

КУДИЦ-ПРЕСС

МОСКВА • 2008

C++ GUI Programming with Qt 4

Second Edition

Jasmin Blanchette

Mark Summerfield

In association with Trolltech Press



Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Capetown • Sydney • Tokyo • Singapore • Mexico City

Qt 4: программирование GUI на C++

Издание второе, дополненное

Жасмин Бланшет
Марк Саммерфилд

ББК 32.973.26-018.1

Бланшет Ж., Саммерфилд М.

Qt 4: программирование GUI на C++. Пер. с англ. 2-е изд., доп. –

М.: КУДИЦ-ПРЕСС, 2008. – 736 с.

ISBN 978-5-91136-059-7

Книга представляет собой дополненное и исправленное издание востребованной на российском рынке книги «Qt4: программирование GUI на C++», выпущенной в 2007 году. Тираж быстро нашел своих покупателей. В новое издание были внесены изменения, связанные с использованием возможностей, появившихся в Qt версий 4.2 и 4.3, добавлены новые главы, посвященные настройке диалога с пользователем и созданию прикладных скриптов, рассмотрены базовые принципы программирования на Qt 4 b. В приложении В дано введение в Qt Jambi, официально поддерживаемую Java-версию API Qt, выпущенную компанией Trolltech в 2007 году.

Жасмин Бланшет, Марк Саммерфилд

Qt 4: программирование GUI на C++

Перевод с англ. яз. С. Лунин, В. Казаченко.

ООО «КУДИЦ-ПРЕСС»

190068, С.-Петербург, Вознесенский пр-т, д. 55, литер А, пом. 44

тел. (495) 333-82-11, ok@kudits.ru, <http://books.kudits.ru>

Подписано в печать 03.06.2008 г.

Формат 70x100/16. Бум. офс. Печать офс.

Усл. печ. л. 59,34. Тираж 2500. Заказ 1083

Отпечатано в полном соответствии с качеством предоставленных диапозитивов

в ППП «Типография «Наука»

121099, Москва, Шубинский пер., 6

ISBN 978-5-91136-059-7 (рус.)

© Макет, обложка ООО «КУДИЦ-ПРЕСС», 2008

ISBN 0132354160

Authorized translation from the English language edition, entitled C++ GUI PROGRAMMING WITH QT4, 2nd Edition, ISBN 0132354160, by BLANCHETTE, JASMIN; and SUMMERFIELD, MARK; published by Pearson Education, Inc., publishing as Prentice Hall, Copyright © 2008 Pearson Education, Inc.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from Pearson Education Inc. RUSSIAN language edition published by KUDITS-PRESS, Copyright © 2008.

Авторизованный перевод англоязычного издания C++ GUI PROGRAMMING WITH QT4, опубликованного Pearson Education, Inc. под издательской маркой *Prentice Hall*.

Все права защищены. Никакая часть данной книги не может быть воспроизведена в любой форме или любыми средствами, электронными или механическими, включая фотографирование, видео- или аудиозапись, а также любыми системами поиска информации без разрешения Pearson Education Inc.

Русское издание опубликовано издательством КУДИЦ-ПРЕСС, © 2008.

Содержание

Примечание редактора серии	ix
Вступление.....	x
Предисловие	xii
Благодарности	xiv
Краткая история Qt	xvi

Часть I.

Основные возможности средств разработки Qt

Глава 1. Первое знакомство	3
«Здравствуй, Qt»	3
Взаимодействие с пользователем	5
Компоновка виджетов	6
Стили виджетов	9
Использование справочной документации	10
Глава 2. Создание диалоговых окон.....	12
Подклассы QDialog	12
Подробное описание технологии сигналов и слотов	19
Метаобъектная система Qt	21
Быстрое проектирование диалоговых окон	22
Изменяющиеся диалоговые окна	31
Динамические диалоговые окна.....	38
Встроенные классы виджетов и диалоговых окон	39
Глава 3. Создание главных окон	44
Создание подкласса QMainWindow	45
Создание меню и панелей инструментов	49
Создание и настройка строки состояния	54
Реализация меню File	56
Применение диалоговых окон	63
Сохранение настроек приложения.....	69
Работа с несколькими документами	71
Экранные заставки	74
Глава 4. Реализация функциональности приложения	76
Центральный виджет	76
Создание подкласса QTableWidgetItem.....	77
Хранение данных в объектах типа «элемент»	82
Загрузка и сохранение	83
Реализация меню Edit	86
Реализация других меню	91
Создание подкласса QTableWidgetItem	95

Глава 5. Создание пользовательских виджетов	104
Настройка виджетов Qt	104
Создание подкласса QWidget	107
Интеграция пользовательских виджетов в Qt Designer	117
Двойная буферизация	120

Часть II.

Средний уровень Qt-программирования

Глава 6. Управление компоновкой	143
Компоновка виджетов на форме	143
Стековая компоновка	149
Разделители	151
Области с прокруткой	155
Прикрепляемые окна и панели инструментов	156
Многодокументный интерфейс	159
Глава 7. Обработка событий	170
Переопределение обработчиков событий	170
Установка фильтров событий	176
Обработка событий во время продолжительных процессов	179
Глава 8. Графика 2D	182
Рисование при помощи QPainter	183
Преобразования координатных систем	188
Высококачественное воспроизведение изображения при помощи QImage	196
Элементное воспроизведение с помощью графического представления	199
Вывод на печатающее устройство	221
Глава 9. Технология «drag-and-drop»	229
Обеспечение поддержки технологии «drag-and-drop»	229
Поддержка пользовательских типов переносимых объектов	235
Работа с буфером обмена	240
Глава 10. Классы отображения элементов	241
Применение удобных классов отображения элементов	243
Применение заранее определенных моделей	249
Реализация пользовательских моделей	255
Реализация пользовательских делегатов	270
Глава 11. Классы-контейнеры	276
Последовательные контейнеры	277
Как работает неявное совместное использование данных	285
Ассоциативные контейнеры	286
Обобщенные алгоритмы	289
Строки, массивы байтов и объекты произвольного типа	291
Глава 12. Ввод-вывод	298
Чтение и запись двоичных данных	299

Чтение и запись текста	305
Работа с каталогами	311
Ресурсы, внедренные в исполняемый модуль	312
Связь между процессами	313
Глава 13. Базы данных	319
Соединение с базой данных и выполнение запросов	320
Просмотр таблиц	326
Редактирование данных с использованием форм	329
Представление данных в табличной форме	335
Глава 14. Многопоточная обработка	343
Создание потоков	344
Синхронизация потоков	347
Взаимодействие с главным потоком	354
Применение классов Qt во вторичных потоках	360
Глава 15. Работа с сетью	363
Написание FTP-клиентов	363
Написание HTTP-клиента	373
Написание клиент-серверных приложений на базе TCP	376
Передача и прием дейтаграмм UDP	387
Глава 16. XML	392
Чтение XML-документов при помощи QDomStreamReader	393
Чтение документов XML при помощи интерфейса DOM	400
Чтение документов XML при помощи интерфейса SAX	405
Запись документов XML	409
Глава 17. Обеспечение интерактивной помощи	413
Всплывающие подсказки, комментарии в строке состояния и справки «что это такое?»	413
Использование web-браузера для предоставления интерактивной помощи	416
Использование QTextBrowser в качестве простого браузера системы помощи	418
Использование Qt Assistant для мощной интерактивной системы помощи	421

Часть III.

Высокий уровень Qt-программирования

Глава 18. Интернационализация	425
Работа с Unicode	426
Создание переводимого интерфейса приложения	430
Динамическое переключение языков	436
Перевод приложений	442
Глава 19. Настройка диалога с пользователем	446
Использование таблиц стилей Qt	447
Создание подклассов класса QStyle	461

Глава 20. Графика 3D	478
Рисование при помощи OpenGL	479
Комбинирование OpenGL и QPainter	484
Создание наложений с помощью объектов Framebuffer	491
Глава 21. Создание подключаемых модулей	498
Расширение Qt с помощью подключаемых модулей	499
Как обеспечить в приложении возможность подключения модулей	510
Написание подключаемых к приложению модулей	513
Глава 22. Создание прикладных скриптов	517
Общий обзор языка ECMAScript	518
Расширение приложений Qt при помощи скриптов	527
Реализация расширений графического интерфейса с помощью скриптов	531
Автоматическое выполнение задач с применением скриптов ...	539
Глава 23. Возможности, зависящие от платформы	551
Применение «родных» программных интерфейсов	552
Применение ActiveX в системе Windows	555
Управление сеансами в системе X11	568
Глава 24. Программирование встроенных систем	575
Первое знакомство с Qt/Embedded Linux	576
Настройка Qt/Embedded Linux	578
Интеграция приложений Qt при помощи Qtopia	579
Использование API Qtopia	585

Приложения

Приложение А. Получение и установка Qt	597
Замечание о лицензировании	597
Установка Qt/Windows	598
Установка Qt/Mac	598
Установка Qt/X11	599
Приложение Б. Создание приложений Qt	601
Применение qmake	602
Применение инструментов независимых разработчиков	607
Приложение В. Введение в Qt Jambi	612
Первое знакомство с Qt Jambi	613
Применение Qt Jambi в Eclipse IDE	618
Интеграции компонентов C++ в Qt Jambi	623
Приложение Г. Введение в C++ для программистов Java и C#	631
Первое знакомство с C++	632
Основные отличия языков	636
Стандартная библиотека C++	670

Примечание редактора серии

Уважаемый читатель,

как практикующий программист, я применяю Qt ежедневно, и на меня действительно производит впечатление та организация, дизайн и мощь, которые Qt предлагает программисту, пишущему на C++.

Хотя Qt начинался как межплатформенный инструментарий для создания графического пользовательского интерфейса, он расширился и включил в себя переносимые между платформами возможности, относящиеся почти ко всем аспектам повседневного программирования, например файлы, процессы, работу в сети и доступ к базам данных. Поскольку Qt можно применять очень широко, вы действительно можете написать код один раз и просто перекомпилировать его на другой платформе, после чего он сразу окажется работоспособным. Это чрезвычайно полезно, если вашим клиентам нужно, чтобы ваш продукт работал на разных платформах.

И конечно, поскольку Qt распространяется на основе лицензии открытого исходного кода, если вы ведете работу с открытым кодом, можете пользоваться всеми преимуществами, которые дает Qt. Хотя с Qt поставляется обширная интерактивная справка, она в основном представляет собой справочное пособие. Приведенные примеры программ полезны, однако только на основании этих примеров трудно воспроизвести правильное применение Qt для ваших программ. И именно здесь на сцену выходит эта книга.

Это, действительно, замечательная книга. Во-первых, это официальная книга по Qt от Trolltech, и этим многое сказано. Но это также отличная книга сама по себе: хорошо организованная, хорошо написанная, ее легко читать и учиться по ней. Сочетание прекрасной книги и прекрасной технологии обеспечивает реальный успех, и именно поэтому я очень рад и горд, что эта книга входит в серию *Prentice Hall Open Source Software Development Series*. Я надеюсь, что вам понравится чтение и вы много узнаете из этой книги. У меня это, определенно, получилось.

Арнольд Роббинс (Arnold Robbins)
Ноф Айалон, Израиль
Ноябрь 2007 года

Вступление

Почему Qt? Почему мы, программисты, выбираем Qt? Конечно, существуют очевидные ответы: совместимость классов Qt, базирующаяся на применении одного источника, богатство его возможностей, производительность C++, наличие исходного кода, его документация, качественная техническая поддержка и множество других причин, указанных в глянцевого маркетинговых материалах компании Trolltech. Все это очень хорошо, но здесь не указано самое важное: Qt пользуется успехом, потому что он нравится программистам.

Почему программистам нравится одна технология и не нравится другая? Сам я считаю, что разработчики программного обеспечения отдадут предпочтение такой технологии, которая «ощущается» как правильная, и не любят все то, что не дает такого ощущения. Как еще можно объяснить то, что некоторым самым лучшим программистам необходима помощь при программировании видеомagneфона, а у большинства инженеров возникают проблемы при работе с телефонной системой компании? Я, например, способен в совершенстве запоминать случайные последовательности чисел и команд, но если с их помощью нужно управлять телефонным автоответчиком, я бы предпочел не иметь такого автоответчика. В Trolltech наша телефонная система заставляет нас нажать клавишу «*», прежде чем можно будет ввести добавочный номер другого человека. Если забыть об этом и начать вводить добавочный номер сразу, вам придется набирать весь номер заново. Почему именно «*»? Почему не «#», «1», «5» или любая другая из 20 клавиш на телефоне? Зачем, вообще, это все? Оказалось, что меня телефон настолько раздражает, что я стараюсь по возможности им не пользоваться. Никому не нравится делать какие-то случайные вещи, особенно если эти случайные вещи зависят от какого-то столь же случайного контекста, о котором вы вообще ничего не хотели бы знать.

Программирование может быть во многом подобно использованию нашей телефонной системы, только еще хуже. И здесь на помощь приходит Qt. Qt не такой. Во-первых, Qt логичен. И, во-вторых, Qt вызывает интерес. Qt позволяет вам сконцентрироваться собственно на вашей задаче. Когда первоначальные создатели Qt сталкивались с проблемой, они не искали просто хорошее решение или самое простое решение. Они искали правильное решение и затем документировали его. Конечно, они делали ошибки и, конечно, их некоторые проектные решения не прошли проверку временем, но все же многое сделано правильно, а неправильное может и должно быть исправлено. Вы можете убедиться в этом на том факте, что система, первоначально задуманная как мостик между Windows 95 и Unix/Motif, теперь объединяет такие непохожие современные настольные системы, как Windows Vista, Mac OS X и GNU/Linux, а также малые устройства, такие как мобильные телефоны.

Задолго до того как инструмент Qt стал столь популярным и столь широко используемым, нацеленность разработчиков Qt на поиск правильных решений сделала Qt особым продуктом. Верность этому принципу столь же сильна сегодня, и она относится к каждому, кто сопровождает и разрабатывает Qt. Для нас работа над проектом Qt является одновременно и ответственным делом, и привилегией. Мы испытываем гордость оттого, что помогаем вам стать профессионалами и что работа с системами с открытым исходным кодом становится более простой и доставляет больше удовольствия.

Одной из вещей, которые делают использование Qt приятным, является интерактивная документация. Однако акцент в этой документации делается, в первую очередь на отдельных классах, и в ней очень мало говорится о том, как создавать сложные реальные приложения. Эта прекрасная книга заполняет упомянутый разрыв. В ней показывается, что может предложить Qt, как вести программирование на Qt и как получить максимум пользы от Qt. Данная книга научит программиста на C++, Java или C# программированию в Qt и предложит достаточно углубленный материал, чтобы удовлетворить и опытных программистов на Qt. Книга снабжена хорошими примерами, советами и разъяснениями – и именно этот текст мы используем для обучения всех новых программистов, приходящих в Trolltech.

В настоящее время можно приобрести и скачать огромное число коммерческих и бесплатных приложений Qt. Некоторые из них являются специализированными для конкретных вертикальных областей рынка, а другие предназначены для массового использования. Когда мы видим такое большое количество приложений, создаваемых с помощью Qt, это наполняет нас гордостью и воодушевляет на дальнейшее совершенствование Qt. А с помощью этой книги число более высококачественных приложений Qt вырастет более, чем когда-либо.

Маттиас Эттрич (Matthias Ettrich)
Берлин, Германия
Ноябрь 2007 года

Предисловие

Qt представляет собой комплексную среду разработки приложений для разработки на C++, предназначенную для создания межплатформенных приложений с графическим пользовательским интерфейсом по принципу «написал программу – компилируй ее в любом месте». Qt позволяет программистам использовать дерево классов с одним источником в приложениях, которые будут работать в системах от Windows 98 до Vista, Mac OS X, Linux, Solaris, HP-UX и во многих других версиях Unix с X11. Библиотеки и утилиты Qt входят также в состав Qt/Embedded Linux – программного продукта, обеспечивающего собственную оконную систему для встроенной системы Linux.

Цель этой книги – обучение вас способам написания программ с графическим пользовательским интерфейсом при помощи средств разработки Qt 4. Книга начинается с примера «Здравствуй, Qt» и быстро переходит к таким более сложным темам, как создание пользовательских виджетов и обеспечение технологии «drag-and-drop». Текст дополняется набором примеров, который можно получить с сайта данной книги, <http://www.informit.com/title/0132354160>. В Приложении А разъясняется, как скачать и установить программное обеспечение, включая бесплатный компилятор C++ для тех, кто работает в Windows.

Данная книга разделена на три части. В части I раскрыты все базовые принципы и даются практические советы, необходимые для программирования приложений с графическим интерфейсом при помощи средств разработки Qt. Знания материала этой части вполне достаточно для создания работоспособных приложений с графическим интерфейсом. В части II более глубоко рассматриваются основные темы Qt и в части III предоставляется более специализированный и углубленный материал. Главы частей II и III можно читать в любой последовательности, но они предполагают знакомство с содержанием части I. Книга также включает несколько приложений: в Приложении В показано, как создавать приложения Qt, а Приложении С вы познакомитесь с Qt Jambi, Java-версией Qt. Первое издание этой книги, посвященное Qt 4, создано на основе издания, посвященного Qt 3, которое было полностью переработано для демонстрации принципов хорошего программирования с применением средств разработки Qt 4. Во многих случаях здесь используются примеры, аналогичные примерам в издании для Qt 3. Это издание содержит новые главы, в которых описывается архитектура Qt 4 модель/представление, новый фреймворк для подключаемых модулей и основы программирования встроенных систем с помощью Qt/Embedded Linux, а также новое приложение.

В данное дополненное и исправленное второе издание были внесены изменения, связанные с использованием возможностей, появившихся в Qt версий 4.2 и 4.3, добавлены новые главы, посвященные настройке диалога с пользователем и созданию прикладных скриптов, а также два новых приложения. Первоначальная глава, посвященная графике, разделена на две главы, посвященные двумерной и трехмерной графике, а между ними теперь рассматриваются новые классы графических представлений и серверные системы QPainter и OpenGL. Кроме того, добавлено много нового материала по базам данных, XML и программированию встроенных систем. В этом издании, как и в предшествующих, акцент делается на объяснении программирования в Qt и рассмотрении реалистичных примеров, а не просто на изложении другими словами и обобщении обширной интерактивной документации Qt. Предполагается, что вы знакомы с основами программирования на C++, Java или C#. Программный код примеров использует подмножество C++, избегая многие его возможности, ко-

торые редко требуются при Qt-программировании. В нескольких местах, где нельзя обойтись без специальных конструкций C++, дается подробное объяснение их применения.

Если у вас уже есть опыт программирования на Java или C#, но мало или совсем нет опыта программирования на C++, мы рекомендуем начать с Приложения D к книге, содержащего введение в C++, вполне достаточное для того, чтобы можно было использовать эту книгу. В качестве более полного введения в объектно-ориентированное программирование на C++ мы рекомендуем книгу «C++ Howto Program» (Как программировать на C++), авторы P.J. Deitel, H.M. Deitel, и «C++ Primer» (Язык программирования C++. Вводный курс), авторы Stanley B. Lippman, Josée Lajoie и Barbara E. Moo.

Qt создал себе репутацию средства разработки межплатформенных приложений, но благодаря своему интуитивному и мощному программному интерфейсу во многих организациях Qt используется для одноплатформенных разработок. Пакет программ «Adobe Photoshop Album» – один из примеров продукта на массовом рынке Windows, написанного средствами Qt. Многие сложные системы программного обеспечения на таких вертикальных рынках, как средства анимации 3D, цифровая обработка фильмов, автоматизация проектирования электронных схем (для проектирования чипов), разведка нефтяных и газовых месторождений, финансовые услуги и формирование изображений в медицине, строятся при помощи Qt. Если свои средства к существованию вы получаете благодаря успешному программному продукту для Windows, который создан при помощи Qt, вы можете легко создать новые рынки для систем Mac OS X и Linux просто путем перекомпиляции программного продукта.

Qt может применяться с различными лицензиями. Если вы собираетесь создавать коммерческие приложения, вы должны приобрести коммерческую лицензию Qt у Trolltech; если вы собираетесь создавать программы с открытым исходным кодом, вы можете использовать версию с открытым исходным кодом (с лицензией GPL). К Desktop Environment (KDE) и большинство других приложений с открытым исходным кодом, которые поставляются с этой средой, созданы на основе Qt.

Кроме сотен классов Qt существуют дополнения, расширяющие рамки и возможности Qt. Некоторые из этих программных продуктов поставляются компанией Trolltech, например компоненты Qt Solutions, в то время как другие подобные программные продукты поставляются другими компаниями и сообществом по разработке приложений с открытым исходным кодом. Обращайтесь на страницу <http://www.trolltech.com/products/qt/3rdparty/>, где можно найти список дополнений Qt. Разработчики Trolltech имеют свой собственный web-сайт, Trolltech Labs (<http://labs.trolltech.com/>), где выкладывается написанный ими неофициальный код, если он является красивым, интересным или полезным. Qt также имеет хорошо зарекомендовавшее и преуспевающее сообщество пользователей, которое использует список почтовой рассылки [qt-interest](http://lists.trolltech.com/qt-interest); подробности вы найдете по адресу <http://lists.trolltech.com/>.

Если вы обнаружили в книге ошибки, имеете предложения для следующего издания или хотите высказать свое впечатление, мы будем рады все это услышать от вас. Вы можете связаться с нами по электронной почте по адресу qt-book@trolltech.com. Ошибки будут размещены в сети Интернет на сайте данной книги <http://www.prenhallprofessional.com/title/0132354160>.

Благодарности

Прежде всего, мы хотим выразить свою благодарность Айрику Чеймб-Ингу (Eirik Chambe-Eng), главному троллю Trolltech и одному из двух основателей компании, который не только с энтузиазмом вдохновлял нас на написание версии этой книги для Qt 3, он также позволил нам потратить много нашего рабочего времени на ее написание. Айрик и исполнительный директор компании Trolltech Хаавард Норд (Haavard Nord) прочитали рукопись и сделали ценные замечания. Их щедрость и предвидение дополнялись и поощрялись Маттиасом Эттричем (Matthias Ettrich), который снисходительно относился к игнорированию нами наших обязанностей, когда мы были полностью вовлечены в процесс написания этой книги, и дал нам множество советов по формированию хорошего стиля Qt-программирования.

Для первого издания мы попросили двух наших заказчиков, Пола Куртиса (Paul Curtis) и Клауса Шмидингера (Klaus Schmidinger), стать нашими внешними рецензентами. Оба являются экспертами по Qt-программированию и обращают особое внимание на технические детали, что позволило им найти некоторые очень тонкие ошибки в нашей рукописи и предложить нам много улучшений. В компании Trolltech кроме Маттиаса нашим самым решительным рецензентом был Реджинальд Стадлбауер (Reginald Stadlbauer). Его глубокое понимание технических делателей было бесценно, и он научил нас некоторым вещам, которые казались нам невозможными в Qt.

При подготовке издания Qt 4 мы по-прежнему получали большую помощь и поддержку от Айрика, Хааварда и Маттиаса. Клаус Шмидингер продолжал нам давать свои ценные советы, тщательное рецензирование части нового материала производил наш клиент по Qt Пол Флойд (Paul Floyd). Благодарим также Дэвида Гарсию Гарзона (David Garcia Garzon) за помощь со SCons в Приложении В. Нашими важными рецензентами из компании Trolltech были Карлос Мануэль Даклос Вергара (Carlos Manuel Duclos Vergara), Эндриас Аардал Хансен (Andreas Aardal Hanssen), Хенрик Харц (Henrik Hartz), Мартин Джонс (Martin Jones), Виви Глукстад Карлсен (Vivi Glückstad Karlsen), Тронд Кьерносен (Trond Kjernosen), Трентон Шультц (Trenton Schulz), Энди Шоу (Andy Shaw), Гуннар Слетта (Gunnar Sletta) и Пал де Вибе (Pel de Vibe).

Кроме упомянутых выше рецензентов мы получали экспертную помощь от Эскила Абрахамсена Бломфельдта (Escil Abrahamsen Blomfeldt) (Qt Jambi), Франса Инглича (Frans English) (XML), Харальда Фернengelа (Harald Fernengel) (базы данных), Кента Хансена (Kent Hansen) (прикладные скрипты), Волкера Хилшаймера (Volker Hilsheimer) (ActiveX), Бредли Хьюза (Bradley Hughes) (многопоточная обработка), Ларса Кнолла (Lars Knoll) (графика 2D и интернационализация), Андерса Ларсена (Anders Larsen) (базы данных), Сэма Магнусона (Sam Magnuson) (qmake), Мариус Бугге Монсен (Marius Bugge Monsen) (классы отображения элементов), Димитри Пападопулоса (Dimitri Papadopoulos) (Qt/X11), Гириша Рамакришны (Girish Ramakrishnan) (таблицы стилей), Самуэля Редала (Samuel Redal) (3D-графика), Рейнера Шмида (Rainer Schmid) (работа с сетью и XML), Амрит Пал Синх (Amrit Pal Singh) (введение в C++), Пола Олава Твета (Paul Olav Tvete) (поль-

зовательские виджеты и программирование встроенных систем), Гейра Ваттекара (Geir Vattekar) (Qt Jambi) и Томаса Зандера (Thomas Zander) (создание систем).

Дополнительную благодарность мы выражаем группам подготовки документации и технической поддержки компании Trolltech за помощь в решении вопросов, связанных с подготовкой документации, пока книга отнимала у нас столь много времени, и системным администраторам компании Trolltech за обеспечение рабочего состояния наших машин и наших сетевых соединений на протяжении всего проекта.

Что касается производственной части, то Джефф Кингстон (Jeff Kingston), автор наборной программы Lout, продолжал вносить усовершенствования в свой инструмент, и многие из них стали ответом на наши предложения. Также благодарим Джеймса Клуза (James Cloos) за создание сокращенной версии шрифта DejaVu Mono, который мы использовали как основу для нашего моноширинного шрифта. Катрин Бор (Cathrine Bore) из Trolltech вела для нас контракты и обеспечивала юридические вопросы. Мы также благодарны Натану Клементу (Nathan Clement) за иллюстрации с троллями и Одри Дойл (Audrey Doyle) за тщательную корректуру. И наконец, мы выражаем благодарность нашему редактору Дебре Уильямс-Коли (Debra Williams-Cauley) за поддержку и обеспечение максимально беспроblemной работы, а также Ларе Уисонг (Lara Wysong) за очень хорошее управление процессом производства.

Краткая история Qt

Средства разработки Qt впервые стали известны общественности в мае 1995 года. Первоначально Qt разрабатывались Хаавардом Нордом (исполнительным директором компании Trolltech) и Айриком Чеймб-Ингом (главным троллем компании Trolltech). Хаавард и Айрик познакомились в Норвежском институте технологии, г. Тронхейм, который они окончили, получив степень магистра по теории вычислительных систем и машин.

Хаавард стал проявлять интерес к разработке графического пользовательского интерфейса на C++, когда он был привлечен шведской компанией к разработке инструментального средства, предназначенного для разработки графического интерфейса на C++. Спустя два года (летом 1990 г.), Хаавард и Айрик работали вместе над разработкой на C++ приложения для баз данных ультразвуковых изображений. Эта система должна была предоставлять графический пользовательский интерфейс в системах Unix, Macintosh и Windows. Однажды этим же летом Хаавард и Айрик вышли на улицу, чтобы понежиться на солнышке, и когда они присели на скамейку в парке, Хаавард сказал: «Нам нужна объектно-ориентированная система отображения». Последующая дискуссия стала интеллектуальной основой объектно-ориентированной межплатформенной системы разработки графического пользовательского интерфейса, к созданию которой они вскоре приступили.

В 1991 году Хаавард начал писать классы, которые фактически образовали Qt, причем проектные решения принимались совместно с Айриком. В следующем году Айрику пришла идея «сигналов и слотов» – простой, но мощной парадигмы программирования графического пользовательского интерфейса, которая в настоящее время заимствована некоторыми другими инструментальными средствами. Хаавард воспринял эту идею и вручную реализовал ее. К 1993 году Хаавард и Айрик разработали первое графическое ядро Qt и могли создавать свои собственные виджеты. В конце этого же года Хаавард предложил совместно заняться бизнесом и построить «самые лучшие в мире инструментальные средства разработки на C++ графического пользовательского интерфейса».

Начало 1994 года не предвещало ничего хорошего, когда два молодых программиста собирались выйти на установившийся рынок, не имея ни заказчиков, ни законченного программного продукта, ни денег. К счастью, жены обоих имели работу и могли поддержать своих мужей в течение двух лет, которых, как считали Айрик и Хаавард, будет достаточно для разработки программного продукта, позволяющего начать зарабатывать деньги.

Буква «Q» была выбрана в качестве префикса классов, поскольку эта буква имела красивое начертание в шрифте редактора Emacs, которым пользовался Хаавард. Была добавлена буква «t», означающая «toolkit» (инструментарий), что похоже на «Xt», то есть X Toolkit. Компания была зарегистрирована 4 марта 1994 года и первоначально называлась Quasar Technologies, затем Troll Tech, и теперь она называется Trolltech.

В апреле 1995 года через посредничество одного университетского профессора, знакомого Хааварда, норвежская компания Metis заключила с ними контракт на разработку программного обеспечения на основе Qt. Примерно в это же

время Trolltech приняла на работу Арнта Гулдбрансена (Arnt Gulbrandsen), который в течение своих шести лет пребывания в этой компании продумал и реализовал оригинальную систему документирования, а также внес определенный вклад в программный код Qt.

20 мая 1995 года Qt 0.90 был установлен на сайте sunsite.unc.edu. Спустя шесть дней о выпуске этой версии было объявлено на comp.os.linux.announce. Это была первая публичная версия Qt. Qt можно было использовать в разработках как Windows, так и Unix, причем программный интерфейс был одинаковый на обеих платформах. С первого дня предусматривались две лицензии применения Qt: коммерческая лицензия предназначалась для коммерческих разработок, а свободно распространяемая версия предназначалась для разработок с открытым исходным кодом. Контракт с Metis сохранил компанию Trolltech на плаву, хотя в течение долгих десяти месяцев не было продано ни одной коммерческой лицензии Qt.

В марте 1996 года Европейское управление космических исследований (European Space Agency) стало вторым заказчиком Qt, оно приобрело десять коммерческих лицензий. Верящие в удачу Айрик и Хаавард приняли на работу еще одного разработчика. Qt 0.97 был выпущен в конце мая, и 24 сентября 1996 года вышла версия Qt 1.0. К концу этого же года вышла версия Qt 1.1; восемь заказчиков – все из разных стран – приобрели в общей сложности 18 лицензий. В этом году был также основан Маттиасом Эттричем проект KDE.

Версия Qt 1.2 была выпущена в апреле 1997 года. Принятое Маттиасом Эттричем решение по применению Qt для построения KDE помогло Qt стать фактическим стандартом по разработке на C++ графического пользовательского интерфейса в системе Linux. Qt 1.3 был выпущен в сентябре 1997 года.

Маттиас присоединился к Trolltech в 1998 году, и последняя значимая версия Qt первого выпуска, 1.40, появилась в сентябре того же года. Qt 2.0 был выпущен в июне 1999 года. Qt 2 имел новую лицензию для открытого исходного кода, Q Public License (QPL), которая соответствовала Определению открытого исходного кода (Open Source Definition). В августе 1999 года Qt выиграл премию журнала Linux World за лучшую библиотеку или инструментальное средство. Примерно в это же время была образована компания Trolltech Pty Ltd (Австралия).

Компания Trolltech выпустила Qt/Embedded Linux в 2000 году. Она спроектирована для работы на устройствах с системой Embedded Linux и обеспечивает свою собственную оконную систему в качестве упрощенной замены X11. Как Qt/X11, так и Qt/Embedded Linux предлагаются теперь по широко распространенной общедоступной лицензии GNU General Public License (GPL), а также на условиях коммерческих лицензий. К концу 2000 года Trolltech учредила компанию Trolltech Inc. (США) и выпустила первую версию Qtopia – платформу для разработки приложений для мобильных телефонов и карманных компьютеров. Qt/Embedded Linux был удостоен премии журнала LinuxWorld в категории «Лучшее решение для системы Embedded Linux» в 2001 и 2002 годах, а Qtopia Phone получил ту же премию в 2004 году.

Qt 3.0 был выпущен в 2001 году. Qt теперь работал в системах Windows, Mac OS X, Unix и Linux (для настольных и встроенных систем). Qt 3 содержал 42 новых класса, и объем его программного кода превышал 500 000 строк. Qt 3

представлял собой важный шаг вперед по сравнению с Qt 2, который, в частности, значительно улучшил поддержку локализации и кодировки Unicode, ввел совершенно новые виджеты по просмотру и редактированию текста и класс регулярных выражений, аналогичных применяемому языком Perl. Qt 3 был удостоен премии «Software Development Times» в категории «Высокая продуктивность» в 2002 году.

Летом 2005 года был выпущен Qt 4.0. Имея около 500 классов и более 9000 функций, Qt 4 оказался больше и богаче любой предыдущей версии; он был разбит на несколько библиотек, чтобы разработчики могли использовать только нужные им части Qt. Версия Qt 4 представляет собой большой шаг вперед по сравнению с предыдущими версиями; она содержит полностью новый набор эффективных и простых в применении классов-контейнеров, усовершенствованную функциональность архитектуры модель/представления, быстрый и гибкий фреймворк графики 2D и мощные классы для просмотра и редактирования текста в кодировке Unicode, не говоря уже о тысячах небольших улучшений по всему спектру классов Qt. Набор возможностей Qt 4 теперь настолько широк, что Qt перерос рамки инструментария для графического пользовательского интерфейса и превратился в полнофункциональную среду для разработки приложений. Qt 4 также является первой версией Qt, доступной на всех поддерживаемых платформах как для коммерческой разработки, так и для разработки с открытым исходным кодом.

Кроме того, в 2005 году компания Trolltech открыла свое представительство в Пекине для предоставления пользователям в Китае и во всем этом регионе услуг по продаже, обучению и технической поддержке Qt/Embedded Linux и Qtopia.

Qt давно доступен для программистов не на C++, поскольку имеются неофициальные связи языков, в частности PyQt для программистов Python. В 2007 году были выпущены неофициальные связи Qyoto для программистов C#. Также в 2007 году Trolltech выпустила Qt Jambi – официально поддерживаемую Java-версию API Qt. В Приложении С приводится введение в Qt Jambi.

Со дня образования компании Trolltech популярность Qt постоянно росла, и она продолжает расти в наши дни. Этот успех является отражением как качества Qt, так и того удовольствия, которое разработчик получает при ее использовании. За последнее десятилетие Qt превратился из «секретного» программного продукта, известного только избранной группе профессионалов, в продукт, которым пользуются по всему миру тысячи коммерческих заказчиков и десятки тысяч разработчиков приложений с открытым исходным кодом.

Часть I

Основные возможности средств разработки Qt



- *«Здравствуй, Qt»*
- *Взаимодействие с пользователем*
- *Компоновка виджетов*
- *Использование справочной документации*

Глава 1. Первое знакомство

В данной главе показано на примере создания простого приложения с графическим интерфейсом пользователя (GUI – graphical user interface), как можно обычные средства C++ совместить с функциональными возможностями Qt. Здесь также рассматриваются две ключевые идеи Qt: «сигналы и слоты» (signals and slots) и компоновка графических элементов (layout). В главе 2 мы рассмотрим более подробно возможности Qt, а в главе 3 начнем разрабатывать более реалистичное приложение.

Если вы уже знакомы с Java или C#, но имеете лишь ограниченный опыт работы с C++, возможно, вы захотите начать с Приложения Г, в котором дается введение в C++.

«Здравствуй, Qt»

Давайте начнем с очень простой Qt-программы. Сначала мы разберем каждую строку этой программы, а затем покажем способы ее компиляции и выполнения.

```
001 #include <QApplication>
002 #include <QLabel>
003 int main(int argc, char *argv[])
004 {
005     QApplication app(argc, argv);
006     QLabel *label = new QLabel("Hello Qt!");
007     label->show();
008     return app.exec();
009 }
```

В строках 1 и 2 в программу включаются определения классов `QApplication` и `QLabel`. Для каждого Qt-класса имеется заголовочный файл с тем же именем (с учетом регистра), содержащий определение этого класса.

В строке 5 создается объект `QApplication` для управления всеми ресурсами приложения. Для конструктора `QApplication` необходимо указывать параметры `argc` и `argv`, поскольку Qt сама обрабатывает некоторые из аргументов командной строки.

В строке 7 создается «виджет» текстовая метка `QLabel`, который выводит на экран сообщение «Hello Qt!» (здравствуй, Qt). По терминологии Qt и Unix *вид-*

жетом (widget) называется любой визуальный элемент графического интерфейса пользователя. Этот термин происходит от «window gadget» и соответствует элементу управления («control») и контейнеру («container») по терминологии Windows. Кнопки, меню, полосы прокрутки и фреймы являются примерами виджетов. Одни виджеты могут содержать в себе другие виджеты. Например, окно приложения обычно является виджетом, содержащим QMenuBar (панель меню), несколько QToolBar (панель инструментов), QStatusBar (строка состояния) и некоторые другие виджеты. Большинство приложений используют QMainWindow или QDialog в качестве окна приложения, однако Qt настолько гибок, что любой виджет может быть окном. В данном примере QLabel является окном приложения.

Строка 7 делает текстовую метку видимой. Виджеты всегда создаются сначала невидимыми, и поэтому до непосредственного вывода на экран вы можете настроить их и тем самым не допустить мерцание экрана.

Строка 8 обеспечивает передачу управления приложением Qt. В этом месте программа переходит в цикл обработки событий, то есть в своего рода режим «простоя», ожидая со стороны пользователя таких действий, как щелчок мыши или нажатие клавиши на клавиатуре.

Для простоты мы не делаем вызов оператора delete для объекта QLabel в конце функции main(). Подобная утечка памяти в такой небольшой программе безвредна, поскольку после завершения программы эта память будет возвращена операционной системой.

Теперь существует возможность опробовать программу на вашей собственной машине. Выглядит это примерно так, как показано на рис. 1.1. Во-первых, вам нужно установить Qt 4.3.2 (или более позднюю версию Qt). Этот процесс описан в Приложении А. Далее мы будем предполагать, что у вас правильно установлена копия Qt 4, а в вашей переменной окружения PATH прописан путь к директории двоичных файлов (bin) Qt. (В Windows это делается автоматически программой установки Qt.) Вам также потребуется код программы в файле, называемомся hello.cpp в директории hello. Вы можете создать файл hello.cpp самостоятельно или скопировать его из примеров, входящих в эту книгу. Путь выглядит так: examples/chap01/hello/hello.cpp. (Все примеры вы найдете на прилагающемся к книге CD или на web-сайте книги <http://www.informit.com/title/0132354160>.)



Рис. 1.1. Вывод приветствия программы Hello в системе Linux

Находясь в консольном режиме, войдите в каталог hello и задайте команду `qmake -project` для создания файла проекта, независимого от платформы (hello.pro), и затем задайте команду `qmake hello.pro` для создания на основе файла проекта зависящего от платформы файла makefile. (Инструмент qmake рассматривается более подробно в Приложении Б.)

Выполните команду `make` для построения программы¹. Затем выполните программу, задавая команду `hello` в системе Windows, или `./hello` в системе Unix, или `open hello.app` в системе Mac OS X. Для завершения программы нажмите кнопку закрытия окна, расположенную в заголовке окна.

Если вы используете Windows и установили версию Qt с открытым исходным кодом вместе с компилятором MinGW, вы получите ярлык для окна MS-DOS, в котором переменные среды правильно настроены на Qt. Вызвав это окно, вы можете компилировать в нем Qt-приложения, используя описанные выше команды `qmake` и `make`. Формируемые исполняемые модули помещаются в папку `debug` или `release` (например, `C:\qt-book\hello\release\hello.exe`).

Если вы используете Visual C++ компании Microsoft в сочетании с коммерческой версией Qt, то вам потребуется выполнить команду `nmake`, а не `make`. Здесь вы можете поступить по-другому и создать проект в Visual Studio на основе файла `hello.pro`, выполняя команду

```
qmake -tp vc hello.pro
```

и затем выполнить построение программы в системе Visual Studio. Если вы используете Xcode на Mac OS X, то можете сгенерировать проект Xcode с помощью следующей команды:

```
qmake -spec macx-xcode hello.pro
```

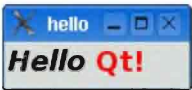


Рис. 1.2. Текстовая метка с простым форматированием HTML

Прежде чем перейти к следующему примеру, позволим себе небольшое развлечение, а именно заменим строку

```
QLabel *label = new QLabel("Hello Qt!");
```

на строку

```
QLabel *label = new QLabel("<h2><i>Hello</i> "
                           "<font color=red>Qt!</font></h2>");
```

и снова выполним построение приложения. При запуске окно будет выглядеть, как показано на рис. 1.2. Как иллюстрирует этот пример, совсем не трудно выделять элементы пользовательского интерфейса Qt-приложения с использованием некоторых простых средств форматирования документов HTML.

Взаимодействие с пользователем

Второй пример показывает возможности взаимодействия пользователя с программой. Приложение представляет собой кнопку, которую пользователь может нажать, и тогда приложение закончит свою работу. Исходный код этой программы очень напоминает исходный код программы Hello, но здесь вместо `QLabel` используется `QPushButton` в качестве главного виджета и добавляется код, обеспечивающий реакцию программы на действие пользователя (нажатие кнопки).

¹ Если вы получаете ошибку при компиляции оператора `#include <QApplication>`, возможно, это происходит из-за применения старой версии Qt. Убедитесь, что вы используете Qt 4.1.1 или более старшую версию Qt 4.

Исходный код этого приложения находится в составе примеров, прилагающихся к этой книге, в файле `examples/chap01/quit/quit.cpp`. Ниже приводится содержимое этого файла.

```
1 #include <QApplication>
2 #include <QPushButton.h>
3 int main(int argc, char *argv[])
4 {
5     QApplication app(argc, argv);
6     QPushButton *button = new QPushButton("Quit");
7     QObject::connect(button, SIGNAL(clicked()),
8                      &app, SLOT(quit()));
9     button->show();
10    return app.exec();
11 }
```

Виджеты Qt генерируют сигналы¹ в ответ на выполнение пользователем какого-то действия или изменение состояния. Например, `QPushButton` генерируют сигнал `clicked()` при нажатии пользователем кнопки. Сигнал может быть связан с функцией (называемой слотом в данном контексте) для автоматического ее выполнения при получении данного сигнала. В нашем примере мы связываем сигнал кнопки `clicked()` со слотом `quit()` объекта приложения `QApplication`. Макросы `SIGNAL()` и `SLOT()` являются частью синтаксиса.

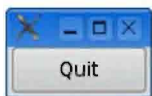


Рис. 1.3. Приложение Quit (завершить работу)

Теперь мы построим приложение. Мы предполагаем, что вами создан каталог `quit` и в нем находится файл `quit.cpp`. Выполните команду `qmake` из каталога `quit` для формирования файла проекта и затем используйте полученный файл для создания файла `makefile`:

```
qmake -project
qmake quit.pro
```

Теперь постройте приложение и запустите его на выполнение. Если вы нажмете кнопку `quit` или клавишу пробела на клавиатуре (она также приводит к нажатию этой кнопки), приложение завершит свою работу.

Компоновка виджетов

В данном разделе мы создадим небольшое приложение, которое демонстрирует применение менеджеров компоновки для размещения виджетов в окне и использование сигналов и слотов для синхронизации работы двух виджетов. Приложение (показанное на рис. 1.4) предлагает пользователю указать свой возраст, что можно сделать при помощи либо наборного счетчика (*spin box*), либо ползунка (*slider*).

¹ Сигналы Qt не надо путать с сигналами системы Unix. В данной книге нами рассматриваются только сигналы Qt.

Это приложение состоит из трех виджетов: `QSpinBox`, `QSlider` и `QWidget`. `QWidget` является главным окном приложения. Виджеты `QSpinBox` и `QSlider` помещены внутрь `QWidget`, и они являются дочерними виджетами по отношению к `QWidget`. С другой стороны, мы можем сказать, что `QWidget` является родительским виджетом по отношению к `QSpinBox` и `QSlider`. Сам `QWidget` не имеет родителя, потому что используется в качестве окна самого верхнего уровня. Конструкторы `QWidget` и все его подклассы принимают параметр `QWidget *`, задающий родительский виджет.



