
139         int arr_married[] = {1, 0, 0};
140
141         for (unsigned int i=0; i < 3; i++) {
142             sql.bindValue(0, arr_id[i]);
143             sql.bindValue(1, arr_name[i]);
144             sql.bindValue(2, arr_born[i]);
145             sql.bindValue(3, arr_salary[i]);
146             sql.bindValue(4, arr_married[i]);
147             sql.exec();
148             if ( sql.lastError().type() == QSqlError::NoError ) {
149                 scr->append( tr(
150                     "Вставили новую запись."));
151             }else{
152                 scr->append( tr(
153                     "Не могу вставить новую запись:"));
```

```
154                 scr->append(sql.lastError().text());
155                 return false;
156             }
157         }
158     }else{
159         scr->append( tr(
160                 "Не могу подготовить запрос:"));
161         scr->append(sql.lastError().text());
162         return false;
163     }
164
165     sql.exec( tr("SELECT * FROM employee ") );
166     if ( sql.isActive() ) {
167         QSqlRecord rec = sql.record();
168         scr->append( tr(
169             "В таблице \"employee\" %1 столбцов: ")
170             .arg(rec.count() ) );
171
172         QString fields;
173         for(int j=0; j<rec.count(); j++)
174             fields += rec.fieldName(j) + ", ";
175
176         scr->append(fields);
177
178         scr->append( tr(
179             "В таблице \"employee\" %1 записей: ")
180             .arg(sql.size() ) );
181
182         while ( sql.next() ) {
183             int id = sql.value(0).toInt();
184             QString name = sql.value(1).toString();
185             QDate born = sql.value(2).toDate();
186             double salary = sql.value(3).toDouble();
187             bool married = sql.value(4).toBool();
188             scr->append( tr(
189                 "%1\t %2\t %3\t %4\t %5")
190                 .arg(id)
191                 .arg(name)
192                 .arg(born.toString(tr("dd/MM/yyyy")))
193                 .arg(salary)
194                 .arg(married) );
195         }
```

```
196     }else{
197         scr->append( tr(
198                 "Не могу получить данные:"));
199         scr->append(sql.lastError().text());
200         return false;
201     }
202
203     scr->append( tr(
204         "При закрытии окна соединение с БД будет завершено."));
205     return true;
206 }
207
208 void MyDialog::closeEvent(QCloseEvent *event) {
209     QSqlDatabase db = QSqlDatabase::database();
210     if (db.isOpen()) {
211         db.close();
212         scr->append("--------------------------");
213         scr->append(tr("Соединение с базой данных закрыто!"));
214         QTest::qWait(1000);    // Ждать 1 сек.
215     }
216 }
217
218 int main(int argc, char *argv[]) {
219     QApplication app(argc, argv);
220
221     MyDialog mainWin;
222     mainWin.show();
223     return app.exec();
224 }
```

Пояснения к программе:

- (14) — установили для диалога флаг `Qt::Windows`, чтобы в заголовке окна появились кнопки сворачивания и восстановления;
- (16–23) — создали поле со списком и заполнили его названиями всех доступных драйверов SQL, кроме устаревших (17–21);
- (25–29) — создали однострочные поля для ввода параметров соединения;
- (31) — кнопка **Старт**;
- (33, 34) — многострочное поле для вывода сообщений;
- (36–65) — расположили все элементы с помощью сеточного менеджера размещения;
- (67) — связали кнопку **Старт** с функцией `start()`;

- (69) — функция, выполняемая при нажатии на кнопку;
- (71) — вывели сообщение;
- (72–78) — создали соединение с базой данных;
- (79–84) — если соединение установлено, то менять параметры уже нельзя;
- (93) — при необходимости сообщили серверу кодовую таблицу клиента;
- (94, 95) — выяснили, существует ли в базе данных таблица `employee` (работники);
- (99) — если да, то удалили ее;
- (111) — создали новую таблицу;
- (128–162) — выполнили серию SQL-запросов с параметрами для вставки новых записей в созданную таблицу;
- (165) — получили все записи таблицы;
- (166–174) — получили информацию о полях (столбцах);
- (178–194) — вывели на экран все записи (строки) таблицы;
- (208) — виртуальная функция `closeEvent()` выполняется при закрытии окна;
- (209–211) — если соединение с базой данных открыто, то закрываем его;
- (212–214) — вывели сообщение и подождали секунду перед закрытием окна.

Перед компиляцией проекта, в котором используется модуль `QtSql`, нужно добавить в pro-файл строку

```
QT += sql
```

В нашей программе задержка перед закрытием окна формируется с помощью модуля `QtTest`, поэтому в файл проекта *.pro следует добавить строку

```
CONFIG += qtestlib
```

В результате, кроме GUI-интерфейса при запуске программы, мы увидим также консольное окно, в которое будут выводиться отладочные сообщения. Даже если вывод отладочных сообщений в нашей программе не предусмотрен (функции `qDebug()` нет), все равно некоторая дополнительная информация может генерироваться служебными модулями самой библиотеки Qt.

Перед выполнением программы требуется создать какую-нибудь базу данных:

```
create database db1 character set utf8;
```

Кодировку `utf8` можно заменить на `cp1251` или `koi8r`. Для PostgreSQL кодировка может быть только `utf8`. Если мы хотим использовать драйвер `QODBC`, то нужно будет настроить ODBC-псевдоним.

17.3. Работа с таблицами баз данных

Самый простой способ отображения информации базы данных в виде таблицы реализуют классы `QSqlQueryModel` и `QTableView` (листинг 17.7).

Листинг 17.7

```
QSqlQueryModel model;
model.setQuery("select * from employee");
QTableView view;
view.setModel(&model);
view.show();
```

Но вместо `QsqlQueryModel` можно применить класс `QSqlTableModel`, позволяющий работать с таблицами баз данных на более высоком уровне, чем выполнение SQL-запросов (листинг 17.8).

Листинг 17.8

```
QSqlTableModel model;
model.setTable("employee");
model.select();
QTableView view;
view.setModel(&model);
view.show();
```

Более сложный пример: вместо выполнения SQL-запроса

```
SELECT * FROM employee WHERE salary >= 1000 ORDER BY id DESC
```

для модели `QSqlTableModel` нужно задать фильтр и условие сортировки (листинг 17.9).

Листинг 17.9

```
QSqlTableModel model = QSqlTableModel(parent, db);
model.setTable("employee");           // Имя таблицы базы данных
model.setFilter("salary >= 1000");    // Условие WHERE
model.setSort(0, Qt::DescendingOrder); // Сортировка по убыванию id
model.select();                       // Получить данные
```

После определения модели можно узнать значение любого поля любой записи, обратившись к нему по номеру (нумерация начинается с нуля) или по имени, например:

```
QString name = model.record(i).value("name").toString();
```

или

```
int salary = model.data(model.index(i, 3)).toInt();
```

Перебор всех записей набора данных иллюстрирует листинг 17.10.

Листинг 17.10

```
for (int i = 0; i < model.rowCount(); ++i) {
  QSqlRecord record = model.record(i);
  QString name = record.value("name").toString();
  double salary = record.value("salary").toDouble();
  .......
}
```

ЗАДАЧА. Вставить в таблицу новую запись.

Решение показано в листинге 17.11.

Листинг 17.11

```
QSqlTableModel model;
model.setTable("employee");
int row = 0;
model.insertRows(row, 1);
model.setData(model.index(row, 0), 567);
model.setData(model.index(row, 1), tr("Иванов П.С."));
model.setData(model.index(row, 2), QDate(1987, 12, 31));
model.setData(model.index(row, 3), 3456.78);
model.setData(model.index(row, 4), false);
if ( ! model.submitAll() )
   // ошибка при вставке новой записи в таблицу
```

ЗАДАЧА. Изменить значения полей имеющейся записи.

Решение иллюстрируют листинги 17.12 и 17.13.

Листинг 17.12. Первый вариант

```
model.setFilter("id = 567");
model.select();
if (model.rowCount() == 1) {
   QSqlRecord record = model.record(0);
   record.setValue("name", tr("Петров А.С."));
```

```
    record.setValue("salary", record.value("salary").toFloat() + 100);
    model.setRecord(0, record);
    model.submitAll();
}
```

Листинг 17.13. Второй вариант

```
model.setFilter("id = 567");
model.select();
if (model.rowCount() == 1) {
    model.setData(model.index(0, 1), tr("Петров А.С."));
    model.setData(model.index(0, 3),
                  model.data(model.index(0, 3)).toFloat() + 100);
    model.submitAll();
}
```

ЗАДАЧА. Удалить запись из таблицы.

Решение показано в листинге 17.14.

Листинг 17.14

```
model.setTable("employee");
model.setFilter("id = 567");
model.select();
if (model.rowCount() == 1) {
    model.removeRows(0, 1);
    model.submitAll();
}
```

ЗАДАЧА. Удалить из таблицы группу записей, удовлетворяющих заданному критерию.

Решение иллюстрирует листинг 17.15.

Листинг 17.15

```
model.setTable("employee");
model.setFilter("salary < 1000");
model.select();
if (model.rowCount() > 0) {
    model.removeRows(0, model.rowCount());
    model.submitAll();
}
```

ЗАДАЧА. Отобразить в виде таблицы данные, сформированные программой из *разд. 17.2*.

Решение. Листинг 17.16 иллюстрирует пример работы с таблицей сотрудников, созданной в результате выполнения программы, приведенной в листингах 17.5 и 17.6, а на рис. 17.2 показан внешний вид таблицы в системе Windows.

Рис. 17.2. Таблица базы данных

Листинг 17.16. Файл chap17/db01/db01.cpp

```
#include <QtSql>

#define SQLDRIVER "QPSQL"
#define HOST "localhost"
#define DBNAME "mydb1"
#define USER "имя_пользователя"
#define PASSWORD "мой_пароль"

int main(int argc, char *argv[]) {

   QApplication app(argc, argv);
   QTextCodec *codec = QTextCodec::codecForName("CP1251");
   QTextCodec::setCodecForTr(codec);
   QTextCodec::setCodecForCStrings(codec);
   QTextCodec::setCodecForLocale(codec);

   QSqlDatabase db = QSqlDatabase::addDatabase(SQLDRIVER);
   if ( ! db.isDriverAvailable(SQLDRIVER) ) {
      QStringList drs = db.drivers();
      QString s;
      for (int i=0; i<drs.size(); ++i)
         s += drs[i] + " ";

```

```
24        QMessageBox::critical(
25            0,      // parent
26            QObject::tr("Database Error"),
27            QObject::tr("Driver %1 is not available!\n"
28                        "Use %2").arg(SQLDRIVER)
29                                 .arg(s)
30        return 1;
31    }
32    db.setHostName(HOST);
33    db.setDatabaseName(DBNAME);
34    db.setUserName(USER);
35    db.setPassword(PASSWORD);
36    if ( ! db.open() ) {
37        QMessageBox::critical(
38            0,      // parent
39            QObject::tr("Open Database Error"),
40            db.lastError().text() );
41        return 2;
42    }
43
44    QSqlQuery q;
45    // Для корректного отображения символов кириллицы,
46    // возможно, придется установить кодировку:
47    // q.exec(QObject::tr("SET NAMES 'cp1251'"));
48
49    QSqlTableModel model;
50    model.setTable("employee");
51
52    model.insertRows(0, 1);
53    model.setData(model.index(0, 0), 159);
54    model.setData(model.index(0, 1), QObject::tr("Сова"));
55    model.setData(model.index(0, 2), QDate(1985, 12, 31));
56    model.setData(model.index(0, 3), 12.34);
57    model.setData(model.index(0, 4), 1);
58    model.submitAll();
59
60    model.setEditStrategy(QSqlTableModel::OnFieldChange);
61
62    model.select();
63    model.setHeaderData(0, Qt::Horizontal,
64                        QObject::tr("Табельн.\nномер"));
65    model.setHeaderData(1, Qt::Horizontal,
66                        QObject::tr("Имя"));
```

```
67      model.setHeaderData(2, Qt::Horizontal,
68                          QObject::tr("День рождения"));
69      model.setHeaderData(3, Qt::Horizontal,
70                          QObject::tr("Зарплата"));
71      model.setHeaderData(4, Qt::Horizontal,
72                          QObject::tr("Женат/\nзамужем"));
73
74      QTableView view;
75      view.setModel(&model);
76
77      view.setAlternatingRowColors(true);
78      view.resizeRowsToContents();
79      view.resizeColumnsToContents();
80      view.show();
81
82      return app.exec();
83  }
```

Пояснения к программе:

- (3–7) — параметры соединения с базой данных;
- (17–36) — установили параметры соединения и подключились к базе данных. Если указанный драйвер недоступен, то сообщили пользователю, какие драйверы можно использовать;
- (45–47) — при необходимости сообщили серверу кодировку строковых данных, которую задействует наша программа (для MySQL);
- (49, 50) — создали модель `QSqlTableModel` и задали имя таблицы БД;
- (52–58) — добавили к таблице новую строку, задали значения всех пяти ячеек этой строки и подтвердили запись изменений в базу данных (диагностика возможных ошибок для простоты опущена);
- (60) — задали режим обновления данных в БД при редактировании таблицы: параметр `QSqlTableModel::OnFieldChange` означает, что изменения в базу данных будут записываться автоматически после окончания редактирования очередной ячейки. Другие возможные режимы: `QSqlTableModel::OnRowChange` (при переходе к другой строке) и `QSqlTableModel::OnManualSubmit` (при выполнении метода `submitAll()`, подтверждающего все изменения, или `revertAll()`, отменяющего их);
- (62) — выбрали данные из таблицы БД;
- (63–72) — указали заголовки столбцов таблицы;
- (74, 75) — создали представление таблицы и задали для него модель;

- (77) — установили чередующийся цвет фона для строк таблицы;
- (78, 79) — выполнили автоподстройку высоты строк и ширины столбцов;
- (80) — показали таблицу на экране.

Проверьте, как работает эта программа: можно изменять данные в ячейках; при редактировании чисел и дат автоматически отображаются кнопки инкремента/декремента (в зависимости от версии библиотеки и типа СУБД для редактирования чисел может использоваться либо элемент `(Double)SpinBox`, либо обычное текстовое поле). Но поскольку для элемента `DoubleSpinBox` по умолчанию задано максимальное значение `99.99`, то при попытке изменить, например, величину зарплаты работника, любое значение, большее этого максимального, автоматически усекается. Разумеется, попытка ввести число больше допустимого, оканчивается неудачей. Кроме того, после редактирования первой же ячейки таблицы, когда происходит автоматическое обновление данных, ширина столбцов и высота строк изменяется, т. к. размеры ячеек по умолчанию отличаются от тех, что установились в результате однократного выполнения методов `resizeRowsToContents()` и `resizeColumnsToContents()`. Для некоторых драйверов могут наблюдаться и другие побочные эффекты. Например, при использовании `QPSQL` отредактированная строка автоматически перемещается в конец таблицы, если мы явно не укажем способ сортировки записей с помощью метода

```
QSqlTableModel::setSort ( int column, Qt::SortOrder order )
```

Таким образом, если нужно начислять большую зарплату, выводить разные столбцы различным цветом, отображать галочки для полей логического типа, вводить даты с помощью календарика, в общем, как-то изменять заданные по умолчанию параметры отображения и редактирования ячеек, то у нас имеются две возможности:

- разрабатывать свой класс модели и/или представления таблицы;
- задействовать специальные объекты-делегаты для ее ячеек.

В *главе 18* мы рассмотрим оба варианта по очереди.

ГЛАВА 18

Модели таблиц баз данных и их представления

Люди редко совершенствуются, если у них нет другой модели для подражания, кроме самих себя.
Оливер Голдсмит

18.1. Разработка модели и представления таблицы БД

ЗАДАЧА. Изменить содержание и внешний вид данных, отображаемых с помощью `QTableView`. В ячейках последнего столбца таблицы, где хранится только два возможных значения, нужно отображать элемент `QCheckBox` и текст **"Да"** или **"Нет"** (рис. 18.1). Кроме того, необходимо запретить редактирование первого столбца, изменить цвет фона ячеек первого и последнего столбцов, а также параметры шрифта во втором столбце.

	Табельн. номер	Имя	День рождения	Зарплата	Женат/ замужем
1	№1	Ослик Иа	02.05.1992	87.23	☐ Нет
2	№345	Винни-Пух	18.06.1997	22.56	☑ Да
3	№501	Пятачок	14.06.1982	11.00	☑ Да
4	№159	Сова	31.12.1985	56.10	☐ Нет

Рис. 18.1. Пример форматирования ячеек таблицы

Решение. Для этого определим свою модель таблицы, выбрав в качестве базового класс `QSqlQueryModel`. А чтобы управлять размерами ячеек таблицы, определим свой класс представления на основе стандартного `QTableView`. В листингах 18.1 и 18.2 приведен текст программы.

Листинг 18.1. Файл chap18/db02/db02.h

```
#include <QSqlQueryModel>
#include <QTableView>

class MyModel : public QSqlQueryModel {
    Q_OBJECT
public:
    MyModel(QObject *parent = 0);
    Qt::ItemFlags flags(const QModelIndex &index) const;
    QVariant data(const QModelIndex &index,
                  int role = Qt::DisplayRole) const;
    bool setData(const QModelIndex &index,
                 const QVariant &value, int role);
private:
    void refresh();
};

class MyView : public QTableView {
    Q_OBJECT
public:
    MyView(QWidget *parent = 0);
private:
    virtual void resizeEvent(QResizeEvent *event);
};
```

Листинг 18.2. Файл chap18/db02/db02.cpp

```
// Таблица базы данных: пользовательская модель и представление

#include <QtGui>
#include <QtSql>

#include "db02.h"

MyModel::MyModel(QObject *parent)
                : QSqlQueryModel(parent) {
    refresh();
}

Qt::ItemFlags MyModel::flags(
        const QModelIndex &index) const {

```

```
16      Qt::ItemFlags flags = QSqlQueryModel::flags(index);
17      if (index.column() >= 1 && index.column() < 4)
18          flags |= Qt::ItemIsEditable;
19      if (index.column() == 4)
20          flags |= Qt::ItemIsUserCheckable;
21      return flags;
22  }
23
24  QVariant MyModel::data(
25              const QModelIndex &index,
26              int role) const {
27
28      QVariant value = QSqlQueryModel::data(index, role);
29
30      switch (role) {
31
32      case Qt::DisplayRole: // Данные для отображения
33      case Qt::EditRole:    // Данные для редактирования
34          if (index.column() == 0)
35              return value.toString().prepend(tr("№"));
36          else if (index.column() == 2 && role == Qt::DisplayRole)
37              return value.toDate().toString("dd.MM.yyyy");
38          else if (index.column() == 3 && role == Qt::DisplayRole)
39              return tr("%1")
40                     .arg(value.toDouble(), 0, 'f', 2);
41          else if (index.column() == 4)
42              return value.toInt() != 0 ? tr("Да") : tr("Нет");
43          else
44              return value;
45
46      case Qt::TextColorRole: // Цвет текста
47          if(index.column() == 1)
48              return qVariantFromValue(QColor(Qt::blue));
49          else
50              return value;
51
52      case Qt::TextAlignmentRole: // Выравнивание
53          if(index.column() == 3)
54              return int(Qt::AlignRight | Qt::AlignVCenter);
55          else if(index.column() == 2 || index.column() == 4)
56              return int(Qt::AlignHCenter | Qt::AlignVCenter);
57          else
58              return int(Qt::AlignLeft | Qt::AlignVCenter);
59
```

```
60      case Qt::FontRole: // Шрифт
61          if(index.column() == 1) {
62              QFont font = QFont("Helvetica", 10, QFont::Bold);
63              return qVariantFromValue(font);
64          }else
65              return value;
66
67      case Qt::BackgroundColorRole: {  // Цвет фона
68          int a = (index.row() % 2) ? 14 : 0;
69          if(index.column() == 0)
70              return qVariantFromValue(QColor(220,240-a,230-a));
71          else if(index.column() == 4)
72              return qVariantFromValue(QColor(200,220-a,255-a));
73          else
74              return value;
75      }
76      case Qt::CheckStateRole:  // Галочка
77          if (index.column() == 4)
78              return (QSqlQueryModel::data(index).toBool()) ?
79                      Qt::Checked : Qt::Unchecked;
80          else
81              return value;
82
83      case Qt::SizeHintRole:  // Размер ячейки
84          if (index.column() == 0)
85              return QSize(70, 10);
86          if (index.column() == 4)
87              return QSize(60, 10);
88          else
89              return QSize(110, 10);
90      }
91      return value;
92  }
93
94  bool MyModel::setData(
95              const QModelIndex &index,
96              const QVariant &value,
97              int /* role */) {
98      if (index.column() < 1 || index.column() > 4)
99          return false;
100
101     QModelIndex primaryKeyIndex = QSqlQueryModel::index(
102                 index.row(), 0);
```

```
103     int id = QSqlQueryModel::data(primaryKeyIndex).toInt();
104
105     //clear(); // Если надо полностью перерисовать таблицу.
106
107     bool ok;
108     QSqlQuery query;
109     if (index.column() == 1) {
110         query.prepare("update employee set name = ? where id = ?");
111         query.addBindValue(value.toString());
112         query.addBindValue(id);
113     }else if(index.column() == 2) {
114         query.prepare("update employee set born = ? where id = ?");
115         query.addBindValue(value.toDate());
116         query.addBindValue(id);
117     }else if(index.column() == 3) {
118         query.prepare(
              "update employee set salary = ? where id = ?");
119         query.addBindValue(value.toDouble());
120         query.addBindValue(id);
121     }else if(index.column() == 4) {
122         query.prepare(
              "update employee set married = ? where id = ?");
123         query.addBindValue(value.toBool());
124         query.addBindValue(id);
125     }
126     ok = query.exec();
127     refresh();
128     return ok;
129 }
130
131 void MyModel::refresh() {
132     setQuery("select * from employee ORDER BY id");
133
134     setHeaderData(0, Qt::Horizontal,
135                   tr("Табельн.\nномер"));
136     setHeaderData(1, Qt::Horizontal,
137                   tr("Имя"));
138     setHeaderData(2, Qt::Horizontal,
139                   tr("День рождения"));
140     setHeaderData(3, Qt::Horizontal,
141                   tr("Зарплата"));
142     setHeaderData(4, Qt::Horizontal,
143                   tr("Женат/\nзамужем"));
144 }
145
```

```
146 //-----------------------------------
147 MyView::MyView(QWidget *parent)
148       : QTableView(parent) {
149
150 }
151
152 void MyView::resizeEvent(QResizeEvent *event) {
153     resizeRowsToContents();
154     resizeColumnsToContents();
155     QTableView::resizeEvent(event);
156 }
157
158 //-----------------------------------
159 int main(int argc, char *argv[]) {
160
161     QApplication app(argc, argv);
162
163     QTextCodec *codec = QTextCodec::codecForName("CP1251");
164     QTextCodec::setCodecForTr(codec);
165     QTextCodec::setCodecForCStrings(codec);
166
167     QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");
168     db.setDatabaseName("db1");
169     db.setUserName("username");
170     db.setPassword("password");
171     db.open();
172
173     // QSqlQuery q;
174     // Для корректного отображения кириллицы, возможно,
175     // придется установить кодировку:
176     //q.exec(QObject::tr("SET NAMES 'cp1251'"));
177
178     MyModel model;
179
180     MyView view;
181     view.setModel(&model);
182
183     view.setAlternatingRowColors(true);
184     view.resizeRowsToContents();
185     view.resizeColumnsToContents();
186     view.show();
187
188     return app.exec();
189 }
```

Пояснения к программе:

- (8–10) — в конструкторе модели таблицы вызвали конструктор базового класса и определенный ниже метод `refresh` (131), в котором указаны заголовки столбцов и текст SQL-запроса, выполняемый для получения данных из БД;
- (13–22) — в методе `flags()` переопределили свойства ячеек таблицы;
- (16) — получили значения флагов-свойств, которые определены для данного столбца в базовом классе;
- (17, 18) — для всех столбцов, кроме первого (вернее, нулевого) и последнего, выставили флаг разрешения редактирования `Qt::ItemIsEditable`;
- (19, 20) — для последнего столбца установили признак `Qt::ItemIsUserCheckable`, в результате во всех его ячейках перед текстовой меткой будет отображаться элемент `QCheckBox`;
- (24–92) — в методе `data()` переопределяются данные, "хранящиеся" в ячейках таблицы. В зависимости от параметра `role` (роль), это либо сами данные, читаемые из БД, либо параметры их отображения;
- (28) — получили значение, определенное в базовом классе;
- (30–90) — определили параметры ячейки таблицы, в зависимости от значения параметра `role`: `Qt::DisplayRole` (данные для отображения ячейки таблицы), `Qt::EditRole` (данные для режима редактирования), `Qt::TextColorRole` (цвет текста), `Qt::TextAlignmentRole` (выравнивание текста), `Qt::FontRole` (параметры шрифта), `Qt::BackgroundColorRole` (цвет фона ячейки), `Qt::CheckStateRole` (надо ли отображать элемент `QCheckBox`), `Qt::SizeHintRole` (предпочитаемые размеры ячейки);
- (34, 35) — для самого первого столбца перед числом добавили символ №;
- (36, 37) — для столбца с датами рождения в режиме отображения задали формат "`число.месяц.год`" (по умолчанию действует формат "`год-месяц-число`");
- (38–40) — для столбца зарплат в режиме отображения установили точность два знака после запятой;
- (41, 42) — в последнем столбце отображаем текст "`Да`" или "`Нет`";
- (43, 44) — для всех остальных столбцов (остался только столбец с именами работников) ничего не изменяем;
- (46–50) — для столбца с именами установили голубой цвет символов, для всех остальных — цвет по умолчанию;
- (52–58) — числа выравниваем по правому краю, даты и надписи "`Да/Нет`" — по центру, остальные ячейки — по левому краю. Во всех случаях центрируем по вертикали;

- (60–65) — для столбца имен задали жирный шрифт "Helvetica" размером 10 пунктов, для всех остальных ячеек — шрифт по умолчанию;
- (67–74) — установили цвет фона для ячеек самого первого и последнего столбцов (для четных строк — немного светлее, чем для нечетных). Для всех остальных столбцов — цвет фона по умолчанию;
- (76–81) — для ячеек последнего столбца установили или сбросили галочку в зависимости от значения, хранящегося в БД;
- (83–90) — установили предпочтительные размеры ячеек;
- (94–129) — в методе `setData()` переопределяются данные, которые будут записываться в БД;
- (98, 99) — могут изменяться только ячейки из столбцов с первого по четвертый;
- (101–103) — выяснили значение поля `id` (целочисленный идентификатор) изменяемой записи;
- (107–128) — определили текст SQL-запроса для обновления данных каждого столбца таблицы;
- (131–144) — задали текст SQL-запроса для чтения данных из БД и отображаемые заголовки столбцов таблицы. Заметим, что если не указать критерий сортировки, то для некоторых драйверов (например, QPSQL) после редактирования строки таблицы будут "прыгать" (только что измененная строка будет перемещаться на последнее место);
- (147–150) — в конструкторе представления таблицы не забыли вызвать конструктор базового класса;
- (152–156) — определили действия, выполняемые при изменении размеров таблицы (в данном случае размеры автоматически подгоняются по содержимому ячеек);
- (159–189) — процедура `main()` повторяет текст листинга 17.16, только в (178–181) используется наш класс модели и класс для представления таблицы.

Для своей модели мы выбрали базовый класс `QSqlQueryModel`, работающий с произвольным набором SQL-запросов для чтения и записи данных в БД. Но за все приходится платить: нам пришлось подробно расписывать реализацию методов `data()` и `setData()`. В данном случае мы имели дело с единственной таблицей базы данных, поэтому можно было в качестве базового класса взять `QSqlTableModel`.

Проверьте, как работает эта программа. Намного лучше, чем предыдущая (см. *главу 17*), не правда ли? Размеры ячеек теперь не "прыгают" при редак-

тировании, а при щелчке левой кнопкой мыши по элементу `QCheckBox` в ячейках последнего столбца автоматически изменяется текстовая метка "`Да/Нет`". Но ввести "трехзначную" зарплату все еще не получается. Мы исправим данный недостаток в следующем разделе.

18.2. Делегаты для ячеек таблицы

Напомним, что *делегаты* — это специальные классы, которые могут использоваться для управления режимами отображения или редактирования ячеек таблицы `QTableView`, равно как и элементов других представлений (`QListView`, `QTreeView`).

ЗАДАЧА. Разработать делегаты для редактирования дат и чисел в ячейках таблицы (рис. 18.2).

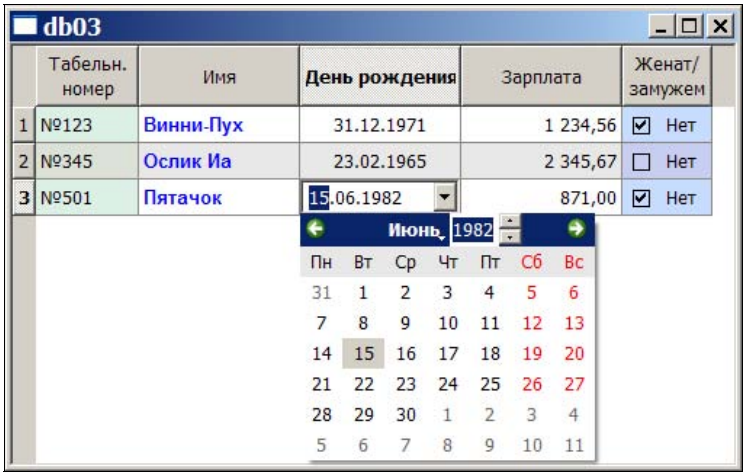


Рис. 18.2. Делегаты для ячеек таблицы

Решение. Создание пользовательского делегата для редактирования данных включает следующие шаги:

- описание класса-делегата на основе `QItemDelegate`;
- реализацию конструктора и виртуальных методов `createEditor()`, `setEditorData()`, `setModelData()` и `updateEditorGeometry()`;
- создание экземпляра делегата (например, в конструкторе объекта-вида) и связывание всех или некоторых элементов вида (в случае таблицы — всех ячеек или определенных строк и/или колонок) с этим экземпляром путем вызова методов `setItemDelegate()`, `setItemDelegateForRow()` или `setItemDelegateForColumn()`.

В листингах 18.3 и 18.4 приведен текст программы.

Листинг 18.3. Файл chap18/db03/db03.h

```
#include <QtGui>
#include <QtSql>

class MyModel : public QSqlQueryModel {
    Q_OBJECT
public:
    MyModel(QObject *parent = 0);
    Qt::ItemFlags flags(const QModelIndex &index) const;
    QVariant data(const QModelIndex &index,
                  int role = Qt::DisplayRole) const;
    bool setData(const QModelIndex &index,
                 const QVariant &value, int role);
private:
    void refresh();
};

//-----------------------------------------------
class MyView : public QTableView {
    Q_OBJECT
public:
    MyView(QWidget *parent = 0);
private:
    virtual void resizeEvent(QResizeEvent *event);
};

//-----------------------------------------------
class MyDSBDelegate : public QItemDelegate {
    Q_OBJECT
public:
    MyDSBDelegate(double min=0.00,
                  double max=999999999.99,
                  double step=0.1,
                  int precision=2,
                  QObject *parent = 0);
    QWidget *createEditor(
                QWidget *parent,
                const QStyleOptionViewItem &option,
                const QModelIndex &index) const;
    void setEditorData(QWidget *editor,
                       const QModelIndex &index) const;
```

```
41     void setModelData(QWidget *editor,
42                       QAbstractItemModel *model,
43                       const QModelIndex &index) const;
44     void updateEditorGeometry(
45             QWidget *editor,
46             const QStyleOptionViewItem &option,
47             const QModelIndex &index) const;
48 private:
49     double m_min;
50     double m_max;
51     double m_step;
52     int m_precision;
53 };
54
55 //-----------------------------------------
56 class MyDEDelegate : public QItemDelegate {
57     Q_OBJECT
58 public:
59     MyDEDelegate(bool calpopup = true,
60                  QObject *parent = 0);
61     QWidget *createEditor(
62                 QWidget *parent,
63                 const QStyleOptionViewItem &option,
64                 const QModelIndex &index) const;
65     void setEditorData(QWidget *editor,
66                        const QModelIndex &index) const;
67     void setModelData(QWidget *editor,
68                       QAbstractItemModel *model,
69                       const QModelIndex &index) const;
70     void updateEditorGeometry(
71             QWidget *editor,
72             const QStyleOptionViewItem &option,
73             const QModelIndex &index) const;
74 private:
75     bool m_calpopup;
76 };
```

Пояснения к программе:

- (4–15) — класс модели таблицы;
- (18–24) — представление таблицы;
- (27–53) — класс-делегат для вещественных чисел (на основе элемента `QDoubleSpinBox`);
- (56–76) — класс-делегат для ввода даты (на основе элемента `QDateEdit`).

Листинг 18.4. Файл chap18/db03/db03.cpp

```
// Таблица базы данных: делегаты

#include <QtGui>
#include <QtSql>

#include "db03.h"

MyModel::MyModel(QObject *parent)
        : QSqlQueryModel(parent) {
    refresh();
}

Qt::ItemFlags MyModel::flags(
        const QModelIndex &index) const {
    // Как в листинге 18.2
}

QVariant MyModel::data(
            const QModelIndex &index,
            int role) const {
    // Как в листинге 18.2
}

bool MyModel::setData(
            const QModelIndex &index,
            const QVariant &value,
            int /* role */) {
    // Как в листинге 18.2
}

void MyModel::refresh() {
    // Как в листинге 18.2
}

//-----------------------------------
MyView::MyView(QWidget *parent)
      : QTableView(parent) {

    MyDSBDelegate *dsbd = new MyDSBDelegate(
        0.0, 999999.99, 0.05, 2, this);
    setItemDelegateForColumn(3, dsbd);

```

```
154     MyDEDelegate *ded = new MyDEDelegate(
155             true, this);
156     setItemDelegateForColumn(2, ded);
157 }
158
159 void MyView::resizeEvent(QResizeEvent *event) {
        // Как в листинге 18.2
163 }
164
165 //-----------------------------------
166 MyDSBDelegate::MyDSBDelegate(
167                   double min,
168                   double max,
169                   double step,
170                   int precision,
171                   QObject *parent)
172              : QItemDelegate(parent),
173                m_min(min),
174                m_max(max),
175                m_step(step),
176                m_precision(precision) {
177 }
178
179 QWidget *MyDSBDelegate::createEditor(
180             QWidget *parent,
181             const QStyleOptionViewItem& /* option */,
182             const QModelIndex& /* index */) const {
183     QDoubleSpinBox *editor = new QDoubleSpinBox(parent);
184     editor->setMinimum(m_min);
185     editor->setMaximum(m_max);
186     editor->setDecimals(m_precision);
187     editor->setSingleStep(m_step);
188     editor->installEventFilter(const_cast<MyDSBDelegate*>(this));
189     return editor;
190 }
191
192 void MyDSBDelegate::setEditorData(
193                 QWidget *editor,
194                 const QModelIndex &index) const {
195     double value = index.model()->data(
196             index, Qt::EditRole).toDouble();
197     QDoubleSpinBox *dsb = static_cast<QDoubleSpinBox*>(editor);
198     dsb->setValue(value);
199 }
```

```
200
201 void MyDSBDelegate::setModelData(
202             QWidget *editor,
203             QAbstractItemModel *model,
204             const QModelIndex& index) const {
205     QDoubleSpinBox *dsb = static_cast<QDoubleSpinBox*>(editor);
206     dsb->interpretText();
207     double value = dsb->value();
208     model->setData(index, value);
209 }
210
211 void MyDSBDelegate::updateEditorGeometry(
212             QWidget *editor,
213             const QStyleOptionViewItem &option,
214             const QModelIndex& /* index */) const {
215     editor->setGeometry(option.rect);
216 }
217
218 //-----------------------------------
219 MyDEDelegate::MyDEDelegate(
220                     bool calpopup,
221                     QObject *parent)
222             : QItemDelegate(parent),
223               m_calpopup(calpopup) {
224 }
225
226 QWidget *MyDEDelegate::createEditor(
227             QWidget *parent,
228             const QStyleOptionViewItem& /* option */,
229             const QModelIndex& /* index */) const {
230     QDateEdit *editor = new QDateEdit(parent);
231     editor->setCalendarPopup(m_calpopup);
        #if QT_VERSION >= 0x040400
        if (m_calpopup)
            editor->calendarWidget()->setFirstDayOfWeek(Qt::Monday);
        #endif
232     editor->installEventFilter(const_cast<MyDEDelegate*>(this));
233     return editor;
234 }
235
236 void MyDEDelegate::setEditorData(
237                 QWidget *editor,
238                 const QModelIndex &index) const {
```

```
239     QDate value = index.model()->data(
240             index, Qt::EditRole).toDate();
241     QDateEdit *de = static_cast<QDateEdit*>(editor);
242     de->setDate(value);
243 }
244
245 void MyDEDelegate::setModelData(
246             QWidget *editor,
247             QAbstractItemModel *model,
248             const QModelIndex& index) const {
249     QDateEdit *de = static_cast<QDateEdit*>(editor);
250     de->interpretText();
251     QDate value = de->date();
252     model->setData(index, value);
253 }
254
255 void MyDEDelegate::updateEditorGeometry(
256             QWidget *editor,
257             const QStyleOptionViewItem &option,
258             const QModelIndex& /* index */) const {
259     editor->setGeometry(option.rect);
260 }
261
262 //-----------------------------------
263 int main(int argc, char *argv[]) {
        // Как в листинге 18.2
291 }
```

Пояснения к программе:

- (1–144) — начало листинга повторяет текст листинга 18.2;
- (150, 151) — создаем экземпляр делегата для редактирования вещественных чисел; первые два параметра конструктора — разрешенный диапазон, третий параметр — шаг изменения, четвертый — число знаков после запятой, пятый — родительский виджет;
- (152) — связали только что созданный экземпляр делегата с третьим (считая с нуля) столбцом таблицы;
- (154–156) — создали экземпляр делегата для редактирования даты (первый параметр конструктора определяет, будет ли при вводе даты использоваться всплывающее окно календаря) и связали его со вторым столбцом таблицы;
- (159–163) — реализация метода `resizeEvent()` листинга 18.2;

- (166–177) — конструктор делегата для редактирования вещественных чисел. Параметры: диапазон, шаг, точность и указатель на родительский виджет;
- (179–190) — переопределяем виртуальный метод `createEditor()`, который выполняется при входе в режим редактирования ячейки таблицы. В нашем случае создается элемент `QDoubleSpinBox` и задаются его параметры;
- (185) — наконец-то наша зарплата стала больше, чем 99,99;
- (188) — установили фильтр для событий: теперь наш экземпляр `MyDSBDelegate` (параметр `this`) будет получать все сообщения, посылаемые элементу редактирования `edit`;
- (192–199) — переопределили виртуальный метод `setEditorData()`, в котором задаются данные, отображаемые элементом редактирования при его создании;
- (201–209) — переопределили виртуальный метод `setModelData()`, в котором определяются данные, передаваемые элементом редактирования в модель;
- (211–216) — переопределили виртуальный метод `updateEditorGeometry()`, в котором осуществляется прорисовка элемента редактирования;
- (219–260) — реализовали класс `MyDEDelegate` для редактирования даты. Свойство `m_calpopup` определяет режим редактирования: будет ли использоваться "всплывающий" календарик или кнопки инкремента/декремента;
- (263–291) — функция `main` (как и в предыдущем примере).

ЗАДАЧА. Требуется изменить вид элемента не только в режиме редактирования, но и в обычном состоянии.

Решение. В классе делегата необходимо переопределить виртуальный метод `paint()`. Например, для `MyDSBDelegate::paint()` это иллюстрирует листинг 18.5.

Листинг 18.5

```
void MyDSBDelegate::paint (
      QPainter *painter,
      const QStyleOptionViewItem& option,
      const QModelIndex& index ) const {

   QString text;
   QRect rect;
   QVariant value;
```

```
    QStyleOptionViewItemV2 opt = setOptions(index, option);
    value = index.data(Qt::DisplayRole);
    text = QLocale().toString(value.toDouble(), 'f', 2);
    opt.displayAlignment = Qt::AlignRight | Qt::AlignVCenter;
    drawDisplay(painter, opt, opt.rect, text);
}
```

Здесь задано отображение чисел с точностью до двух знаков после десятичной точки, с выравниванием по правому краю и центрированием по вертикали. Сравните данный фрагмент с реализацией метода `QItemDelegate::paint()`, которую можно найти в исходных текстах библиотеки Qt4, в файле

`src/gui/itemviews/qitemdelegate.cpp`

ЗАДАЧА. Необходимо изменить вид линий обрамления вокруг некоторых ячеек таблицы.

Решение. Если в конструкторе класса `MyView` или в главной программе отключить показ сетки таблицы, заданный по умолчанию:

```
view->setShowGrid(false);
```

а в методе `paint()` дописать несколько строк (листинг 18.6), то можно получить произвольный цвет, толщину и стиль линий обрамления ячеек (рис. 18.3).

Листинг 18.6

```
QPen pen = QPen(Qt::red, 2, Qt::DashLine,
                Qt::FlatCap, Qt::MiterJoin);
painter->setPen(pen);
painter->drawLine(opt.rect.left(), opt.rect.bottom(),
                  opt.rect.right(), opt.rect.bottom());
painter->drawLine(opt.rect.right(), opt.rect.top(),
                  opt.rect.right(), opt.rect.bottom());
```

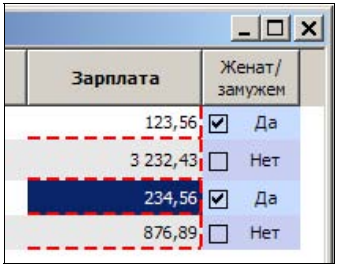


Рис. 18.3. Изменение обрамления ячеек с помощью делегатов

ЗАДАЧА. Требуется разместить в ячейке таблицы не только текст, но и графическое изображение.

Решение. Ручной прорисовкой области ячейки в методе `paint()` класса-делегата можно добиться любых визуальных эффектов, но для размещения графического изображения в ячейке таблицы проще использовать роль `Qt::DecorationRole` в методе `data()` класса-модели, например, так, как показано в листинге 18.7.

Листинг 18.7

```
switch (role) {
  case Qt::DecorationRole:
     if (index.column() == 0)
         return Qt::red;
     else if (index.column() == 1)
         return qApp->style()->standardIcon(QStyle::SP_DirIcon);
     else if (index.column() == 4)
         return qApp->style()->standardIcon(QStyle::SP_ArrowUp);
     else
         return value;
. . . . .
```

В результате в каждой ячейке нулевого столбца перед номером работника появится квадратик красного цвета, в первом столбце перед фамилиями будут отображаться значки в виде желтых папок, а в последнем столбце между элементами `QCheckBox` и текстом — стрелки (рис. 18.4). Вместо стандартного значка можно загрузить любое изображение из файла:

```
return QIcon("имя файла или ресурса");
```

Замечание

Чтобы таблица прорисовывалась максимально быстро, изображение `QIcon` лучше загрузить из файла заранее, а в методе `data()` указать ссылку на него.

Рис. 18.4. Графические изображения в ячейках таблицы

Получается, что всю информацию, связанную с внешним видом ячеек таблицы (шрифт, цвет, выравнивание), можно перенести из модели в классы делегатов. Хотя идеологически этот вариант кажется более предпочтительным (модель — для данных, вид — для их отображения), но на самом деле вопрос сводится к тому, что́ считать данными: перейдя на более высокий уровень абстракции, мы понимаем, что шрифт, цвет, параметры выравнивания и точность представления чисел — это тоже особый вид данных (вернее, метаданных), которые могут храниться в БД и изменяться пользователем.

18.3. Связывание элементов управления форм с моделью данных

Для связывания строки (записи) таблицы базы данных (модели) с элементами формы для ввода и редактирования данных в библиотеке Qt предусмотрен класс `QDataWidgetMapper`. Вот общий алгоритм его использования:

- создать объект отображения:

```
QDataWidgetMapper *mapper = new QDataWidgetMapper(parent);
```

- установить модель данных, например:

```
QStandardItemModel *model;
mapper->setModel(model);
```

- добавить элементы ввода для каждого поля, например:

```
QLineEdit *nameEdit = new QLineEdit(parent);
QTextEdit *addressEdit = new QTextEdit(parent);
QSpinBox *ageSpinBox = new QSpinBox(parent);
mapper->addMapping(nameEdit, 0);
mapper->addMapping(addressEdit, 1);
mapper->addMapping(ageSpinBox, 2);
```

Второй параметр — это номер столбца модели, с которым связывается данный элемент управления.

- навигация по строкам модели осуществляется с помощью методов-слотов `QDataWidgetMapper::toFirst()`, `toLast()`, `toNext()` и `toPrevious()`. При смене текущей записи (индекса модели) генерируется сигнал `QDataWidgetMapper::currentIndexChanged(int index)`;
- следует выбрать один из двух способов обновления информации в модели, устанавливаемый методом `QDataWidgetMapper::setSubmitPolicy()`:
  - `QDataWidgetMapper::AutoSubmit` — новые данные записываются в модель из текущего элемента управления в тот момент, когда последний

теряет фокус, т. е. пользователь переходит к следующему элементу управления формы;

- `QDataWidgetMapper::ManualSubmit` — новые данные записываются в текущую строку модели из всех элементов управления формы при выполнении метода-слота `QDataWidgetMapper::submit()`. При удачном выполнении операции возвращается логическое значение `true`. Чтобы отказаться от записи изменений и вернуть первоначальные значения в элементы ввода формы, необходимо вызвать метод-слот `QDataWidgetMapper::revert()`;

❐ если требуется, можно применить классы-делегаты, чтобы, например, в качестве элемента ввода данных для некоторого столбца таблицы сформировать поле с раскрывающимся списком, значения в котором читаются из другой таблицы базы данных:

```
QComboBox *comboBox = new QComboBox(parent);
comboBox->setModel(mymodel);
mapper->setItemDelegate(new MyDelegate(parent));
mapper->addMapping(comboBox, 3);
```

Листинг 18.8 содержит пример класса-делегата.

Листинг 18.8

```
class Delegate : public QItemDelegate {
   Q_OBJECT
public:
   Delegate(QObject *parent = 0);
   void setEditorData(QWidget *editor,
                      const QModelIndex &index) const;
   void setModelData(QWidget *editor,
                     QAbstractItemModel *model,
                     const QModelIndex &index) const;
};
```

Метод `setEditorData()` инициализирует элемент редактирования данными из модели (листинг 18.9).

Листинг 18.9

```
void MyDelegate::setEditorData(
        QWidget *editor,
        const QModelIndex &index) const {
```

```
   if (!editor->metaObject()->userProperty().isValid()){
      if (editor->property("currentIndex").isValid()) {
         editor->setProperty("currentIndex", index.data());
         return;
      }
   }
   QItemDelegate::setEditorData(editor, index);
}
```

Метод `setModelData()` выполняет обратную операцию: записывает данные из элемента редактирования в модель, например, так, как иллюстрирует листинг 18.10.

Листинг 18.10

```
void MyDelegate::setModelData(
               QWidget *editor,
               QAbstractItemModel *model,
               const QModelIndex &index) const {
   if (!editor->metaObject()->userProperty().isValid()) {
      QVariant value = editor->property("currentIndex");
      if (value.isValid()) {
         model->setData(index, value);
         return;
      }
   }
   QItemDelegate::setModelData(editor, model, index);
}
```

ЗАДАЧА. Необходимо разработать форму для ввода информации в текущую строку таблицы базы данных (рис. 18.5).

Решение иллюстрируют листинги 18.11 и 18.12.

Листинг 18.11. Файл chap18/db04/db04.h

```
1  #include <QtGui>
2  #include <QtSql>
3
4  class MyModel : public QSqlTableModel {
5     Q_OBJECT
6  public:
7     MyModel(QObject *parent = 0);
8  };
```

```
 9
10 //-----------------------------------------------
11 class MyView : public QTableView {
12    Q_OBJECT
13 public:
14    MyView(QWidget *parent = 0);
15 private:
16    virtual void resizeEvent(QResizeEvent *event);
17 public slots:
18    void saveData();
19    void insertRow();
20    void deleteRow();
21 };
```

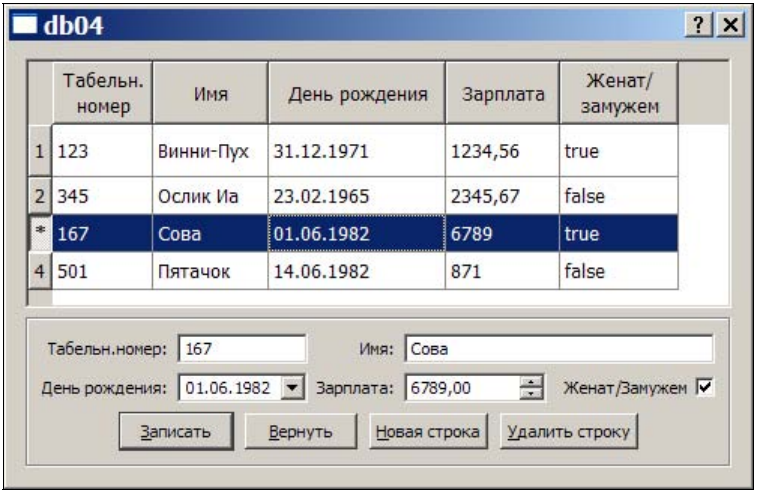


Рис. 18.5. Таблица и форма для ввода данных

Листинг 18.12. Файл chap18/db04/db04.cpp

```
22 #include <QtGui>
23 #include <QtSql>
24 #include "db04.h"
25
26 #define tr  QObject::tr
27
28 MyModel::MyModel(QObject *parent)
29     : QSqlTableModel(parent) {
30
```

```
31    setTable(tr("employee"));
32    setSort(0, Qt::AscendingOrder);
33    select();
34
35    setHeaderData(0, Qt::Horizontal,
36                  tr("Табельн.\nномер"));
37    setHeaderData(1, Qt::Horizontal,
38                  tr("Имя"));
39    setHeaderData(2, Qt::Horizontal,
40                  tr("День рождения"));
41    setHeaderData(3, Qt::Horizontal,
42                 tr("Зарплата"));
43    setHeaderData(4, Qt::Horizontal,
44                 tr("Женат/\nзамужем"));
45  }
46
47  //-----------------------------------
48  MyView::MyView(QWidget *parent)
49        : QTableView(parent) {
50
51     setSelectionBehavior(QAbstractItemView::SelectRows);
52     setEditTriggers(QAbstractItemView::NoEditTriggers);
53  }
54
55  void MyView::resizeEvent(QResizeEvent *event) {
56     QTableView::resizeEvent(event);
57     resizeRowsToContents();
58  }
59
60  void MyView::saveData(){
61     MyModel *m = qobject_cast<MyModel*>(model());
62     if ( ! m ) return;
63
64     m->database().transaction();
65     if (m->submitAll()) {
66         m->database().commit();
67     }else{
68         m->database().rollback();
69         QMessageBox::warning(
70             this, tr("Error"),
71             tr("The database reported an error: %1")
72             .arg(m->lastError().text()));
```

```
 73         m->select();
 74     }
 75 }
 76
 77 void MyView::insertRow(){
 78     MyModel *m = qobject_cast< MyModel*>(model());
 79     if ( ! m )  return;
 80
 81     QModelIndex insertIndex = currentIndex();
 82     int row = insertIndex.row() == -1 ? 0 : insertIndex.row();
 83     m->insertRow(row);
 84     insertIndex = m->index(row, 0);
 85     setCurrentIndex(insertIndex);
 86     // edit(insertIndex);
 87 }
 88
 89 void MyView::deleteRow(){
 90     MyModel *m = qobject_cast<MyModel*>(model());
 91     if ( ! m ) return;
 92     QModelIndexList indexlist = selectionModel()->selectedIndexes();
 93     for (int i = 0; i < indexlist.count(); ++i){
 94        m->removeRow(indexlist.at(i).row());
 95     }
 96     m->submitAll();
 97 }
 98
 99 //-----------------------------------
100 int main(int argc, char *argv[]) {
101
102     QApplication app(argc, argv);
103
104     QTextCodec *codec = QTextCodec::codecForName("CP1251");
105     QTextCodec::setCodecForTr(codec);
106     QTextCodec::setCodecForCStrings(codec);
107
108     QDialog dlg;
109     QVBoxLayout *layout = new QVBoxLayout(&dlg);
110
111     QSqlDatabase db = QSqlDatabase::addDatabase("QPSQL");
112     db.setDatabaseName("...");
113     db.setUserName("...");
114     db.setPassword("...");
115     db.open();
116
```

```
117     // QSqlQuery q;
118     // q.exec(QObject::tr("SET NAMES 'cp1251'"));
119
120     MyModel *model = new MyModel(&dlg);
121     model->setEditStrategy(QSqlTableModel::OnManualSubmit);
122
123     MyView *view = new MyView(&dlg);
124     view->setModel(model);
125     view->setAlternatingRowColors(false);
126     view->resizeRowsToContents();
127     view->resizeColumnsToContents();
128     view->resize(400,300);
129     layout->addWidget(view, 1);
130
131     QGroupBox *gb = new QGroupBox(&dlg);
132     QGridLayout *gblayout = new QGridLayout(gb);
133
134     gblayout->addWidget(
135        new QLabel(tr("Табельн.номер:"), gb),
136        0, 0, Qt::AlignRight);
137     QLineEdit *fId = new QLineEdit(gb);
138     gblayout->addWidget(fId, 0, 1);
139
140     QLineEdit *fName = new QLineEdit(gb);
141     gblayout->addWidget(
142        new QLabel(tr("Имя:"), gb),
143        0, 2, Qt::AlignRight);
144     gblayout->addWidget(fName, 0, 3, 1, 2);
145
146     QDateEdit *fBorn = new QDateEdit(gb);
147     gblayout->addWidget(
148        new QLabel(tr("День рождения:"), gb),
149        1, 0, Qt::AlignRight);
150     fBorn->setCalendarPopup(true);
151     gblayout->addWidget(fBorn, 1, 1);
152
153     QDoubleSpinBox *fSalary = new QDoubleSpinBox(gb);
154     fSalary->setMinimum(0.0);
155     fSalary->setMaximum(999999999.99);
156     fSalary->setSingleStep(0.05);
157     gblayout->addWidget(
158        new QLabel(tr("Зарплата:"), gb),
159        1, 2, Qt::AlignRight);
160     gblayout->addWidget(fSalary, 1, 3);
```

```
161
162     QCheckBox *fMarried = new QCheckBox(
163        tr("Женат/Замужем"), gb);
164     fMarried->setAttribute(Qt::WA_RightToLeft);
165     gblayout->addWidget(fMarried, 1, 4);
166
167     QDialogButtonBox * btns = new QDialogButtonBox;
168
169     QPushButton *btnSave = new QPushButton(tr("&Записать"));
170
171     QObject::connect( btnSave, SIGNAL(clicked()),
171                       view, SLOT(saveData()));
172     btns->addButton(btnSave, QDialogButtonBox::ActionRole);
173
174     QPushButton *btnRevert = new QPushButton(tr("&Вернуть"));
175     QObject::connect( btnRevert, SIGNAL(clicked()),
176                       model, SLOT(revertAll()));
177     btns->addButton(btnRevert, QDialogButtonBox::ActionRole);
178
179     QPushButton *btnNewRow = new QPushButton(tr("&Новая строка"));
180     QObject::connect( btnNewRow, SIGNAL(clicked()),
181                       view, SLOT(insertRow()));
182     btns->addButton(btnNewRow, QDialogButtonBox::ActionRole);
183
184     QPushButton *btnDelRow = new QPushButton(tr("&Удалить строку"));
185     QObject::connect( btnDelRow, SIGNAL(clicked()),
186                       view, SLOT(deleteRow()));
187     btns->addButton(btnDelRow, QDialogButtonBox::ActionRole);
188
189     btns->setCenterButtons(true);
190     gblayout->addWidget(btns, 2, 0, 1, 5);
191
192     QDataWidgetMapper *mapper = new QDataWidgetMapper(&dlg);
193     mapper->setModel(model);
194     mapper->addMapping(fId, 0);
195     mapper->addMapping(fName, 1);
196     mapper->addMapping(fBorn, 2);
197     mapper->addMapping(fSalary, 3);
198     mapper->addMapping(fMarried, 4);
199     mapper->setSubmitPolicy(QDataWidgetMapper::AutoSubmit);
200     mapper->toFirst();
201     QObject::connect(
202              view->selectionModel(),
203              SIGNAL(currentRowChanged(QModelIndex, QModelIndex)),
```

```
204             mapper,
205             SLOT(setCurrentModelIndex(QModelIndex)));
206     view->setCurrentIndex(model->index(0, 0));
207
208     gblayout->setColumnStretch(0, 0);
209     gblayout->setColumnStretch(1, 1);
210     gblayout->setColumnStretch(2, 0);
211     gblayout->setColumnStretch(3, 1);
212     gblayout->setColumnStretch(4, 1);
213     gblayout->setRowStretch(0, 0);
214     gblayout->setRowStretch(1, 0);
215
216     gb->setLayout(gblayout);
217     layout->addWidget(gb, 0);
218
219     dlg.setLayout(layout);
220     dlg.resize(550, 360);
221     dlg.show();
222
223     return app.exec();
224 }
```

Пояснения к программе:

- (4–8, 28–45) — здесь мы разрабатываем свой класс модели таблицы базы данных на основе `QSqlTableModel`;
- (31) — устанавливаем название таблицы базы данных;
- (32) — если не задать порядок сортировки, то для некоторых драйверов (например, `QPSQL`) только что отредактированная строка будет "перепрыгивать" на последнее место;
- (33) — прочитали данные, хранящиеся в таблице;
- (35–44) — установили текст заголовков столбцов таблицы;
- (11–21, 48–91) — здесь мы разрабатываем свой класс представления таблицы;
- (51) — выделяться будут не ячейки таблицы, а строки;
- (52) — редактирование ячеек запрещено;
- (55–58) — при каждом изменении размеров таблицы высота строк будет подгоняться по содержимому ячеек;
- (60–74) — слот `saveData()`, который вызывается при нажатии кнопки **Запомнить**;

- (61) — поскольку метод `QTableView::model()`, унаследованный от `QAbstractItemView`, возвращает объект типа `QAbstractItemModel`, мы сначала преобразуем указатель к нашему классу `MyModel`;
- (64–74) — начинаем транзакцию и пытаемся записать в базу данных все сделанные пользователем изменения, при неудаче отменяем всю транзакцию и выводим сообщение об ошибке;
- (77–87) — слот, выполняемый при нажатии кнопки **Новая строка**;
- (86) — если бы наша таблица редактировалась непосредственно в ячейках, а не с помощью элементов формы, то здесь нужно было вызвать метод `QAbstractItemView::edit()`;
- (89–97) — слот, выполняемый при нажатии кнопки **Удалить строку**;
- (92) — получили список выделенных строк;
- (93, 94) — удалили все выделенные строки;
- (96) — подтвердили все сделанные изменения (на этот раз обошлись без транзакции);
- (100–224) — главная программа;
- (134–165) — элементы формы для ввода информации в текущую строку таблицы;
- (167–187) — кнопки формы;
- (192–206) — связываем элементы формы с моделью данных.

Замечание

Разумеется, в этом примере управлять способом отображения и/или редактирования данных в ячейках таблицы можно посредством делегатов.

18.4. Консоль запросов

ЗАДАЧА. Разработать приложение, позволяющее подключаться к базе данных и выполнять запросы SQL.

Решение. Сначала с помощью программы Qt Designer создадим самый простой диалог для подключения к базе данных (рис. 18.6). При запуске Qt Designer выберем режим **Dialog with Buttons Bottom** (Диалог с нижним расположением кнопок). С помощью мыши выберем слева в галерее **Widget Box**, в группе **Display Widgets** виджет **Label** (надпись) и разместим его в левой верхней части окна.

Если в правой части окна посередине или внизу мы не видим панель для редактирования свойств виджетов, то отобразим ее, выполнив пункт меню

Tools (Инструменты) | **Property Editor** (Редактор свойств). Зададим имя `lbMode` для нового виджета (свойство **objectName** в группе **QObject**) и текст надписи `&Mode` (Режим) (свойство **text** в группе **QLabel**).

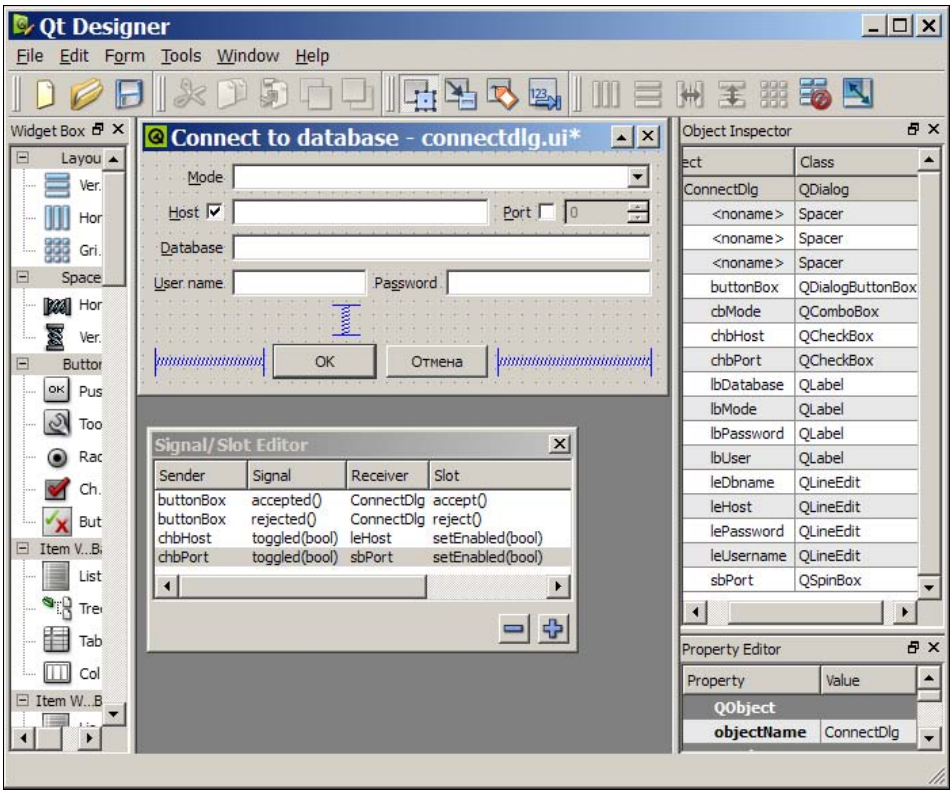


Рис. 18.6. Разработка диалога подключения к базе данных

Справа от надписи разместим виджет **Combo Box** (он находится в группе **Input Widgets**). Назовем его `cbMode`.

Аналогично добавим остальные надписи `lbHost`, `lbPort`, `lbDatabase`, `lbUsername` и `lbPassword`, элементы **Check Box** из группы **Buttons** с именами `chbBox` и `chbPort`, однострочные поля редактирования **Line Edit** из группы **Input Widgets** с именами `leHost`, `leDatabase`, `leUsername` и `lePassword`, поле для ввода целых чисел с кнопками инкремента/декремента **Spin Box** из группы **Input Widgets** с именем `sbPort` (см. рис. 18.6). Для последнего элемента изменим свойство **maximum**: вместо установленного по умолчанию числа `99` напишем `65535`. Слева, справа и сверху от кнопок **QDialogButtonBox** вставим две растяжки **Horizontal Spacer** и одну **Vertical Spacer** (их можно найти в группе **Spacers**).

С помощью контекстного меню, вызываемого щелчком правой кнопкой мыши по пустому месту на форме, выполним команду **Lay out | Lay out in a Grid** (Расположить внутри сетки). В результате будет автоматически создан сеточный менеджер размещения `QGridLayout`, и теперь при изменении размеров окна диалога будут соответствующим образом изменяться размеры всех его элементов.

Теперь останется определить свой класс диалога, взяв в качестве базового диалог, разработанный в Qt Designer (листинги 18.13 и 18.14).

Листинг 18.13. Файл chap18/sqlquery/connectdlg.h

```
#include "ui_connectdlg.h"
class ConnectDlg : public QDialog, public Ui_ConnectDlg {
    Q_OBJECT
public:
    ConnectDlg(const QString& mode = "",
               const QString& host = "",
               int port = 0,
               const QString& dbname = "",
               const QString& username = "" );
};
```

Листинг 18.14. Файл chap18/sqlquery/connectdlg.cpp

```
#include <QtSql>
#include "connectdlg.h"

ConnectDlg::ConnectDlg(
               const QString& mode,
               const QString& host,
               int port,
               const QString& dbname,
               const QString& username )
          : QDialog() {
    setupUi(this);
    QStringList drivers = QSqlDatabase::drivers();
    drivers.removeAll("QMYSQL3");
    drivers.removeAll("QOCI8");
    drivers.removeAll("QODBC3");
    drivers.removeAll("QPSQL7");
    drivers.removeAll("QTDS7");
    cbMode->addItems(drivers);
    int i = cbMode->findText(mode, Qt::MatchFixedString);
```

```
    if ( i > -1 )
        cbMode->setCurrentIndex(i);

    if (!host.isEmpty()){
        chbHost->setCheckState(Qt::Checked);
        leHost->setText(host);
    }else{
        chbHost->setCheckState(Qt::Unchecked);
        leHost->clear();
    }
    if (port > 0){
        chbPort->setCheckState(Qt::Checked);
        sbPort->setValue(port);
    }else{
        chbPort->setCheckState(Qt::Unchecked);
        sbPort->setValue(0);
    }
    leDbname->setText(dbname);
    leUsername->setText(username);
    lePassword->clear();
}
```

Аналогично создадим основной диалог консоли запросов (рис. 18.7).

Для отображения результата выполнения запроса `tvResult` (основная часть диалога) вполне хватило бы функциональности стандартного класса `QSqlTableView`, но мы хотим показать, как в Qt Designer можно использовать собственные виджеты, разработанные на основе стандартных. Поэтому сразу после того, как мы поместим на форму виджет **QSqlTableView**, следует воспользоваться правой кнопкой мыши, чтобы вызвать на экран диалог **Promoted Widgets** (рис. 18.8), где можно задать имя нашего класса и соответствующий заголовочный файл.

Сам класс `SqlTableView` пока можно максимально упростить (листинги 18.15 и 18.16).

Листинг 18.15. Файл chap18/sqlquery/sqltableview.h

```
#include <QtGui>
#include <QtSql>
class SqlTableView : public QTableView {
    Q_OBJECT
public:
    SqlTableView(QWidget *parent = 0);
};
```

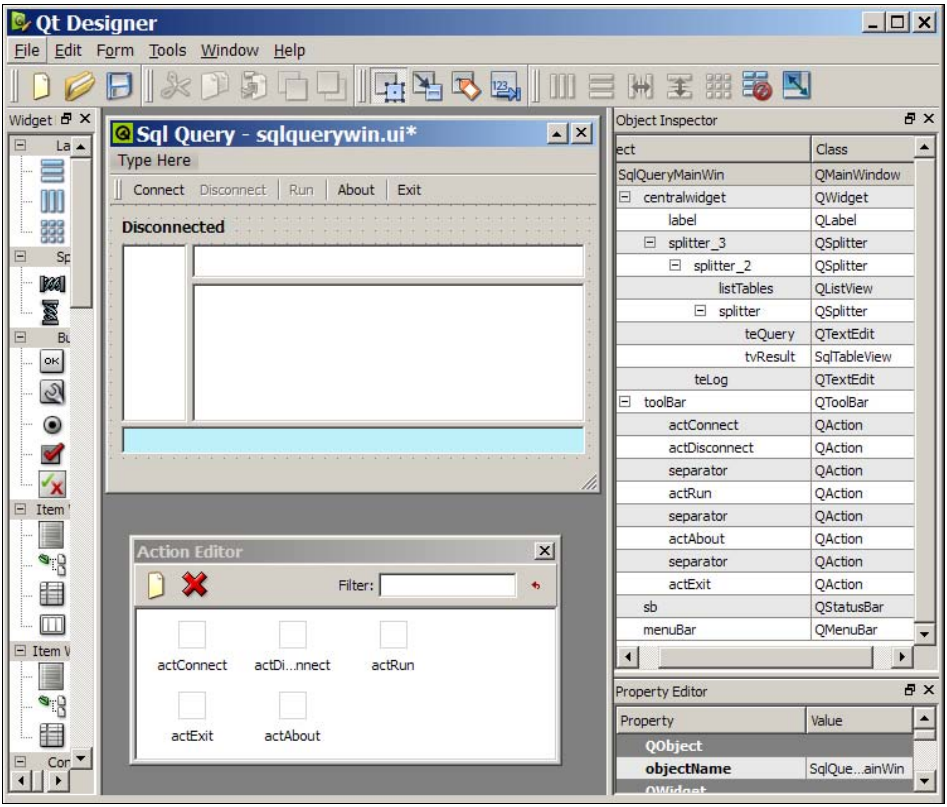


Рис. 18.7. Разработка главного окна приложения

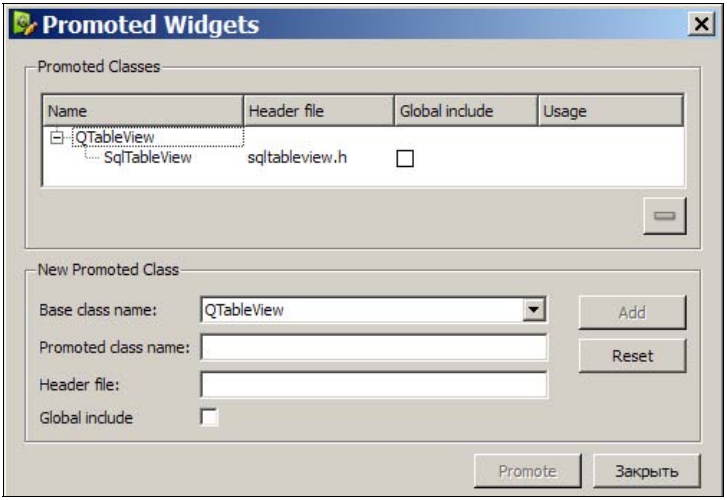


Рис. 18.8. Диалог **Promoted Widgets**

Листинг 18.16. Файл chap18/sqlquery/sqltableview.cpp

```
#include "sqltableview.h"
SqlTableView::SqlTableView(QWidget * parent)
            : QTableView(parent) {}
```

Теперь на основе созданного в Qt Designer диалога нужно определить класс главного окна нашего приложения (листинги 18.17 и 18.18).

Листинг 18.17. Файл chap18/sqlquery/mainwin.h

```
#include "ui_sqlquerywin.h"

class MainWin : public QMainWindow, public Ui_SqlQueryMainWin {
    Q_OBJECT
public:
    MainWin();
signals:
    void connectChanged(bool state);
private slots:
    void on_actExit_triggered();
    void on_actConnect_triggered();
    void on_actDisconnect_triggered();
    void getTables();
    void showStatus(bool state);
    void on_teQuery_textChanged();
    void on_actRun_triggered();
    void showResult();
    void showTable(const QString& name);
    void on_listTables_doubleClicked(const QModelIndex& index);
protected:
    virtual void closeEvent(QCloseEvent *event);

    QSqlDatabase db;
    QSqlQuery *query;
    bool connected;
    QStringList tables;
    QStringListModel *listmodel;
    QString dbmode;
    QString host;
    int port;
    QString dbname;
    QString username;
    QString password;
};
```

Листинг 18.18. Файл chap18/sqlquery/mainwin.cpp

```
#include "mainwin.h"
#include "connectdlg.h"

MainWin::MainWin() : QMainWindow() {
    setupUi(this);
    connected = false;
    dbname = "";
    host = "";
    port = 0;
    username = "";
    password = "";
    tables.clear();
    listmodel = new QStringListModel(tables);
    listmodel->setStringList(tables);
    listTables->setModel(listmodel);
    listTables->setEditTriggers(QAbstractItemView::NoEditTriggers);
    connect(this, SIGNAL(connectChanged(bool)),
            this, SLOT(showStatus(bool)));
    emit connectChanged(false);
}

void MainWin::on_actExit_triggered(){
    close();
}

void MainWin::on_actConnect_triggered(){
    ConnectDlg dlg(dbmode, host, port, dbname, username);
    if ( dlg.exec() ) {
        dbmode = dlg.cbMode->currentText();
        db = QSqlDatabase::addDatabase(dbmode);
        if (dlg.chbHost->isChecked())
            host = dlg.leHost->text();
        else
            host = "127.0.0.1";
        db.setHostName(host);

        int port;
        if (dlg.chbPort->isChecked()){
            port = dlg.sbPort->value();
            db.setPort(port);
        }else
            port = 0;
```

```
        dbname = dlg.leDbname->text().trimmed();
        if (!dbname.isEmpty()){
            db.setDatabaseName(dbname);
        }

        username = dlg.leUsername->text().trimmed();
        password = dlg.lePassword->text();
        if (!username.isEmpty()){
            db.setUserName(username);
            db.setPassword(password);
        }
        connected = db.open();
        emit connectChanged(connected);
        if (!connected)
            QMessageBox::critical(
                &dlg, tr("Connect Error"),
                db.lastError().text(),
                QMessageBox::Ok );
    }
}

void MainWin::on_actDisconnect_triggered(){
    if (connected)
        db.close();

    connected = false;
    connectChanged(connected);
}

void MainWin::closeEvent(QCloseEvent *event){
    if ( QMessageBox::question(
            this, tr("Exit application"),
            tr("Confirm to exit"),
            QMessageBox::Yes | QMessageBox::No,
            QMessageBox::Yes)
          == QMessageBox::Yes ) {
        if (connected){
            db.close();
            emit connectChanged(false);
        }
        event->accept();
    }else
        event->ignore();
}
```

```
void MainWin::getTables(){
    if (connected){
        tables = db.tables();
    }else{
        tables.clear();
    }
    listmodel->setStringList(tables);
    listTables->setModel(listmodel);
    listTables->update();
}

void MainWin::showStatus(bool state){
    connected = state;
    actConnect->setEnabled(!connected);
    actDisconnect->setEnabled(connected);
    actRun->setEnabled( connected &&
                    (teQuery->toPlainText().trimmed().length() > 0));
    if (state)
        label->setText(tr("Connected: host=%1 dbname=%2")
                        .arg(host) .arg(dbname) );
    else
        label->setText(tr("Disconnected"));

    getTables();
}

void MainWin::on_teQuery_textChanged(){
    actRun->setEnabled( connected &&
          (teQuery->toPlainText().trimmed().length() > 0));
}

void MainWin::on_actRun_triggered(){
    query = new QSqlQuery(db);
    if ( query->exec(teQuery->toPlainText()) ){
        if (query->isSelect())
            showResult();

        teLog->append( tr("<p><b>Query succeeded:</b><br />")
                       + teQuery->toPlainText() );
        sb->showMessage(tr("Query executed successfully"), 3000);
    }else{
        sb->showMessage(tr("Query error"), 3000);
        QMessageBox::critical(
                this, tr("Query Error"),
```

```
                query->lastError().text(),
                QMessageBox::Ok );
    }
}

void MainWin::showResult(){
    if (query->isActive()){
        QSqlQueryModel *model = new QSqlQueryModel(tvResult);
        model->setQuery(*query);
        tvResult->setModel(model);
        tvResult->resizeRowsToContents();
    }
}

void MainWin::on_listTables_doubleClicked(const QModelIndex& index){
    if (index.isValid()){
        showTable(listTables->model()->data(index).toString());
    }
}

void MainWin::showTable(const QString& name){
    QSqlTableModel *model = new QSqlTableModel(tvResult);
    model->setEditStrategy(QSqlTableModel::OnRowChange);
    model->setTable(name);
    model->setSort(0, Qt::AscendingOrder);
    model->select();
    if (model->lastError().type() != QSqlError::NoError){
        sb->showMessage(tr("Query error"), 3000);
        QMessageBox::critical(
                this, tr("Query Error"),
                model->lastError().text(),
                QMessageBox::Ok );
    }
    tvResult->setModel(model);
    tvResult->setEditTriggers( QAbstractItemView::DoubleClicked
                            | QAbstractItemView::EditKeyPressed );
    tvResult->resizeRowsToContents();
}
```

В тексте основной программы, как обычно, требуется создать экземпляр приложения, определить кодек, создать разработанное нами окно и вывести его на экран (листинг 18.19).

Листинг 18.19. Файл chap18/sqlquery/main.cpp

```
#include <QtGui>
#include <QtSql>
#include "mainwin.h"

int main(int argc, char *argv[]) {

    QApplication app(argc, argv);
    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);
    QTextCodec::setCodecForCStrings(codec);

    MainWin mw;
    mw.show();

    return app.exec();
}
```

В файле описания проекта *.pro нужно не забыть подключить модуль `QtSql`, как показано в листинге 18.20.

Листинг 18.20. Файл chap18/sqlquery/sqlquery.pro

```
TEMPLATE = app
TARGET =
DEPENDPATH += .
INCLUDEPATH += .
QT += sql
HEADERS += mainwin.h sqltableview.h connectdlg.h
FORMS += connectdlg.ui sqlquerywin.ui
SOURCES += main.cpp mainwin.cpp sqltableview.cpp connectdlg.cpp
```

Внешний вид получившегося приложения показан на рис. 18.9.

SQL-запросы, использованные для создания таблиц и заполнения их данными, приведены в листинге 18.21.

Листинг 18.21

```
create table employee (   -- работники
  id integer primary key, -- целочисленный идентификатор
  name varchar(50),       -- имя
```

```
  department integer,      -- отдел (номер отдела из таблицы department)
  salary number(14,2),     -- оклад
  married logical);        -- семейное положение

create table department ( -- отделы
  id integer primary key, -- целочисленный идентификатор (номер отдела)
  name varchar(35));       -- название отдела

insert into department(id, name) values(1, 'Дирекция');
insert into department(id, name) values(2, 'Бухгалтерия');
insert into department(id, name) values(3, 'Отдел кадров');

insert into employee(id, name, department, salary, married)
  values(1, 'Винни-Пух', 1, 1234.56, 1);

insert into employee(id, name, department, salary, married)
  values(2, 'Пятачок', 3, 6543.21, 0);

insert into employee(id, name, department, salary, married)
  values(3, 'Ослик Иа', 2, 23.45, 0);

select * from employee;
```

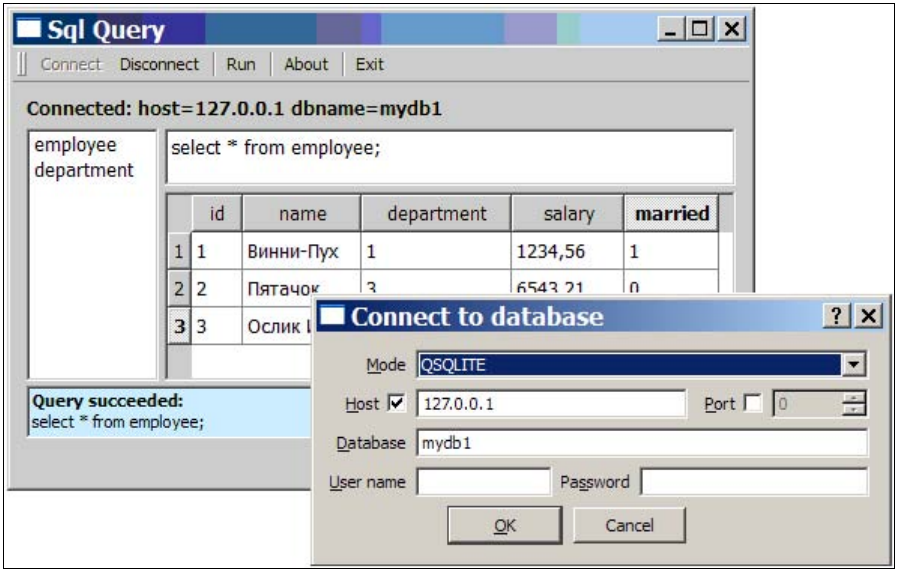


Рис. 18.9. Консоль запросов

Замечание

Исправьте текст программы так, чтобы при создании/удалении таблицы базы данных соответствующим образом сразу обновлялся список таблиц в левой части диалога.

18.5. Связывание таблиц

Обычно базы данных разрабатываются таким образом, что в некоторых полях таблицы приходится хранить не сами данные, а ссылку на запись, находящуюся в другой таблице. Примером может служить поле `department` таблицы `employee` из предыдущего раздела. Вместо названия отдела мы храним его номер, а соответствие номеров и названий (а также, возможно, другие параметры отделов) задаются в отдельной таблице `department`.

Но при такой организации данных возникает следующая проблема: в поле `department` хранится номер отдела, но нам хотелось бы отображать его название. Более того, при редактировании поля `department` удобно видеть список отделов вместо того, чтобы вспоминать, какие у них номера. Мы бы выбирали нужный отдел из списка, а соответствующий номер записывался бы автоматически в поле `department`.

Такие возможности предоставляют классы `QSqlRelation`, `QSqlRelationalTableModel` и `QSqlRelationalDelegate`.

ЗАДАЧА. Разработать модель и вид для таблицы базы данных, имеющей связи с двумя другими таблицами (рис. 18.10).

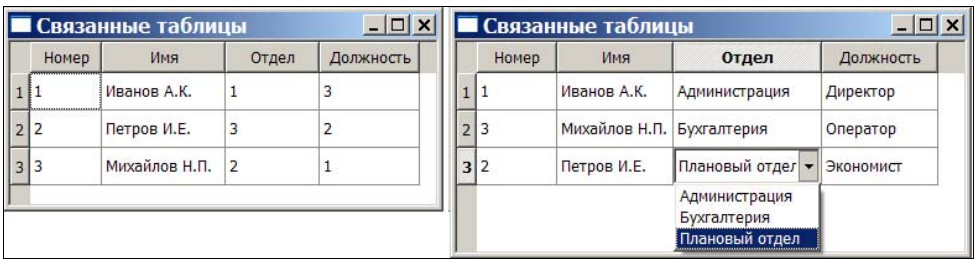


Рис. 18.10. Не правда ли, правая таблица выглядит намного лучше левой?

Решение иллюстрирует листинг 18.22.

Листинг 18.22. Файл chap18/db05/db05.cpp

```
#include <QtGui>
#include <QtSql>
```

```
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);

    QSqlDatabase db = QSqlDatabase::addDatabase("QSQLITE");
    db.setDatabaseName("mydb1"); // Имя файла базы данных
    // db.setUserName("...");
    // db.setPassword("...");
    db.open();

    // Создаем таблицы и заполняем их данными:
    QSqlQuery query;
    query.exec("create table employee( "
               " id int primary key, "
               " name varchar(20), "
               " department int, "
               " position int)" );
    query.exec(QObject::tr(
         "insert into employee values(1, 'Иванов А.К.', 1, 3)"));
    query.exec(QObject::tr(
         "insert into employee values(2, 'Петров И.Е.', 3, 2)"));
    query.exec(QObject::tr(
         "insert into employee values(3, 'Михайлов Н.П.', 2, 1)"));

    query.exec("create table department(id int, name varchar(20))");
    query.exec(QObject::tr(
         "insert into department values(1, 'Администрация')"));
    query.exec(QObject::tr(
         "insert into department values(2, 'Бухгалтерия')"));
    query.exec(QObject::tr(
         "insert into department values(3, 'Плановый отдел')"));

    query.exec("create table position(id int, name varchar(20))");
    query.exec(QObject::tr(
         "insert into position values(1, 'Оператор')"));
    query.exec(QObject::tr(
         "insert into position values(2, 'Экономист')"));
    query.exec(QObject::tr(
         "insert into position values(3, 'Директор')"));

    QSqlRelationalTableModel model;
    model.setTable("employee");
```

```
    model.setSort(1, Qt::AscendingOrder);
    model.setEditStrategy(QSqlTableModel::OnFieldChange);

    // Задаем связи между таблицами:
    model.setRelation(2, QSqlRelation("department", "id", "name"));
    model.setRelation(3, QSqlRelation("position", "id", "name"));

    // Задаем заголовки столбцов таблицы:
    model.setHeaderData(0, Qt::Horizontal, QObject::tr("Номер"));
    model.setHeaderData(1, Qt::Horizontal, QObject::tr("Имя"));
    model.setHeaderData(2, Qt::Horizontal, QObject::tr("Отдел"));
    model.setHeaderData(3, Qt::Horizontal, QObject::tr("Должность"));

    model.select();

    // Создаем представление (вид) таблицы:
    QTableView view;
    view.setModel(&model);

    // Создаем делегатов для связанных ячеек:
    view.setItemDelegate(new QSqlRelationalDelegate(&view));

    view.setWindowTitle(QObject::tr("Связанные таблицы"));
    view.show();
    return app.exec();
}
```

Чтобы не отвлекаться, мы не стали проверять, существуют ли уже нужные таблицы в базе данных и есть ли в них какие-то данные. Поэтому если перед очередным запуском программы не удалить файл `mydb1`, то мы рискуем получить множество одинаковых записей.

18.6. Выполнение запросов в отдельных потоках

Выполнение запросов к базе данных при использовании стандартных SQL-моделей библиотеки Qt производится в синхронном (блокирующем) режиме. Это означает, что пока не будет получен результат очередного запроса, интерфейс программы блокируется. Разумеется, при работе с большими наборами данных такое поведение не слишком радует. Разумнее было бы выполнять обмен информацией между моделью таблицы и базой данных в отдельном потоке, асинхронно обновляя данные модели по мере поступления новой информации от базы данных.

Поскольку, как уже отмечалось, объект `QSqlDatabase` нельзя передать в другой поток, в каждом потоке придется устанавливать отдельное соединение со своим именем:

```
m_database = QSqlDatabase::addDatabase( "QPSQL", "MyConnection1" );
```

Синхронизировать потоки можно с помощью стандартных примитивов: `QMutex`, `QSemaphore` и `QWaitCondition`, но наиболее удобен механизм сигналов и слотов. Поскольку межпотоковые передачи сообщений задействуют события `QEvent`, в каждом потоке потребуется запустить свой цикл обработки событий. Другими словами, в методе `QTread::run()` необходимо вызвать метод `QThread::exec()`.

Если параметрами сигнала должны быть наборы записей вида `QList<QSqlRecord>`, то данный тип придется сначала зарегистрировать:

```
qRegisterMetaType<QList<QSqlRecord>>( "QList<QSqlRecord>" );
```

ЗАДАЧА. Выполнить SQL-запросы в отдельных потоках, передавая результат в основной поток с помощью сигналов.

Решение. В листингах 18.23–18.27 приведен текст программы, а на рис. 18.11 показан результат ее работы.

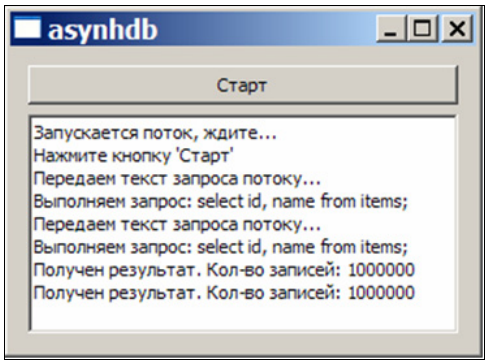


Рис. 18.11. Запросы выполняются в отдельных потоках

Листинг 18.23. Файл chap18/asyncdb/querythread.h

```
#include <QtGui>
#include <QtSql>

class Worker; // Предварительное описание рабочего класса

class QueryThread : public QThread { // Класс потока
  Q_OBJECT
```

```
public:
    QueryThread(QObject *parent = 0);
    ~QueryThread();
    void execute( const QString& query ); // Выполнить запрос
signals:
    void progress( const QString& msg );
    void ready(bool);
    void results( const QList<QSqlRecord>& records );
protected:
    void run();
signals:
    void executefwd( const QString& query );
private:
     Worker* m_worker;
};