Рис. 1.4. Приложение Age (возраст)

Ниже приводится исходный код.

```

1  #include <QApplication>
2  #include <QHBoxLayout>
3  #include <QSlider>
4  #include <QSpinBox>
5  int main(int argc, char *argv[])
6  {
7      QApplication app(argc, argv);
8      QWidget *window = new QWidget;
9      window->setWindowTitle("Enter Your Age");
10     QSpinBox *spinBox = new QSpinBox;
11     QSlider *slider = new QSlider(Qt::Horizontal);
12     spinBox->setRange(0, 130);
13     slider->setRange(0, 130);
14     QObject::connect(spinBox, SIGNAL(valueChanged(int)),
15                     slider, SLOT(setValue(int)));
16     QObject::connect(slider, SIGNAL(valueChanged(int)),
17                     spinBox, SLOT(setValue(int)));
18     spinBox->setValue(35);
19     QHBoxLayout *layout = new QHBoxLayout;
20     layout->addWidget(spinBox);
21     layout->addWidget(slider);
22     window->setLayout(layout);
23     window->show();
24     return app.exec();
25 }
```

Строки 8 и 9 создают и настраивают виджет `QWidget`, который является главным окном приложения. Нами вызывается функция `setWindowTitle()` для вывода текстовой строки в заголовке окна.

Затем мы устанавливаем промежуток (в 6 пикселей) между дочерними виджетами и вокруг них.

Строки 10 и 11 создают виджеты `QSpinBox` и `QSlider`, а строки 12 и 13 устанавливают допустимый диапазон изменения их значений. Мы вполне можем допустить, что возраст человека не будет превышать 130 лет. Мы могли бы передать `window` в конструкторах `QSpinBox` и `QSlider`, указывая на то, что `window` должен быть их ро-

дительским виджетом, но здесь это делать необязательно, поскольку система компоновки определит это самостоятельно и автоматически установит родительский виджет для наборного счетчика и ползунка, как мы это увидим вскоре.

Два вызова функции `QObject::connect()`, выполненные в строках с 14-й по 17-ю, обеспечивают синхронизацию работы наборного счетчика и ползунка, заставляя их всегда показывать одинаковое значение. Если один из виджетов изменяет значение, то генерируется сигнал `valueChanged(int)` и вызывается слот `setValue(int)` другого виджета с новым значением возраста.

В строке 18 наборный счетчик устанавливается в значение 35. В результате виджет `QSpinBox` генерирует сигнал `valueChanged(int)` с целочисленным аргументом 35. Этот аргумент передается слоту `setValue(int)` виджета `QSlider`, и в результате ползунков устанавливается в значение 35. Ползунок затем также генерирует сигнал `valueChanged(int)`, поскольку его значение изменилось, и вызывает слот `setValue(int)` наборного счетчика. Но на этот раз функция `setValue(int)` не будет генерировать сигнал, поскольку наборный счетчик уже имеет значение 35. Это не позволяет повторять эти действия бесконечно. Описанная ситуация продемонстрирована на рис. 1.5.

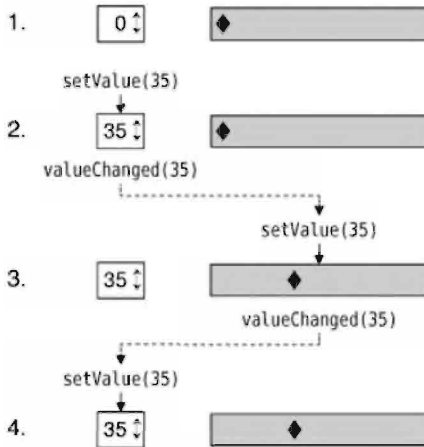


Рис. 1.5. Изменение значения в одном из виджетов приводит к изменению значения в другом виджете

В строках с 19-й по 22-ю мы размещаем виджеты наборного счетчика и ползунка, используя менеджер компоновки. Менеджер компоновки – это объект, который устанавливает размер и положение виджетов, располагающихся в зоне его действия. Qt имеет три основных класса менеджеров компоновки:

- `QHBoxLayout` размещает виджеты по горизонтали слева направо (или справа налево, в зависимости от культурных традиций);
- `QVBoxLayout` размещает виджеты по вертикали сверху вниз;
- `QGridLayout` размещает виджеты в ячейках сетки.

Выполненный в строке 22 вызов `QWidget::setLayout()` устанавливает менеджер компоновки для окна. За кулисами создаются дочерние связи `QSpinBox` и `QSlider` с виджетом, для которого установлен менеджер компоновки, и по этой причине нам не требуется в явной форме задавать родительский виджет при конструировании виджета, размещаемого в зоне действия менеджера компоновки.

Стили виджетов

Снимки экрана, которые мы делали до сих пор, мы получали в Linux, однако приложения Qt естественно выглядят на любой поддерживаемой платформе. В Qt это достигается путем эмуляции изобразительных средств данной платформы, а не путем создания «оболочки» для конкретной платформы или путем набора виджетов из инструментария.

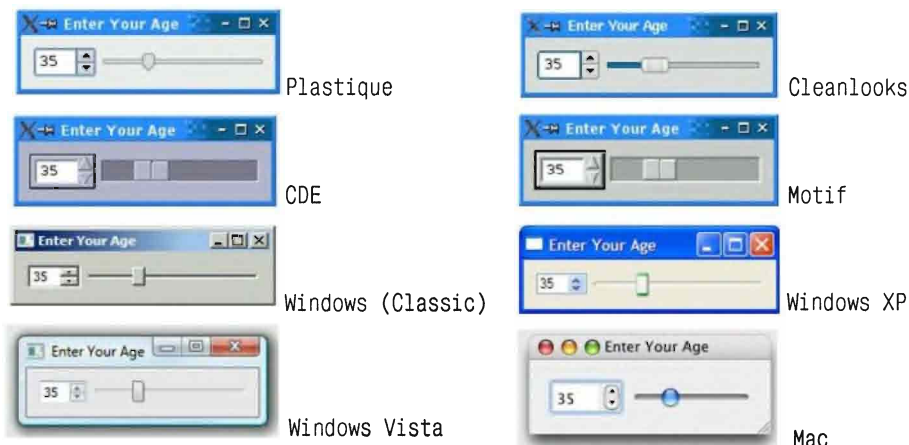


Рис. 1.6. Готовые стили

Стиль Plastique – это заданный по умолчанию стиль для приложений Qt/X11, работающих под KDE, а Cleanlooks – это стиль по умолчанию для GNOME.

В этих стилях используются градиенты и сглаживание, что обеспечивает стилю современный вид. Пользователи приложения Qt могут изменить заданный по умолчанию стиль при помощи опции командной строки `-style`. Например, чтобы запустить приложение Age в X11 с использованием стиля Motif, просто введите следующую команду:

```
./age -style motif
```

В отличие от других стилей, стили WindowsXP, Windows Vista и Mac доступны только на своих «родных» платформах, поскольку они используют систему тем платформы.

Существует также дополнительный стиль, QtDotNet, от Qt Solutions. Кроме того, можно создавать собственные стили, как это описано в главе 19.

Несмотря на то что мы не задавали в явной форме положение и размер ни одного из виджетов, QSpinBox и QSlider аккуратно расположились в ряд. Это объясняется тем, что QHBoxLayout автоматически определяет разумные размеры и положение виджетов, попадающих в зону его действия, в зависимости от потребностей этих виджетов. Менеджеры компоновки освобождают нас от нудного кодирования размещения виджетов нашего приложения на экране и гарантируют плавное изменение размеров окон.

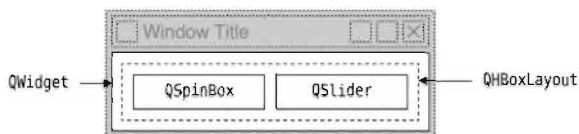


Рис. 1.7. Виджеты приложения Age и компоновка

Используемый средствами разработки Qt подход к построению графического пользовательского интерфейса легко понятен и очень гибок. Среди работающих в Qt программистов наиболее распространен подход, когда сначала создаются все необходимые графические элементы и затем соответствующим образом настраиваются их свойства. Программисты добавляют виджеты к компоновщикам графических элементов, которые автоматически устанавливают для них нужный размер и положение. Управление работой графического интерфейса осуществляется через взаимодействие виджетов друг с другом посредством применения механизма сигналов и слотов Qt.

Использование справочной документации

Справочная документация по средствам разработки Qt является важным инструментом в руках любого разработчика Qt-программ, поскольку в ней есть все необходимые сведения по любому классу и любой функции Qt. В данной книге используются многие из классов и функций Qt, но далеко не все, и они описываются не во всей полноте. Для более эффективного использования Qt вам необходимо хорошо разбираться в ее справочной документации и следует сделать это как можно скорее.

Эта документация имеется в формате HTML (каталог doc/html в системе Qt) и ее можно просматривать любым веб-браузером. Вы можете также использовать программу *Qt Assistant* (помощник Qt) – браузер системы помощи в Qt, который обладает мощными средствами поиска и индексирования информации, поэтому он быстрее находит нужную информацию и им легче пользоваться, чем веб-браузером.

Для запуска *Qt Assistant* необходимо выбрать функцию Qt by Trolltech v4.x.y | Assistant в меню Start (пуск) системы Windows, задать команду assistant в системе Unix или дважды щелкнуть по Assistant в системе Mac OS X Finder. Ссылки в разделе «API Reference» (ссылки программного интерфейса) домашней страницы обеспечивают различные пути навигации по классам Qt. На странице «All Classes» (все классы) приводится список всех классов программного интерфейса Qt. На странице «Main Classes» (основные классы) перечисляются только наиболее используемые классы Qt. Например, вы можете просмотреть классы и функции, использованные нами в этой главе.

Следует отметить, что описание наследуемых функций приводится в базовом классе; например, класс QPushButton не имеет описания функции show(), но это описание имеется в QWidget. На рис. 1.9 показана взаимосвязь классов, которые использовались в этой главе.

Справочную документацию для текущей версии Qt и нескольких более старых версий можно найти в сети Интернет по адресу <http://doc.trolltech.com/>. На этом сайте также находятся избранные статьи из журнала *Qt Quarterly* (ежеквартальное обозрение по средствам разработки Qt); этот журнал предназначен для программистов Qt и распространяется по всем коммерческим лицензиям.

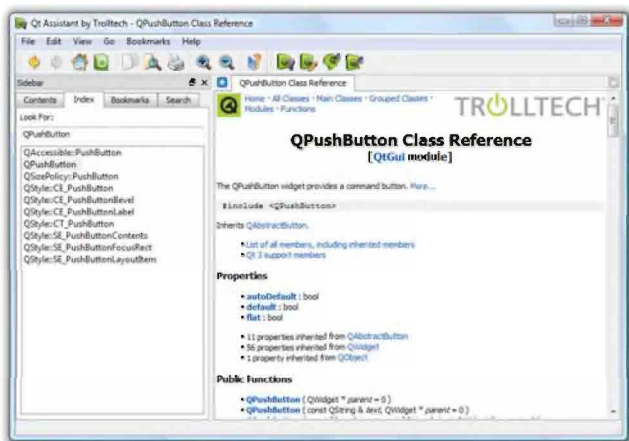


Рис. 1.8. Просмотр документации Qt программой *Qt Assistant* в системе Windows Vista

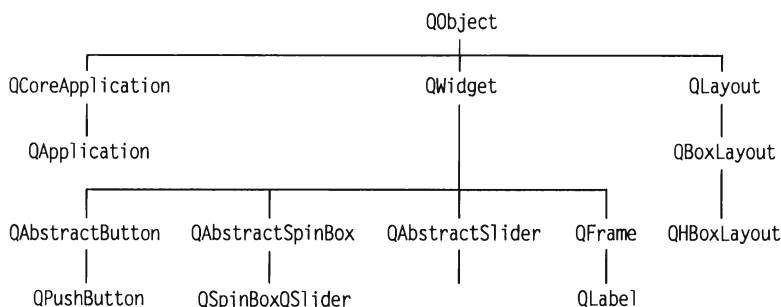
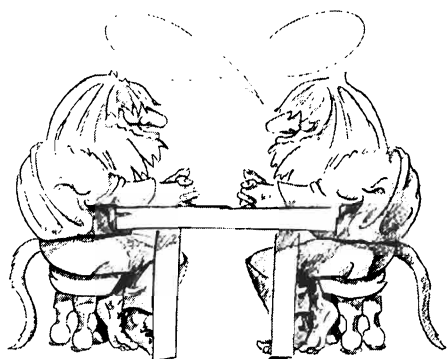


Рис. 1.9. Дерево наследования для классов, используемых в данной главе

В данной главе представлены ключевые концепции связи сигналов и слотов и компоновки графических элементов. Здесь также начал раскрываться последовательный и полностью объектно-ориентированный подход к конструированию и применению виджетов. Если вы просмотрите документацию Qt, то обнаружите, что в ней применяется единый подход, позволяющий легко понять способы применения новых виджетов; вы также обнаружите, что тщательный подбор в Qt имен функций, параметров, элементов перечислений и т. д. делает удивительно приятным и простым программирование в Qt.

Последующие главы части I построены на фундаменте, заложенном в этой главе; они показывают, как следует создавать приложения с полнофункциональным графическим интерфейсом, содержащим меню, панели инструментов, окна документов, строку состояния и диалоговые окна вместе с соответствующими функциональными средствами по чтению, обработке и записи файлов.



- *Подклассы QDialog*
- *Подробное описание технологии сигналов и слотов*
- *Быстрое проектирование диалоговых окон*
- *Изменяющиеся диалоговые окна*
- *Динамические диалоговые окна*
- *Встроенные классы виджетов и диалоговых окон*

Глава 2. Создание диалоговых окон

В данной главе вы научитесь создавать диалоговые окна с использованием средств разработки Qt. Диалоговые окна предоставляют пользователю возможность задавать необходимые значения параметров и выбирать определенные режимы работы. Они называются диалоговыми окнами или просто «диалогами» (dialogs), поскольку представляют собой средство, с помощью которого пользователи и приложения могут «переговариваться» друг с другом.

Большинство приложений с графическим пользовательским интерфейсом имеют главное окно с панелью меню и инструментальной панелью, а также десятки диалоговых окон, естественно дополняющих главное окно. Можно также создать приложение из одного диалогового окна, которое будет непосредственно реагировать на выбор пользователя, выполняя соответствующие действия (например, таким приложением может быть калькулятор).

Первое диалоговое окно мы создадим полностью вручную, чтобы было ясно, как выглядит исходный код такой программы. Затем мы покажем способы построения диалоговых окон в *Qt Designer*, который является средством визуального проектирования в Qt. Использование *Qt Designer* позволяет получать результат значительно быстрее, чем при ручном кодировании, и полученные в нем различные варианты проектов легче тестировать и изменять в будущем.

Подклассы QDialog

Первым нашим примером будет диалоговое окно Find (найти) (показано на рис. 2.1) для поиска заданной пользователем последовательности символов, и оно будет полностью написано на C++. Мы реализуем это диалоговое окно в виде его собственного класса. Причем сделаем его независимым и самодостаточным компонентом, со своими сигналами и слотами.

Исходный код программы содержится в двух файлах: `finddialog.h` и `finddialog.cpp`. Сначала приведем файл `finddialog.h`:

```
1 #ifndef FINDDIALOG_H
2 #define FINDDIALOG_H
3 #include <QDialog.h>
```

```

4 class QCheckBox;
5 class QLabel;
6 class QLineEdit;
7 class QPushButton;

```

Строки 1 и 2 (а также строка 27) предотвращают многократное включение в программу этого заголовочного файла.

В строке 3 в программу включается определение `QDialog` – базового класса для диалоговых окон в Qt. Класс `QDialog` происходит от класса `QWidget`.

В строках с 4-й по 7-ю даются предварительные объявления классов Qt, использующихся для реализации диалогового окна. Предварительное объявление (*forward declaration*) указывает компилятору C++ только на существование класса, не давая подробного определения этого класса (обычно определение класса содержится в его собственном заголовочном файле). Чуть позже мы поговорим об этом более подробно.



Рис. 2.1. Диалоговое окно поиска

Затем мы определяем `FindDialog` как подкласс `QDialog`:

```

8 class FindDialog : public QDialog
9 {
10     Q_OBJECT

11 public:
12     FindDialog(QWidget *parent = 0);

```

Макрос `Q_OBJECT` необходимо задавать в начале определения любого класса, содержащего сигналы или слоты.

Конструктор `FindDialog` является типичным для классов виджетов в Qt. В параметре `parent` (родитель) указывается родительский виджет. По умолчанию задается нулевой указатель, указывая на то, что у данного диалога нет родительского виджета.

```

13 signals:
14     void findNext(const QString &str, Qt::CaseSensitivity cs);
15     void findPrev(const QString &str, Qt::CaseSensitivity cs);

```

В секции `signals` объявляется два сигнала, которые генерируются диалоговым окном при нажатии пользователем кнопки **Find** (найти). Если установлен флажок поиска в обратном направлении (**Search backward**), генерируется сигнал `findPrevious()`; в противном случае генерируется сигнал `findNext()`.

Ключевое слово `signals` на самом деле является макросом. Препроцессор C++ преобразует его в стандартные инструкции языка C++ и затем передает их компилятору. `Qt::CaseSensitivity` является перечислением и может принимать значение `Qt::CaseSensitive` или `Qt::CaseInsensitive`.

```

16 private slots:
17     void findClicked();
18     void enableFindButton(const QString &text);

19 private:
20     QLabel *label;
21     QLineEdit *lineEdit;
22     QCheckBox *caseCheckBox;
23     QCheckBox *backwardCheckBox;
24     QPushButton *findButton;
25     QPushButton *closeButton;
26 };

27 #endif

```

В закрытой (*private*) секции класса мы объявляем два слота. Для реализации слотов нам потребуется большинство дочерних виджетов диалогового окна, поэтому мы резервируем для них соответствующие переменные-указатели. Ключевое слово `slots`, так же как и `signals`, является макросом, который преобразуется в последовательность инструкций, понятных компилятору C++.

Для закрытых переменных мы использовали предварительные объявления их классов. Это допустимо, потому что все они являются указателями, и мы не используем их в заголовочном файле – поэтому компилятору не требуется иметь полные определения классов. Мы могли бы воспользоваться соответствующими заголовочными файлами (`<QCheckBox>`, `<QLabel>` и т. д.), но при использовании предварительных объявлений компилятор работает немного быстрее.

Теперь рассмотрим файл `finddialog.cpp`, в котором находится реализация класса `FindDialog`.

```

1 #include <QtGui >
2 #include "finddialog.h"

```

Во-первых, мы включаем `<QtGui>` – заголовочный файл, который содержит определения классов графического интерфейса Qt. Qt состоит из нескольких модулей, каждый из которых находится в своей собственной библиотеке. Наиболее важными модулями являются `QtCore`, `QtGui`, `QtNetwork`, `QtOpenGL`, `QtSql`, `QtSvg` и `QtXml`. Заголовочный файл `<QtGui>` содержит определение всех классов, входящих в модули `QtCore` и `QtGui`. Включив этот заголовочный файл, мы можем не беспокоиться о включении каждого отдельного класса.

В `finddialog.h` вместо включения `<QDialog>` и использования предварительных объявлений для классов `QCheckBox`, `QLabel`, `QLineEdit` и `QPushButton` мы могли бы просто включить `<QtGui>`. Однако включение такого большого заголовочного файла, взятого из другого заголовочного файла, обычно свидетельствует о плохом стиле кодирования, особенно при разработке больших приложений.

```

3 FindDialog::FindDialog(QWidget *parent)
4     : QDialog(parent)
5 {
6     label = new QLabel(tr("Find &what:"));

```

```
7    lineEdit = new QLineEdit;
8    label->setBuddy(lineEdit);

9    caseCheckBox = new QCheckBox(tr("Match &case"));
10   backwardCheckBox = new QCheckBox(tr("Search backward"));

11   findButton = new QPushButton(tr("&Find"));
12   findButton->setDefault(true);
13   findButton->setEnabled(false);

14   closeButton = new QPushButton(tr("Close"));
```

В строке 4 конструктору базового класса передается указатель на родительский виджет (параметр `parent`). Затем мы создаем дочерние виджеты. Функция `tr()` переводит строковые литералы на другие языки. Она объявляется в классе `QObject` и в каждом подклассе, содержащем макрос `Q_OBJECT`. Любое строковое значение, которое пользователь будет видеть на экране, полезно преобразовывать функцией `tr()`, даже если вы не планируете в настоящий момент переводить ваше приложение на какой-нибудь другой язык. Перевод приложений Qt на другие языки рассматривается в главе 18.

Мы используем знак амперсанда ('&') для задания клавиш быстрого доступа. Например, в строке 11 создается кнопка Find, которая может быть активирована нажатием пользователем сочетания клавиш Alt+F на платформах, поддерживающих клавиши быстрого доступа. Амперсанды могут также применяться для управления фокусом: в строке 6 мы создаем текстовую метку с клавишей быстрого доступа (Alt+W), а в строке 8 устанавливаем строку редактирования в качестве «партнера» этой текстовой метки. Партнером (*buddy*) называется виджет, на который передается фокус при нажатии клавиши быстрого доступа текстовой метки. Поэтому при нажатии пользователем сочетания клавиш Alt+W (клавиша быстрого доступа текстовой метки) фокус переходит на строку редактирования (которая является партнером текстовой метки).

В строке 12 мы делаем кнопку Find используемой по умолчанию, вызывая функцию `setDefault(true)`¹. Кнопка, для которой задан режим использования по умолчанию, будет срабатывать при нажатии пользователем клавиши Enter. В строке 13 мы устанавливаем кнопку Find в неактивный режим. В неактивном режиме виджет обычно имеет серый цвет и не реагирует на действия пользователя.

```
15   connect(lineEdit, SIGNAL(textChanged(const QString &)),
16           this, SLOT(enableFindButton(const QString &)));
17   connect(findButton, SIGNAL(clicked()),
18           this, SLOT(findClicked()));
19   connect(closeButton, SIGNAL(clicked()),
20           this, SLOT(close()));
```

¹ Qt позволяет применять значения TRUE и FALSE на любой платформе и везде использует их в качестве синонимов стандартных значений true и false. Тем не менее нет никакой необходимости в ваших собственных программах использовать написание этих значений большими буквами, если только не приходится применять старый компилятор, не поддерживающий значения true и false. — *Примеч. авт.*

Закрытый слот `enableFindButton(const QString &)` вызывается при всяком изменении значения в строке редактирования. Закрытый слот `findClicked()` вызывается при нажатии пользователем кнопки Find. Само диалоговое окно закрывается при нажатии пользователем кнопки Close (закрыть). Слот `close()` наследуется от класса `QWidget`, и по умолчанию он делает виджет невидимым (но не удаляет его). Программный код слотов `enableFindButton()` и `findClicked()` мы рассмотрим позднее.

Поскольку `QObject` является одним из прародителей `FindDialog`, мы можем не указывать префикс `QObject::` перед вызовами `connect()`.

```
21   QHBoxLayout *topLeftLayout = new QHBoxLayout;
22   topLeftLayout->addWidget(label);
23   topLeftLayout->addWidget(lineEdit);

24   QVBoxLayout *leftLayout = new QVBoxLayout;
25   leftLayout->addLayout(topLeftLayout);
26   leftLayout->addWidget(caseCheckBox);
27   leftLayout->addWidget(backwardCheckBox);

28   QVBoxLayout *rightLayout = new QVBoxLayout;
29   rightLayout->addWidget(findButton);
30   rightLayout->addWidget(closeButton);
31   rightLayout->addStretch();

32   QHBoxLayout *mainLayout = new QHBoxLayout;
33   mainLayout->addLayout(leftLayout);
34   mainLayout->addLayout(rightLayout);
35   setLayout(mainLayout);
```

Затем для размещения виджетов в окне мы используем менеджеры компоновки (*layout managers*). Менеджеры компоновки могут содержать как виджеты, так и другие менеджеры компоновки. Используя различные вложенные комбинации менеджеров компоновки `QHBoxLayout`, `QVBoxLayout` и `QGridLayout`, можно построить очень сложные диалоговые окна.

Для диалогового окна поиска мы используем два менеджера горизонтальной компоновки `QHBoxLayout` и два менеджера вертикальной компоновки `QVBoxLayout` (см. рис. 2.2). Внешний менеджер компоновки является главным; он устанавливается в `FindDialog` в строке 35 и ответствен за всю область, занимаемую диалоговым окном. Остальные три менеджера компоновки являются внутренними. Показанная в нижнем правом углу на рис. 2.2 маленькая «пружинка» является пустым промежутком («распоркой»). Она применяется для образования ниже кнопок Find и Close пустого пространства, обеспечивающего перемещение кнопок в верхнюю часть своего менеджера компоновки.

Одна из особенностей классов менеджеров компоновки заключается в том, что они не являются виджетами. Взамен этого они происходят от класса `QLayout`, который в свою очередь происходит от класса `QObject`. На данном рисунке виджеты выделены сплошными линиями, а менеджеры компоновки очерчены пунктирными линиями, чтобы подчеркнуть их различие. При работе приложения менеджеры компоновки невидимы.

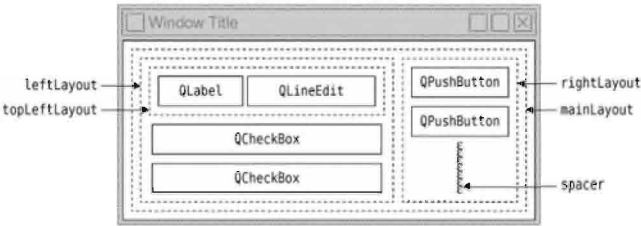


Рис. 2.2. Менеджеры компоновки диалогового окна поиска данных

При добавлении внутренних менеджеров компоновки к родительскому менеджеру компоновки (строки 25, 33 и 34) для них автоматически устанавливается родительская связь. Затем, когда главный менеджер компоновки устанавливается для диалога (строка 35), он становится дочерним элементом диалога и все виджеты в менеджерах компоновки становятся дочерними элементами диалога. Иерархия полученных родословных связей представлена на рис. 2.3.

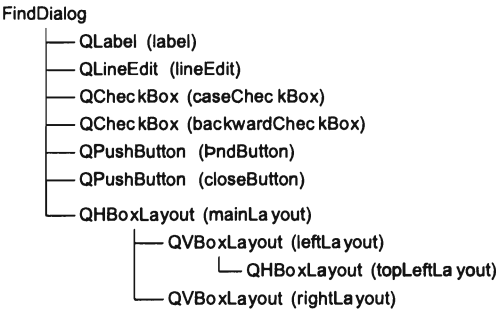


Рис. 2.3. Родословная объектов диалогового окна поиска данных

```
36     setWindowTitle(tr("Find"));
37     setFixedHeight(sizeHint().height());
38 }
```

Наконец, мы задаем название диалогового окна и устанавливаем фиксированной его высоту, поскольку в диалоговом окне нет виджетов, которым может понадобиться дополнительное пространство по вертикали. Функция `QWidget::sizeHint()` возвращает «идеальный» размер виджета.

На этом завершается рассмотрение конструктора `FindDialog`. Поскольку нами использован оператор `new` при создании виджетов и менеджеров компоновки, нам, по-видимому, придется написать деструктор, где будут предусмотрены операторы `delete` для удаления каждого созданного нами виджета и менеджера компоновки. Но поступать так не обязательно, поскольку Qt автоматически удаляет дочерние объекты при разрушении родительского объекта, а все дочерние виджеты и менеджеры компоновки являются потомками `FindDialog`.

Теперь мы рассмотрим слоты диалогового окна.

```

39 void FindDialog::findClicked()
40 {
41     QString text = lineEdit->text();
42     Qt::CaseSensitivity cs =
43         caseCheckBox->isChecked() ? Qt::CaseSensitive
44                                     : Qt::CaseInsensitive;
45     if (backwardCheckBox->isChecked()) {
46         emit findPrevious(text, cs);
47     } else {
48         emit findNext(text, cs);
49     }
50 }

51 void FindDialog::enableFindButton(const QString &text)
52 {
53     findButton->setEnabled(!text.isEmpty());
54 }

```

Слот `findClicked()` вызывается при нажатии пользователем кнопки `Find`. Он генерирует сигнал `findPrevious()` или `findNext()` в зависимости от состояния флажка `Search backward` (поиск в обратном направлении). Ключевое слово `emit` (генерировать сигнал) имеет особый смысл в Qt; как и другие расширения Qt, оно преобразуется препроцессором C++ в стандартные инструкции C++.

Слот `enableFindButton()` вызывается при любом изменении значения в строке редактирования. Он устанавливает активный режим кнопки, если в редактируемой строке имеется какой-нибудь текст; в противном случае кнопка устанавливается в неактивный режим.

Эти два слота завершают написание программы диалогового окна. Теперь мы можем создать файл `main.cpp` и протестировать наш виджет `FindDialog`.

```

1 #include <QApplication>

2 #include "finddialog.h"

3 int main(int argc, char *argv[])
4 {
5     QApplication app(argc, argv);
6     FindDialog *dialog = new FindDialog;
7     dialog->show();
8     return app.exec();
9 }

```

Для компиляции этой программы выполните обычную команду `qmake`. Поскольку определение класса `FindDialog` содержит макрос `Q_OBJECT`, сформированный командой `qmake` файл `makefile` будет содержать специальные правила для запуска `moc` — метаобъектного компилятора Qt. (Мы рассматриваем метаобъектную систему Qt в следующем разделе.)

Для правильной работы `moc` мы должны включить определение класса в заголовочный файл, то есть отделить его от файла реализации класса. Сформированный `moc` программный код содержит этот заголовочный файл и собственно шаблонный код C++.

Классы с макросом `Q_OBJECT` сначала должны пройти через компилятор `moc`. Здесь не будет проблем, поскольку `qmake` автоматически добавляет в файл `makefile` необходимые команды. Однако, если вы забудете сгенерировать файл `makefile` командой `qmake`, программа не пройдет через компилятор `moc` и компоновщик программы пожалуется на то, что некоторые объявленные функции не реализованы. Эти сообщения могут выглядеть достаточно странно. GCC выдает сообщения об ошибках следующего вида:

```
finddialog.o: In function 'FindDialog::tr(char const*, char const*)':  
/usr/lib/qt/src/corelib/global/qglobal.h:1430: undefined reference to  
'FindDialog::staticMetaObject'
```

(В функции `'FindDialog::tr(... )'` не определена ссылка на `'FindDialog::staticMetaObject'`.)

Сообщения в Visual C++ выглядят следующим образом:

```
finddialog.obj : error LNK2001: unresolved external symbol  
"public: virtual int __thiscall MyClass::qt_metacall(enum QMetaObject  
::Call,int,void * *)"
```

(Ошибка NK2001: неразрешенная внешняя ссылка.)

При появлении подобных сообщений снова выполните команду `qmake` для обновления файла `makefile`, затем заново постройте приложение.

Теперь выполните программу. Если клавиши быстрого доступа доступны на вашей платформе, убедитесь в правильной работе клавиш `Alt+W`, `Alt+C`, `Alt+B` и `Alt+F`. Для перехода с одного виджета на другой используйте клавишу табуляции `Tab`. По умолчанию последовательность таких переходов соответствует порядку создания виджетов. Эту последовательность можно изменить с помощью функции `QWidget::setTabOrder()`.

Обеспечение осмысленного порядка переходов с одного виджета на другой с помощью клавиши табуляции и применение клавиш быстрого доступа позволяют использовать все возможности приложений тем пользователям, которые не хотят (или не могут) пользоваться мышкой. Тот, кто быстро работает с клавиатурой, также предпочитает иметь возможность полного управления приложением посредством клавиатуры.

В главе 3 диалоговое окно поиска будет использовано нами в реальном приложении, и мы подключим сигналы `findPrevious()` и `findNext()` к некоторым слотам.

Подробное описание технологии сигналов и слотов

Механизм сигналов и слотов играет решающую роль в разработке программ Qt. Он позволяет прикладному программисту связывать различные объекты, которые ничего не знают друг о друге. Мы уже соединяли некоторые сигналы и слоты, объявляли наши собственные сигналы и слоты, реализовывали наши собственные слоты и генерировали наши собственные сигналы. Давайте рассмотрим этот механизм более подробно.

Слоты почти совпадают с обычными функциями, которые объявляются внутри классов C++ (функции-члены). Они могут быть виртуальными, могут быть перегруженными, они могут быть открытыми (public), защищенными (protected) и закрытыми (private), они могут вызываться непосредственно, как и любые другие функции-члены C++, и их параметры могут быть любого типа. Однако слоты (в отличие от обычных функций-членов) могут подключаться к сигналам, и в результате они будут вызываться при каждом генерировании соответствующего сигнала.

Оператор connect() выглядит следующим образом:

```
connect(отправитель, SIGNAL(сигнал), получатель, SLOT(слот));
```

где отправитель и получатель являются указателями на объекты QObject и где сигнал и слот являются сигнатурами функций без имен параметров. Макросы SIGNAL() и SLOT() фактически преобразуют свои аргументы в строковые переменные.

В приводимых ранее примерах мы всегда подключали разные слоты к разным сигналам. Существует несколько вариантов подключения слотов к сигналам.

- **К одному сигналу можно подключать много слотов:**

```
connect(slider, SIGNAL(valueChanged(int)),
        spinBox, SLOT(setValue(int)));
connect(slider, SIGNAL(valueChanged(int)),
        this, SLOT(updateStatusBarIndicator(int));
```

При генерировании сигнала последовательно вызываются все слоты, причем порядок их вызова не определен.

- **Один слот можно подключать ко многим сигналам:**

```
connect(lcd, SIGNAL(overflow()),
        this, SLOT(handleMathError()));
connect(calculator, SIGNAL(divisionByZero()),
        this, SLOT(handleMathError()));
```

Данный слот будет вызываться при генерировании любого сигнала.

- **Один сигнал может соединяться с другим сигналом:**

```
connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SIGNAL(updateRecord(const QString &)));
```

При генерировании первого сигнала будет также генерироваться второй сигнал. В остальном связь «сигнал – сигнал» не отличается от связи «сигнал – слот».

- **Связь можно аннулировать:**

```
disconnect(lcd, SIGNAL(overflow()),
           this, SLOT(handleMathError()));
```

Это редко приходится делать, поскольку Qt автоматически убирает все связи при удалении объекта.

При успешном соединении сигнала со слотом (или с другим сигналом) их параметры должны задаваться в одинаковом порядке и иметь одинаковый тип:

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),
        this, SLOT(processReply(int, const QString &)));
```

Метаобъектная система Qt

Одним из главных преимуществ средств разработки Qt является расширение языка C++ механизмом создания независимых компонентов программного обеспечения, которые можно соединять вместе, несмотря на то что они могут ничего не знать друг о друге.

Этот механизм называется *метаобъектной системой*, и он обеспечивает две основные служебные функции: взаимодействие сигналов и слотов и анализ внутреннего состояния приложения (introspection). Анализ внутреннего состояния необходим для реализации сигналов и слотов и позволяет прикладным программистам получать «метаинформацию» о подклассах QObject во время выполнения программы, включая список поддерживаемых объектом сигналов и слотов и имена их классов. Этот механизм также поддерживает свойства (широко используемые Qt Designer) и перевод текстовых значений (для интернационализации приложений), а также создает основу для модуля QtScript. Начиная с версии Qt 4.2 свойства можно добавлять динамически, и мы увидим это на практике в главах 19 и 22.

В стандартном языке C++ не предусмотрена динамическая поддержка метаданных, необходимых системе метаобъектов Qt. В Qt эта проблема решена за счет применения специального инструментального средства, компилятора moc, который просматривает определения классов с макросом Q_OBJECT и делает соответствующую информацию доступной функциям C++. Поскольку все функциональные возможности moc обеспечиваются только с помощью «чистого» C++, метаобъектная система Qt будет работать с любым компилятором C++.

Этот механизм работает следующим образом:

- макрос Q_OBJECT объявляет некоторые функции, которые необходимы для анализа внутреннего состояния и которые должны быть реализованы в каждом подклассе QObject: metaObject(), tr(), qt_metacall() и некоторые другие;
- компилятор moc генерирует реализации функций, объявленных макросом Q_OBJECT, и всех сигналов;
- такие функции-члены класса QObject, как connect() и disconnect(), во время своей работы используют функции анализа внутреннего состояния.

Все это выполняется автоматически при работе qmake, moc и при компиляции QObject, и поэтому у вас крайне редко может возникнуть необходимость вспомнить об этом механизме. Однако, если вам интересны детали реализации этого механизма, вы можете воспользоваться документацией по классу QMetaObject и просмотреть файлы исходного кода C++, сгенерированные компилятором moc.

Имеется одно исключение, а именно: если у сигнала больше параметров, чем у подключенного слота, то дополнительные параметры просто игнорируются:

```
connect(ftp, SIGNAL(rawCommandReply(int, const QString &)),  
this, SLOT(checkErrorCode(int)));
```

Если параметры имеют несовместимые типы или будет отсутствовать сигнал или слот, то Qt выдаст предупреждение во время выполнения программы, если

сборка программы проводилась в отладочном режиме. Аналогично Qt выдаст предупреждение, если в сигнатуре сигнала или слота будут указаны имена параметров.

До сих пор мы использовали сигналы и слоты только при работе с виджетами. Но сам по себе этот механизм реализован в классе `QObject`, и его не обязательно применять только в пределах программирования графического пользовательского интерфейса. Этот механизм можно использовать в любом подклассе `QObject`.

```
class Employee : public QObject
{
    Q_OBJECT
public:
    Employee() { mySalary = 0; }
    int salary() const { return mySalary; }
public slots:
    void setSalary(int newSalary);
signals:
    void salaryChanged(int newSalary);
private:
    int mySalary;
};

void Employee::setSalary(int newSalary)
{
    if (newSalary != mySalary) {
        mySalary = newSalary;
        emit salaryChanged(mySalary);
    }
}
```

Обратите внимание на реализацию слота `setSalary()`. Мы генерируем сигнал `salaryChanged()` только при выполнении условия `newSalary != mySalary`. Это позволяет предотвратить бесконечный цикл генерирования сигналов и вызовов слотов.

Быстрое проектирование диалоговых окон

Средства разработки Qt спроектированы таким образом, чтобы было приятно программировать «вручную» и чтобы этот процесс был интуитивно понятен; и нет ничего необычного в разработке всего приложения Qt на «чистом» языке C++. Все же многие программисты предпочитают применять визуальные средства проектирования форм, поскольку этот метод представляется более естественным и позволяет получать конечный результат быстрее, чем при программировании «вручную», и такой подход дает возможность программистам быстрее и легче экспериментировать и изменять дизайн.

Qt Designer расширяет возможности программистов, предоставляя визуальные средства проектирования. Qt Designer может использоваться для разработки всех или только некоторых форм приложения. Формы, созданные с помощью Qt Designer, в конце концов представляются в виде программного кода на C++,

поэтому *Qt Designer* может использоваться совместно с обычными средствами разработки, и он не налагает никаких специальных требований на компилятор.

В данном разделе мы применяем *Qt Designer* для создания диалогового окна (рис. 2.4), которое управляет переходом на заданную ячейку таблицы (Go-to-Cell dialog). Создание диалогового окна как при ручном кодировании, так и при использовании *Qt Designer* предусматривает выполнение следующих шагов:

- 1) создание и инициализация дочерних виджетов;
- 2) размещение дочерних виджетов в менеджерах компоновки;
- 3) определение последовательности переходов по клавише табуляции;
- 4) установка соединений «сигнал – слот»;
- 5) реализация пользовательских слотов диалогового окна.

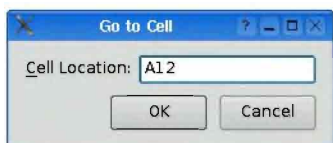


Рис. 2.4. Диалоговое окно для перехода на заданную ячейку таблицы

Для запуска *Qt Designer* выберите функцию Qt by Trolltech v4.x.y|Designer в меню Start системы Windows, наберите designer в командной строке системы Unix или дважды щелкните по Designer в системе Mac OS X Finder. После старта *Qt Designer* выдает список шаблонов. Выберите шаблон «Widget», затем нажмите на кнопку Create (Создать). (Привлекательным может показаться шаблон «Dialog with Buttons Bottom» (диалог с кнопками в нижней части), но в этом примере мы покажем, как создавать кнопки OK и Cancel вручную.) Вы получите на экране окно с заголовком «Untitled».

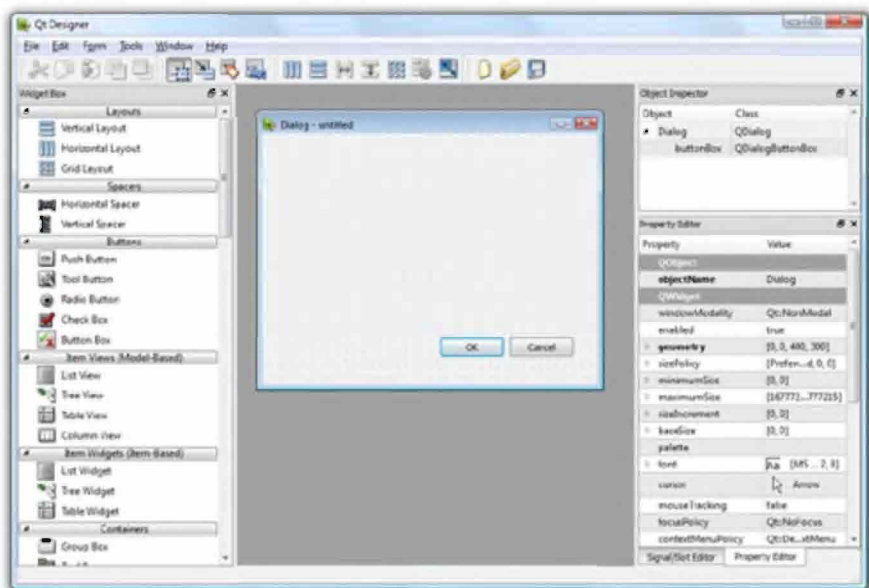


Рис. 2.5. Qt Designer в режиме пристыкованного окна в системе Windows

По умолчанию интерфейс пользователя в *Qt Designer* содержит несколько окон верхнего уровня. Если вы предпочитаете интерфейс в стиле MDI с одним окном верхнего уровня и несколькими подчиненными окнами, как показано на рис. 2.5, выберите функцию *Edit|Preferences* и задайте режим *User Interface* для *Docked Window*.

На первом этапе создайте дочерние виджеты и поместите их в форму. Создайте одну текстовую метку, одну строку редактирования, одну (горизонтальную) распорку (*spacer*) и две кнопки. При создании любого элемента перенесите его название или пиктограмму из окна виджетов *Qt Designer* на форму приблизительно в то место, где он должен располагаться. Элемент Распорка, который не будет видим при работе формы, в *Qt Designer* показан в виде синей пружинки.

Затем передвиньте низ формы вверх, чтобы она стала короче. В результате вы получите форму, похожую на показанную на рис. 2.6. Не тратьте слишком много времени на позиционирование элементов на форме; менеджеры компоновки *Qt* позже выполнят точное их позиционирование.

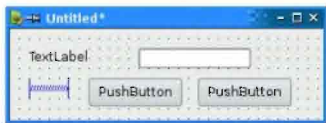


Рис. 2.6. Форма с несколькими виджетами

Задайте свойства каждого виджета, используя редактор свойств *Qt Designer*.

1. Щелкните по текстовой метке. Убедитесь, что свойство *objectName* (имя объекта) имеет значение «*label*» (текстовая метка), а свойство *text* (текст) установите на значение «&Cell Location» (расположение ячейки).
2. Щелкните по строке редактирования. Убедитесь, что свойство *objectName* имеет значение «*lineEdit*» (строка редактирования).
3. Щелкните по первой кнопке. Установите свойство *objectName* на значение «*okButton*» (кнопка подтверждения), свойство *enabled* (включена) на значение «*false*» (ложь), свойство *default* (режим умолчания) на «*true*» (истина), свойство *text* на значение «ОК» (подтвердить).
4. Щелкните по второй кнопке. Установите свойство *objectName* на значение «*cancelButton*» (кнопка отмены) и свойство *text* на значение «Cancel» (отменить).
5. Щелкните по свободному месту формы для выбора самой формы. Установите *objectName* на значение «*GoToCellDialog*» (диалоговое окно перехода на ячейку) и *windowTitle* (заголовок окна) на значение «Go to Cell» (перейти на ячейку).

Теперь все виджеты выглядят привлекательно, кроме текстовой метки &Cell Location. Выберите *Edit|Edit Buddies* (Правка|Редактировать партнеров) для входа в специальный режим, позволяющий задавать партнеров. Щелкните по этой метке, перенесите красную стрелку на строку редактирования и затем отпустите кнопку мыши. Теперь эта метка будет выглядеть как Cell Location (рис. 2.7) и иметь строку редактирования в качестве партнера. Выберите *Click Edit|Edit Widgets* (Правка|Редактировать виджеты) для выхода из режима установки партнеров.

На следующем этапе виджеты размещаются в форме требуемым образом.

1. Щелкните по текстовой метке Cell Location и нажмите клавишу *Shift* одновременно со щелчком по полю редактирования, обеспечив одновременный

- выбор этих виджетов. Выберите в меню **Form|Lay Out Horizontally** (Форма | Горизонтальная компоновка).
- Щелкните по растяжке, затем, удерживая клавишу **Shift**, щелкните по клавишам **OK** и **Cancel**. Выберите в меню **Form|Lay Out Horizontally**.
 - Щелкните по свободному месту формы, аннулируя выбор любых виджетов, затем выберите в меню функцию **Form|Lay Out Vertically** (Форма | Вертикальная компоновка).
 - Выберите в меню функцию **Form|Adjust Size** для установки предпочитаемого размера формы.



Рис. 2.7. Вид формы после установки свойств виджетов

Красными линиями на форме обозначаются созданные менеджеры компоновки (см. рис. 2.8). Они невидимы при выполнении программы.

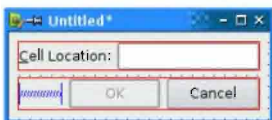


Рис. 2.8. Форма с менеджерами компоновки

Теперь выберите в меню функцию **Edit|Edit Tab Order** (Правка | Редактировать порядок перехода по клавише табуляции). Рядом с каждым виджетом, которому может передаваться фокус, появятся синие прямоугольники (рис. 2.9). Щелкните по каждому виджету, соблюдая необходимую вам последовательность перевода фокуса, затем выберите в меню функцию **Edit|Edit Widgets** для выхода из режима редактирования переходов по клавише табуляции.

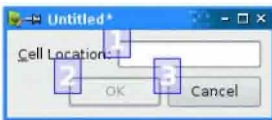


Рис. 2.9. Установка последовательности перевода фокуса по виджетам формы

Для предварительного просмотра спроектированного диалогового окна выберите в меню функцию **Form|Preview** (Форма | Предварительный просмотр). Проверьте последовательность перехода фокуса, нажимая несколько раз клавишу табуляции. Нажмите одновременно клавиши **Alt+C** для перевода фокуса на строку редактирования. Нажмите на кнопку **Cancel** для прекращения работы.

Сохраните спроектированное диалоговое окно в файле `gotocelldialog.ui` в каталоге с названием `gotocell` и создайте файл `main.cpp` в том же каталоге с помощью обычного текстового редактора.