//-------------------------------------
class Worker : public QObject { // Этот класс выполняет всю работу
  Q_OBJECT
public:
    Worker(const QString& name,
           QObject* parent = 0);
    ~Worker();
public slots:
    void slotExecute( const QString& query );
signals:
    void results( const QList<QSqlRecord>& records );
private:
    QSqlDatabase m_database;  // Соединение с БД
    QString m_name;           // Имя соединения с БД
};
```

Листинг 18.24. Файл chap18/asyncdb/querythread.cpp

```
#include <QtGui>
#include <QDebug>
#include <QtSql>
#include "querythread.h"

#define DATABASE_USER "user1"     // Имя пользователя
#define DATABASE_PASS "password"  // Пароль
#define DATABASE_NAME "data.db"   // База данных
#define DATABASE_HOST ""          // Хост
```

```
#define DATABASE_DRIVER "QSQLITE" // Драйвер БД
#define SAMPLE_RECORDS 1000000    // Количество записей в таблице

Worker::Worker( const QString& name,
                QObject* parent )
    : QObject( parent ), m_name(name) {
    m_database = QSqlDatabase::addDatabase(
        DATABASE_DRIVER,
        m_name );
    m_database.setDatabaseName( DATABASE_NAME );
    m_database.setHostName( DATABASE_HOST );
    m_database.setUserName( DATABASE_USER );
    m_database.setPassword( DATABASE_PASS );
    if ( !m_database.open() ) {
        qWarning() << "Error:"
                   << m_database.lastError().text();
        return;
    }

    if ( !m_database.tables().contains("items")){ //Если таблицы нет
        m_database.exec(   // Создаем таблицу items
            "CREATE TABLE items(id int, name varchar(50));" );
        m_database.transaction();
        QSqlQuery query(m_database);
        query.prepare("INSERT INTO items (id, name) "
                      "VALUES (?, ?)");
        for ( int i = 0; i < SAMPLE_RECORDS; ++i ) {
            query.addBindValue(i);
            query.addBindValue(QString::number(i));
            query.exec();  // Заполняем данными
        }
        m_database.commit();
    }
}

Worker::~Worker() { }

void Worker::slotExecute( const QString& query ) {
    QList<QSqlRecord> recs;  // Список записей — результат запроса
    QSqlQuery sql( query, m_database );
    while ( sql.next() ) {
        recs.append( sql.record() );
    }
```

```
    emit results( recs ); // Высылаем сигнал с результатом запроса
}

//-----------------------------------------
QueryThread::QueryThread(QObject *parent)
    : QThread(parent) { }

QueryThread::~QueryThread() {
    delete m_worker;
}

void QueryThread::execute( const QString& query ) {
    emit executefwd( query ); // Высылаем сигнал с текстом запроса
}

void QueryThread::run() { // Тело потока
    emit ready(false); // Пока не готов
    emit progress( tr("Запускается поток, ждите..." ));

    m_worker = new Worker("MyConnection1");
    connect( this, SIGNAL( executefwd( const QString& ) ),
             m_worker, SLOT( slotExecute( const QString& ) ) );

    // Зарегистрируем тип, чтобы передать результат с сигналом:
    qRegisterMetaType< QList<QSqlRecord> >( "QList<QSqlRecord>" );

    connect( m_worker, SIGNAL( results( const QList<QSqlRecord>& ) ),
             this, SIGNAL( results( const QList<QSqlRecord>& ) ) );

    emit progress( tr("Нажмите кнопку 'Старт'") );
    emit ready(true); // Теперь готов

    exec(); // Чтобы работал межпоточный механизм сигналов/слотов
}
```

Листинг 18.25. Файл chap18/asyncdb/win.h

```
#include <QtGui>
#include <QtSql>

class QueryThread;

class Win : public QWidget {
    Q_OBJECT
```

```
public:
    Win( QWidget* parent = 0 );
    ~Win();
signals:
    void exec( const QString& );
private slots:
    void slotStart();
    void slotResults( const QList<QSqlRecord>& records );
private:
    void dispatch( const QString& query );

    QueryThread* m_querythread;
    QPushButton *m_btnStart; // Кнопка "Старт"
    QTextBrowser *m_log;     // Журнал для отображения событий
};
```

Листинг 18.26. Файл chap18/asyncdb/win.cpp

```
#include "win.h"
#include "querythread.h"
#include <QDebug>

Win::Win( QWidget* parent )
    : QWidget(parent) {

    m_querythread = new QueryThread();

    QVBoxLayout *lay = new QVBoxLayout(this);

    m_btnStart = new QPushButton(tr("Старт"), this);
    lay->addWidget(m_btnStart);

    m_log = new QTextBrowser(this);
    lay->addWidget(m_log);

    connect( m_btnStart,
             SIGNAL( clicked() ),
             this,
             SLOT( slotStart() ) );
    connect( m_querythread,
             SIGNAL( progress(const QString&) ),
             m_log,
             SLOT( append(const QString&) ) );
```

```
    connect( m_querythread,
             SIGNAL( ready(bool) ),
             m_btnStart,
             SLOT( setEnabled(bool) ) );
    connect( m_querythread,
             SIGNAL( results( const QList<QSqlRecord>& ) ),
             this,
             SLOT( slotResults( const QList<QSqlRecord>& ) ) );

    m_querythread->start();
}

Win::~Win() {
    m_querythread->quit();
    m_querythread->wait();
    delete m_querythread;
}

void Win::slotStart() {  // При нажатии кнопки "Старт"
    m_log->append( tr("Передаем текст запроса потоку...") );
    dispatch( "select id, name from items;" );
}

void Win::dispatch( const QString& query ) { // Новое задание
    m_log->append( tr("Выполняем запрос: %1").arg(query) );
    m_querythread->execute( query );
}

void Win::slotResults( const QList<QSqlRecord>& records ) {
    m_log->append( tr("Получен результат. Кол-во записей: %1")
                   .arg( records.count() ) );
}
```

Листинг 18.27. Файл chap18/asyncdb/main.cpp

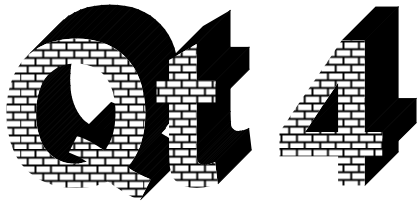
```
#include <QApplication>
#include "win.h"

int main( int argc, char **argv ){
  QApplication app( argc, argv );
  QTextCodec *codec = QTextCodec::codecForName("CP1251");
  QTextCodec::setCodecForTr(codec);
```

```
  Win win;
  win.show();
  return app.exec();
}
```

Обратите внимание: во время выполнения запроса интерфейс нашей программы не блокируется. Мы несколько раз подряд можем нажать кнопку **Старт**, пока ждем прихода результата. При каждом нажатии выполняется слот `Win::slotStart()`, который с помощью метода `Win::dispatch()` вызывает метод потока `QueryThread::execute()`, передавая в качестве параметра текст запроса. Но объект `QueryThread` выполняет этот запрос не сам, он только высылает сигнал `executefwd()` с текстом запроса. Данный сигнал передается объекту `Worker`, вызывая его метод `Worker::slotExecute()`, который выполняется внутри потока `QueryThread`. Только теперь создается объект `QSqlQuery`, выполняется текст запроса, а результат (список записей `QSqlRecord`) высылается с сигналом `Worker::results()`. Получив этот сигнал, объект `QueryThread` высылает свой сигнал с тем же названием, в результате вызывается слот `Win::slotResults()` и мы видим сообщение на экране.

Вы спросите: зачем нужен класс `Worker`, почему бы не выполнить запрос непосредственно из `QueryThread`? Дело в том, что этот объект создается внутри основного потока, в конструкторе класса `Win`. Если мы начнем выполнение запроса в каком-либо методе `QueryThread`, то цикл обработки событий основного потока окажется заблокированным, а именно этого мы хотели избежать. Нам нужен объект, который принадлежит другому потоку, вот почему появился `Worker`. Он создается не в конструкторе `QueryThread`, а в его методе `run()`, т. е. в отдельном потоке, поэтому работает со своей собственной очередью событий.

ГЛАВА 19



Средства XML

Мудрено пишут только о том, чего не понимают.
Василий Осипович Ключевский

XML (*Extensible Markup Language*) — расширяемый язык разметки, удобный для представления структурированной информации. В листинге 19.1 приведен пример файла XML, в котором хранится одна запись книги регистрации документов на некотором предприятии.

Листинг 19.1. Пример документа в формате XML

```
<?xml version="1.0" encoding="windows-1251"?>
<!-- Комментарий -->
   <doc name="Приказ №13">
      <author name="Директор Васечкин И.О." />
      Назначить Петрова А.К. начальником отдела ИТ с 13.12.2007.
   </doc>
```

Здесь имеется один элемент верхнего уровня (заголовок документа), ограниченный парой тегов `<doc>` и `</doc>`, в который вложены еще два элемента: сведения об авторе (тег `<author>`) и собственно текст документа. Элементы `<doc>` и `<author>` имеют каждый по одному атрибуту. Оба атрибута имеют имя `name`, но в первом записано название документа, а во втором — имя автора документа. Тег `<author>`, в отличие от `<doc>`, не имеет пары `</author>`, поскольку мы храним всю необходимую информацию об авторе в виде значения атрибута `name`.

Замечание

В отличие от HTML, имена тегов и атрибутов XML чувствительны к регистру.

Для работы с классами XML в библиотеке Qt имеется модуль `QtXml`, в котором реализованы два основных подхода к обработке XML-документов: DOM

и SAX. Для использования данного модуля в программе необходимо подключить библиотеку QtXml, а в файл описания проекта *.pro добавить строку

```
QT += xml
```

19.1. DOM API

Объектная модель документа (Document Object Model — DOM) представляет файл XML как дерево объектов в памяти компьютера. Такой подход позволяет легко читать и изменять XML-документ, но неэкономно расходует память.

Сам документ XML представлен классом QDomDocument, каждый узел дерева документа — классом QDomNode. В качестве узла может выступать элемент — класс QDomElement, атрибут элемента — класс QDomAttr, текст QDomText, комментарий QDomComment и т. д.

ЗАДАЧА. Создать XML-документ, текст которого приведен в листинге 19.1.

Решение иллюстрирует листинг 19.2.

Листинг 19.2. Создание файла XML средствами DOM API

```
#include <iostream>
#include <QtCore>
#include <QtXml>

#define  tr  QObject::tr
#define STR(s) ((outcodec->fromUnicode(s)).constData())

int main( int argc, char **argv ) {
  QCoreApplication app(argc, argv);
  QTextCodec *incodec = QTextCodec::codecForName("CP1251");
  QTextCodec::setCodecForTr(incodec);

  #ifdef Q_WS_WIN
  QTextCodec *outcodec = QTextCodec::codecForName("CP866");
  QTextCodec::setCodecForLocale(outcodec);
  #endif

  QDomDocument document;
  QDomElement d = document.createElement( "doc" );
  d.setAttribute( "name", tr("Приказ №13"));
  QDomElement a = document.createElement( "author" );
  a.setAttribute( "name", tr("Директор Васечкин И.О."));
```

```
23    QDomText text = document.createTextNode(
24      tr("Назначить Петрова А.К. ...") );
25    document.appendChild( d );
26    d.appendChild( a );
27    d.appendChild( text );
28    QDomNode xmlNode = document.createProcessingInstruction(
29      "xml",
30      "version=\"1.0\" encoding=\"windows-1251\"");
31    document.insertBefore(xmlNode, document.firstChild());
32
33    QFile file( "simple.xml" );
34    if( !file.open( QIODevice::WriteOnly | QIODevice::Text ) ){
35      std::cout << STR( "\nНе могу открыть файл для записи!" );
36      return -1;
37    }
38    QTextStream out( &file );
39    document.save(out, 3);  // или  out << document.toString();
40    file.close();
41    return 0;
42 }
```

Пояснения к программе:

- (5, 6) — макросы, облегчающие вывод кириллицы в консольное окно;
- (13–16) — кодировка консоли Windows отличается от системной;
- (18) — создали документ XML;
- (19, 20) — создали новый элемент с именем `doc` (документ) и установили значение его атрибута `name` (имя);
- (21, 22) — создали новый элемент с именем `author` (автор) и установили значение его атрибута `name`;
- (23, 24) — создали текстовый узел для хранения текста документа;
- (25–27) — добавили к документу XML элемент `doc` и два его дочерних элемента: сведения об авторе и текст;
- (28–31) — добавили в начало документа заголовок XML, устанавливающий версию и кодировку (иначе по умолчанию используется кодировка `UTF-8`);
- (33–40) — открыли файл и записали в него документ XML.

Не забудьте перед компиляцией в файл проекта *.pro добавить строки:

```
QT += xml
win32: CONFIG += console
```

ЗАДАЧА. Прочитать документ XML из файла (рис. 19.1).

Решение иллюстрирует листинг 19.3.

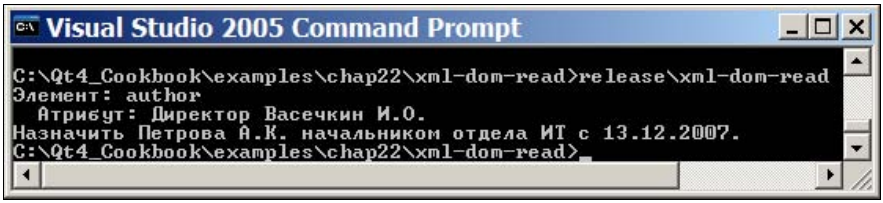


Рис. 19.1. Пример разбора файла XML

Листинг 19.3. Чтение файла XML средствами DOM API

```
#include <iostream>
#include <QtCore>
#include <QtXml>

#define tr  QObject::tr
#define STR(s) ((outcodec->fromUnicode(s)).constData())

int main( int argc, char **argv ) {
   QCoreApplication app(argc, argv);
   QTextCodec *incodec = QTextCodec::codecForName("CP1251");
   QTextCodec::setCodecForTr(incodec);

   #ifdef Q_WS_WIN
   QTextCodec *outcodec = QTextCodec::codecForName("CP866");
   QTextCodec::setCodecForLocale(outcodec);
   #endif

   QFile file( "simple.xml" );
   if( !file.open( QIODevice::ReadOnly | QIODevice::Text ) ) {
      std::cout << STR(tr("Ошибка открытия файла 'simple.txt'") );
      return -1;
   }
   QDomDocument document;
   if( !document.setContent( &file ) ) {
      std::cout << STR(tr("Ошибка при разборе файла") );
      file.close();
      return -2;
   }
   file.close();
```

```
    QDomElement documentElement = document.documentElement();
    QDomNode node = documentElement.firstChild();
    while( !node.isNull() ) {
       if( node.isElement() ) {
          QDomElement element = node.toElement();
          std::cout << STR(tr("Элемент: "))
                    << STR(element.tagName()) << std::endl;
          std::cout << STR(tr("  Атрибут: "))
                    << STR(element.attribute("name", "---"))
                    << std::endl;
       }
       if( node.isText() ) {
          QDomText text = node.toText();
          std::cout << STR(text.data());
       }
       node = node.nextSibling();
    }
    return 0;
}
```

ЗАДАЧА. Прочитать и модифицировать файл, текст которого приведен в листинге 19.4.

Листинг 19.4

```
<?xml version='1.0' encoding='utf-8'?>
<Notebook>
  <NotebookRecord Surname="Сидоров" Name="Иван"
                  Middlename="Петрович" Phone="12-23-34" />
  <NotebookRecord Surname="Матросов" Name="Семен"
                  Middlename="Иванович" Phone="98-87-65" />
  <NotebookRecord Surname="Зайцев" Name="Максим"
                  Middlename="Петрович" />
</Notebook>
```

Решение иллюстрирует листинг 19.5.

Листинг 19.5

```
#include <QtGui>
#include <QtXml>
#define tr  QObject::tr
```

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

    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);

    QTextEdit te;
    te.show();

    QDomDocument doc("mydoc");
    QFile file("example.xml");
    if (!file.open(QFile::ReadOnly | QFile::Text)){
        te.append( tr(
            "Не могу открыть файл 'example.xml'"));
        return 1;
    }
    if (!doc.setContent(&file)) {
        file.close();
        te.append( tr(
            "Ошибка при чтении файла 'example.xml'"));
        return 2;
    }
    file.close();

    QDomElement root = doc.documentElement();
    if ( root.tagName() != tr("Notebook") ) {
        te.append( tr(
            "Неправильный формат файла!"));
        return 3;
    }

    QDomNode n = root.firstChild();
    while( !n.isNull() ) {
        QDomElement e = n.toElement();
        if( !e.isNull() &&
            (e.tagName() == tr("NotebookRecord"))){

            te.append(
                e.attribute(tr("Name")) + " " +
                e.attribute(tr("Middlename")) + " " +
                e.attribute(tr("Surname")) );
        }
        n = n.nextSibling();
    }
```

```
    QDomElement elem = doc.createElement( tr(
        "NotebookRecord") );
    elem.setAttribute( tr("Name"),
                       tr("Максим") );
    elem.setAttribute( tr("Middlename"),
                       tr("Петрович") );
    elem.setAttribute( tr("Surname"),
                       tr("Зайцев") );
    root.appendChild(elem);
    te.append( tr("Добавили 1 запись.") );

    if (!file.open(QFile::WriteOnly | QFile::Text)){
        te.append( tr(
            "Не могу открыть файл 'example.xml'"));
        return 4;
    }

    QTextStream ts(&file);
    doc.save(ts, 4);
    file.close();

    return app.exec();
}
```

19.2. SAX API

Другой подход к чтению (и только к чтению!) документов XML носит название *Simple API for XML* (SAX — простой программный интерфейс для XML). Его достоинство заключается в том, что загрузка в память сразу всего документа не требуется, а возможна работа с отдельными фрагментами. Основа SAX — обработка событий. Событие происходит, когда встречается стартовый или конечный тег языка XML, атрибут элемента или текстовый фрагмент.

ЗАДАЧА. Выполнить разбор документа XML средствами SAX API.

Решение. Самый простой способ — разработать класс на основе `QXmlDefaultHandler`, в котором переопределить следующие виртуальные методы:

- `startDocument()` — вызывается при начале чтения документа XML;
- `endDocument()` — по окончании обработки документа XML;
- `startElement()` — в начале обработки элемента документа XML;
- `endElement()` — в конце обработки элемента документа XML;

- `characters()` — вызывается при чтении данных элемента;
- `fatalError()` — вызывается в случае ошибки.

Эту возможность иллюстрируют листинги 19.6, 19.7 и рис. 19.2.

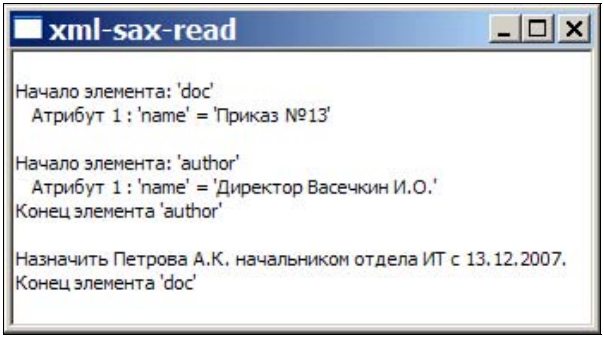


Рис. 19.2. Результат разбора файла XML с помощью SAX API

Листинг 19.6. Файл chap19/xml-sax-read/xml-sax-read.h

```
#include <QtGui>
#include <QtXml>

class MyHandler : public QXmlDefaultHandler {
public:
    MyHandler(QTextEdit *textEdit)
         : QXmlDefaultHandler(),
           te(textEdit) { }
    bool startDocument() {return true;}
    bool endDocument() {return true;}
    bool startElement(
        const QString &namespaceURI,
        const QString &localName,
        const QString &qName,
        const QXmlAttributes &atts );
    bool endElement(
        const QString &namespaceURI,
        const QString &localName,
        const QString &qName );
    bool characters( const QString& s );
    bool fatalError(const QXmlParseException& exception);
private:
     QTextEdit *te;
};
```

Листинг 19.7. Файл chap19/xml-sax-read/xml-sax-read.cpp

```
#include "xml-sax-read.h"

#define tr  QObject::tr

bool MyHandler::startElement( const QString& /* namespaceURI */,
                              const QString& /* localName */,
                              const QString& qName,
                              const QXmlAttributes &atts ){
    te->append(tr("\nНачало элемента: '%1'").arg(qName));
    for( int i=0; i<atts.length(); ++i )
        te->append(tr("   Атрибут %1 : '%2' = '%3'")
                   .arg(i+1)
                   .arg(atts.qName(i))
                   .arg(atts.value(i)) );
    return true;
}

bool MyHandler::endElement( const QString& /* namespaceURI */,
                            const QString& /* localName */,
                            const QString& qName ) {
    te->append(tr("Конец элемента '%1'\n").arg(qName));
    return true;
}

bool MyHandler::characters(const QString& s) {
    if (s.trimmed().length() > 0)
        te->append(s);
    return true;
}

bool MyHandler::fatalError(const QXmlParseException& exception){
     te->append(tr("\nОшибка: строка %1, позиция %2:\n%3")
                .arg(exception.lineNumber())
                .arg(exception.columnNumber())
                .arg(exception.message() ));
     return false;
}

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

```
    QTextCodec *incodec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(incodec);

    QTextEdit te;
    te.show();

    QFile file( "simple.xml" );
    if( !file.open( QIODevice::ReadOnly | QIODevice::Text ) ) {
        te.append("Ошибка открытия файла");
        return -1;
    }

    QXmlInputSource source( &file );
    MyHandler handler(&te);
    QXmlSimpleReader reader;
    reader.setContentHandler( &handler );
    reader.setErrorHandler( &handler );
    reader.parse( source );
    file.close();

    return app.exec();
}
```

19.3. Класс *QStreamReader*

В библиотеке Qt имеется класс `QStreamReader`, который обеспечивает последовательный разбор документа XML и работает быстрее, чем SAX API.

ЗАДАЧА. Прочитать содержимое XML-документа (листинг 19.8) с помощью `QStreamReader`.

Листинг 19.8

```
<?xml version="1.0" encoding="windows-1251"?>
<docs>
<doc name="Приказ №13">
   <author name="Директор Васечкин И.О." />
     Назначить Петрова А.К. начальником отдела ИТ с 13.04.2008.
</doc>
<doc name="Приказ №14">
   <author name="Начальник отдела ИТ Петров А.К." />
     Уволить Васечкина И.О. по собственному желанию с 14.04.2008.
</doc>
</docs>
```

Решение иллюстрируют листинги 19.9 и 19.10.

Листинг 19.9. Файл chap19/streamreader/main.cpp

```
#include <QtCore>
#include <QXmlStreamReader>

int main(int argc, char *argv[]){
    QCoreApplication app(argc, argv);
    QTextCodec *incodec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(incodec);

    #ifdef Q_WS_WIN
    QTextCodec *outcodec = QTextCodec::codecForName("CP866");
    QTextCodec::setCodecForLocale(outcodec);
    #endif
    QStringList args = QCoreApplication::arguments();

    if (args.count() < 2) {
        qDebug() << QObject::tr("Надо указать имя XML-файла!");
        return 1;
    }

    QFile file(args[1]); // Имя файла — параметр командной строки
    if (!file.open(QFile::ReadOnly | QFile::Text)) {
        qDebug() << QObject::tr("Не могу прочитать файл!")
                 << qPrintable(file.errorString());
        return 1;
    }
    QXmlStreamReader reader(&file);
    reader.readNext(); // Прочитать очередной токен
    qDebug() << "0:" << reader.tokenString() // Тип токена,
             << reader.name().toString()     // его имя
             << reader.text().toString().trimmed(); // и текст
    int n = 0;
    while (!reader.atEnd()) { // Пока не конец файла
        n++;
        qDebug() << n << ":"
                 << reader.tokenString()           // Тип токена,
                 << reader.name().toString()            // его имя
                 << reader.text().toString().trimmed(); // и текст
        reader.readNext(); // Перейти к следующему токену
    }
```

```
    file.close();  // Закрыть файл
    if (reader.hasError()) { // Если были ошибки при разборе XML
        qDebug() << QObject::tr("Неверный формат файла!")
                 << qPrintable(reader.errorString());
        return 2;
    } else if (file.error() != QFile::NoError) { // Если ошибка чтения
        qDebug() << QObject::tr("Не могу прочитать файл!")
                 << qPrintable(file.errorString());
        return 3;
    }
    return 0;
}
```

Листинг 19.10. Файл chap19/streamreader/streamreader.pro

```
TEMPLATE      = app
QT           -= gui
QT           += xml
SOURCES       = main.cpp
win32: TARGET = ../streamreader
win32: CONFIG += console
```

Для запуска программы нужно в командной строке после имени исполняемого файла указать имя файла XML, например:

```
streamreader simple.xml
```

В результате работы программы на консольный экран будет выведен текст листинга 19.11.

Листинг 19.11

```
0: "StartDocument" "" ""
1 : "StartDocument" "" ""
2 : "StartElement" "docs" ""
3 : "Characters" "" ""
4 : "StartElement" "doc" ""
5 : "Characters" "" ""
6 : "StartElement" "author" ""
7 : "EndElement" "author" ""
8 : "Characters" "" "Назначить Петрова А.К. начальником отдела ИТ
с 13.04.2008."
9 : "EndElement" "doc" ""
10 : "Characters" "" ""
```

```
11 : "StartElement" "doc" ""
12 : "Characters" "" ""
13 : "StartElement" "author" ""
14 : "EndElement" "author" ""
15 : "Characters" "" "Уволить Васечкина И.О. по собственному желанию
с 14.04.2008."
16 : "EndElement" "doc" ""
17 : "Characters" "" ""
18 : "EndElement" "docs" ""
```

Теперь если вместо отладочных сообщений добавить в текст программы содержательную обработку прочитанных элементов файла XML, то можно, например, построить на экране дерево `QTreeWidget`, в котором названия приказов (атрибут `name` токена `doc`) соответствуют группам, а при их раскрытии будут отображаться имя автора и текст приказа. Оставляем эту несложную доработку читателю.

19.4. Модуль *QtXmlPatterns*

Начиная с релиза 4.4, в библиотеке Qt появился модуль `QtXmlPatterns`, предоставляющий средства для работы с языками запросов XQuery и XPath. Для его использования необходимо в тексте программы подключить заголовочный файл `<QtXmlPatterns>`, а в файле описания проекта добавить строку

```
QT += xmlpatterns
```

Кроме того, теперь имеется утилита командной строки `xmlpatterns`, которая позволяет отлаживать запросы на языке XQuery.

ЗАДАЧА. Прочитать XML-документ (листинг 19.8) с помощью запроса XQuery.

Решение иллюстрируют листинги 19.12, 19.13 и рис. 19.3.

Листинг 19.12. Файл chap19/xquery/xquery.cpp

```
#include <QtGui>
#include <QtXmlPatterns>

int main(int argc, char *argv[]){
    QApplication app(argc, argv);
    QTextCodec *incodec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(incodec);
```

```
    #ifdef Q_WS_WIN // Для кодировки консоли в Windows
    QTextCodec *outcodec = QTextCodec::codecForName("CP866");
    QTextCodec::setCodecForLocale(outcodec);
    #endif

    QXmlQuery query; // Запрос XQuery

    QFile source("simple.xml"); // Источник данных — файл XML
    source.open(QIODevice::ReadOnly);

    // Присваиваем значение внешней переменной запроса XQuery:
    query.bindVariable("inputDocument", &source);

    // Текст запроса XQuery:
    query.setQuery(QObject::tr(
        "declare variable $inputDocument external;" // Внешняя переменная
        "<html><body><h1>Тексты приказов:</h1>"
        "<font size='4'><ul> "
        "{for $x in doc($inputDocument)/docs/doc "
        "  order by $x "
        "  return <li>{data($x)}</li> "
        "}</ul></font></body></html>" ) );

    if(!query.isValid()) // Если есть ошибки в синтаксисе запроса
        return 1;

    QByteArray outArr; // Для вывода результата запроса
    QBuffer buffer(&outArr);
    buffer.open(QIODevice::ReadWrite);

    QXmlFormatter formatter(query, &buffer); // Для форматирования
    if(!query.evaluateTo(&formatter)) // Выполняем запрос
        return 2;      // Если была ошибка при выполнении

    buffer.close();
    QString result = QString::fromUtf8(outArr.constData());
    qDebug() << result;   // Вывели результат на консоль
    QTextEdit te(result); // Вывели результат в TextEdit
    te.show(); // Показали форматированный результат выполнения запроса

    return app.exec();
}
```

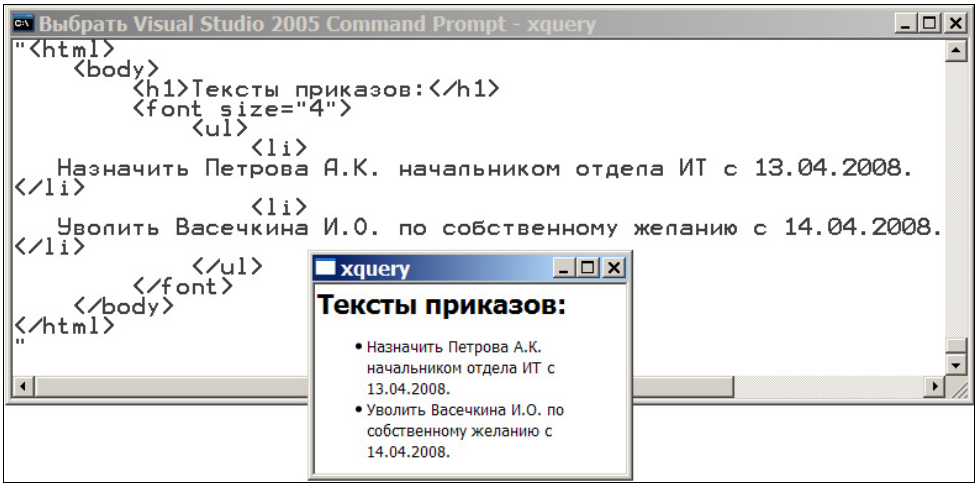


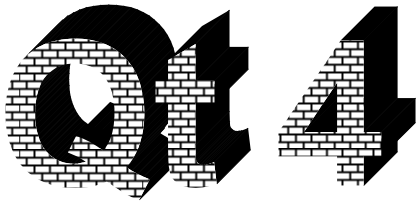
Рис. 19.3. Результат запроса XQuery в виде кода HTML и в окне `QTextEdit`

Листинг 19.13. Файл chap19/xquery/xquery.pro

```
TEMPLATE = app
win32: TARGET = ../xquery
win32: CONFIG += console
DEPENDPATH += .
INCLUDEPATH += .
QT += xmlpatterns
SOURCES += xquery.cpp
```

В спецификацию XQuery 1.0 также включен язык XPath 2.0, который поддерживается модулем `QtXmlPatterns`.

ГЛАВА 20



Локальные сети и Интернет

Знание бывает двух видов. Мы сами знаем предмет — или же знаем, где найти о нем сведения.
Бенджамин Франклин

Для передачи информации между компьютерами используются *сокеты* двух типов: работающие по протоколу UDP (*User Datagram Protocol*, без установления соединения) и работающие по протоколу TCP (*Transmission Control Protocol*, с установлением соединения). Сокет определяется именем (или IP-адресом) компьютера и номером порта.

Сокет TCP может быть *серверным* и *клиентским*. Первый прослушивает порт с некоторым номером, ожидая подключения со стороны клиентов, а второй на компьютере клиента, наоборот, пытается подключиться к серверному сокету. Приняв запрос на подключение, процесс-сервер создает для нового клиента со своей стороны также клиентский сокет, через который продолжает общаться с клиентом, а серверный сокет освобождается для ожидания новых запросов на подключение от других клиентов.

Сокеты могут работать в *блокирующем* и *неблокирующем* режиме. В первом случае выполнение потока, в котором работает сокет, блокируется до окончания выполнения очередной операции. Во втором режиме используется программирование, управляемое событиями.

В библиотеке Qt для работы с сокетами имеется модуль `QtNetwork`. Сокеты UDP представлены классом `QUdpSocket`, а сокеты TCP — классами `QTcpSocket` и `QTcpServer`. Те и другие наследуют многие свойства и методы от базового класса `QAbstractSocket`. Главное различие между ними заключается в том, что `QUdpSocket` реализует ненадежную передачу данных, не ориентированную на установление соединения между отправителем и получателем сообщения, в то время как класс `QTcpSocket`, наоборот, обеспечивает надежную передачу данных с предварительной установкой соединения между источником сообщения и адресатом.

Далее приведен алгоритм работы с серверным сокетом TCP в неблокирующем режиме.

- Сначала в программе нужно создать экземпляр класса `QTcpServer` (или производный от него):

```
QTcpServer *tcpServer = new QTcpServer(parent);
```

- Затем попытаться перевести серверный сокет в рабочее ("слушающее") состояние с помощью метода `listen()`; в случае неудачи сообщить пользователю причину ошибки (листинг 20.1).

Листинг 20.1

```
if ( ! tcpServer->listen(QHostAddress::Any, 5678) ) {
   QMessageBox::critical(parent, tr("TCP Server"),
      tr( "Не могу запустить сервер: %1." )
          .arg(tcpServer->errorString() ),
      QMessageBox::Ok );
   close();
   return;
}
```

Два необязательных параметра метода `listen()` — это адрес `QHostAddress` и номер порта компьютера, от которого ожидаются запросы на подключение. По умолчанию сервер принимает запросы от любого клиента, а номер порта выбирается автоматически, его можно узнать с помощью метода `serverPort()`. Но чаще используют какой-нибудь фиксированный номер порта, заранее известный всем клиентам.

- После этого следует установить слот-обработчик для сигнала `newConnection()`, который генерируется, когда клиент пытается подключиться к серверу. Назовем соответствующую функцию, например, `addConnection()`, тогда:

```
connect ( tcpServer, SIGNAL( newConnection() ),
          this,      SLOT( addConnection() ) );
```

- В этом обработчике необходимо создать для нового клиента свой сокет:

```
QTcpSocket *clientConnection = tcpServer->nextPendingConnection();
```

Заранее нужно побеспокоиться о том, чтобы уничтожить данный сокет по окончании работы с клиентом, для чего нужно связать сигнал `disconnected()` с соответствующим слотом:

```
connect ( clientConnection, SIGNAL( disconnected() ),
          clientConnection, SLOT( deleteLater() ) );
```

Обычно используют слот `QObject::deleteLater()`, он ставит указанный объект в очередь на удаление, а само удаление происходит, когда управление в очередной раз возвращается к циклу обработки событий.

- Наконец, после всех описанных приготовлений можно выполнить передачу данных между клиентом и сервером. Данные могут передаваться по строкам (разделителем служит символ перехода на новую строку) или блоками заранее указанного размера. Сам процесс передачи может происходить асинхронно (после получения очередной порции данных генерируется сигнал и соответствующий слот обрабатывает принятые данные или готовит для передачи новую порцию данных) или в блокирующем режиме (когда выход из функций отправки или получения данных происходит только после того, как процесс передачи или приема будет завершен). Подробнее мы рассмотрим все эти варианты чуть позже.
- Если инициатором окончания работы с клиентом должен быть сервер, то нужно разорвать установленное между клиентом и сервером соединение с помощью метода `QAbstractSocket::disconnectFromHost()`:

```
clientConnection->disconnectFromHost();
```

- По окончании работы сервера (когда больше не требуется ожидать подключения клиентов) его следует вывести из "слушающего" состояния:

```
tcpServer->close();
```

- Перед компиляцией программы к тексту файла описания проекта *.pro необходимо добавить строку

```
QT += network
```

и не забыть подключить заголовочный файл `<QtNetwork>`.

20.1. Эхо-сервер в блокирующем режиме

ЗАДАЧА. Разработать *эхо-сервер*: приложение, которое принимает строку данных от любого клиента и отсылает ему эту же строку обратно.

Решение иллюстрирует листинг 20.2.

Листинг 20.2. Эхо-сервер (файл chap20/echo1/echoserver/echoserver.cpp)

```
1  #include <iostream>
2  #include <QtCore>
3  #include <QtNetwork>
4
```

```
#define tr  QObject::tr
#define STR(s) ((outcodec->fromUnicode(s)).constData())
using namespace std;

int main(int /*argc*/, char ** /*argv*/) {
  QTextCodec *incodec = QTextCodec::codecForName("CP1251");
  QTextCodec *outcodec = QTextCodec::codecForName("CP866");
  QTextCodec::setCodecForTr(incodec);

  QTcpServer server;
  QTcpSocket *client;
  QByteArray block;
  QString s;
  QHostAddress addr;

  if ( ! server.listen(QHostAddress::Any, 5678) ) {
      cout << STR(tr("Не могу запустить сервер: %1\n")
                 .arg( server.errorString() ) );
      server.close();
      return 1;
  }
  cout << STR(tr("Сервер запущен\n"));
  forever {
      while ( !server.waitForNewConnection (5000) ) {
          // cout << STR(tr(
          //   "Ждем запрос на подключение от клиента...\n"));
      }
      client = server.nextPendingConnection();
      addr = client->peerAddress();
      cout << STR(tr("Подключился новый клиент с адреса %1\n")
                  .arg( addr.toString() ));
      block = outcodec->fromUnicode(tr("Hello! Привет!")+0x0d);
      client->write(block);
      forever {
          if ( !client || !client->isValid() || client->state() ==
                       QAbstractSocket::UnconnectedState ) {
              delete client;
              break;
          }
          if ( client->waitForReadyRead(3000) ) {
              block = client->readAll();
              s = incodec->toUnicode(block.data());
```

```
47                if ( s.contains(0x08) ) {
48                    cout << STR(tr("Сервер остановлен\n"));
49                    return 0;
50                }
51                cout << STR(tr("Получены новые данные: %1\n").arg(s));
52                client->write(outcodec->fromUnicode(s));
53                //client->disconnectFromHost();
54            }
55        }
56        cout << STR(tr("Клиент отключился\n"));
57    }
58    return 0;
59 }
```

Пояснения к программе:

- (14) — создаем серверный сокет;
- (20–25) — и переводим его в слушающий режим. Если данная операция завершилась неудачно (например, если указанный порт уже занят другим сокетом), то выполнение программы завершается (24). Иначе мы входим в бесконечный цикл (27–57);
- (28–31) — ожидаем подключения клиента в течение 5 с; если за это время ни один клиент не попытался к нам подключиться, то повторяем попытку пятисекундного ожидания, пока она не завершится успехом. Функция `waitForNewConnection(мсек)` ожидает приход запроса на подключение от какого-нибудь клиента. Выход из нее произойдет при поступлении запроса или по истечении указанного времени (в зависимости от того, какое из этих событий наступит раньше). В первом случае функция возвращает значение `true` и цикл `while` (28) завершается;
- (33–35) — если подключение произошло, принимаем запрос клиента, создавая для него новый сокет с помощью функции `QTcpSocket::nextPendingConnection()`, определяем адрес клиента и выводим его на консоль сервера;
- (36, 37) — отправляем клиенту приветственное сообщение. Обычные эхо-серверы не пытаются "говорить" с клиентом по собственной инициативе, но пусть наш сервер хоть чем-нибудь отличится от других;
- (38–55) — во вложенном бесконечном цикле проверяем, не отключился ли клиент. Если да, то уничтожаем клиентский сокет и завершаем цикл. Иначе в течение трех секунд ожидаем прихода от клиента каких-либо данных (44). Если ничего не пришло, то повторяем трехсекундный цикл ожидания, а иначе выводим полученную строку на консоль сервера и отправляем ее обратно клиенту. Если в принятой от клиента строке содержится

символ с кодом `08` (клавиша <Backspace>), то сервер завершает свою работу (47–49);

- (41) — нарушаем обычное правило библиотеки Qt и явно удаляем клиентский сокет после отключения клиента. В следующих версиях этой программы мы будем разрабатывать собственные классы, косвенно порожденные от `QObject`, поэтому снова забудем про использование `delete`.

Пока мы не разработали клиентское приложение для нашего эхо-сервера, поэтому воспользуемся стандартной для системы Windows программой HyperTerminal. Для этого выполним команду главного меню **Пуск** | **Программы** | **Стандартные** | **Связь** | **HyperTerminal**. В программе HyperTerminal выполним команду **Файл** | **Свойства** и перейдем на вкладку **Параметры**. Нажмем кнопку **Параметры ASCII** и поставим опцию **Дополнять символы возврата каретки (CR) переводами строк (LF)**. Снимем галочку **Отображать введенные символы на экране**. Два раза нажмем кнопку **OK**. Затем подключимся к нашему серверу: **Файл** | **Новое подключение**. Введем любое название создаваемого подключения и нажмем кнопку **OK**. В раскрывающемся списке **Подключаться через** выберем режим **TCP/IP (Winsock)**. Укажем адрес узла (имя или IP-адрес компьютера, на котором запущен наш эхо-сервер) и номер порта 5678. Если и сервер, и клиент запущены на одном и том же компьютере, то в качестве адреса сервера можно написать имя `localhost` или адрес `127.0.0.1`. Нажмем кнопку **OK**.

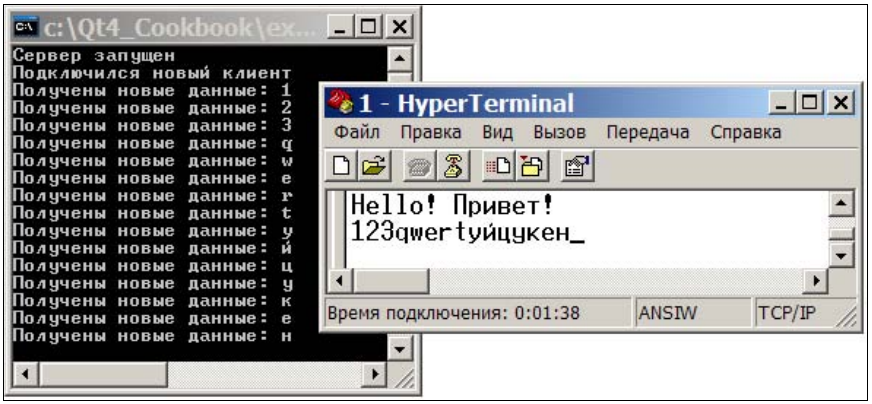


Рис. 20.1. Эхо-сервер и HyperTerminal в качестве клиента

Если подключение установится без ошибок, в экране терминала мы увидим приветственное сообщение от сервера, а в консольном окне сервера — информацию о подключенном клиенте (рис. 20.1). Теперь, если на компьютере, где запущено клиентское приложение, вводить с клавиатуры любые символы, то они будут отправляться назад и появляться на экране терминала, а инфор-

мация о переданных символах будет отображаться в окне сервера. Если клиент и сервер находятся на разных компьютерах, а клавиши нажимаются достаточно быстро, то они имеют шанс попасть в один и тот же сетевой пакет, в результате в окне сервера мы сможем увидеть не одиночные буквы и цифры, как на рис. 20.1, а несколько символов подряд.

Попробуем нажать клавишу <Backspace> и убедимся, что сервер при этом завершает свою работу. Подключимся снова, а затем запустим еще одно окно HyperTerminal и проверим, сможет ли сервер принять подключение от второго клиента, пока обрабатывает запросы от первого. Разумеется, нет. Мы исправим этот недостаток в *разд. 20.3*.

20.2. Клиент для эхо-сервера

ЗАДАЧА. Разработать консольное клиентское приложение для эхо-сервера.

Решение. Текст программы приведен в листинге 20.3.

Листинг 20.3. Клиент для эхо-сервера (файл chap20/echo1/echoclient/echoclient.cpp)

```
#include <iostream>
#include <QtCore>
#include <QtNetwork>

#define STR(s) ((outcodec->fromUnicode(s)).constData())
#define tr  QObject::tr
using namespace std;

int main(int /*argc*/, char ** /*argv*/) {

   QTextCodec *incodec = QTextCodec::codecForName("CP1251");
   QTextCodec *outcodec = QTextCodec::codecForName("CP866");
   QTextCodec::setCodecForTr(incodec);

   QTcpSocket socket;
   QByteArray block;
   QString s;
   char r[80];
   qint64 port;

   cout << STR(tr("\nВведите адрес или имя сервера:\n"));
   cin >> r;
   s = outcodec->toUnicode(r);
```