```

#include <QApplication>
#include <QDialog>

#include "ui_gotocelldialog.h"

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

    return app.exec();
}

```

Теперь выполните команду `qmake` для создания файла с расширением `.pro` и затем создайте файл `makefile` (команды `qmake -project`; `qmake gotocell.pro`). Программе `qmake` «хватит ума» обнаружить файл пользовательского интерфейса `gotocelldialog.ui` и сгенерировать соответствующие команды для вызова `uic` – компилятора пользовательского интерфейса, входящего в состав средств разработки Qt. Компилятор `uic` преобразует `gotocelldialog.ui` в инструкции C++ и помещает результат в `ui_gotocelldialog.h`.

Полученный файл `ui_gotocelldialog.h` содержит определение класса `Ui::GoToCellDialog`, который содержит инструкции C++, эквивалентные файлу `gotocelldialog.ui`. В этом классе объявляются переменные-члены, в которых содержатся дочерние виджеты и менеджеры компоновки формы, а также функция `setupUi()`, которая инициализирует форму. Сгенерированный класс выглядит следующим образом:

```

class Ui::GoToCellDialog
{
public:
    QLabel *label;
    QLineEdit *lineEdit;
    QSpacerItem *spacerItem;
    QPushButton *okButton;
    QPushButton *cancelButton;
    ...
    void setupUi(QWidget *widget) {
        ...
    }
};

```

Сгенерированный класс не имеет никакого базового класса. При использовании формы в `main.cpp` мы создаем `QDialog` и передаем его функции `setupUi()`.

Если вы станете выполнять программу в данный момент, она будет работать, но не совсем так, как требуется:

- кнопка ОК всегда будет в неактивном состоянии;
- кнопка Cancel не выполняет никаких действий;
- поле редактирования будет принимать любой текст, а оно должно принимать только допустимое обозначение ячейки.

Правильную работу диалогового окна мы можем обеспечить, написав некоторый программный код. Лучше всего создать новый класс, который происходит от `QDialog` и `Ui::GoToCellDialog` и реализует недостающую функциональность (подтверждая известное утверждение, что любую проблему программного обеспечения можно решить, просто добавив еще один уровень представления объектов). По нашим правилам мы даем этому новому классу такое же имя, которое генерируется компилятором `uic`, но без префикса `Ui::`.

Используя текстовый редактор, создайте файл с именем `gotocelldialog.h`, который будет содержать следующий код:

```
#ifndef GOTOCELLDIALOG_H
#define GOTOCELLDIALOG_H

#include <QDialog>

#include "ui_gotocelldialog.h"

class GoToCellDialog : public QDialog, public Ui::GoToCellDialog
{
    Q_OBJECT

public:
    GoToCellDialog(QWidget *parent = 0);

private slots:
    void on_lineEdit_textChanged();
};

#endif
```

Здесь мы использовали `public`-наследование, поскольку хотим, чтобы доступ к виджетам диалогового окна осуществлялся из-за пределов этого окна. Реализация методов класса делается в файле `gotocelldialog.cpp`.

```
#include <QtGui>

#include "gotocelldialog.h"

GoToCellDialog::GoToCellDialog(QWidget *parent)
    : QDialog(parent)
{
    setupUi(this);
    QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
    lineEdit->setValidator(new QRegExpValidator(regExp, this));
```

```

connect(okButton, SIGNAL(clicked()), this, SLOT(accept()));
connect(cancelButton, SIGNAL(clicked()), this, SLOT(reject()));
}

void GoToCellDialog::on_lineEdit_textChanged()
{
    okButton->setEnabled(lineEdit->hasAcceptableInput());
}

```

В конструкторе мы вызываем `setupUi()` для инициализации формы. Благодаря множественному наследованию мы можем непосредственно получить доступ к членам класса `Ui::GoToCellDialog`. После создания пользовательского интерфейса `setupUi()` будет также автоматически подключать все слоты с именами типа `on_objectName_signalName()` к соответствующему сигналу `signalName()` виджета `objectName`. В нашем примере это означает, что `setupUi()` будет устанавливать следующее соединение «сигнал–слот»:

```

connect(lineEdit, SIGNAL(textChanged(const QString &)),
        this, SLOT(on_lineEdit_textChanged()));

```

Также в конструкторе мы задаем ограничение на допустимый диапазон вводимых значений. Qt обеспечивает три встроенных класса по проверке правильности значений: `QIntValidator`, `QDoubleValidator` и `QRegExpValidator`. В нашем случае мы используем `QRegExpValidator`, задавая регулярное выражение «`[A-Za-z][1-9][0-9]{0,2}`», которое означает следующее: допускается одна маленькая или большая буква, за которой следует одна цифра в диапазоне от 1 до 9; затем идут ноль, одна или две цифры в диапазоне от 0 до 9. (Введение в регулярные выражения вы можете найти в документации по классу `QRegExp`.)

Указывая в конструкторе `QRegExpValidator` значение `this`, мы его делаем дочерним элементом объекта `GoToCellDialog`. После этого нам можно не беспокоиться об удалении в будущем `QRegExpValidator`; этот объект будет удален автоматически после удаления его родительского элемента.

Механизм взаимодействия объекта с родительскими и дочерними элементами реализован в `QObject`. Когда мы создаем объект (виджет, функция по проверке правильности значений или любой другой объект) и он имеет родительский объект, то к списку дочерних элементов этого родителя добавится и данный объект. При удалении родительского элемента будет просмотрен список его дочерних элементов, и все они будут удалены. Эти дочерние элементы в свою очередь сами удалят все свои дочерние элементы, и эта процедура будет выполняться до тех пор, пока ничего не останется.

Механизм взаимодействия объекта с родительскими и дочерними элементами значительно упрощает управление памятью, снижая риск утечек памяти. Явным образом мы должны удалять только объекты, которые созданы оператором `new` и которые не имеют родительского элемента. А если мы удаляем дочерний элемент до удаления его родителя, то Qt автоматически удалит этот объект из списка дочерних объектов этого родителя.

Для виджетов родительский объект имеет дополнительный смысл: дочерние виджеты размещаются внутри области, которую занимает родительский объект. При удалении родительского виджета не только освобождается занимаемая дочерними объектами память – он исчезает с экрана.

В конце конструктора мы подключаем кнопку ОК к слоту `accept()` виджета `QDialog` и кнопку Cancel к слоту `reject()`. Оба слота закрывают диалог, но `accept()` устанавливает результат диалога на значение `QDialog::Accepted` (которое равно 1), а `reject()` устанавливает результат на значение `QDialog::Rejected` (которое равно 0). При использовании этого диалога мы можем использовать результат, чтобы узнать, была ли нажата кнопка ОК, и действовать соответствующим образом.

Слот `on_lineEdit_textChanged()` устанавливает кнопку ОК в активное или неактивное состояние в зависимости от наличия в строке редактирования допустимого обозначения ячейки. `QLineEdit::hasAcceptableInput()` использует функцию проверки допустимости значений, которую мы задали в конструкторе.

На этом завершается построение диалога. Теперь мы можем переписать `main.cpp` следующим образом:

```
#include <QApplication>
#include "gotocelldialog.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    GoToCellDialog *dialog = new GoToCellDialog;
    dialog->show();
    return app.exec();
}
```

Заново сгенерируйте файл `gotocell.pro` с помощью команды `qmake -project` (поскольку мы добавили в проект файлы исходного кода), запустите команду `qmake gotocell.pro`, чтобы обновить `make`-файл, а затем снова постройте и запустите приложение. Наберите в строке редактирования значение «A12» и обратите внимание на то, как кнопка ОК становится активной. Попытайтесь ввести какой-нибудь произвольный текст и посмотрите, как сработает функция по проверке допустимости значения. Нажмите кнопку Cancel для закрытия диалогового окна.

Диалоговое окно работает правильно, но у пользователей Mac OS X кнопки работают по-другому. Мы решили добавлять кнопки по отдельности, чтобы показать, как это делается, но на практике нам следовало бы использовать `QDialogButtonBox`, виджет, содержащий указанные нами кнопки и представляющий их правильно в той системе отображения окон, в которой было запущено приложение (см. рис. 2.10). Чтобы диалоговое окно использовало `QDialogButtonBox`, мы должны изменить как дизайн, так и код. В *Qt Designer* нужно выполнить всего четыре действия.

1. Щелкните мышью по форме (но не по виджетам и не по менеджерам компоновки), а затем выберите пункт меню `Form|Break Layout` (Форма | Прервать компоновку).
2. Щелкайте и удаляйте кнопки ОК и Cancel, горизонтальный разделитель и горизонтальную компоновку (теперь пустую).
3. Перетащите элемент «Button Box» на форму, под меткой Cell Location и строкой редактирования.
4. Щелкните по форме, а затем выберите пункт меню `Form|Layout Vertically` (Форма | Вертикальная компоновка).

Если бы мы внесли изменения только в дизайн формы, например изменили бы компоновку диалогового окна и свойства виджетов, мы могли бы просто заново

построить приложение. Однако здесь мы удалили несколько виджетов и добавили новый виджет, и в этом случае мы должны внести изменения также и в код. Изменения нужно внести в файл `gotocelldialog.cpp`. Вот новая версия конструктора:

```
GoToCellDialog::GoToCellDialog(QWidget *parent)
    : QDialog(parent)
{
    setupUi(this);
    buttonBox->button(QDialogButtonBox::Ok)->setEnabled(false);
    QRegExp regexp("[A-Za-z][1-9][0-9]{0,2}");
    lineEdit->setValidator(new QRegExpValidator(regexp, this));
    connect(buttonBox, SIGNAL(accepted()), this, SLOT(accept()));
    connect(buttonBox, SIGNAL(rejected()), this, SLOT(reject()));
}
```

В предыдущей версии мы исходно отключили кнопку **OK** в *Qt Designer*. В `QDialogButtonBox` мы не можем этого сделать, так что мы сделаем это в коде, сразу после вызова функции `setupUi()`. Класс `QDialogButtonBox` содержит перечислимый тип со стандартными кнопками, и мы можем использовать его для доступа к конкретным кнопкам, в данном случае к кнопке **OK**.



Рис. 2.10. Диалоговое окно перехода на ячейку в Windows Vista и в Mac OS X

Очень удобно, что по умолчанию в *Qt Designer* именем объекта `QDialogButtonBox` будет `buttonBox`. Оба соединения устанавливаются с полями кнопок, а не с самими кнопками. Сигнал `accepted()` генерируется при нажатии кнопки с опцией `AcceptRole`, и, аналогично, сигнал `rejected()` генерируется при нажатии кнопки с опцией `RejectRole`. По умолчанию стандартная кнопка `QDialogButtonBox::Ok` имеет опцию `AcceptRole`, а кнопка `QDialogButtonBox::Cancel` — опцию `RejectRole`. Необходимо внести еще только одно изменение — в слоте `on_lineEdit_textChanged()`:

```
void GoToCellDialog::on_lineEdit_textChanged()
{
    buttonBox->button(QDialogButtonBox::Ok)->setEnabled(
        lineEdit->hasAcceptableInput());
}
```

Единственным отличием здесь будет то, что вместо обращения к конкретной кнопке, хранящейся как переменная-член, мы обращаемся к кнопке **OK** поля кнопок.

Привлекательной особенностью применения *Qt Designer* является возможность программисту действовать достаточно свободно при изменении дизайна формы, причем при этом исходный код программы не будет нарушен. При разработке формы с непосредственным написанием операторов `C++` на изменение дизайна уходит много времени. При использовании *Qt Designer* не будет тратить много времени, поскольку `uis` просто заново генерирует исходный код про-

граммы для форм, которые были изменены. Пользовательский интерфейс диалога сохраняется в файле .ui (который имеет формат XML), а соответствующая функциональная часть реализуется путем создания подкласса сгенерированного компилятором `uiс` класса.

Изменяющиеся диалоговые окна

Нами были рассмотрены способы формирования диалоговых окон, которые всегда содержат одни и те же виджеты. В некоторых случаях требуется иметь диалоговые окна, форма которых может меняться. Наиболее известны два типа изменяющихся диалоговых окон: *расширяемые диалоговые окна* (extension dialogs) и *многостраничные диалоговые окна* (multi-page dialogs). Оба типа диалоговых окон можно реализовать в Qt либо с помощью непосредственного кодирования, либо посредством применения *Qt Designer*.

Расширяемые диалоговые окна, как правило, имеют обычное (нерасширенное) представление и содержат кнопку для переключения между обычным и расширенным представлением этого диалогового окна. Расширяемые диалоговые окна обычно применяются в тех приложениях, которые предназначены как для неопытных, так и опытных пользователей и скрывают дополнительные опции до тех пор, пока пользователь явным образом не захочет ими воспользоваться. В данном разделе мы используем *Qt Designer* для создания расширяемого диалогового окна, показанного на рис. 2.11.

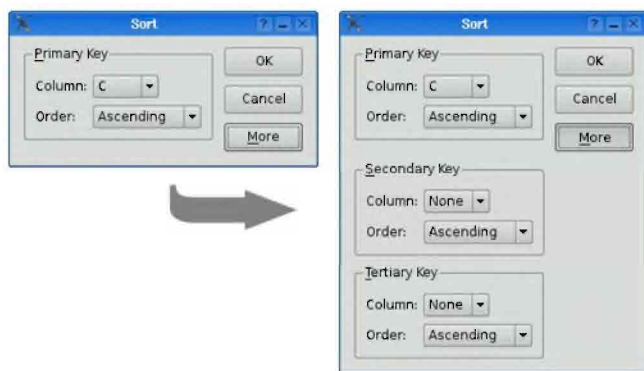
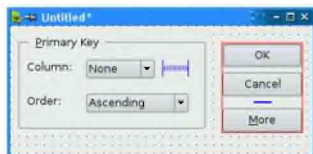


Рис. 2.11. Обычный и расширенный вид окна сортировки данных

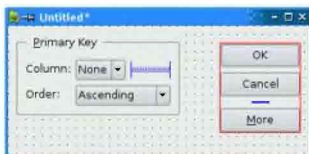
Данное диалоговое окно является окном сортировки в приложении *Электронная таблица*, позволяющим пользователю задавать один или несколько столбцов сортировки. В обычном представлении этого окна пользователь может ввести один ключ сортировки, а в расширенном представлении он может ввести дополнительно еще два ключа сортировки. Кнопка *More* (больше) позволяет пользователю переключаться с обычного представления на расширенное и наоборот.

Мы создадим в *Qt Designer* расширенное представление виджета, второй и третий ключи сортировки которого не будут видны при выполнении программы, когда они не нужны. Этот виджет кажется сложным, однако он очень легко строится в *Qt Designer*. Сначала нужно создать ту часть, которая относится к первичному ключу, затем сдублировать ее дважды, получая вторичный и третичный ключ.

1. Выберите функцию меню `File|New Form` и затем шаблон «Dialog with Buttons Right» (диалог с кнопками, расположенными справа).
2. Создайте кнопку `OK` и перетащите ее в верхнюю правую часть формы. Измените ее свойство `objectName` на «`okButton`» и установите свойство `default` кнопки в значение «`true`».
3. Создайте кнопку `Cancel` и поместите ее под кнопкой `OK`. Измените ее свойство `objectName` на «`cancelButton`».
4. Создайте вертикальную распорку (`spacer`) и поместите ее под кнопкой `OK`, а затем создайте кнопку `More` (больше) и поместите ее ниже вертикальной распорки. Установите свойство `objectName` кнопки `More` в «`moreButton`», установите ее свойство `text` в значение «`&More`», а свойство `checkable` в значение «`true`».
5. Щелкните мышью по кнопке `OK`, а затем, удерживая клавишу `Shift`, щелкните по кнопке `Cancel`, по вертикальной распорке и кнопке `More`, после чего выберите пункт меню `Form|Lay Out Vertically` (Форма | Вертикальная компоновка).
6. Создайте объект группа элементов (`group box`), две текстовые метки, два поля с выпадающим списком (`comboboxes`) и одну горизонтальную распорку и разместите их где-нибудь на форме.
7. Передвиньте нижний правый угол элемента группа, увеличивая его. Затем перенесите другие виджеты внутрь элемента группа и расположите их приблизительно так, как показано на рис. 2.12 (а).
8. Перетащите правый край второго поля с выпадающим списком так, чтобы оно было в два раза шире первого поля.
9. Свойство `title` (заголовок) группы установите на значение «`&PrimaryKey`» (первичный ключ), свойство `text` первой текстовой метки установите на значение «`Column:`» (столбец), а свойство `text` второй текстовой метки установите на значение «`Order:`» (порядок сортировки).
10. Щелкните правой клавишей мыши по первому полю с выпадающим списком и выберите функцию `Edit Items` (редактировать элементы) в контекстном меню для вызова в *Qt Designer* редактора списков. Создайте один элемент со значением «`None`» (нет значений).
11. Щелкните правой клавишей мыши по второму полю с выпадающим списком и выберите функцию `Edit Items`. Создайте элементы «`Ascending`» (по возрастанию) и «`Descending`» (по убыванию).
12. Щелкните по группе и выберите в меню функцию `Form|Lay Out in a Grid` (Форма | Размещение в сетке). Еще раз щелкните по группе и выберите в меню функцию `Form|Adjust Size` (Форма | Настроить размер). В результате получите изображение, представленное на рис. 2.12 (б).



(а) Без менеджера компоновки



(б) С менеджером компоновки

Рис. 2.12. Размещение дочерних виджетов группового элемента в табличной сетке

Если изображение оказалось не совсем таким или вы ошиблись, то всегда можно выбрать в меню функцию **Edit|Undo (Правка|Отменить)** или **Form|Break Layout (Форма|Прервать компоновку)**, затем изменить положение виджетов и снова повторить все действия.

Теперь мы добавим групповые элементы для второго и третьего ключа сортировки.

1. Увеличьте высоту диалогового окна, чтобы можно было в нем разместить дополнительные части.
2. При нажатой клавише **Ctrl (Alt в системе Mac)** щелкните по элементу группы **Primary Key** (первичный ключ) и тащите его для создания копии элемента группа (и его содержимого) над оригинальным элементом. Перетащите эту копию ниже оригинального элемента группа, по-прежнему нажимая клавишу **Ctrl (или Alt)**. Повторите этот процесс для создания третьего элемента группа, размещая его ниже второго элемента группа.
3. Измените их свойство **title** на значения «**&Secondary Key**» (вторичный ключ) и «**&Tertiary Key**» (третичный ключ).
4. Создайте одну вертикальную растяжку и расположите ее между элементом группы первичного ключа и элементом группы вторичного ключа.
5. Расположите виджеты в сетке, как показано на рис. 2.13 (а).
6. Щелкните по форме, чтобы отменить выбор любых виджетов, затем выберите функцию меню **Form|Lay Out in a Grid (Форма| Расположить в сетке)**. Теперь потяните верхний правый угол формы вверх и влево, чтобы сделать форму достаточно маленькой. Форма должна иметь вид, показанный на рис.2.13 (б).
7. Свойство **sizeHint («идеальный» размер)** двух вертикальных растяжек установите на значение **[20, 0]**.

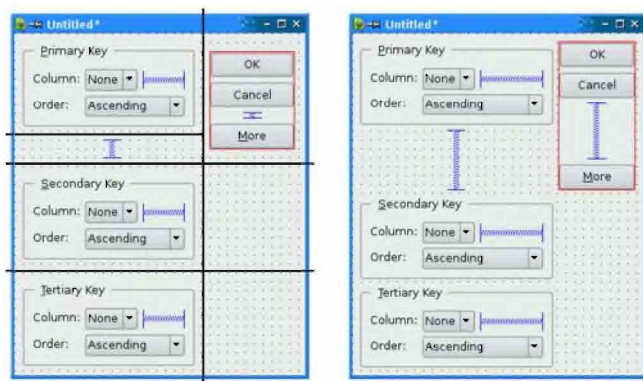


Рис. 2.13. Расположение дочерних элементов формы в сетке

В результате менеджер компоновки в ячейках сетки будет иметь два столбца и четыре строки – всего восемь ячеек. Элемент группа первичного ключа, левая вертикальная распорка, элемент группа вторичного ключа и элемент группа третичного ключа – каждый из них занимает одну ячейку. Менеджер вертикальной компоновки, содержащий кнопки **OK**, **Cancel** и **More**, занимает две ячейки. Справа

внизу диалогового окна будет две свободные ячейки. Если у вас получилась другая картинка, отмените компоновку, измените положение виджетов и повторите все сначала.

Переименуйте форму на «SortDialog» (диалоговое окно сортировки) и измените заголовок на «Sort» (сортировка). Задайте имена дочерним виджетам, как показано на рис. 2.14.

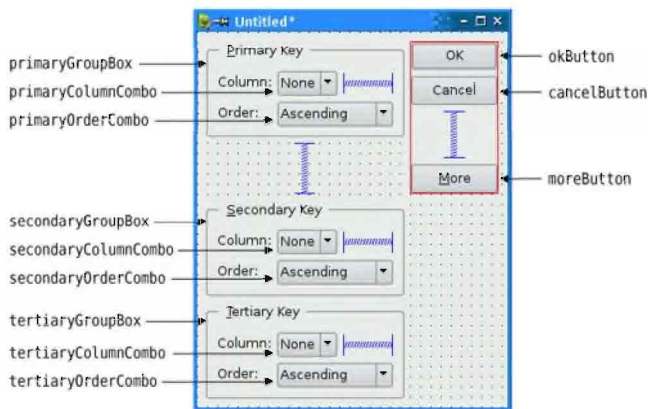


Рис. 2.14. Имена виджетов формы

Выберите функцию меню **Edit | Edit Tab Order**. Щелкайте поочередно по каждому выпадающему списку, начиная с верхнего и заканчивая нижним, затем щелкайте по кнопкам **OK**, **Cancel** и **More**, которые расположены справа. Выберите функцию меню **Edit | Edit Widgets** для выхода из режима установки переходов по клавише табуляции.

Теперь, когда форма спроектирована, мы готовы обеспечить ее функциональное наполнение, устанавливая некоторые соединения «сигнал–слот». *Qt Designer* позволяет устанавливать соединения между виджетами одной формы. Нам требуется обеспечить два соединения.

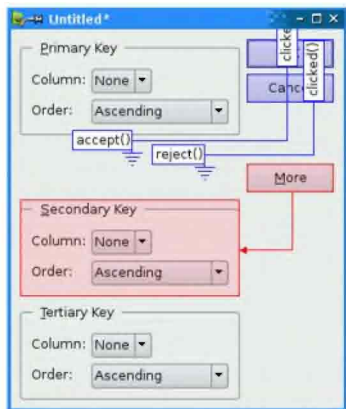


Рис. 2.15. Соединение виджетов формы

Выберите функцию меню **Edit | Edit Signals/Slots (Правка | Редактировать сигналы и слоты)** для входа в режим формирования соединений в *Qt Designer*. Соеди-

нения представлены синими стрелками между виджетами формы, как показано на рис. 2.15, и также они указаны в окне редактора сигналов/слотов *Qt Designer*.

Для установки соединения между двумя виджетами щелкните по виджету, передающему сигнал, соедините красную стрелку с виджетом, получателем сигнала, и отпустите клавишу мышки. В результате будет выдано диалоговое окно, позволяющее выбрать для соединения сигнал и слот.

Сначала устанавливается соединение между `okButton` и слотом `accept()` формы. Перетащите красную стрелку от `okButton` к пустой части формы и отпустите кнопку мыши. Появится диалоговое окно *Configure Connection* (конфигурирование соединения), показанное на рис. 2.16. Выберите сигнал `clicked()` и слот `accept()` и нажмите OK.

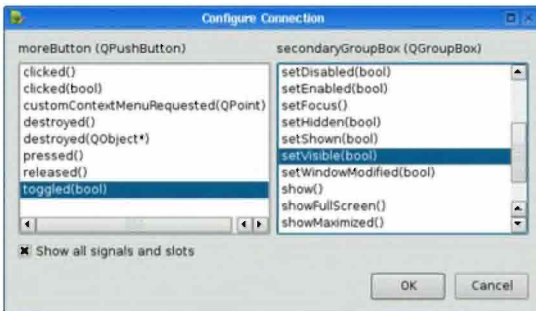


Рис. 2.16. Редактор соединений в *Qt Designer*

Для создания второго соединения перетащите красную стрелку от `cancelButton` на пустую область формы и в диалоговом окне *Configure Connection* соедините сигнал `clicked()` кнопки со слотом `reject()` формы.

Третье соединение устанавливается между `moreButton` и `secondaryGroupBox`. Соедините эти два виджета красной стрелкой, затем выберите `toggled(bool)` в качестве сигнала и `setVisible(bool)` в качестве слота. По умолчанию *Qt Designer* не имеет в списке слотов `setVisible(bool)`, но он появится, если вы включите режим «Show all signals and slots» (Показывать все сигналы и слоты).

Четвертое, и последнее, соединение устанавливается между сигналом `toggled(bool)` виджета `moreButton` и слотом `setVisible(bool)` виджета `tertiaryGroupBox`. После установки соединения выберите функцию меню *Edit|Edit Widgets* для выхода из режима установки соединений.

Сохраните диалог под именем `sortdialog.ui` в каталоге `sort`. Для добавления программного кода в форму мы будем использовать тот же подход на основе множественного наследования, который нами применялся в предыдущем разделе для диалога «Go-to-Cell».

Сначала создаем файл `sortdialog.h` со следующим содержимым:

```
#ifndef SORTDIALOG_H
#define SORTDIALOG_H
#include <QDialog>

#include "ui_sortdialog.h"
class SortDialog : public QDialog, public Ui::SortDialog
```

```

{
    Q_OBJECT

public:
    SortDialog(QWidget *parent = 0);

    void setColumnRange(QChar first, QChar last);
};

#endif

```

Теперь создаем sortdialog.cpp:

```

1 #include <QtGui>

2 #include "sortdialog.h"

3 SortDialog::SortDialog(QWidget *parent)
4     : QDialog(parent)
5 {
6     setupUi(this);

7     secondaryGroupBox->hide();
8     tertiaryGroupBox->hide();
9     layout()->setSizeConstraint(QLayout::SetFixedSize);

10    setColumnRange('A', 'Z');
11 }

12 void SortDialog::setColumnRange(QChar first, QChar last)
13 {
14     primaryColumnCombo->clear();
15     secondaryColumnCombo->clear();
16     tertiaryColumnCombo->clear();

17     secondaryColumnCombo->addItem(tr("None"));
18     tertiaryColumnCombo->addItem(tr("None"));
19     primaryColumnCombo->setMinimumSize(
20         secondaryColumnCombo->sizeHint());
21     QChar ch = first;
22     while (ch <= last) {
23         primaryColumnCombo->addItem(QString(ch));
24         secondaryColumnCombo->addItem(QString(ch));
25         tertiaryColumnCombo->addItem(QString(ch));
26         ch = ch.unicode() + 1;
27     }
28 }

```

Конструктор прячет ту часть диалогового окна, где располагаются поля второго и третьего ключа. Он также устанавливает свойство `sizeConstraint` менеджера компоновки формы на значение `QLayout::SetFixedSize`, не позволяя пользова-

тельно изменять размеры диалогового окна. В этом случае менеджер компоновки отвечает за изменение размеров, и он автоматически изменяет его размеры при появлении или исчезновении дочерних виджетов, всегда обеспечивая оптимальный размер диалогового окна.

Слот `setColumnRange()` инициализирует содержимое полей с выпадающим списком в зависимости от выделенных столбцов электронной таблицы. Мы вставили элемент «None» (значение отсутствует) для полей второго и третьего ключа сортировки (эти ключи необязательны).

В строках 19 и 20 дается пример особо тонкой компоновки. Функция `QWidget::sizeHint()` возвращает размер виджета, «идеальный» с точки зрения системы компоновки. Именно поэтому виджетам различного типа или подобным виджетам с различным содержимым может назначаться системой компоновки неодинаковый размер. Это объясняет разный размер полей с выпадающим списком, когда размер поля второго и третьего ключа сортировки со значением «None» больше, чем размер поля первичного ключа, в качестве значения которого используется только одна буква. Чтобы не было такого несоответствия, мы устанавливаем минимальный размер поля первичного ключа на «идеальный» размер поля вторичного ключа.

Ниже приводится исходный текст функции тестирования, в которой задается диапазон столбцов с «C» по «F» и затем делается видимым диалоговое окно.

```
#include <QApplication>

#include "sortdialog.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    SortDialog *dialog = new SortDialog;
    dialog->setColumnRange('C', 'F');
    dialog->show();
    return app.exec();
}
```

На этом завершается построение расширяемого диалогового окна. Как показывает этот пример, проектировать расширяемые диалоговые окна не намного сложнее, чем проектировать обычные диалоговые окна. Все, что нам здесь потребовалось, — это кнопка переключения, несколько дополнительных связей «сигнал-слот» и использование менеджера компоновки с запрещением изменения размера. В промышленных приложениях достаточно часто можно встретить кнопку, управляющую расширением окна, которая имеет надпись «Advanced >>>» при отображении основного варианта диалогового окна и надпись «Advanced <<<» при отображении расширенного варианта окна. В Qt это можно легко обеспечить путем вызова функции `setText()` при каждом нажатии кнопки `QPushButton`.

Создавать в Qt другой распространенный тип изменяющихся диалоговых окон, многостраничные диалоговые окна, даже еще проще как при ручном кодировании, так и при использовании *Qt Designer*. Такие диалоговые окна можно строить различными способами:

- можно непосредственно воспользоваться виджетом `QTabWidget`. Здесь сверху окна имеется полоска вкладок, которая контролирует встроенный стек `QStackedWidget`;
- можно совместно использовать список `QListWidget` и стек `QStackedWidget`, где текущий элемент списка будет определять страницу, показываемую стеком `QStackedWidget`, обеспечив связь сигнала `QListWidget::currentRowChanged()` со слотом `QStackedWidget::setCurrentIndex()`;
- можно виджет древовидной структуры `QTreeWidget` совместно использовать со стеком `QStackedWidget`, как в предыдущем случае.

Мы рассмотрим класс стека `QStackedWidget` в главе 6.

Динамические диалоговые окна

Динамическими называются диалоговые окна, которые создаются на основе файлов `.ui`, сделанных в *Qt Designer*, во время выполнения приложения. Вместо преобразования файла `.ui` компилятором `uic` в программу на C++ мы можем загрузить этот файл на этапе выполнения, используя класс `QUiLoader`:

```
QUiLoader uiLoader;
QFile file("sortdialog.ui");
QWidget *sortDialog = uiLoader.load(&file);
if (sortDialog) {
    ...
}
```

Мы можем осуществлять доступ к дочерним виджетам формы при помощи функции `QObject::findChild<T>()`:

```
QComboBox *primaryColumnCombo =
    sortDialog->findChild<QComboBox *>("primaryColumnCombo");
if (primaryColumnCombo) {
    ...
}
```

Функция `findChild<T>()` является шаблонной функцией-членом, которая возвращает дочерний объект по заданному имени и типу. Эта функция отсутствует для MSVC 6 из-за ограничений этого компилятора. Если вам необходимо использовать компилятор MSVC 6, вместо этой функции следует вызывать глобальную функцию `qFindChild<T>()`, которая работает в основном так же.

Класс `QUiLoader` расположен в отдельной библиотеке. Для использования класса `QUiLoader` в приложении Qt мы должны добавить в файл `.pro` следующую строку:

```
CONFIG += uitools
```

Динамические диалоговые окна позволяют изменять компоновку элементов формы без повторной компиляции приложения. Они могут также использоваться для создания «тонких» клиентских приложений, когда в исполняемый модуль встраивается только основная форма пользовательского интерфейса, а все другие формы создаются по мере необходимости.

Встроенные классы виджетов и диалоговых окон

Qt содержит большой набор встроенных виджетов и стандартных диалоговых окон, с помощью которых можно реализовать большинство возможных ситуаций. В данном разделе мы представим изображения экранов почти со всеми из них. Несколько специальных виджетов мы рассмотрим позже: такие виджеты главного окна, как `QMenuBar`, `QToolBar` и `QStatusBar`, обсуждаются в главе 3, а виджеты, связанные с компоновкой элементов (такие как `QSplitter` и `QScrollArea`), рассматриваются в главе 6. Большинство встроенных виджетов и диалоговых окон входят в примеры данной книги. На снимках экрана, показанных на рис. с 2.17 по 2.26, виджеты используют стиль `Plastique`.

Qt содержит четыре вида кнопок: `QPushButton`, `QToolButton`, `QCheckBox` и `QRadioButton`, показанных на рис. 2.17. Кнопки `QPushButton` и `QToolButton` получили наибольшее распространение и используются для инициации какого-то действия при их нажатии, но они также могут применяться как переключатели (один щелчок нажимает кнопку, другой щелчок отпускает кнопку). Флажок `QCheckBox` может использоваться для включения и выключения независимых опций, в то время как переключатели (радиокнопки) `QRadioButton` обычно задают взаимоисключающие возможности.



Рис. 2.17. Виджеты кнопок Qt

Контейнеры Qt – это виджеты, которые содержат в себе другие виджеты. Фрейм `QFrame`, кроме того, может использоваться самостоятельно просто для вычерчивания линий, и он наследуется многими другими классами виджетов, включая `QToolBox` и `QLabel`.

`QTabWidget` и `QToolBox` являются многостраничными виджетами. Каждая страница является дочерним виджетом, и страницы нумеруются с нуля.

Виджеты для просмотра списков элементов оптимизированы для работы с большими наборами данных и в них часто используются полосы прокрутки. Работа полосы прокрутки реализуется классом `QAbstractScrollArea`, который является базовым для просмотра списков элементов и для других виджетов, обеспечивающих скроллинг.

Библиотеки Qt содержат систему работы с форматом `rich text`, который может использоваться для отображения и редактирования форматированного текста. Система поддерживает описания шрифтов, выравнивания текста, списков, таблиц и гиперссылок. Документы `rich text` могут создаваться программно, элемент за элементом, или предлагаться в виде текста в формате HTML. Точные теги HTML и свойства CSS, поддерживаемые системой, описаны на сайте <http://doc.trolltech.com/4.3/richtext-html-subset.html>.

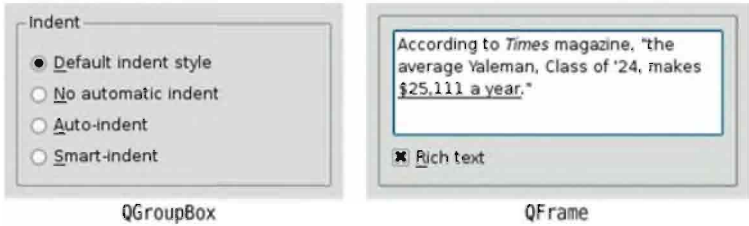


Рис. 2.18. Виджеты одностраничных контейнеров Qt

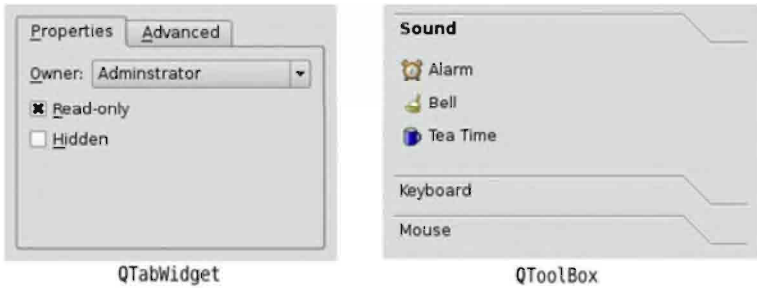


Рис. 2.19. Виджеты многостраничных контейнеров Qt

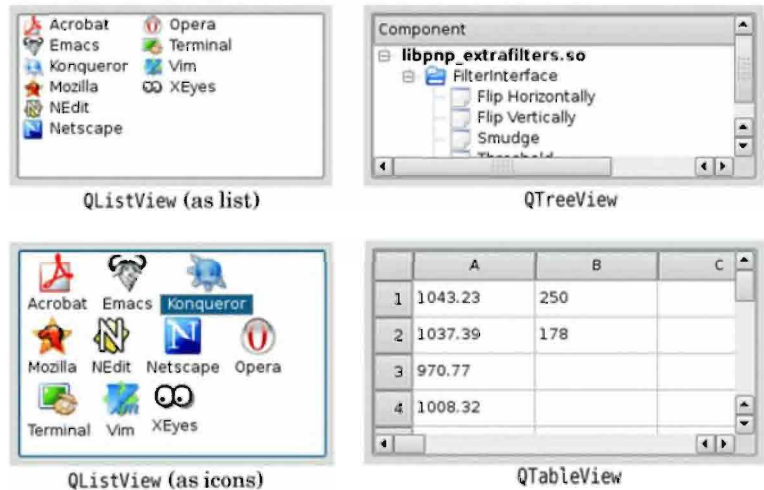


Рис. 2.20. Виджеты для просмотра списков объектов

Qt имеет несколько виджетов, которые предназначены для простого отображения информации. Они показаны на рис. 2.21. Наиболее важным из них является текстовая метка `QLabel`, и она может также использоваться для отображения простого текста, HTML и изображений.

Текстовый браузер `QTextBrowser` представляет собой подкласс `QTextEdit`, работающий только в режиме чтения и отображающий форматированный текст.

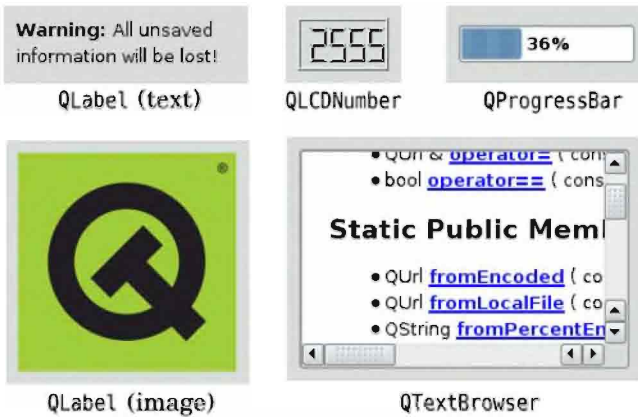


Рис. 2.21. Виджеты отображения данных в Qt

Этот класс используется вместо QLabel для больших форматированных текстовых документов, поскольку в отличие от QLabel он при необходимости автоматически отображает полосы прокрутки, а также предлагает обширные возможности навигации с использованием клавиатуры и мыши. В Qt Assistant 4.3 класс QTextBrowser используется для представления документации для пользователя.

Qt содержит несколько виджетов для ввода данных, показанных на рис. 2.22. Строка редактирования QLineEdit может ограничивать ввод данных, применяя маску ввода и/или функцию проверки допустимости данных. Поле редактирования QTextEdit является подклассом QAbstractScrollArea, позволяющим редактировать тексты большого объема.



Рис. 2.22. Виджеты ввода данных в Qt

Класс `QTextEdit` можно использовать для редактирования простого текста или формата rich text. В последнем случае он позволяет отображать все элементы, поддерживаемые системой работы с rich text. Классы `QLineEdit` и `QTextEdit` полностью интегрированы с буфером обмена.

Qt содержит стандартный набор диалоговых окон для выбора пользователем цвета, шрифта, файла или для печати документа. Они показаны на рис. 2.24 и 2.25.

В Qt предлагается универсальное окно вывода сообщений и диалоговое окно сообщения об ошибке, которое запоминает сообщения, которые им отображались (показано на рис. 2.23). Следить за ходом выполнения операций можно с помощью `QProgressDialog` или `QProgressBar`, показанных на рис. 2.21. Класс `QInputDialog` очень удобен, если необходимо получить от пользователя одну строку текста или одиночное число.

В системах Windows и Mac OS X по мере возможности используются «родные» диалоговые окна, а не их общие аналоги. Цвета также можно выбрать с помощью одного из виджетов для выбора цвета от Qt Solution, а шрифты можно выбирать с помощью встроенного класса `QFontComboBox`.

Наконец, класс `QWizard` предлагает основу для создания мастеров (которые в Mac OS X также называются помощниками – assistant). Мастеры полезны для сложных и редко выполняемых задач, изучение которых может оказаться трудным для пользователя. Пример мастера показан на рис. 2.26.



Рис. 2.23. Диалоговые окна обратной связи в Qt

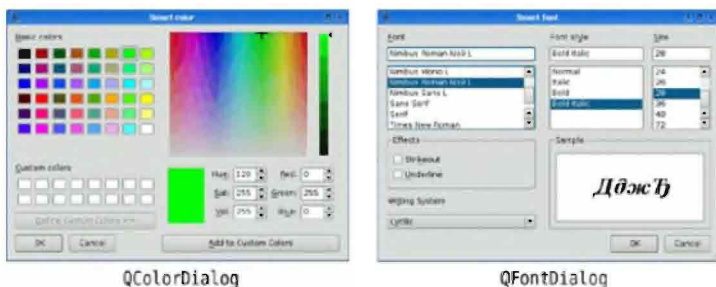


Рис. 2.24. Диалоговое окно выбора цвета и диалоговое окно выбора шрифта в Qt

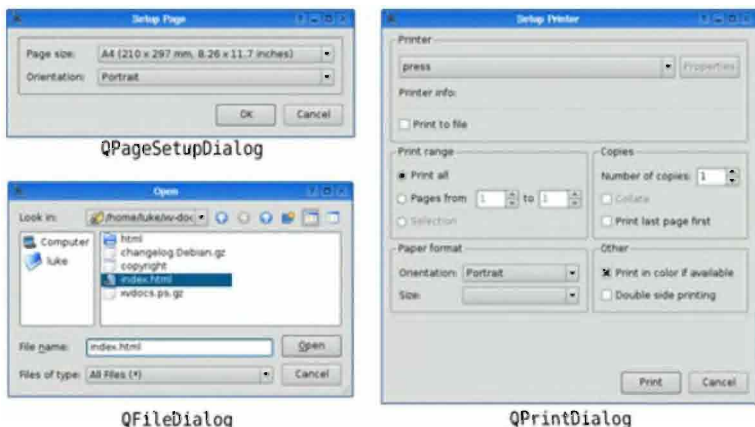


Рис. 2.25. Диалоговое окно для выбора файла и диалоговое окно печати документов в Qt

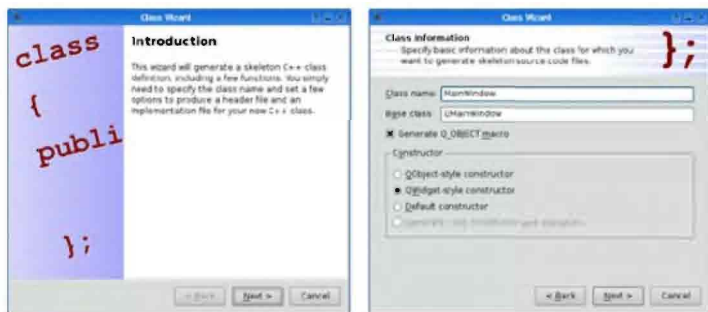
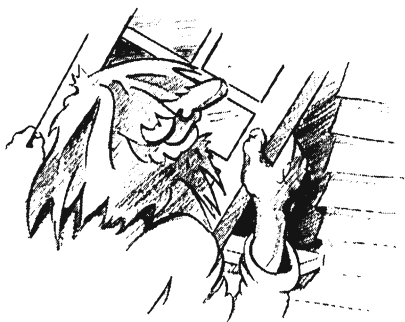


Рис. 2.26. Диалоговое окно QWizard в Qt

Встроенные виджеты и стандартные диалоговые окна обладают большими функциональными возможностями, которыми можно пользоваться без дополнительного программирования. Многие специальные требования обеспечиваются установкой свойств виджетов или путем применения механизма сигналов и слотов и реализуя пользовательские определения слотов.

Если ни один из виджетов или стандартных диалоговых окон, поставляемых с Qt, не подходит для решения задачи, нужное решение можно найти у Qt Solutions или у сторонних коммерческих и некоммерческих производителей. Qt Solutions предлагает ряд дополнительных виджетов, в том числе различные средства определения цвета, средства обработки вращения колесика мыши, особые меню, собственный браузер, а также диалоговое окно копирования.

В некоторых случаях может возникнуть потребность в создании пользовательского виджета без помощи стандартных средств. В Qt это делается просто, и возможности пользовательских виджетов будут обладать таким же свойством независимости от платформы, как и возможности встроенных виджетов Qt. Пользовательские виджеты даже можно интегрировать в *Qt Designer*, и тогда они могут применяться так же, как и встроенные виджеты Qt. В главе 5 объясняются способы создания пользовательских виджетов.



- *Создание подкласса QMainWindow*
- *Создание меню и панелей инструментов*
- *Создание и настройка строки состояния*
- *Реализация меню File*
- *Использование диалоговых окон*
- *Сохранение настроек приложения*
- *Работа со многими документами*
- *Экранные заставки*

Глава 3. Создание главных окон

В данной главе вы научитесь создавать главные окна при помощи средств разработки Qt. К концу главы вы будете способны построить законченный графический пользовательский интерфейс приложения, который имеет меню, панели инструментов и строку состояния и все необходимые приложению диалоговые окна.

Главное окно приложения обеспечивает каркас для построения пользовательского интерфейса приложения. Данная глава будет строиться на основе главного окна приложения *Электронная таблица*, показанного на рис. 3.1. В приложении *Электронная таблица* используются созданные в главе 2 диалоговые окна *Find*, *Go-to-Cell* и *Sort* (найти, перейти на ячейку и сортировать).

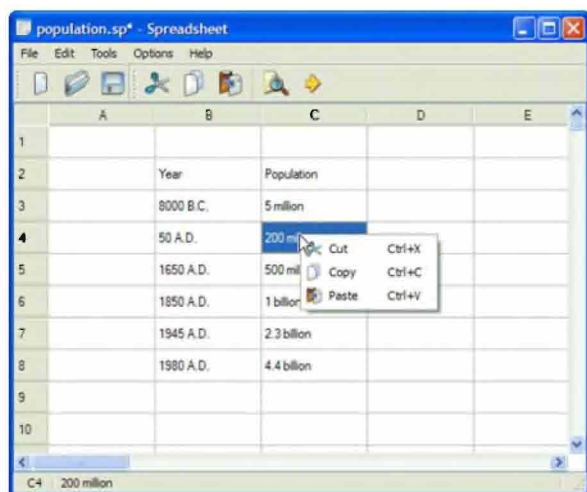


Рис. 3.1. Приложение *Электронная таблица*

В основе большинства приложений с графическим интерфейсом лежит программный код, обеспечивающий базовые функции – например, чтения и записи файлов или обработки данных, представленных в пользовательском интерфейсе.

В главе 4 мы рассмотрим способы реализации такой функциональности, вновь используя в качестве примера приложение Электронная таблица.

Создание подкласса QMainWindow

Главное окно приложения создается в виде подкласса QMainWindow. Многие из представленных в главе 2 методов также подходят для построения главных окон, поскольку оба класса, QDialog и QMainWindow, являются потомками QWidget.

Главные окна можно создавать при помощи *Qt Designer*, но в данной главе мы продемонстрируем, как это все делается при непосредственном программировании. Если вы предпочитаете пользоваться визуальными средствами проектирования, то необходимую информацию сможете найти в главе «Creating a Main Window Application» в онлайн-овом руководстве по *Qt Designer*.

Исходный код программы главного окна приложения Электронная таблица содержится в двух файлах: mainwindow.h и mainwindow.cpp. Сначала приведем заголовочный файл.

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

class QAction;
class QLabel;
class FindDialog;
class Spreadsheet;

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow();

protected:
    void closeEvent(QCloseEvent *event);
```

Мы определяем класс MainWindow как подкласс QMainWindow. Он содержит макрос Q_OBJECT, поскольку имеет собственные сигналы и слоты.