```
24
25      cout << STR(tr("\nВведите номер порта сервера:\n"));
26      cin >> port;
27
28      socket->connectToHost(s, port);
29      if ( !socket.waitForConnected(5000) ) {
30         cout << STR(tr(
31             "\nНе удалось связаться с сервером: %1\n")
32                              .arg(socket.errorString()));
33         return 1;
34      }
35      if (!socket.waitForReadyRead(3000)) {
36         cout << STR(tr("\nНет отклика от сервера: %1\n")
37                              .arg(socket.errorString()));
38         return 2;
39      }
40      block = socket.readAll();
41      s = outcodec->toUnicode(block.data());
42      cout << STR(tr("Получено от сервера: %1\n").arg(s));
43
44      cout << STR(tr("\nВведите строку:\n"));
45      cin >> r;
46      s = outcodec->toUnicode(r);
47
48      if (socket.write(incodec->fromUnicode(s)) < 0){
49         cout << STR(tr(
50             "\nНе могу записать данные для отправки: %1\n")
51                              .arg(socket.errorString()));
52         return 3;
53      }
54
55      if (socket.waitForBytesWritten(3000))
56         cout << STR(tr("Отправили на сервер строку:\n%1\n")
57                              .arg(s));
58      else{
59         cout << STR(tr("\nНе могу отправить данные: %1\n")
60                              .arg(socket.errorString()));
61         return 4;
62      }
63
65      if (!socket.waitForReadyRead(3000)) {
66         cout << STR(tr("\nНет отклика от сервера: %1\n")
67                              .arg(socket.errorString()));
```

```
68          return 5;
69      }
70      block = socket.readAll();
71      s = outcodec->toUnicode(block.data());
72      cout << STR(tr("Получено от сервера: %1\n").arg(s));
73
74      socket.disconnectFromHost();
75      cout << STR(tr("Отключились от сервера\n"));
76      return 0;
77  }
```

Пояснения к программе:

- (5) — тот же макрос, что и для сервера, упрощает преобразование строк в кодировку консоли в системе Windows;
- (15–19) — создаем сокет, буфер для чтения/записи данных и другие служебные переменные;
- (21–23) — узнаем у пользователя имя или IP-адрес сервера;
- (25, 26) — узнаем у пользователя номер порта сервера;
- (28) — пытаемся подключиться к серверу;
- (29–34) — если в течение пяти секунд подключение не произошло, выводим сообщение об ошибке и завершаем работу;
- (35–38) — если в течение трех последующих секунд от сервера не получена строка приветствия, выводим сообщение об ошибке и завершаем работу;
- (40–42) — читаем полученную строку приветствия и выводим ее на экран;
- (44–46) — просим пользователя ввести строку текста с клавиатуры (напоминаем, что в переменную `r` будет записано первое слово введенной строки, до пробельного символа);
- (48–53) — записываем строку для отправки во внутренний буфер. Если при этом возникла ошибка, функция `write()` возвращает значение `-1`, тогда мы выводим сообщение об ошибке на экран и завершаем работу;
- (55–62) — если в течение трех секунд так и не удалось отправить сообщение на сервер, выводим сообщение об ошибке и завершаем работу;
- (65–72) — ждем не более трех секунд отклик от сервера. Выводим полученную строку или сообщение об ошибке;
- (74) — отключаемся от сервера.

20.3. Асинхронный однопоточный эхо-сервер

ЗАДАЧА. Разработать консольное приложение — однопоточный эхо-сервер, который мог бы обслуживать сразу несколько подключенных пользователей.

Решение. Чтобы обслуживать больше одного клиента, нам придется либо использовать асинхронные (неблокирующие) функции работы с сокетами, либо создавать для каждого клиента отдельный поток выполнения на сервере. Сначала рассмотрим первый вариант (однопоточный асинхронный эхо-сервер, листинги 20.4 и 20.5).

Листинг 20.4. Однопоточный эхо-сервер с неблокирующими сокетами (файл chap20/echo2/echoserver/echoserver.h)

```
#include <QtCore>
#include <QtNetwork>

class EchoServer : public QTcpServer {
   Q_OBJECT
public:
   EchoServer(QObject *parent=0);
   ~EchoServer();
   bool start();

   QList<QTcpSocket*> m_clients;

public slots:
   void addConnection();
   void removeConnection();
   void onRead();
   void displayError(QAbstractSocket::SocketError);
};
```

Листинг 20.5. Однопоточный эхо-сервер с неблокирующими сокетами (файл chap20/echo2/echoserver/echoserver.cpp)

```
#include <iostream>
#include <QtCore>
#include <QtNetwork>
#include "echoserver.h"

```

```
24 #define STR(s) ( ( QTextCodec::codecForLocale()->
25                       fromUnicode(s) ).constData() )
26
27 using namespace std;
28
29 EchoServer::EchoServer(QObject *parent) : QTcpServer(parent){}
31
32 EchoServer::~EchoServer() {
33    while (!m_clients.isEmpty()) {
34       QTcpSocket *client = m_clients.takeFirst();
35       client->write(QTextCodec::codecForLocale()->fromUnicode(
36                     QObject::tr("\nСервер отключается") + 0x0d ) );
37       client->flush();
38       client->close();
39       client->deleteLater();
40    }
41 }
42
43 bool EchoServer::start() {
44    QHostAddress addr;
45    if ( !listen(QHostAddress::Any, 5678) ) {
46        cout << STR(QObject::tr("Не могу запустить сервер: %1\n")
47                    .arg( errorString() ) );
48        close();
49        return false;
50    }
51    connect(this, SIGNAL( newConnection() ),
52            this, SLOT( addConnection() ) );
53    cout << STR(QObject::tr("Сервер запущен\n"));
54    return true;
55 }
56
57 void EchoServer::addConnection() {
58    QTcpSocket *client = nextPendingConnection();
59    connect(client, SIGNAL(disconnected()),
60            this, SLOT(removeConnection()));
62    connect(client, SIGNAL(readyRead()), this,  SLOT(onRead()));
63    connect(client, SIGNAL(error(QAbstractSocket::SocketError)),
64            this, SLOT(displayError(QAbstractSocket::SocketError)));
65    client->write(QTextCodec::codecForLocale()->fromUnicode(
66                     QObject::tr("Hello! Привет!") + 0x0d ) );
67    QHostAddress addr = client->peerAddress();
68    cout << STR(QObject::tr("Подключился новый клиент с адреса %1\n")
69                        .arg( addr.toString() ));
```

```
70     m_clients.append(client);
71  }
72
73  void EchoServer::removeConnection(){
74     QTcpSocket *client = qobject_cast<QTcpSocket*>(sender());
75     QHostAddress addr = client->peerAddress();
76     cout << STR(QObject::tr("Клиент %1 отключен\n")
77                          .arg( addr.toString() ));
78     m_clients.removeAt(m_clients.indexOf(client));
79     client->deleteLater();
80  }
81
82  void EchoServer::onRead() {
83     QTcpSocket *client = qobject_cast<QTcpSocket*>(sender());
84     QByteArray block = client->readAll();
85     QString s = QTextCodec::codecForTr()->toUnicode(block.data());
86     if ( s.contains(0x08) ) {
87        cout << STR(QObject::tr("\nСервер остановлен\n"));
88        deleteLater();
89        QCoreApplication::instance()->exit();
90        return;
91     }
92     QHostAddress addr = client->peerAddress();
93     cout << STR(QObject::tr(
94                    "Получены новые данные от клиента %1: %2\n")
95                 .arg( addr.toString() ) . arg(s) );
96     client->write(QTextCodec::codecForLocale()->fromUnicode(s));
97  }
98
99  void EchoServer::displayError(QAbstractSocket::SocketError){
100    cout << STR(QObject::tr("\nОшибка: %1\n").arg(errorString()));
101 }
102
103 //=============================================
104 int main(int argc, char **argv) {
105    QCoreApplication app(argc, argv);
106    QTextCodec *incodec = QTextCodec::codecForName("CP1251");
107    QTextCodec *outcodec = QTextCodec::codecForName("CP866");
108    QTextCodec::setCodecForTr(incodec);
109    QTextCodec::setCodecForLocale(outcodec);
110
```

```
111     EchoServer server;
112     server.start();
113     return app.exec();
114 }
```

Пояснения к программе:

- (32–41) — в деструкторе мы закрываем соединения со всеми клиентами, но предварительно посылаем им сообщение об отключении сервера;
- (43–55) — данный метод выполняется при старте сервера: пытаемся установить серверный сокет, в случае неудачи выводим сообщение об ошибке, иначе связываем сигнал `newConnection()` с методом `addConnection()`, в котором будет обрабатываться запрос на подключение клиента;
- (58) — при подключении нового клиента создаем для него отдельный сокет, через который будет осуществляться дальнейший обмен информацией с данным клиентом (а серверный сокет, на который поступил запрос клиента на подключение, сразу освобождается, чтобы иметь возможность обрабатывать запросы следующих клиентов);
- (59, 60) — связываем сигнал `disconnected()`, генерируемый при отключении клиента, с методом `removeConnection()`;
- (62) — связываем сигнал `readyRead()`, генерируемый по приходу информации от клиента, с методом `onRead()`;
- (63, 64) — связываем сигнал `error()`, генерируемый при возникновении ошибки сокета, с методом `displayError()`;
- (65, 66) — посылаем клиенту приветственное сообщение;
- (67–69) — определяем IP-адрес клиента и выводим на консоль сервера информацию о новом клиенте;
- (70) — добавляем клиентский сокет в список активных соединений;
- (73–80) — данный метод выполняется при разрыве соединения: определив, какой именно клиент отключился (83), мы выводим информацию о нем на консоль сервера (75–77) и удаляем соответствующий клиентский сокет из списка активных соединений (78), после чего разрешаем уничтожение сокета (79);
- (82–97) — данный метод выполняется по приходу информации от клиента: определяем, от какого клиента поступило сообщение (83), считываем его в буфер (84), преобразуем в строку Unicode (85); если в принятом сообщении содержится символ с кодом `08`, то завершаем работу сервера (86–91); иначе выводим полученное сообщение на консоль сервера (93–95) и отправляем его назад клиенту (96).

20.4. Многопоточный эхо-сервер

ЗАДАЧА. Разработать многопоточный эхо-сервер.

Решение. Текст программы приведен в листингах 20.6 и 20.7, а внешний вид окна сервера в момент обслуживания двух клиентов, описанных в *разд. 20.2*, показан на рис. 20.2.

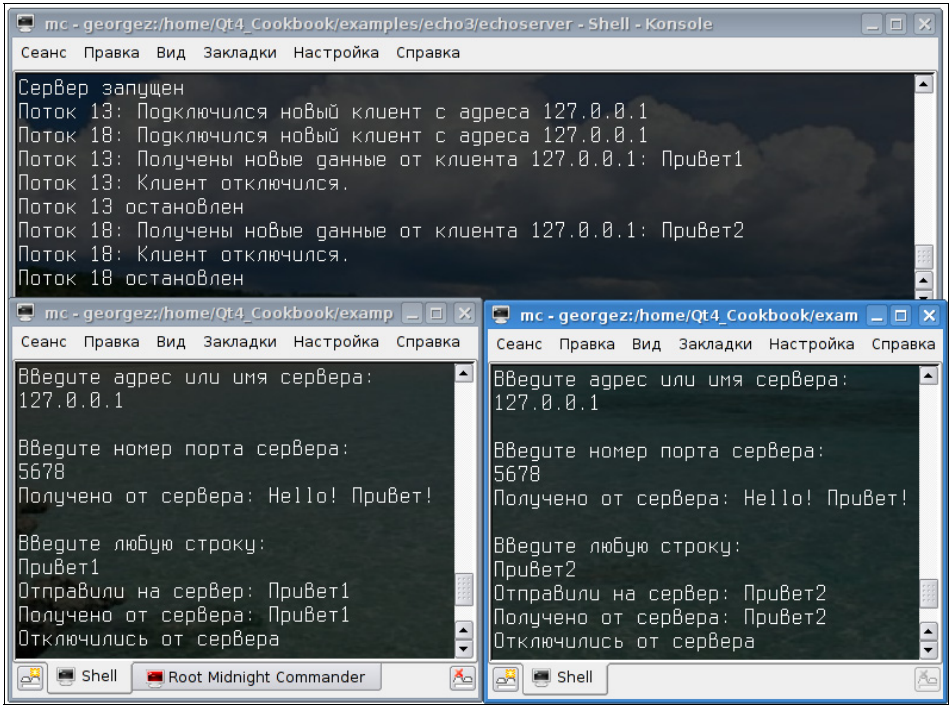


Рис. 20.2. Многопоточный сервер работает с несколькими клиентами одновременно

Листинг 20.6. Многопоточный эхо-сервер (файл chap20/echo3/echoserver/echoserver.h)

```
1 #include <QtCore>
2 #include <QtNetwork>
3
4 class EchoThread : public QThread {
5     Q_OBJECT
6 public:
7     EchoThread(int socketDescriptor, QObject *parent=0);
8     void run();
```

```
 9 private:
10     int m_socketDescriptor;
11     QTcpSocket *m_client;
12     QHostAddress m_addr;
13 public slots:
14     void onRead();
15     void onDisconnect();
16     void displayError(QAbstractSocket::SocketError error);
17 signals:
18     void error(QAbstractSocket::SocketError);
19 };
20 //---------------------------------------
21 class EchoServer : public QTcpServer {
22     Q_OBJECT
23 public:
24     EchoServer(QObject *parent=0);
25     bool start();
26 protected:
27     void incomingConnection(int socketDescriptor);
28     QList<EchoThread*> m_threads;
29 public slots:
30     void removeThread();
31 };
```

Листинг 20.7. Многопоточный эхо-сервер (файл chap20/echo3/echoserver/echoserver.cpp)

```
32 #include <iostream>
33 #include <QtCore>
34 #include <QtNetwork>
35 #include "echoserver.h"
36
37 #define STR(s) ( (QTextCodec::codecForLocale() ->
                          fromUnicode(s)).constData())
38
39 using namespace std;
40
41 //---------------------------------------
42
43 EchoThread::EchoThread(int socketDescriptor, QObject *parent)
44    : QThread(parent), m_socketDescriptor(socketDescriptor) {}
45
```

```
46 void EchoThread::run(){
47     m_client = new QTcpSocket();
48     if (!m_client->setSocketDescriptor(m_socketDescriptor)) {
49         emit error(m_client->error());
50         return;
51     }
52     connect(m_client, SIGNAL(readyRead()), this,  SLOT(onRead()),
53             Qt::DirectConnection);
54     connect(m_client, SIGNAL(error(QAbstractSocket::SocketError)),
55             this, SIGNAL(error(QAbstractSocket::SocketError)),
56             Qt::DirectConnection);
57     connect(this, SIGNAL(error(QAbstractSocket::SocketError)),
58             this, SLOT(displayError(QAbstractSocket::SocketError)),
59             Qt::DirectConnection);
60     connect(m_client, SIGNAL(disconnected()),
61             this, SLOT(onDisconnect()),  Qt::DirectConnection);
62     m_client->write(QTextCodec::codecForLocale()->fromUnicode(
63                         QObject::tr("Hello! Привет!") + 0x0d ) );
64     m_addr = m_client->peerAddress();
65     cout << STR(QObject::tr(
           "Поток %1: Подключился новый клиент с адреса %2\n")
66              .arg(m_socketDescriptor).arg( m_addr.toString() ));
67     exec();
68 }
69
70 void EchoThread::onRead() {
71     QByteArray block = m_client->readAll();
72     QString s = QTextCodec::codecForTr()->toUnicode(block.data());
73     cout << STR(QObject::tr(
          "Поток %1: Получены новые данные от клиента %2: %3\n")
74        .arg(m_socketDescriptor).arg(m_addr.toString()).arg(s) );
75     m_client->write(QTextCodec::codecForLocale()->fromUnicode(s));
76 }
77
78 void EchoThread::onDisconnect() {
79     m_client->deleteLater();
80     cout << STR(QObject::tr(
             "\nПоток %1 остановлен\n").arg(m_socketDescriptor));
81     quit();
82 }
83
84 void EchoThread::displayError(QAbstractSocket::SocketError error){
85     if (error == QAbstractSocket::RemoteHostClosedError)
```

```
 86         cout << STR(QObject::tr("\nПоток %1: Клиент отключился.")
 87                              .arg(m_socketDescriptor));
 88     else
 89         cout << STR(QObject::tr(
                "\nОшибка: %1\n").arg(m_client->errorString()));
 90 }
 91
 92 //---------------------------------------
 93
 94 EchoServer::EchoServer(QObject *parent)
 95           : QTcpServer(parent) {}
 96
 97 bool EchoServer::start() {
 98     QHostAddress addr;
 99     if ( !listen(QHostAddress::Any, 5678) ) {
100         cout << STR(QObject::tr("Не могу запустить сервер: %1\n")
101                     .arg( errorString() ) );
102         close();
103         return false;
104     }
105     cout << STR(QObject::tr("Сервер запущен\n"));
106     return true;
107 }
108
109 void EchoServer::incomingConnection(int socketDescriptor) {
110     EchoThread *thread = new EchoThread(socketDescriptor, this);
111     connect(thread, SIGNAL(finished()),
112             this, SLOT(removeThread()), Qt::DirectConnection);
113     m_threads.append(thread);
114     thread->start();
115 }
116
117 void EchoServer::removeThread(){
118     EchoThread *thread = qobject_cast<EchoThread*>(sender());
119     m_threads.removeAt(m_threads.indexOf(thread));
120     thread->deleteLater();
121 }
122
123 //=============================================
124 int main(int argc, char **argv) {
125     QCoreApplication app(argc, argv);
126     QTextCodec *incodec = QTextCodec::codecForName("CP1251");
127
```

```
128     #ifdef Q_WS_WIN
129     QTextCodec *outcodec = QTextCodec::codecForName("CP866");
130     QTextCodec::setCodecForLocale(outcodec);
131     #endif
132
133     QTextCodec::setCodecForTr(incodec);
134
135     qRegisterMetaType<QAbstractSocket::SocketError> (
                              "QAbstractSocket::SocketError");
136
137     EchoServer server;
138     server.start();
139     return app.exec();
140 }
```

Пояснения к программе:

- (4–19) — класс потока;
- (7) — конструктор потока;
- (8) — обязательный метод `run()`;
- (10) — целочисленный идентификатор клиентского сокета;
- (11) — сам клиентский сокет;
- (12) — IP-адрес клиента;
- (14) — метод-слот, исполняемый после прихода данных от клиента;
- (15) — метод-слот, исполняемый после отключения клиента;
- (16) — метод-слот, исполняемый после возникновения ошибки сокета;
- (18) — сигнал, который будет генерировать поток при возникновении ошибки клиентского сокета;
- (21–31) — класс многопоточного эхо-сервера;
- (27) — переопределили виртуальный метод, вызываемый при подключении к серверу нового клиента;
- (28) — список указателей на запущенные потоки. В нашем примере не используется, но в реальных программах часто необходим;
- (30) — метод-слот, выполняемый при завершении потока;
- (37) — макрос, облегчающий вывод сообщений на консольный экран;
- (43, 44) — в конструкторе потока вызывается конструктор родителя и сохраняется целочисленный дескриптор клиентского сокета;
- (46–51) — создаем сокет и назначаем ему дескриптор, полученный от основного потока, в котором от клиента был принят запрос на подключение.

При возникновении ошибки (например, если исчерпаны системные ресурсы, отводимые для работы сокетов) генерируем сигнал `error`;

- (52) — связываем сигнал, уведомляющий о приходе данных от клиента, с методом-слотом `onRead()`;
- (53) — поскольку экземпляр потока `EchoThread` и клиентский сокет созданы в разных потоках, приходится явно указать метод соединения сигнала и слота. Если этого не сделать, то вместо того, чтобы напрямую направить слоту необходимый сигнал, последний после преобразования в объект-сообщение будет поставлен в очередь сообщений того потока, которому принадлежит слот. В результате мы получим ошибку *"Socket notifies cannot be enabled in another thread"*;
- (54–59) — если клиентский сокет или наш поток генерируют сигнал `error`, то надо вывести на экран сообщение об ошибке (84–90);
- (60) — при отключении клиента (неважно, по собственной инициативе или со стороны сервера) будет выполнен метод `onDisconnect()`;
- (62) — отправляем клиенту приветственное сообщение;
- (64–66) — выясняем IP-адрес клиента и выводим на экран сервера сообщение о том, что подключился новый клиент;
- (67) — запускаем для нового потока собственный цикл обработки событий;
- (70–76) — читаем полученную от клиента строку, выводим ее на экран и отправляем назад клиенту. Преобразование принятой строки в Unicode и тут же обратно в нашем простом случае выглядит не слишком разумно, но реальный сервер обычно анализирует запрос клиента и формирует соответствующий ответ на основе каких-либо данных, поэтому без преобразований не обойтись;
- (78–80) — при отключении клиента ставим клиентский сокет в очередь на уничтожение и выводим сообщение на экран;
- (81) — завершаем выполнение потока;
- (84–90) — выводим на экран сообщение об ошибке сокета. Вариант `RemoteHostClosedError` обрабатываем отдельно, чтобы в этом случае на экран не выводилось слово "Ошибка": ведь на самом деле это просто отключение клиента;
- (94, 95) — конструктор сервера ничего не делает, только вызывает конструктор базового класса;
- (97–107) — при запуске сервера пытаемся занять порт 5678 и выводим сообщение об успехе или неудаче;

- (109–115) — виртуальный метод `incomingConnection()` выполняется при подключении нового клиента. Здесь мы создаем и запускаем поток, добавляем его в список потоков и связываем сигнал о его завершении с методом-слотом `removeThread()`;
- (117–121) — при получении сигнала о завершении потока выясняем, какой именно поток сгенерировал этот сигнал, ставим этот поток в очередь на уничтожение и удаляем из списка потоков;
- (124–140) — главная программа;
- (128–131) — если программа скомпилирована для Windows, то устанавливаем кодировку для вывода на консольный экран. Во всех остальных случаях по умолчанию будет задана системная кодировка;
- (135) — регистрируем тип `QAbstractSocket::SocketError`, чтобы использовать его в сигналах и слотах (в противном случае возможны сообщения об ошибках при выполнении программы);
- (137–139) — создаем объект-сервер, вызываем для него метод `start()` и запускаем основной цикл обработки событий.

20.5. Работа с протоколом FTP

ЗАДАЧА. Подключиться к серверу FTP, получить список каталогов и файлов, перейти в нужный каталог и получить заданный файл.

Решение. Для реализации клиента FTP в Qt используется класс `QFtp`. Алгоритм работы следующий:

- Создать экземпляр класса `QFtp`

```
QFtp *ftp = new QFtp(parent);
```

- Подключиться к FTP-серверу

```
ftp -> connectToHost("ftp.trolltech.com", 21);
ftp -> login(user, password);
```

Порт для подключения по умолчанию считается равным 21. Имя и пароль можно не указывать (если сервер FTP принимает такие соединения). Если необходим прокси-сервер, то перед подключением придется установить его параметры, например:

```
ftp -> setProxy("192.168.0.1", 8080);
```

Режим прокси "FTP поверх HTTP" классом `QFtp` напрямую не поддерживается (равно как и авторизация), но можно использовать класс `QNetworkProxy` (листинг 20.8).

Листинг 20.8. Класс QNetworkProxy

```
QNetworkProxy proxy;
proxy.setType(QNetworkProxy::HttpProxy);
proxy.setHostName("192.168.0.1");
proxy.setPort(8080);
proxy.setUser("ИмяПользователя");
proxy.setPassword("Пароль");
QNetworkProxy::setApplicationProxy(proxy);
```

- Сменить текущий каталог можно с помощью метода `FTP::cd(ИмяКаталога)`, создать новый каталог — `mkdir(ИмяКаталога)`. Список объектов (подкаталогов и файлов), находящихся в некотором каталоге, возвращает асинхронно выполняемый (т. е неблокирующий) метод `list(ИмяКаталога)`. Если имя каталога не указано, то используется текущий. При выполнении метода `list()` для каждого найденного каталога генерируется сигнал `listInfo(const QUrlInfo& i)`.
- Для получения файла с сервера FTP служит неблокирующий метод `get(ИмяФайла)`. Вторым параметром может быть задан указатель на поток `QIODevice`, в который будет происходить запись данных. Если поток не указан, то по окончании процесса скачивания файла генерируется сигнал `readyRead()`, а сами данные после этого можно прочитать с помощью методов

  ```
  qint64 QFtp::read ( char * data, qint64 maxlen )
  ```

  или

  ```
  QByteArray QFtp::readAll()
  ```

- В конце работы необходимо закрыть соединение с помощью метода `close();`.
- Перед компиляцией программы к тексту файла описания проекта *.pro нужно добавить строку

  ```
  QT += network
  ```

  и не забыть подключить заголовочный файл `<QtNetwork>`.

20.6. Отправка электронной почты по протоколу SMTP

ЗАДАЧА. Отправить указанный файл HTML по электронной почте с помощью протокола SMTP.

Решение иллюстрируют листинги 20.9 и 20.10.

Листинг 20.9. Файл chap20/smtp/smtp.h

```
#ifndef _SMTP_H_
#define _SMTP_H_

#include <QtCore>
#include <QtNetwork>
#include <QDebug>
#include <QtTest/QtTest>

class Smtp : public QObject {
    Q_OBJECT
public:
    Smtp(QString host, QString username, QString pass);
    ~Smtp();
    void SetHost(QString host, QString username, QString pass);
    void Send(const QString &from, const QString &to,
              const QString &subject, const QString &body);
    int linesend;
    QStringList ErrorMSG;
    QStringList GetErrorlist();
signals:
    void status( const QString &);
    void ConnectorSuccess();
    void SendLine();
    void ErrorCloseAll();
    void SuccessQuit();
private slots:
    void disconnected();
    void connected();
    void ReadLiner();
    void PutSendLine();
private:
    bool isconnected;
    QString smtphost;
    QString smtpusername;
    QString smtppass;
    QString message;
    QString output;
    QString RemoteServerName;
    QString mailstatus;
```

```
    QTextStream *t;
    QTcpSocket *smtpsocket;
    QString from;
    QString rcpt;
    QString response;
    QString SendLineAndGrab(QString senddata);
    int Timeout;
    QString encodeBase64( QString xml );
    QString decodeBase64( QString xml );
    int dateswap(QString form, uint unixtime );
    QString TimeStampMail();
};
#endif // _SMTP_H_
```

Листинг 20.10. Файл chap20/smtp/smtp.cpp

```
#include "smtp.h"

// Конструктор:
Smtp::Smtp(QString host, QString username, QString pass){
    qDebug() << "--- Smtp start";
    this->smtphost = smtphost;          // Сервер
    this->smtpusername = smtpusername; // Имя пользователя
    this->smtppass = smtppass;          // Пароль
}

// Задать имя сервера, пользователя и пароль:
void Smtp::SetHost(QString host, QString username, QString pass){
    this->smtphost = host;
    this->smtpusername = username;
    this->smtppass = pass;
}

// Отправить письмо:
void Smtp::Send(const QString &from, const QString &to,
                const QString &subject, const QString &body){
    int waittime = 5000; // мс (время ожидания ответа от сервера)
    this->from = from;   // От кого
    rcpt = to;           // Кому
    ErrorMSG.clear();    // Ошибок пока нет
    Timeout = waittime;
    linesend = 0;        // Пока ничего не отправлено
    isconnected = false; // Еще не соединялись
    message = TimeStampMail()+"\n";
```

```
    // Заголовок почтового сообщения:
    message.append("User-Agent: Qt SMTP client\n");
    message.append("X-Accept-Language: en-us, en\n");
    message.append("MIME-Version: 1.0\n");
    message.append("To: " + to + "\n");
    message.append("From: "+from+" <" + from + ">\n");
    message.append("Subject: " + subject + "\n");
    // ...
    smtpsocket = new QTcpSocket(this); // Создаем сокет

    connect( this, SIGNAL(ConnectorSuccess()),
             this, SLOT(ReadLiner()));
    connect( this, SIGNAL(SendLine()),
             this, SLOT(PutSendLine()));
    if (smtphost.size() > 0)
        smtpsocket->connectToHost(smtphost, 25); // Соединяемся
    else // Если хост не задан — подключаемся локально:
        smtpsocket->connectToHost("localhost", 25);

    if (smtpsocket->waitForConnected(Timeout)) {
        qDebug() <<"--- connected on  " << smtphost;
        if (smtpsocket->waitForReadyRead(Timeout)) {
            qDebug() << "waitForReadyRead is ready";
            isconnected = true;
            emit ConnectorSuccess(); // Подключились!
        }
    }else{
        emit ErrorCloseAll(); // Ответ так и не пришел
    }
}

// Читать ответ от сервера SMTP:
void Smtp::ReadLiner(){
    if (isconnected) {
        qDebug() << "socketType = " << smtpsocket->socketType();
        QTextCodec *codecx;
        codecx = QTextCodec::codecForMib(106);
        t = new QTextStream( smtpsocket );
        t->setCodec(codecx);

        int loops = 0;
        while (!t->atEnd()) {
           loops++;
```

```
            response = t->readLine();
            qDebug() << loops << " in line  " << response;
        }
        if (response.size() > 0) {
            RemoteServerName = response;
            mailstatus = response.left(3);
            qDebug() << "--- Status = " << mailstatus;
            if (mailstatus == "220") {
                 response="";
                 linesend = 1;
                 emit SendLine();
             }
        }else{
             emit ErrorCloseAll();
        }
    }
}

Smtp::~Smtp(){  // Деструктор
    qDebug() << "Smtp stop and delete";
}

// Формирование сообщения для отправки:
void Smtp::PutSendLine(){
    int current = linesend; // Счетчик отправленных строк
    qDebug() <<"Send line" << linesend;
    switch(current) {
      case 1: // Отправляется первая строка
          response = SendLineAndGrab("ehlo localhost");
          if (response.size() > 0) {
              ErrorMSG.append(response);
              qDebug() << "1---- " << response;
              linesend = 2;
              emit SendLine();
          }else{
              qDebug() << "Connection lost";
          }
          response ="";
      break;
      case 2: // Отправляется вторая строка
          response = SendLineAndGrab("AUTH LOGIN");
           if (response.size() > 0) {
           ErrorMSG.append(response);
                qDebug() << "2---- " << response;
```

```
                linesend = 3;
                emit SendLine();
            } else {
                qDebug() << "Connection lost"; // Разрыв соединения
            }
           response ="";
      break;
      // ...
      case 8: // Отправляется восьмая строка
          SendLineAndGrab("QUIT");
          break;
      default:
        qDebug() << "Last line ...";
        emit ErrorCloseAll();
        return;
        break;
    }
}

// Послать строку сообщения и ждать ответ от сервера:
QString Smtp::SendLineAndGrab(QString senddata){
    QString incommingData = "";
    if (isconnected) {
      // QApplication::setOverrideCursor(QCursor(Qt::WaitCursor));
      int current = linesend;
      int loops = 0;

      qDebug() << "Send" << current << "Now => " << senddata;
      *t << senddata << "\r\n";
      t->flush();
      if (senddata != "QUIT") {
         if (smtpsocket->waitForReadyRead(Timeout)) {
             while (!t->atEnd()) {
                loops++;
                QString opera = t->readLine()+"\n";
                incommingData = opera + incommingData;
                qDebug() << loops << "|#" << opera << "#|";
             }
          } else {
            // QApplication::restoreOverrideCursor();
            delete smtpsocket;
            delete t;
            isconnected = false;
```

```
            emit SuccessQuit();
            return incommingData;
          }
          // QApplication::restoreOverrideCursor();
    } else {
        emit ErrorCloseAll();
    }
    return incommingData;
}

void Smtp::disconnected(){
    qDebug() <<"!!! disconnected";
    qDebug() << "Error:"  << smtpsocket->errorString();
}

void Smtp::connected(){
    output.append("connected");
    qDebug() << "Connected ";
}

// Кодировать текст сообщения:
QString Smtp::encodeBase64( QString xml ) {
    QByteArray text;
    text.append(xml);
    return text.toBase64();
}

// Декодировать текст сообщения:
QString Smtp::decodeBase64( QString xml ) {
    QByteArray xcode("");;
    xcode.append(xml);
    QByteArray precode(QByteArray::fromBase64(xcode));
    QString notetxt = precode.data();
    return notetxt;
}

// Получить список строк с описаниями ошибок:
QStringList Smtp::GetErrorlist(){
    return ErrorMSG;
}

int Smtp::dateswap(QString form, uint unixtime ) {
     QDateTime fromunix;
```

```
     fromunix.setTime_t(unixtime);
     QString numeric = fromunix.toString((const QString)form);
     bool ok;
     return (int)numeric.toFloat(&ok);
}

QString Smtp::TimeStampMail(){
     /* Формат времени в теле сообщения e-mail согласно RFC 788
        http://www.faqs.org/rfcs/rfc788.html */
       QDateTime timer1( QDateTime::currentDateTime() );
       // timer1.setTimeSpec(Qt::UTC);

     uint unixtime = timer1.toTime_t();
     QDateTime fromunix;
     fromunix.setTime_t(unixtime);
     // ...
    return QString(rtfd_line.join(""));
}
//---------------------------------
#include <QtCore>
#include "smtp.h"

// Прочитать файл, в котором хранится текст письма:
QString fopenutf8(QString fullFileName) {
    QString s = "";
    QFile file(fullFileName);
    if (file.exists()) {
        if (file.open(QFile::ReadOnly | QFile::Text)) {
            s = QString::fromUtf8(file.readAll());
            file.close();
        }
    }
    return s;
}
//--------- Главная программа: ------------
int main(int argc, char *argv[]) {
    QCoreApplication a( argc, argv );
    Smtp *smtp  = new Smtp("smtp.mail.ru", "имя", "пароль");
    smtp->Send("имя@mail.ru", "получатель@сервер.ru",
               QObject::tr("Тема письма"), fopenutf8("файл.html") );
    delete smtp;
    return 0;
}
```

20.7. Использование модуля WebKit: простой Web-браузер

> В хорошей книге больше истин, чем хотел вложить в нее автор.
>
> *Мария Эбнер-Эшенбах*

Начиная с релиза 4.4, в состав библиотеки Qt включен модуль WebKit (**http://webkit.org**), в результате появилась возможность встраивать в Qt-приложения полноценный Web-браузер.

ЗАДАЧА. Разработать приложение для просмотра Web-страниц.

Решение. Текст программы приведен в листингах 20.11–20.13, а на рис. 20.3 показан внешний вид окна при просмотре новостей на сайте **www.google.ru**. Как видим, поддерживается также работа с динамическим содержимым страницы, поскольку в модуль WebKit встроена поддержка языка JavaScript.

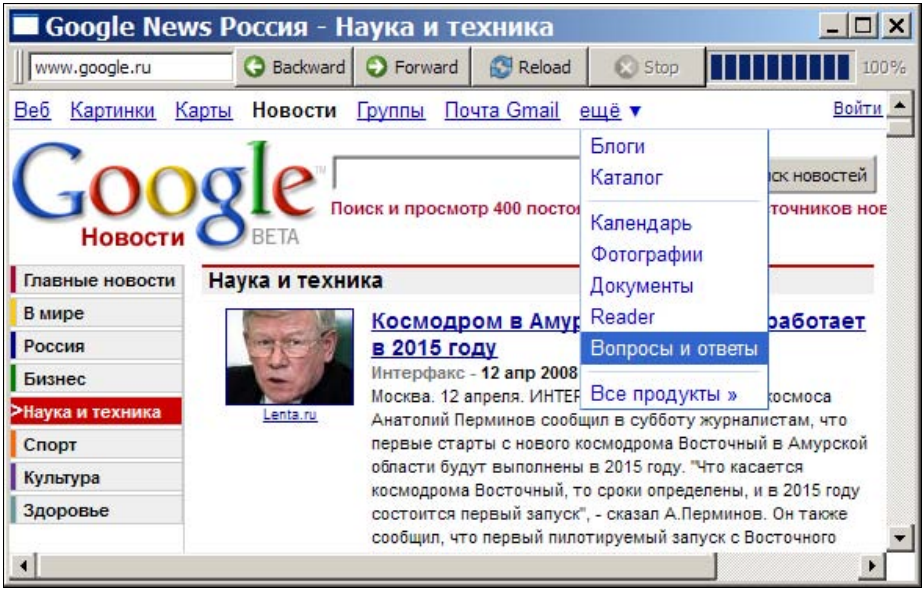


Рис. 20.3. Внешний вид окна Web-браузера

Листинг 20.11. Файл chap20/webbrowser/webbrowser.h

```
#ifndef _WEBBROWSER_H_
#define _WEBBROWSER_H_
```

```
#include <QMainWindow>
class QWebView;
class QLineEdit;
class QPushButton;
class QProgressBar;

class Webbrowser : public QMainWindow {
    Q_OBJECT
public:
    Webbrowser();
public slots:
    void load(); // загрузить страницу
    void started(); // начата загрузка страницы
    void finished(); // окончена загрузка страницы
    void onTitleChanged(const QString& text); // при изменении
                                              // заголовка страницы
private:
    QWebView *m_view;
    QToolBar *m_fileToolBar;
    QLineEdit *m_adress;
    QPushButton *m_backward, *m_forward, *m_reload, *m_stop;
    QProgressBar *m_progress;
};

#endif // _WEBBROWSER_H_
```

Листинг 20.12. Файл chap20/webbrowser/webbrowser.cpp

```
#include <QtGui>
#include <QtWebKit>
#include "webbrowser.h"

#define HTTP "http://"            // протокол по умолчанию
#define START_URL "www.google.ru" // стартовая страница по умолчанию

Webbrowser::Webbrowser(){
    m_view = new QWebView(this);
    setCentralWidget(m_view);
    setWindowTitle("Simple Web Browser");
    connect(m_view, SIGNAL(titleChanged(const QString&)),
            this, SLOT(onTitleChanged(const QString&)));

    m_fileToolBar = addToolBar(tr("File"));
```

```
    // Строка для ввода адреса страницы:
    m_adress = new QLineEdit(START_URL, m_fileToolBar);
    m_fileToolBar->addWidget(m_adress);
    connect(m_adress, SIGNAL(returnPressed()), this, SLOT(load()));

    // Кнопка "Назад":
    m_backward = new QPushButton(
        style()->standardIcon(QStyle::SP_ArrowBack),
        tr("Backward"), m_fileToolBar);
    m_fileToolBar->addWidget(m_backward);
    connect(m_backward, SIGNAL(clicked()), m_view, SLOT(back()));

    // Кнопка "Вперед":
    m_forward = new QPushButton(
        style()->standardIcon(QStyle::SP_ArrowForward),
        tr("Forward"), m_fileToolBar);
    m_fileToolBar->addWidget(m_forward);
    connect(m_forward, SIGNAL(clicked()), m_view, SLOT(forward()));

    // Кнопка "Обновить":
    m_reload = new QPushButton(
        style()->standardIcon(QStyle::SP_BrowserReload),
        tr("Reload"), m_fileToolBar);
    m_fileToolBar->addWidget(m_reload);
    connect(m_reload, SIGNAL(clicked()), m_view, SLOT(reload()));

    // Кнопка "Стоп":
    m_stop = new QPushButton(
        style()->standardIcon(QStyle::SP_BrowserStop),
        tr("Stop"), m_fileToolBar);
    m_fileToolBar->addWidget(m_stop);
    connect(m_stop, SIGNAL(clicked()), m_view, SLOT(stop()));

    // Индикатор процесса загрузки:
    m_progress = new QProgressBar(m_fileToolBar);
    m_fileToolBar->addWidget(m_progress);
    connect(m_view, SIGNAL(loadProgress(int)),
            m_progress, SLOT(setValue(int)));

    connect(m_view, SIGNAL(loadStarted()), this, SLOT(started()));
    connect(m_view, SIGNAL(loadFinished()), this, SLOT(finished()));

    load();
}
```

```
void Webbrowser::load(){   // Загрузить страницу
    QString name(m_adress->text());
    if (!name.startsWith(HTTP))
        name = HTTP + name.trimmed(); // добавим "http://"

    m_view->load(QUrl(name));
}

void Webbrowser::started(){
    m_stop->setEnabled(true);
    m_progress->setEnabled(true);
}

void Webbrowser::finished(){
    m_stop->setEnabled(false);
    m_progress->setEnabled(false);
}

void Webbrowser::onTitleChanged(const QString& text){
    setWindowTitle(text); // изменить заголовок окна
}
```

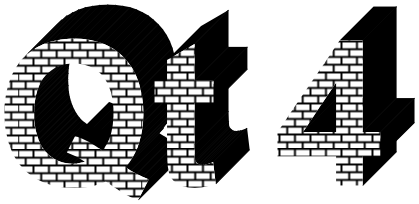
Листинг 20.13. Файл chap20/webbrowser/main.cpp

```
#include <QApplication>
#include "webbrowser.h"

int main(int argc, char *argv[]){
    QApplication app(argc, argv);
    Webbrowser win;
    win.show();
    return app.exec();
}
```

Перед компиляцией в файл описания проекта *.pro необходимо добавить строку:

```
QT += webkit
```

ГЛАВА 21



Использование модуля QtScript

Даже из мечты можно сварить варенье, если добавить фруктов и сахару.

Станислав Ежи Лец

В библиотеке Qt имеется модуль QtScript, позволяющий разрабатывать программы, расширяемые пользователем. Подобно тому, как в приложения Microsoft Office встроена поддержка языка Visual Basic for Application, так и в Qt-приложения может быть добавлен интерпретатор языка ECMAScript, по своему синтаксису аналогичного JavaScript.

ЗАДАЧА. Вычислить арифметические выражения с помощью модуля QtScript.

Решение. Для использования модуля QtScript необходимо:

- в файл описания проекта *.pro добавить строку `QT += script`;
- в программе подключить заголовочный файл `<QtScript>`;
- создать экземпляр класса `QScriptEngine`, после чего вызвать метод `QScriptEngine::evaluate()`, передав ему в качестве параметра текст скрипта.

В результате возвращается значение типа `QScriptValue`, которое можно привести к большинству других типов C++ и Qt с помощью соответствующих методов `QScriptValue::toInteger()`, `toInt32()`, `toUInt16()`, `toUInt32()`, `toNumber()`, `toBoolean()`, `toString()`, `toDateTime()`, `toVariant()`, `toQObject()`.

Глобальные переменные для использования в тексте скрипта можно определять как в самом скрипте (перед ними ставится необязательное ключевое слово `var`), так и в программе на C++. В последнем случае доступ к глобальному пространству имен осуществляется с помощью функции `QScriptEngine::globalOnject()`, а определение имени новой переменной и ее значения — с помощью функции `QScriptValue::setProperty()`:

```
QScriptValue myValue(&engine, значение);   // скриптовая переменная
QScriptValue global = engine.globalObject(); // пространство имен
global.setProperty("x", myValue);          // определили переменную x
```

В листинге 21.1 приведен пример программы, а на рис. 21.1 показан результат ее работы.

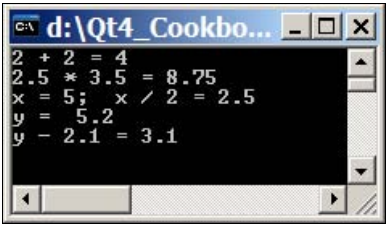


Рис. 21.1. Арифметические вычисления

Листинг 21.1. Файл chap21/qscript1/qscript1.cpp

```
#include <QCoreApplication>
#include <QtDebug>
#include <QtScript>

int main( int argc, char *argv[] ) {
    QCoreApplication app(argc, argv);
    QScriptEngine engine;

    qDebug() << "2 + 2 ="
             << engine.evaluate("2 + 2").toInteger();

    qDebug() << "2.5 * 3.5 ="
             << engine.evaluate("2.5 * 3.5").toNumber();

    // Работа с переменными:
    QString s = "x = 5; return x / 2;";
    QScriptValue r = engine.evaluate(s);
    qDebug() << "x = 5;  x / 2 =" << r.toNumber();