Функция closeEvent() определена в QWidget как виртуальная функция; она автоматически вызывается при закрытии окна пользователем. Она переопределяется в MainWindow для того, чтобы можно было задать пользователю стандартный вопрос относительно возможности сохранения изменений («Do you want to save your changes?») и чтобы сохранить на диске пользовательские настройки.

```
private slots:
    void newFile();
    void open();
    bool save();
```

```
bool saveAs();
void find();
void goToCell();
void sort();
void about();
```

Некоторые функции меню, как, например, File|New (Файл | Создать) или Help|About (Помощь | О программе), реализованы в MainWindow в виде закрытых слотов. Большинство слотов возвращают значение типа void, однако save() и saveAs() возвращают значение типа bool. Возвращаемое значение игнорируется при выполнении слота в ответ на сигнал, но при вызове слота в качестве функции мы можем воспользоваться возвращаемым значением, как это мы можем делать при вызове любой обычной функции C++.

```
void openRecentFile();
void updateStatusBar();
void spreadsheetModified();
```

```
private:
    void createActions();
    void createMenus();
    void createContextMenu();
    void createToolBars();
    void createStatusBar();
    void readSettings();
    void writeSettings();
    bool okToContinue();
    bool loadFile(const QString &fileName);
    bool saveFile(const QString &fileName);
    void setCurrentFile(const QString &fileName);
    void updateRecentFileActions();
    QString strippedName(const QString &fullFileName);
```

Для поддержки пользовательского интерфейса главному окну потребуется еще несколько закрытых слотов и закрытых функций.

```
Spreadsheet *spreadsheet;
FindDialog *findDialog;
QLabel *locationLabel;
QLabel *formulaLabel;
QStringList recentFiles;
QString curFile;

enum { MaxRecentFiles = 5 };
QAction *recentFileActions[MaxRecentFiles];
QAction *separatorAction;

QMenu *fileMenu;
QMenu *editMenu;
```

```
    QToolBar *fileToolBar;  
    QToolBar *editToolBar;  
    QAction *newAction;  
    QAction *openAction;  
    ...  
    QAction *aboutQtAction;  
};  
#endif
```

Кроме этих закрытых слотов и закрытых функций в подклассе `MainWindow` имеется также много закрытых переменных. По мере их использования мы будем объяснять их назначение.

Теперь кратко рассмотрим реализацию этого подкласса.

```
#include <QtGui>  
#include "finddialog.h"  
#include "gotocelldialog.h"  
#include "mainwindow.h"  
#include "sortdialog.h"  
#include "spreadsheet.h"
```

Мы включаем заголовочный файл `<QtGui>`, который содержит определения всех классов Qt, используемых нашим подклассом. Мы также включаем некоторые пользовательские заголовочные файлы из главы 2, а именно `finddialog.h`, `gotocelldialog.h` и `sortdialog.h`.

```
MainWindow::MainWindow()  
{  
    spreadsheet = new Spreadsheet;  
    setCentralWidget(spreadsheet);  
  
    createActions();  
    createMenus();  
    createContextMenu();  
    createToolBars();  
    createStatusBar();  
  
    readSettings();  
    findDialog = 0;  
  
    setWindowIcon(QIcon(":/images/icon.png"));  
    setCurrentFile("");  
}
```

В конструкторе мы начинаем создание виджета электронная таблица `Spreadsheet` и определяем его в качестве центрального виджета главного окна. Центральный виджет занимает среднюю часть главного окна (см. рис. 3.2). Класс `Spreadsheet` является подклассом `QTableWidget`, который обладает некоторыми возможностями электронной таблицы; например, он поддерживает

формулы электронной таблицы. Реализацию этого класса мы рассмотрим в главе 4.

Мы вызываем закрытые функции `createActions()`, `createMenus()`, `createContextMenu()`, `createToolBars()` и `createStatusBar()` для построения остальной части главного окна. Мы также вызываем закрытую функцию `readSettings()` для чтения настроек, сохраненных в приложении.

Мы инициализируем указатель `findDialog` в нулевое значение, а при первом вызове `MainWindow::find()` создадим объект `FindDialog`. В конце конструктора в качестве пиктограммы окна мы задаем PNG-файл: `icon.png`. Qt поддерживает многие форматы графических файлов, включая BMP, GIF, JPEG, PNG, PNM, SVG, TIFF, XBM и XPM. Функция `QWidget::setWindowIcon()` устанавливает пиктограмму в левый верхний угол окна. К сожалению, не существует независимого от платформы способа установки пиктограммы приложения, отображаемого на рабочем столе компьютера. Описание этой процедуры для различных платформ можно найти в сети Интернет по адресу <http://doc.trolltech.com/4.1/appicon.html>.

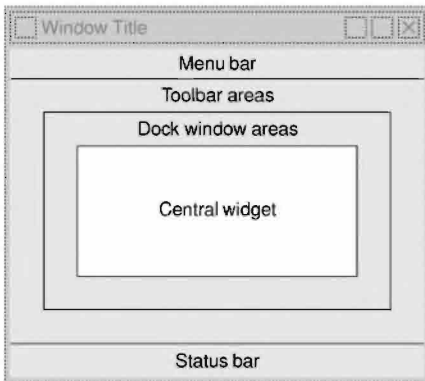


Рис. 3.2. Области главного окна QMainWindow

В приложениях с графическим пользовательским интерфейсом обычно используется много изображений. Существует много различных методов, предназначенных для работы приложения с изображениями. Наиболее распространенными являются следующие:

- хранение изображений в файлах и загрузка их во время выполнения приложения;
- включение файлов XPM в исходный код программы (это возможно, поскольку файлы XPM являются совместимыми с файлами исходного кода C++);
- использование механизма определения ресурсов, предусмотренного в Qt.

Мы используем здесь механизм определения ресурсов, поскольку он более удобен, чем загрузка файлов во время выполнения приложения, и он работает со всеми поддерживаемыми форматами файлов. Мы храним изображения в подкаталоге `images` исходного дерева.

Для применения системы ресурсов Qt мы должны создать файл ресурсов и добавить в файл .pro строку, которая задает этот файл ресурсов. В нашем примере мы назвали файл ресурсов spreadsheet.qrc, поэтому в файл .pro мы добавляем следующую строку:

```
RESOURCES = spreadsheet.qrc
```

Сам файл ресурсов имеет простой XML-формат. Ниже показан фрагмент из используемого нами файла ресурсов.

```
RCC>
<qresource>
  <file>images/icon.png</file>
  ...
  <file>images/gotocell.png</file>
</qresource>
</RCC>
```

Файлы ресурсов после компиляции входят в состав исполняемого модуля приложения, поэтому они не могут теряться. При ссылке на ресурсы мы используем префикс пути `:/` (двоеточие и слеш), и именно поэтому пиктограмма задается как `:/images/icon.png`. Ресурсами могут быть любые файлы (не только изображения), и мы можем их использовать в большинстве случаев, когда в Qt ожидается применение имени файла. Более подробно мы рассматриваем их в главе 12.

Создание меню и панелей инструментов

Большинство современных приложений с графическим пользовательским интерфейсом содержат меню, контекстное меню и панели инструментов. Меню позволяют пользователям исследовать возможности приложения и узнать новые способы работы, а контекстные меню и панели инструментов обеспечивают быстрый доступ к часто используемым функциям. На рис. 3.3 показано меню приложения Электронная таблица.



Рис. 3.3. Меню приложения Электронная таблица

Использование понятия «действие» упрощает программирование меню и панелей инструментов при помощи средств разработки Qt. Элемент action (действие) можно добавлять к любому количеству меню и панелей инструментов. Создание в Qt меню и панелей инструментов разбивается на следующие этапы:

- создание и настройка действий;
- создание меню и добавление к ним действий;
- создание панелей инструментов и добавление к ним действий.

В приложении Электронная таблица действия создаются в `createActions()`:

```
void MainWindow::createActions()
{
    newAction = new QAction(tr("&New"), this);
    newAction->setIcon(QIcon(":/images/new.png"));
    newAction->setShortcut(QKeySequence::New);
    newAction->setStatusTip(tr("Create a new spreadsheet file"));
    connect(newAction, SIGNAL(triggered()), this, SLOT(newFile()));
}
```

Действие **New (создать)** имеет клавишу быстрого выбора пункта меню (**New**), родительское окно (главное окно), пиктограмму, клавишу быстрого вызова команды и сообщение в строке состояния. Большинство оконных систем имеют стандартизованные клавиши быстрого вызова определенных действий. Например, действие **New (Создать)** вызывается в **Windows**, **KDE** и **GNOME** нажатием **Ctrl+N**, а в **Mac OS X** — **Command+N**. С помощью соответствующего перечислимого значения `QKeySequence::StandardKey` мы сделаем так, что **Qt** будет правильно использовать клавиши быстрого вызова той платформы, на которой запущено приложение.

Мы подсоединяем к сигналу этого действия `triggered()` закрытый слот главного окна `newFile()`; этот слот реализуем в следующем разделе. Это соединение гарантирует, что при выборе пользователем пункта меню **File | New (файл | создать)**, при нажатии им кнопки **New** на панели инструментов или при нажатии клавиш **Ctrl+N** будет вызван слот `newFile()`.

Создание действий **Open (открыть)**, **Save (сохранить)** и **Save As (сохранить как)** очень похоже на создание действия **New**, поэтому мы сразу переходим к строке «recently opened files» (недавно открытые файлы) меню **File**:

```
...
for (int i = 0; i < MaxRecentFiles; ++i) {
    recentFileActions[i] = new QAction(this);
    recentFileActions[i]->setVisible(false);
    connect(recentFileActions[i], SIGNAL(triggered()),
            this, SLOT(openRecentFile()));
}
```

Мы заполняем действиями массив `recentFileActions`. Каждое действие скрыто и подключается к слоту `openRecentFile()`. Далее мы покажем, как действия в списке недавно используемых файлов сделать видимыми, чтобы можно было ими воспользоваться.

```
exitAction = new QAction(tr("E&xit"), this);
exitAction->setShortcut(tr("Ctrl+Q"));

exitAction->setStatusTip(tr("Exit the application"));
connect(exitAction, SIGNAL(triggered()), this, SLOT(close()));
```


Действие Exit несколько отличается от действий, которые мы видели до сих пор. Не существует стандартизированной клавиши быстрого вызова для завершения работы приложения, так что мы указываем последовательность клавиш явным образом. Еще одно отличие в том, что мы подключаемся к слоту close() окна, который предоставлен Qt.

Теперь перейдем к действию Select All (выделить все):

```
...
selectAllAction = new QAction(tr("&All"), this);
selectAllAction->setShortcut(QKeySequence::SelectAll);
selectAllAction->setStatusTip(tr("Select all the cells in the "
                                "spreadsheet"));
connect(selectAllAction, SIGNAL(triggered()),
        spreadsheet, SLOT(selectAll()));
```

Слот selectAll() обеспечивается в QAbstractItemView, который является одним из базовых классов QTableWidgetItem, поэтому нам самим не надо его реализовывать.

Давайте теперь перейдем к действию Show Grid (показать сетку) из меню Options (опции):

```
...
showGridAction = new QAction(tr("&Show Grid"), this);
showGridAction->setCheckable(true);
showGridAction->setChecked(spreadsheet->showGrid());
showGridAction->setStatusTip(tr("Show or hide the spreadsheet's "
                                "grid"));
connect(showGridAction, SIGNAL(toggled(bool)),
        spreadsheet, SLOT(setShowGrid(bool)));
```

Действие Show Grid является включаемым. Включаемые действия обозначаются флажком в меню и реализуются как кнопки-переключатели на панели инструментов. Когда это действие включено, на компоненте Spreadsheet отображается сетка. При запуске приложения мы инициализируем это действие в соответствии со значениями, которые принимаются по умолчанию компонентом Spreadsheet, и поэтому работа этого переключателя будет с самого начала синхронизирована. Затем мы соединяем сигнал toggled(bool) действия Show Grid со слотом setShowGrid(bool) компонента Spreadsheet, который наследуется от QTableWidgetItem. После добавления этого действия к меню или панели инструментов пользователь сможет включать и выключать сетку.

Действия-переключатели Show Grid и Auto-Recalculate (автопересчет) работают независимо. Кроме того, Qt обеспечивает возможность определения взаимоисключающих действий путем применения своего собственного класса QActionGroup.

```
...
aboutQtAction = new QAction(tr("About &Qt"), this);
aboutQtAction->setStatusTip(tr("Show the Qt library's About box"));
connect(aboutQtAction, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
}
```

Для действия About Qt (справка по средствам разработки Qt) мы используем слот aboutQt() объекта QApplication, который доступен через глобальную переменную qApp. Это всплывающее диалоговое окно показано на рис. 3.4.

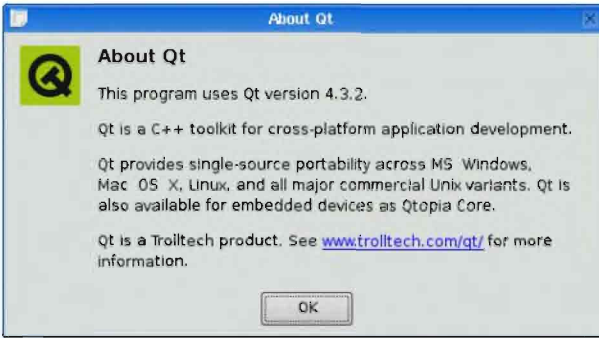


Рис. 3.4. Диалоговое окно About Qt

Действия нами созданы, и теперь мы можем перейти к построению системы меню с этими действиями.

```
void MainWindow::createMenus()
{
    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(newAction);
    fileMenu->addAction(openAction);
    fileMenu->addAction(saveAction);
    fileMenu->addAction(saveAsAction);
    separatorAction = fileMenu->addSeparator();
    for (int i = 0; i < MaxRecentFiles; ++i)
        fileMenu->addAction(recentFileActions[i]);
    fileMenu->addSeparator();
    fileMenu->addAction(exitAction);
}
```

В Qt все меню являются экземплярами класса `QMenu`. Функция `addMenu()` создает виджет `QMenu` с заданным текстом и добавляет его в строку меню. Функция `QMainWindow::menuBar()` возвращает указатель на `QMenuBar`. Строка меню создается при первом вызове `menuBar()`.

Сначала мы создаем меню `File` и затем добавляем к нему действия `New`, `Open`, `Save` и `Save As`. Мы вставляем разделитель для визуального выделения группы взаимосвязанных пунктов меню, используем цикл `for` для добавления (первоначально скрытых) действий из массива `recentFileActions`, а в конце добавляем действие `exitAction`.

Мы сохранили указатель на один из разделителей. Это позволяет нам скрывать этот разделитель (если файлы не использовались) или показывать его, поскольку мы не хотим отображать два разделителя, когда между ними ничего нет.

```
editMenu = menuBar()->addMenu(tr("&Edit"));
editMenu->addAction(cutAction);
editMenu->addAction(copyAction);
editMenu->addAction(pasteAction);
editMenu->addAction(deleteAction);

selectSubMenu = editMenu->addMenu(tr("&Select"));
```

```
selectSubMenu->addAction(selectRowAction);
selectSubMenu->addAction(selectColumnAction);
selectSubMenu->addAction(selectAllAction);
```

```
editMenu->addSeparator();
editMenu->addAction(findAction);
editMenu->addAction(goToCellAction);
```

В меню **Edit (правка)** включается подменю. Это подменю (как и меню, к которому оно принадлежит) является экземпляром класса `QPopupMenu`. Мы просто создаем подменю путем указания `this` в качестве его родителя и вставляем его в то место меню **Edit**, где собираемся его расположить.

Теперь мы создаем меню **Edit (правка)**, добавляя действия при помощи `QMenu::addAction()`, как мы это делали для меню **File**, и добавляя подменю в нужную позицию при помощи `QMenu::addMenu()`. Подменю, как и меню, к которому оно относится, имеет тип `QMenu`:

```
toolsMenu = menuBar()->addMenu(tr("&Tools"));
toolsMenu->addAction(recalculateAction);
toolsMenu->addAction(sortAction);

optionsMenu = menuBar()->addMenu(tr("&Options"));
optionsMenu->addAction(showGridAction);
optionsMenu->addAction(autoRecalcAction);

menuBar()->addSeparator();

helpMenu = menuBar()->addMenu(tr("&Help"));
helpMenu->addAction(aboutAction);
helpMenu->addAction(aboutQtAction);
}
```

Подобным же образом мы создаем меню **Tools**, **Options** и **Help**. Мы вставляем разделитель между меню **Options** и **Help**. В системе **Motif** и **CDE** этот разделитель сдвигает меню **Help** вправо; в других случаях этот разделитель игнорируется. На рис. 3.5 показаны оба случая.

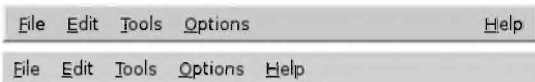


Рис. 3.5. Полоса главного меню в стилях систем **Motif** и **Windows**

```
void MainWindow::createContextMenu()
{
    spreadsheet->addAction(cutAction);
    spreadsheet->addAction(copyAction);
    spreadsheet->addAction(pasteAction);
    spreadsheet->setContextMenuPolicy(Qt::ActionsContextMenu);
}
```

Любой виджет в Qt может иметь связанный с ним список действий QAction. Для обеспечения в приложении контекстного меню мы добавляем необходимые нам действия в виджет Spreadsheet и устанавливаем политику контекстного меню виджета на отображения контекстного меню с этими действиями. Контекстные меню вызываются при щелчке правой клавишей мыши по виджету или при нажатии специальной клавиши клавиатуры, зависящей от платформы. Контекстное меню приложения Электронная таблица показано на рис. 3.6.

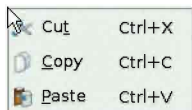


Рис. 3.6. Контекстное меню приложения Электронная таблица

Более сложный способ обеспечения контекстного меню заключается в переопределении функции `QWidget::contextMenuEvent()`, создании виджета `QMenu`, заполнении его требуемыми действиями и вызове для него функции `exec()`.

```
void MainWindow::createToolBars()
{
    fileToolBar = addToolBar(tr("&File"));
    fileToolBar->addAction(newAction);
    fileToolBar->addAction(openAction);
    fileToolBar->addAction(saveAction);

    editToolBar = addToolBar(tr("&Edit"));
    editToolBar->addAction(cutAction);
    editToolBar->addAction(copyAction);
    editToolBar->addAction(pasteAction);
    editToolBar->addSeparator();
    editToolBar->addAction(findAction);
    editToolBar->addAction(goToCellAction);
}
```

Создание панелей инструментов очень похоже на создание меню. Мы создаем панель инструментов File и панель инструментов Edit. Как и меню, панель инструментов может иметь разделители, как это показано на рис. 3.7.



Рис. 3.7. Панели инструментов приложения Электронная таблица

Создание и настройка строки состояния

После создания меню и панелей инструментов мы готовы приступить к созданию строки состояния приложения Электронная таблица.

Обычно строка состояния содержит два индикатора: положение текущей ячейки и формулу текущей ячейки. Полоса состояния также используется для

вывода подсказок и других временных сообщений. На рис. 3.8 показаны разные состояния строки состояний.

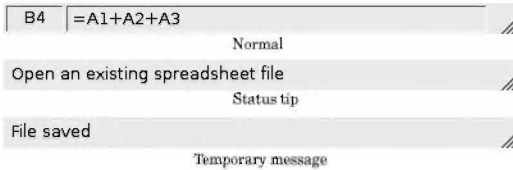


Рис. 3.8. Строка состояния приложения Электронная таблица

Для создания строки состояния в конструкторе `MainWindow` вызывается функция `createStatusBar()`:

```
void MainWindow::createStatusBar()
{
    locationLabel = new QLabel(" W999 ");
    locationLabel->setAlignment(Qt::AlignHCenter);
    locationLabel->setMinimumSize(locationLabel->sizeHint());

    formulaLabel = new QLabel;
    formulaLabel->setIndent(3);
    statusBar()->addWidget(locationLabel);
    statusBar()->addWidget(formulaLabel, 1);

    connect(spreadsheet, SIGNAL(currentCellChanged(int, int, int, int)),
           this, SLOT(updateStatusBar()));
    connect(spreadsheet, SIGNAL(modified()),
           this, SLOT(spreadsheetModified()));

    updateStatusBar();
}
```

Функция `QMainWindow::statusBar()` возвращает указатель на строку состояния. (Строка состояния создается при первом вызове функции `statusBar()`.) В качестве индикаторов состояния просто используются текстовые метки `QLabel`, текст которых изменяется по мере необходимости. Мы добавили отступ для `formulaLabel`, чтобы указанный здесь текст отображался с небольшим смещением от левого края. При добавлении текстовых меток `QLabel` в строку состояния они автоматически становятся дочерними по отношению к строке состояния.

Рис. 3.8 показывает, что эти две текстовые метки занимают различное пространство. Индикатор ячейки занимает очень немного места, и при изменении размеров окна дополнительное пространство будет использовано для правого индикатора, где отображается формула ячейки. Это достигается путем установки фактора растяжения на 1 при вызове функции `QStatusBar::addWidget()` для формулы ячейки. При создании двух других индикаторов для индикатора позиции фактор растяжения по умолчанию равен 0, и поэтому он не будет растягиваться.

Когда `QStatusBar` располагает виджеты индикаторов, он постарается обеспечить «идеальный» размер виджетов, заданный функцией `QWidget::sizeHint()`, и затем растян timer виджеты, которые допускают растяжение, заполняя дополни-

тельное пространство. Идеальный размер виджета зависит от его содержания и будет сам изменяться по мере изменения содержания. Чтобы предотвратить постоянное изменение размера индикатора ячейки, мы устанавливаем его минимальный размер на значение, достаточное для размещения в нем самого большого возможного текстового значения («W999»), и добавляем еще немного пространства. Мы также устанавливаем его параметр выравнивания на значение `AlignHCenter` для выравнивания по центру текста в области индикатора.

Перед завершением функции мы соединяем два сигнала `Spreadsheet` с двумя слотами главного окна `MainWindow`: `updateStatusBar()` и `spreadsheetModified()`.

```
void MainWindow::updateStatusBar()
{
    locationLabel->setText(spreadsheet->currentLocation());
    formulaLabel->setText(spreadsheet->currentFormula());
}
```

Слот `updateStatusBar()` обновляет индикаторы расположения ячейки и формулы ячейки. Он вызывается при любом перемещении пользователем курсора ячейки на новую ячейку. В конце функции `createStatusBar()` этот слот используется как обычная функция для инициализации индикаторов. Это необходимо, поскольку `Spreadsheet` при запуске не генерирует сигнал `currentCellChanged()`.

```
void MainWindow::spreadsheetModified()
{
    setWindowModified(true);
    updateStatusBar();
}
```

Слот `spreadsheetModified()` обновляет все три индикатора для отражения ими текущего состояния приложения и устанавливает переменную `modified` на значение `true`. (Мы использовали переменную `modified` при реализации меню `File` для контроля несохраненных изменений.) Слот `spreadsheetModified()` устанавливает свойство `windowModified` в значение `true`, обновляя строку заголовка. Эта функция обновляет также индикаторы расположения и формулы ячейки, чтобы они отражали текущее состояние.

Реализация меню File

В данном разделе мы определим слоты и закрытые функции, необходимые для обеспечения работы меню `File` и для управления списком недавно используемых файлов.

```
void MainWindow::newFile()
{
    if (okToContinue ()) {
        spreadsheet->clear();
        setCurrentFile("");
    }
}
```

Слот `newFile()` вызывается при выборе пользователем пункта меню `File | New` или при нажатии кнопки `New` на панели инструментов. Закрытая функция `okToContinue()` отображает всплывающее диалоговое окно, показанное на рис. 3.9

(«Do you want to save your changes?» – сохранить изменения?), если изменения до этого не были сохранены. Она возвращает значение `true`, если пользователь отвечает Yes или No (сохраняя документ при ответе Yes), и она возвращает значение `false`, если пользователь отвечает Cancel. Функция `Spreadsheet::clear()` очищает все ячейки и формулы электронной таблицы. Закрытая функция `setCurrentFile()` кроме установки закрытой переменной `curFile` и обновления списка недавно используемых файлов изменяет заголовок окна, отражая тот факт, что редактируемый документ не имеет заголовка.



Рис. 3.9. «Сохранить изменения?»

```
bool MainWindow::okToContinue()
{
    if (isWindowModified()) {
        int r = QMessageBox::warning(this, tr("Spreadsheet"),
                                     tr("The document has been modified.\n"
                                         "Do you want to save your changes?"),
                                     QMessageBox::Yes | QMessageBox::No |
                                     QMessageBox::Cancel);
        if (r == QMessageBox::Yes) {
            return save();
        } else if (r == QMessageBox::Cancel) {
            return false;
        }
    }
    return true;
}
```

В `okToContinue()` мы проверяем свойство `windowModified`. Если оно имеет значение `true`, мы выводим на экран сообщение, показанное на рис. 3.9. Окно сообщения содержит кнопки Yes, No и Cancel.

`QMessageBox` предлагает много стандартных кнопок и автоматически пытается сделать одну кнопку выбранной по умолчанию (активируемой при нажатии пользователем клавиши Enter), а другую – кнопкой отмены (активируемой при нажатии пользователем клавиши Esc). Также существует возможность выбирать те кнопки, которые будут использоваться по умолчанию или будут кнопкой отмены. Кроме того, можно настраивать текст на кнопке.

Вызов функции `warning()` на первый взгляд может показаться слишком сложным, но он имеет очень простой формат:

```
QMessageBox::warning(родительский объект, заголовок, сообщение, кнопки);
```

Наряду с `warning()` `QMessageBox` содержит функции `information()`, `question()` и `critical()`, каждая из которых имеет собственную пиктограмму. Пиктограммы показаны на рис. 3.10.



Рис. 3.10. Пиктограммы окна сообщения в стиле Windows

```
void MainWindow::open()
{
    if (okToContinue()){
        QString fileName = QFileDialog::getOpenFileName(this,
                                                         tr("Open Spreadsheet"), ".",
                                                         tr("Spreadsheet files (*.sp)"));

        if (!fileName.isEmpty())
            loadFile(fileName);
    }
}
```

Слот `open()` соответствует пункту меню `File|Open`. Как и слот `newFile()`, он сначала вызывает `okToContinue()` для обработки несохраненных изменений. Затем он вызывает удобную статическую функцию `QFileDialog::getOpenFileName()` для получения от пользователя нового имени файла. Эта функция выводит на экран диалоговое окно для выбора пользователем файла и возвращает имя файла – или пустую строку при нажатии пользователем клавиши `Cancel`.

В первом аргументе функции `QFileDialog::getOpenFileName()` задается родительский виджет. Взаимодействие родительских и дочерних объектов для диалоговых окон и для других виджетов будет различно. Диалоговое окно всегда является самостоятельным окном, однако если у него имеется родитель, то оно размещается по умолчанию в верхней части родительского объекта. Кроме того, дочернее диалоговое окно использует панель задач родительского объекта.

Во втором аргументе задается название диалогового окна. В третьем аргументе задается каталог начала просмотра файлов; в нашем случае это будет текущий каталог.

Четвертый аргумент определяет фильтры файлов. Фильтр файла состоит из описательной части и образца поиска. Если допустить поддержку не только родного формата файлов приложения *Электронная таблица*, а также формата файлов с запятой в качестве разделителя и файлов *Lotus 1–2–3*, нам пришлось бы инициализировать переменные следующим образом:

```
tr("Spreadsheet files (*.sp)\n"
   "Comma-separated values files (*.csv)\n"
   "Lotus 1-2-3 files (*.wk1 *.wks)")
```

Закрывающая функция `loadFile()` вызвана в `open()` для загрузки файла. Мы делаем эту функцию независимой, поскольку нам потребуется выполнить те же действия для загрузки файлов, которые открывались недавно:

```
bool MainWindow::loadFile(const QString &fileName)
{
    if (!spreadsheet->readFile(fileName)) {
        statusBar()->showMessage(tr("Loading canceled"), 2000);
        return false;
    }
}
```



```

    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File loaded"), 2000);
    return true;
}

```

Мы используем функцию `Spreadsheet::readFile()` для чтения файла с диска. Если загрузка завершилась успешно, мы вызываем функцию `setCurrentFile()` для обновления заголовка окна; в противном случае функция `Spreadsheet::readFile()` уведомит пользователя о возникшей проблеме, выдав соответствующее сообщение. В целом полезно предусматривать выдачу сообщений об ошибках в компонентах низкого уровня, поскольку они могут обеспечить получение точной информации о причинах ошибки.

В обоих случаях мы будем выдавать сообщение в строке состояния в течение двух секунд (2000 миллисекунд) для того, чтобы пользователь знал о выполняемых приложением действиях.

```

bool MainWindow::save()
{
    if (curFile.isEmpty()) {
        return saveAs();
    } else {
        return saveFile(curFile);
    }
}

bool MainWindow::saveFile(const QString &fileName)
{
    if (!spreadsheet->writeFile(fileName)) {
        statusBar()->showMessage(tr("Saving canceled"), 2000);
        return false;
    }
    setCurrentFile(fileName);
    statusBar()->showMessage(tr("File saved"), 2000);
    return true;
}

```

Слот `save()` соответствует пункту меню `File|Save`. Если файл уже имеет имя, потому что уже открывался до этого или уже сохранялся, слот `save()` вызывает `saveFile()`, задавая это имя; в противном случае он просто вызывает `saveAs()`.

```

bool MainWindow::saveAs()
{
    QString fileName = QFileDialog::getSaveFileName(this,
                                                    tr("Save Spreadsheet"), ".",
                                                    tr("Spreadsheet files (*.sp)"));

    if (fileName.isEmpty())
        return false;

    return saveFile(fileName);
}

```

Слот `saveAs()` соответствует пункту меню `File|Save As`. Мы вызываем `QFileDialog::getSaveFileName()` для получения имени файла от пользователя. Если пользователь нажимает кнопку `Cancel`, мы возвращаем значение `false`, которое передается дальше вплоть до вызвавшей функции (`save()` или `okToContinue()`).

Если файл с данным именем уже существует, функция `getSaveFileName()` попросит пользователя подтвердить его перезапись. Такое поведение можно предотвратить, передавая функции `getSaveFileName()` дополнительный аргумент `QFileDialog::DontConfirmOverwrite`.

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    if (okToContinue()) {
        writeSettings();
        event->accept();
    } else {
        event->ignore();
    }
}
```

Когда пользователь выбирает пункт меню `File|Exit` или щелкает по кнопке `X` заголовка окна, вызывается слот `QWidget::close()`. В результате будет сгенерировано событие виджета «close». Переопределяя функцию `QWidget::closeEvent()`, мы можем перехватывать команды по закрытию главного окна и принимать решения относительно возможности его фактического закрытия.

Если изменения не сохранены и пользователь нажимает кнопку `Cancel`, мы «игнорируем» это событие, и оно никак не повлияет на окно. В обычном случае мы реагируем на это событие, и в результате Qt закроет окно. Мы вызываем также закрытую функцию `writeSettings()` для сохранения текущих настроек приложения.

Когда закрывается последнее окно, приложение завершает работу. При необходимости мы можем отменить такой режим работы, устанавливая свойство `quitOnLastWindowClosed` класса `QApplication` на значение `false`, и в результате приложение продолжит выполняться до тех пор, пока мы не вызовем функцию `QApplication::quit()`.

```
void MainWindow::setCurrentFile(const QString &fileName)
{
    curFile = fileName;
    setWindowModified(false);

    QString shownName = "Untitled";
    if (!curFile.isEmpty()) {
        shownName = strippedName(curFile);
        recentFiles.removeAll(curFile);
        recentFiles.prepend(curFile);
        updateRecentFileActions();
    }

    setWindowTitle(tr("%1[*] - %2").arg(shownName)
        .arg(tr("Spreadsheet")));
}
```

```

}

QString MainWindow::strippedName(const QString &fullFileName)
{
    return QFileInfo(fullFileName).fileName();
}

```

В функции `setCurrentFile()` мы задаем значение закрытой переменной `curFile`, в которой содержится имя редактируемого файла. Перед тем как отобразить имя файла в заголовке, мы убираем путь к файлу с помощью функции `strippedName()`, чтобы имя файла выглядело более привлекательно.

Каждый `QWidget` имеет свойство `windowModified`, которое должно быть установлено на значение `true`, если документ окна содержит несохраненные изменения, и на значение `false` в противном случае. В системе `Mac OS X` несохраненные документы отмечаются точкой на кнопке закрытия, расположенной в заголовке окна, в других системах такие документы отмечаются звездочкой в конце имени файла. Все это обеспечивается в `Qt` автоматически, если мы своевременно обновляем свойство `windowModified` и помещаем маркер «[*]» в заголовок окна по мере необходимости.

В функцию `setWindowTitle()` мы передали следующий текст:

```

tr("%1[*] - %2").arg(shownName)
    .arg(tr("Spreadsheet"))

```

Функция `QString::arg()` заменяет своим аргументом параметр «%n» с наименьшим номером и возвращает параметр «%n» с аргументом и полученную строку. В нашем случае `arg()` имеет два параметра «%n». При первом вызове функция `arg()` заменяет параметр «%1»; второй вызов заменяет «%2». Если файл имеет имя «`budget.sp`» и файл перевода не загружен, мы получим строку «`budget.sp[*] – Spreadsheet`». Проще написать:

```

setWindowTitle(shownName + tr("[*] - Spreadsheet"));

```

но применение `arg()` облегчает человеку перевод сообщения на другие языки.

Если задано имя файла, мы обновляем `recentFiles` – список имен файлов, которые открывались в приложении недавно. Мы вызываем функцию `removeAll()` для удаления всех файлов с этим именем из списка, чтобы избежать дублирования; затем мы вызываем функцию `prepend()` для помещения имени данного файла в начало списка. После обновления списка имен файлов мы вызываем функцию `updateRecentFileActions()` для обновления пунктов меню `File`.

```

void MainWindow::updateRecentFileActions()
{
    QMutableStringListIterator i(recentFiles);
    while (i.hasNext()) {
        if (!QFile::exists(i.next()))
            i.remove();
    }

    for (int j = 0; j < MaxRecentFiles; ++j) {
        if (j < recentFiles.count()) {
            QString text = tr("&%1 %2")
                .arg(j + 1)

```

```

        .arg(strippedName(recentFiles[j]));
recentFileActions[j]->setText(text);
recentFileActions[j]->setData(recentFiles[j]);
recentFileActions[j]->setVisible(true);
    } else {
        recentFileActions[j]->setVisible(false);
    }
}
separatorAction->setVisible(!recentFiles.isEmpty());
}

```

Сначала мы удаляем все файлы, которые больше не существуют, используя итератор в стиле Java. Некоторые файлы могли использоваться в предыдущем сеансе, но с этого момента их уже не будет. Переменная `recentFiles` имеет тип `QStringList` (список `QStrings`). В главе 11 подробно рассматриваются такие классы-контейнеры, как `QStringList`, и их связь со стандартной библиотекой шаблонов C++ (Standard Template Library – STL), а также применение в Qt классов итераторов в стиле Java.

Затем мы снова проходим по списку файла, на этот раз пользуясь индексацией массива. Для каждого файла мы создаем строку из амперсанда, номера файла (`j + 1`), пробела и имени файла (без пути). Для соответствующего пункта меню мы задаем этот текст. Например, если первым был файл `C:\My Documents\tab04.sp`, пункт меню первого недавно используемого файла будет иметь текст «&1 tab04.sp». На рис. 3.11 показано соответствие между массивом `recentFileActions` и получающимся меню.

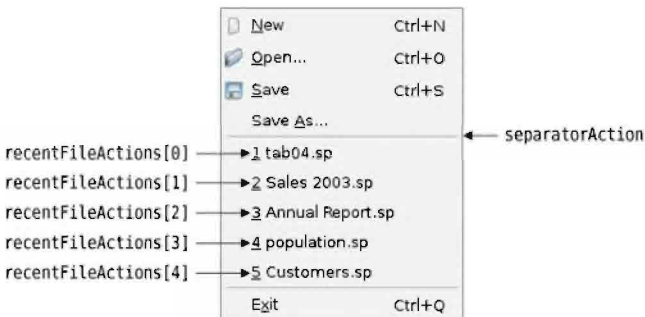


Рис. 3.11. Меню File со списком файлов, которые открывались недавно

С каждым пунктом меню `recentFileActions` может быть связан элемент данных «data» типа `QVariant`. Тип `QVariant` может хранить многие типы C++ и Qt; мы рассмотрим его в главе 11. Здесь в элементе меню «data» мы храним полное имя файла, чтобы можно было позже его легко найти. Мы также делаем этот пункт меню видимым.

Если пунктов меню (массив `recentFileActions`) больше, чем недавно открытых файлов (массив `recentFiles`), мы просто не отображаем дополнительные пункты. Наконец, если существует по крайней мере один недавно используемый файл, мы делаем разделитель видимым.

```
void MainWindow::openRecentFile()
{
    if (okToContinue()) {
        QAction *action = qobject_cast<QAction *>(sender());
        if (action)
            loadFile(action->data().toString());
    }
}
```

При выборе пользователем какого-нибудь недавно используемого файла вызывается слот `openRecentFile()`. Функция `okToContinue()` используется в том случае, когда имеются несохраненные изменения, и если пользователь не отменил сохранение изменений, мы определяем, какой конкретный пункт меню вызвал слот, используя функцию `QObject::sender()`.

Функция `qobject_cast<T>()` выполняет динамическое приведение типов на основе метаинформации, сгенерированной `moc`-компилятором метаобъектов Qt. Она возвращает указатель на запрошенный подкласс `QObject` или 0, если нельзя объект привести к данному типу. В отличие от функции `dynamic_cast<T>()` стандартного C++ функция Qt `qobject_cast<T>()` работает правильно за пределами динамической библиотеки. В нашем примере мы используем `qobject_cast<T>()` для приведения указателя `QObject` в указатель `QAction`. Если приведение успешно (а оно должно быть удачным), мы вызываем функцию `loadFile()`, задавая полное имя файла, которое мы извлекаем из элемента данных пункта меню.

Поскольку мы знаем, что слот вызывался объектом `QAction`, в данном случае программа все же правильно сработала бы при использовании функции `static_cast<T>()` или при традиционном приведении C-типов. См. раздел «Преобразование типов» в Приложении Г, где дается обзор различных методов приведения типов в C++.

Применение диалоговых окон

В данном разделе мы рассмотрим способы применения диалоговых окон в Qt: как они создаются и инициализируются и как реагируют на действия пользователя при работе с ними. Мы будем использовать диалоговые окна `Find`, `Go-to-Cell` и `Sort` (найти, перейти в ячейку и сортировать), которые были созданы нами в главе 2. Мы также создадим простое окно `About` (справка о программе).

Начнем с диалогового окна `Find`, показанного на рис. 3.12. Поскольку мы хотим, чтобы пользователь имел возможность свободно переключаться с главного окна приложения `Электронная таблица` на диалоговое окно `Find` и обратно, это диалоговое окно должно быть *немодальным*. *Немодальным* называется окно, которое может работать независимо от других окон приложения.



Рис. 3.12. Диалоговое окно `Find` приложения `Электронная таблица`

При создании немодальных диалоговых окон они обычно имеют свои сигналы, соединенные со слотами, которые реагируют на действия пользователя:

```
void MainWindow::find()
{
    if (!findDialog) {
        findDialog = new FindDialog(this);
        connect(findDialog, SIGNAL (findNext(const QString &,
                                           Qt::CaseSensitivity)),
                spreadsheet, SLOT (findNext(const QString &,
                                           Qt::CaseSensitivity)));
        connect(findDialog, SIGNAL(findPrevious(const QString &,
                                                Qt::CaseSensitivity)),
                spreadsheet, SLOT(findPrevious(const QString &,
                                                Qt::CaseSensitivity)));
    }

    findDialog->show();
    findDialog->raise();
    findDialog->activateWindow();
}
```

Диалоговое окно Find позволяет пользователю выполнять поиск текста в электронной таблице. Слот find() вызывается при выборе пользователем пункта меню Edit|Find (Правка|Найти) для вывода на экран диалогового окна Find. После этого возможны три сценария развития событий в зависимости от следующих условий:

- диалоговое окно Find вызывается пользователем первый раз;
- диалоговое окно Find уже вызывалось, но пользователь его закрыл;
- диалоговое окно Find уже вызывалось и оно по-прежнему видимо.

Если нет диалогового окна Find, мы создаем его, а его функции findNext() и findPrevious() подсоединяем к соответствующим слотам электронной таблицы Spreadsheet. Мы могли бы также создать это диалоговое окно в конструкторе MainWindow, но отсрочка его создания ускоряет запуск приложения. Кроме того, если это диалоговое окно никогда не будет использовано, то оно и не будет создаваться, что экономит время и память.

Затем мы вызываем функции show(), raise() и activateWindow() и тем самым делаем это окно видимым поверх других и активным. Чтобы сделать скрытое окно видимым поверх других и активным, достаточно вызвать функцию show(), но диалоговое окно Find может вызываться, когда оно уже имеется на экране. В этом случае функция show() ничего не будет делать, и необходимо вызвать raise() и activateWindow(), чтобы сделать окно активным. Можно поступить по-другому и написать

```
if (findDialog->isHidden()) {
    findDialog->show();
} else {
    findDialog->raise();
    findDialog->activateWindow();
}
```

но это аналогично ситуации, когда вы смотрите в обе стороны при переходе улицы с односторонним движением.

Теперь мы перейдем к созданию диалогового окна Go-to-Cell (перейти на ячейку), показанного на рис. 3.13. Мы хотим, чтобы пользователь мог его вызвать, произвести соответствующие действия с его помощью и затем закрыть его; причем пользователь не должен иметь возможность переходить на любое другое окно приложения. Это означает, что диалоговое окно перехода на ячейку должно быть модалным. Окно называется *модалным*, если после его вызова работа приложения блокируется и оказывается невозможной работа с другими окнами приложения до закрытия этого окна. Все используемые нами до сих пор файловые диалоговые окна и окна с сообщениями были модалными.

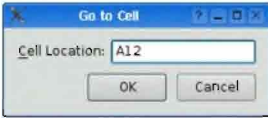


Рис. 3.13. Диалоговое окно Go-to-Cell приложения Электронная таблица

Диалоговое окно будет немодальным, если оно вызывается с помощью функции `show()` (если мы не сделали до этого его модалным, воспользовавшись функцией `setModal()`); оно будет модалным, если вызывается при помощи функции `exec()`.

```
void MainWindow::goToCell()
{
    GoToCellDialog dialog(this);
    if (dialog.exec()) {
        QString str = dialog.lineEdit->text().toUpper();
        spreadsheet->setCurrentCell(str.mid(1).toInt() - 1,
                                    str[0].unicode() - 'A');
    }
}
```

Функция `QDialog::exec()` возвращает значение `true` (`QDialog::Accepted`), если через диалоговое окно подтверждается действие, и значение `false` (`QDialog::Rejected`) в противном случае. Напомним, что мы в главе 2 создали диалоговое окно перехода на ячейку при помощи *Qt Designer* и подсоединили кнопку `OK` к слоту `accept()`, а кнопку `Cancel` к слоту `reject()`. Если пользователь нажимает кнопку `OK`, мы устанавливаем текущую ячейку таблицы на значение, заданное в строке редактирования.

В функции `QTableWidget::setCurrentCell()` задаются два аргумента: индекс строки и индекс столбца. В приложении Электронная таблица обозначение `A1` относится к ячейке (0, 0), а обозначение `B27` относится к ячейке (26, 1). Для получения индекса строки из возвращаемого функцией `QLineEdit::text()` значения типа `QString` мы выделяем номер строки с помощью функции `QString::mid()` (которая возвращает подстроку с первой позиции до конца этой строки), преобразуем ее в целое число типа `int` при помощи функции `QString::toInt()` и вычитаем единицу. Для получения номера столбца мы вычитаем числовой код буквы «A» из числового кода первой буквы строки, преобразованной в прописную. Мы


```

if (dialog.exec()) {
    SpreadsheetCompare compare;
    compare.keys[0] =
        dialog.primaryColumnCombo->currentIndex();
    compare.keys[1] =
        dialog.secondaryColumnCombo->currentIndex() - 1;
    compare.keys[2] =
        dialog.tertiaryColumnCombo->currentIndex() - 1;
    compare.ascending[0] =
        (dialog.primaryOrderCombo->currentIndex() == 0);
    compare.ascending[1] =
        (dialog.secondaryOrderCombo->currentIndex() == 0);
    compare.ascending[2] =
        (dialog.tertiaryOrderCombo->currentIndex() == 0);
    spreadsheet->sort(compare);
}
}

```

Порядок действий при программировании функции `sort()` аналогичен порядку действий, применяемому при программировании функции `goToCell()`:

- мы создаем диалоговое окно в стеке и инициализируем его;
- мы вызываем диалоговое окно при помощи функции `exec()`;
- если пользователь нажимает кнопку ОК, мы используем введенные пользователем в диалоговом окне значения соответствующим образом.

Вызов `setColumnRange()` задает столбцы, выбранные для сортировки. Например, при выделении области, показанной на рис. 3.14, функция `range.leftColumn()` возвратит 0, давая в результате 'A' + 0 = 'A', а `range.rightColumn()` возвратит 2, давая в результате 'A' + 2 = 'C'.

В объекте `compare` хранятся первичный, вторичный и третичный ключи, а также порядок сортировки по ним. (Определение класса `SpreadsheetCompare` мы рассмотрим в следующей главе.) Этот объект используется функцией `Spreadsheet::sort()` для сортировки строк. В массиве `keys` содержатся номера столбцов ключей. Например, если выбрана область с C2 по E5, то столбец C будет иметь индекс 0. В массиве `ascending` в переменных типа `bool` хранятся значения направления сортировки для каждого ключа. Функция `QComboBox::currentIndex()` возвращает индекс текущего элемента (начиная с 0). Для вторичного и третичного ключей мы вычитаем единицу из текущего элемента, чтобы учесть значение «None» (отсутствует).

Функция `sort()` сделает свою работу, но она не совсем надежна. Она предполагает определенный способ реализации диалогового окна, а именно использование выпадающих списков и элементов со значением «None». Это означает, что при изменении дизайна диалогового окна `Sort` нам, возможно, потребуется изменить также программный код. Такой подход можно использовать для диалогового окна, применяемого только в одном месте; однако это может вызвать серьезные проблемы сопровождения, если это диалоговое окно станет использоваться в различных местах.

Более надежным будет такой подход, когда класс `SortDialog` делается более «разумным» и может создавать свой собственный объект `SpreadsheetCompare`, доступный вызывающему его компоненту. Это значительно упрощает функцию `MainWindow::sort()`:

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    QTableWidgetItemSelectionRange range = spreadsheet->selectedRange();
    dialog.setColumnRange('A' + range.leftColumn(),
                          'A' + range.rightColumn());
    if (dialog.exec())
        spreadsheet->performSort(dialog.comparisonObject());
}
```

Такой подход приводит к созданию слабо связанных компонентов, и выбор его почти всегда будет правилен для диалоговых окон, которые вызываются из нескольких мест.

Более «радикальный» подход мог бы заключаться в передаче указателя на объект `Spreadsheet` при инициализации объекта `SortDialog` и разрешении диалоговому окну работать непосредственно с объектом `Spreadsheet`. Это значительно снизит универсальность диалогового окна `SortDialog`, поскольку оно будет работать только с виджетами определенного типа, но это позволит еще больше упростить программу из-за возможности исключения функции `SortDialog::setColumnRange()`. В этом случае функция `MainWindow::sort()` примет следующий вид:

```
void MainWindow::sort()
{
    SortDialog dialog(this);
    dialog.setSpreadsheet(spreadsheet);
    dialog.exec();
}
```

Этот подход является зеркальным отражением первого: вместо знания вызывающим компонентом характерных особенностей диалогового окна теперь само диалоговое окно должно иметь представление об особенностях структур данных, передаваемых вызывающим компонентом. Этот подход полезно применять, когда диалоговому окну требуется отслеживать изменения. В то время как при первом подходе не надежен код вызвавшего компонента, третий подход перестает работать при изменении структуры данных.

Некоторые разработчики выбирают один из подходов и всегда следуют ему. При этом разработка диалоговых окон становится более привычным и простым делом, однако достоинства других подходов не будут использованы. В идеале решение по выбору конкретного подхода должно учитывать в каждом случае особенности конкретного диалогового окна.

Мы завершим данный раздел созданием диалогового окна `About` (справка о программе). Мы могли бы создать для представления данных о программе специальное диалоговое окно наподобие созданных нами ранее диалоговых окон `Find` или `Go-to-Cell`, но поскольку диалоговые окна `About` сильно стилизованы, в средствах разработки `Qt` предусмотрено простое решение:

```
void MainWindow::about()
{
    QMessageBox::about(this, tr("About Spreadsheet"),
                       tr("<h2>Spreadsheet 1.1</h2>"))
}
```

```

    "<p>Copyright &copy; 2008 Software Inc."
    "<p>Spreadsheet is a small application that "
    "demonstrates QAction, QMainWindow, QMenuBar, "
    "QStatusBar, QTableWidget, QToolBar, and many other "
    "Qt classes."));
}

```

Диалоговое окно **About** получается путем вызова удобной статической функции `QMessageBox::about()`. Эта функция очень напоминает функцию `QMessageBox::warning()`, однако здесь вместо стандартных «предупреждающих» пиктограмм используется пиктограмма родительского окна. Получившееся диалоговое окно показано на рис. 3.15.



Рис. 3.15. Справка о приложении Электронная таблица

Таким образом, мы уже сумели воспользоваться несколькими удобными статическими функциями, определенными в классах `QMessageBox` и `QFileDialog`. Эти функции создают диалоговое окно, инициализируют его и вызывают для него функцию `exec()`. Кроме того, вполне возможно, хотя и менее удобно, создать виджет `QMessageBox` или `QFileDialog` так же, как это делается для любого другого виджета, и явно вызвать для него функцию `exec()` или даже `show()`.

Сохранение настроек приложения

В конструкторе `MainWindow` мы уже вызывали функцию `readSettings()` для загрузки сохраненных приложением настроек. Аналогично в функции `closeEvent()` мы вызывали `writeSettings()` для сохранения настроек. Эти функции являются последними функциями-членами `MainWindow`, которые необходимо реализовать.

```

void MainWindow::writeSettings()
{
    QSettings settings("Software Inc.", "Spreadsheet");
    settings.setValue("geometry", saveGeometry());
    settings.setValue("recentFiles", recentFiles);
    settings.setValue("showGrid", showGridAction->isChecked());
    settings.setValue("autoRecalc", autoRecalcAction->isChecked());
}

```

Функция `writeSettings()` сохраняет «геометрию» окна (положение и размер), список последних открывавшихся файлов и опции **Show Grid** (показать сетку) и **Auto-Recalculate** (автоматический повтор вычислений).

По умолчанию `QSettings` сохраняет настройки приложения в месте, которое зависит от используемой платформы. В системе Windows для этого используется системный реестр; в системе Unix данные хранятся в текстовых файлах; в системе Mac OS X для этого используется прикладной интерфейс задания установок Core Foundation Preferences.

В аргументах конструктора задаются название организации и имя приложения. Эта информация используется затем (причем по-разному для различных платформ) для определения места расположения настроек.

`QSettings` хранит настройки в виде пары «ключ–значение». Здесь ключ подобен пути файловой системы. Подключи могут задаваться, используя синтаксис, подобный тому, который применяется при указании пути (например, `findDialog/matchCase`), или используя `beginGroup()` и `endGroup()`:

```
settings.beginGroup("findDialog");
settings.setValue("matchCase", caseCheckBox->isChecked());
settings.setValue("searchBackward", backwardCheckBox->isChecked());
settings.endGroup();
```

Значение `value` может иметь тип `int`, `bool`, `double`, `QString`, `QStringList` или любой другой, поддерживаемый `QVariant`, включая зарегистрированные пользовательские типы.

```
void MainWindow::readSettings()
{
    QSettings settings("Software Inc.", "Spreadsheet");

    restoreGeometry(settings.value("geometry").toByteArray());
    recentFiles = settings.value("recentFiles").toStringList();
    updateRecentFileActions();

    bool showGrid = settings.value("showGrid", true).toBool();
    showGridAction->setChecked(showGrid);
    bool autoRecalc = settings.value("autoRecalc", true).toBool();
    autoRecalcAction->setChecked(autoRecalc);
}
```

Функция `readSettings()` загружает настройки, которые были сохранены функцией `writeSettings()`. Второй аргумент функции `value()` определяет значение, принимаемое по умолчанию в случае отсутствия запрашиваемого параметра. Принимаемые по умолчанию значения будут использованы при первом запуске приложения. Поскольку второй аргумент не задан для геометрии или для списка недавно используемых файлов, окно будет иметь произвольный, но разумный размер и положение, а список недавно используемых файлов будет пустым при первом запуске приложения.

Весь программный код `MainWindow`, относящийся к объектам `QSettings`, мы разместили в функциях `readSettings()` и `writeSettings()`; такой подход лишь один из возможных. Объект `QSettings` может создаваться для запроса или модификации каких-нибудь настроек в любой момент во время выполнения приложения и из любого места программы.

Теперь мы завершили построение главного окна `MainWindow` приложения Электронная таблица. В следующих разделах мы рассмотрим возможность модификации

приложения Электронная таблица для обеспечения работы со многими документами и реализации экранных заставок. Мы завершим реализацию этих функций, в том числе обеспечивающих обработку формул и сортировку, в следующей главе.

Работа с несколькими документами

Теперь мы готовы написать функцию `main()` приложения Электронная таблица:

```
#include <QApplication>

#include "mainwindow.h"

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

Данная функция `main()` немного отличается от написанных ранее: мы создали экземпляр `MainWindow` в виде переменной стека, а не использовали оператор `new`. Экземпляр `MainWindow` будет автоматически уничтожен после завершения функции.

При применении только что показанной функции `main()` приложение Электронная таблица обеспечивает вывод на экран только одного главного окна и позволяет работать только с одним документом. Если мы хотим одновременно редактировать несколько документов, нам придется запускать несколько приложений Электронная таблица. Но это будет не так удобно, как если бы один экземпляр приложения обеспечивал вывод на экран многих главных окон, подобно тому как один экземпляр веб-браузера позволяет просматривать одновременно несколько окон.

Мы модифицируем приложение Электронная таблица для обеспечения возможности работы со многими документами. Для начала нам потребуется немного видоизменить меню `File`:

- пункт меню `File|New` создает новое главное с пустым документом вместо повторного использования существующего главного окна;
- пункт меню `File|Close` закрывает текущее главное окно;
- пункт меню `File|Exit` закрывает все окна.

В первоначальной версии меню `File` не было пункта `Close` (закрыть), поскольку он выполнял бы ту же функцию, что и пункт меню `Exit`. Новое меню `File` показано на рис. 3.16.

Новая функция `main()` примет следующий вид:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    MainWindow *mainWin = new MainWindow;
    mainWin->show();
    return app.exec();
}
```

При работе со многими окнами теперь имеет смысл создавать `MainWindow` оператором `new`, потому что затем мы можем использовать оператор `delete` для удаления главного окна после завершения работы с ним с целью экономии памяти.

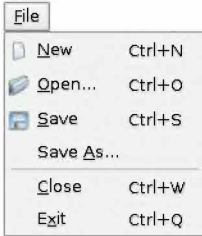


Рис. 3.16. Новое меню File

Новый слот `MainWindow::newFile()` будет выглядеть следующим образом:

```
void MainWindow::newFile()
{
    MainWindow *mainWin = new MainWindow;
    mainWin->show();
}
```

Мы просто создаем новый экземпляр `MainWindow`. Может показаться странным, что мы нигде не сохраняем указатель на новое окно, но это не составит проблемы, поскольку Qt отслеживает все окна.

Действия `Close` и `Exit` будут задаваться следующим образом:

```
void MainWindow::createActions()
{
    ...
    closeAction = new QAction(tr("&Close"), this);
    closeAction->setShortcut(QKeySequence::Close);
    closeAction->setStatusTip(tr("Close this window"));
    connect(closeAction, SIGNAL(triggered()), this, SLOT(close()));

    exitAction = new QAction(tr("E&xit"), this);
    exitAction->setShortcut(tr("Ctrl+Q"));
    exitAction->setStatusTip(tr("Exit the application"));
    connect(exitAction, SIGNAL(triggered()),
            qApp, SLOT(closeAllWindows()));
    ...
}
```

Слот `closeAllWindows()` объекта `QApplication` закрывает все окна приложения, если только никакое из них не отклоняет запрос (event) на его закрытие. Именно такой режим работы нам здесь нужен. Нам не надо беспокоиться о несохраненных изменениях, поскольку обработка этого события выполняется функцией `MainWindow::closeEvent()` при каждом закрытии окна.

Можно подумать, что на этом завершается построение приложения, работающего со многими документами. К сожалению, одна проблема оказалась незаме-

ченной. Если пользователь будет постоянно создавать и закрывать главные окна, в конце концов может не хватить памяти компьютера. Это происходит из-за того, что мы создаем виджеты `MainWindow` в функции `newFile()`, но никогда не удаляем их. Когда пользователь закрывает главное окно, оно исчезнет с экрана, но по-прежнему останется в памяти. При создании многих окон может возникнуть проблема.

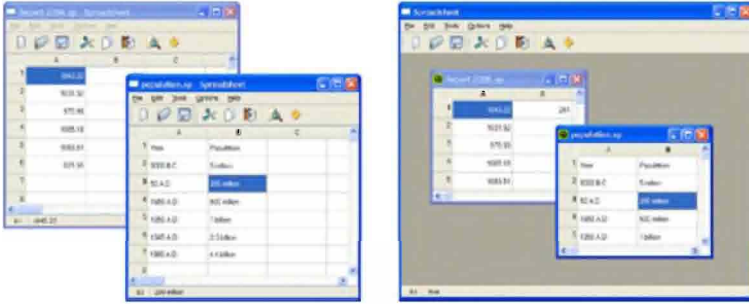


Рис. 3.17. Однодокументный и многодокументный интерфейсы

Решение состоит в установке признака `Qt::WA_DeleteOnClose` в конструкторе: `MainWindow::MainWindow()`

```
{
    ...
    setAttribute(Qt::WA_DeleteOnClose);
    ...
}
```

Это указывает Qt на необходимость удаления окна при его закрытии. Кроме `Qt::WA_DeleteOnClose` в конструкторе `QWidget` можно устанавливать много других флажков, задавая необходимый режим работы виджета.

Утечка памяти – не единственная проблема, с которой мы можем столкнуться. В нашем первоначальном проекте приложения подразумевалось, что у нас будет только одно главное окно. При работе со многими окнами каждое главное окно будет иметь свой список файлов, открывавшихся последними, и свои параметры работы. Очевидно, что список последних открывавшихся файлов должен относиться ко всему приложению. Это можно обеспечить очень просто – путем объявления статической переменной `recentFiles` – и тогда во всем приложении будет только один ее экземпляр. Но здесь мы должны обеспечить при каждом вызове функции `updateRecentFileActions()` для обновления меню `File` вызов ее для всех главных окон. Это выполняет следующий программный код:

```
foreach (QWidget *win, QApplication::topLevelWidgets()) {
    if (MainWindow *mainWin = qobject_cast<MainWindow *>(win))
        mainWin->updateRecentFileActions();
}
```

Здесь используется конструкция Qt `foreach` (она рассматривается в главе 11) для прохода по всем имеющимся в приложении виджетам и делается вызов функции `updateRecentFileItems()` для всех виджетов типа `MainWindow`. Аналогичным образом можно синхронизировать установку опций `ShowGrid` и `Auto-Recalculate` или убедиться в том, что не загружено два файла с одинаковым именем.

Приложения, обеспечивающие работу с одним документом в главном окне, называются приложениями с однодокументным интерфейсом (SDI – single document interface). Распространенной альтернативой ему в Windows стал многодокументный интерфейс (MDI – multiple document interface), когда приложение имеет одно главное окно, в центральной области которого могут находиться окна многих документов. С помощью средств разработки Qt можно создавать как приложения SDI, так и приложения MDI на всех поддерживаемых платформах. На рис. 3.17 показан вид приложения Электронная таблица при использовании обоих подходов. Интерфейс MDI мы рассматриваем в главе 6.

Экранные заставки

Многие приложения при запуске выводят на экран заставки, например такую, которая показана на рис. 3.18. Некоторыми разработчиками заставки используются, чтобы сделать менее заметным медленный запуск приложения, а в других случаях это делается для удовлетворения требований отделений, отвечающих за маркетинг. Можно очень просто добавить заставку в приложение Qt, используя класс `QSplashScreen`.

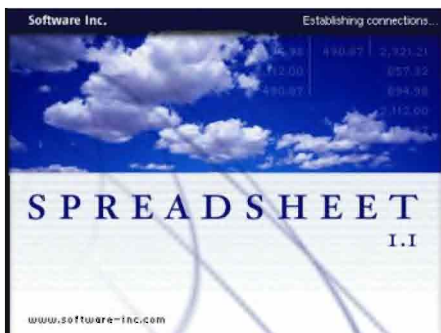


Рис. 3.18. Экранная заставка

Класс `QSplashScreen` выводит на экран изображение до появления главного окна. Он также может вывести на изображение сообщение, информирующее пользователя о ходе процесса инициализации приложения. Обычно вызов заставки делается в функции `main()` до вызова функции `QApplication::exec()`.

Ниже приводится пример функции `main()`, которая использует `QSplashScreen` для вывода заставки приложения, которое загружает модули и устанавливает сетевые соединения при запуске.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QSplashScreen *splash = new QSplashScreen;
    splash->setPixmap(QPixmap(":/images/splash.png"));
    splash->show();

    Qt::Alignment topRight = Qt::AlignRight | Qt::AlignTop;
```



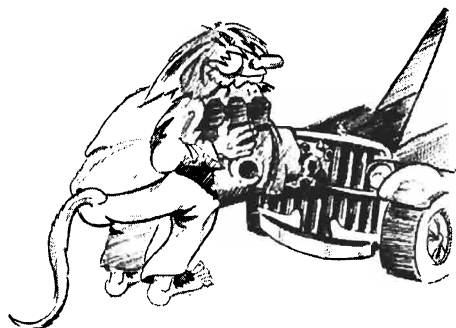
```
    splash->showMessage(QObject::tr("Setting up the main window..."),
                       topRight, Qt::white);
    MainWindow mainWin;
    splash->showMessage(QObject::tr("Loading modules..."),
                       topRight, Qt::white);
    loadModules();

    splash->showMessage(QObject::tr("Establishing connections..."),
                       topRight, Qt::white);
    establishConnections();

    mainWin.show();
    splash->finish(&mainWin);
    delete splash;

    return app.exec();
}
```

Теперь мы завершили пользовательский интерфейс приложения Электронная таблица. В следующей главе мы реализуем базовые функции электронной таблицы и на этом завершим построение этого приложения.



- *Центральный виджет*
- *Создание подкласса `QTableWidget`*
- *Загрузка и сохранение*
- *Реализация меню `Edit`*
- *Реализация других меню*
- *Создание подкласса `QTableWidgetItem`*

Глава 4. Реализация функциональности приложения

В двух предыдущих главах мы показали способы создания пользовательского интерфейса приложения Электронная таблица. В данной главе мы завершим программирование функций, обеспечивающих работу этого интерфейса. Кроме того, мы рассмотрим способы загрузки и сохранения файлов, хранения данных в памяти, реализации операций с буфером обмена (clipboard) и добавления поддержки формул электронной таблицы к классу `QTableWidget`.

Центральный виджет

Центральную область `QMainWindow` может занимать любой виджет. Ниже дается краткий обзор возможных вариантов.

1. Стандартный виджет Qt

В качестве центрального могут использоваться стандартные виджеты, например `QTableWidget` или `QTextEdit`. В данном случае такие функции, как загрузка и сохранение файлов, должны быть реализованы в другом месте (например, в подклассе `QMainWindow`).

2. Пользовательский виджет

В специализированных приложениях часто требуется показывать данные в пользовательском виджете. Например, программа редактирования пиктограмм могла бы в качестве центрального использовать виджет `IconEditor`. В главе 5 рассматриваются способы написания пользовательских виджетов с помощью средств разработки Qt.

3. Базовый виджет `QWidget` с менеджером компоновки

Иногда в центральной области приложения размещается много виджетов. Это можно сделать путем применения `QWidget` в качестве родительского виджета по отношению ко всем другим виджетам и использовать менеджеры компоновки для задания дочерним виджетам их размера и положения.

4. Разделитель

Другой способ размещения в центральной области нескольких виджетов заключается в применении разделителя `QSplitter`. `QSplitter` размещает свои дочерние виджеты по горизонтали или по вертикали и предоставляет пользователю некоторые возможности по управлению размерами виджетов. Разделители могут содержать любые виджеты, включая другие разделители.

5. Область интерфейса MDI

Если в приложении используется интерфейс MDI, центральную область будет занимать виджет `QMdiArea`, а каждое окно интерфейса MDI будет являться дочерним виджетом.

Менеджеры компоновки, разделители и области MDI могут использоваться совместно со стандартными виджетами Qt или с пользовательскими виджетами. В главе 6 подробно рассматриваются эти классы.

В приложении Электронная таблица в качестве центрального виджета применяется некоторый подкласс класса `QTableWidget`. Класс `QTableWidget` уже обеспечивает большинство необходимых нам функций электронной таблицы, но он не может понимать формулы электронной таблицы вида «=A1+A2+A3» и не поддерживает операции с буфером обмена. Мы реализуем эти недостающие функции в классе `Spreadsheet`, который наследует `QTableWidget`.

Создание подкласса `QTableWidget`

Класс `Spreadsheet` происходит от `QTableWidget`, как показано на рис. 4.1. Виджет `QTableWidget` фактически является сеткой, представляющей двумерный разреженный массив. На нем отображается часть ячеек всей сетки, полученная при прокрутке изображения пользователем. При вводе пользователем текста в пустую ячейку `QTableWidget` автоматически создает элемент `QTableWidgetItem` для хранения текста. `QTableWidget` происходит от виджета `QTableView`, одного из классов модели/представления, о котором мы поговорим подробнее в главе 10. Другая таблица, имеющая гораздо большую функциональность, — это `QicsTable`, которую можно получить с сайта <http://www.ics.com/>.

Давайте начнем с реализации виджета `Spreadsheet` и сначала приведем заголовочный файл:

```
#ifndef SPREADSHEET_H
#define SPREADSHEET_H

#include <QTableWidget>

class Cell;
class SpreadsheetCompare;
```

Заголовочный файл начинается с предварительных объявлений классов `Cell` и `SpreadsheetCompare`.

Такие атрибуты ячейки `QTableWidget`, как ее текст и выравнивание, хранятся в `QTableWidgetItem`. В отличие от `QTableWidget`, класс `QTableWidgetItem` не является виджетом; это обычный класс данных. Класс `Cell` происходит от `QTableWidgetItem`, и мы рассмотрим этот класс в последнем разделе данной главы.

```

class Spreadsheet : public QTableWidgetItem
{
    Q_OBJECT

public:
    Spreadsheet(QWidget *parent = 0);

    bool autoRecalculate() const { return autoRecalc; }
    QString currentLocation() const;
    QString currentFormula() const;
    QTableWidgetItemSelectionRange selectedRange() const;
    void clear();
    bool readFile(const QString &fileName);
    bool writeFile(const QString &fileName);
    void sort(const SpreadsheetCompare &compare);

```

Функция `autoRecalculate()` **реализуется как встроенная (inline), поскольку она лишь показывает, задействован или нет режим автоматического перерасчета.**

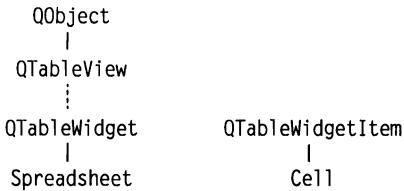


Рис. 4.1. Деревья наследования для классов `Spreadsheet` и `Cell`

В главе 3 мы опирались на использование некоторых открытых функций класса электронной таблицы `Spreadsheet` при реализации `MainWindow`. Например, из `MainWindow::newFile()` мы вызывали функцию `clear()` для очистки электронной таблицы. Кроме того, мы вызывали некоторые функции, унаследованные от `QTableWidgetItem`, а именно `setCurrentCell()` и `setShowGrid()`.

```

public slots:
    void cut();
    void copy();
    void paste();
    void del();
    void selectCurrentRow();
    void selectCurrentColumn();
    void recalculate();
    void setAutoRecalculate(bool recalc);
    void findNext(const QString &str, Qt::CaseSensitivity cs);
    void findPrevious(const QString &str, Qt::CaseSensitivity cs);

signals:
    void modified();

```

Класс Spreadsheet содержит много слотов, которые реализуют действия пунктов меню Edit, Tools и Options, и он содержит один сигнал, modified(), для уведомления о возникновении любого изменения.

```
private slots:
    void somethingChanged();
```

Мы определяем один закрытый слот, который используется внутри класса Spreadsheet.

```
private:
    enum { MagicNumber = 0x7F51C883, RowCount = 999, ColumnCount = 26 };
    Cell *cell(int row, int column) const;
    QString text(int row, int column) const;
    QString formula(int row, int column) const;
    void setFormula(int row, int column, const QString &formula);

    bool autoRecalc;
};
```

В закрытой секции этого класса мы объявляем три константы, четыре функции и одну переменную.

```
class SpreadsheetCompare
{
public:
    bool operator()(const QStringList &row1,
                   const QStringList &row2) const;
    enum { KeyCount = 3 };
    int keys[KeyCount];
    bool ascending[KeyCount];
};

#endif
```

Заголовочный файл заканчивается определением класса SpreadsheetCompare. Мы объясним назначение этого класса при рассмотрении функции Spreadsheet::sort().

Теперь мы рассмотрим реализацию:

```
#include <QtGui>
#include "cell.h"
#include "spreadsheet.h"

Spreadsheet::Spreadsheet(QWidget *parent)
    : QTableWidgetItem(parent)
{
    autoRecalc = true;

    setItemPrototype(new Cell);
    setSelectionMode(ContiguousSelection);
```

```

connect(this, SIGNAL(itemChanged(QTableWidgetItem *)),
        this, SLOT(somethingChanged()));

clear();
}

```

Обычно при вводе пользователем некоторого текста в пустую ячейку `QTableWidgetItem` будет автоматически создавать элемент `QTableWidgetItem` для хранения этого текста. Вместо этого мы хотим, чтобы создавались элементы `Cell`. Это достигается с помощью вызова в конструкторе функции `setItemPrototype()`. Всякий раз, когда требуется новый элемент, `QTableWidgetItem` дублирует элемент, переданный в качестве прототипа.

Кроме того, в конструкторе мы устанавливаем режим выделения области на значение `QAbstractItemView::ContiguousSelection`, чтобы могла быть выделена только одна прямоугольная область. Мы соединяем сигнал `itemChanged()` виджета таблицы с закрытым слотом `somethingChanged()`; это гарантирует вызов слота `somethingChanged()` при редактировании ячейки пользователем. Наконец, мы вызываем `clear()` для изменения размеров таблицы и задания заголовков столбцов.

```

void Spreadsheet::clear()
{
    setRowCount(0);
    setColumnCount(0);
    setRowCount(RowCount);
    setColumnCount(ColumnCount);

    for (int i = 0; i < ColumnCount; ++i) {
        QTableWidgetItem *item = new QTableWidgetItem;
        item->setText(QString(QChar('A' + i)));
        setHorizontalHeaderItem(i, item);
    }
    setCurrentCell(0, 0);
}

```

Функция `clear()` вызывается из конструктора `Spreadsheet` для инициализации электронной таблицы. Она также вызывается из `MainWindow::newFile()`.

Мы могли бы использовать `QTableWidgetItem::clear()` для очистки всех элементов и любых выделений, но в этом случае заголовки имели бы текущий размер. Вместо этого мы уменьшаем размер электронной таблицы до 0×0 . Это приводит к очистке всей электронной таблицы, включая заголовки. Затем мы опять устанавливаем ее размер на `ColumnCount × RowCount` (26×999) и заполняем строку горизонтального заголовка элементами `QTableWidgetItem`, содержащими обозначения столбцов. Нам не надо задавать метки строк, потому что по умолчанию строки обозначаются как «1», «2», ..., «26». В конце мы перемещаем курсор на ячейку A1.

`QTableWidgetItem` содержит несколько дочерних виджетов. Сверху располагается горизонтальный заголовок `QHeaderView`, слева – вертикальный заголовок `QHeaderView` и две полосы прокрутки `QScrollBar`. В центральной области размещается специальный виджет, называемый областью отображения (`viewport`), в котором `QTableWidgetItem` вычерчивает ячейки. Доступ к различным дочерним

виджетам осуществляется с помощью функций, унаследованных от `QTableView` и `QAbstractScrollArea` (рис. 4.2). `QAbstractScrollArea` содержит перемещаемую область отображения и две полосы прокрутки, которые могут включаться и отключаться. Подкласс `QScrollArea` мы рассматриваем в главе 6.

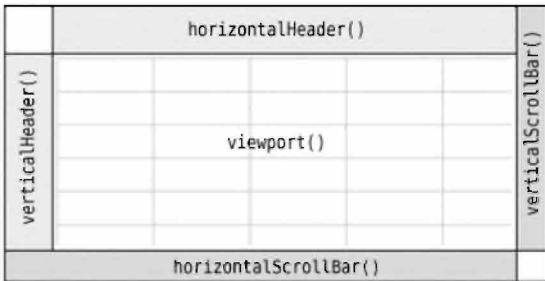


Рис. 4.2. Виджеты, составляющие `QTableWidget`

```
Cell *Spreadsheet::cell(int row, int column) const
{
    return static_cast<Cell *>(item(row, column));
}
```

Закрытая функция `cell()` возвращает для заданной строки и столбца объект `Cell`. Она работает почти так же, как `QTableWidget::item()`, но возвращает указатель на `Cell`, а не указатель на `QTableWidgetItem`.

```
QString Spreadsheet::text(int row, int column) const
{
    Cell *c = cell(row, column);
    if (c) {
        return c->text();
    } else {
        return "";
    }
}
```

Закрытая функция `text()` возвращает формулу заданной ячейки. Если `cell()` возвращает нулевой указатель, то это означает, что ячейка пустая, и поэтому мы возвращаем пустую строку.

```
QString Spreadsheet::formula(int row, int column) const
{
    Cell *c = cell(row, column);
    if (c) {
        return c->formula();
    } else {
        return "";
    }
}
```

Функция `formula()` возвращает формулу ячейки. Во многих случаях формула и текст совпадают; например, формула «Hello» соответствует строке «Hello», поэтому при вводе пользователем в ячейку строки «Hello» и нажатии клавиши Enter в ячейке отобразится текст «Hello». Но имеется несколько исключений.

- Если формула представлена числом, именно оно и будет отображаться. Например, формула «1.50» обозначает значение 1.5 типа `double`, которое отображается в электронной таблице как выровненное вправо значение «1.5».
- Если формула начинается с одиночной кавычки, остальная часть формулы интерпретируется как текст. Например, результатом формулы «'12345» будет строка «12345».
- Если формула начинается со знака равенства («=»), то ее значение интерпретируется как арифметическое выражение. Например, если ячейка A1 содержит «12» и ячейка A2 содержит «6», то результатом формулы «=A1+A2» будет 18.

Хранение данных в объектах типа «элемент»

В приложении Электронная таблица каждая непустая ячейка хранится в памяти в виде одного объекта `QTableWidgetItem` (элемент табличного виджета). Хранение данных в объектах типа «элемент» используется также виджетами `QListWidget` и `QTreeWidget`, которые работают с объектами `QListWidgetItem` и `QTreeWidgetItem`.

В Qt классы элементов могут использоваться вне таблиц как самостоятельные структуры данных. Например, `QTableWidgetItem` уже содержит некоторые атрибуты, в том числе строку, шрифт, цвет и пиктограмму, а также обратный указатель на `QTableWidget`. Такие элементы могут содержать также данные типа `QVariants`, включая зарегистрированные пользовательские типы, и создавая подкласс такого элемента, можно обеспечить дополнительную функциональность.

Другие инструментальные средства предусматривают наличие в классах элементов указателя типа `void` для хранения пользовательских данных. В Qt используется более естественный подход с применением `setData()` для типа `QVariant`, однако если требуется иметь указатель `void`, это можно сделать просто путем создания подкласса для класса элемента, который будет содержать переменную-указатель на член типа `void`.

Для данных, к которым предъявляются повышенные требования, например для больших наборов данных, для сложных элементов данных, для интеграции баз данных и для множественных представлений данных, Qt предоставляет набор классов «модель/представление», в которых данные отделены от их визуального представления. Эти классы рассматриваются в главе 10.

Задача преобразования формулы в значение выполняется классом `Cell`. Здесь следует иметь в виду, что отображаемый в ячейке текст соответствует значению, полученному в результате расчета формулы, а не является текстом самой формулы.

```
void Spreadsheet::setFormula(int row, int column, const QString &formula)
{
    Cell *c = cell(row, column);
```



```

    if (!c) {
        c = new Cell;
        setItem(row, column, c);
    }
    c->setFormula(formula);
}

```

Закрытая функция `setFormula()` задает формулу для указанной ячейки. Если ячейка уже имеет объект `Cell`, мы его повторно используем. В противном случае мы создаем новый объект `Cell` и вызываем `QTableWidget::setItem()` для вставки его в таблицу. В конце мы вызываем для этой ячейки функцию `setFormula()`, что приводит к перерисовке ячейки, если она отображается на экране. Нам не надо беспокоиться об удалении в будущем объекта `Cell`; `QTableWidget` является собственником ячейки и будет автоматически удалять ее содержимое в нужное время.

```

QString Spreadsheet::currentLocation() const
{
    return QChar('A' + currentColumn())
        + QString::number(currentRow() + 1);
}

```

Функция `currentLocation()` возвращает текущее положение ячейки, используя обычную форму представления ее координат в электронной таблице с обозначением буквой положения столбца, за которой идет номер строки. Функция `MainWindow::updateStatusBar()` использует ее для отображения положения ячейки в строке состояния.

```

QString Spreadsheet::currentFormula() const
{
    return formula(currentRow(), currentColumn());
}

```

Функция `currentFormula()` возвращает формулу текущей ячейки. Она вызывается из функции `MainWindow::updateStatusBar()`.

```

void Spreadsheet::somethingChanged()
{
    if (autoRecalc)
        recalculate();
    emit modified();
}

```

Закрытый слот `somethingChanged()` делает перерасчет всей электронной таблицы, если включен режим `Auto-Recalculate` (автоматический пересчет). Он также генерирует сигнал `modified()`.

Загрузка и сохранение

Теперь мы реализуем загрузку и сохранение файла данных для приложения Электронная таблица, используя двоичный пользовательский формат. Для этого мы используем объекты `QFile` и `QDataStream`, которые совместно обеспечивают независимый от платформы ввод-вывод в двоичном формате.

Мы начнем с записи файла данных Электронная таблица:

```
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QIODevice::WriteOnly)) {
        QMessageBox::warning(this, tr("Spreadsheet"),
                               tr("Cannot write file %1:\n%2.")
                               .arg(file.fileName())
                               .arg(file.errorString()));

        return false;
    }

    QDataStream out(&file);
    out.setVersion(QDataStream::Qt_4_3);
    out << quint32(MagicNumber);

    QApplication::setOverrideCursor(Qt::WaitCursor);
    for (int row = 0; row < RowCount; ++row) {
        for (int column = 0; column < ColumnCount; ++column) {
            QString str = formula(row, column);
            if (!str.isEmpty())
                out << quint16(row) << quint16(column) << str;
        }
    }
    QApplication::restoreOverrideCursor();
    return true;
}
```

Функция `writeFile()` вызывается из `MainWindow::saveFile()` для записи файла на диск. Она возвращает `true` при успешном завершении и `false` при ошибке.

Мы создаем объект `QFile`, задавая имя файла, и вызываем функцию `open()` для открытия файла для записи данных. Мы также создаем объект `QDataStream`, который предназначен для работы с `QFile` и использует его для записи данных.

Непосредственно перед записью данных мы изменяем курсор приложения на стандартный курсор ожидания (обычно он имеет вид песочных часов) и затем восстанавливаем нормальный курсор после окончания записи данных. В конце функции файл автоматически закрывается деструктором `QFile`.

`QDataStream` поддерживает основные типы C++ совместно со многими типами Qt. Их синтаксис напоминает синтаксис классов `<iostream>` стандартного C++. Например,

```
out << x << y << z;
```

выполняет запись в поток значений переменных `x`, `y` и `z`, а

```
in >> x >> y >> z;
```

считывает их из потока. Поскольку элементарные целочисленные типы C++ на различных платформах могут иметь различный размер, надежнее преобразовать их типы в `qint8`, `quint8`, `qint16`, `quint16`, `qint32`, `quint32`, `qint64` и `quint64`, что гарантирует использование объявленного в них размера (в битах).

Файл данных Электронная таблица имеет очень простой формат. Он начинается с 32-битового числа, идентифицирующего формат файла («волшебное» число `MagicNumber` определено в `spreadsheet.h` как `0x7F51C883` – произвольное случайное число). Затем идет последовательность блоков, содержащих строку, столбец и формулу одной ячейки. Для экономии места мы не записываем пустые ячейки. Формат показан на рис. 4.3.



Рис. 4.3. Формат файла данных для приложения Электронная таблица

Точно представление типов данных определяется в `QDataStream`. Например, `quint16` представляется двумя байтами со старшим байтом в конце, а `QString` задается длиной строки, за которой следуют символы в коде Unicode.

Двоичное представление типов в Qt достаточно сильно усовершенствовалось со времени выхода версии Qt 1.0. Такая тенденция, вероятно, сохранится в будущих версиях Qt, чтобы идти вровень с развитием существующих типов и обеспечить новые типы в Qt. По умолчанию класс `QDataStream` использует самую последнюю версию двоичного формата (версия 9 в Qt 4.3), но он также может быть настроен на чтение прошлых версий. Для того чтобы избежать проблем совместимости при перекомпиляции приложения в будущем, в новой версии Qt мы заставляем `QDataStream` использовать версию 9 вне зависимости от версии Qt, в которой оно компилируется. (Для удобства используется константа `QDataStream::Qt_4_3`, равная 9).

Класс `QDataStream` достаточно универсален. Он может использоваться для объекта `QFile`, но также и для `QBuffer`, `QProcess`, `QTcpSocket`, `QUdpSocket` или `QSslSocket`. Qt также предоставляет класс `QTextStream`, который может использоваться с `QDataStream` для чтения и записи текстовых файлов. В главе 10 подробно рассматриваются эти классы и описываются различные методы работы с разными версиями `QDataStream`.

```
bool Spreadsheet::readFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QIODevice::ReadOnly)) {
        QMessageBox::warning(this, tr("Spreadsheet"),
                               tr("Cannot read file %1:\n%2.")
                               .arg(file.fileName())
                               .arg(file.errorString()));
        return false;
    }

    QDataStream in(&file);
    in.setVersion(QDataStream::Qt_4_3);

    quint32 magic;
    in >> magic;
```

```

if (magic != MagicNumber) {
    QMessageBox::warning(this, tr("Spreadsheet"),
                        tr("The file is not a Spreadsheet file.));
    return false;
}

clear();

quint16 row;
quint16 column;
QString str;

QApplication::setOverrideCursor(Qt::WaitCursor);
while (!in.atEnd()) {
    in >> row >> column >> str;
    setFormula(row, column, str);
}
QApplication::restoreOverrideCursor();
return true;
}

```

Функция `readFile()` очень напоминает `writeFile()`. Для чтения файла мы пользуемся объектом `QFile`, но теперь используем флажок `QIODevice::ReadOnly`, а не `QIODevice::WriteOnly`. Затем мы устанавливаем версию `QDataStream` на значение 9. Формат чтения всегда должен совпадать с форматом записи.

Если в начале файла содержится правильное «волшебное» число, мы вызываем функцию `clear()` для очистки в электронной таблице всех ячеек и затем считываем данные ячеек. Поскольку файл содержит только данные для непустых ячеек, маловероятно, что будет заполнена каждая ячейка электронной таблицы, поэтому мы должны очистить все ячейки перед чтением файла.

Реализация меню Edit

Теперь мы готовы приступить к реализации слотов, относящихся к меню Edit данного приложения. Это меню показано на рис. 4.4.

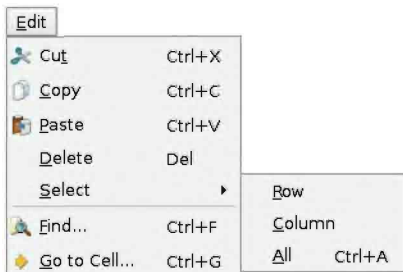


Рис. 4.4. Меню Edit приложения Электронная таблица

```
void Spreadsheet::cut()
{
    copy();
    del();
}
```

Слот `cut()` соответствует пункту меню **Edit | Cut (Правка | Вырезать)**. Он реализуется просто, поскольку операция **Cut** выполняется с помощью операции **Copy**, за которой следует операция **Delete**.