    // Создание глобальных переменных скрипта в программе на C++:
    QScriptValue val(&engine, 5.2);
    engine.globalObject().setProperty("y", val);
    qDebug() << "y = " << engine.evaluate("y").toNumber();
    qDebug() << "y - 2.1 ="
             << engine.evaluate("y - 2.1").toNumber();

    return app.exec();
}
```

Не забудьте в файле *.pro добавить строки

```
QT += script
win32 : CONFIG += console
```

ЗАДАЧА. Выполнить скрипт, текст которого находится в отдельном файле.

Решение задачи показано в листингах 21.2 и 21.3.

Листинг 21.2. Файл chap21/qscript2/qscript2.cpp

```
#include <QCoreApplication>
#include <QtDebug>
#include <QtScript>

int main( int argc, char *argv[] ) {
    QCoreApplication app(argc, argv);
    QScriptEngine engine;

    QFile file("script.js");
    file.open(QIODevice::ReadOnly);
    QScriptValue r = engine.evaluate(file.readAll());
    file.close();
    if (not r.isNull())
        if (r.isNumber())
            qDebug() << r.toNumber();
        else if (r.isString())
            qDebug() << r.toString();
        // ...
        else if (r.isVariant())
            qDebug() << r.toVariant();
        else if (r.isError())
            qDebug() << "!!! Error:" << r.toString();
        else
            qDebug() << "unrecognized type";

    return app.exec();
}
```

Листинг 21.3. Пример скрипта (файл chap21/script2/script.js)

```
x = 5;
print("x = ", x);
y = x / 2;
return y;
```

Файл `script.js` должен находиться в том каталоге, откуда выполняется запуск программы.

Можете проверить, как прореагирует наша программа на синтаксическую ошибку или деление на ноль.

Замечание

Можно выполнять текст скрипта не целиком, а строка за строкой.

ЗАДАЧА. Как из скрипта получить доступ к любому объекту — потомку `QObject`?

Решение. Если у нас имеется некоторый объект, реализованный на основе `QObject`, то после выполнения метода `QScriptEngine::newObject()` этот объект становится доступен внутри скрипта. Текст программы приведен в листингах 21.4 и 21.5, а на рис. 21.2 показан внешний вид окна приложения.

Рис. 21.2. Внешний вид кнопки и текст надписи определяется текстом скрипта

Листинг 21.4. Файл chap21/qscript3/qscript3.cpp

```
#include <QtGui>
#include <QtScript>

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QScriptEngine engine;

    QPushButton btn;
    QScriptValue scriptBtn = engine.newQObject(&btn);
    engine.globalObject().setProperty("button", scriptBtn);

    QFile file("script.js");
    file.open(QIODevice::ReadOnly);
    engine.evaluate(file.readAll());
    file.close();

    return app.exec();
}
```

Листинг 21.5. Файл chap21/qscript3/script.js

```
button.text = 'Hello!';
button.styleSheet = 'color: red;'
                  + 'font-size: 20px;'
                  + 'font-style: italic;';
button.show();
```

Как видите, мы свободно обращаемся к свойствам объекта `QPushButton` и вызываем его методы.

ЗАДАЧА. С помощью скрипта реализовать обработку сигналов, генерируемых объектом библиотеки Qt из программы на C++.

Решение. Допустим, сигнал соединен с каким-то слотом в тексте программы на C++, но при выполнении этого слота происходит обращение к некоторой функции, определенной текстом скрипта. Если данный текст хранится в отдельном файле, то редактируя его, можно изменить поведение программы даже после ее компиляции.

Метод вызова из программы на C++ безымянной функции, определенной текстом скрипта, иллюстрирует листинг 21.6.

Листинг 21.6

```
QScriptEngine engine;
QScriptValue fun = engine.evaluate("function(a, b){return a + b;}");
QScriptValueList args; // Список аргументов
args << QScriptValue(&engine, 5) << QScriptValue(&engine, 7);
QScriptValue y = fun.call(QScriptValue(), args);   // 5 + 7 = 12
qDebug() << "Result =" << y.toNumber();
```

Рассмотрим пример программы (рис. 21.3, листинги 21.7–21.10). Первоначальный текст надписи на кнопке определяется сриптом `init.js` (см. листинг 21.9). После каждого нажатия выполняется функция, заданная в скрипте `onclick.js` (см. листинг 21.10). В нашем случае данная функция увеличивает счетчик на единицу и отображает на кнопке его новое значение.

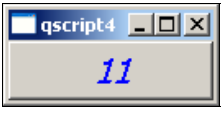


Рис. 21.3. При каждом нажатии кнопки счетчик увеличивается на единицу

Листинг 21.7. Файл chap21/qscript4/qscript4.h

```
#include <QtGui>
#include <QtScript>

class MyWin : public QMainWindow {
    Q_OBJECT
    Q_PROPERTY(qint32 counter WRITE setCounter READ getCounter)
public:
    MyWin();
public slots:
    void onClick();
    qint32 getCounter() const;
    void setCounter(qint32 n);
private:
    QPushButton *btn;
    QScriptEngine *engine;
    QScriptValue onClickHandle;
    qint32 counter;
};
```

Листинг 21.8. Файл chap21/qscript4/qscript4.cpp

```
#include <QtDebug>
#include "qscript4.h"

MyWin::MyWin()
     : QMainWindow(0) {

    btn = new QPushButton(this);
    setCentralWidget(btn);

    engine = new QScriptEngine(this);

    QScriptValue scriptBtn = engine->newQObject(btn);
    engine->globalObject().setProperty("btn", scriptBtn);

    QScriptValue scriptWin = engine->newQObject(this);
    engine->globalObject().setProperty("win", scriptWin);

    setCounter(0);

    QFile file("onclick.js");
    if (file.open(QIODevice::ReadOnly)){
        onClickHandle = engine->evaluate(file.readAll());
```

```
        file.close();
    }else
        qDebug() << "Cannot open file 'onclick.js'";

    file.setFileName("init.js");
    if (file.open(QIODevice::ReadOnly)){
        engine->evaluate(file.readAll());
        file.close();
    }else
        qDebug() << "Cannot open file 'init.js'";

    connect(btn, SIGNAL(clicked()),
            this, SLOT(onClick()) );
}

void MyWin::onClick() {
    qDebug() << "click";
    if (not onClickHandle.isNull())
        onClickHandle.call();
    else
        qDebug() << "null handle";
}

qint32 MyWin::getCounter() const {
    return counter;
}

void MyWin::setCounter(qint32 n){
    counter = n;
}

//---------------------------------
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    MyWin win;
    win.show();
    return app.exec();
}
```

Листинг 21.9. Файл chap21/qscript4/init.js

```
btn.text = 'Hello!';
btn.styleSheet = 'color: blue;'
```

```
                + 'font-size: 20px;'
                + 'font-style: italic;';
```

Листинг 21.10. Файл chap21/qscript4/onclick.js

```
function() {
    n = win.counter;
    win.counter = ++n;
    btn.text = n;
}
```

Заметим, что "слот-обертка" `MyWin::onClick()` для указателя на скриптовую функцию `MyWin::onClickHandle` вовсе не обязателен: при желании можно было связать сигнал `clicked()` непосредственно со скриптовой функцией, если вместо

```
connect(btn, SIGNAL(clicked()), this, SLOT(onClick()) );
```

написать:

```
qScriptConnect(btn, SIGNAL(clicked()), scriptWin, onClickHandle);
```

и исключить из текста листингов 21.7 и 21.8 упоминание слота `MyWin::onClick()`. Обратите внимание, что макрос `SLOT()` в последнем случае не используется.

Если нам не нравится, что обработчик нажатия на кнопку задан в отдельном файле и не имеет имени, то можно немного модифицировать текст конструктора (листинг 21.11).

Листинг 21.11

```
 1 MyWin::MyWin() : QMainWindow(0) {
 2    QFile file("init.js");
 3    if (file.open(QIODevice::ReadOnly)){
 4       engine->evaluate(file.readAll());
 5       file.close();
 6       onClickHandle = engine->globalObject().property("onclick");
 7    }else
 8       qDebug() << "Cannot open file 'init.js'";
 9
10    connect(btn, SIGNAL(clicked()), this, SLOT(onClick()) );
```

Здесь мы с помощью метода `property()` обращаемся к определенной в скрипте функции по ее имени (6). Теперь весь скрипт может размещаться в единственном файле `init.js` (листинг 21.12).

Листинг 21.12

```
function onclick() {
   n = win.counter;
   win.counter = ++n;
   btn.text = n;
}

btn.text = 'Hello!';
btn.styleSheet = 'color: blue;'
               + 'font-size: 20px;'
               + 'font-style: italic;';
```

(Полный текст всех файлов проекта можно найти на прилагаемом компакт-диске в каталоге chap21/qtscript4a.)

Другой вариант решения нашей задачи показан в листингах 21.13–21.15. Здесь, в отличие от предыдущей программы, сигнал `clicked()` связывается с именованной функцией, определенной в тексте скрипта, причем эта связь задана в самом скрипте.

Листинг 21.13. Файл chap21/qscript5/qscript5.h

```
#include <QtGui>
#include <QtScript>

class MyWin : public QMainWindow {
    Q_OBJECT
    Q_PROPERTY(qint32 counter WRITE setCounter READ getCounter)
public:
    MyWin();
public slots:
    qint32 getCounter() const;
    void setCounter(qint32 n);
private:
    QPushButton *btn;
    QScriptEngine *engine;
    QScriptValue onClickHandle;
    qint32 counter;
};
```

Листинг 21.14. Файл chap21/qscript5/qscript5.cpp

```
#include <QtDebug>
#include "qscript5.h"
```

```
MyWin::MyWin()
     : QMainWindow(0) {

    btn = new QPushButton(this);
    setCentralWidget(btn);

    engine = new QScriptEngine(this);

    QScriptValue scriptBtn = engine->newQObject(btn);
    engine->globalObject().setProperty("btn", scriptBtn);

    QScriptValue scriptWin = engine->newQObject(this);
    engine->globalObject().setProperty("win", scriptWin);

    setCounter(0);

    QFile file("init.js");
    if (file.open(QIODevice::ReadOnly)){
        engine->evaluate(file.readAll());
        file.close();
    }else
        qDebug() << "Cannot open file 'init.js'";
}

qint32 MyWin::getCounter() const {
    return counter;
}

void MyWin::setCounter(qint32 n){
    counter = n;
}

//---------------------------------
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    MyWin win;
    win.show();
    return app.exec();
}
```

Листинг 21.15. Файл chap21/qscript5/init.js

```
function onclick() {
    n = win.counter;
```

```
    win.counter = ++n;
    btn.text = n;
}

btn.text = 'Hello!';
btn.styleSheet = 'color: blue;'
                + 'font-size: 20px;'
                + 'font-style: italic;';
btn["clicked()"].connect(onclick); //или btn.clicked.connect(onclick);
```

В последней строке мы связали сигнал `clicked()` (нажатие на кнопку) с функцией `onclick()`.

Наконец, рассмотрим последний вариант: окно с кнопкой может быть создано не в программе на C++, а в самом скрипте (листинги 21.16–21.18).

Листинг 21.16. Файл chap21/qscript6/qscript6.h

```
#include <QtGui>
#include <QtScript>

class MyWin : public QMainWindow {
    Q_OBJECT
    Q_PROPERTY(qint32 counter WRITE setCounter READ getCounter)
    Q_PROPERTY(QWidget *btn READ getBtn)
public:
    MyWin(QWidget *parent=0);
public slots:
    qint32 getCounter() const;
    void setCounter(qint32 n);
    QPushButton* getBtn() const;
private:
    QPushButton *btn;
    qint32 counter;
};

Q_SCRIPT_DECLARE_QMETAOBJECT(MyWin, QWidget*)
Q_DECLARE_METATYPE(MyWin*)
```

Листинг 21.17. Файл chap21/qscript6/qscript6.cpp

```
#include <QtDebug>
#include "qscript6.h"
```

```
MyWin::MyWin(QWidget *parent)
     : QMainWindow(parent) {

    btn = new QPushButton(this);
    setCentralWidget(btn);

    setCounter(0);
}

qint32 MyWin::getCounter() const {
    return counter;
}

void MyWin::setCounter(qint32 n){
    counter = n;
}

QPushButton* MyWin::getBtn() const {
    return btn;
}

//-----------------------------------
int main(int argc, char *argv[]) {
    QApplication app(argc, argv);

    qRegisterMetaType<MyWin*>("MyWin");

    QScriptEngine engine;

    QScriptValue scriptMyWin=engine.scriptValueFromQMetaObject<MyWin>();
    engine.globalObject().setProperty("MyWin", scriptMyWin);

    QFile file("init.js");
    if (file.open(QIODevice::ReadOnly)){
        engine.evaluate(file.readAll());
        file.close();
    }else
        qDebug() << "Cannot open file 'init.js'";

    return app.exec();
}
```

Листинг 21.18. Файл chap21/qscript6/init.js

```
function onclick() {
    n = win.counter;
    win.counter = ++n;
    btn.text = n;
}

var win = new MyWin;
var btn = win.btn;
btn.text = 'Hello!';
btn.styleSheet = 'color: blue;'
               + 'font-size: 20px;'
               + 'font-style: italic;';
btn["clicked()"].connect(onclick);
win.show();
```

Предлагаем читателю в качестве несложного упражнения решить следующую задачу, а потом сравнить свое решение с нашим.

ЗАДАЧА. Дополнить таблицу базы данных, рассмотренную в *разд. 18.1*, новым столбцом "Премия". Величина премии должна рассчитываться по произвольной формуле, заданной конечным пользователем программы, исходя из оклада работника и его семейного положения (рис. 21.4).

	Табельн. номер	Имя	День рождения	Зарплата	Женат/ замужем	Премия
1	№123	Винни-Пух	31.12.1971	100.00	☐ Нет	20.00
2	№345	Ослик Иа	23.02.1965	200.00	☑ Да	40.00
3	№501	Пятачок	14.06.1982	50.00	☐ Нет	5.00

Рис. 21.4. Премия рассчитывается с помощью внешней функции

Решение. Допустим, холостым сотрудникам мы установили премию в размере 10%, а семейным — 20% от оклада. Тогда пользовательский скрипт будет выглядеть так, как показано в листинге 21.19.

Листинг 21.19. Функция для расчета премии (файл chap21/db02a/script.js)

```
function fun(salary, married) {
 print(salary, married);
```

```
 if (married)
   return salary * 0.2;
 else
   return salary * 0.1;
}
```

Вторая строка здесь добавлена только для отладки (мы выводим на консольный экран значения фактических параметров вызова).

В тексте самой программы (см. листинги 18.1 и 18.2) останется переопределить виртуальную функцию `QSqlQueryModel::columnCount()`, чтобы увеличить на единицу число столбцов таблицы; ввести новые переменные в раздел `private` класса `MyModel`:

```
QScriptEngine *engine;
QScriptValue glob;
```

в конструкторе `MyModel` инициализировать их, как показано в листинге 21.20, а также учесть новый столбец в методе `data()` (листинг 21.21).

Листинг 21.20

```
MyModel::MyModel(QObject *parent)
               : QSqlQueryModel(parent) {

   engine = new QScriptEngine(this);
   QFile file("script.js");
   if (file.open(QIODevice::ReadOnly)){
       engine->evaluate(file.readAll());
       file.close();
       glob = engine->globalObject();
   }else{
       glob = engine->nullValue();
   }
   refresh();
}
```

Листинг 21.21

```
QVariant MyModel::data(
          const QModelIndex &index,
          int role) const {

  QVariant value = QSqlQueryModel::data(index, role);
```

```
switch (role) {

case Qt::DisplayRole: // Данные для отображения
case Qt::EditRole:    // Данные для редактирования
    if (index.column() == 0)
        return value.toString().prepend(tr("№"));
    else if (index.column() == 2 && role == Qt::DisplayRole)
        return value.toDate().toString("dd.MM.yyyy");
    else if (index.column() == 3 && role == Qt::DisplayRole)
        return tr("%1")
               .arg(value.toDouble(), 0, 'f', 2);
    else if (index.column() == 4)
        return value.toInt() != 0 ? tr("Да") : tr("Нет");

    else if (index.column() == 5){ // Вычисляемый столбец!
        if (glob.isValid()) { // Если скрипт загружен
            QScriptValueList args; // Список аргументов функции

            QModelIndex j = createIndex( // из 3-го столбца (оклад)
                index.row(), 3, index.internalPointer());
            QString s = data(j, Qt::DisplayRole).toString();
            args << QScriptValue(engine, s);

            j = createIndex( //из 4-го столбца (семейное положение)
                index.row(), 4, index.internalPointer());
            bool m = QSqlQueryModel::data(
                j, Qt::DisplayRole).toBool();
            args << QScriptValue(engine, m);

            // Вызываем функцию из скрипта:
            QScriptValue f = glob.property("fun").call(glob, args);
            return tr("%1").arg(f.toNumber(), 0, 'f', 2);
        }else
            return QVariant(0);
    }else
        return value;
   . . .
```

Здесь мы опустили проверку того, имеется ли в загруженном скрипте определение функции с нужным именем. При желании такую проверку можно добавить, аналогично тому, как это сделано в примере на прилагаемом компакт-диске: chap21/qtscript4a/qscript4.cpp.

Обратите внимание, что для столбца "Семейное положение" наша модель возвращает строку "Да" или "Нет", поэтому передачу скриптовой функции

логического значения мы осуществили методом родительского класса `QSqlQueryModel::data()` вместо `data()`. Разумеется, можно было использовать просто `data()` с параметром `Qt::CheckStateRole`.

Метод `setData()` переделывать не требуется, ведь вычисляемая колонка у нас нередактируемая (метод `flags()` мы не исправляли).

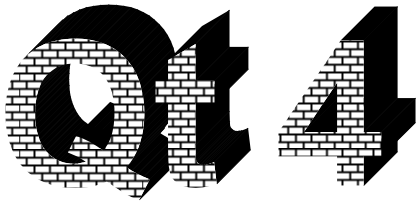
Можете проверить, что при изменении данных в столбце "Оклад" или "Семейное положение" соответствующее значение в столбце "Премия" сразу же пересчитывается. Благодаря отладочному оператору `print` в теле скриптовой функции, мы можем убедиться, что обращение к ней происходит только в том случае, если какие-то ячейки вычисляемой колонки видны на экране, но пересчет значений вызывается даже при изменении размеров окна.

Полные исходные тексты программы можно найти на прилагаемом компакт-диске в каталоге chap21/db02a.

Замечание

Разумеется, для простых алгоритмов расчета вместо добавления вычисляемой колонки можно изменять загружаемый из файла текст SQL-запроса.

ГЛАВА 22



Технология COM. Интеграция с MS Office. Средства мультимедиа

Если хочешь доставить удовольствие, научись его получать.

Уильям Хэзлитт

Коммерческая версия библиотеки *Qt Desktop Edition* для Windows содержит модуль `ActiveQt`, который позволяет работать с объектами ActiveX, а также создавать такие объекты. Модуль `ActiveQt` состоит из следующих подмодулей:

- `QAxContainer` — контейнеры для объектов COM. Содержит определения классов `QAxObject` (невизуальные объекты COM), `QAxWidget` (визуальные объекты COM), `QAxScript` (работа со скриптами);
- `QAxServer` — серверы COM.

22.1. Импорт и экспорт таблиц MS Excel

Сначала рассмотрим работу с невизуальными объектами `QAxObject`.

ЗАДАЧА. Выгрузить в HTML-файл описание свойств и методов COM-объекта `Excel.Application`.

Решение иллюстрирует листинг 22.1.

Листинг 22.1

```
#include <QAxObject>
. . .
QFile file("excel.html");
```

```
QTextStream out( &file );
file.open( QIODevice::WriteOnly | QIODevice::Truncate );
QAxObject excel = QAxObject("Excel.Application");
QString d = excel.generateDocumentation();
out << d;
file.close();
```

В файл описания проекта *.pro необходимо добавить строку

```
CONFIG += qaxcontainer
```

Объекты, методы и константы, описанные в генерируемом здесь HTML-файле, понадобятся нам при решении следующих двух задач.

ЗАДАЧА. Загрузить данные из файла Excel в таблицу `QTableWidget` (рис. 22.1).

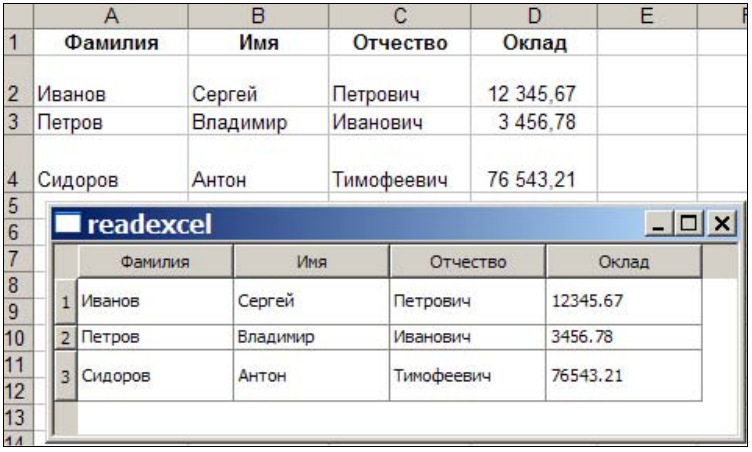


Рис. 22.1. Импорт данных и оформления из файла Microsoft Excel в таблицу `QTableWidget`

Решение иллюстрирует листинг 22.2.

Листинг 22.2. Файл chap25/readexcel/readexcel.cpp

```
#include <QtGui>
#include <QAxObject>
#include <QtDebug>

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

```

```
 8    QTableWidget table;
 9    table.show();
10
11    QAxObject *excel = new QAxObject( "Excel.Application",
12                                      &table );
13    excel->dynamicCall( "SetVisible(bool)", false );
14    QAxObject *workbooks = excel->querySubObject( "WorkBooks" );
15    QAxObject *workbook = workbooks->querySubObject( "Open(QString)",
16                                                     "c:/data.xls" );
17    if (workbook) {
18        QAxObject *sheets = workbook->querySubObject( "Sheets" );
19        QAxObject *sheet = sheets->querySubObject("Item(1)");
20        sheet->dynamicCall( "Select()" );
21
22        int r = 0;
23        table.setColumnCount(4);
24        table.setRowCount(0);
25        QString v = "";
26        while(r >= 0) {
27            for (int c=0; c<4 && r>=0; ++c){
28                QAxObject *cell = sheet->querySubObject(
29                    "cells(int, int)", r+1, c+1 );
30                cell->dynamicCall( "Select()" );
31
32                if (cell)
33                    v = cell->property("Value").toString();
34                else
35                    v = "";
36
37                qDebug() << r << c << v;
38                if (c == 0 && r > 0){
39                    if (v.length() != 0 && r < 10000) {
40                        table.insertRow(table.rowCount());
41                        QAxObject *row = sheet->querySubObject(
42                            "rows(int)", r+1 );
43                        int points = row->property("Height").toInt();
44                        int pixels = points * 1.2;
45                        table.setRowHeight(r-1, pixels);
46                    }else
47                        r = -11111;
48                }
49                if (r >= 0){
50                    QTableWidgetItem *item = new QTableWidgetItem(v);
```

```
51                      if (r == 0)
52                          table.setHorizontalHeaderItem(c, item);
53                      else
54                          table.setItem(r-1, c, item);
55                  }
56              }
57              r++;
58          }
59      }
60      excel->dynamicCall("Quit()");
61      return app.exec();
62  }
```

Здесь мы читаем только содержимое ячеек (значение свойства `Value` ячейки `cell`) и высоту строк (свойство `Height` строки `row`). Аналогично можно обрабатывать другие параметры ячеек таблицы: шрифт, выравнивание и т. п. Высоту строки можно было бы получить и через свойство `Height` ячейки `cell`, но мы хотели показать, как работать с различными объектами рабочего листа Excel. В строках (20) и (30) нужный рабочий лист и очередная ячейка таблицы делаются текущими, в результате пользователь может следить за процессом перебора ячеек, но это замедляет работу программы.

ЗАДАЧА. Выгрузить таблицу `QTableWidget` в файл Excel.

Решение.

- подключаем модуль `QAxObject`:

```
#include <QAxObject>
```

- в файл описания проекта *.pro добавляем строку

```
CONFIG += qaxcontainer
```

- в соответствующем методе своего класса, порожденного от `QTableWidget`, создаем COM-объект Excel:

```
QAxObject* excel = new QAxObject("Excel.Application", this );
```

- делаем окно Excel видимым на экране (это дает пользователю возможность при необходимости отредактировать и сохранить результат работы программы):

```
excel->dynamicCall( "SetVisible(bool)", true );
```

- создаем новую рабочую книгу:

```
QAxObject *workbooks = excel->querySubObject( "Workbooks" );
QAxObject *workbook = workbooks->querySubObject( "Add()" );
```

- создаем новый рабочий лист и делаем его текущим:

```
QAxObject *sheets = workbook->querySubObject( "Sheets" );
QAxObject *sheet = sheets->querySubObject("Item(1)");
sheet->dynamicCall( "Select()" );
```

- устанавливаем размер шрифта по умолчанию:

```
sheet->dynamicCall("SetStandardFontSize(double)", 9);
```

- вставляем заголовок таблицы в ячейку `A1`:

```
QAxObject *cell = sheet->querySubObject( "cells(1, 1)");
cell->dynamicCall( "Select()" );
cell->setProperty( "Value", tr("Hello! Привет!") );
// или: cell->dynamicCall( "Value", tr("...") );
```

- очищаем, если требуется, диапазон ячеек:

```
QAxObject *range = sheet->querySubObject(
     "Range(const QVariant&)", QVariant( QString("A1:D12")));
range->dynamicCall( "Clear()" );
```

- заголовки строк таблицы формирует листинг 22.3;
- столбцы таблицы — листинг 22.4;
- обрамление таблицы — листинг 22.5.

Листинг 22.3. Заголовки строк таблицы

```
QString s;
for(int r = 0; r < rowCount(); ++r) {
    s = headerData(r, Qt::Vertical, Qt::DisplayRole).toString();
    if (s.size() > 0) {
        QAxObject *cell = sheet->querySubObject(
            "cells(int, int)", r+4, 1 );
        QAxObject *font = cell->querySubObject("Font");
        font->setProperty( "Size", 9 );
        font->setProperty( "Bold", true);
        cell->setProperty( "HorizontalAlignment",  -4152 );
        cell->setProperty( "Value", s );
    }
}
```

Листинг 22.4. Столбцы таблицы

```
for(int c = 0; c < columnCount; ++c) {
    cell = sheet->querySubObject(
        "cells(int, int)", 3, c+2 );
```

```
    // Заголовок столбца:
    s = headerData(c, Qt::Horizontal, Qt::DisplayRole).toString();
    QAxObject *font = cell->querySubObject("Font");
    font->setProperty( "Size", 9 );
    font->setProperty( "Bold", true);
    cell->setProperty( "HorizontalAlignment",  -4108 );
    cell->setProperty( "VerticalAlignment",  -4108 );
    cell->setProperty( "Value", s );

    // Ширина столбца:
    QAxObject *row = sheet->querySubObject("Columns(int)", c+2);
    row->setProperty( "ColumnWidth", 14);

   // Ячейки текущего столбца таблицы:
    for(int r = 0; r < rowCount(); ++r) {
        s = data(index(r, c), Qt::DisplayRole).toString();
        if (s.size() > 0) {
            QAxObject *cell = sheet->querySubObject(
                "cells(int, int)", r+4, c+2 );
            //cell->dynamicCall( "Select()" );
            cell->setProperty( "Value", s );
        }
    }
}
```

Листинг 22.5. Обрамление таблицы

```
// левый верхний угол:
QAxObject *cell1 = sheet->querySubObject(
    "cells(int, int)", 3, 1 );

// правый нижний угол:
QAxObject *cell2 = sheet->querySubObject(
    "cells(int, int)", rowCount()+3, columnCount()+1 );

// диапазон ячеек:
QAxObject *range = sheet->querySubObject(
    "Range(const QVariant&, const QVariant&)",
    QVariant(cell1->dynamicCall("Address()")),
    QVariant(cell2->dynamicCall("Address()")) );

// обрамление:
QAxObject *borders = range->querySubObject( "Borders" );
borders->setProperty( "LineStyle", 1 );
```

22.2. Работа со звуком

Любое искусство стремится к тому, чтобы стать музыкой.

Уолтер Пейтер

С выходом релиза 4.4 наконец-то можно сказать, что в библиотеке Qt появилась поддержка мультимедиа. Если до этого возможность воспроизведения музыкальных файлов ограничивалась только форматом WAVE, а со словом "анимация" было связано лишь отображение анимированных картинок GIF и MNG, то теперь, с появлением в библиотеке Qt модуля Phonon, появились средства для работы с большинством распространенных форматов аудио- и видеофайлов.

ЗАДАЧА. Необходимо подать звуковой сигнал.

Решение. Можно использовать статический метод `QApplication::beep()`.

ЗАДАЧА. Воспроизвести звуковой файл WAVE.

Решение. Для асинхронного проигрывания звуковых файлов имеется класс `QSound`. Два варианта работы с ним иллюстрирует листинг 22.6.

Листинг 22.6

```
// 1-й вариант
   QSound::play("sounds/file1.wav"); // Проиграть звуковой файл
// 2-й вариант
   if ( ! QSound::isAvailable() ) {
      // Звуковая подсистема недоступна
   }else{
      QSound sound("sounds/file1.wav"); // Имя файла
      sound.setLoops(5); // Число циклов воспроизведения
      sound.play();      // Играть
   }
```

Управлять громкостью звука, к сожалению, нельзя. В системе Windows поддерживается только звуковой формат WAVE, а функция `QSound::isAvailable()` всегда возвращает `true`, даже если в компьютере нет звуковой карты. В системе X11 используется Network Audio System (если она установлена) и поддерживаются форматы WAVE и AU. В Qtopia Core можно проигрывать только звуковые файлы формата WAVE.

Число циклов воспроизведения устанавливается с помощью метода `setLoops(n)`. При `n=-1` файл будет проигрываться до бесконечности. В любой момент времени можно узнать, сколько еще циклов воспроизведения оста-

лось, для этого предназначена функция `loopsRemaining()`. Если воспроизведение звукового файла было прервано, то метод `isFinished()` возвращает `true` (в системе Windows для объекта с однократным циклом воспроизведения всегда возвращается `true`). Чтобы прервать воспроизведение звука, нужно вызвать метод `stop()`. Обращаем внимание, что в системе Windows при этом текущий цикл воспроизведения не прекратится сразу, как можно было бы предполагать, а доиграет до конца звукового файла (если установленное число циклов больше единицы, но не бесконечно). Поэтому более надежный способ остановить звучание сразу — уничтожить сам объект `sound`, но для этого он должен быть создан с помощью конструктора:

```
QSound *sound = new QSound("ИмяФайла.wav", родитель);
```

Второй вариант решения этой задачи — использование модуля Phonon.

22.3. Анимация

ЗАДАЧА. Отобразить анимированное графическое изображение.

Решение. В библиотеке Qt имеются средства для воспроизведения анимированных изображений в форматах GIF и MNG, для этого предназначен класс `QMovie` (листинг 22.7).

Листинг 22.7

```
QLabel label;
QMovie *movie = new QMovie("file.gif");
label.setMovie(movie);
movie->start();
// . . .
movie->stop();
```

Также в составе Qt имеется модуль `QtSvg`, предназначенный для отображения файлов в формате SVG, который, в свою очередь, поддерживает анимированные изображения. Пример воспроизведения анимированного файла SVG показан в листинге 22.8.

Листинг 22.8. Воспроизведение анимированного файла SVG

```
#include <QtGui>
#include <QtSvg>
int main(int argc, char **argv){
    QApplication app(argc, argv);
    QSvgWidget svg("movie.svg");
```

```
    svg.show();
    QObject::connect( svg.renderer(), SIGNAL(repaintNeeded()),
                      &svg,           SLOT(repaint()) );
    return app.exec();
}
```

Перед компиляцией проекта в файл *.pro необходимо добавить строку

```
QT += svg
```

22.4. Библиотека Phonon

Начиная с версии 4.4, в библиотеку Qt входит модуль Phonon, предназначенный для работы с видео и звуком. Для работы с ним в Windows потребуется DirectX 9 (или выше) и DirectX SDK, а в Linux — библиотека GStreamer версии не ниже 0.10. Перед компиляцией библиотеки Qt в Linux рекомендуется установить все доступные пакеты, связанные с GStreamer, чтобы получить максимально возможное число форматов, доступных для воспроизведения. Если нужные библиотеки установлены, то при запуске `configure` опция `-phonon` включена по умолчанию.

ЗАДАЧА. Воспроизвести видеофайлы AVI, WMV, MPEG, MOV, FLV, …

Решение. В листингах 22.9–22.11 приведен текст программы, а на рис. 22.2 показан внешний вид окна во время воспроизведения видео.

Рис. 22.2. Внешний вид окна при воспроизведении видео

Листинг 22.9. Файл chap22/video/mainwin.h

```
#ifndef _MAINWIN_H_
#define _MAINWIN_H_

#include <QtGui>
#include <phonon>

// using namespace Phonon; // если не хотим писать Phonon::xxx

class Mainwin : public QMainWindow {
    Q_OBJECT
public:
    Mainwin();
private slots:
    void selectfile(); // выбор файла для воспроизведения
    void playfile();   // воспроизведение видеофайла
    void setvolume(int); // установка уровня громкости
private:
    Phonon::VideoPlayer *player; // виджет-видеопроигрыватель
    QToolBar *toolbar;           // панель инструментов
    QPushButton *openbutton,  // кнопка "Открыть"
                *playbutton,  // кнопка "Играть"
                *pausebutton, // кнопка "Пауза"
                *stopbutton;  // кнопка "Стоп"
    QSlider *volumeslider; // ползунок для установки громкости
    Phonon::QSlider *seekslider; // ползунок для отображения и
                        // установки текущей позиции в видеофайле
};
#endif // _MAINWIN_H_
```

Листинг 22.10. Файл chap22/video/mainwin.cpp

```
#include <QtGui>
#include "mainwin.h"

Mainwin::Mainwin(){
    QWidget *widget = new QWidget(this); // Центральный виджет
    QVBoxLayout *lay = new QVBoxLayout(widget);

    // Создаем виджет - проигрыватель видеофайлов:
    player = new Phonon::VideoPlayer(Phonon::VideoCategory);
    lay->addWidget(player, 1); // будет растягиваться
```

```
    // Создаем ползунок для отображения и установки
    // текущей позиции в видеопотоке:
    seekslider = new Phonon::SeekSlider;

    // Связываем ползунок с проигрывателем:
    seekslider->setMediaObject(player->mediaObject());
    lay->addWidget(seekslider, 0); // без растяжения

    // Создаем панель инструментов:
    toolbar = addToolBar(tr("toolbar"));

    // Создаем кнопку "Открыть":
    openbutton = new QPushButton(tr("Open"), toolbar);
    // Добавляем кнопку к панели инструментов:
    toolbar->addWidget(openbutton);
    // Связываем кнопку со слотом:
    connect(openbutton, SIGNAL(clicked()), this, SLOT(selectfile()));

    // Аналогично для кнопки "Играть":
    playbutton = new QPushButton(tr("Play"), toolbar);
    toolbar->addWidget(playbutton);
    connect(playbutton, SIGNAL(clicked()), this, SLOT(playfile()));

    // Аналогично для кнопки "Стоп":
    stopbutton = new QPushButton(tr("Stop"), toolbar);
    toolbar->addWidget(stopbutton);
    connect(stopbutton, SIGNAL(clicked()), player, SLOT(stop()));

    // Аналогично для кнопки "Пауза":
    pausebutton = new QPushButton(tr("Pause"), toolbar);
    toolbar->addWidget(pausebutton);
    connect(pausebutton, SIGNAL(clicked()), player, SLOT(pause()));

    // Ползунок для установки уровня громкости:
    volumeslider = new QSlider(Qt::Horizontal, toolbar);
    volumeslider->setRange(0, 100);
    volumeslider->setValue(100);
    connect(volumeslider, SIGNAL(valueChanged(int)), this,
            SLOT(setvolume(int)));
    toolbar->addWidget(volumeslider);

    widget->setLayout(lay);
    setCentralWidget(widget);
}
```

```
void Mainwin::selectfile(){
    QString filename = QFileDialog::getOpenFileName(
        this, tr("Select video file"), ".");

    if (filename.isEmpty())
        return;

    player->play(filename);
    resize(sizeHint());
}

void Mainwin::playfile(){
    player->play();
    resize(sizeHint());
}

void Mainwin::setvolume(int value){
    float volume;
    volume = (static_cast<float>(value)) / 100;
    player->setVolume(volume);
}
```

Листинг 22.11. Файл chap22/video/main.cpp

```
#include <QtGui>
#include <mainwin.h>

int main(int argv, char **args){
    QApplication app(argv, args);
    app.setApplicationName("Video Player");
    Mainwin win;
    win.show();
    return app.exec();
}
```

Сразу после выполнения команды `qmake -project` перед вторым запуском `qmake` в файл *.pro необходимо добавить строку

```
QT += phonon
```

ЗАДАЧА. Воспроизвести звуковые файлы MP3, WAVE, AU, …

Решение. Начиная с релиза 4.4, среди примеров программ, поставляемых вместе с библиотекой Qt, имеется простое приложение "Музыкальный про-

игрыватель", исходные тексты которого можно найти в каталоге examples/phonon/musicplayer.

22.5. Воспроизведение видео с помощью Media Player

ЗАДАЧА. Воспроизвести видеофильм с помощью Windows Media Player (рис. 22.3).

Рис. 22.3. Использование Windows Media Player

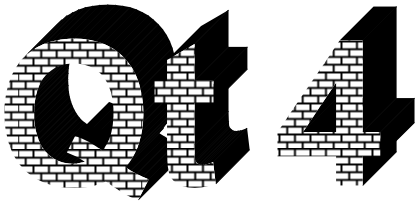
Решение иллюстрирует листинг 22.12.

Листинг 22.12. Файл chap22/mplayer/mplayer.cpp

```
#include <QtGui>
#include <QAxWidget>

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

    QAxWidget mp = QAxWidget("MediaPlayer.MediaPlayer");
    // или QAxWidget("{22D6F312-B0F6-11D0-94AB-0080C74C7E95}");

    mp.setProperty("ShowControls", true);
    mp.resize(500,400);
    mp.show();
    mp.dynamicCall("Open(QString)", "video.wmv");
    mp.dynamicCall("Play()");

    return app.exec();
}
```

Здесь для работы с Media Player мы используем технологию COM.

ГЛАВА 23



Внешний вид элементов управления

Красота — как драгоценный камень: чем она проще, тем драгоценнее.

Фрэнсис Бэкон

23.1. Палитры

Класс `QPalette` (*палитра*) хранит информацию о цвете для любого визуального элемента интерфейса, для трех его возможных состояний: активного (*active*, элемент имеет фокус ввода), неактивного (*inactive*) и недоступного (*disabled*).

ЗАДАЧА. Изменить цвет элемента интерфейса.

Решение. Необходимо сначала получить копию палитры для данного элемента:

```
QPalette pal = widget.palette();
```

Если требуется изменить палитру всего приложения, то используйте

```
QPalette pal = qApp->palette();
```

После этого можно изменять любые элементы палитры, например, цвет:

```
pal.setColor(group, role, color);
```

или

```
pal.setColor(role, color);
```

Здесь `group` — это группа цветов (`QPalette::Active`, `QPalette::Inactive` или `QPalette::Disabled`); `role` — роль элемента палитры:

- `QPalette::Window` — цвет фона окна;
- `QPalette::WindowText` — цвет переднего плана;

- `QPalette::Base` — цвет фона для полей ввода (обычно белый);
- `QPalette::AlternateBase` — альтернативный цвет фона, устанавливаемый методом `QAbstractItemView::setAlternatingRowColors()` для четных строк таблиц и списков;
- `QPalette::Text` — цвет текста (обычно черный);
- `QPalette::Button` — цвет фона для кнопок;
- `QPalette::ButtonText` — цвет текста на кнопках

и т. д.

Последний параметр `color` (цвет) определяется именем (`Qt::white`, `Qt::black`, `Qt::lightGray`, `Qt::darkBlue` и т. д.) или разложением на составляющие в системе координат RGB (red-green-blue), HSV (hue-saturation-value) или CMYK (cyan-magenta-yellow-black). Если группа не указана, то заданный цвет назначается для всех трех возможных состояний. Например:

```
pal.setColor(QPalette::Text, Qt::darkBlue);
```

или

```
pal.setColor(QPalette::Button, QColor(50, 40, 180));
```

Кроме цвета, для палитры можно изменить тип кисти, например:

```
pal.setBrush(QPalette::Window, QBrush(Qt::cyan, Qt::CrossPattern));
```

Наконец, после всех модификаций нужно связать палитру с данным элементом:

```
widget.setPalette( pal );
```

или со всем приложением:

```
qApp->setPalette( pal );
```

Необходимо помнить, что эксперименты с палитрами, как правило, портят внешний вид приложения, поэтому прибегать к ним стоит только в особых случаях.

ЗАДАЧА. Изменить цвет переключателя (рис. 23.1).

Решение задачи показано в листинге 23.1.

Рис. 23.1. Внешний вид переключателя

Листинг 23.1. Файл chap23/palette/palette.cpp

```
#include <QApplication>
#include <QWidget>
#include <QRadioButton>

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

    QWidget win;
    QRadioButton *button = new QRadioButton("RadioButton", &win);
    button->move(10,10);

    QPalette pal = button->palette();

    // Цвет фона круга:
    pal.setColor(QPalette::Base, QColor(100,100,255));

    // Цвет пометки внутри круга:
    pal.setColor(QPalette::Text, Qt::red);

    // Цвет надписи:
    pal.setColor(QPalette::WindowText, QColor(30,160,20));

    button->setPalette(pal);

    win.show();
    return app.exec();
}
```

23.2. Стили

> В делах первостепенной важности самое главное — стиль, а не искренность.
>
> *Оскар Уайльд*

Для управления внешним видом (*look and feel*) приложения в библиотеке Qt используется базовый класс `QStyle` и производные от него классы `QWindowsStyle`, `QWindowsXPStyle`, `QWindowsVistaStyle`, `QCleanlooksStyle`, `QMacStyle`, `QPlastiqueStyle`, `QCDEStyle`, `QMotifStyle`.

ЗАДАЧА. Изменить стиль интерфейса приложения.

Решение. Имеется несколько вариантов.

Вариант 1. При запуске программы в командной строке можно указать ключ `-style` и через пробел название стиля: `Windows`, `WindowsXP`, `WindowsVista`, `Cleanlooks`, `Mac`, `Plastique`, `CDE` или `Motif`.

Замечание

Стиль `Mac` доступен только в системе Mac OS X, стили `WindowsXP` и `WindowsVista` — только в системе Windows при выборе соответствующего стиля оформления (**Пуск | Настройка | Панель управления | Экран | Оформление | Окна и кнопки**).

Вариант 2. Можно воспользоваться статическим методом `QApplication::setStyle()`, например

```
QApplication::setStyle(new QWindowsStyle);
```

или

```
QApplication::setStyle(new QPlastiqueStyle);
```

Заметим, что старый объект стиля при этом автоматически уничтожается. Пример создания меню для программного изменения стиля приложения мы уже встречали в *разд. 23.1*.

Вариант 3. Существует возможность изменить стиль отдельного элемента интерфейса, используя для него метод `setStyle()`, хотя по эстетическим соображениям так поступать не рекомендуется.

Вариант 4. Можно "заставить" приложение отказаться от стандартных системных цветов и шрифтов, как показано в листинге 23.2 (по умолчанию они используются).

Листинг 23.2

```
int main( int argc, char *argv[] ) {
    QApplication::setDesktopSettingsAware(false);
    QApplication app(argc, argv);
    // ...
    return app.exec();

}
```

Вызов метода `QApplication::setDesktopSettingsAware()` обязательно должен предшествовать созданию объекта приложения `QApplication`.

Вариант 5. Можно создать текстовый файл, например:

```
QMenuBar { background: red; }
QToolButton::hover { border: 4px solid blue; }
```

и подключить его при запуске программы, указав ключ `-stylesheet`:

```
МояПрограмма -stylesheet ИмяСтилевогоФайла
```

В результате строка меню окрасится в красный цвет, а при наведении указателя мыши на любую кнопку вокруг нее будет показываться синее обрамление. Аналогично изменяется внешний вид других элементов интерфейса.

ЗАДАЧА. Как из сложного элемента управления исключить какой-либо составной элемент?

Решение. Можно разработать свой класс стиля, в котором переопределить метод `drawComplexControl()`. Например, удалим рамку вокруг группы `QGroupBox` (рис. 23.2, листинг 23.3).

Рис. 23.2. Группа `QGroupBox` без рамки

Листинг 23.3. Файл chap23/removesub/removesub.cpp

```
#include <QtGui>
class MyStyle : public QWindowsStyle {
    Q_OBJECT
public:
    MyStyle() {}
    void drawComplexControl(ComplexControl control,
                            const QStyleOptionComplex *option,
                            QPainter *painter,
                            const QWidget *widget = 0 ) const {
        if (const QStyleOptionGroupBox *group =
            qstyleoption_cast<const QStyleOptionGroupBox*>(option)){
            QStyleOptionGroupBox newGroupBox(*group);
            newGroupBox.subControls &= ~SC_GroupBoxFrame;
            QWindowsStyle::drawComplexControl(
                control, &newGroupBox, painter, widget);
        } else {
            QWindowsStyle::drawComplexControl(
                control, option, painter, widget);
        }
    }
};
```

```
#include "removesub.moc"

int main(int argc, char **argv){
    QApplication app(argc, argv);
    QWidget widget;
    QHBoxLayout *layout = new QHBoxLayout(&widget);

    // Обычная группа с рамкой:
    QGroupBox *box1 = new QGroupBox(&widget);
    box1->setTitle("box1");
    layout->addWidget(box1);

    // Наша группа без рамки:
    QGroupBox *box2 = new QGroupBox(&widget);
    box2->setStyle(new MyStyle()); // Применили свой стиль
    box2->setTitle("box2");
    layout->addWidget(box2);

    widget.show();
    return app.exec();
}
```

ЗАДАЧА. Как изменить внешний вид какой-либо части сложного элемента управления?

Решение. Можно разработать свой класс стиля, в котором переопределить метод `drawComplexControl()`. Например, изменим вид кнопки у поля с раскрывающимся списком `QComboBox` (рис. 23.3, листинг 23.4).

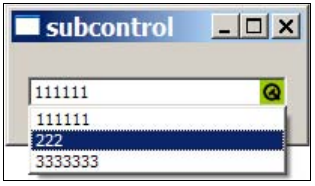


Рис. 23.3. Изменение вида кнопки для раскрытия списка

Листинг 23.4. Файл chap23/subcontrol1/subcontrol.cpp

```
#include <QtGui>
class MyStyle : public QWindowsStyle {
public:
   MyStyle() {}
```

```
   void drawComplexControl(
          ComplexControl control,
          const QStyleOptionComplex *option,
          QPainter *painter, const QWidget *widget=0 ) const{

       if (const QStyleOptionComboBox *group =
          qstyleoption_cast<const QStyleOptionComboBox*>(option)) {
          QStyleOptionComboBox newComboBox(*group);

          // Удаляем стандартную кнопку:
          newComboBox.subControls &= ~QStyle::SC_ComboBoxArrow;

          // Рисуем элемент без кнопки:
          QWindowsStyle::drawComplexControl(
              control, &newComboBox, painter, widget);

          // Дорисовываем кнопку:
          QRect rect = subControlRect(
              control, option, SC_ComboBoxArrow, widget);
          QPixmap pix = standardPixmap(
              QStyle::SP_TitleBarMenuButton, option, widget);
          painter->save();
          painter->setBrushOrigin(rect.topLeft());
          painter->fillRect(rect, QBrush(pix));
          painter->restore();
      } else {
          QWindowsStyle::drawComplexControl(
              control, option, painter, widget);
      }
   }
};

int main(int argc, char **argv){
   QApplication app(argc, argv);
   QWidget widget;
   QVBoxLayout *layout = new QVBoxLayout(&widget);
   QComboBox *cb = new QComboBox(&widget);
   cb->addItem("111111");
   cb->addItem("222");
   cb->addItem("3333333");
   cb->setStyle(new MyStyle()); // Применили свой стиль
   layout->addWidget(cb);
   widget.show();
   return app.exec();
}
```

Другой вариант решения той же задачи — использование таблицы стилей (*Style Sheet*) (рис. 23.4, листинг 23.5).

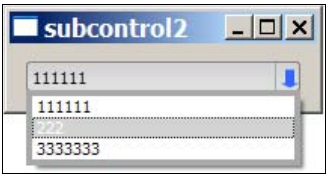


Рис. 23.4. Использование таблиц стилей для изменения внешнего вида элемента управления

Листинг 23.5. Файл chap23/subcontrol2/subcontrol.cpp

```
#include <QtGui>

int main(int argc, char **argv){
    QApplication app(argc, argv);
    QString s =
        " QComboBox {"
        "   border: 1px solid gray; "
        "   border-radius: 3px; "
        "   padding: 1px 18px 1px 3px; "
        "   min-width: 6em; "
        " } "
        " QComboBox:editable {"
        "   background: white; "
        " } "
        " QComboBox:!editable, QComboBox::drop-down:editable {"
        "   background: qlineargradient( "
        "       x1: 0, y1: 0, x2: 0, y2: 1, "
        "       stop: 0 #E1E1E1, stop: 0.4 #DDDDDD,"
        "       stop: 0.5 #D8D8D8, stop: 1.0 #D3D3D3); "
        " } "
        " QComboBox:!editable:on, QComboBox::drop-down:editable:on{"
        "   background: qlineargradient( "
        "       x1: 0, y1: 0, x2: 0, y2: 1, "
        "       stop: 0 #D3D3D3, stop: 0.4 #D8D8D8, "
        "       stop: 0.5 #DDDDDD, stop: 1.0 #E1E1E1); "
        " } "
        // Смещение текста в поле ввода при раскрывшемся списке:
        " QComboBox:on { "
        "   padding-top: 3px; "
```

```
        "   padding-left: 3px; "
        " } "
        " QComboBox::drop-down { "
        "   subcontrol-origin: padding; "
        "   subcontrol-position: top right; "
        "   width: 15px; "
        "   border-left-width: 1px; "
        "   border-left-color: darkgray; "
        "   border-left-style: solid; "
        "   border-top-right-radius: 3px; "
        "   border-bottom-right-radius: 3px; "
        " }"
        " QComboBox::down-arrow { "
        "   image: url(downarrow.png); " // значок для кнопки
        " }"
        " QComboBox::down-arrow:on {" //смещение для нажатой кнопки
        "   top: 1px; "
        "   left: 1px; "
        " }"
        " QComboBox QAbstractItemView {" //для раскрывающегося списка
        "   border: 4px solid darkgray;"
        "   selection-background-color: lightgray;"
        "}";
    app.setStyleSheet(s);
    QWidget widget;
    QVBoxLayout *layout = new QVBoxLayout(&widget);
    QComboBox *cb = new QComboBox(&widget);
    cb->addItem("111111");
    cb->addItem("222");
    cb->addItem("3333333");

    layout->addWidget(cb);
    widget.show();
    return app.exec();
}
```

ЗАДАЧА. Увеличить размеры стандартных элементов интерфейса, например, для сенсорного экрана.

Решение. Минимальные размеры всех элементов можно установить с помощью статического метода

```
QApplication::setGlobalStrut(QSize(w, h));
```

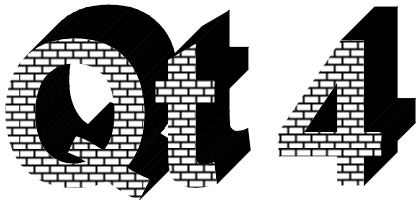
ЗАДАЧА. Увеличить размеры определенных элементов интерфейса, например, полос прокрутки `QScrollBar`.

Решение. Придется создать свой стиль на основе любого потомка класса `QStyle` и определить для него виртуальный метод

```
pixelMetric( PixelMetric metric,
             const QStyleOption * option = 0,
             const QWidget * widget = 0 ) const
```

чтобы он возвращал нужное значение, когда первый аргумент равен `QStyle::PM_ScrollBarExtent`, после чего можно применить новый стиль с помощью метода `setStyle()`.

ГЛАВА 24



Интернационализация приложений

> Если сказано хорошо, то не все ли равно, на каком языке?
>
> *Лопе де Вега*

Чтобы приложение можно было легко *локализовать* (перевести на другие языки), следует:

- использовать статическую функцию `QObject::tr()` для всех строковых литералов (не строковых переменных!), которые должны быть переведены на другой язык (в том числе для клавиатурных сокращений, обозначений валют, форматов представления дат и, возможно, для названий каталогов и имен файлов);
- с помощью программы `Qt Linguist`, а также утилит `lupdate` и `lrelease` подготовить файлы перевода на нужные языки;
- в начале программы с помощью функции `QTranslator::load()` загрузить нужный файл перевода *.qm и установить его в качестве рабочего, вызвав функцию `QCoreApplication::installTranslator()`.

Рассмотрим все эти вопросы подробнее.

24.1. Подготовка исходного текста программы

Для загрузки в программу файлов перевода на нужный язык в библиотеке Qt имеется класс `QTranslator`. Обычно требуется, как минимум, два файла перевода: для своего приложения и для сообщений самой библиотеки Qt (листинг 24.1).

Листинг 24.1

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

   QTranslator qtTranslator; // "переводчик" для библиотеки Qt
   qtTranslator.load(
      ":/qt_" + QLocale::system().name(), // системная локаль
      QLibraryInfo::location(QLibraryInfo::TranslationsPath) );
   app.installTranslator(&qtTranslator);

   QTranslator myAppTranslator; // "переводчик" для нашей программы
   myAppTranslator.load(":/translations/myapp_" +
                        QLocale::system().name());
   app.installTranslator(&myAppTranslator);
   ...
   return app.exec();
}
```

Здесь считается, что файлы перевода хранятся в программе в виде ресурсов. Разумеется, это могут быть и обычные файлы (тогда надо удалить символы `:/` перед именем файла).

Статическая функция `QLocale::system()` возвращает текущую локаль системы, а метод `QLocale::name()` — строку вида `язык_страна`, где подстрока `язык` — два символа в нижнем регистре (международный стандарт ISO 639), подстрока `страна` — два заглавных символа (стандарт ISO 3166). Далее через точку может указываться кодировка. Например, для российского варианта русского языка в кодировке KOI8-R эта комбинация выглядит так: `ru_RU.KOI8-R`. Если именно такая строка указана в качестве первого параметра функции `QTranslator::load()`, то сначала она попытается найти в каталоге, указанном вторым параметром, файл *_ru_RU.KOI8-R (сначала с расширением qm, потом без него). Если файл перевода не найден, будет сделана попытка загрузить файл *_ru_RU (опять-таки, сначала с расширением qm, а затем, если такого файла нет, без расширения). Наконец, если и это невозможно — то файл *_ru (с расширением или без). Как правило, при переводе интерфейса приложения формируют только файл *_язык, а файлы *_язык_СТРАНА имеет смысл подготавливать только в тех случаях, когда требуется какой-нибудь диалект, например, канадский вариант французского языка: *_fr_CA. Если файл перевода так и не найден, то никаких сообщений об ошибке не выводится, просто в интерфейсе приложения используются те строки, которые указаны в исходных текстах.

Чтобы создать файлы перевода, необходимо сначала пометить в исходном тексте программы все строковые константы, которые нужно будет переводить на другой язык. Для этого в большинстве случаев подходит уже знакомая нам функция `tr()`.

Функцию `tr()` обычно применяют даже в программах, которые не собираются локализовывать. Если перевод для какой-либо строки недоступен (т. е. файл перевода не загружен или перевод данной строки в нем отсутствует), то пользователь увидит исходную версию строки.

Для литералов, зависящих от значений некоторых переменных, указывают шаблонные символы `%1`, `%2`,... и функцию `arg()`, например:

```
tr("Уже в %1-й раз пытаюсь открыть файл '%2'!").arg(n).arg(filename)
```

Во втором необязательном параметре функции `tr()` программист может при необходимости указать любую строку — комментарий переводчику.

Если мы хотим использовать функцию `tr()` внутри класса, не являющегося наследником `QObject`, то придется либо везде писать `QObject::tr()`, либо определить свой макрос

```
#define  tr   QObject::tr
```

либо упомянуть в объявлении класса специальный макрос

```
Q_DECLARE_TR_FUNCTIONS(ИмяКласса)
```

который делает функцию `tr()` членом этого класса. Последний вариант наиболее предпочтителен, т. к. позволяет сохранить текущий контекст (имя класса, в котором встретилась данная строка). Ведь эта же строка может встретиться и еще в каком-нибудь классе, где ее, возможно, потребуется перевести по-другому. Пример иллюстрирует листинг 24.2.

Листинг 24.2

```
class MyClass {
   Q_DECLARE_TR_FUNCTIONS(MyClass)
public:
   MyClass();
   ...
};
```

Функцию `tr()` (например, в главной программе) можно заменить методом

```
QCoreApplication::translate( "СтрокаДляПеревода",
                             "НеобязательныйКомментарий")
```

Если очень хочется применить функцию `tr()` не к строковому литералу, а к строковой переменной, то для того, чтобы пометить строку, присвоенную

этой переменной, задействуют специальный макрос `QT_TR_NOOP` (листинг 24.3).

Листинг 24.3

```
void OrderForm::init(){
  static const char * const flowers[] = {
      QT_TR_NOOP("Розы"),
      QT_TR_NOOP("Ромашки"),
      QT_TR_NOOP("Тюльпаны"),
      0
  };

  for (int i = 0; flowers[i]; ++i)
      comboBox->addItem(tr(flowers[i]));
}
```

Если присваивание производится не внутри метода класса, то применяют макрос

```
QT_TRANSLATE_NOOP ( "Контекст", "СтрокаДляПеревода" )
```

Чтобы программист случайно не забыл использовать функцию `tr()` или какой-либо из ее аналогов в своей программе, лучше запретить неявное преобразование `const char*` в `QString`, для этого в файл описания проекта *.pro нужно добавить строку

```
DEFINES += QT_NO_CAST_FROM_ASCII
```

Тогда ко всем строковым литералам, которые не требуется переводить, придется применять функцию `QLatin1String()`.

Как правило, все строковые литералы в программе, предполагаемой в дальнейшем переводить на другие языки, записываются по-английски, с применением только 128 стандартных символов кодовой таблицы ASCII. Хотя это и не является обязательным условием, но позволяет избежать некоторых проблем. Если в строке встречаются "не ASCII"-символы, Trolltech рекомендует указывать Esc-последовательности, например:

```
label->setText(tr("F\374r \310lise"));
```

24.2. Подготовка файлов описания проекта и ресурсов

Когда все файлы с исходным текстом программы уже подготовлены, непосредственно перед или даже после компиляции программы, можно начать

переводить ее на другой язык. Для этого сначала следует добавить в файл описания проекта *.pro строку `TRANSLATIONS` и перечислить в ней имена необходимых файлов *.ts (по одному файлу для каждого языка, на который предполагается переводить интерфейс приложения) (листинг 24.4).

Листинг 24.4

```
HEADERS          = file1.h \
                   file2.h
SOURCES          = file1.cpp \
                   file2.cpp \
                   main.cpp
FORMS            = form1.ui form2.ui
RESOURCES        = myapp.qrc
TRANSLATIONS     = translations/myapp_ru.ts \
                   translations/myapp_fr.ts \
                   translations/myapp_de.ts
```

Здесь предполагается, что для хранения файлов перевода выделен каталог `translations`. Данный каталог предварительно необходимо создать.

Скомпилированный файл перевода, который получится в конце процесса локализации, будет иметь расширение qm. При желании его можно поместить в исполняемый файл программы в виде ресурса, для этого в XML-файл описания ресурсов `myapp.qrc` нужно будет добавить строки, приведенные в листинге 24.5.

Листинг 24.5

```
<RCC>
<qresource>
   <file>translations/myapp_ru.qm</file>
   <file>translations/myapp_fr.qm</file>
   <file>translations/myapp_de.qm</file>
   <file>translations/qt_ru.qm</file>
   <file>translations/qt_fr.qm</file>
   <file>translations/qt_de.qm</file>
</qresource>
</RCC>
```

Файлы qt_*.qm в этом случае нужно будет скопировать из каталога Qtxxx/translations в каталог myapp/translations. Что касается файлов myapp_*.qm, то они появятся только после выполнения перевода всех строк в программе Qt Linguist (см. *разд. 24.4*).

Если в программе встречаются строковые литералы, записанные с использованием символов национальных алфавитов, отличных от стандартной латиницы, то перед первым вызовом функции `tr()` устанавливается соответствующий кодек (листинг 24.6).

Листинг 24.6

```
int main(int argc, char *argv[]) {
   QApplication app(argc, argv);
   QTextCodec::setCodecForTr(QTextCodec::codecForName("CP1251"));
   ...
}
```

Тогда нужно указать этот же кодек в файле *.pro:

```
CODECFORTR      = CP1251
CODECFORSRC     = CP1251
```

Последняя строка нужна только для Microsoft Visual C++ 2005.

Часто при переводе интерфейса программы приходится изменять не только строковые надписи, но и изображения на кнопках. Например, на панели инструментов форматирования в нелокализованной программе "Текстовый редактор" имеется кнопка с изображением курсивной буквы ***I*** (от слова *Italic* — курсив). Разумеется, в испанском варианте интерфейса мы захотим заменить ее буквой ***C*** (от слова *Cursivo*), в немецком — буквой ***K*** (*Kursiv*), а в русском — буквой ***К***, которая хоть и выглядит практически так же, как и ***K*** из латинского алфавита, но все же соответствует другому символу кодовой таблицы и во многих шрифтах отображается несколько иначе. Поэтому либо в программе придется заранее предусмотреть несколько вариантов (листинг 24.7), либо указать все эти варианты в файле ресурсов (листинг 24.8).

Листинг 24.7

```
if (tr("Italic")[0] == 'C') {            // испанский
   italicAction->setIcon(iconC);
} else if (tr("Italic")[0] == 'K') {     // немецкий
   italicAction->setIcon(iconK);
} else if (tr("Italic")[0] == tr('К'){ // русский (здесь латинская K)
   italicAction->setIcon(iconKru);
} else {
   italicAction->setIcon(iconI);  // для всех остальных случаев
}
```

Листинг 24.8

```
<qresource>
   <file>italic.png</file>
</qresource>
<qresource lang="es">
  <file alias="italic.png">cursivo.png</file>
</qresource>
<qresource lang="de">
  <file alias="italic.png">kursiv.png</file>
</qresource>
<qresource lang="ru">
   <file alias="italic.png">kursivru.png</file>
</qresource>
```

Здесь для всех пиктограмм принят один и тот же псевдоним (*alias*), поэтому в исходном тексте программы фигурирует только имя файла `:/italic.png`. Разумеется, второй вариант намного лучше первого, т. к. при разработке текста программы обычно заранее трудно предвидеть, на какие языки ее впоследствии придется переводить.

Нельзя забывать и то обстоятельство, что существуют языки, на которых пишут справа налево (например, арабский, иврит), поэтому придется "изменить ориентацию" всего приложения:

```
qApp->setLayoutDirection(Qt::RightToLeft);
```

Как правило, соответствующий признак хранится в самом файле перевода *.qm, поэтому функция `QApplication::setLayoutDirection()` вызывается автоматически при установке объекта `QTranslator`. Но это влечет за собой дополнительные переделки. Например, стрелки на кнопках **Вперед** и **Назад**, возможно, придется поменять местами (листинг 24.9).

Листинг 24.9

```
// 1-й вариант
   if (QApplication::isRightToLeft()) {
      backAction->setIcon(forwardIcon);
      forwardAction->setIcon(backIcon);
   } else {
      backAction->setIcon(backIcon);
      forwardAction->setIcon(forwardIcon);
   }
```

```
// 2-й (предпочтительный) вариант
   <qresource lang="ru">
      <file alias="forward.png">forward.png</file>
      <file alias="back.png">back.png</file>
   </qresource>
   <qresource lang="en">
      <file alias="forward.png">forward.png</file>
      <file alias="back.png">back.png</file>
   </qresource>
   <qresource lang="ar">
      <file alias="forward.png">back.png</file>
      <file alias="back.png">forward.png</file>
   </qresource>
   <qresource lang="iw">
      <file alias="forward.png">back.png</file>
      <file alias="back.png">forward.png</file>
   </qresource>
```

24.3. Утилиты *lupdate* и *lrelease*

После того как исходные тексты программы, а также файлы *.pro и *.qrc подготовлены, можно запустить на выполнение утилиту `lupdate`, которая находится в каталоге Qtxxx/bin:

```
lupdate myapp.pro
```

Эта утилита просматривает исходный текст программы и записывает все помеченные с помощью функции `tr()` или ее аналогов строковые литералы в отдельный XML-файл *.ts. Количество таких файлов и их имена утилита узнает из строки `TRANSLATIONS` файла *.pro. Файлы *.ts затем следует по очереди открыть в программе Qt Linguist, чтобы задать перевод каждой строки на нужный язык.

Если в файле *.ts уже задан перевод строк, то скомпилированный файл перевода *.qm можно сформировать посредством утилиты `lrelease`. В командной строке при ее запуске нужно указать файл описания проекта *.pro.

24.4. Программа Qt Linguist

Программа Qt Linguist находится в каталоге Qtxxx/bin. При ее запуске из командной строки можно сразу указать имя файла *.ts:

```
linguist myfile.ts
```

а можно загрузить файл перевода позже, воспользовавшись командой меню **File** (Файл) | **Open** (Открыть).

Интерфейс программы Qt Linguist достаточно прост (рис. 24.1). В левой части окна располагается список контекстов (классов, определенных в исходных текстах программы), справа вверху — список строк для перевода, имеющихся в текущем контексте, а под ним — окно для редактирования перевода текущей строки. После того как мы впишем текст перевода, нужно нажать

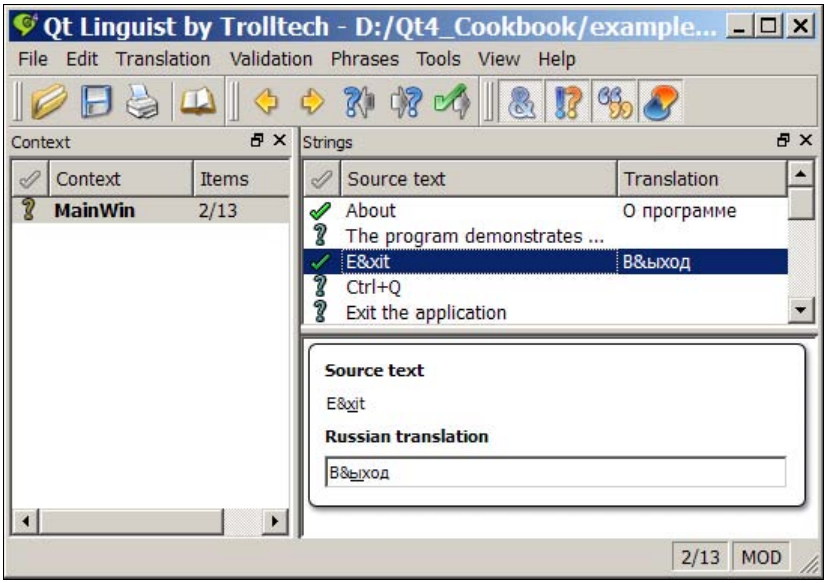


Рис. 24.1. Программа Qt Linguist

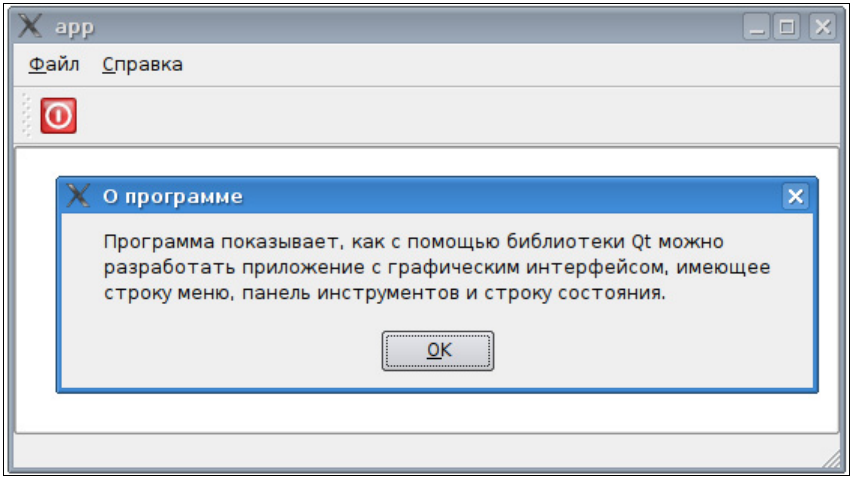


Рис. 24.2. Результат локализации: пункты меню и текст сообщений теперь можно прочитать по-русски

кнопку **Mark item as done** (Пометить пункт как завершенный) на панели инструментов (кнопка с зеленой галочкой). Еще не отмеченные таким образом пункты обозначены знаком вопроса. Если требуется отменить поставленную галочку, то можно щелкнуть по ней левой кнопкой мыши. Заданные переводы можно сохранить, выполнив команду меню **File** | **Save**.

Когда все строки будут переведены (по крайней мере те, что мы захотели перевести), следует выполнить пункт меню **File** (Файл) | **Release** (Релиз). В результате будет сформирован файл *.qm.

В каталоге chap24/app1 на прилагаемом компакт-диске можно найти исходные тексты и файл перевода для простого текстового редактора, описанного в *разд. 12.1*. На рис. 24.2 показан результат русификации программы (сравните его с рис. 12.1).

24.5. Переключение языка во время выполнения программы

Иногда язык интерфейса (вернее, соответствующий файл перевода) должен не только выбираться при запуске программы, но также изменяться в процессе ее выполнения. Например, когда в диалоге настройки параметров предусмотрен выбор языка пользователем и мы хотим, чтобы изменение текущего языка отразилось на интерфейсе приложения не после его перезагрузки, как часто бывает, а немедленно. В этом случае мы должны, во-первых, выгрузить из памяти старые файлы перевода:

```
qApp->removeTranslator(qtTranslator);
qApp->removeTranslator(myAppTranslator);
```

во-вторых, загрузить новые:

```
qtTranslator.load(...);
qApp->installTranslator(qtTranslator);
myAppTranslator.load(...);
qApp->installTranslator(myAppTranslator);
```

и в-третьих, перерисовать все отображенные в настоящее время элементы интерфейса с учетом нового языка. Это можно сделать вручную, вызвав некоторую функцию, предназначенную для смены языка интерфейса. Разумеется, предварительно придется поместить код для изменения всех объектов, связанных с интерфейсом приложения, в отдельную функцию (листинг 24.10).

Листинг 24.10

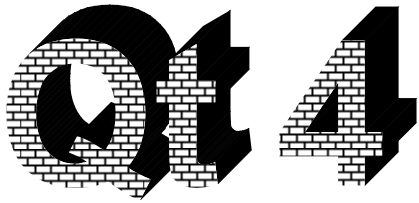
```
void MyMainWin::retranslateUi(){
    createAction->setText(tr("&Create"));
    createAction->setStatusTip(tr("Create a new file"));
    ...
    exitAction->setText(tr("E&xit"));
    exitAction->setShortcut(tr("Ctrl+Q"));
    ...
    fileMenu->setTitle(tr("&File"));
    editMenu->setTitle(tr("&Edit"));
    helpMenu->setTitle(tr("&Help"));

    setWindowTitle(tr("..."));
}
```

А можно с самого начала проектировать все визуальные элементы таким образом, чтобы они автоматически воспринимали событие изменения языка приложения (листинг 24.11).

Листинг 24.11

```
void MyWidget::changeEvent(QEvent *event){
    if (e->type() == QEvent::LanguageChange) {
        createAction->setText(tr("&Create"));
        createAction->setStatusTip(tr("Create a new file"));
        ...
    } else
        QWidget::changeEvent(event);
}
```

ГЛАВА 25



Использование интерпретатора Python

Границы моего языка — это границы моего мира.
Людвиг Витгенштейн

25.1. Зачем это нужно

Если нашему приложению нужен интерпретатор, то мы можем воспользоваться `QtScript`. Но JavaScript, к сожалению, не является полноценным объектно-ориентированным языком. К тому же, нельзя сказать, что на этом языке имеется исчерпывающее количество прикладных библиотек, которые позволяют проводить обработку текста и изображений, выполнять громоздкие вычисления, строить графики, формировать печатные формы документов, работать с базами данных и сетевыми протоколами... Если мы хотим получить не только встроенный интерпретатор, но также и доступ к готовым библиотекам "на все случаи жизни", то не можем пройти мимо языка Python.

25.2. Основные сведения о языке Python

Тот, кто учится самостоятельно, преуспевает в семь раз больше, чем тот, которому все объяснили.
Артур Гитерман

Python — язык программирования высокого уровня с динамической типизацией, автоматическим управлением памятью и удобными высокоуровневыми структурами данных, такими как словари (хеш-таблицы), списки и кортежи. Бóльшая часть реализаций Python — это интерпретаторы, но имеется также возможность компиляции программ. Python поддерживает классы, модули (которые могут быть объединены в пакеты), обработку исключений, многопоточные вычисления. Язык поддерживает несколько парадигм программи-

рования: структурное, объектно-ориентированное, функциональное, аспектно-ориентированное, обладает простым и выразительным синтаксисом. Неоспоримое преимущество Python — существенное сокращение труда программиста, а также объема исходных текстов программ. Разумеется, за все приходится платить: программы, написанные на этом языке, выполняются несколько медленнее, чем C/C++/Qt-приложения.

В системах Linux/FreeBSD язык Python встречается гораздо чаще, чем в мире Windows. Например, на нем написан инсталлятор Anaconda (Red Hat Linux и Fedora Core).

Как интерпретатор языка Python, так и все его многочисленные прикладные библиотеки (за некоторым исключением) распространяются свободно и могут использоваться без ограничений при создании свободного или проприетарного программного обеспечения. Для получения более подробной информации, а также для загрузки инсталлятора читатель может обратиться к сайту **www.python.org**.

Установка Python в Windows или Linux не должна вызвать затруднений, а интерактивные оболочки IDLE и PythonWin для редактирования и отладки скриптов имеют интуитивно понятный интерфейс. Чтобы читатель получил хотя бы общее представление об особенностях языка, приведем небольшой скрипт на Python (листинг 25.1).

Листинг 25.1. Типичная структура программы на языке Python

```
#!/usr/bin/env python
# -*- coding: cp1251 -*-

# Импорт внешних модулей:
import module1
from module2 import *
from module3 import f3
.............

# Определение функций:
def func1(a, b):
    if a < b:
        return a
    else:
        return b

..............
```

```
# Определение классов и методов:
class Employee:                  # Класс "Работник".
    def __init__(self, name, salary):  # Конструктор.
        self.name = name
        self.salary = salary

    def getSalary(self):         # Узнать зарплату.
        return self.salary

..............

# Главная программа (точка входа при выполнении скрипта):
if __name__ == '__main__':

    # Вызываем функцию f1, импортированную из модуля module1:
    result1 = module1.f1( 23.34, 'Строка1' )

    # Вызываем функцию f2, импортированную из модуля module2:
    result2 = f2( 23.5, "Строка2" )

    # Вызываем функцию f3, импортированную из модуля module3:
    result3 = f3( 10, "Строка3")

    # Вызываем функцию func1, определенную в нашем модуле:
    result4 = func1( result1, result2 )

    # Создаем экземпляр класса Employee с помощью конструктора:
    man1 = Employee("Иванов Иван Иванович", 4507.35)

    # Вызываем метод getSalary для экземпляра man1:
    print man1.getSalary()

    ..............

    # Если надо ждать нажатие клавиши <Enter> перед закрытием:
    raw_input("Press ENTER...")

    # Конец программы никак не обозначается.
```

Непривычная особенность языка Python — значимость отступов в начале строки.

Другой пример иллюстрирует листинг 25.2.

Листинг 25.2. Простая программа с линейной структурой

```
#!/usr/bin/env python
# -*- coding: cp1251 -*-

"Вычисляем сумму двух чисел."

# Вводим первое число:
x = input("x = ")

# Вводим второе число:
y = input("y = ")

# Вычисляем сумму:
z = x + y

# Выводим результат:
print "%f + %f = %f" % (x, y, z)

# Ждем нажатия <Enter>:
raw_input("Press Enter...")
```

25.3. Встраивание интерпретатора Python в приложения, написанные на C/C++

Пусть наше приложение написано на языке C или C++ и требуется вызвать интерпретатор Python, чтобы выполнить пользовательский скрипт. Для решения этой задачи можно использовать стандартный программный интерфейс Python/C API. С его помощью задача решается в три действия:

- `Py_Initialize()` — загружаем динамическую библиотеку Python;
- `PyRun_SimpleString()` — выполняем любой фрагмент кода Python;
- `Py_Finalize()` — выгружаем динамическую библиотеку Python.

Приведем простейший пример выполнения строк исходного текста Python в консольной программе, написанной на языке C (листинг 25.3).

Листинг 25.3. Выполнение строк скрипта Python из программы, написанной на C

```
#include <stdio.h>
#include "Python.h"
```

```
int main(int argc, char *argv[]){

    // Инициализация интерпретатора Python:
    Py_Initialize();

    // Выполнение строки программы на Python:
    // print '\n%d + %d = %d\n' % (2, 3, 2+3)
    PyRun_SimpleString("print '\\n%d+%d=%d\\n' % (2,3,2+3)");

    // Выгрузка интерпретатора:
    Py_Finalize();

    return 0;
}
```

В системе Windows компилировать программу на C нужно с опцией `/MD` для Multithreaded DLL или `/MDd` для Debug Multithreaded DLL (в Microsoft Visual C++ 2005 для этого необходимо установить параметр **Runtime Library**, находящийся в разделе **Code Generation** группы **C/C++**). В первом случае программа при выполнении будет загружать файл pythonXX.dll, во втором — pythonXX_d.dll. Кроме того, компилятору необходимо указать каталог для заголовочных файлов `PythonXX/include`, а компоновщику — каталог для поиска библиотек `PythonXX/libs` или конкретную библиотеку: `pythonXX.lib` для Microsoft Visual C++, `libpythonXX.a` для MinGW. Вместо символов `XX` пишут номер установленной версии Python. Единственный файл, который потребуется программе во время ее выполнения, — это динамическая библиотека Python.

Листинг 25.4 содержит более сложный пример использования Python в программе, написанной на C.

Листинг 25.4. Пример встраивания Python в C-программу (файл chap25/00/00.c)

```
// Пример встраивания Python в программу на C/C++.
// Выполнение отдельных строк исходного текста Python
// без передачи результата в вызывающую программу.

 1  #include<stdio.h>
 2  #include "Python.h"
 3
 4  int main(int argc, char *argv[]){
 5     int err;
 6     char *s;
 7
```

```
 8     printf("\nEmbedded Python.\n");
 9
10     // Инициализация интерпретатора Python:
11     Py_Initialize();
12
13     // Проверка, инициализирован ли интерпретатор:
14     if ( Py_IsInitialized() ) {
15
16        // Номер версии Python:
17        printf("Python started. Version %s\n", Py_GetVersion());
18
19     } else {
20
21        // Если была ошибка инициализации Python:
22        printf("\nCannot initialize Python, aborting");
23        exit(1);
24     }
25
26     // Выполнение отдельной строки программы на Python:
27     // print '\n%d+%d=%d' % (2, 3, 2+3)
28     err = PyRun_SimpleString("print '\\n%d+%d=%d\\n' % (2,3,2+3)");
29     if (err)    // Если была ошибка:
30        printf("\nError executing Python string!\n");
31
32     // Можно передавать несколько строк, разделенных "\n",
33     // можно импортировать модули Python
34     // (вывод Python направляется в консольное окно программы C):
35     PyRun_SimpleString("from time import time, ctime\n"
36                        "print 'Today is ', ctime( time() )");
37
38     // Можно передавать строки по отдельности,
39     // они все равно выполняются в одном контексте __main__:
40     PyRun_SimpleString("import os");
41     PyRun_SimpleString("print '\\nCurrent dir is ', os.getcwd()");
42
43     // Импорт пользовательского модуля и вызов функции,
44     // определенной в нем:
45     printf("\nImporting Python module 'mytest'\n");
46     PyRun_SimpleString("import mytest");
47     PyRun_SimpleString("x = 5; y = 2");
48     PyRun_SimpleString("print '%d+%d=%d' % (x,y,mytest.f1(x,y))");
49
```

```
50      // Запуск интерактивного сеанса интерпретатора Python
51      // в консольном окне вызывающей программы:
52      printf("\nStarting Python session\n");
53      printf("Press 'Ctrl+Z' and <Enter> for exit\n");
54      Py_Main(argc, argv);
55      // Выход из сеанса интерпретатора произойдет после
56      // нажатия комбинации клавиш 'Ctrl+Z' и <Enter>.
57
58      // Ждать нажатия <Enter>:
59      printf("\nPress <Enter>...");
60      fgetc(stdin);
61
62      // Выгрузка интерпретатора:
63      Py_Finalize();
64
65      return 0;
66  }
```

Пояснения к программе:

- (14) — после вызова `Py_Initialize()` мы сначала проверяем, как прошла инициализация;
- (17) — выводим на экран номер версии интерпретатора Python;
- (28) — выполняем одиночную строку кода Python;
- (29) — проверяем на наличие ошибки;
- (35, 36) — единственному вызову `PyRun_SimpleString()` передаем сразу несколько строк кода, разделенные символом `\n`;
- (40, 41) — при выполнении последовательности строк кода Python используются отдельные вызовы `PyRun_SimpleString()`. Заметим, что выполнение всех строк происходит в одном контексте `__main__`;
- (46–48) — мы импортируем собственный модуль `mytest.py` и вызываем функцию `f1`, определенную в нем;
- (54) — вызывается интерактивный сеанс интерпретатора, который будет происходить в консольном окне С-программы. В ответ на приглашение `>>>` можно выполнять любые команды Python. При вводе неправильной команды интерпретатор выводит сообщение об ошибке и продолжает работу.

Для завершения сеанса нужно нажать комбинацию клавиш <Ctrl>+<Z>, а затем <Enter>.

Содержимое файла mytest.py может быть любым. Главное, чтобы это была правильно оформленная программа на языке Python (листинг 25.5).

Листинг 25.5

```
def f1(a, b, c=0):
   return a + b + c

if __name__ == '__main__':
   print "\nExecuting '__main__'!\n"
   x = float(raw_input("x = "))
   y = float(raw_input("y = "))
   print "%f+%f=%f" % (x, y, f1(x,y))
```

Если в скрипте окажется ошибка, то в консольное окно программы будет выведено обычное сообщение Python об ошибке, а программа на C продолжит свою работу. Разумеется, файл скрипта при выполнении программы необходимо поместить в каталог, который Python просматривает при поиске импортируемых библиотек. Проще всего, если он будет располагаться в том же каталоге, что и сама программа.

ЗАДАЧА. Выполнить из программы, написанной на C/C++, фрагмент кода Python, хранящийся в отдельном текстовом файле.

Решение. В Python/C API имеется функция

```
int PyRun_SimpleFile( FILE *fp, const char *filename )
```

которая работает аналогично `PyRun_SimpleString()`, только исходный текст скрипта Python берется не из строки, а из файла с именем `filename`. При вызове функции `PyRun_SimpleFile()` требуется указать файловый дескриптор `fp`, который создается с помощью обычного вызова C

```
fp = fopen(filename, "r")
```

Однако передавать интерпретатору Python файловый дескриптор, созданный внешней программой, довольно опасно. Дело в том, что разные компиляторы C/C++ создают различные структуры `FILE`. В результате, если компилирует программу не тот же самый компилятор, с помощью которого собиралась библиотека python.dll, то при выполнении программы может возникнуть ошибка и она аварийно завершится.

Поэтому вместо того, чтобы напрямую загружать скрипт на выполнение с помощью `PyRun_SimpleFile()`, в некоторых случаях лучше средствами C/C++ читать файл строка за строкой (или прочитать сразу все строки в буфер) и выполнять их с помощью `PyRun_SimpleString()`. Имеется и другая возможность: для открытия файла вместо средств C/C++, т. е. вместо `fopen()`, вызвать функцию Python/C API

```
PyObject* PyFile_FromString( char *filename, char *mode )
```

где `mode` задает режим открытия файла и имеет те же возможные значения, что и функция `fopen()`.

При удачном выполнении `PyFile_FromString()` возвращается объект типа `PyObject`, который всегда можно преобразовать в обычный файловый дескриптор с помощью функции Python/C API

```
FILE* PyFile_AsFile( PyFileObject *p )
```

Пример приведен в листинге 25.6.

Листинг 25.6. Выполнение скрипта Python из C-программы (файл chap25/01/01.c)

```
// Пример встраивания Python в программу на C/C++.
// Выполнение скрипта Python, загруженного из файла.

#include <stdio.h>
#include "Python.h"

int main(int argc, char *argv[]){
    char *filename;
    PyObject* f;

    Py_Initialize();

    filename = "mytest.py";
    f = PyFile_FromString(filename, "r");

    if (f == NULL) {
        printf("\nCannot open file '%s'", filename);
        exit(1);
    }

    PyRun_SimpleFile(PyFile_AsFile(f), filename);
    Py_DECREF(f);

    // Ждать нажатия <Enter>:
    printf("\nPress <Enter>...");
    fgetc(stdin);

    Py_Finalize();
    return 0;
}
```

Обратите внимание, что переменная `f` здесь указывает на объект Python, созданный внешней по отношению к нему программой, поэтому следить за использованием этого объекта программа на C/C++ должна сама. В частности, когда значение `f` уже не требуется, нужно уменьшить на единицу значение счетчика ссылок на этот объект. Мы сделали это с помощью функции `Py_DECREF()`.

Файл mytest.py может быть любым, например, можно средствами стандартных библиотек Python сгенерировать почтовое сообщение в формате HTML, вложить в него графическое изображение и отправить через сервер SMTP (листинг 25.7).

Листинг 25.7. Отправка письма с рисунком через сервер SMTP

```
#!/usr/bin/env python
# -*- coding: cp1251 -*-
# Отправка e-mail в формате HTML с рисунком

from email.MIMEText import MIMEText
from email.MIMEImage import MIMEImage
from email.mime.multipart import MIMEMultipart

strFrom = 'имя@mail.ru'
strTo = 'кому@yandex.ru'

msgRoot = MIMEMultipart()
msgRoot['Subject'] = 'Тестовое сообщение'
msgRoot['From'] = strFrom
msgRoot['To'] = strTo

msgText = MIMEText('<b>Это <i>тестовое</i> сообщение</b> и рисунок:'
         '<br><img src="cid:image1">', 'html')
msgRoot.attach(msgText)

fp = open('test.png', 'rb')
msgImage = MIMEImage(fp.read())
fp.close()

msgImage.add_header('Content-ID', '<image1>')
msgRoot.attach(msgImage)

import smtplib
smtp = smtplib.SMTP()
```

```
smtp.connect('smtp.mail.ru')  # Сервер SMTP
smtp.login('имя', 'пароль')   # Логин и пароль
smtp.sendmail(strFrom, strTo, msgRoot.as_string())
smtp.quit()
```

Не правда ли, данная программа получилась намного короче той, что приведена в *разд. 20.6*?

ЗАДАЧА. Вызвать из программы на C/C++ функцию, определенную в модуле Python.

Решение задачи приведено в листинге 25.8.