```
void Spreadsheet::copy()
{
    QTableWidgetItemSelectionRange range = selectedRange();
    QString str;

    for (int i = 0; i < range.rowCount(); ++i) {
        if (i > 0)
            str += "\n";
        for (int j = 0; j < range.columnCount(); ++j) {
            if (j > 0)
                str += "\t";
            str += formula(range.topRow() + i, range.leftColumn() + j);
        }
    }
    QApplication::clipboard()->setText(str);
}
```

Слот `copy()` соответствует пункту меню **Edit|Copy (Правка | Копировать)**. Он в цикле обрабатывает всю выделенную область ячеек (если нет явно выделенной области, то ею будет просто текущая ячейка). Формула каждой выделенной ячейки добавляется в `QString`, причем строки отделяются символом новой строки, а столбцы разделяются символом табуляции. Иллюстрация приводится на рис. 4.5.

Доступ к буферу обмена в Qt осуществляется при помощи статической функции `QApplication::clipboard()`. Вызывая функцию `QClipboard::setText()`, мы делаем текст доступным через буфер обмена; причем этот текст могут использовать и данное, и другие приложения, поддерживающие работу с простыми текстами. Применяемый нами формат со знаками табуляции и новой строки в качестве разделителей понятен многими приложениями, включая Excel от компании Microsoft.

	C	D	E
2	Red	Green	Blue
3	Cyan	Magenta	Yellow



"Red\tGreen\tBlue\nCyan\tMagenta\tYellow"

Рис. 4.5. Копирование выделенных ячеек в буфер обмена

Функция `QTableWidgetItem::selectedRange()` возвращает список выделенных диапазонов. Мы знаем, что может быть не более одного диапазона, потому что мы задали в конструкторе режим выделения `QAbstractItemView::ContiguousSelection`.

Для удобства мы определяем функцию `selectedRange()`, которая возвращает выделенный диапазон:

```
QTableWidgetSelectionRange Spreadsheet::selectedRange() const
{
    QList<QTableWidgetSelectionRange> ranges = selectedRanges();
    if (ranges.isEmpty())
        return QTableWidgetSelectionRange();
    return ranges.first();
}
```

Если выделение вообще имеет место, мы возвращаем первую (и единственную) выделенную область. Какая-нибудь область всегда должна быть выбрана, поскольку в режиме `ContiguousSelection` текущая ячейка рассматривается как выделенная. Однако такую ситуацию мы все же обрабатываем, чтобы защититься от ошибки в нашей программе, приводящей к отсутствию текущей ячейки.

```
void Spreadsheet::paste()
{
    QTableWidgetSelectionRange range = selectedRange();
    QString str = QApplication::clipboard()->text();
    QStringList rows = str.split('\n');
    int numRows = rows.count();
    int numColumns = rows.first().count('\t') + 1;

    if (range.rowCount() * range.columnCount() != 1
        && (range.rowCount() != numRows
            || range.columnCount() != numColumns)) {
        QMessageBox::information(this, tr("Spreadsheet"),
            tr("The information cannot be pasted because the copy "
               "and paste areas aren't the same size."));
        return;
    }

    for (int i = 0; i < numRows; ++i) {
        QStringList columns = rows[i].split('\t');
        for (int j = 0; j < numColumns; ++j) {
            int row = range.topRow() + i;
            int column = range.leftColumn() + j;
            if (row < RowCount && column < ColumnCount)
                setFormula(row, column, columns[j]);
        }
    }
    somethingChanged();
}
```

Слот `paste()` соответствует пункту меню `Edit|Paste (Правка | Вставить)`. Мы считываем текст из буфера обмена и вызываем статическую функцию `QString::split()` для разбиения строки и представления ее в виде списка `QStringList`. Каждая строка таблицы представлена в этом списке одной строкой.

Затем мы определяем размеры области копирования. Номер строки в таблице является номером строки в QStringList; номер столбца является номером символа табуляции в первой строке плюс 1. Если выделена только одна ячейка, мы используем ее в качестве верхнего левого угла области вставки; в противном случае мы используем текущую выделенную область для вставки.

При выполнении операции вставки мы в цикле проходим по строкам и разбиваем каждую строку на значения ячеек, снова используя функцию QString::split(), но теперь в качестве разделителя применяется знак табуляции. Рис. 4.6 иллюстрирует эти действия.

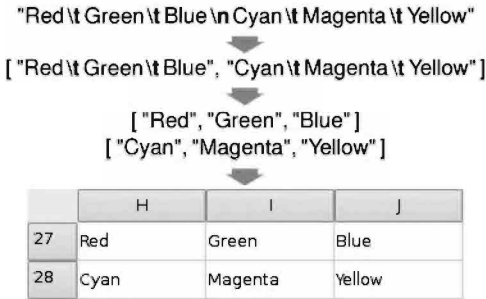


Рис. 4.6. Вставка текста из буфера обмена в электронную таблицу

```
void Spreadsheet::del()
{
    QList<QTableWidgetItem *> items = selectedItems();
    if (!items.isEmpty()) {
        foreach (QTableWidgetItem *item, items)
            delete item;
        somethingChanged();
    }
}
```

Слот del() соответствует пункту меню Edit|Delete (Правка | Удалить). Если есть выделенные ячейки, они удаляются, и вызывается функция somethingChanged(). Для очистки ячеек достаточно использовать оператор delete для каждого объекта Cell. Объект QTableWidgetItem замечает, когда удаляются его элементы QTableWidgetItem, и автоматически перерисовывает себя, если какой-нибудь из элементов оказывается видимым. Если мы вызываем функцию cell(), указывая координаты удаленной ячейки, то она возвратит нулевой указатель.

```
void Spreadsheet::selectCurrentRow()
{
    selectRow(currentRow());
}

void Spreadsheet::selectCurrentColumn()
{
    selectColumn(currentColumn());
}
```

Функции `selectCurrentRow()` и `selectCurrentColumn()` соответствуют пунктам меню `Edit|Select|Row` и `Edit|Select|Column` (**Правка | Выделить | Строка** и **Правка | Выделить | Столбец**). Здесь используется реализация функций `selectRow()` и `selectColumn()` класса `QTableWidget`. Нам не требуется реализовывать функциональность пункта меню `Edit|Select|All` (**Правка | Выделить | Все**), поскольку она обеспечивается в `QTableWidget` унаследованной функцией `QAbstractItemView::selectAll()`.

```
void Spreadsheet::findNext(const QString &str, Qt::CaseSensitivity cs)
{
    int row = currentRow();
    int column = currentColumn() + 1;

    while (row < RowCount) {
        while (column < ColumnCount) {
            if (text(row, column).contains(str, cs)) {
                clearSelection();
                setCurrentCell(row, column);
                activateWindow();
                return;
            }
            ++column;
        }
        column = 0;
        ++row;
    }
    QApplication::beep();
}
```

Слот `findNext()` в цикле просматривает ячейки, начиная с ячейки, расположенной правее курсора, и двигается вправо до достижения последнего столбца; затем процесс идет с первого столбца строки, расположенной ниже, и так продолжается, пока не будет найден требуемый текст или пока не будет достигнута самая последняя ячейка. Например, если текущей является ячейка C24, поиск будет продолжаться по ячейкам D24, E24, ..., Z24, затем по A25, B25, C25, ..., Z25 и т. д., пока не будет достигнута ячейка Z999. Если соответствующее значение найдено, мы сбрасываем текущее выделение и перемещаем курсор на ячейку, в которой оно находится, и делаем активным окно, содержащее эту электронную таблицу `Spreadsheet`. При неудачном завершении поиска мы заставляем приложение выдать соответствующий звуковой сигнал.

```
void Spreadsheet::findPrevious(const QString &str,
                               Qt::CaseSensitivity cs)
{
    int row = currentRow();
    int column = currentColumn() - 1;

    while (row >= 0) {
        while (column >= 0) {
            if (text(row, column).contains(str, cs)) {
                clearSelection();
            }
        }
        row--;
        column = ColumnCount - 1;
    }
}
```



```

        setCurrentCell(row, column);
        activateWindow();
        return;
    }
    --column;
}
column = ColumnCount - 1;
--row;
}
QApplication::beep();
}

```

Слот `findPrevious()` похож на `findNext()`, но здесь цикл выполняется в обратном направлении и заканчивается в ячейке A1.

Реализация других меню

Теперь мы реализуем слоты для пунктов меню `Tools` и `Options`. Эти меню показаны на рис. 4.7.



Рис. 4.7. Меню `Tools` и `Options` приложения Электронная таблица

```

void Spreadsheet::recalculate()
{
    for (int row = 0; row < RowCount; ++row) {
        for (int column = 0; column < ColumnCount; ++column) {
            if (cell(row, column))
                cell(row, column)->setDirty();
        }
    }
    viewport()->update();
}

```

Слот `recalculate()` соответствует пункту меню `Tools|Recalculate` (Инструменты | Пересчитать). Он также вызывается в `Spreadsheet` автоматически по мере необходимости.

Мы выполняем цикл по всем ячейкам и вызываем функцию `setDirty()`, которая помечает каждую из них для перерасчета значения. Следующий раз, когда `QTableWidget` для получения отображаемого в электронной таблице значения вызовет `text()` для некоторой ячейки `Cell`, значение этой ячейки будет пересчитано.

Затем мы вызываем для области отображения функцию `update()` для перерисовки всей электронной таблицы. При этом используемый в `QTableWidget` программный код по перерисовке вызывает функцию `text()` для каждой видимой ячейки для получения отображаемого значения. Поскольку функция `setDirty()`

вызывалась нами для каждой ячейки, в вызовах `text()` будет использовано новое рассчитанное значение. В этом случае может потребоваться расчет невидимых ячеек, который будет проводиться до тех пор, пока не будут рассчитаны все ячейки, влияющие на правильное отображение текста в перерассчитанной области отображения. Этот расчет выполняется в классе `Cell`.

```
void Spreadsheet::setAutoRecalculate(bool recalc)
{
    autoRecalc = recalc;
    if (autoRecalc)
        recalculate();
}
```

Слот `setAutoRecalculate()` соответствует пункту меню `Options|Auto-Recalculate`. Если эта опция включена, мы сразу же пересчитаем всю электронную таблицу и будем уверены, что она показывает обновленные значения; впоследствии функция `recalculate()` будет автоматически вызываться из `somethingChanged()`.

Нам не нужно реализовывать специальную функцию для пункта меню `Options|Show Grid`, поскольку в `QTableWidget` уже содержится слот `setShowGrid()`, который наследуется от `QTableView`. Остается только реализовать функцию `Spreadsheet::sort()`, которая вызывается из `MainWindow::sort()`:

```
void Spreadsheet::sort(const SpreadsheetCompare &compare)
{
    QList<QStringList> rows;
    QTableWidgetSelectionRange range = selectedRange();
    int i;
    for (i = 0; i < range.rowCount(); ++i) {
        QStringList row;
        for (int j = 0; j < range.columnCount(); ++j)
            row.append(formula(range.topRow() + i,
                               range.leftColumn() + j));
        rows.append(row);
    }
    qStableSort(rows.begin(), rows.end(), compare);
    for (i = 0; i < range.rowCount(); ++i) {
        for (int j = 0; j < range.columnCount(); ++j)
            setFormula(range.topRow() + i, range.leftColumn() + j,
                      rows[i][j]);
    }
    clearSelection();
    somethingChanged();
}
```

Сортировка работает на текущей выделенной области и переупорядочивает строки в соответствии со значениями ключей порядка сортировки и хранящимися в объекте `compare`. Мы представляем каждую строку данных в `QStringList`, а выделенную область храним в виде списка строк. Мы используем алгоритм `Qt qStableSort()` и для простоты сортируем по выражениям формул, а не по их значениям. Процесс иллюстрируется на рис. 4.8 и 4.9. Стандартные алгоритмы и структуры данных `Qt` мы рассматриваем в главе 11.

	C	D	E
2	Edsger	Dijkstra	1930-05-11
3	Tony	Hoare	1934-01-11
4	Niklaus	Wirth	1934-02-15
5	Donald	Knuth	1938-01-10

index	value
0	["Edsger", "Dijkstra", "1930-05-11"]
1	["Tony", "Hoare", "1934-01-11"]
2	["Niklaus", "Wirth", "1934-02-15"]
3	["Donald", "Knuth", "1938-01-10"]

Рис. 4.8. Хранение выделенной области в виде списка строк

В качестве аргументов функции `qStableSort()` используется итератор начала, итератор конца и функция сравнения. Функция сравнения имеет два аргумента (оба имеют тип `QStringLists`), и она возвращает `true`, когда первый аргумент «больше, чем» второй аргумент, и `false` в противном случае. Передаваемый как функция сравнения объект `compare` фактически не является функцией, но он может использоваться и в таком качестве, в чем мы вскоре сможем убедиться.

index	value
0	["Donald", "Knuth", "1938-01-10"]
1	["Edsger", "Dijkstra", "1930-05-11"]
2	["Niklaus", "Wirth", "1934-02-15"]
3	["Tony", "Hoare", "1934-01-11"]

	C	D	E
2	Donald	Knuth	1938-01-10
3	Edsger	Dijkstra	1930-05-11
4	Niklaus	Wirth	1934-02-15
5	Tony	Hoare	1934-01-11

Рис. 4.9. Помещение данных в таблицу после сортировки

После выполнения функции `qStableSort()` мы помещаем данные обратно в таблицу, сбрасываем выделение области и вызываем функцию `somethingChanged()`.

Класс `SpreadsheetCompare` в `spreadsheet.h` определен следующим образом:

```
class SpreadsheetCompare
{
public:
    bool operator()(const QStringList &row1,
                   const QStringList &row2) const;

    enum { KeyCount = 3 };
    int keys[KeyCount];
    bool ascending[KeyCount];
};
```

Класс `SpreadsheetCompare` является специальным классом, реализующим оператор `()`. Это позволяет нам применять этот класс в качестве функции. Такие классы называются объектами функций или функторами (functors). Для лучшего понимания работы функторов мы сначала разберем простой пример.

```
class Square
{
public:
    int operator()(int x) const { return x * x; }
};
```

Класс `Square` содержит одну функцию, `operator()(int)`, которая возвращает квадрат переданного ей значения параметра. Обозначая функцию в виде `operator()(int)`, а не в виде, например, `compute(int)`, мы получаем возможность применения объекта типа `Square` как функции:

```
Square square;
int y = square(5);
//y равно 25
```

Теперь рассмотрим пример с применением объекта `SpreadsheetCompare`.

```
QStringList row1, row2;
QSpreadsheetCompare compare;
...
if (compare(row1, row2)) {
    // строка row1 меньше, чем row2
}
```

Объект `compare` можно использовать так же, как если бы он был обычной функцией `compare()`. Кроме того, он может быть реализован таким образом, что будет осуществлять доступ ко всем ключам сортировки и всем признакам порядка сортировки, которые хранятся в переменных-членах класса.

Можно использовать другой подход, когда ключи сортировки и признаки порядка сортировки хранятся в глобальных переменных и используется функция обычного типа `compare()`. Однако связь через глобальные переменные выглядит не изящно и может быть причиной тонких ошибок. Функторы представляют собой более мощное средство связи для таких функций-шаблонов, как `qStableSort()`.

Ниже приводится реализация функции, которая применяется для сравнения двух строк электронной таблицы.

```
bool SpreadsheetCompare::operator()(const QStringList &row1,
                                     const QStringList &row2) const
{
    for (int i = 0; i < KeyCount; ++i) {
        int column = keys[i];
        if (column != -1) {
            if (row1[column] != row2[column]) {
                if (ascending[i]) {
                    return row1[column] < row2[column];
                } else {
                    return row1[column] > row2[column];
                }
            }
        }
    }
    return false;
}
```

Этот оператор возвращает `true`, если первая строка меньше второй; в противном случае он возвращает `false`. Функция `qStableSort()` для выполнения сортировки использует результат этой функции.

Массивы `keys` и `ascending` объекта `SpreadsheetCompare` заполняются при работе функции `MainWindow::sort()` (она приводится в главе 2). Каждый ключ содержит индекс столбца или имеет значение `-1` («None» – нет значения).

Мы сравниваем значения соответствующих ячеек двух строк, учитывая порядок ключей сортировки. Как только оказывается, что они различны, мы возвращаем соответствующее значение: `true` или `false`. Если все значения оказываются равными, мы возвращаем `false`. При совпадении значений функция `qStableSort()` сохраняет порядок до сортировки; если строка `row1` располагалась первоначально перед строкой `row2` и ни одна из них не оказалась «меньше другой», то в результате строка `row1` по-прежнему будет предшествовать строке `row2`. Именно этим функция `qStableSort()` отличается от своего нестабильного «родственника» `qSort()`.

Теперь мы закончили класс `Spreadsheet`. В следующем разделе мы рассмотрим класс `Cell`. Этот класс применяется для хранения формул ячеек и обеспечивает переопределение функции `QTableWidgetItem::data()`, которая вызывается в `Spreadsheet` через функцию `QTableWidgetItem::text()` для отображения результата вычисления формулы ячейки.

Создание подкласса QTableWidgetItem

Класс `Cell` наследует `QTableWidgetItem`. Этот класс спроектирован для удобства работы с `Spreadsheet`, но он не имеет никаких особых связей с данным классом электронной таблицы и теоретически может применяться для любого объекта `QTableWidgetItem`. Ниже приводится заголовочный файл.

```
#ifndef CELL_H
#define CELL_H

#include <QTableWidgetItem>

class Cell : public QTableWidgetItem
{
public:
    Cell();

    QTableWidgetItem *clone() const;
    void setData(int role, const QVariant &value);
    QVariant data(int role) const;
    void setFormula(const QString &formula);
    QString formula() const;
    void setDirty();

private:
    QVariant value() const;
    QVariant evalExpression(const QString &str, int &pos) const;
    QVariant evalTerm(const QString &str, int &pos) const;
    QVariant evalFactor(const QString &str, int &pos) const;
    mutable QVariant cachedValue;
    mutable bool cacheIsDirty;
};

#endif
```

Класс `Cell` расширяет `QTableWidgetItem`, добавляя две закрытые переменные:

- переменная `cachedValue` кеширует значение ячейки в виде значения типа `QVariant`;
- переменная `cacheIsDirty` принимает значение `true`, если кешируемое значение устарело.

Мы используем `QVariant`, поскольку некоторые ячейки имеют тип числа двойной точности `double`, а другие имеют тип строки `QString`.

При объявлении переменных `cachedValue` и `cacheIsDirty` используется ключевое слово `mutable` языка C++. Это позволяет нам модифицировать эти переменные в функциях с модификатором `const`. Мы могли бы поступить по-другому и заново выполнять расчет при каждом вызове функции `text()`, но эта неэффективность будет не оправдана.

Следует отметить, что в определении класса не используется макрос `Q_OBJECT`. Класс `Cell` является «чистым» классом C++, который не имеет сигналов и слотов. На самом деле из-за того, что `QTableWidgetItem` не происходит от `QObject`, мы не можем использовать в `Cell` как таковые сигналы и слоты. Классы элементов Qt не наследуют `QObject`, чтобы свести к минимуму затраты на их обработку. Если сигналы и слоты необходимы, они могут быть реализованы в виджете, содержащем элементы, или (в виде исключения) при помощи множественного наследования класса `QObject`.

Теперь мы перейдем к написанию `cell.cpp`:

```
#include <QtGui>

#include "cell.h"

Cell::Cell()
{
    setDirty();
}
```

В конструкторе нам необходимо установить признак «dirty» («грязный») только для кеша. Передавать родительский объект нет необходимости; когда делается вставка ячейки в `QTableWidgetItem` с помощью `setItem()`; `QTableWidgetItem` автоматически станет ее владельцем.

Каждый элемент `QTableWidgetItem` может иметь некоторые данные — до одного типа `QVariant` на каждую «роль» данных. Наиболее распространенными ролями являются `Qt::EditRole` и `Qt::DisplayRole` (роль правки и роль отображения). Роль правки используется для данных, которые должны редактироваться, а роль отображения, для данных, которые должны отображаться на экране. Часто обе роли используются для одних и тех же данных, однако в `Cell` роль правки соответствует формуле ячейки, а роль отображения — значению ячейки (результату вычисления формулы).

```
QTableWidgetItem *Cell::clone() const
{
    return new Cell(*this);
}
```

Функция `clone()` вызывается в `QTableWidgetItem`, когда необходимо создать новую ячейку, например, когда пользователь начинает вводить данные в пустую ячейку, которая до сих пор не использовалась. Переданный функции `QTableWidgetItem::`

`setItemPrototype()` экземпляр является дубликатом. Поскольку для копирования `Cell` можно ограничиться функцией-членом, мы полагаемся на используемый по умолчанию конструктор копирования, автоматически создаваемый C++ при создании экземпляров новых ячеек `Cell` в функции `clone()`.

```
void Cell::setFormula(const QString &formula)
{
    setData(Qt::EditRole, formula);
}
```

Функция `setFormula()` задает формулу ячейки. Это просто удобная функция для вызова `setData()` с указанием роли правки. Она вызывается из функции `Spreadsheet::setFormula()`.

```
QString Cell::formula() const
{
    return data(Qt::EditRole).toString();
}
```

Функция `formula()` вызывается из `Spreadsheet::formula()`. Подобно `setFormula()` этой функцией удобно пользоваться на этот раз для получения данных `EditRole` заданного элемента.

```
void Cell::setData(int role, const QVariant &value)
{
    QTableWidgetItem::setData(role, value);
    if (role == Qt::EditRole)
        setDirty();
}
```

Если мы имеем новую формулу, мы устанавливаем `cacheIsDirty` на значение `true`, чтобы обеспечить перерасчет ячейки при последующем вызове `text()`.

В `Cell` нет определения функции `text()`, хотя мы и вызываем `text()` для экземпляров `Cell` в функции `Spreadsheet::text()`. `QTableWidgetItem` содержит удобную функцию `text()`, которая эквивалентна вызову `data(Qt::DisplayRole).toString()`.

```
void Cell::setDirty()
{
    cacheIsDirty = true;
}
```

Функция `setDirty()` вызывается для принудительного перерасчета значения ячейки. Она просто устанавливает флажок `cacheIsDirty` на значение `true`, указывая на то, что значение `cachedValue` больше не отражает текущее состояние. Перерасчет не будет выполняться до тех пор, пока он не станет действительно необходим.

```
QVariant Cell::data(int role) const
{
    if (role == Qt::DisplayRole) {
        if (value().isValid()) {
            return value().toString();
        } else {
            return "####";
        }
    }
}
```

```

    } else if (role == Qt::TextAlignmentRole) {
        if (value().type() == QVariant::String) {
            return int(Qt::AlignLeft | Qt::AlignVCenter);
        } else {
            return int(Qt::AlignRight | Qt::AlignVCenter);
        }
    } else {
        return QTableWidgetItem::data(role);
    }
}

```

Функция `data()` класса `QTableWidgetItem` переопределяется. Она возвращает текст, который должен отображаться в электронной таблице, если в вызове указана роль `Qt::DisplayRole`, или формулу, если в вызове указана роль `Qt::EditRole`. Она обеспечивает подходящее выравнивание, если вызывается с ролью `Qt::TextAlignmentRole`. При задании роли `DisplayRole`, она использует функцию `value()` для расчета значения ячейки. Если нельзя получить достоверное значение (из-за того, что формула неверна), мы возвращаем значение «####».

Функция `Cell::value()`, используемая в `data()`, возвращает значение типа `QVariant`. Объекты типа `QVariant` могут содержать значения различных типов, например `double` или `QString`, и поддерживают функции для преобразования их в другие типы. Например, при вызове `toString()` для переменной типа `QVariant`, содержащей значение типа `double`, в результате мы получим строковое представление числа с двойной точностью. Используемый по умолчанию конструктор `QVariant` устанавливает значение «invalid» (недопустимое).

```

const QVariant Invalid;
QVariant Cell::value() const
{
    if (cacheIsDirty) {
        cacheIsDirty = false;
        QString formulaStr = formula();
        if (formulaStr.startsWith('=')) {
            cachedValue = formulaStr.mid(1);
        } else if (formulaStr.startsWith('=')) {
            cachedValue = Invalid;
            QString expr = formulaStr.mid(1);
            expr.replace(" ", "");
            expr.append(QChar::Null);

            int pos = 0;
            cachedValue = evalExpression(expr, pos);
            if (expr[pos] != QChar::Null)
                cachedValue = Invalid;
        } else {
            bool ok;
            double d = formulaStr.toDouble(&ok);
            if (ok) {

```



```
        cachedValue = d;  
    } else {  
        cachedValue = formulaStr;  
    }  
}  
}  
return cachedValue;  
}
```

Закрытая функция `value()` возвращает значение ячейки. Если флажок `cacheIsValid` имеет значение `true`, нам необходимо выполнить перерасчет значения.

Если формула начинается с одиночной кавычки (например, «'12345'»), то одиночная кавычка занимает позицию 0, а значение представляет собой строку в позициях с 1 до последней.

Если формула начинается со знака равенства («=»), мы выделяем строку, начиная с позиции 1, и удаляем из нее любые пробелы. Затем мы вызываем функцию `evalExpression()` для вычисления значения выражения. Аргумент `pos` передается по ссылке; он задает позицию символа, с которого должен начинаться синтаксический анализ выражения. После вызова функции `evalExpression()` в позиции `pos` нами должен быть установлен символ `QChar::Null`, если синтаксический анализ завершился успешно. Если синтаксический анализ не закончился успешно, мы устанавливаем `cachedValue` на значение `Invalid`.

Если формула не начинается с одиночной кавычки или знака равенства, мы пытаемся преобразовать ее в число с плавающей точкой, используя функцию `toDouble()`. Если преобразование удастся выполнить, мы устанавливаем `cachedValue` на полученное значение; в противном случае устанавливаем `cachedValue` на строку формулы. Например, формула «1.50» приводит к тому, что функция `toDouble()` устанавливает переменную `ok` на значение `true` и возвращает 1.5, а формула «World Population» (население Земли) приводит к тому, что функция `toDouble()` устанавливает переменную `ok` на значение `false` и возвращает 0.0.

Благодаря заданному в функции `toDouble()` указателю на булево значение мы можем отличать строку преобразования, представляющую числовое значение 0.0, от ошибки преобразования (в последнем случае также возвращается 0.0, но булева переменная устанавливается в значение `false`). Иногда нулевое значение при неудачном преобразовании оказывается именно тем, что нам нужно; в этом случае нет необходимости передавать указатель на переменную типа `bool`. По причинам, связанным с производительностью и переносимостью, в Qt никогда не используются исключения C++ для вывода сообщений об ошибках. Это не значит, что вы не можете использовать их в своих Qt-программах, если ваш компилятор поддерживает исключения C++.

Функция `value()` объявлена с модификатором `const`. При объявлении переменных `cachedValue` и `cacheIsValid` мы использовали ключевое слово `mutable`, чтобы компилятор позволял нам модифицировать эти переменные в функциях типа `const`. Может показаться заманчивой возможность сделать функцию `value()` не типа `const` и удалить ключевые слова `mutable`, но это не пропустит компилятор, поскольку мы вызываем `value()` из `data()`, функции с модификатором `const`.

Теперь можно считать, что мы завершили приложение Электронная таблица, если не брать в расчет синтаксический анализ формул. В остальной части данного раздела рассматривается функция `evalExpression()` и две вспомогательные функции `evalTerm()` и `evalFactor()`. Их программный код немного сложен, но он включен сюда, чтобы приложение имело законченный вид. Поскольку этот программный код не относится к программированию графического интерфейса, вы можете спокойно его пропустить и продолжить чтение с главы 5.

Функция `evalExpression()` возвращает значение выражения из ячейки электронной таблицы. Выражение состоит из одного или нескольких термов, разделенных знаками операций «+» или «-». Термы состоят из одного или нескольких факторов (*factors*), разделенных знаками операций «*» или «/». Разбивая выражения на термы, а термы на факторы, мы обеспечиваем правильную последовательность выполнения операций.

Например, «2*C5+D6» является выражением, первый терм которого будет «2*C5», а второй терм – «D6». «2*C5» является термом, первый фактор которого будет «2», а второй фактор – «C5»; «D6» состоит из одного фактора – «D6». Фактором может быть число («2»), обозначение ячейки («C5») или выражение в скобках, перед которым может стоять знак минуса.

Блок-схема синтаксического анализа выражений электронной таблицы представлена на рис. 4.10. Для каждого грамматического символа (*Expression*, *Term* и *Factor* – выражение, терм и фактор) имеется соответствующая функция-член, которая выполняет его синтаксический анализ и структура которой очень хорошо отражает его грамматику. Построенные таким образом синтаксические анализаторы называются парсерами с *рекурсивным спуском* (*recursive-descent parsers*).

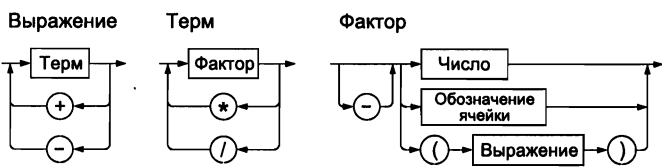


Рис. 4.10. Блок-схема синтаксического анализа выражений электронной таблицы

Давайте начнем с `evalExpression()`, то есть с функции, которая выполняет синтаксический разбор выражения:

```
QVariant Cell::evalExpression(const QString &str, int &pos) const
{
    QVariant result = evalTerm(str, pos);
    while (str[pos] != QChar::Null) {
        QChar op = str[pos];
        if (op != '+' && op != '-')
            return result;
        ++pos;

        QVariant term = evalTerm(str, pos);
        if (result.type() == QVariant::Double
            && term.type() == QVariant::Double) {
```

```

        if (op == '+') {
            result = result.toDouble() + term.toDouble();
        } else {
            result = result.toDouble() - term.toDouble();
        }
    } else {
        result = Invalid;
    }
}
return result;
}

```

Во-первых, мы вызываем функцию `evalTerm()` для получения значения первого термина. Если за ним идет символ «+» или «-», мы вызываем второй раз `evalTerm()`; в противном случае выражение состоит из единственного термина, и мы возвращаем его значение в качестве значения всего выражения. После получения значений первых двух термов мы вычисляем результат операции в зависимости от оператора. Если при оценке обоих термов их значения будут иметь тип `double`, мы рассчитываем результат в виде числа типа `double`; в противном случае мы устанавливаем результат на значение `Invalid`.

Мы продолжаем эту процедуру, пока не закончатся термы. Это даст правильный результат, потому что операции сложения и вычитания обладают свойством «ассоциативности слева» (*left-associative*); то есть «1-2-3» означает «(1-2)-3», а не «1-(2-3)».

```

QVariant Cell::evalTerm(const QString &str, int &pos) const
{
    QVariant result = evalFactor(str, pos);
    while (str[pos] != QChar::Null) {
        QChar op = str[pos];
        if (op != '*' && op != '/')
            return result;
        ++pos;

        QVariant factor = evalFactor(str, pos);
        if (result.type() == QVariant::Double
            && factor.type() == QVariant::Double) {
            if (op == '*') {
                result = result.toDouble() * factor.toDouble();
            } else {
                if (factor.toDouble() == 0.0) {
                    result = Invalid;
                } else {
                    result = result.toDouble() / factor.toDouble();
                }
            }
        }
    }
} else {
    result = Invalid;
}

```

```

    }
}
return result;
}

```

Функция evalTerm() очень напоминает функцию evalExpression(), но в отличие от последней она имеет дело с операциями умножения и деления. В функции evalTerm() необходимо учитывать одну тонкость, а именно, нельзя допускать деление на нуль, так как это приводит к ошибке на некоторых процессорах. Хотя не рекомендуется проверять равенство чисел с плавающей точкой из-за ошибки округления, можно спокойно делать проверку на равенство значению 0.0 для предотвращения деления на нуль.

```

QVariant Cell::evalFactor(const QString &str, int &pos) const
{
    QVariant result;
    bool negative = false;

    if (str[pos] == '-') {
        negative = true;
        ++pos;
    }

    if (str[pos] == '(') {
        ++pos;
        result = evalExpression(str, pos);
        if (str[pos] != ')')
            result = Invalid;
        ++pos;
    } else {
        QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
        QString token;
        while (str[pos].isLetterOrNumber() || str[pos] == '.') {
            token += str[pos];
            ++pos;
        }

        if (regExp.exactMatch(token)) {
            int column = token[0].toUpper().unicode() - 'A';
            int row = token.mid(1).toInt() - 1;
            Cell *c = static_cast<Cell *>(
                tableWidget()->item(row, column));

            if (c) {
                result = c->value();
            } else {
                result = 0.0;
            }
        } else {
            bool ok;

```

```
        result = token.toDouble(&ok);
        if (!ok)
            result = Invalid;
    }
}

if (negative) {
    if (result.type() == QVariant::Double) {
        result = -result.toDouble();
    } else {
        result = Invalid;
    }
}
return result;
}
```

Функция `evalFactor()` немного сложнее, чем `evalExpression()` и `evalTerm()`. Мы начинаем с проверки, не является ли фактор отрицательным. Затем мы проверяем наличие открытой скобки. Если она имеется, мы анализируем значение внутри скобок как выражение, вызывая `evalExpression()`. При анализе выражения в скобках `evalExpression()` вызывает функцию `evalTerm()`, которая вызывает функцию `evalFactor()`, которая вновь вызывает функцию `evalExpression()`. Именно в этом месте осуществляется рекурсия при синтаксическом анализе.

Если фактором не является вложенное выражение, мы выделяем следующую лексему (`token`), и она должна задавать обозначение ячейки или быть числом. Если эта лексема удовлетворяет регулярному выражению в переменной `QRegExp`, мы считаем, что она является ссылкой на ячейку, и вызываем функцию `value()` для этой ячейки. Ячейка может располагаться в любом месте в электронной таблице, и она может ссылаться на другие ячейки. Такая зависимость не вызывает проблемы и просто приводит к дополнительным вызовам функции `value()` и к дополнительному синтаксическому анализу ячеек с признаком «dirty» для перерасчета значений всех зависимых ячеек. Если лексема не является ссылкой на ячейку, мы рассматриваем ее как число.

Что произойдет, если ячейка `A1` содержит формулу «`=A1`»? Или если ячейка `A1` содержит «`=A2`», а ячейка `A2` содержит «`=A1`»? Хотя нами не написан специальный программный код для обнаружения бесконечных циклов в рекурсивных зависимостях, парсер прекрасно справится с этой ситуацией и возвратит недопустимое значение переменной типа `QVariant`. Это даст нужный результат, поскольку мы устанавливаем флажок `cacheIsDirty` на значение `false` и переменную `cachedValue` на значение `Invalid` в функции `value()` перед вызовом `evalExpression()`. Если `evalExpression()` рекурсивно вызывает функцию `value()` для той же ячейки, она немедленно возвращает значение `Invalid`, и тогда все выражение принимает значение `Invalid`.

Теперь мы завершили программу синтаксического анализа формул. Ее можно легко модифицировать для обработки стандартных функций электронной таблицы, например «`sum()`» и «`avg()`», расширяя грамматическое определение фактора. Можно также легко расширить эту реализацию, обеспечив возможность выполнения операции «`+`» над строковыми операндами (для их конкатенации); это не потребует внесения изменений в грамматику.



- *Настройка виджетов Qt*
- *Создание подкласса QWidget*
- *Интеграция пользовательских виджетов в Qt Designer*
- *Двойная буферизация*

Глава 5. Создание пользовательских виджетов

В данной главе объясняются способы создания пользовательских виджетов с помощью средств разработки Qt. Пользовательские виджеты могут создаваться путем определения подкласса существующего виджета Qt или путем определения непосредственно подкласса QWidget. Мы продемонстрируем оба подхода и рассмотрим также способы интеграции пользовательского виджета в *Qt Designer*, чтобы его можно было применять совершенно так же, как встроенный виджет Qt. Мы закончим данную главу примером пользовательского виджета, в котором используется двойная буферизация – эффективный метод быстрого рисования.

Настройка виджетов Qt

В некоторых случаях мы обнаруживаем необходимость в более специализированной настройке виджета Qt по сравнению с той, которую можно обеспечить путем установки его свойств в *Qt Designer* или с помощью вызова его функций. Простое и прямое решение заключается в создании подкласса соответствующего виджетного класса и адаптации его под наши требования.

Чтобы показать, как это делается, в данном разделе мы разработаем шестнадцатеричный наборный счетчик. Наборный счетчик `QSpinBox` поддерживает только десятичные целые числа, но путем создания подкласса достаточно легко можно заставить его принимать и отображать шестнадцатеричные значения.

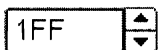


Рис. 5.1. Виджет HexSpinBox

```
#ifndef HEXSPINBOX_H
#define HEXSPINBOX_H

#include <QSpinBox>
```

```

class QRegExpValidator;

class HexSpinBox : public QSpinBox
{
    Q_OBJECT

public:
    HexSpinBox(QWidget *parent = 0);

protected:
    QValidator::State validate(QString &text, int &pos) const;
    int valueFromText(const QString &text) const;
    QString textFromValue(int value) const;

private:
    QRegExpValidator *validator;
};

#endif

```

Шестнадцатеричный наборный счетчик HexSpinBox наследует большую часть функциональности от QSpinBox. Он содержит обычный конструктор и переопределяет три виртуальные функции класса QSpinBox.

```

#include <QtGui>

#include "hexspinbox.h"

HexSpinBox::HexSpinBox(QWidget *parent)
    : QSpinBox(parent)
{
    setRange(0, 255);
    validator = new QRegExpValidator(QRegExp("[0-9A-Fa-f]{1,8}"), this);
}

```

Мы устанавливаем по умолчанию диапазон от 0 до 255 (от 0×00 до $0 \times FF$), который лучше соответствует шестнадцатеричному наборному счетчику, чем диапазон от 0 до 99, принимаемый по умолчанию в QSpinBox.

Пользователь может модифицировать текущее значение наборного счетчика, щелкая по верхней или нижней стрелке, или путем ввода значения в строке редактирования наборного счетчика. В последнем случае мы хотим, чтобы пользователь мог вводить только правильные шестнадцатеричные числа. Для достижения этого мы используем QRegExpValidator, который принимает один или несколько символов, которые все должны входить в диапазоны { «0», ..., «9», «A», ..., «F», «a», ..., «f»}.

```

QValidator::State HexSpinBox::validate(QString &text, int &pos) const
{
    return validator->validate(text, pos);
}

```

Эта функция вызывается в `QSpinBox` для проверки допустимости введенного текста. Результат может иметь одно из трех значений: `Invalid` (текст не соответствует регулярному выражению), `Intermediate` (текст, вероятно, является частью допустимого значения) и `Acceptable` (текст допустим). `QRegExpValidator` имеет подходящую функцию `validate()`, поэтому мы просто возвращаем результат ее вызова. Теоретически следует возвращать `Invalid` или `Intermediate` для значений, лежащих вне диапазона наборного счетчика, но `QSpinBox` достаточно «умен» и может самостоятельно отследить эту ситуацию.

```
QString HexSpinBox::textFromValue(int value) const
{
    return QString::number(value, 16).toUpper();
}
```

Функция `textFromValue()` преобразует целое число в строку. `QSpinBox` вызывает ее для обновления строки редактирования в наборном счетчике, когда пользователь нажимает клавиши верхней или нижней стрелки наборного счетчика. Мы используем статическую функцию `QString::number()`, задавая 16 в качестве второго аргумента для преобразования значения в представленное в нижнем регистре шестнадцатеричное число, и вызываем функцию `QString::toUpper()` для преобразования результата в верхний регистр.

```
int HexSpinBox::valueFromText(const QString &text) const
{
    bool ok;
    return text.toInt(&ok, 16);
}
```

Функция `valueFromText()` выполняет обратное преобразование из строки в целое число. Она вызывается в `QSpinBox`, когда пользователь вводит значение в строку редактирования наборного счетчика и нажимает клавишу `Enter`. Мы используем функцию `QString::toInt()` для попытки преобразования текущего текстового значения (возвращаемого `QSpinBox::text()`) в целое число, вновь используя 16 в качестве базы. Если строка не является правильным шестнадцатеричным числом, `ok` устанавливается на значение `false` и `toInt()` возвращает 0. Здесь нет необходимости рассматривать такую возможность, поскольку контролирующая функция (`validator`) позволяет вводить только правильные шестнадцатеричные значения. Вместо передачи адреса переменной `ok` мы могли бы задать нулевой указатель в первом аргументе функции `toInt()`.

Этим мы завершили создание шестнадцатеричного наборного счетчика. Настройка других виджетов Qt осуществляется по тому же образцу: подобрать подходящий виджет Qt, создать его подкласс и переопределить несколько виртуальных функций для изменения режима его работы. Если все, что нам нужно, — это настроить внешний облик и функции существующего виджета, мы можем применить таблицу стилей или реализовать пользовательский стиль, не создавая подкласса виджета, как описывается в главе 19.

Создание подкласса QWidget

Многие пользовательские виджеты являются простой комбинацией существующих виджетов, либо встроенных в Qt, либо других пользовательских виджетов (таких как `HexSpinBox`). Если пользовательские виджеты строятся на основе существующих виджетов, то они, как правило, могут разрабатываться в *Qt Designer*:

- создайте новую форму, используя шаблон «Widget» (виджет);
- добавьте в эту форму необходимые виджеты и затем расположите их соответствующим образом;
- установите соединения сигналов и слотов;
- если необходима функциональность, которую нельзя обеспечить с помощью механизма сигналов и слотов, необходимый программный код следует писать в рамках класса, который происходит как от класса `QWidget`, так и от класса, сгенерированного компилятором `uic`.

Естественно, комбинация существующих виджетов может быть также полностью запрограммирована вручную. При любом подходе полученный класс является подклассом `QWidget`.

Если виджет не имеет своих собственных сигналов и слотов и не переопределяет никакую виртуальную функцию, можно просто собрать виджет из существующих виджетов, не создавая подкласс. Этим методом мы пользовались в главе 1 для создания приложения *Age* с применением `QWidget`, `QSpinBox` и `QSlider`. Даже в этом случае мы могли бы легко определить подкласс `QWidget` и в его конструкторе создать `QSpinBox` и `QSlider`.

Когда под рукой нет подходящих виджетов Qt и когда нельзя получить желаемый результат, комбинируя и адаптируя существующие виджеты, мы можем все же создать требуемый виджет. Это достигается путем создания подкласса `QWidget` и переопределением обработчиков некоторых событий, связанных с рисованием виджета и реагированием на щелчки мыши. При таком подходе мы свободно можем определять и управлять как внешним видом, так и режимом работы нашего виджета. Такие встроенные в Qt виджеты, как `QLabel`, `QPushButton` и `QTableWidget`, реализованы именно так. Если бы их не было в Qt, все же можно было создать их самостоятельно при помощи предусмотренных в классе `QWidget` открытых функций, обеспечивающих полную независимость от платформы.

Для демонстрации данного подхода при написании пользовательского виджета мы создадим виджет `IconEditor`, показанный на рис. 5.2. Виджет `IconEditor` может использоваться в программе редактирования пиктограмм.

На практике, прежде чем погружаться в создание пользовательского виджета, всегда полезно сначала проверить, не существует ли уже подобного виджета, созданного либо Qt Solution (<http://www.trolltech.com/products/qt/addon/solutions/catalog/4/>), либо коммерческим или некоммерческим сторонним производителем (<http://www.trolltech.com/products/qt/3rdparty/>), поскольку это может сэкономить вам массу времени и сил. В данном случае мы предположим, что подходящего виджета нет, и будем создавать свой собственный.

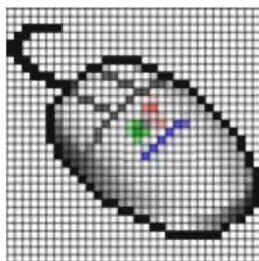


Рис. 5.2. Виджет IconEditor

Сначала рассмотрим заголовочный файл.

```
#ifndef ICONEDITOR_H
#define ICONEDITOR_H

#include <QColor>
#include <QImage>
#include <QWidget>

class IconEditor : public QWidget
{
    Q_OBJECT
    Q_PROPERTY(QColor penColor READ penColor WRITE setPenColor)
    Q_PROPERTY(QImage iconImage READ iconImage WRITE setIconImage)
    Q_PROPERTY(int zoomFactor READ zoomFactor WRITE setZoomFactor)

public:
    IconEditor(QWidget *parent = 0);

    void setPenColor(const QColor &newColor);
    QColor penColor() const { return curColor; }

    void setZoomFactor(int newZoom);
    int zoomFactor() const { return zoom; }
    void setIconImage(const QImage &newImage);
    QImage iconImage() const { return image; }
    QSize sizeHint() const;
```

Класс `IconEditor` использует макрос `Q_PROPERTY()` для объявления трех пользовательских свойств: `penColor`, `iconImage` и `zoomFactor`. Каждое свойство имеет тип данных, функцию «чтения» и необязательную функцию «записи». Например, свойство `penColor` имеет тип `QColor` и может считываться и записываться при помощи функций `penColor()` и `setPenColor()`.

Когда мы используем виджет в *Qt Designer*, пользовательские свойства появляются в редакторе свойств *Qt Designer* ниже свойств, унаследованных от `QWidget`. Свойства могут иметь любой тип, поддерживаемый `QVariant`. Макрос `Q_OBJECT` необходим для классов, в которых определяются свойства.

```
protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void paintEvent(QPaintEvent *event);

private:
    void setImagePixel(const QPoint &pos, bool opaque);
    QRect pixelRect(int i, int j) const;

    QColor curColor;
    QImage image;
    int zoom;
};

#endif
```

IconEditor переопределяет три защищенные функции QWidget и имеет несколько закрытых функций и переменных. В трех закрытых переменных содержатся значения трех свойств.

Файл реализации класса начинается с конструктора IconEditor:

```
#include <QtGui>

#include "iconeditor.h"

IconEditor::IconEditor(QWidget *parent)
    : QWidget(parent)
{
    setAttribute(Qt::WA_StaticContents);
    setSizePolicy(QSizePolicy::Minimum, QSizePolicy::Minimum);

    curColor = Qt::black;
    zoom = 8;

    image = QImage(16, 16, QImage::Format_ARGB32);
    image.fill(qRgb(0, 0, 0));
}
```

В конструкторе имеется несколько тонких моментов, связанных с применением атрибута Qt::WA_StaticContents и вызовом функции setSizePolicy(). Вскоре мы обсудим их.

Устанавливается черный цвет пера. Коэффициент масштабирования изображения (zoom factor) устанавливается на 8, то есть каждый пиксель пиктограммы представляется квадратом 8×8 .

Данные пиктограммы хранятся в переменной-члене image, и доступ к ним может осуществляться при помощи функций setIconImage() и iconImage(). Программа редактирования пиктограмм обычно вызывает функцию setIconImage() при открытии пользователем файла пиктограммы и функцию iconImage() для считывания пиктограммы из памяти, когда пользователь хочет ее сохранить. Пере-

менная `image` имеет тип `QImage`. Мы инициализируем ее областью 16×16 пикселей и на 32-битовый формат `ARGB`, который поддерживает полупрозрачность. Мы очищаем данные изображения, устанавливая признак прозрачности.

Способ хранения изображения в классе `QImage` не зависит от оборудования. При этом его глубина может устанавливаться на 1, 8 или 32 бита. Изображения с 32-битовой глубиной используют по 8 бит на красный, зеленый и синий компоненты пикселя. В остальных 8 битах хранится альфа-компонент пикселя (уровень его прозрачности). Например, компоненты красный, зеленый и синий «числота» красного цвета и альфа-компонент имеют значения 255, 0, 0 и 255. В Qt этот цвет можно задавать так:

```
QRgb red = qRgba(255, 0, 0, 255);
```

или так (поскольку этот цвет непрозрачен):

```
QRgb red = qRgb(255, 0, 0);
```

Тип `QRgb` просто синоним типа `unsigned int`, созданный с помощью директивы `typedef`, а `qRgb()` и `qRgba()` являются встроенными функциями (то есть со спецификатором `inline`), которые преобразуют свои аргументы в 32-битовое целое число `ARGB`. Допускается также запись

```
QRgb red = 0xFFFF0000;
```

где первые FF соответствуют альфа-компоненту, а вторые FF – красному компоненту. В конструкторе класса `IconEditor` мы делаем `QImage` прозрачным, используя 0 в качестве значения альфа-компонента.

В Qt для хранения цветов предусмотрено два типа: `QRgb` и `QColor`. В то время как `QRgb` всего лишь определяется в `QImage` ключевым словом `typedef` для представления пикселей 32-битовым значением, `QColor` является классом, который имеет много полезных функций и широко используется в Qt для хранения цветов. В виджете `IconEditor` мы используем `QRgb` только при работе с `QImage`; мы применяем `QColor` во всех остальных случаях, включая свойство цвет пера `penColor`.

```
QSize IconEditor::sizeHint() const
{
    QSize size = zoom * image.size();
    if (zoom >= 3)
        size += QSize(1, 1);
    return size;
}
```

Функция `sizeHint()` класса `QWidget` переопределяется и возвращает «идеальный» размер виджета. Здесь мы размер изображения умножаем на масштабный коэффициент и в случае, когда масштабный коэффициент равен или больше 3, добавляем еще один пиксель по каждому направлению для размещения сетки. (Мы не показываем сетку при масштабном коэффициенте 1 или 2, поскольку в этом случае едва ли найдется место для пикселей пиктограммы.)

Идеальный размер виджета играет очень заметную роль при размещении виджетов. Менеджеры компоновки Qt стараются максимально учесть идеальный размер виджета при размещении дочерних виджетов. Для того чтобы `IconEditor` был удобен для менеджера компоновки, он должен сообщить свой правдоподобный идеальный размер.

Кроме идеального размера виджет имеет «политику размера», которая говорит системе компоновки о желательности или нежелательности его растяжения или сжатия. Вызывая в конструкторе функцию `setSizePolicy()` со значением `QSizePolicy::Minimum` в качестве горизонтальной и вертикальной политики, мы указываем менеджеру компоновки, который отвечает за размещение этого виджета, на то, что идеальный размер является фактически его минимальным размером. Другими словами, при необходимости виджет может растягиваться, но он никогда не должен сжиматься до размеров меньших, чем идеальный. Политику размера можно изменять в *Qt Designer* путем установки свойства виджета `sizePolicy`. Мы разъясняем смысл различной политики размеров в главе 6.

```
void IconEditor::setPenColor(const QColor &newColor)
{
    curColor = newColor;
}
```

Функция `setPenColor()` устанавливает текущий цвет пера. Этот цвет будет использоваться при выводе на экран новых пикселей.

```
void IconEditor::setIconImage(const QImage &newImage)
{
    if (newImage != image) {
        image = newImage.convertToFormat(QImage::Format_ARGB32);
        update();
        updateGeometry();
    }
}
```

Функция `setIconImage()` задает изображение для редактирования. Мы вызываем `convertToFormat()` для установки 32-битовой глубины изображения с альфа-буфером, если это еще не сделано. В дальнейшем везде мы будем предполагать, что изображение хранится в 32-битовых элементах типа `ARGB`.

После установки переменной `image` мы вызываем функцию `QWidget::update()` для запланированной перерисовки виджета с новым изображением. Затем мы вызываем `QWidget::updateGeometry()`, чтобы сообщить всем содержащим этот виджет менеджерам компоновки об изменении идеального размера виджета. Размещение виджета затем будет автоматически адаптировано к его новому идеальному размеру.

```
void IconEditor::setZoomFactor(int newZoom)
{
    if (newZoom < 1)
        newZoom = 1;

    if (newZoom != zoom) {
        zoom = newZoom;
        update();
        updateGeometry();
    }
}
```

Функция `setZoomFactor()` устанавливает масштабный коэффициент изображения. Для предотвращения деления на нуль мы корректируем всякое значение

меньшее, чем 1. Мы опять вызываем функции `update()` и `updateGeometry()` для перерисовки виджета и уведомления всех менеджеров компоновки об изменении идеального размера.

Функции `penColor()`, `iconImage()` и `zoomFactor()` реализуются в заголовочном файле как встроенные функции.

Теперь мы рассмотрим программный код функции `paintEvent()`. Эта функция играет очень важную роль в классе `IconEditor`. Она вызывается всякий раз, когда требуется перерисовать виджет. Используемая по умолчанию ее реализация в `QWidget` ничего не делает, оставляя виджет пустым.

Так же как рассмотренная нами в главе 3 функция `closeEvent()`, функция `paintEvent()` является обработчиком события. В Qt предусмотрено много других обработчиков событий, каждый из которых относится к определенному типу события. Обработка событий подробно рассматривается в главе 7.

Существует множество ситуаций, когда генерируется событие рисования (`paint`) и вызывается функция `paintEvent()`. Например:

- при первоначальном выводе на экран виджета система автоматически генерирует событие рисования, чтобы виджет нарисовал сам себя;
- при изменении размеров виджета система генерирует событие рисования;
- если виджет перекрывается другим окном и затем вновь оказывается видимым, генерируется событие рисования для областей, которые закрывались (если только система управления окнами не сохранит закрытую область).

Мы можем также принудительно сгенерировать событие рисования путем вызова функции `QWidget::update()` или `QWidget::repaint()`. Различие между этими функциями следующее: `repaint()` приводит к немедленной перерисовке, а функция `update()` просто передает событие рисования в очередь событий, обрабатываемых Qt. (Обе функции ничего не будут делать, если виджет не видим на экране.) Если `update()` вызывается несколько раз, Qt из нескольких следующих друг за другом событий рисования делает одно событие для предотвращения мерцания. В классе `IconEditor` мы всегда используем функцию `update()`.

Ниже приводится программный код.

```
void IconEditor::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    if (zoom >= 3) {
        painter.setPen(palette().foreground().color());
        for (int i = 0; i <= image.width(); ++i)
            painter.drawLine(zoom * i, 0, zoom * i, zoom * image.height());
        for (int j = 0; j <= image.height(); ++j)
            painter.drawLine(0, zoom * j, zoom * image.width(), zoom * j);
    }

    for (int i = 0; i < image.width(); ++i) {
        for (int j = 0; j < image.height(); ++j) {
            QRect rect = pixelRect(i, j);
            if (!event->region().intersect(rect).isEmpty()) {
                QColor color = QColor::fromRgba(image.pixel(i, j));
```

```

        if (color.alpha() < 255)
            painter.fillRect(rect, Qt::white);
        painter.fillRect(rect, color);
    }
}
}

```

Мы начинаем с построения объекта `QPainter` нашего виджета. Если масштабный коэффициент равен или больше 3, мы вычерчиваем с помощью функции `QPainter::drawLine()` горизонтальные и вертикальные линии сетки.

Вызов функции `QPainter::drawLine()` имеет следующий формат:

```
painter.drawLine(x1, y1, x2, y2);
```

где $(x1, y1)$ задает положение одного конца линии и $(x2, y2)$ задает положение другого конца линии. Существует перегруженный вариант функции, которая принимает два объекта типа `QPoints` вместо четырех целых чисел.

Пиксель в верхнем левом углу виджета `Qt` имеет координаты $(0, 0)$, а пиксель в нижнем правом углу имеет координаты $(width() - 1, height() - 1)$. Это напоминает обычную декартовскую систему координат, но только перевернутую сверху вниз, как показано на рис. 5.3. Мы можем изменить систему координат в `QPainter`, трансформируя ее такими способами, как смещение, масштабирование, вращение и отсечение. Эти вопросы мы рассматриваем в главе 8.

$(0, 0)$

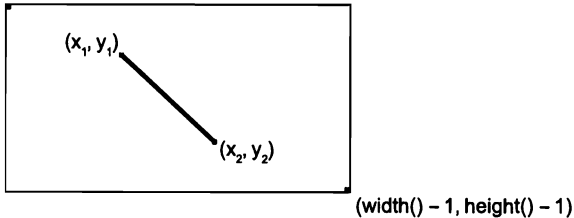


Рис. 5.3. Вычерчивание линии при помощи `QPainter`

Перед вызовом в `QPainter` функции `drawLine()` мы устанавливаем цвет линии, используя функцию `setPen()`. Мы могли бы жестко запрограммировать цвет (например, черный или серый), но лучше использовать палитру виджета.

Каждый виджет имеет палитру, которая определяет назначение цветов. Например, предусмотрен цвет фона виджетов (обычно светло-серый) и цвет текста на этом фоне (обычно черный). По умолчанию палитра виджета адаптирована под схему цветов оконной системы. Используя цвета из палитры, мы обеспечим в `IconEditor` учет пользовательских настроек.

Палитра виджета состоит из трех цветовых групп: активная, неактивная и нерабочая. Цветовая группа выбирается в зависимости от текущего состояния виджета:

- группа `Active` используется для виджетов текущего активного окна;
- группа `Inactive` используется виджетами других окон;
- группа `Disabled` используется отключенными виджетами любого окна.

Функция `QWidget::palette()` возвращает палитру виджета в виде объекта `QPalette`. Цветовые группы определяются как элементы перечисления типа `QPalette::ColorGroup`. Удобная функция `QWidget::colorGroup()` возвращает правильную цветовую группу текущего состояния виджета, и поэтому нам редко придется выбирать цвет непосредственно из палитры.

Когда нам нужно получить соответствующую кисть или цвет для рисования, правильный подход связан с применением текущей палитры, полученной функцией `QWidget::palette()`, и соответствующей ролевой функции, например `QPalette::foreground()`. Каждая ролевая функция возвращает кисть, что обычно и требуется, однако если нам нужен только цвет, его можно извлечь из кисти, как мы это делали в `paintEvent()`. По умолчанию возвращаемые кисти соответствуют состоянию виджета, поэтому нам не надо указывать цветовую группу.

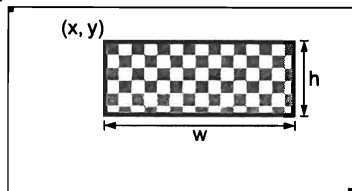
Функция `paintEvent()` завершает рисование изображения. Вызов `IconEditor::pixelRect()` возвращает `QRect`, который определяет область перерисовки. (На рис. 5.4 показано, как рисуется прямоугольник.) Мы не перерисовываем пиксели, которые попадают за пределы данной области, обеспечивая простую оптимизацию.

Мы вызываем `QPainter::fillRect()` для вывода на экран масштабируемого пикселя. `QPainter::fillRect()` принимает `QRect` и `QBrush`. Передавая `QColor` в качестве кисти, мы обеспечиваем равномерное заполнение области. Если цвет не является полностью непрозрачным (значение альфа-канала меньше 255), то сначала мы рисуем белый фон.

```
QRect IconEditor::pixelRect(int i, int j) const
{
    if (zoom >= 3) {
        return QRect(zoom * i + 1, zoom * j + 1, zoom - 1, zoom - 1);
    } else {
        return QRect(zoom * i, zoom * j, zoom, zoom);
    }
}
```

Функция `pixelRect()` возвращает объект `QRect`, который может использоваться функцией `QPainter::fillRect()`. Параметры `i` и `j` являются координатами пикселя в `QImage`, а не в виджете. Если коэффициент масштабирования равен 1, обе системы координат будут полностью совпадать.

(0, 0)



(width() - 1, height() - 1)

Рис. 5.4. Вычерчивание прямоугольника при помощи `QPainter`

Конструктор `QRect` имеет синтаксис `QRect(x, y, width, height)`, где `(x, y)` являются координатами верхнего левого угла прямоугольника, а `width` и `height` являются

размерами прямоугольника (шириной и высотой). Если коэффициент масштабирования равен не менее 3, мы уменьшаем размеры прямоугольника на один пиксель по горизонтали и по вертикали, чтобы не загромождать линии сетки.

```
void IconEditor::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton) {
        setImagePixel(event->pos(), true);
    } else if (event->button() == Qt::RightButton) {
        setImagePixel(event->pos(), false);
    }
}
```

Когда пользователь нажимает кнопку мыши, система генерирует событие «клавиша мыши нажата» (mouse press). Путем переопределения функции `QWidget::mousePressEvent()` мы можем обработать это событие и установить или стереть пиксель изображения, находящийся под курсором мыши.

Если пользователь нажал левую кнопку мыши, мы вызываем закрытую функцию `setImagePixel()` с `true` в качестве второго аргумента, указывая на необходимость установки цвета пикселя на текущий цвет пера. Если пользователь нажал правую кнопку мыши, мы также вызываем функцию `setImagePixel()`, но передаем `false` для стирания пикселя.

```
void IconEditor::mouseMoveEvent(QMouseEvent *event)
{
    if (event->buttons() & Qt::LeftButton) {
        setImagePixel(event->pos(), true);
    } else if (event->buttons() & Qt::RightButton) {
        setImagePixel(event->pos(), false);
    }
}
```

Функция `mouseMoveEvent()` обрабатывает события «перемещение мыши». По умолчанию эти события генерируются только при нажатой пользователем кнопки мыши. Можно изменить этот режим работы с помощью вызова функции `QWidget::setMouseTracking()`, но нам не нужно это делать в нашем примере.

Как при нажатии левой или правой кнопки мыши устанавливается или стирается пиксель, так и при удерживании нажатой кнопки над пикселем тоже будет устанавливаться или стираться пиксель. Поскольку допускается удерживать нажатыми одновременно несколько кнопок, возвращаемое функцией `QMouseEvent::buttons()` значение представляет собой результат логической операции поразрядного ИЛИ для кнопок. Мы проверяем нажатие определенной кнопки при помощи оператора `&` и при наличии соответствующего состояния вызываем функцию `setImagePixel()`.

```
void IconEditor::setImagePixel(const QPoint &pos, bool opaque)
{
    int i = pos.x() / zoom;
    int j = pos.y() / zoom;

    if (image.rect().contains(i, j)) {
        if (opaque) {
            image.setPixel(i, j, penColor().rgba());
        }
    }
}
```

```

    } else {
        image.setPixel(i, j, qRgba(0, 0, 0, 0));
    }
    update(pixelRect(i, j));
}
}

```

Функция `setImagePixel()` вызывается из `mousePressEvent()` и `mouseMoveEvent()` для установки или стирания пикселя. Параметр `pos` определяет положение мыши на виджете.

На первом этапе надо преобразовать положение мыши из системы координат виджета в систему координат изображения. Это достигается путем деления координат положения мыши `x()` и `y()` на коэффициент масштабирования. Затем мы проверяем попадание точки в нужную область. Это легко сделать при помощи функций `QImage::rect()` и `QRect::contains()`. Фактически здесь проверяется попадание значения переменной `i` в промежуток между 0 и значением `image.width() - 1`, а переменной `j` в промежуток между 0 и значением `image.height() - 1`.

В зависимости от значения параметра `opaque` мы устанавливаем или стираем пиксель в изображении. При стирании пиксель фактически становится прозрачным. Для вызова `QImage::setPixel()` мы должны преобразовать перо `QColor` в 32-битовое значение `ARGB`. В конце мы вызываем функцию `update()` с передачей объекта `QRect`, задающего область перерисовки.

Теперь, когда уже рассмотрены функции-члены, мы вернемся к используемому в конструкторе атрибуту `Qt::WA_StaticContents`. Этот атрибут указывает Qt на то, что содержимое виджета не изменяется при изменении его размеров и что его верхний левый угол остается на прежнем месте. Qt использует эту информацию, чтобы лишний раз не перерисовывать при изменении размеров виджета уже видимые его области. Иллюстрация приводится на рис. 5.5.

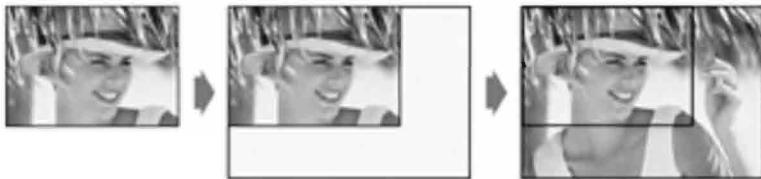


Рис. 5.5. Изменение размеров виджета `Qt::WA_StaticContents`

Обычно при изменении размеров виджета Qt генерирует событие рисования для всей видимой области виджета. Но если виджет создается с установленным флажком `Qt::WA_StaticContents`, область рисования ограничивается не показанными ранее пикселями. Это подразумевает, что если размеры виджета уменьшаются, событие рисования вообще не будет сгенерировано.

Теперь виджет `IconEditor` полностью построен. На основе применения приводимых в предыдущих главах сведений и примеров мы можем написать программу, в которой виджет `IconEditor` будет сам являться окном, использоваться в качестве центрального виджета в главном окне `QMainWindow`, в качестве дочернего виджета менеджера компоновки или в качестве дочернего виджета объекта `QScroll Area`. В следующем разделе мы рассмотрим способы его интеграции в *Qt Designer*.

Интеграция пользовательских виджетов в Qt Designer

Прежде чем мы сможем использовать пользовательские виджеты в *Qt Designer*, мы должны сделать так, что *Qt Designer* будет знать о них. Для этого существует два способа: метод «продвижения» («promotion») и метод подключения (plugin).

Метод продвижения является самым быстрым и самым простым. Он заключается в выборе некоторого встроенного виджета Qt, программный интерфейс которого похож на программный интерфейс пользовательского виджета, и заполнении полей диалогового окна пользовательского виджета (оно показано на рис. 5.6) *Qt Designer* некоторыми данными об этом виджете. Затем этот пользовательский виджет может использоваться в формах, разработанных с помощью *Qt Designer*, но при редактировании или просмотре он отображается просто в виде выбранного встроенного виджета Qt.

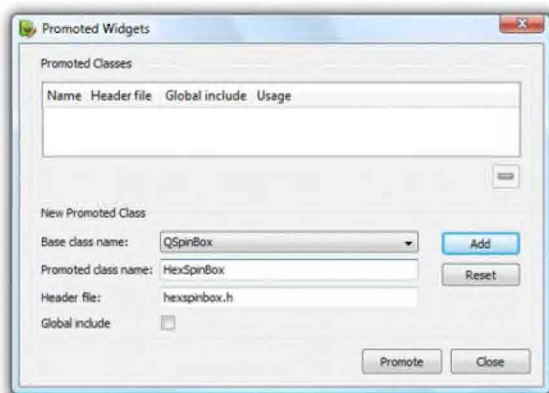


Рис. 5.6. Диалоговое окно для создания пользовательских виджетов *Qt Designer*

Ниже приводится порядок действий при интеграции данным методом виджета *HexSpinBox*.

1. Создайте наборный счетчик *QSpinBox*, перетаскивая его с панели виджетов *Qt Designer* на форму.
2. Щелкните правой клавишей мыши по наборному счетчику и выберите пункт контекстного меню *Promote to Custom Widget* (Преобразовать в пользовательский виджет).
3. Заполните в появившемся диалоговом окне поле названия класса значением «*HexSpinBox*» и поле заголовочного файла значением «*hexspinbox.h*».

Вот и все! Сгенерированный компилятором *uic* программный код будет содержать оператор `#include hexspinbox.h` вместо `<QSpinBox>` и будет инстанцировать *HexSpinBox*. В *Qt Designer* виджет *HexSpinBox* будет представлен виджетом *QSpinBox*, позволяя нам устанавливать любые свойства *QSpinBox* (например, допустимый диапазон значений и текущее значение).

Недостатками метода продвижения являются недоступность в *Qt Designer* свойств, характерных для пользовательского виджета, и то, что пользовательский виджет представляется в *Qt Designer* не своим изображением. Обе эти проблемы могут быть решены при применении метода подключения.

Метод подключения требует создания библиотеки подключаемых модулей, которую *Qt Designer* может загружать во время выполнения и использовать для создания экземпляров виджетов. В этом случае при редактировании формы и ее просмотре в *Qt Designer* будет использован реальный виджет, и благодаря метаобъектной системе Qt можно динамически получать список его свойств в *Qt Designer*. Для демонстрации этого метода мы с его помощью выполним интеграцию редактора пиктограмм *IconEditor*, описанного в предыдущем разделе.

Во-первых, мы должны создать подкласс *QDesignerCustomWidgetInterface* и переопределить несколько виртуальных функций. Предположим, что исходный файл подключаемого модуля расположен в каталоге с именем *iconeditorplugin*, а исходный текст программы *IconEditor* расположен в параллельном каталоге с именем *iconeditor*.

Ниже приводится определение класса.

```
#include <QDesignerCustomWidgetInterface>

class IconEditorPlugin : public QObject,
                        public QDesignerCustomWidgetInterface
{
    Q_OBJECT
    Q_INTERFACES(QDesignerCustomWidgetInterface)

public:
    IconEditorPlugin(QObject *parent = 0);

    QString name() const;
    QString includeFile() const;
    QString group() const;
    QIcon icon() const;
    QString toolTip() const;
    QString whatsThis() const;
    bool isContainer() const;
    QWidget *createWidget(QWidget *parent);
};
```

Подкласс *IconEditorPlugin* является фабрикой класса (factory class), который инкапсулирует виджет *IconEditor*. Он является потомком классов *QObject* и *QDesignerCustomWidgetInterface* и использует макрос *Q_INTERFACES()*, указывая компилятору то, что второй базовый класс представляет собой подключаемый интерфейс. Его функции применяются *Qt Designer* для создания экземпляров класса и получения информации о нем.

```
IconEditorPlugin::IconEditorPlugin(QObject *parent)
    : QObject(parent)
{
}
```

IconEditorPlugin имеет тривиальный конструктор.

```
QString IconEditorPlugin::name() const
{
    return "IconEditor";
}
```

Функция name() возвращает имя подключаемого виджета.

```
QString IconEditorPlugin::includeFile() const
{
    return "iconeditor.h";
}
```

Функция includeFile() возвращает имя заголовочного файла для заданного виджета, который инкапсулирован в подключаемом модуле. Заголовочный файл включается в программный код, сгенерированный компилятором uic.

```
QString IconEditorPlugin::group() const
{
    return tr("Image Manipulation Widgets");
}
```

Функция group() возвращает имя группы на панели виджетов, к которой принадлежит пользовательский виджет. Если это имя еще не используется, Qt Designer создаст новую группу для виджета.

```
QIcon IconEditorPlugin::icon() const
{
    return QIcon(":/images/iconeditor.png");
}
```

Функция icon() возвращает пиктограмму, которая будет использоваться для представления пользовательского виджета на панели виджетов Qt Designer. В нашем случае мы предполагаем, что IconEditorPlugin имеет ресурсный файл Qt, содержащий соответствующий элемент для изображения редактора пиктограмм.

```
QString IconEditorPlugin::toolTip() const
{
    return tr("An icon editor widget");
}
```

Функция toolTip() возвращает всплывающую подсказку, которая появляется, когда мышь находится на пользовательском виджете в панели виджетов Qt Designer.

```
QString IconEditorPlugin::whatsThis() const
{
    return tr("This widget is presented in Chapter 5 of <i>C++ GUI "
              "Programming with Qt 4</i> as an example of a custom Qt "
              "widget.");
}
```

Функция whatsThis() возвращает текст «What's This?» для отображения в Qt Designer.

```
bool IconEditorPlugin::isContainer() const
{
    return false;
}
```

Функция `isContainer()` возвращает `true`, если данный виджет может содержать другие виджеты; в противном случае возвращает `false`. Например, `QFrame` представляет собой виджет, который может содержать другие виджеты. В целом любой виджет может содержать другие виджеты, но *Qt Designer* не позволяет это делать, если `isContainer()` возвращает `false`.

```
QWidget *IconEditorPlugin::createWidget(QWidget *parent)
{
    return new IconEditor(parent);
}
```

Qt Designer вызывает функцию `createWidget()` для создания экземпляра класса указанного родительского виджета.

```
Q_EXPORT_PLUGIN2(iconeditorplugin, IconEditorPlugin)
```

В конце исходного файла реализации класса подключаемого модуля мы должны использовать макрос `Q_EXPORT_PLUGIN2()`, чтобы сделать его доступным для *Qt Designer*. Первый аргумент – назначаемое нами имя подключаемого модуля, второй аргумент – имя класса, который его реализует.

Используемый для построения подключаемого модуля файл `.pro` выглядит следующим образом:

```
TEMPLATE       = lib
CONFIG         += designer plugin release
HEADERS        = ../iconeditor/iconeditor.h \
                iconeditorplugin.h
SOURCES        = ../iconeditor/iconeditor.cpp \
                iconeditorplugin.cpp
RESOURCES      = iconeditorplugin.qrc
DESTDIR        = $$[QT_INSTALL_PLUGINS]/designer
```

У инструмента построения `qmake` имеется несколько встроенных заранее заданных переменных. Одна из них – это `$$[QT_INSTALL_PLUGINS]`, содержащая путь к директории плагинов, находящейся в каталоге, где располагается *Qt*. Когда вы вводите команду `make` или `nmake` для построения подключаемого модуля, он автоматически устанавливается в каталог `plugins/designer Qt`. После построения подключаемого модуля виджет `IconEditor` может использоваться в *Qt Designer* таким же образом, как любые встроенные виджеты *Qt*.

Если требуется интегрировать в *Qt Designer* несколько пользовательских виджетов, вы можете либо создать отдельный подключаемый модуль для каждого из них, либо объединить всех в один подключаемый модуль, реализова интерфейс `QDesignerCustomWidgetCollectionInterface`.

Двойная буферизация

Двойная буферизация является методом программирования графического пользовательского интерфейса, при котором изображение виджета формируется вне экрана в виде пиксельной карты, и затем эта пиксельная карта выводится на экран. В ранних версиях *Qt* этот метод часто использовался для предотвращения мерцания изображения и для построения более быстрого пользовательского интерфейса.

В Qt 4 класс `QWidget` это делает автоматически, поэтому нам редко приходится беспокоиться о мерцании виджетов. Все же явная двойная буферизация оказывается полезной, если виджет воспроизводится сложным образом и это приходится делать постоянно. Мы можем постоянно хранить с виджетом пиксельную карту, которая всегда будет готова отреагировать на следующее событие рисования, и копировать пиксельную карту в виджет при получении нами любого события рисования. Она особенно полезна в тех случаях, когда мы хотим выполнить небольшие модификации, например начертить резиновую ленту без необходимости постоянной перерисовки виджета.

Мы закончим данную главу рассмотрением пользовательского виджета `Plotter` (построитель графиков), который показан на рис. 5.7 и 5.9. Этот виджет использует двойную буферизацию и демонстрирует некоторые другие аспекты Qt-программирования, в том числе обработку событий клавиатуры, ручную компоновку виджетов и координатные системы.

В реальном приложении, где необходим виджет для построения кривых и диаграмм, вместо создания пользовательского виджета, как мы это делаем здесь, мы скорее всего использовали бы один из доступных виджетов от сторонних производителей. Например, можно использовать `GraphPak` с сайта <http://www.ics.com/>, `KD Chart` с сайта <http://www.kdab.net/> или `Qwt` с сайта <http://qwt.sourceforge.net/>.

Виджет `Plotter` выводит на экран одну или несколько кривых, задаваемых вектором ее координат. Пользователь может начертить на изображении резиновую ленту, и `Plotter` отобразит крупным планом заключенную в ней область. Пользователь вычерчивает резиновую ленту, делая сначала щелчок в некоторой точке изображения, перетаскивая затем мышью с нажатой левой кнопкой в другую позицию и освобождая клавишу мыши. В Qt есть класс `QRubberBand`, предназначенный для рисования резиновых лент, но здесь мы нарисуем ее самостоятельно, чтобы лучше контролировать ее внешний вид, а также чтобы продемонстрировать двойную буферизацию.

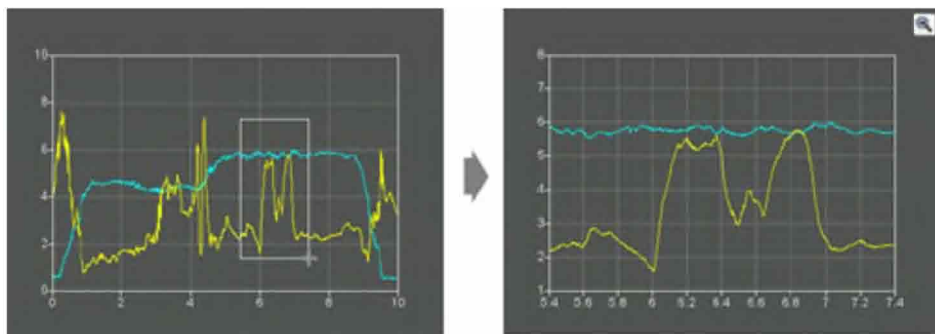


Рис. 5.7. Увеличение изображения виджета `Plotter`

Пользователь может увеличивать изображение, несколько раз используя резиновую ленту, уменьшить изображение при помощи кнопки `Zoom Out` (уменьшить изображение) и затем вновь его увеличить с помощью кнопки `Zoom In` (увеличить изображение). Кнопки `Zoom In` и `Zoom Out` появляются при первом изменении масштаба изображения, и поэтому они не будут заслонять экран, если пользователь не изменяет масштаб представления диаграммы.

Виджет `Plotter` может содержать данные любого количества кривых. Он также содержит стек параметров графика `PlotSettings`, каждое значение которого соответствует конкретному масштабу изображения.

Давайте рассмотрим этот класс, начиная с заголовочного файла `plotter.h`.

```
#ifndef PLOTTER_H
#define PLOTTER_H

#include <QMap>
#include <QPixmap>
#include <QVector>
#include <QWidget>

class QToolButton;
class PlotSettings;
class Plotter : public QWidget
{
    Q_OBJECT

public:
    Plotter(QWidget *parent = 0);

    void setPlotSettings(const PlotSettings &settings);
    void setCurveData(int id, const QVector<QPointF> &data);
    void clearCurve(int id);
    QSize minimumSizeHint() const;
    QSize sizeHint() const;

public slots:
    void zoomIn();
    void zoomOut()
```

Сначала мы включаем заголовочные файлы для Qt-классов, используемых в заголовочном файле построителя графиков, и предварительно объявляем классы, на которые имеются указатели или ссылки в заголовочном файле.

В классе `Plotter` мы предоставляем три открытые функции для настройки графика и два открытых слота для увеличения и уменьшения масштаба изображения. Мы также переопределяем функции `minimumSizeHint()` и `sizeHint()` класса `QWidget`. Мы храним точки кривой в векторе `QVector<QPointF>`, где `QPointF` — версия `QPoint` для значений с плавающей точкой.

```
protected:
    void paintEvent(QPaintEvent *event);
    void resizeEvent(QResizeEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mouseReleaseEvent(QMouseEvent *event);
    void keyPressEvent(QKeyEvent *event);
    void wheelEvent(QWheelEvent *event);
```


В защищенной секции класса мы объявляем все обработчики событий `QWidget`, которые хотим переопределить.

```
private:
    void updateRubberBandRegion();
    void refreshPixmap();
    void drawGrid(QPainter *painter);
    void drawCurves(QPainter *painter);

    enum { Margin = 50 };

    QToolButton *zoomInButton;
    QToolButton *zoomOutButton;
    QMap<int, QVector<QPointF> > curveMap;
    QVector<PlotSettings> zoomStack;
    int curZoom;
    bool rubberBandIsShown;
    QRect rubberBandRect;
    QPixmap pixmap;
};
```

В закрытой секции класса мы объявляем несколько функций для рисования виджета, константу и несколько переменных-членов. Константа `Margin` применяется для обеспечения некоторого свободного пространства вокруг диаграммы.

Среди переменных-членов находится `pixmap`, которая имеет тип `QPixmap`. Эта переменная содержит копию всего виджета, идентичную его изображению на экране. График всегда сначала строится вне экрана на пиксельной карте, и затем пиксельная карта помещается на виджет.

```
class PlotSettings
{
public:
    PlotSettings();

    void scroll(int dx, int dy);
    void adjust();
    double spanX() const { return maxX - minX; }
    double spanY() const { return maxY - minY; }
    double minX;
    double maxX;
    int numXTicks;
    double minY;
    double maxY;
    int numYTicks;

private:
    static void adjustAxis(double &min, double &max, int &numTicks);
};
#endif
```

Класс `PlotSettings` задает диапазон значений по осям x и y и количество отметок на этих осях. На рис. 5.8 показано соответствие между объектом `PlotSettings` и виджетом `Plotter`.

По условному соглашению значение в `numXTicks` и `numYTicks` задается на единицу меньше; если `numXTicks` равно 5, `Plotter` будет на самом деле выводить шесть отметок по оси x . Это упростит расчеты в будущем.

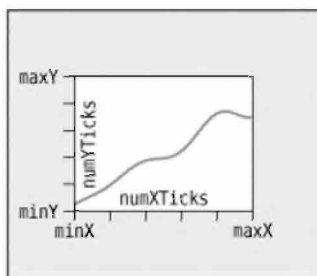


Рис. 5.8. Переменные-члены настроек графика `PlotSettings`

Теперь давайте рассмотрим файл реализации.

```
Plotter::Plotter(QWidget *parent)
    : QWidget(parent)
{
    setBackgroundRole(QPalette::Dark);
    setAutoFillBackground(true);
    setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
    setFocusPolicy(Qt::StrongFocus);
    rubberBandIsShown = false;

    zoomInButton = new QToolButton(this);
    zoomInButton->setIcon(QIcon(":/images/zoomin.png"));
    zoomInButton->adjustSize();
    connect(zoomInButton, SIGNAL(clicked()), this, SLOT(zoomIn()));

    zoomOutButton = new QToolButton(this);
    zoomOutButton->setIcon(QIcon(":/images/zoomout.png"));
    zoomOutButton->adjustSize();
    connect(zoomOutButton, SIGNAL(clicked()), this, SLOT(zoomOut()));

    setPlotSettings(PlotSettings());
}
```

Вызов `setBackgroundRole()` указывает `QWidget` на необходимость использования для цвета стирания виджета «темного» компонента палитры вместо компонента «window». Этим мы определяем цвет, который будет использоваться в Qt по умолчанию для заполнения любых вновь появившихся пикселей при увеличении размеров виджета прежде, чем `paintEvent()` получит возможность рисования нового пикселя. Для включения этого механизма необходимо также вызвать `setAutoFillBackground(true)`. (По умолчанию дочерние виджеты наследуют фон своего родительского виджета.)

Вызов `setSizePolicy()` устанавливает политику размера виджета по обоим направлениям на значение `QSizePolicy::Expanding`. Это подсказывает любому менеджеру компоновки, который ответствен за виджет, что он, прежде всего, склонен к росту, но может также сжиматься. Такая настройка параметров типична для виджетов, которые занимают много места на экране. По умолчанию в обоих направлениях устанавливается политика `QSizePolicy::Preferred`, означающая, что для виджета предпочтительно устанавливать размер на основе его идеального размера, но он может сжиматься до своего минимального идеального размера или расширяться в любых пределах при необходимости.

Вызов `setFocusPolicy(Qt::StrongFocus)` заставляет виджет получать фокус при нажатии клавиши табуляции `Tab`. Когда `Plotter` получает фокус, он будет реагировать на события нажатия клавиш. Виджет `Plotter` понимает несколько клавиш: «+» для увеличения изображения, «-» для уменьшения изображения и клавиш-стрелок для прокрутки вверх, вниз, влево и вправо.

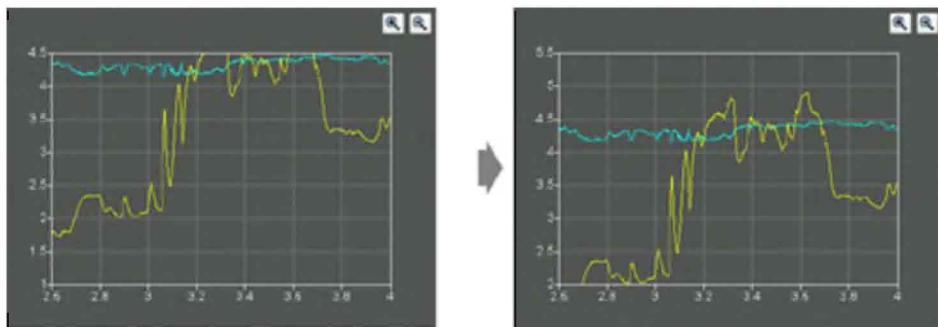


Рис. 5.9. Скроллинг виджета `Plotter`

Также в конструкторе мы создаем две кнопки `QToolButtons`, каждая из которых имеет пиктограмму. Эти кнопки дают возможность пользователю увеличивать и уменьшать масштаб изображения. Пиктограммы кнопок хранятся в файле ресурсов, поэтому любое приложение, использующее виджет `Plotter`, должно иметь следующую строку в файле `.pro`:

```
RESOURCES = plotter.qrc
```

Этот файл ресурсов похож на файл, который мы использовали для приложения *Электронная таблица*:

```
<RCC>
<qresource>
    <file>images/zoomin.png</file>
    <file>images/zoomout.png</file>
</qresource>
</RCC>
```

Вызовы функции `adjustSize()` устанавливают для кнопок их идеальные размеры. Кнопки не размещаются в менеджере компоновки; вместо этого мы задаем их положение вручную при обработке события изменения размеров виджета `Plotter`. Поскольку мы не пользуемся никакими менеджерами компоновки, не-

обходимо явно задавать родительский виджет кнопок, передавая `this` конструктору `QToolButton`.

Вызов в конце функции `setPlotSettings()` завершает инициализацию.

```
void Plotter::setPlotSettings(const PlotSettings &settings)
{
    zoomStack.clear();
    zoomStack.append(settings);
    curZoom = 0;
    zoomInButton->hide();
    zoomOutButton->hide();
    refreshPixmap();
}
```

Функция `setPlotSettings()` устанавливает настройки `PlotSettings` для отображения графика. Ее вызывает конструктор `Plotter`, и она может также вызываться пользователями класса. Построитель кривых начинает работу с принятого по умолчанию масштаба изображения. Каждый раз, когда пользователь увеличивает изображение, создается новый экземпляр `PlotSettings`, который затем помещается в стек масштабов изображения. Этот стек масштабов изображений представлен двумя переменными-членами:

- `zoomStack` содержит настройки для различных масштабов изображения в объекте `QVector<PlotSettings>`;
- `curZoom` содержит индекс текущего элемента `PlotSettings` стека `zoomStack`.

После вызова функции `setPlotSettings()` в стеке масштабов изображений будет находиться только один элемент, а кнопки `Zoom In` и `Zoom Out` будут скрыты. Эти кнопки не будут видны на экране до тех пор, пока мы не вызовем для них функцию `show()` в слотах `zoomIn()` и `zoomOut()`. (Обычно для показа всех дочерних виджетов достаточно вызвать функцию `show()` для виджета верхнего уровня. Но когда мы явным образом вызываем для дочернего виджета функцию `hide()`, этот виджет будет скрыт до вызова для него функции `show()`.)

Вызов функции `refreshPixmap()` необходим для обновления изображения на экране. Обычно мы вызываем функцию `update()`, но здесь поступаем немного по-другому, потому что хотим иметь пиксельную карту `QPixmap` постоянно в обновленном состоянии. После регенерации пиксельной карты функция `refreshPixmap()` вызывает `update()` для помещения пиксельной карты на виджет.

```
void Plotter::zoomOut()
{
    if (curZoom > 0) {
        --curZoom;
        zoomOutButton->setEnabled(curZoom > 0);
        zoomInButton->setEnabled(true);
        zoomInButton->show();
        refreshPixmap();
    }
}
```

Слот `zoomOut()` уменьшает масштаб диаграммы, если она отображена крупным планом. Он уменьшает на единицу текущий масштаб изображения и вклю-

чает или выключает кнопку `ZoomOut` в зависимости от возможности дальнейшего уменьшения диаграммы. Кнопка `Zoom In` включается и отображается на экране, а изображение диаграммы обновляется посредством вызова функции `refreshPixmap()`.

```
void Plotter::zoomIn()
{
    if (curZoom < zoomStack.count() - 1) {
        ++curZoom;
        zoomInButton->setEnabled(curZoom < zoomStack.count() - 1);
        zoomOutButton->setEnabled(true);
        zoomOutButton->show();
        refreshPixmap();
    }
}
```

Если пользователь сначала увеличил изображение, а затем вновь его уменьшил, настройки `PlotSettings` для следующего масштаба изображения уже будут в стеке масштабов изображения, и мы можем увеличить его. (В противном случае можно все же увеличить изображение при помощи резиновой ленты.)

Слот увеличивает на единицу значение `curZoom` для перехода на один уровень в глубь стека масштабов изображения, включает или выключает кнопку `Zoom In` в зависимости от возможности дальнейшего увеличения изображения и включает и показывает кнопку `Zoom Out`. И вновь мы вызываем `refreshPixmap()` для использования построителем графиков настроек самого последнего масштаба изображения.