Листинг 25.8. Файл chap25/02/call.c

```
// Импорт модуля, указанного первым параметром командной строки,
// и выполнение функции, заданной вторым параметром.
// Далее могут быть указаны значения аргументов функции.

#include <stdio.h>
#include <Python.h>

int main(int argc, char *argv[]) {
    PyObject *pName, *pModule, *pFunc;
    PyObject *pArgs, *pValue;
    int i;
    if (argc < 3) {
        fprintf(stderr,"Usage: call pythonfile funcname [args]\n");
        return 1;
    }

    Py_Initialize();

    pName = PyString_FromString(argv[1]);
    pModule = PyImport_Import(pName);
    Py_DECREF(pName);
    if (pModule != NULL) {
        pFunc = PyObject_GetAttrString(pModule, argv[2]);
        if (pFunc && PyCallable_Check(pFunc)) {
            pArgs = PyTuple_New(argc - 3);
            for (i = 0; i < argc - 3; ++i) {
                pValue = PyInt_FromLong(atoi(argv[i + 3]));
                if (!pValue) {
                    Py_DECREF(pArgs);
```

```
                    Py_DECREF(pModule);
                    fprintf(stderr, "Cannot convert argument\n");
                    return 1;
                }
                PyTuple_SetItem(pArgs, i, pValue);
            }
            pValue = PyObject_CallObject(pFunc, pArgs);
            Py_DECREF(pArgs);
            if (pValue != NULL) {
                printf("Result: %ld\n", PyInt_AsLong(pValue));
                Py_DECREF(pValue);
            }else{
                Py_DECREF(pFunc);
                Py_DECREF(pModule);
                PyErr_Print();
                fprintf(stderr,"Call failed\n");
                return 1;
            }
        }else{
            if (PyErr_Occurred())
                PyErr_Print();
            fprintf(stderr, "Cannot find function \"%s\"\n", argv[2]);
        }
        Py_XDECREF(pFunc);
        Py_DECREF(pModule);
    }else{
        PyErr_Print();
        fprintf(stderr, "Failed to load \"%s\"\n", argv[1]);
        return 1;
    }
    Py_Finalize();
    return 0;
}
```

Мы предполагаем, что в командной строке при запуске программы указывается:

- ❒ имя модуля Python (без расширения);
- ❒ имя функции, которую предстоит вызвать;
- ❒ значения параметров функции (если они требуются).

Сначала мы проверяем число параметров командной строки и завершаем программу, если не задано хотя бы два параметра (имя модуля и название

функции). Затем инициализируем библиотеку Python; преобразуем строку C (имя модуля, т. е. первый параметр командной строки) в объект Python типа `PyObject`, а затем выполняем импорт данного модуля. После этого объект Python, в котором хранится имя модуля, можно удалить, для чего достаточно уменьшить на единицу его счетчик ссылок.

Если импорт модуля выполнен удачно, требуется получить доступ к определенной в нем функции. При этом для надежности можно проверить, является ли объект с указанным именем выполняемой функцией. Затем необходимо передать параметры функции, заданные в командной строке (если они есть). Но C-программе они известны в виде строк, поэтому требуется сначала преобразовать их к нужному типу (для простоты мы полагаем, что все параметры представляют целые значения).

Наконец, все готово для вызова функции, после чего можно вывести на экран результат своего выполнения или сообщение об ошибке.

Уменьшив на единицу счетчики ссылок для всех использованных объектов Python, мы выгружаем библиотеку Python из памяти и завершаем работу C-программы.

Пример модуля, из которого будет импортироваться функция, приведен в листинге 25.9. Здесь мы просто выводим на консольный экран версию интерпретатора Python.

Листинг 25.9. Файл chap25/02/test1.py

```
import sys

def multiply(a, b):
    print "Python ", sys.version
    c = a * b
    print a, "*", b, "=", c
    return c
```

Компилируется C-программа, как и раньше. Вызвать получившийся исполняемый файл можно из следующего командного файла:

```
call.exe test1 multiply 3 4
pause
```

ЗАДАЧА. Вычислить выражение, заданное в виде строки программы Python, в котором используется функция, заданная во внешнем файле Python.

Решение иллюстрируют листинги 25.10 и 25.11.

Листинг 25.10. Файл chap25/03/calc.cpp

```
// Импорт модуля и вычисление выражения:
// Использование:  calc ИмяМодуляPython Выражение

#include <Python.h>

void process_expression( char* filename,
                         int num,
                         char** exp) {
    PyObject* f;


    PyRun_SimpleString("x = 0");

    f = PyFile_FromString(filename, "r");

    if (f == NULL) {
        printf("\nCannot open file '%s'", filename);
        exit(1);
    }

    PyRun_SimpleFile(PyFile_AsFile(f), filename);
    Py_DECREF(f);

    while(num--) {
        PyRun_SimpleString(*exp++);
        PyRun_SimpleString("print x");
    }
}

int main(int argc, char** argv) {
    char* filename;
    char* exp;
    PyObject* f;
    int n;

    if(argc != 3) {
        printf("Usage: calc filename expression\n");
        return 1;
    }

    filename = argv[1];
    n = argc - 1;
    exp = *(argv + 2);
```

```
    Py_Initialize();
    PyRun_SimpleString("x = 0");
    f = PyFile_FromString(filename, "r");
    if (f == NULL) {
        printf("\nCannot open file '%s'", filename);
        exit(1);
    }
    PyRun_SimpleFile(PyFile_AsFile(f), filename);
    Py_DECREF(f);

    PyRun_SimpleString(exp);
    PyRun_SimpleString("print x");

    Py_Finalize();
    return 0;
}
```

Листинг 25.11. Файл chap25/03/test.py

```
def MyAdd(a, b):
    return a + b
```

Вызывать исполняемый файл нужно следующим образом:

```
calc.exe test.py "x = MyAdd(5, 3)"
```

25.4. Библиотека PythonQt

Для более удобной работы с объектами Qt из скриптов Python можно использовать библиотеку PythonQt (**http://pythonqt.sourceforge.net**).

ЗАДАЧА. С помощью библиотеки PythonQt выполнить скрипт Python из приложения Qt.

Решение иллюстрирует листинг 25.12.

Листинг 25.12

```
#include "PythonQt.h" // Подключили заголовочный файл библиотеки
#include <QApplication>

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

```
    // Инициализируем библиотеку PythonQt
    PythonQt::init();

    // Получаем указатель на модуль Python __main__
    PythonQtObjectPtr mainModule =
                      PythonQt::self()->getMainModule();

    // Выполняем скрипт и получаем результат
    QVariant result = mainModule.evalScript(
                mainModule, "19*2+4", Py_eval_input);

    // Создаем для примера виджеты Qt
    QVBoxLayout*  vbox = new QVBoxLayout;
    QGroupBox*    box  = new QGroupBox;
    QTextBrowser* browser = new QTextBrowser(box);
    QLineEdit*    edit = new QLineEdit(box);
    QPushButton*  button = new QPushButton(box);
    button->setObjectName("button1");
    edit->setObjectName("edit");
    browser->setObjectName("browser");
    vbox->addWidget(browser);
    vbox->addWidget(edit);
    vbox->addWidget(button);
    box->setLayout(vbox);

    // Сообщаем Python о виджете box
    mainModule.addObject("box", box);

    // Выполняем скрипт, хранящийся как ресурс
    mainModule.evalFile(":/myscript.py");

    // Определяем новую функцию Python appendText()
    // (не забываем про отступы!)
    mainModule.evalScript("
        def appendText(text):\n    box.browser.append(text)");

    // Вызываем эту функцию
    mainModule.call("
          appendText",
          QVariantList() << "The ultimate answer is " );

    return qapp.exec();
}
```

Скрипт Python может свободно работать с объектами библиотеки Qt, для этого достаточно импортировать модуль `PythonQt`, например:

```
from PythonQt import *
print Qt.AlignLeft
print QDate.currentDate()
a = QSize(1,2)
```

25.5. Библиотека PyQt4

Если мы хотим "отдохнуть" от C++ и реализовать свой проект полностью на языке Python, то все возможности библиотеки Qt все равно остаются в нашем распоряжении, поскольку существует реализация Qt на языке Python, которая называется PyQt (**www.riverbankcomputing.com/software/pyqt**). Объекты библиотеки PyQt имеют все те же методы и свойства, как и в Qt, только работать с ними нужно не на C++, а на Python.

ЗАДАЧА. Создать окно с кнопкой, при нажатии которой происходит выход из приложения.

Решение иллюстрирует листинг 25.13.

Листинг 25.13. Файл chap25/pyqt/quitbutton.py

```
#!/usr/bin/env python
# -*- coding: cp1251 -*-

# Сигналы и слоты: кнопка в окне
import sys
from PyQt4 import QtCore, QtGui

app = QtGui.QApplication(sys.argv)
btn = QtGui.QPushButton(u"&Выход")
btn.setFont(QtGui.QFont("Times", 16, QtGui.QFont.Bold))
btn.resize(120, 35)
QtCore.QObject.connect(
   btn,                        # Источник сигнала.
   QtCore.SIGNAL("clicked()"), # Сигнал о нажатии кнопки.
   app,                        # Приемник сигнала.
   QtCore.SLOT("quit()") );    # Функция-слот (обработчик события).
btn.show()
sys.exit(app.exec_())
```

ЗАДАЧА. Создать главное окно приложения PyQt4 с панелью меню.

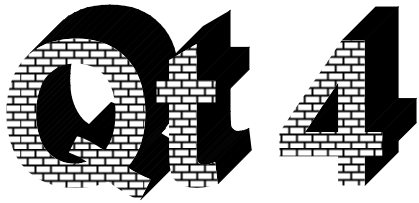
Решение иллюстрирует листинг 25.14.

Листинг 25.14. Файл chap25/pyqt/mainwin.py

```
#!/usr/bin/env python
# -*- coding: cp1251 -*-

# Главное окно с меню

import sys
from PyQt4.QtCore import SIGNAL, SLOT, QTextCodec
from PyQt4.QtGui import QApplication, QMainWindow, QMenu, QMessageBox, \
     QWidget, qApp

class MainWindow(QMainWindow):

    def __init__(self):
        QMainWindow.__init__(self)

        fileMenu = self.menuBar().addMenu(self.tr("&Файл"))
        exitAction = fileMenu.addAction(self.tr("В&ыход"))

        helpMenu = self.menuBar().addMenu(self.tr("&Справка"))
        aboutAction = helpMenu.addAction(self.tr("&О программе"))

        self.connect(exitAction, SIGNAL("triggered()"),
                     qApp, SLOT("quit()"))
        self.connect(aboutAction, SIGNAL("triggered()"),
                     self.showAboutBox)

    def showAboutBox(self):
        QMessageBox.information(
            self, self.tr("О программе"),
            self.tr("Пример приложения PyQt4\n"
                    "с простым меню" ) )

if __name__ == "__main__":
    app = QApplication(sys.argv)
    codec = QTextCodec.codecForName('CP1251')
    QTextCodec.setCodecForTr(codec)

    window = MainWindow()
    window.show()
    sys.exit(app.exec_())
```

Текст данной программы, если отвлечься от особенностей языка Python, практически полностью совпадает со своим аналогом, написанным на C++.

ГЛАВА 26



Создание библиотек и плагинов для Qt Designer

Эпиграфы — костыли неумелого автора. Или чересчур уж закомплексованного. Пробуйте ходить своими ногами.

Юрий Никитин "Как стать писателем"

Рано или поздно программист доходит до осознания того, что ему уже не хватает привычных средств и пора подумать о разработке собственных проблемно-ориентированных библиотек и нестандартных виджетов. Этим мы и займемся в данной главе.

26.1. Создание библиотеки

ЗАДАЧА. Разработать библиотеку, экспортирующую функцию, которая возвращает все заглавные буквы, содержащиеся в переданной строке `QString`.

Решение. Общий алгоритм создания библиотек:

- в файле описания проекта *.pro вместо обычного `app` задать шаблон

  ```
  TEMPLATE = lib
  ```

- там же в секции `CONFIG` добавить описатель типа библиотеки: `dll`, `staticlib` или `plugin`;
- в заголовочном файле описать экспортируемую функцию как `extern "C"`;
- в системе Windows, кроме того, необходимо указать директиву `__declspec(dllexport)`.

Пример иллюстрируют листинги 26.1–26.3.

Листинг 26.1. Файл описания проекта chap26/00/dynlib.pro

```
TEMPLATE = lib
CONFIG += staticlib      #  или  dll
QT -= gui
SOURCES = dynlib.cpp
HEADERS = dynlib.h
win32: TARGET = ../dynlib
```

Листинг 26.2. Заголовочный файл chap26/dynlib/dynlib.h

```
#include <QString>

#ifdef Q_WS_WIN
#define MY_EXPORT __declspec(dllexport)
#else
#define MY_EXPORT
#endif

extern "C" MY_EXPORT QString getCaps(const QString& s);
```

Листинг 26.3. Файл chap26/ dynlib /dynlib.cpp

```
#include "dynlib.h"

QString getCaps(const QString& s) {
    QString r;
    foreach (QChar c, s) {
        if ( c.isUpper() )
            r += c;
    }
    return r;
}
```

Главная программа для компиляции библиотеки, разумеется, не нужна. Для сборки проекта, как обычно, следует сначала выполнить команду `qmake`, а потом вызвать `nmake` или `make`. При изменении значения параметра `CONFIG` не забывайте обновлять `makefile`, снова повторяя команду `qmake`.

Получившийся в результате компиляции файл динамической библиотеки поместите в каталог, где программа сможет его найти: в каталог самой программы или в Windows\System32 (в системе Windows) или в /usr/lib (в системе Linux).

26.2. Статическая компоновка

ЗАДАЧА. Использовать библиотеку, разработанную в предыдущем разделе, в приложении Qt.

Решение. Имеются две возможности: либо подключить библиотеку к своему приложению статически, на этапе его компоновки, либо загрузить библиотеку во время работы приложения. Здесь рассмотрим первый метод.

Общий алгоритм:

- в файле описания проекта *.pro в разделе `LIBS` указать файл библиотеки (с префиксом `-l`, без расширения):

  ```
  LIBS += -ldynlib
  ```

- там же можно указать путь к каталогу, где эта библиотека находится, только с префиксом `-L`;
- в разделе `INCLUDEPATH` задать каталог, в котором располагаются необходимые заголовочные файлы библиотеки (по отношению к каталогу проекта):

  ```
  INCLUDEPATH += ../dynlib
  ```

- в тексте программы подключить заголовочный файл библиотеки.

Решение нашей задачи показано в листингах 26.4–26.7, а внешний вид окна программы приведен на рис. 26.1.

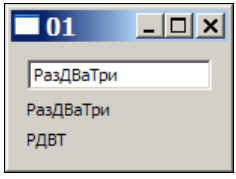


Рис. 26.1. Нижняя надпись формируется с помощью функции, загруженной из библиотеки

Листинг 26.4. Файл chap26/app1/app.pro

```
TEMPLATE = app
win32: TARGET = ../app
DEPENDPATH += . \
              ../dynlib
INCLUDEPATH += . \
               ../dynlib
LIBS += -ldynlib
LIBS += -L../dynlib
```

```
HEADERS += win.h
SOURCES += main.cpp \
           win.cpp
```

Листинг 26.5. Файл chap26/app1/win.h

```
#include <QtGui>

class Win : public QWidget {
    Q_OBJECT
public:
    Win();
public slots:
    void mySlot(const QString& text);
private:
    QLabel *m_label1;
    QLabel *m_label2;
    QLineEdit *m_edit;
};
```

Листинг 26.6. Файл chap26/app1/win.cpp

```
#include "win.h"
#include "dynlib.h"

Win::Win() : QWidget(0) {
    QVBoxLayout *lay = new QVBoxLayout(this);
    m_edit = new QLineEdit;
    m_label1 = new QLabel;
    m_label2 = new QLabel;
    lay->addWidget(m_edit);
    lay->addWidget(m_label1);
    lay->addWidget(m_label2);
    setLayout(lay);

    connect( m_edit,
             SIGNAL(textChanged(const QString&)),
             this,
             SLOT(mySlot(const QString&)) );
}

void Win::mySlot(const QString& text) {
    m_label1->setText(text);
```

```
    QString s = getCaps(text);
    m_label2->setText(s);
}
```

Листинг 26.7. Файл chap26/app1/main.cpp

```
#include "win.h"

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

    Win win;
    win.show();

    return app.exec();
}
```

При изменении текста в строке редактирования вызывается слот `mySlot()`, в котором первой надписи `QLabel` передается текст, содержащийся в этой строке, а второй надписи — текст, формируемый внешней функцией `getCaps()`.

26.3. Динамическая загрузка

Для загрузки динамической библиотеки во время работы приложения используется класс `QLibrary`. Пример иллюстрируют листинги 26.8–26.10.

Листинг 26.8. Файл chap26/app2/app.pro

```
TEMPLATE = app
win32: TARGET = ../app
DEPENDPATH += .
HEADERS += win.h
SOURCES += main.cpp \
           win.cpp
```

Листинг 26.9. Файл chap26/app2/win.h

```
#include <QtGui>

typedef QString (*Fun)(const QString&);

class Win : public QWidget {
    Q_OBJECT
```

```
public:
    Win();
public slots:
    void mySlot(const QString& text);
private:
    QLabel *m_label1;
    QLabel *m_label2;
    QLineEdit *m_edit;
    Fun m_fun;
};
```

Листинг 26.10. Файл chap26/app2/win.cpp

```
#include "win.h"

Win::Win() : QWidget(0) {
    QVBoxLayout *lay = new QVBoxLayout(this);
    m_edit = new QLineEdit;
    m_label1 = new QLabel;
    m_label2 = new QLabel;
    lay->addWidget(m_edit);
    lay->addWidget(m_label1);
    lay->addWidget(m_label2);
    setLayout(lay);

    // Создаем объект "внешняя библиотека":
    QLibrary *dll = new QLibrary(this);
    dll->setFileName("dynlib"); // Имя файла без расширения

    if (dll->load()){ // Загружаем и проверяем
        m_fun = (Fun)(dll->resolve("getCaps")); // Функция getCaps
        if (!m_fun)  // Функция с таким именем не найдена
            m_label2->setText("Cannot find 'getCaps'");
    }else{
        m_fun = 0; // Библиотека не найдена
        m_label2->setText("Cannot load DLL");
    }
    connect( m_edit,
             SIGNAL(textChanged(const QString&)),
             this,
             SLOT(mySlot(const QString&)) );
}
```

```
void Win::mySlot(const QString& text) {
    m_label1->setText(text);
    QString s("---");
    if (m_fun)
        s = m_fun(text);

    m_label2->setText(s);
}
```

Предполагается, что файл загружаемой библиотеки находится или в текущем каталоге, или доступен по стандартным путям поиска (например, Windows\System32 в системе Windows или /usr/lib в Linux).

Тест главной программы — как в листинге 26.7.

Обратите внимание: мы загружаем внешнюю библиотеку и ищем в ней нужную функцию в конструкторе окна, а не в самом слоте. Конечно, если мы заранее не уверены, что данный слот точно сработает, то можем загружать библиотеку позже, в тот момент, когда внешняя функция точно потребуется, но в любом случае — не каждый раз при срабатывании слота.

При динамической загрузке обязательно нужно проверять, найдена ли требуемая функция из внешней библиотеки, иначе мы рискуем вызвать аварийное завершение программы. Также следует внимательно относиться к приведению типов: поскольку компилятор не видит заголовочного файла внешней библиотеки, то за правильным описанием типа `typedef` должен следить сам программист.

26.4. Расширение библиотеки виджетов Qt Designer

Чтобы разработать новый виджет и сделать его доступным для использования в Qt Designer, необходимо создать свой класс на основе класса `QDesignerCustomWidgetInterface`. При этом придется реализовать следующие функции:

- `name()` — имя класса;
- `group()` — название группы виджетов Qt Designer (например, `Input Widgets`);
- `toolTip()` — краткое описание виджета;
- `whatsThis()` — более подробное описание виджета;
- `includeFile()` — имя заголовочного файла, который должен включаться в те приложения, которые задействуют данный виджет. Эта информация хранится в файле *.ui и необходима компилятору `uic`;

- `icon()` — графическое изображение виджета на панели Qt Designer;
- `isContainer()` — эта функция должна возвращать значение `true`, если виджет является контейнером для хранения других виджетов;
- `createWidget()` — функция создает новый экземпляр виджета и возвращает указатель на него;
- `domXml()` — функция должна возвращать фрагмент XML-описания свойств виджета (с их значениями по умолчанию), который запишется потом в файл *.ui. Если разрабатываемый элемент не предполагается показывать на панели виджетов, а он нужен лишь для работы других объектов, то данная функция должна возвращать пустую строку;
- `codeTemplate()` — эта функция зарезервирована на будущее, пока ее можно не использовать.

Следующие две функции переопределяют при необходимости:

- `initialize()` — для многостраничных виджетов или если требуется добавить новые пункты в контекстное меню виджета;
- `isInitialized()` — эта функция возвращает значение `true`, если виджет был инициализирован. Раз мы переопределили функцию инициализации, то, возможно, потребуется переопределить и функцию, которая проверяет результат ее выполнения.

Кроме того, при описании класса плагина и его реализации требуется выполнить следующие условия:

- при объявлении класса плагина после макроса `Q_OBJECT` указать `Q_INTERFACES()`;
- в объявлении каждого пользовательского класса-виджета (перед его именем), экземпляр которого будет создаваться нашим плагином, должен быть упомянут макрос `QDESIGNER_WIDGET_EXPORT`;
- в разделе реализации класса плагина указать макрос `Q_EXPORT_PLUGIN2()`. Первым его параметром должно быть название плагина в том виде, как оно написано в файле *.pro, а в качестве второго параметра задается имя класса плагина (вариант этого макроса без цифры `2` предназначен для плагинов, разрабатываемых в виде статических библиотек);
- в объявлении классов-виджетов после макроса `Q_OBJECT` перечислить все свойства разрабатываемого виджета, используя макрос `Q_PROPERTY()`;
- для чтения и записи свойств реализовать `get`- и `set`-методы;
- объявить `set`-методы как слоты;
- всю дополнительную информацию об авторе, лицензии, номере версии разрабатываемого плагина задать с помощью макроса `Q_CLASSINFO`.

Пример оформления файла описания проекта приведен в листинге 26.11.

Листинг 26.11

```
CONFIG      += designer plugin debug_and_release  # или только release
TARGET      = $$qtLibraryTarget($$TARGET)
TEMPLATE    = lib
QTDIR_build:DESTDIR = $$QT_BUILD_TREE/plugins/designer
HEADERS     = . . . # здесь, как обычно, указываются файлы .h
SOURCES     = . . . # файлы .cpp
RESOURCES   = имяфайла.qrc # файл описания ресурсов
target.path   = $$[QT_INSTALL_PLUGINS]/designer
sources.files = $$SOURCES $$HEADERS *.pro
sources.path  = $$[QT_INSTALL_EXAMPLES]/designer/имяплагина
INSTALLS += target sources
```

ЗАДАЧА. Разработать собственный виджет в виде светодиодного индикатора, имеющего два состояния (включен/выключен), и сделать его доступным для использования в Qt Designer.

Решение. В листингах 26.12 и 26.13 показано, как можно оформить класс-виджет. Мы определили три свойства: состояние индикатора `value`, цвет в выключенном `color0` и во включенном `color1` состояниях.

Листинг 26.12. Файл chap26/led/led.h

```
#ifndef _LED_H_
#define _LED_H_

#include <QtGui>
#include <QWidget>
#include <QtDesigner/QDesignerExportWidget>

class QDESIGNER_WIDGET_EXPORT Led : public QWidget {
    Q_OBJECT
    Q_PROPERTY(bool value READ value WRITE setValue);
    Q_PROPERTY(QColor color0 READ color0 WRITE setColor0);
    Q_PROPERTY(QColor color1 READ color1 WRITE setColor1);
public:
    Led(QWidget *parent = 0);
    bool value() const { return m_value; }
    QColor color0() const { return m_color0; }
    QColor color1() const { return m_color1; }
```

```
public slots:
    void setValue(bool);
    void setColor0(QColor);
    void setColor1(QColor);
    void toggle();
protected:
    bool m_value;
    QColor m_color0;
    QColor m_color1;
    void paintEvent(QPaintEvent *event);
};
#endif // _LED_H_
```

Листинг 26.13. Файл chap26/led/led.cpp

```
#include <QtGui>
#include "led.h"

Led::Led(QWidget *parent)
    : QWidget(parent) {
    m_value=false;
    m_color1 = Qt::red;
    m_color0 = Qt::lightGray;
    setMinimumSize(QSize(30, 30));
}

void Led::paintEvent(QPaintEvent *) {
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    painter.setWindow(-30, -30, 60, 60);

    painter.setPen(Qt::darkGray);
    painter.drawEllipse(-28, -28, 56, 56);

    QColor bc;
    if(m_value)
       bc = m_color1;
    else
       bc = m_color0;

    QBrush brush(bc);
    painter.setBrush(brush);
    painter.drawEllipse(-25, -25, 50, 50);
}
```

```
void Led::setColor0(QColor color) {
   m_color0 = color;
   update();
}

void Led::setColor1(QColor color) {
   m_color1 = color;
   update();
}

void Led::setValue(bool value) {
   m_value = value;
   update();
}

void Led::toggle() {
    m_value = !m_value;
    update();
}
```

В листингах 26.14 и 26.15 показан класс плагина.

Листинг 26.14. Файл chap26/led/ledplugin.h

```
#ifndef _LEDPLUGIN_H_
#define _LEDPLUGIN_H_

#include <QDesignerCustomWidgetInterface>

class LedPlugin : public QObject,
                  public QDesignerCustomWidgetInterface {
    Q_OBJECT
    Q_INTERFACES(QDesignerCustomWidgetInterface)
public:
    LedPlugin(QObject *parent = 0);
    bool isContainer() const;
    bool isInitialized() const;
    QIcon icon() const;
    QString domXml() const;
    QString group() const;
    QString includeFile() const;
    QString name() const;
    QString toolTip() const;
    QString whatsThis() const;
```

```
    QWidget *createWidget(QWidget *parent);
    void initialize(QDesignerFormEditorInterface *core);

private:
    bool initialized;
};
#endif // _LEDPLUGIN_H_
```

Листинг 26.15. Файл chap26/led/ledplugin.cpp

```
#include "led.h"
#include "ledplugin.h"

#include <QtPlugin>

LedPlugin::LedPlugin(QObject *parent)
          : QObject(parent) {
    initialized = false;
}

void LedPlugin::initialize(
             QDesignerFormEditorInterface * /* core */){
    if (initialized)
        return;

    initialized = true;
}

bool LedPlugin::isInitialized() const {
    return initialized;
}

QWidget *LedPlugin::createWidget(QWidget *parent){
    return new Led(parent);
}

QString LedPlugin::name() const {
    return "Led";
}

QString LedPlugin::group() const
{
    return "My Widgets";
}
```

```
QIcon LedPlugin::icon() const {
    return QIcon(":/led.png");
}

QString LedPlugin::toolTip() const {
    return "Led indicator";
}

QString LedPlugin::whatsThis() const {
    return "Binary led indicator";
}

bool LedPlugin::isContainer() const {
    return false;
}

QString LedPlugin::domXml() const {
    return "<widget class=\"Led\" name=\"led\">\n"
           " <property name=\"geometry\">\n"
           "  <rect>\n"
           "   <x>0</x>\n"
           "   <y>0</y>\n"
           "   <width>30</width>\n"
           "   <height>30</height>\n"
           "  </rect>\n"
           " </property>\n"
           " <property name=\"toolTip\" >\n"
           "  <string>Led indicator</string>\n"
           " </property>\n"
           " <property name=\"value\" >\n"
           " <bool>false</bool>\n"
           " </property>\n"
           " <property name=\"whatsThis\" >\n"
           "  <string>Led widget</string>\n"
           " </property>\n"
           " <property name=\"color1\" >\n"
           " <enum>Qt::red</enum>\n"
           " </property>\n"
           " <property name=\"color0\" >\n"
           " <enum>Qt::lightGray</enum>\n"
           " </property>\n"
           "</widget>\n";
}
```

```
QString LedPlugin::includeFile() const {
    return "led.h";
}

Q_EXPORT_PLUGIN2(ledplugin, LedPlugin)
```

Конечно, для большей выразительности однотонную заливку можно заменить на градиентную.

Наконец, осталось только подготовить файл с изображением нового виджета, которое будет показано на панели Qt Designer, добавить этот файл в виде ресурса (листинг 26.16) и создать файл описания проекта (листинг 26.17).

Листинг 26.16. Файл chap26/led/led.qrc

```
<!DOCTYPE RCC><RCC version="1.0">
    <qresource>
        <file>led.png</file>
    </qresource>
 </RCC>
```

Листинг 26.17. Файл chap26/led/ledplugin.pro

```
CONFIG      += designer plugin debug_and_release
TARGET      = $$qtLibraryTarget($$TARGET)
TEMPLATE    = lib
QTDIR_build:DESTDIR = $$QT_BUILD_TREE/plugins/designer
HEADERS     = led.h \
              ledplugin.h
SOURCES     = led.cpp \
              ledplugin.cpp
RESOURCES = led.qrc

# install
target.path = $$[QT_INSTALL_PLUGINS]/designer
sources.files = $$SOURCES $$HEADERS *.pro
sources.path = $$[QT_INSTALL_EXAMPLES]/designer/ledplugin
INSTALLS += target sources
```

Для компиляции нужно последовательно выполнить команды `qmake`, `nmake` и `nmake install`. Разумеется, при использовании MinGW или GCC команду `nmake` следует заменить на `mingw32-make` или `make`. Файл с плагином будет скопирован в каталог qtXXX/plugins/designer.

Теперь можно открыть Qt Designer и найти новую панель **My Widgets** (именно такое название возвращает наш метод `LedPlugin::group()`) с единственным виджетом на ней (рис. 26.2). Если что-то не так, то информацию об ошибке во время загрузки плагина можно прочитать, выполнив команду меню **Help** (Справка) | **About Plugins** (О плагинах).

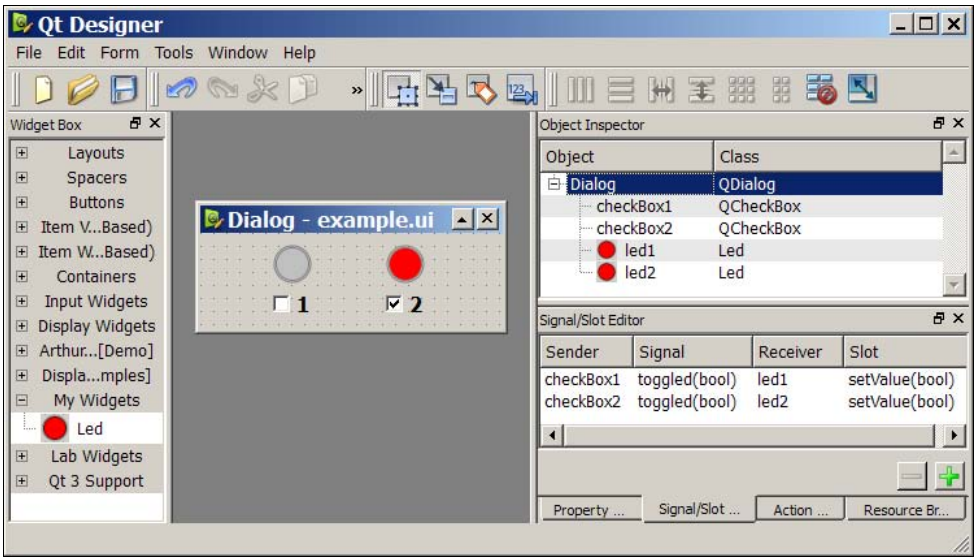
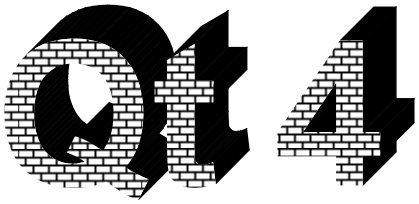


Рис. 26.2. В Qt Designer появился новый виджет

Для примера мы создали форму с двумя индикаторами `Led` (их имена `led1` и `led2`) и элементами `QCheckBox` (с именами `checkBox1` и `checkBox2`), а затем связали сигналы `toggled(bool)` элементов `QCheckBox` со слотами `setValue(bool)` наших индикаторов.

ГЛАВА 27



Сравнение библиотеки Qt с другими средствами межплатформенной разработки

Всякая вещь есть форма проявления беспредельного разнообразия.

Козьма Прутков

Если читатель держит в руках эту книгу и, тем более, дошел до данной главы, то он, вероятно, знает кое-что о платформах Java и .NET, а также, возможно, слышал о существовании библиотеки wxWidgets и других.

27.1. Qt и Java

Нетрудно увидеть то общее, что объединяет библиотеку Qt и Java. Они задуманы как средство создания межплатформенных приложений, но справляются со своей задачей по-разному. Qt — это библиотека C++, со всеми преимуществами и недостатками данного языка. После компиляции приложения Qt мало чем отличаются от всех остальных программ, написанных на C++. И сама библиотека Qt, и использующие ее программы компилируются в обычный двоичный код, являющийся "родным" для той платформы, на которой выполняется компиляция. Все дополнения, которые привносит библиотека Qt в стандартный язык C++, касаются только препроцессора и поэтому никак не отражаются на производительности приложений. Исключение, правда, составляет механизм сигналов и слотов: он примерно раз в 10 "медлительнее", чем обычный вызов невиртуального метода в C++, но поскольку даже эта, казалось бы, значительная прибавка, не так уж велика по сравнению с тем временем, которое требуется, чтобы выполнить операции `new` или `delete` (ведь в большинстве практических случаев приходится иметь дело не

с одиночными числами, а с массивами и строками), то общая задержка реагирования на события у Qt-приложений оказывается практически такой же, что и у обычных программ на C++.

Java — это не только другой язык программирования, свой набор системных и прикладных библиотек, но также новое звено, прослойка между программой и аппаратным обеспечением компьютера, — виртуальная машина, которая обеспечивает выполнение программ. Как правило, приложения Java более "прожорливы" в отношении ресурсов компьютера и выполняются несколько медленнее, чем приложения Qt. GUI-библиотека Swing, разработанная для Java, по своей идеологии сходна с Qt: обе они используют платформозависимое API лишь для выполнения базовых операций, а всю работу по отрисовке и управлению виджетами берут на себя. В результате приложения Swing и Qt выглядят на различных платформах одинаково, но последние заметно выигрывают по своей производительности. Но реализации Java существуют для большего количества платформ, хотя многие считают, что к списку Windows, Linux (в том числе Embedded) и Mac OS вряд ли можно добавить что-либо существенное.

Приложения Java выполняются на различных платформах без перекомпиляции (это обеспечивается идентичностью API, предоставляемым разными реализациями виртуальной машины Java), в то время как приложения Qt требуют перекомпиляции при переносе на другую платформу. Наверно, это можно назвать недостатком, но вряд ли он может повлиять на конечный выбор.

Перевес в сторону Qt стал еще очевиднее после появления Qt Jambi — Java-версии библиотеки Qt (**http://trolltech.com/products/qt/features/language-support/java**).

27.2. Qt и .NET

Технология .NET основывается на том, что программы компилируются в некоторый промежуточный универсальный язык (байт-код), выполняемый виртуальной машиной CLR (*Common Language Runtime*), которая при каждом запуске программы переводит промежуточный код в обычный двоичный. Разумеется, это приводит к некоторому снижению производительности, как и в случае Java.

Безусловно, платформа .NET предоставляет больше возможностей для создания приложений, чем Qt. Но пока она жестко связана с миром Microsoft Windows и только претендует на то, чтобы когда-нибудь, возможно, стать по-настоящему межплатформенной.

27.3. Qt и wxWidgets

Существует еще одна достаточно мощная межплатформенная библиотека для программирования на C++, а также на нескольких других языках (в том числе Java и Python) — wxWidgets (**www.wxwidgets.org**) — библиотека с открытым исходным кодом, работающая в системах X Window, Mac OS, Microsoft Windows и OS/2. В отличие от Qt, wxWidgets использует, где только возможно, нативные элементы интерфейса, т. е. является, по существу, дополнительной оберткой для API текущей платформы, предоставляя программисту унифицированный программный интерфейс. Как следствие, приложения wxWidgets, скомпилированные в Windows, выглядят как обычные для данной системы программы, написанные, например, с использованием MFC, но после компиляции в Linux становятся неотличимыми от других приложений Linux.

Не так давно в рамках проекта wxWidgets начал развиваться другой вариант реализации — wxUniversal, в котором реализован тот же подход, что и в Qt (самостоятельная отрисовка элементов интерфейса), но он еще далек от завершения.

ЗАДАЧА. Создать с помощью библиотеки wxWidgets пустое окно с заголовком.

Решение иллюстрирует листинг 27.1.

Листинг 27.1. Файл chap27/wx00/00.cpp

```
// Простейшее приложение wxWidgets
#include <wx/wx.h>

class MyApp: public wxApp {
   virtual bool OnInit();
};

DECLARE_APP(MyApp)
IMPLEMENT_APP(MyApp)

bool MyApp::OnInit() {
   wxFrame *frame = new wxFrame(
       NULL, wxID_ANY, _("Здравствуй, wxWidgets!"),
       wxPoint(50,50), wxSize(400,300) );
       frame->Show(TRUE);
       SetTopWindow(frame);
       return TRUE;
}
```

На рис. 27.1 показан внешний вид получившегося окна.

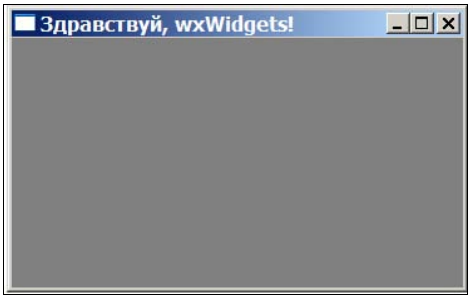


Рис. 27.1. Пустое окно с заголовком

ЗАДАЧА. Создать главное окно с панелью меню и строкой состояния, в которой отображаются размеры окна и текущие координаты указателя мыши (рис. 27.2).

Решение иллюстрируют листинги 27.2 и 27.3.

Листинг 27.2. Файл chap27/wx01/01.h

```
#include <wx/wx.h>

class MyFrame: public wxFrame {
public:
    MyFrame::MyFrame(wxWindow* parent, wxWindowID id,
                     const wxString& title,
                     const wxPoint& pos = wxDefaultPosition,
                     const wxSize& size = wxDefaultSize);
    void OnAbout(wxCommandEvent& event); // О программе.
    void OnExit(wxCommandEvent& event);  // Выход.
    void OnClose(wxCloseEvent& event);   // При закрытии.
    void OnSize(wxSizeEvent& event); // При изменении размеров.
    void OnMouseMotion(wxMouseEvent& event); // При перемещении
                                             // указателя мыши.
private:
    DECLARE_EVENT_TABLE()
};

class MyApp: public wxApp {
public:
    virtual bool OnInit();
};
DECLARE_APP(MyApp)
```

```
enum { // Идентификаторы элементов меню
    ID_MENU_ABOUT = wxID_HIGHEST + 1,
    ID_MENU_EXIT
};
```

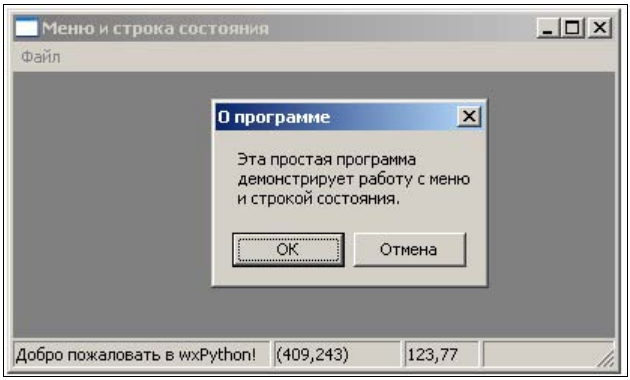


Рис. 27.2. Главное окно с панелью меню и строкой состояния, в которой отображаются размеры окна и текущие координаты указателя мыши

Листинг 27.3. Файл chap27/wx01/01.cpp

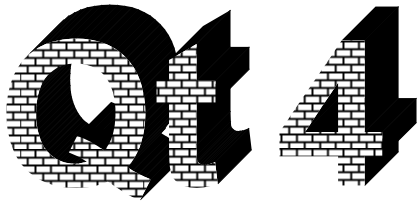
```
#include "01.h"

BEGIN_EVENT_TABLE(MyFrame, wxFrame) // Таблица обработчиков событий:
    EVT_MENU(ID_MENU_ABOUT, MyFrame::OnAbout) // О программе.
    EVT_MENU(ID_MENU_EXIT, MyFrame::OnExit)   // Выход.
    EVT_CLOSE(MyFrame::OnClose)  // При закрытии фрейма.
    EVT_SIZE(MyFrame::OnSize) // При изменении размеров фрейма.
    EVT_MOTION(MyFrame::OnMouseMotion) //При перемещении указателя мыши.
END_EVENT_TABLE()

MyFrame::MyFrame( wxWindow* parent, wxWindowID id,
                  const wxString& title, // Заголовок.
                  const wxPoint& pos,    // Позиция.
                  const wxSize& size)    // Размер.
        : wxFrame(parent, id, title, pos, size) {

    wxMenuBar *mb = new wxMenuBar(); // Панель меню.
    wxMenu *menu = new wxMenu();     // Группа пунктов.
    menu->Append( ID_MENU_ABOUT,     // Пункт меню.
                  wxT("&О программе"),
                  wxT("О программе"));
```

```
    menu->AppendSeparator();          // Разделитель.
    menu->Append( ID_MENU_EXIT,       // Пункт меню.
                  wxT("B&ыход"),
                  wxT("Выход из программы"));
    mb->Append(menu, wxT("&Файл"));
    SetMenuBar(mb);

    CreateStatusBar(4); // Строка состояния.
    int widths[] = { -2, -1, 50, 90 };
    SetStatusWidths(4, widths);
    SetStatusText(wxT("Добро пожаловать в wxWidgets!"), 0);
}

void MyFrame::OnAbout(wxCommandEvent& event) { // О программе.
    wxMessageBox( wxT("Эта простая программа\n")
                  wxT("демонстрирует работу с меню\n")
                  wxT("и строкой состояния."),
                  wxT("О программе"),
                  wxOK | wxICON_INFORMATION, this );
}

void MyFrame::OnExit(wxCommandEvent& event) {
    Close(); // Закрытие главного окна — выход из программы.
}

void MyFrame::OnClose(wxCloseEvent& event) { // Перед закрытием.
    wxMessageDialog *d = new wxMessageDialog( this,
                             wxT("Выйти из программы?"),
                             wxT("Подтвердите"),
                             wxOK | wxCANCEL );
    if (d->ShowModal() == wxID_OK) { // Ответ "Да".
        Destroy();    // Закрытие фрейма ( или Close() ).
    } else  {                        // Ответ "Нет".
        d->Destroy(); // Уничтожение диалога с вопросом.
    }
}

void MyFrame::OnSize(wxSizeEvent& event) { // При изменении размеров.
       wxSize sz = GetSize(); // Размер фрейма.
       if (GetStatusBar())
        SetStatusText( wxString::Format( wxT("(%d,%d)"),
                             sz.GetX(), sz.GetY() ),
                       1 );
}
```

```
void MyFrame::OnMouseMotion(wxMouseEvent& event) { //При движении мыши.
    if (GetStatusBar())
        SetStatusText( wxString::Format( wxT("%d,%d"),
                            event.GetX(), event.GetY() ),
                       2 );
}

//-----------------------------------------------
IMPLEMENT_APP(MyApp)

bool MyApp::OnInit() {
    MyFrame *frame = new MyFrame(
                NULL, -1, wxT("Меню и строка состояния"),
                wxDefaultPosition,  wxSize(500,300) );
    frame->CenterOnScreen(wxBOTH); // По центру экрана.
    frame->Show();
    SetTopWindow(frame);
    return TRUE;
}
```

Наверно, wxWidget покажется ближе тем, кто программировал в Windows, используя MFC: они увидят знакомые целочисленные идентификаторы и карты сообщений, но автор считает, что от этого как раз достаточно просто отвыкнуть. На самом деле, основное (и, пожалуй, единственное) преимущество wxWidgets по сравнению с Qt — это свободная лицензия, позволяющая бесплатно скачать библиотеку wxWidgets и разрабатывать не только приложения Open Source, но также и коммерческие программы с закрытым исходным кодом. Что касается недостатков wxWidgets, то следует упомянуть явную нехватку визуальных инструментов разработки графического интерфейса, а также то, что программы, написанные на основе wxWidgets, менее предсказуемы при переносе на другую платформу и поэтому требуют тщательного тестирования.

ГЛАВА 28



Разное

Наши взгляды как наши часы: все они показывают разное время, но каждый верит только своим.
Александр Поуп

В данной главе приведены примеры решения различных частных задач, не вошедшие в предыдущие разделы.

28.1. Борьба с "зависанием" интерфейса пользователя

При выполнении длительных операций нельзя забывать о пользователе, который "от нечего делать" обязательно захочет понажимать на какие-нибудь клавиши или кнопки и подвигать окошки. Поэтому в тело многократно выполняемого цикла желательно вставлять вызов

```
qApp->processEvents();
```

чтобы обработать все накопившиеся к данному моменту события. Разумеется, нет особого смысла выполнять эту команду при каждой итерации цикла, если таких итераций за секунду оказывается больше пяти. Как следует из исследований психологов, пользователь начинает замечать "заторможенность" интерфейса приложения, если время реакции становится больше 0,7 с. Но в любом случае следует предусмотреть некоторый "запас прочности".

Второй вариант — выносить трудоемкие вычисления или взаимодействие с удаленными компьютерами в отдельный поток.

28.2. Перемещение виджетов с помощью мыши

ЗАДАЧА. Реализовать возможность перемещения элементов управления с помощью мыши.

Решение. Текст программы приведен в листингах 28.1 и 28.2, а внешний вид окна приложения показан на рис. 28.1.

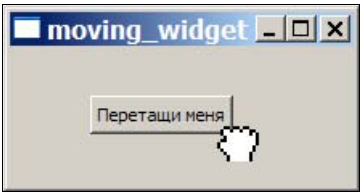


Рис. 28.1. Перемещение виджета с помощью мыши

Листинг 28.1. Файл chap28/moving_widget/moving_widget.h

```
#include <QtGui>

class MyButton : public QPushButton {
  Q_OBJECT
public:
  MyButton(const QString& text, QWidget *parent = 0);
  void mousePressEvent(QMouseEvent *event);
  void mouseMoveEvent(QMouseEvent *event);
  void mouseReleaseEvent(QMouseEvent *event);

  QPoint m_offset;

};
```

Листинг 28.2. Файл chap28/moving_widget/moving_widget.cpp

```
#include "moving_widget.h"

MyButton::MyButton(const QString& text, QWidget *parent)
        : QPushButton(text, parent) { }

void MyButton::mousePressEvent(QMouseEvent *event) {
   event->accept();
   setCursor(Qt::ClosedHandCursor);
   if (isWindow())
      m_offset = event->globalPos() - pos();
   else
      m_offset = event->pos();
}
```

```
27
28 void MyButton::mouseMoveEvent(QMouseEvent *event) {
29    event->accept();
30    if (isWindow())
31       move(event->globalPos() - m_offset);
32    else
33       move(mapToParent(event->pos() - m_offset));
34 }
35
36 void MyButton::mouseReleaseEvent(QMouseEvent *event) {
37    event->accept();
38    m_offset = QPoint();
39    setCursor(Qt::OpenHandCursor);
40 }
41
42 //-------------------------------------------------
43 int main(int argc, char *argv[]) {
44
45    QApplication app(argc, argv);
46
47    QTextCodec *codec = QTextCodec::codecForName("CP1251");
48    QTextCodec::setCodecForTr(codec);
49
50    QFrame *win = new QFrame();
51    win->resize(400, 300);
52    MyButton *widget = new MyButton(
53                   QObject::tr("Перетащи меня"), win);
54    widget->move(100, 150);
55    widget->setCursor(Qt::OpenHandCursor);
56    win->show();
57
58    return app.exec();
59 }
```

28.3. Заголовок окна

ЗАДАЧА. Убрать стандартные кнопки в заголовке окна или весь заголовок.

Решение. Для управления внешним видом окна и его заголовка используется свойство `QWidget::windowFlags`, хранящее набор битовых флагов, и функции доступа: `windowsFlags()` для чтения, `setWindowFlags()` для записи (листинг 28.3).

Листинг 28.3. Файл chap28/winflags/winflags.cpp

```
#include <QApplication>
#include <QMainWindow>

int main(int argc, char *argv[]) {

    QApplication app(argc, argv);
    QMainWindow mw;
    Qt::WindowFlags f = mw.windowFlags();

    // Убрать кнопки "Свернуть" и "Восстановить":
    f &= ~Qt::WindowMinMaxButtonsHint;

    // Убрать кнопку "Закрыть" и системное меню:
    f &= ~Qt::WindowSystemMenuHint;

    // Убрать сам заголовок:
    f |= Qt::CustomizeWindowHint;
    f &= ~Qt::WindowTitleHint;

    mw.setWindowFlags( f );
    mw.resize(400, 300);
    mw.show();
    return app.exec();
}
```

ЗАДАЧА. Элементы декорации и управления, находящиеся в заголовке окна, не реализованы в библиотеке Qt, их отрисовка осуществляется средствами нативного API, в результате мы не можем изменять их внешний вид и обрабатывать связанные с ними события. Поэтому требуется самостоятельно разработать заголовок со стандартными кнопками сворачивания, раскрытия и закрытия окна.

Решение. Текст программы приведен в листингах 28.4 и 28.5, внешний вид окна показан на рис. 28.2.

Листинг 28.4. Файл chap28/titlebar/titlebar.h

```
#include <QtGui>

class TitleBar : public QWidget {
    Q_OBJECT
public:
    TitleBar(QWidget *parent);
```

```
public slots:
    void showSmall();
    void showMaxRestore();
protected:
    void mousePressEvent(QMouseEvent *me);
    void mouseMoveEvent(QMouseEvent *me);
private:
    QToolButton *minimize;
    QToolButton *maximize;
    QToolButton *close;
    QPixmap restorePix, maxPix;
    bool maxNormal;
    QPoint startPos;
    QPoint clickPos;
};

class Frame : public QFrame {
public:
    Frame();
    QWidget *contentWidget() const;
    TitleBar *titleBar() const;
    void mousePressEvent(QMouseEvent *e);
    void mouseMoveEvent(QMouseEvent *e);
    void mouseReleaseEvent(QMouseEvent *);
private:
    TitleBar *m_titleBar;
    QWidget *m_content;
    QPoint m_old_pos;
    bool m_mouse_down;
    bool left, right, bottom;
};
```

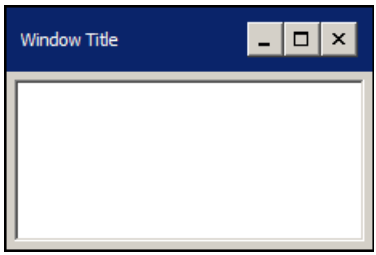


Рис. 28.2. Изменение вида заголовка окна

Листинг 28.5. Файл chap28/titlebar/titlebar.cpp

```
#include "titlebar.h"

TitleBar::TitleBar(QWidget *parent) {
    setAutoFillBackground(true);
    setBackgroundRole(QPalette::Highlight);
    minimize = new QToolButton(this);
    maximize = new QToolButton(this);
    close = new QToolButton(this);

    QPixmap pix = style()->standardPixmap(
        QStyle::SP_TitleBarCloseButton);
    close->setIcon(pix);

    maxPix = style()->standardPixmap(
        QStyle::SP_TitleBarMaxButton);
    maximize->setIcon(maxPix);

    pix = style()->standardPixmap(QStyle::SP_TitleBarMinButton);
    minimize->setIcon(pix);

    restorePix = style()->standardPixmap(
        QStyle::SP_TitleBarNormalButton);

    minimize->setMinimumHeight(20);
    close->setMinimumHeight(20);
    maximize->setMinimumHeight(20);

    QLabel *label = new QLabel(this);
    label->setText("Window Title");
    parent->setWindowTitle("Window Title");

    QHBoxLayout *hbox = new QHBoxLayout(this);

    hbox->addWidget(label);
    hbox->addWidget(minimize);
    hbox->addWidget(maximize);
    hbox->addWidget(close);

    hbox->insertStretch(1, 500);
    hbox->setSpacing(0);
    setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Fixed);
    maxNormal = false;
```

```
    connect( close, SIGNAL( clicked() ),
             parent, SLOT(close() ) );
    connect( minimize, SIGNAL( clicked() ),
             this, SLOT(showSmall() ) );
    connect( maximize, SIGNAL( clicked() ),
             this, SLOT(showMaxRestore() ) );
}

void TitleBar::showSmall() {
    parentWidget()->showMinimized();
}

void TitleBar::showMaxRestore() {
    if (maxNormal) {
        parentWidget()->showNormal();
        maxNormal = !maxNormal;
        maximize->setIcon(maxPix);
    } else {
        parentWidget()->showMaximized();
        maxNormal = !maxNormal;
        maximize->setIcon(restorePix);
    }
}

void TitleBar::mousePressEvent(QMouseEvent *me) {
    startPos = me->globalPos();
    clickPos = mapToParent(me->pos());
}

void TitleBar::mouseMoveEvent(QMouseEvent *me) {
    if (maxNormal)
        return;
    parentWidget()->move(me->globalPos() - clickPos);
}

//----------------------------------------
Frame::Frame() {
    m_mouse_down = false;
    setFrameShape(Panel);

    setWindowFlags(Qt::FramelessWindowHint);
    setMouseTracking(true);
```

```
    m_titleBar = new TitleBar(this);
    m_content = new QWidget(this);

    QVBoxLayout *vbox = new QVBoxLayout(this);
    vbox->addWidget(m_titleBar);
    vbox->setMargin(0);
    vbox->setSpacing(0);

    QVBoxLayout *layout = new QVBoxLayout(this);
    layout->addWidget(m_content);
    layout->setMargin(5);
    layout->setSpacing(0);
    vbox->addLayout(layout);
}

QWidget *Frame::contentWidget() const {
    return m_content;
}

TitleBar *Frame::titleBar() const {
    return m_titleBar;
}

void Frame::mousePressEvent(QMouseEvent *e) {
    m_old_pos = e->pos();
    m_mouse_down = e->button() == Qt::LeftButton;
}

void Frame::mouseMoveEvent(QMouseEvent *e) {
    int x = e->x();
    int y = e->y();
    if (m_mouse_down) {
        int dx = x - m_old_pos.x();
        int dy = y - m_old_pos.y();

        QRect g = geometry();

        if (left)
            g.setLeft(g.left() + dx);
        if (right)
            g.setRight(g.right() + dx);
        if (bottom)
            g.setBottom(g.bottom() + dy);
```

```
        setGeometry(g);
        m_old_pos = QPoint(!left ? e->x() : m_old_pos.x(), e->y());
    } else {
        QRect r = rect();
        left = qAbs(x - r.left()) <= 5;
        right = qAbs(x - r.right()) <= 5;
        bottom = qAbs(y - r.bottom()) <= 5;
        bool hor = left | right;

        if (hor && bottom) {
            if (left)
                setCursor(Qt::SizeBDiagCursor);
            else
                setCursor(Qt::SizeFDiagCursor);
        } else if (hor) {
                setCursor(Qt::SizeHorCursor);
        } else if (bottom) {
                setCursor(Qt::SizeVerCursor);
        } else {
                setCursor(Qt::ArrowCursor);
        }
    }
}

void Frame::mouseReleaseEvent(QMouseEvent * ) {
    m_mouse_down = false;
}

//---------------------------------------
int main(int argc, char **argv){
    QApplication app(argc, argv);
    Frame win;
    win.move(0,0);

    QVBoxLayout *lay = new QVBoxLayout(win.contentWidget());
    lay->setMargin(0);
    QTextEdit *edit = new QTextEdit(win.contentWidget());
    lay->addWidget(edit);

    win.show();
    return app.exec();
}
```

28.4. Область прокрутки

ЗАДАЧА. Отобразить на форме графическое изображение с возможностью прокрутки по вертикали и горизонтали (рис. 28.3).

Рис. 28.3. Область прокрутки

Решение. Необходимо использовать класс `QScrollArea` (листинг 28.6).

Листинг 28.6. Файл chap28/scroll/scroll.cpp

```
#include <QtGui>

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

    QLabel *imageLabel = new QLabel;
    imageLabel->setPixmap(QPixmap("picture.jpeg"));

    QScrollArea scrollArea;
    scrollArea.setBackgroundRole(QPalette::Dark);
    scrollArea.setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
    scrollArea.setHorizontalScrollBarPolicy(Qt::ScrollBarAsNeeded);
    scrollArea.setWidget(imageLabel);
    scrollArea.show();

    return app.exec();
}
```

При выполнении метода `QScrollArea::setWidget()` область прокрутки становится родителем для добавляемого виджета, поэтому при создании надписи `QLabel` мы не указали родителя.

28.5. Сведения о текущей платформе

Библиотека Qt предназначена для разработки платформонезависимых приложений, но иногда все-таки приходится обращаться к возможностям, реали-

зованным только в определенной операционной системе или работающим поразному в различных системах. Поэтому полезно знать, на какой платформе программа компилируется и/или выполняется, а также какая версия самой библиотеки Qt при этом используется.

- Чтобы узнать версию установленной библиотеки Qt, нужно ввести в командной строке:

```
qmake -v
```

  В некоторых системах Linux вместо `qmake` следует писать `qmake-qt4`.

- При компиляции программы в системе Windows всегда определен символ `Q_WS_WIN`, в Linux — `Q_WS_X11`, в Mac OS — `Q_WS_MAC`, в Qtopia Core — `Q_WS_QWS`. Поэтому фрагменты программы, относящиеся к конкретной платформе, можно оформлять с помощью условной директивы препроцессора, например:

```
#ifdef Q_WS_WIN
// Фрагмент, предназначенный для работы в системе Microsoft Windows
#endif
```

- При выполнении программы можно узнать, в какой версии Windows или Mac OS она запущена. Для этого в классе `QSysInfo` определены перечисления `WinVersion` и `MacVersion`.

- Получить сведения о версии библиотеки Qt во время выполнения программы можно с помощью класса `QLibraryInfo`.

ЗАДАЧА. Вывести на экран основные сведения о текущей платформе и библиотеке Qt.

Решение. Текст небольшой программы приведен в листинге 28.7, а внешний вид окна приложения в системе Windows показан на рис. 28.4.

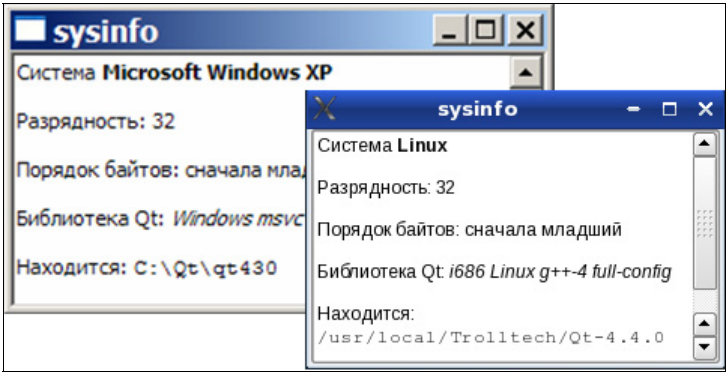


Рис. 28.4. Получение сведений о текущей платформе

Листинг 28.7. Файл chap28/sysinfo/sysinfo.cpp

```
#include <QtGui>

int main(int argc, char **argv) {

   QApplication app(argc, argv);
   QTextCodec *codec = QTextCodec::codecForName("CP1251");
   QTextCodec::setCodecForTr(codec);

   QTextEdit te;
   te.setReadOnly(true);
   QString str;
   QTextStream s(&str,  QIODevice::ReadOnly);
   QString v;

   #ifdef Q_WS_WIN
      switch ( QSysInfo::WindowsVersion  ) {
      case QSysInfo::WV_98:    v = "98";    break;
      case QSysInfo::WV_Me:    v = "ME";    break;
      case QSysInfo::WV_NT:    v = "NT";    break;
      case QSysInfo::WV_2000:  v = "2000";  break;
      case QSysInfo::WV_XP:    v = "XP";    break;
      case QSysInfo::WV_2003:  v = "2003";  break;
      case QSysInfo::WV_VISTA: v = "Vista"; break;
      default:                 v = "";
      }
      s << QObject::tr(
          "Система <b>Microsoft Windows %1</b><p>").arg(v);
   #endif

   #ifdef Q_WS_X11
      s << QObject::tr("Система <b>Linux</b><p>").arg(v);
   #endif

   s << QObject::tr("Разрядность: %1<p>").arg(QSysInfo::WordSize);

   switch ( QSysInfo::ByteOrder ) {
   case QSysInfo::BigEndian:
        v = QObject::tr("сначала старший"); break;
   case QSysInfo::LittleEndian:
        v = QObject::tr("сначала младший"); break;
   default: v = "???";
   }
```

```
43      s << QObject::tr("Порядок байтов: %1<p>").arg(v);
44
45      s << QObject::tr("Библиотека Qt: <i>%1</i><p>")
46            .arg( QLibraryInfo::buildKey() );
47      s << QObject::tr("Находится: <tt>%1</tt><p>")
48            .arg( QLibraryInfo::location(QLibraryInfo::PrefixPath));
49
50      te.setText(str);
51      te.show();
52      return app.exec();
53  }
```

28.6. Бегущая строка

ЗАДАЧА. Требуется отобразить на форме "бегущую строку".

Решение. Для решения задачи используется таймер `QTimer` (рис. 28.5, листинги 28.8–28.10).

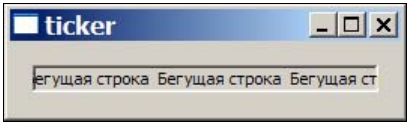


Рис. 28.5. Бегущая строка

Листинг 28.8. Файл chap28/ticker/ticker.h

```
#include <QtGui>

class Ticker : public QFrame {
    Q_OBJECT
    Q_PROPERTY(QString text READ text WRITE setText)
    Q_PROPERTY(int speed READ speed WRITE setSpeed)
public:
    Ticker(const QString& text, QWidget *parent = 0);
    void setText(const QString &text) { txt = text; update(); }
    QString text() const { return txt; }
    QSize sizeHint() const;
    void setSpeed(int speed) { timer.setInterval(speed); }
    int speed() const { return timer.interval(); }
protected:
    void showEvent(QShowEvent *) { timer.start(); }
```

```
    void hideEvent(QHideEvent *) { timer.stop(); }
    void paintEvent(QPaintEvent *event);
private slots:
    void progress();
private:
    QString txt;
    QTimer timer;
    int offset;
};
```

Листинг 28.9. Файл chap28/ticker/ticker.cpp

```
#include "ticker.h"

Ticker::Ticker(const QString& text, QWidget *parent)
      : QFrame(parent), txt(text), offset(0) {

    setFrameStyle( WinPanel | Sunken );
    setBackgroundRole( QPalette::Base );
    setSizePolicy( QSizePolicy::Expanding, QSizePolicy::Fixed );
    timer.setInterval(20); // мс
    connect(&timer, SIGNAL(timeout()), this, SLOT(progress()));
}

QSize Ticker::sizeHint() const {
    return QSize(
        40 * fontMetrics().width('x'),
        fontMetrics().lineSpacing() + 2 * frameWidth());
}

void Ticker::paintEvent(QPaintEvent *event) {
    QFrame::paintEvent(event);
    QPainter painter(this);
    for (int x=0; x<width() - offset; x += fontMetrics().width(txt))
        painter.drawText( x + offset, 0, 65536,
                          height(), Qt::AlignVCenter, txt );
}

void Ticker::progress() {
    if (--offset < -fontMetrics().width(txt)) offset = 0;
    update();
}
```

Листинг 28.10. Файл chap28/ticker/main.cpp

```
#include "ticker.h"

int main(int argc, char *argv[]) {
    QApplication app(argc, argv);
    QTextCodec *codec = QTextCodec::codecForName("CP1251");
    QTextCodec::setCodecForTr(codec);

    QWidget window;
    QGridLayout layout(&window);
    layout.setMargin(15);
    Ticker ticker(QObject::tr("Бегущая строка  "));
    layout.addWidget(&ticker, 1, 1);
    window.show();
    return app.exec();
}
```

28.7. Работа с переменными окружения

ЗАДАЧА. Требуется узнать путь к каталогу пользователя и к его рабочему столу в системе Windows.

Решение. Для получения значений переменных окружения в библиотеке Qt имеется функция `qgetenv(const char *varName)`. Полное имя каталога текущего пользователя хранится в переменной окружения `PROFILE`. Например, для пользователя `user` это обычно `"C:\Documents and Settings\user"`. Элементы рабочего стола хранятся в подкаталоге Desktop (для нелокализованной системы) или его аналоге. В русифицированной системе Windows эта папка называется **Рабочий стол** (листинг 28.11). Чтобы узнать ее имя в общем случае, придется обратиться к Windows API.

Листинг 28.11. Файл chap28/envvars/envvars.cpp

```
#include <QtGui>
#include <iostream>

int main(int argc, char **argv){
 QApplication app(argc, argv);
 QTextCodec *incodec = QTextCodec::codecForName("CP1251");
 QTextCodec::setCodecForTr(incodec);
 QTextCodec *outcodec = QTextCodec::codecForName("CP866");
```

```
 QString str = qgetenv("USERPROFILE");
 std::cout << outcodec->fromUnicode( QObject::tr(
        "User profile: ") + str).constData()
      << std::endl ;
 str += QObject::tr("\\Рабочий стол");
 std::cout << outcodec->fromUnicode( QObject::tr(
        "User desktop: ") + str).constData()
      << std::endl ;
 return app.exec();
}
```

В файле *.pro не забудьте добавить строку:

```
win32: CONFIG += console
```

28.8. Типы MIME

MIME — Multipurpose Internet Mail Extensions (многоцелевые расширения почтовой службы Интернет) — стандарт, изложенный в RFC 2045, 2046 и 2047. Он определяет обозначения и формат представления для данных различного типа:

- `text/plain` — простой текст;
- `text/html` — форматированный текст;
- `text/uri-list` — список URL (в том числе список имен файлов);
- `image/*` — графическое изображение (вместо звездочки пишется обозначение графического формата, например, `png`).

Для работы с типами MIME в библиотеке Qt имеется класс `QMimeData`.

ЗАДАЧА. Упаковать изображение в объект `QMimeData` для последующей передачи через буфер обмена или механизм "drag&drop".

Решение иллюстрирует листинг 28.12.

Листинг 28.12

```
QMimeData* packImage (const QString& filename){
   QImage image(filename);
   QMimeData *mimeData = new QMimeData;
   mimeData->setImageData(image);
   return mimeData;
}
```

ЗАДАЧА. Упаковать в объект `QMimeData` само графическое изображение, имя файла, в котором оно хранится, и текстовое описание картинки.

Решение иллюстрирует листинг 28.13.

Листинг 28.13

```
QMimeData* packImage (const QString& filename,
                      const QString& comment ) {
   QImage image(filename);
   QMimeData *mimeData = new QMimeData;
   mimeData->setImageData(image);

   QList<QUrl> urls;
   QUrl url(filename);
   urls.append(url);
   mimeData->setUrls(urls);

   mimeData->setText(comment);

   return mimeData;
}
```

28.9. Буфер обмена

Обмен данными между различными приложениями часто осуществляется через так называемый *буфер обмена*. В библиотеке Qt он представлен классом `QClipboard`. Единственный экземпляр этого класса создается автоматически при запуске приложения. В буфере обмена могут находиться данные различного типа: простой текст; форматированный текст, заданный на языке HTML; графическое изображение и т. п. Соответственно, получить данные из буфера можно с помощью различных методов: `QClipboard::text()`, `QClipboard::html()`, `QClipboard::image()`, `QClipboard::pixmap()`. Для получения данных произвольного типа служит метод `QClipboard::mimeData()`, возвращающий объект типа `QMimeData`. Для записи данных в буфер обмена имеются соответствующие `set`-методы: `QClipboard::setText()`, `setHtml()`, `setImage()`, `setPixmap()`, `setMimeData()`. При изменении данных в буфере обмена (даже если это произошло по инициативе другого приложения) генерируется сигнал `dataChanged()`.

Стандартные виджеты библиотеки Qt, например `QLineEdit` или `QTextEdit`, уже умеют работать с буфером обмена, т. е. воспринимают нажатие клавиатурных комбинаций <Ctrl>+<C>, <Ctrl>+<X> и <Ctrl>+<V> как команды "Копиро-

вать", "Вырезать" и "Вставить" соответственно, а также отображают эти команды в своем контекстном меню. Но иногда требуется запрограммировать какие-либо специфические операции с буфером обмена или добавить его поддержку к своему собственному элементу управления.

ЗАДАЧА. Преобразовать текст, находящийся в буфере обмена, к верхнему регистру. Если в буфере обмена находятся нетекстовые данные, то ничего не делать.

Решение. Текст программы приведен в листингах 28.14 и 28.15.

Листинг 28.14. Файл chap28/clipboard/clipboard.h

```
#include <QtGui>

class MyApp : public QApplication {
   Q_OBJECT
public:
   MyApp(int& argc, char **argv)
      : QApplication(argc, argv) {
      QClipboard *clipboard = QApplication::clipboard();
      connect(clipboard, SIGNAL(dataChanged()),
              this, SLOT(onDataChanged()));
  }
public slots:
   void onDataChanged() {
      QClipboard *clipboard = QApplication::clipboard();
      QString s = clipboard->text();
      if ( s.length() ) {
         s = s.toUpper();
         clipboard->setText(s);
      }
   }
};
```

Листинг 28.15. Файл chap28/clipboard/clipboard.cpp

```
#include "clipboard.h"

int main(int argc, char** argv) {
   MyApp app(argc, argv);
   QTextEdit te;

```

```
28      te.show();
29      return app.exec();
30  }
```

Запустите эту программу и попробуйте поработать с операциями копирования-вставки в любом текстовом редакторе, например в Microsoft Word.

28.10. Перетаскивание файлов в окно текстового редактора

> Главное в этом мире не то, где мы стоим, а то, в каком направлении движемся.
>
> *Оливер Уэнделл Холмс*

ЗАДАЧА. Реализовать простейший текстовый редактор на основе элемента `QTextEdit`, который мог бы принимать файлы, перетаскиваемые в его окно с помощью мыши.

Решение. Нам необходимо переопределить всего два виртуальных метода, имеющихся у всех потомков класса `QWidget`: `dragEnterEvent()` и `dropEvent()`. Первый метод вызывается, когда перетаскиваемый объект входит в область, ограничиваемую виджетом. При этом нужно проанализировать тип перемещаемого объекта: если это файл (тип MIME `"text/uri-list"`), то мы разрешаем операцию перетаскивания. Второй метод вызывается, когда пользователь отпускает кнопку мыши, "бросая" перетаскиваемый объект в пределах области, занимаемой виджетом. В этом случае остается только получить имя файла (если файлов несколько, мы принимаем во внимание только первый из них) и загрузить его содержимое в окно нашего редактора.

Текст программы приведен в листингах 28.16 и 28.17.

Листинг 28.16. Файл chap28/dndfile/dndfile.h

```
1  #include <QtGui>
2
3  class MainWin : public QMainWindow {
4      Q_OBJECT
5  public:
6      MainWin();
7  protected:
8      void dragEnterEvent(QDragEnterEvent *event);
9      void dropEvent(QDropEvent *event);
```

```
10 private:
11     bool readFile(const QString &fileName);
12     QTextEdit *textEdit;
13 };
```

Листинг 28.17. Файл chap28/dndfile/dndfile.cpp

```
14 #include "dndfile.h"
15
16 MainWin::MainWin() {
17    textEdit = new QTextEdit;
18    setCentralWidget(textEdit);
19    textEdit->setAcceptDrops(false);
20    setAcceptDrops(true);
21 }
22
23 void MainWin::dragEnterEvent(QDragEnterEvent *event) {
24    if (event->mimeData()->hasFormat("text/uri-list"))
25       event->acceptProposedAction();
26 }
27
28 void MainWin::dropEvent(QDropEvent *event) {
29   QList<QUrl> urls = event->mimeData()->urls();
30   if (urls.isEmpty())
31      return;
32   QString fileName = urls.first().toLocalFile();
33   if (fileName.isEmpty())
34      return;
35   if (readFile(fileName))
36      setWindowTitle(tr("%1 — %2").arg(fileName)
37                                  .arg(tr("Drag File")));
38 }
39 bool MainWin::readFile(const QString &fileName){
40    QFile file(fileName);
41    if (!file.open(QFile::ReadOnly | QFile::Text)) {
42       QMessageBox::warning(this, tr("Application"),
43                            tr("Cannot read file %1:\n%2.")
44                              .arg(fileName)
45                              .arg(file.errorString()));
46       return false;
47   }
48   QTextStream in(&file);
49   QApplication::setOverrideCursor(Qt::WaitCursor);
```

```
50    textEdit->setPlainText(in.readAll());
51    QApplication::restoreOverrideCursor();
52    return true;
53  }
54  // -------------------------------
55  int main(int argc, char *argv[]){
56     QApplication app(argc, argv);
57     MainWin win;
58     win.show();
59     return app.exec();
60  }
```

Разумеется, программа нуждается в доработке. Ведь мы просто затираем весь исходный и загружаем новый текст из файла. Возможно, требуется сначала спросить пользователя о нужном действии, чтобы затем или отказаться от выполнения операции, или вставить текст из файла в середину имеющегося текста, начиная с текущей позиции курсора.

Заключение

Уходят мудрые от дома, как лебеди покинув пруд.
Им наша жажда не знакома — увидеть завершенным труд.
Им ничего не жаль на свете: ни босых ног своих, ни лет.
Их путь непостижим и светел, как в небе лебединый след.
Дхаммапада

Вот и подошло к концу наше путешествие в мир Qt. Не все возможности этой замечательной библиотеки нашли свое отражение в данной книге, но если читатель, перевернув последнюю страницу, почувствует, что он готов продолжить свое знакомство с Qt, готов вернуться к рассмотренным здесь примерам, но уже за экраном компьютера, чтобы экспериментируя с объектами и их свойствами, пробуя разные варианты и предлагая свои решения, постепенно перейти к настоящим приложениям и вдруг понять, что эта книга ему больше не нужна, — тогда автор будет считать свою задачу выполненной.

Все замечания и пожелания вы можете присылать по адресу электронной почты **george-zemskov@mail.ru**.

ПРИЛОЖЕНИЕ

Описание содержимого компакт-диска

На прилагаемом компакт-диске можно найти исходные тексты всех рассмотренных в книге примеров, а также установочные файлы библиотеки Qt 4 и набор полезных утилит.

Каталог	Описание
examples/chap02	Примеры к главе 2 console — консольная программа; template — использование шаблонов в C++
examples/chap03	Примеры к главе 3 emptywin — пустое окно с заголовком; emptywin-rus1, emptywin-rus2, emptywin-rus3 — варианты использования символов кириллицы в исходном тексте программы; qtconsole — консольное приложение Qt; mywin — шаблон приложения с классом виджета
examples/chap04	Примеры к главе 4 label — надпись QLabel; piclabel — вставка картинок в надпись; clicklabel — ссылки в тексте надписи; boxlayout, gridlayout — использование менеджеров размещения; lineedit — однострочное поле ввода; inoutstring — ввод-вывод текстовой информации

(продолжение)

Каталог	Описание
examples/chap05	Примеры к главе 5 btnquit — кнопка "Выход"; signal_slot — сигналы и слоты; slider — работа с QSlider; idle — измерение периода неактивности пользователя; altf4 — перехват нажатия комбинации клавиш <Alt>+<F4>; mycheckbox — элемент QCheckbox с необычным поведением; mywin — запрет на закрытие окна приложения
examples/chap06	Примеры к главе 6 pushbuttons — кнопки нажатия; radiobuttons — радиокнопки; btngroup — группы кнопок; addbutton — добавление виджетов к стандартным диалогам; click — программная эмуляция нажатия кнопок; dialog — использование диалога в главной программе; embeddedwin — встраивание главного окна внутрь диалога; selectdirs — выбор нескольких каталогов
examples/chap07	Примеры к главе 7 action — работа с действиями; statusbar — виджеты в строке состояния; menumouse — главное меню и работа с событиями мыши; breakmenu — разрыв в строке меню
examples/chap08	Примеры к главе 8 resizeevent — обработка события изменения размеров окна; autoresize — автоматическое изменение размеров элементов; sizehint — работа с QSizePolicy
examples/chap09	Примеры к главе 9 designer1, designer2, designer3, designer4 — варианты разработки интерфейса приложения в Qt Designer; qdebug — вывод отладочных сообщений

(продолжение)

Каталог	Описание
examples/chap10	Пример к главе 10 compare — поэлементное сравнение разнотипных последовательностей данных
examples/chap11	Примеры к главе 11 drives — список дисков/каталогов; zlib — сжатие данных
examples/chap12	Примеры к главе 12 app1 — простое приложение SDI; app2 — более сложное приложение SDI (текстовый редактор); mdi — приложение MDI; dock — плавающие и прикрепляемые окна; child2top — динамическое изменение родителя окна; settings — хранение настроек приложения
examples/chap13	Примеры к главе 13 html — работа с форматированным текстом; textdocmargins — поля вокруг документа и абзаца; clock — разрабатываем свой виджет (часы); print1 — вывод на печать графического изображения; printtext — печать текста; printwidget — выводим изображение виджета на принтер; scene — работа с графическими сценами; customitem1, customitem2, customitem3 — создание пользовательских элементов графических сцен; hoveritem — интерактивные элементы графических сцен; previewtext, previewtext2 — предварительный просмотр перед выводом на принтер
examples/chap14	Примеры к главе 14 listwidget — список элементов; clickedlist — реакция элементов списка на щелчки мыши; table — работа с таблицей; sortindicator1, sortindicator2 — сортировка строк таблицы; doubleclick — чем двойной щелчок отличается от одиночного;

(продолжение)

Каталог	Описание
	multisort — интерактивная сортировка по нескольким столбцам; treewidget — дерево элементов; sortedtreewidget — сортировка элементов дерева; checkitem — элементы дерева с пометками; expansion — перехват события разворачивания групп дерева
examples/chap15	Примеры к главе 15 csvmodel — модель таблицы с хранением данных в текстовом файле; listwidget — выравнивание элементов списка; checkboxalign — расположение галочек внутри клеток таблицы; dirview — дерево каталогов и файлов; treemodel — иерархическая модель данных и ее отображение в виде дерева; treemodelview — дерево групп и таблица элементов; itemsiterate — перебор элементов дерева; sharemodel — одна модель на несколько представлений; checkcombo — разворачивающийся список с пометками
examples/chap16	Примеры к главе 16 process — запуск процесса и взаимодействие с ним; consprod — модель производитель-потребитель; moviethread — многопоточная мультипликация; run — многопоточное приветствие от QtConcurrent; qfuture — однотипная обработка набора данных; images — масштабирование нескольких изображений одновременно в блокирующем режиме
examples/chap17	Примеры к главе 17 db00 — подключение к базе данных и выполнение запросов SQL; db01 — работа с таблицами
examples/chap18	Примеры к главе 18 db02 — модель таблицы базы данных и ее представление; db03 — делегаты для ячеек таблицы; db04 — форма для редактирования данных; db05 — связывание таблиц; sqlquery — консоль запросов; asyncdb — выполнение запросов SQL в отдельном потоке

(продолжение)

Каталог	**Описание**
examples/chap19	Примеры к главе 19 streamreader — чтение файла XML; xml-dom-read — чтение с помощью интерфейса DOM; xml-dom-write — запись с помощью интерфейса DOM; xml-dom-rw — чтение и запись с помощью интерфейса DOM; xml-sax-read — чтение с помощью интерфейса SAX; xquery — использование языка запросов
examples/chap20	Примеры к главе 20 echo1 — эхо-сервер в блокирующем режиме и клиент для него; echo2 — асинхронный однопоточный эхо-сервер; echo3 — многопоточный эхо-сервер; smtp — отправка электронной почты; ftpclient — клиент FTP; webbrowser — простой Web-браузер
examples/chap21	Примеры к главе 21 qscript1 — выполнение строк скрипта из приложения Qt; qscript2 — запуск скрипта, находящегося в отдельном файле; qscript3 — скрипт управляет внешним видом виджета; qscript4 и qscript4a — скрипт работает с сигналами; qscript5 — связывание сигнала и слота в тексте скрипта; qscript6 — скрипт сам создает для себя виджеты; db02a — расширяемое пользователем приложение БД
examples/chap22	Примеры к главе 22 readexcel — чтение из таблицы Excel; mplayer — используем Microsoft Media Player; svgmovie — показ анимированной картинки; video — видеопроигрыватель AVI/MPEG
examples/chap23	Примеры к главе 23 palette — работа с палитрами; removesub — удаляем часть виджета; subcontrol1, subcontrol2 — работа со стилями

(окончание)

Каталог	Описание
examples/chap24	Примеры к главе 24 app1 — пример локализации приложений
examples/chap25	Примеры к главе 25 00 — выполнение отдельных строк исходного текста Python; 01 — выполнение скрипта Python, загруженного из файла; 02, 03 — импорт модуля Python; pyqt — примеры приложений PyQt
examples/chap26	Примеры к главе 26 dynlib — простая динамическая библиотека; app1 — пример статической компоновки; app2 — динамическая загрузка библиотеки; led — плагин для Qt Designer
examples/chap27	Примеры к главе 27 wx00 — простое приложение wxWidgets; wx01 — меню, строка состояния и обработка перемещения указателя мыши
examples/chap28	Примеры к главе 28 titlebar — наш заголовок окна; envvars — работа с переменными окружения; scroll — прокрутка изображения; sysinfo — сведения о системе; moving_widget — перемещение виджетов с помощью мыши; clearclipboard — работа с буфером обмена; dndfile — перетаскивание файла на окно текстового редактора
Install/Qt	Библиотека Qt 4.4.0
Install/QDevelop	Программа QDevelop 0.25
Install/MinGW32	Компилятор MinGW32
Install/Python2.5	Python 2.5 для Windows
Install/PythonQt	Библиотека PythonQt
Install/ PyQt4	Библиотека PyQt4

Список литературы

Книг бесконечно много, а время коротко; поэтому секрет приобретения знаний заключается в том, чтобы брать существенное. Выбирайте его и затем старайтесь жить согласно с ним.

Свами Вивекананда

1. Шлее М. Qt4. Профессиональное программирование на C++. — СПб.: БХВ-Петербург, 2007. — 880 с.
2. Бланшет Ж., Саммерфилд М. Qt4. Программирование GUI на C++. 2-е изд. — М.: КУДИЦ-Пресс, 2007. — 648 с.
3. Чеботарев А. В. Библиотека Qt4. Создание прикладных приложений в среде Linux. — М.: Диалектика, 2006. — 256 с.
4. Шилдт Г. Полный справочник по C++. 4-е изд. — М.: Вильямс, 2006. — 800 с.
5. Стефенс Д. Р. C++. Сборник рецептов. — М.: КУДИЦ-Пресс, 2007. — 624 с.

Предметный указатель

A

Action 115
ActiveQt 495

C

CLR 566
COM 495
connect 71
CSV 288

D

disconnect 73
DOM 432

E

ECMAScript 479
emit 73
Excel 495

F

FTP 466

I

IDE 15, 17

L

L&F 511
Linux, embedded 9

M

MDI 189, 203
MIME 588

N

.NET 566

P

Phonon 503

Q

Q_OBJECT 54
QApp 72
QButtonGroup 101
QCheckBox 80
QDataStream 176
QDevelop 18
QDialogButtonBox 103
QDomDocument 432
QDomNode 432
QEqual 162
QEvent 79

QFile 173
QGraphicsScene 254
QLabel 59
QLibrary 553
QLineEdit 68
QList 160
QListWidget 263
Qmake 45
QMenuBar 115
QMimeData 588
QObject 165
QPainter 222
QPalette 509
QPixmap 237
QPrinter 239
QPushButton 74, 91
QSettings 214
QSlider 75
QSpinBox 75
QStatusBar 113
QStringList 160
QStyleSheet 516
Qt Assistant 154
Qt Jambi 566
Qt Linguist 519
QTableWidget 267
QtConcurrent 350
QTextDocument 230
QTextEdit 228
QtHelp 157
QTimer 86
QtNetwork 447
QTranslator 519
QTreeWidget 278
QtScript 479
QtXmlPatterns 443
QVector 160

S

SAX 437
SDI 189
SMTP 467, 540
solution 26
SQL 408
STL 163
SVG 502

T

TCP 447
Template 32

U

UDP 447

W

WAVE 501
WebKit 475
wxWidgets 567

X

XML 431
XPath 443
XQuery 443, 444

А

Анимация 502
Антиалиасинг 222

Б

Библиотека:
◊ динамическая 553
◊ статическая 551
Браузер 475
Буфер обмена 589

В

Валидатор 147
Ввод текстовой строки 68
Виджет:
◊ печать 247
◊ создание 232

Г

Графическая сцена 254
◊ печать 261

Д

Действие 115
Дерево 278, 303
◊ сортировка элементов 280
Диалог 66
◊ модальный 104
◊ стандартный 108
Драйвер СУБД 359

И

Изображение, печать 249
Иконка приложения 185
Интерфейс:
◊ многодокументный 203
◊ однодокументный 189
Информация, сжатие 186
Итератор 163

К

Каталог 180
◊ приложения 181
Кнопка 74, 91
Кодек 50
Кодировка символов 48
Компилятор 15
◊ выбор 20
Компиляция:
◊ библиотеки Qt 40
◊ приложений Qt 45, 48
Консоль запросов 408

Л

Локализация 190, 519

М

Массив 160
Менеджер размещения 64, 134
Меню 113, 115, 123, 125
◊ контекстное 91
Метод виртуальный 80
Модель 287
◊ иерархическая 306
◊ таблицы БД 381
Мультипликация 346
Мьютекс 338

Н

Надпись:
◊ редактируемая 62
◊ с рисунком 63

О

Окно плавающее 211
Отладка 150
Отладчик 15

П

Палитра 509
Панель инструментов 125
Перегрузка 31
Плагин 557
Полоса прокрутки 72
Поток 333, 335, 423
◊ синхронизация 338
Потокобезопасность 336
Предварительный просмотр 252
Представление 287
Приложение:
◊ завершение 74
◊ консольное 21, 52
◊ хранение настроек 214
Принтер 239
Прокси-модель 321
Пространство имен 22
Протокол:
◊ FTP 466
◊ SMTP 467
◊ TCP 447
◊ UDP 447
Процесс 333

Р

Размещение 127
Регулярное выражение 147
Реентерабельность 336
Ресурс 96, 183, 184

С

Семафор 342
Сериализация 178
Сигнал 71
◊ звуковой 501
◊ сопоставление 89
Система помощи 154
Слот 71, 148
Событие 79
Совместное использование данных:
◊ неявное 167
◊ явное 170
Сокет 447
Список 160, 263
◊ выравнивание элементов 299
Сравнение поэлементное 162
Среда разработки приложений 15
Ссылки 61
Стиль 91, 511
◊ линий 227
Строка состояния 116
СУБД 359

Т

Таблица:
◊ базы данных 374
◊ вставка записи 375
◊ модель 287
◊ сортировка строк 270, 275
◊ удаление записи 376
Текст:
◊ форматирование 230
◊ форматированный 60, 232
Текстовый редактор 15, 201
◊ печать 250
Типы MIME 588
Транзакция 365

У

Условие ожидания 342

Ф

Файл:
◊ временный 180
◊ диалог выбора 185
◊ запись 175, 176
◊ копирование 181
◊ описания проекта Qt 45, 523
◊ открытие и сохранение 194
◊ текстовый 173
◊ чтение 174, 178
Файловый менеджер 23
Фильтр 84
Форматирование 67
Функция:
◊ обратного вызова 71
◊ перевода 51, 519

Ш

Шаблон 30

Э

Элемент графической сцены 256
Эхо-сервер 449
◊ асинхронный 456
◊ многопоточный 460

Я

Язык:
◊ ECMAScritp 479
◊ Java 565
◊ Python 531
◊ XML 431
◊ XPath 443
◊ XQuery 443