```
void Plotter::setCurveData(int id, const QVector<QPointF> &data)
{
    curveMap[id] = data;
    refreshPixmap();
}
```

Функция `setCurveData()` устанавливает данные для кривой с заданным идентификатором. Если в `curveMap` уже имеется кривая с таким идентификатором, ее данные заменяются новыми значениями; в противном случае просто добавляется новая кривая. Переменная-член `curveMap` имеет тип `QMap<int, QVector<QPointF>>`.

```
void Plotter::clearCurve(int id)
{
    curveMap.remove(id);
    refreshPixmap();
}
```

Функция `clearCurve()` удаляет заданную кривую из `curveMap`.

```
QSize Plotter::minimumSizeHint() const
{
    return QSize(6 * Margin, 4 * Margin);
}
```

Функция `minimumSizeHint()` напоминает `sizeHint()`; в то время как функция `sizeHint()` устанавливает идеальный размер виджета, `minimumSizeHint()` задает идеальный минимальный размер виджета. Менеджер компоновки никогда не станет задавать виджету размеры ниже идеального минимального размера.

Мы возвращаем значение 300×200 (поскольку Margin равен 50) для того, чтобы можно было разместить окаймляющую кромку по всем четырем сторонам и обеспечить некоторое пространство для самого графика. При меньших размерах считается, что график будет слишком мал и бесполезен.

```
QSize Plotter::sizeHint() const
{
    return QSize(12 * Margin, 8 * Margin);
}
```

В функции `sizeHint()` мы возвращаем «идеальный» размер относительно константы `Margin`, причем горизонтальный и вертикальный компонент этого размера составляют ту же самую приятную для глаза пропорцию 3:2, которую мы использовали для `minimumSizeHint()`.

Мы завершаем рассмотрение открытых функций и слотов построителя графиков `Plotter`. Теперь давайте рассмотрим защищенные обработчики событий.

```
void Plotter::paintEvent(QPaintEvent * /* event */)
{
    QStylePainter painter(this);
    painter.drawPixmap(0, 0, pixmap);

    if (rubberBandIsShown) {
        painter.setPen(palette().light().color());
        painter.drawRect(rubberBandRect.normalized()
                        .adjusted(0, 0, -1, -1));
    }

    if (hasFocus()) {
        QStyleOptionFocusRect option;
        option.initFrom(this);
        option.backgroundColor = palette().dark().color();
        painter.drawPrimitive(QStyle::PE_FrameFocusRect, option);
    }
}
```

Обычно все действия по рисованию выполняются функцией `paintEvent()`. Но в данном случае вся диаграмма уже нарисована функцией `refreshPixmap()`, и поэтому мы можем воспроизвести весь график, просто копируя пиксельную карту в виджет в позицию (0, 0).

Если резиновая лента должна быть видимой, мы рисуем ее поверх графика. Мы используем светлый («light») компонент из текущей цветовой группы виджета в качестве цвета пера для обеспечения хорошего контраста с темным («dark») фоном. Следует отметить, что мы рисуем непосредственно на виджете, оставляя нетронутым внеэкранное изображение на пиксельной карте. Вызов `QRect::normalized()` гарантирует наличие положительных значений ширины и высоты прямоугольника резиновой ленты (выполняя обмен значений координат при необходимости), а вызов `adjusted()` уменьшает размер прямоугольника на один пиксель, позволяя вывести на экран его контур шириной в один пиксель.

Если `Plotter` получает фокус, вывод фокусного прямоугольника выполняется с использованием функции `drawPrimitive()`, задающей стиль виджета, с передачей `QStyle::PE_FrameFocusRect` в качестве первого аргумента и объекта `QStyleOptionFocusRect` в качестве второго аргумента. Опции рисования фокусного прямоугольника наследуются от виджета `Plotter` (путем вызова `initFrom()`). Цвет фона должен задаваться явно.

Если при рисовании требуется использовать текущий стиль, мы можем либо непосредственно вызвать функцию `QStyle`, например:

```
style()->drawPrimitive(QStyle::PE_FrameFocusRect, &option, &painter, this);
```

либо использовать `QStylePainter` вместо обычного `QPainter` (как мы это делали в `Plotter`), что делает рисование более удобным.

Функция `QWidget::style()` возвращает стиль, который будет использован для рисования виджета. В `Qt` стиль виджета является подклассом `QStyle`. Встроенными являются стили `QWindowsStyle`, `QWindowsXPStyle`, `QWindowsVistaStyle`, `QMotifStyle`, `QCEStyle`, `QMacStyle`, `QPlastiqueStyle` и `QCleanlooksStyle`. Все стили переопределяют виртуальные функции класса `QStyle`, чтобы обеспечить корректное рисование в стиле имитируемой платформы. Функция `drawPrimitive()` класса `QStylePainter` вызывает функцию класса `QStyle` с тем именем, которое используется для рисования таких «примитивов», как панели, кнопки и фокусные прямоугольники. Обычно все виджеты используют стиль приложения (`QApplication::style()`), но в любом виджете стиль может переопределяться с помощью функции `QWidget::setStyle()`.

Путем создания подкласса `QStyle` можно определить пользовательский стиль. Это можно делать с целью придания отличительных стилевых особенностей одному какому-то приложению или группе из нескольких приложений, как мы увидим в главе 19. Хотя рекомендуется в целом придерживаться «родного» стиля выбранной платформы, `Qt` предлагает достаточно гибкие средства по управлению стилем тем, у кого большая фантазия.

Встроенные в `Qt` виджеты при рисовании самих себя почти полностью зависят от `QStyle`. Именно поэтому они выглядят естественно на всех платформах, поддерживаемых `Qt`. Пользовательские виджеты могут создаваться чувствительными к стилю либо путем применения `QStyle` (через `QStylePainter`) при рисовании самих себя, либо используя встроенные виджеты `Qt` в качестве дочерних. В `Plotter` мы используем оба подхода: фокусный прямоугольник рисуется с применением `QStyle`, а кнопки `Zoom In` и `Zoom Out` являются встроенными виджетами `Qt`.

```
void Plotter::resizeEvent(QResizeEvent * /* event */)
{
    int x = width() - (zoomInButton->width()
                     + zoomOutButton->width() + 10);
    zoomInButton->move(x, 5);
    zoomOutButton->move(x + zoomInButton->width() + 5, 5);
    refreshPixmap();
}
```

При всяком изменении размера виджета `Plotter` `Qt` генерирует событие «изменение размера». Здесь мы переопределяем функцию `resizeEvent()` для размещения кнопок `ZoomIn` и `ZoomOut` в верхнем правом углу виджета `Plotter`.

Мы располагаем кнопки Zoom In и Zoom Out рядом, отделяя их 5-пиксельным промежутком от верхнего и правого краев родительского виджета.

Если бы нам захотелось оставить эти кнопки в верхнем левом углу, который имеет координаты (0, 0), мы бы просто переместили их туда в конструкторе Plotter. Но мы хотим, чтобы они находились в верхнем правом углу, координаты которого зависят от размеров виджета. По этой причине необходимо переопределить функцию `resizeEvent()` и в ней устанавливать положение кнопок.

Мы не устанавливали положение каких-либо кнопок в конструкторе Plotter. Это сделано из-за того, что Qt всегда генерирует событие изменения размера до первого появления на экране виджета.

В качестве альтернативы переопределению функции `resizeEvent()` и размещению дочерних виджетов «вручную» можно использовать менеджер компоновки (например, `QGridLayout`). При применении менеджеров компоновки это выполнить немного сложнее и такой подход потребовал бы больше ресурсов; с другой стороны, это дало бы элегантное решение компоновки справа налево, что необходимо для таких языков, как арабский и еврейский.

В конце мы вызываем функцию `refreshPixmap()` для перерисовки пиксельной карты с новым размером.

```
void Plotter::mousePressEvent(QMouseEvent *event)
{
    QRect rect(Margin, Margin,
               width() - 2 * Margin, height() - 2 * Margin);
    if (event->button() == Qt::LeftButton) {
        if (rect.contains(event->pos())) {
            rubberBandIsShown = true;
            rubberBandRect.setTopLeft(event->pos());
            rubberBandRect.setBottomRight(event->pos());
            updateRubberBandRegion();
            setCursor(Qt::CrossCursor);
        }
    }
}
```

Когда пользователь нажимает левую кнопку мыши, мы начинаем отображать на экране резиновую ленту. Для этого необходимо установить флажок `rubberBandIsShown` на значение `true`, инициализировать переменную-член `rubberBandRect` на значение текущей позиции курсора мыши, поставить в очередь событие рисования для вычерчивания резиновой ленты и изменить изображение курсора мыши на перекрестие.

Переменная `rubberBandRect` имеет тип `QRect`. Объект `QRect` может задаваться либо четырьмя параметрами (`x`, `y`, `w`, `h`), где (`x`, `y`) является позицией верхнего левого угла и `w` $\times$ `h` определяет размеры четырехугольника, либо парой точек верхнего левого и нижнего правого угла. Здесь мы используем формат с парой точек. То место, где пользователь первый раз щелкнул мышью, становится верхним левым углом, а текущая позиция курсора определяет позицию нижнего правого угла. Затем мы вызываем `updateRubberBandRegion()` для принудительной перерисовки (небольшой) области, покрываемой резиновой лентой.

В Qt предусмотрено два способа управления формой курсора мыши:

- `QWidget::setCursor()` устанавливает форму курсора, которая используется при его нахождении на конкретном виджете. Если для виджета курсор не задан, используется курсор родительского виджета. По умолчанию для виджета верхнего уровня назначается курсор в виде стрелки.
- `QApplication::setOverrideCursor()` устанавливает форму курсора для всего приложения, отменяя формы курсоров отдельных виджетов до вызова функции `restoreOverrideCursor()`.

В главе 4 мы вызывали функцию `QApplication::setOverrideCursor()` с параметром `Qt::WaitCursor` для установки курсора приложения на стандартный курсор ожидания.

```
void Plotter::mouseMoveEvent(QMouseEvent *event)
{
    if (rubberBandIsShown) {
        updateRubberBandRegion();
        rubberBandRect.setBottomRight(event->pos());
        updateRubberBandRegion();
    }
}
```

Когда пользователь перемещает курсор мыши с нажатой левой кнопкой, мы сначала вызываем функцию `updateRubberBandRegion()` для постановки в очередь события рисования для перерисовки области, занятой резиновой лентой, затем пересчитываем значение переменной `rubberBandRect` для учета перемещения курсора и, наконец, второй раз вызываем функцию `updateRubberBandRegion()` для перерисовки области, в которую переместилась резиновая лента. Это фактически приводит к стиранию резиновой ленты и ее вычерчиванию с новыми координатами.

Если пользователь перемещает мышь вверх или влево, может оказаться, что номинальный нижний правый угол резиновой ленты `rubberBandRect` выше или левее верхнего левого угла. В этом случае `QRect` будет иметь отрицательную ширину или высоту. В `paintEvent()` нами использована функция `QRect::normalized()`, которая настраивает координаты верхнего левого и нижнего правого углов для получения положительного значения ширины и высоты.

```
void Plotter::mouseReleaseEvent(QMouseEvent *event)
{
    if ((event->button() == Qt::LeftButton) && rubberBandIsShown) {
        rubberBandIsShown = false;
        updateRubberBandRegion();
        unsetCursor();
        QRect rect = rubberBandRect.normalized();
        if (rect.width() < 4 || rect.height() < 4)
            return;
        rect.translate(-Margin, -Margin);
        PlotSettings prevSettings = zoomStack[curZoom];
        PlotSettings settings;
        double dx = prevSettings.spanX() / (width() - 2 * Margin);
        double dy = prevSettings.spanY() / (height() - 2 * Margin);
        settings.minX = prevSettings.minX + dx * rect.left();
```

```
        settings.maxX = prevSettings.minX + dx * rect.right();
        settings.minY = prevSettings.maxY - dy * rect.bottom();
        settings.maxY = prevSettings.maxY - dy * rect.top();
        settings.adjust();
        zoomStack.resize(curZoom + 1);
        zoomStack.append(settings);
        zoomIn();
    }
}
```

Когда пользователь отпускает левую кнопку мыши, мы стираем резиновую ленту и восстанавливаем стандартный курсор в виде стрелки. Если резиновая лента ограничивает прямоугольник, по крайней мере, размером 4×4 , мы изменяем масштаб изображения. Если резиновая лента выделяет прямоугольник меньшего размера, то, по-видимому, пользователь сделал щелчок мышью по ошибке или просто перевел фокус, и поэтому мы ничего не делаем.

Программный код по изменению масштаба изображения немного сложен. Это вызвано тем, что мы работаем сразу с двумя системами координат: виджета и построителя графиков. Большинство выполняемых здесь действий связано с преобразованием координат объекта `rubberBandRect` (прямоугольник резиновой ленты) из системы координат виджета в систему координат построителя графиков. После выполнения преобразований мы вызываем функцию `PlotSettings::adjust()` для округления чисел и определения разумного количества отметок по обоим осям. Этот процесс отражен на рис. 5.10 и 5.11.

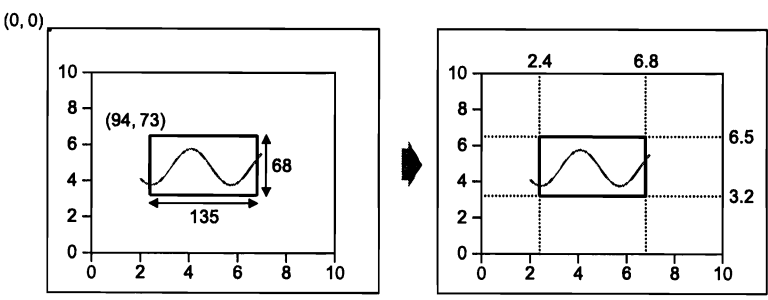


Рис. 5.10. Преобразование прямоугольника резиновой ленты из системы координат виджета в систему координат построителя графиков

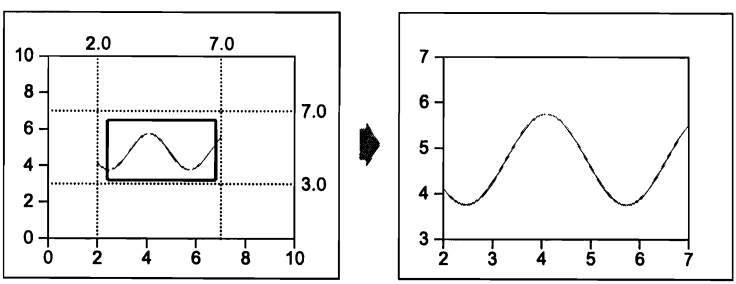


Рис. 5.11. Настройка прямоугольника резиновой ленты в системе координат построителя графиков и увеличение изображения

Затем мы изменяем масштаб изображения. Это достигается путем помещения новых, только что рассчитанных настроек `PlotSettings` в вершину стека масштабов изображения и вызова функции `zoomIn()`, которая выполняет всю остальную работу.

```
void Plotter::keyPressEvent(QKeyEvent *event)
{
    switch (event->key()) {
        case Qt::Key_Plus:
            zoomIn();
            break;
        case Qt::Key_Minus:
            zoomOut();
            break;
        case Qt::Key_Left:
            zoomStack[curZoom].scroll(-1, 0);
            refreshPixmap();
            break;
        case Qt::Key_Right:
            zoomStack[curZoom].scroll(+1, 0);
            refreshPixmap();
            break;
        case Qt::Key_Down:
            zoomStack[curZoom].scroll(0, -1);
            refreshPixmap();
            break;
        case Qt::Key_Up:
            zoomStack[curZoom].scroll(0, +1);
            refreshPixmap();
            break;
        default:
            QWidget::keyPressEvent(event);
    }
}
```

Когда пользователь нажимает на клавиатуре какую-нибудь клавишу и фокус имеет построитель графиков `Plotter`, вызывается функция `keyPressEvent()`. Мы ее переопределяем здесь, чтобы она реагировала на шесть клавиш: `+`, `-`, `Up` (вверх), `Down` (вниз), `Left` (влево) и `Right` (вправо). Если пользователь нажимает другую клавишу, мы вызываем реализацию этой функции из базового класса. Для простоты мы не учитываем ключи модификаторов `Shift`, `Ctrl` и `Alt`, доступ к которым осуществляется с помощью функции `QKeyEvent::modifiers()`.

```
void Plotter::wheelEvent(QWheelEvent *event)
{
    int numDegrees = event->delta() / 8;
    int numTicks = numDegrees / 15;

    if (event->orientation() == Qt::Horizontal) {
```

```

        zoomStack[curZoom].scroll(numTicks, 0);
    } else {
        zoomStack[curZoom].scroll(0, numTicks);
    }
    refreshPixmap();
}

```

События колесика мыши возникают при повороте колесика мышки. В большинстве мышей предусматривается колесико для перемещения по вертикали, но некоторые имеют также колесико для перемещения по горизонтали. Qt поддерживает оба вида колесиков. События колесика мыши передаются виджету, на котором находится фокус. Функция `delta()` возвращает перемещение колесика, выраженное в восьмых долях градуса. Обычно шаг работы колесика мыши составляет 15 градусов. Здесь мы перемещаемся на заданное количество отметок, модифицируя верхний элемент стека масштабов изображений, и обновляем изображение, используя `refreshPixmap()`.

Наиболее распространенное применение колесико мыши получило для продвижения по полосе прокрутки. При использовании нами `QScrollArea` (рассматривается в главе 6) с полосами прокрутки `QScrollArea` автоматически управляет событиями колесика мыши и нам не приходится самим переопределять функцию `wheelEvent()`.

Этим завершается реализация обработчиков событий. Теперь давайте рассмотрим закрытые функции.

```

void Plotter::updateRubberBandRegion()
{
    QRect rect = rubberBandRect.normalized();
    update(rect.left(), rect.top(), rect.width(), 1);
    update(rect.left(), rect.top(), 1, rect.height());
    update(rect.left(), rect.bottom(), rect.width(), 1);
    update(rect.right(), rect.top(), 1, rect.height());
}

```

Функция `updateRubberBand()` вызывается из `mousePressEvent()`, `mouseMoveEvent()` и `mouseReleaseEvent()` для стирания или перерисовки резиновой ленты. Она состоит из четырех вызовов функции `update()`, которая устанавливает в очередь событие рисования для четырех небольших прямоугольных областей, составляющих изображение резиновой ленты (две вертикальные и две горизонтальные линии).

```

void Plotter::refreshPixmap()
{
    pixmap = QPixmap(size());
    pixmap.fill(this, 0, 0);
    QPainter painter(&pixmap);
    painter.initFrom(this);
    drawGrid(&painter);
    drawCurves(&painter);
    update();
}

```

Функция `refreshPixmap()` перерисовывает график на внеэкранной пиксельной карте и обновляет изображение на экране. Мы изменяем размеры пиксельной карты на размеры виджета и заполняем ее цветом стертого виджета. Этот цвет является «темным» компонентом палитры из-за вызова функции `setBackgroundRole()` в конструкторе `Plotter`. Если фон задается неоднородной кистью, в функции `QPixmap::fill()` необходимо указать смещение в виджете, где будет заканчиваться пиксельная карта, чтобы правильно выравнивать образец кисти. Здесь пиксельная карта соответствует всему виджету, поэтому мы задаем позицию `(0, 0)`.

Затем мы создаем `QPainter` для вычерчивания диаграммы на пиксельной карте. Вызов `initFrom()` устанавливает в рисовальщике перо, фон и шрифт такими же, как для виджета `Plotter`. Затем мы вызываем функции `drawGrid()` и `drawCurves()`, которые рисуют диаграмму. В конце мы вызываем функцию `update()` для инициации события рисования всего виджета. Пиксельная карта копируется в виджет функцией `paintEvent()`.

```
void Plotter::drawGrid(QPainter *painter)
{
    QRect rect(Margin, Margin,
               width() - 2 * Margin, height() - 2 * Margin);
    if (!rect.isValid())
        return;

    PlotSettings settings = zoomStack[curZoom];
    QOpen quiteDark = palette().dark().color().light();
    QOpen light = palette().light().color();

    for (int i = 0; i <= settings.numXTicks; ++i) {
        int x = rect.left() + (i * (rect.width() - 1)
                               / settings.numXTicks);
        double label = settings.minX + (i * settings.spanX()
                                         / settings.numXTicks);
        painter->setPen(quiteDark);
        painter->drawLine(x, rect.top(), x, rect.bottom());
        painter->setPen(light);
        painter->drawLine(x, rect.bottom(), x, rect.bottom() + 5);
        painter->drawText(x - 50, rect.bottom() + 5, 100, 20,
                         Qt::AlignHCenter | Qt::AlignTop,
                         QString::number(label));
    }
    for (int j = 0; j <= settings.numYTicks; ++j) {
        int y = rect.bottom() - (j * (rect.height() - 1)
                                 / settings.numYTicks);
        double label = settings.minY + (j * settings.spanY()
                                         / settings.numYTicks);
        painter->setPen(quiteDark);
        painter->drawLine(rect.left(), y, rect.right(), y);
        painter->setPen(light);
        painter->drawLine(rect.left() - 5, y, rect.left(), y);
    }
}
```

```

painter->drawText(rect.left() - Margin, y - 10, Margin - 5, 20,
                  Qt::AlignRight | Qt::AlignVCenter,
                  QString::number(label));
}
painter->drawRect(rect.adjusted(0, 0, -1, -1));
}

```

Функция `drawGrid()` чертит сетку под кривыми и осями. Область для вычерчивания сетки задается прямоугольником `rect`. Если размеры виджета недостаточны для размещения графика, мы сразу возвращаем управление.

Первый цикл `for` выводит вертикальные линии сетки и отметки по оси *x*. Второй цикл `for` выводит горизонтальные линии и отметки по оси *y*. В конце мы рисуем прямоугольники по окаймляющей кромке. Функция `drawText()` применяется для вывода числовых значений для отметок обеих осей.

Вызовы функции `drawText()` имеют следующий формат:

```
painter.drawText(x, y, ширина, высота, смещение, текст);
```

где (*x*, *y*, ширина, высота) определяют прямоугольник, смещение задает позицию текста в этом прямоугольнике и текст представляет собой выводимый текст. В этом примере мы вычислили координаты прямоугольника, в котором будет выводиться текст, вручную; более приемлемой альтернативой было бы вычисление прямоугольника, ограничивающего текст, при помощи класса `QFontMetrics`.

```

void Plotter::drawCurves(QPainter *painter)
{
    static const QColor colorForIds[6] = {
        Qt::red, Qt::green, Qt::blue, Qt::cyan, Qt::magenta, Qt::yellow
    };
    PlotSettings settings = zoomStack[curZoom];
    QRect rect(Margin, Margin,
               width() - 2 * Margin, height() - 2 * Margin);
    if (!rect.isValid())
        return;

    painter->setClipRect(rect.adjusted(+1, +1, -1, -1));
    QMapIterator<int, QVector<QPointF> > i(curveMap);
    while (i.hasNext()) {
        i.next();

        int id = i.key();
        QVector<QPointF> data = i.value();
        QPolygonF polyline(data.count());

        for (int j = 0; j < data.count(); ++j) {
            double dx = data[j].x() - settings.minX;
            double dy = data[j].y() - settings.minY;
            double x = rect.left() + (dx * (rect.width() - 1)
                                     / settings.spanX());
            double y = rect.bottom() - (dy * (rect.height() - 1)

```

```

                                / settings.spanY());
        polyline[j] = QPointF(x, y);
    }
    painter->setPen(colorForIds[uint(id) % 6]);
    painter->drawPolyline(polyline);
}
}

```

Функция `drawCurves()` рисует кривые поверх сетки. Мы начинаем с вызова функции `setClipRect()` для ограничения области отображения `QPainter` прямоугольником, содержащим кривые (без окаймляющей кромки и рамки вокруг графика). После этого `QPainter` будет игнорировать вывод пикселей вне этой области.

Затем мы выполняем цикл по всем кривым, используя итератор в стиле Java, и для каждой кривой выполняем цикл по ее точкам `QPointF`. Мы вызываем функцию итератора `key()`, чтобы получить идентификатор кривой, и функцию `value()` для получения данных соответствующей кривой в виде вектора `QVector<QPointF>`. Внутри цикла `for` производится преобразование всех точек `QPointF` из системы координат построителя графика в систему координат виджета и сохранение их в переменной `polyline`.

После преобразования всех точек кривой в систему координат виджета мы устанавливаем цвет пера для кривой (используя один из наборов заранее определенных цветов) и вызываем `drawPolyline()` для вычерчивания линии, которая проходит по всем точкам кривой.

Этим мы завершаем построение класса `Plotter`. Остается только рассмотреть несколько функций настроек графика `PlotSettings`.

```

PlotSettings::PlotSettings()
{
    minX = 0.0;
    maxX = 10.0;
    numXTicks = 5;

    minY = 0.0;
    maxY = 10.0;
    numYTicks = 5;
}

```

Конструктор `PlotSettings` инициализирует обе оси координат диапазоном от 0 до 10 с пятью отметками.

```

void PlotSettings::scroll(int dx, int dy)
{
    double stepX = spanX() / numXTicks;
    minX += dx * stepX;
    maxX += dx * stepX;

    double stepY = spanY() / numYTicks;
    minY += dy * stepY;
    maxY += dy * stepY;
}

```

Функция `scroll()` увеличивает (или уменьшает) `minX`, `maxX`, `minY` и `maxY` на интервал между двух отметок, помноженный на заданное число. Данная функция применяется для реализации скроллинга в функции `Plotter::keyPressEvent()`.

```
void PlotSettings::adjust()
{
    adjustAxis(minX, maxX, numXTicks);
    adjustAxis(minY, maxY, numYTicks);
}
```

Функция `adjust()` вызывается из `mouseReleaseEvent()` для округления значений `minX`, `maxX`, `minY` и `maxY`, чтобы получить «удобные» значения, и определения количества меток на каждой оси. Закрытая функция `adjustAxis()` выполняет эти действия отдельно для каждой оси.

```
void PlotSettings::adjustAxis(double &min, double &max,
                             int &numTicks)
{
    const int MinTicks = 4;
    double grossStep = (max - min) / MinTicks;
    double step = std::pow(10.0, std::floor(std::log10(grossStep)));

    if (5 * step < grossStep) {
        step *= 5;
    } else if (2 * step < grossStep) {
        step *= 2;
    }

    numTicks = int(std::ceil(max / step) - std::floor(min / step));
    if (numTicks < MinTicks)
        numTicks = MinTicks;
    min = std::floor(min / step) * step;
    max = std::ceil(max / step) * step;
}
```

Функция `adjustAxis()` преобразует свои параметры `min` и `max` в «удобные» числа и устанавливает свой параметр `numTicks` на количество меток, которое по ее расчету подходит для заданного диапазона `[min, max]`. Поскольку в функции `adjustAxis()` фактически требуется модифицировать переменные (`minX`, `maxX`, `numXTicks` и т. д.), а не просто копировать их, для этих параметров не используется модификатор `const`.

Большая часть программного кода в `adjustAxis()` предназначена просто для определения соответствующего значения интервала между двумя метками (переменная `step` – шаг). Для получения на оси удобных чисел мы должны тщательно выбирать этот шаг. Например, значение шага 3.8 привело бы к появлению на оси чисел, кратных 3.8, что затрудняет восприятие диаграммы человеком. Для осей с десятичной системой обозначения «удобными» значениями шага являются числа вида 10^n , $2 \cdot 10^n$ или $5 \cdot 10^n$.

Мы начинаем расчет с «крупного шага», то есть с определенного максимального значения шага. Затем мы находим соответствующее число вида 10^n , мень-

шее или равное крупному шагу. Мы его получаем путем взятия десятичного логарифма от крупного шага, затем округляем полученное значение до целого числа, после чего возводим 10 в степень равную этому округленному значению. Например, если крупный шаг равен 236, мы вычисляем $\log 236 = 2.37291\dots$; затем округляем это значение до 2 и получаем $10^2 = 100$ в качестве кандидата на значение шага в форме числа 10^n .

После получения первого кандидата на значение шага мы можем его использовать для расчета двух других кандидатов: $2 \cdot 10^n$ и $5 \cdot 10^n$. Для нашего примера два других кандидата являются числами 200 и 500. Кандидат 500 имеет значение большее, чем крупный шаг, и поэтому мы не можем использовать его. Но 200 меньше, чем 236, и поэтому мы можем использовать 200 в качестве размера шага в нашем примере.

Достаточно легко получить `numTicks`, `min` и `max` из значения шага. Новое значение `min` получается путем округления снизу первоначального `min` до ближайшего числа, кратного этому шагу, а новое значение `max` получается путем округления до ближайшего числа, кратного этому шагу. Новое значение `numTicks` представляет собой количество интервалов между округленными значениями `min` и `max`. Например, если при входе в функцию `min` равно 240, а `max` равно 1184, то новый диапазон будет равен [200, 1200] с пятью отметками.

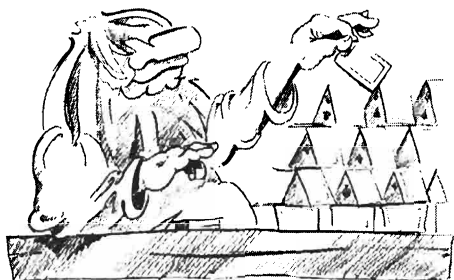
Этот алгоритм в некоторых случаях даст почти оптимальный результат. Более изощренный алгоритм описан в статье Поля С. Хекберта (Paul S. Heckbert) «Nice Numbers for Graph Labels» (удобные числа для меток графа), опубликованной в *Graphics Gems* (Morgan Kaufmann, 1990).

Данная глава является последней в части I нашей книги. В ней объяснены способы настройки существующего виджета Qt и способы построения виджета с использованием в качестве основы базового класса виджетов `QWidget`. В главе 2 мы уже узнали, как компоновать виджеты-потомки с помощью менеджеров компоновки, и мы еще вернемся к этой теме в главе 6.

К этому моменту у нас достаточно знаний для написания законченных приложений с графическим интерфейсом с помощью средств разработки Qt. В частях II и III мы проведем более глубокое исследование Qt, чтобы можно было в полной мере использовать возможности Qt.

Часть II

Средний уровень Qt-программирования



- *Компоновка виджетов на форме*
- *Стековая компоновка*
- *Разделители*
- *Области с прокруткой*
- *Прикрепляемые окна и панели инструментов*
- *Многодокументный интерфейс*

Глава 6. Управление компоновкой

Каждому размещаемому в форме виджету необходимо задать соответствующий размер и позицию. Qt содержит несколько классов, обеспечивающих компоновку виджетов на форме: `QHBoxLayout`, `QVBoxLayout`, `QGridLayout` и `QStackedLayout`. Эти классы настолько удобно и просто применять, что почти каждый Qt-разработчик их использует либо непосредственно в исходном коде программы, либо через *Qt Designer*.

Другая причина применения классов Qt по компоновке виджетов – гарантия автоматической адаптации формы к различным шрифтам, языкам и платформам. Если пользователь изменяет настройки шрифта системы, формы приложения немедленно на это отреагируют, изменяя при необходимости свои размеры. И если вы переводите интерфейс пользователя приложения на другие языки, классы компоновки будут учитывать содержание переведенных виджетов, чтобы избежать усечения текста.

К другим классам, управляющим компоновкой, относятся `QSplitter`, `QScrollArea`, `QMainWindow` и `QMdiArea`. Все эти классы обеспечивают гибкую компоновку виджетов, которой может управлять пользователь. Например, `QSplitter` обеспечивает наличие разделительной линии, которую пользователь может передвигать для изменения размеров виджетов, а `QMdiArea` обеспечивает поддержку MDI (multiple document interface – многодокументный интерфейс), позволяющего в главном окне приложения показывать сразу несколько документов. Поскольку эти классы часто используются как альтернатива основным классам компоновки, мы их также рассмотрим в данной главе.

Компоновка виджетов на форме

Существует три основных способа управления компоновкой дочерних виджетов формы: абсолютное позиционирование, ручная компоновка и применение менеджеров компоновки. Мы рассмотрим по очереди каждый из этих методов, используя в качестве нашего примера диалоговое окно Find File, показанное на рис. 6.1.



Рис. 6.1. Окно диалога Find File

Абсолютное позиционирование является самым негибким способом компоновки виджетов. Он предусматривает жесткое кодирование в программе размеров и позиций дочерних виджетов формы и фиксированный размер самой формы. Ниже показано, какой вид принимает конструктор `FindFileDialog` при применении абсолютного позиционирования.

```
FindFileDialog::FindFileDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    namedLabel->setGeometry(9, 9, 50, 25);
    namedLineEdit->setGeometry(65, 9, 200, 25);
    lookInLabel->setGeometry(9, 40, 50, 25);
    lookInLineEdit->setGeometry(65, 40, 200, 25);
    subfoldersCheckBox->setGeometry(9, 71, 256, 23);
    tableWidget->setGeometry(9, 100, 256, 100);
    messageLabel->setGeometry(9, 206, 256, 25);
    findButton->setGeometry(271, 9, 85, 32);
    stopButton->setGeometry(271, 47, 85, 32);
    closeButton->setGeometry(271, 84, 85, 32);
    helpButton->setGeometry(271, 199, 85, 32);

    setWindowTitle(tr("Find Files or Folders"));
    setFixedSize(365, 240);
}
```

Абсолютное позиционирование имеет много недостатков:

- пользователь не может изменить размер окна;
- некоторый текст может оказаться отсеченным, если пользователь выбирает необычно большой шрифт или если приложение переводится на другой язык;
- виджеты могут иметь неправильные размеры для некоторых стилей;
- расчет позиций и размеров должен производиться вручную. Этот процесс утомителен и приводит к ошибкам; кроме того, это сильно затрудняет сопровождение.

В качестве альтернативы абсолютному позиционированию используется ручная компоновка. При ручной компоновке виджетам все же придаются абсолютные позиции, но размеры виджетов становятся пропорциональны размеру окна, а не жестко кодируются в программе. Это может достигаться путем переопределения функции формы `resizeEvent()` для установки геометрических размеров своих дочерних виджетов.

```
FindFileDialog::FindFileDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    setMinimumSize(265, 190);
    resize(365, 240);
}

void FindFileDialog::resizeEvent(QResizeEvent * /* event */)
{
    int extraWidth = width() - minimumWidth();
    int extraHeight = height() - minimumHeight();
    namedLabel->setGeometry(9, 9, 50, 25);
    namedLineEdit->setGeometry(65, 9, 100 + extraWidth, 25);
    lookInLabel->setGeometry(9, 40, 50, 25);
    lookInLineEdit->setGeometry(65, 40, 100 + extraWidth, 25);
    subfoldersCheckBox->setGeometry(9, 71, 156 + extraWidth, 23);
    tableWidget->setGeometry(9, 100, 156 + extraWidth,
                             50 + extraHeight);
    messageLabel->setGeometry(9, 156 + extraHeight, 156 + extraWidth,
                             25);
    findButton->setGeometry(171 + extraWidth, 9, 85, 32);
    stopButton->setGeometry(171 + extraWidth, 47, 85, 32);
    closeButton->setGeometry(171 + extraWidth, 84, 85, 32);
    helpButton->setGeometry(171 + extraWidth, 149 + extraHeight, 85,
                           32);
}
```

Мы устанавливаем в конструкторе `FindFileDialog` минимальный размер формы на значение 265×190 и ее начальный размер на значение 365×240 . В обработчике событий `resizeEvent()` мы отдаем все дополнительное пространство виджетам, размеры которых мы хотим увеличить. Это обеспечивает плавное изменение вида формы при изменении пользователем ее размеров.

Точно так же как при абсолютном позиционировании, при ручной компоновке в программе приходится жестко задавать много констант, рассчитываемых программистом. Написание подобной программы представляет собой нудное занятие, особенно если проект изменяется. И все-таки существует риск отсечения текста. Этого риска можно избежать, принимая во внимание идеальные размеры дочерних виджетов, но это еще больше усложняет программу.

Самый удобный метод компоновки виджетов на форме – использование менеджеров компоновки Qt. Менеджеры компоновки обеспечивают осмысленные

принимаемые по умолчанию значения параметров для каждого типа виджета и учитывают идеальный размер каждого виджета, который в свою очередь обычно зависит от шрифта виджета, его стиля и содержимого. Менеджеры компоновки также учитывают максимальные и минимальные размеры и автоматически подстраивают компоновку в ответ на изменения шрифта, изменения содержимого и изменения размеров окна. Версия диалогового окна Find File с изменяемыми размерами показана на рис. 6.2.

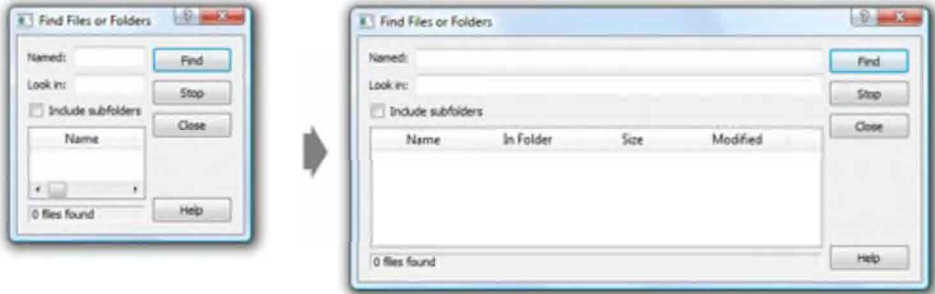


Рис. 6.2. Изменение размеров диалогового окна, допускающего изменение своих размеров

Существует три наиболее важных менеджера компоновки: `QHBoxLayout`, `QVBoxLayout` и `QGridLayout`. Эти классы происходят от `QLayout`, который обеспечивает основной каркас для менеджеров компоновки. Все эти три класса полностью поддерживаются *Qt Designer* и могут также использоваться непосредственно в программе.

Ниже приводится программный код `FindFileDialog`, в котором используются менеджеры компоновки.

```
FindFileDialog::FindFileDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    QGridLayout *leftLayout = new QGridLayout;
    leftLayout->addWidget(namedLabel, 0, 0);
    leftLayout->addWidget(namedLineEdit, 0, 1);
    leftLayout->addWidget(lookInLabel, 1, 0);
    leftLayout->addWidget(lookInLineEdit, 1, 1);
    leftLayout->addWidget(subfoldersCheckBox, 2, 0, 1, 2);
    leftLayout->addWidget(tableWidget, 3, 0, 1, 2);
    leftLayout->addWidget(messageLabel, 4, 0, 1, 2);

    QVBoxLayout *rightLayout = new QVBoxLayout;
    rightLayout->addWidget(findButton);
    rightLayout->addWidget(stopButton);
    rightLayout->addWidget(closeButton);
    rightLayout->addStretch();
    rightLayout->addWidget(helpButton);
}
```



```

QHBoxLayout *mainLayout = new QHBoxLayout;
mainLayout->addLayout(leftLayout);
mainLayout->addLayout(rightLayout);
setLayout(mainLayout);

setWindowTitle(tr("Find Files or Folders"));
}

```

Компоновка обеспечивается одним менеджером компоновки по горизонтали – QHBoxLayout, одним менеджером компоновки в ячейках сетки – QGridLayout и одним менеджером компоновки по вертикали – QVBoxLayout. Менеджер QGridLayout слева и менеджер QVBoxLayout справа размещаются рядом внутри внешнего менеджера QHBoxLayout. Кромка по периметру диалогового окна и промежутки между дочерними виджетами устанавливаются в значения по умолчанию, которые зависят от текущего стиля виджета; они могут быть изменены, используя функции `QLayout::setContentsMargins()` и `QLayout::setSpacing()`.

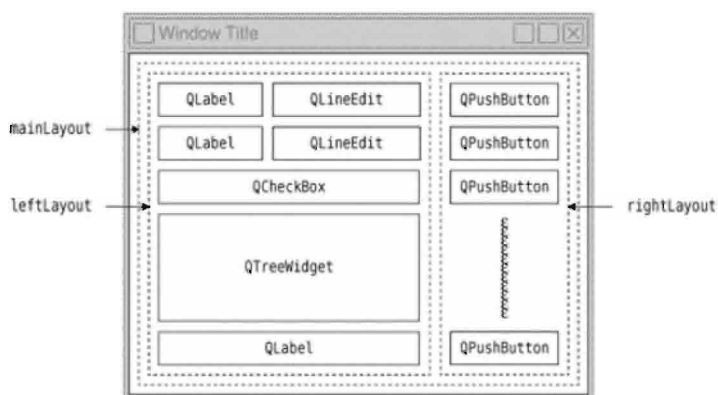


Рис. 6.3. Компоновка диалогового окна Find File

Такое же диалоговое окно можно было бы создать с помощью визуальных средств разработки *Qt Designer*, задавая приблизительное положение дочерним виджетам, выделяя те, которые необходимо расположить рядом, и выбирая пункты меню *Form|Lay Out Horizontally*, *Form|Lay Out Vertically* или *Form|Lay Out in a Grid*. Мы использовали данный подход в главе 2 для создания диалоговых окон *Go-to-Cell* и *Sort приложения Электронная таблица*.

Применение QHBoxLayout и QVBoxLayout достаточно очевидное, однако с QGridLayout дело обстоит несколько сложнее. Менеджер QGridLayout работает с двумерной сеткой ячеек. Текстовая метка QLabel, расположенная в верхнем левом углу этого менеджера компоновки, имеет координаты (0, 0), а соответствующая строка редактирования QLineEdit имеет координаты (0, 1). Флажок QCheckBox размещается в двух столбцах; он занимает ячейки с координатами (2, 0) и (2, 1). Расположенные под ним объекты QTreeWidget и QLabel также занимают два столбца. Вызовы функции `QGridLayout::addWidget()` имеют следующий формат:

```
layout->addWidget(виджет, строка, столбец, колСтрок, колСтолбцов);
```

Здесь виджет является дочерним виджетом, который вставляется в менеджер компоновки, (строка, столбец) – координаты верхней левой ячейки, занимаемой виджетом, колСтрок – количество строк, занимаемое виджетом, и колСтолбцов – количество столбцов, занимаемое виджетом. Если аргументы колСтрок и колСтолбцов не заданы, они принимают значение по умолчанию, равное 1.

Вызов `addStretch()` говорит менеджеру вертикальной компоновки о необходимости выделения свободного пространства в данной точке. Добавив элемент распорки, мы заставляем менеджера компоновки выделить дополнительное пространство между кнопкой Close и кнопкой Help. В *Qt Designer* мы можем добиться того же самого эффекта, вставляя распорку. Распорки в *Qt Designer* отображаются в виде синих «пружинок».

Помимо рассмотренных нами до сих пор случаев использование менеджеров компоновки дает дополнительные выгоды. Если мы добавляем виджет к менеджеру или убираем виджет из него, менеджер компоновки автоматически адаптируется к новой ситуации. То же самое происходит, если мы вызываем `hide()` или `show()` для дочернего виджета. Если идеальный размер дочернего виджета изменяется, компоновка автоматически перестраивается, учитывая новый идеальный размер. Кроме того, менеджеры компоновки автоматически устанавливают минимальный размер всей формы на основе минимальных размеров и идеальных размеров дочерних виджетов формы.

В представленных до сих пор примерах мы просто помещали виджеты в менеджеры и использовали распорки для выделения дополнительного пространства. Иногда этого недостаточно для того, чтобы компоновка приняла нужный нам вид. В таких ситуациях мы можем настроить компоновку, изменяя политику размеров и идеальные размеры размещаемых виджетов.

Политика размера виджета говорит системе компоновки, как его следует растягивать или сжимать. Qt обеспечивает разумные принимаемые по умолчанию значения политик размеров для всех своих встроенных виджетов, но поскольку ни одно принимаемое по умолчанию значение не может учесть всевозможные варианты компоновки, все-таки обычной практикой для разработчиков является изменение политики размеров одного или двух виджетов формы. `QSizePolicy` имеет как горизонтальный, так и вертикальный компоненты. Ниже приводятся наиболее полезные значения.

- **Fixed** (фиксированное) означает, что виджет не может увеличиваться или сжиматься. Размер виджета всегда сохраняет значение его идеального размера.
- **Minimum** означает, что идеальный размер виджета является его минимальным размером. Размер виджета не может стать меньше идеального размера, но он может при необходимости вырасти для заполнения доступного пространства.
- **Maximum** означает, что идеальный размер виджета является его максимальным размером. Размер виджета может уменьшаться до его минимального идеального размера.
- **Preferred** (предпочитаемое) означает, что идеальный размер виджета является его предпочитаемым размером, но виджет может при необходимости сжиматься или растягиваться.
- **Expanding** (расширяемый) означает, что виджет может сжиматься или растягиваться, но в первую очередь он стремится увеличить свои размеры.

На рис. 6.4 приводится иллюстрация смысла различных политик размеров, причем в качестве примера здесь используется текстовая метка QLabel с текстом «Какой-то текст».

На рисунке политики Preferred и Expanding представлены одинаково. Так в чем же их отличие? При изменении размеров формы, содержащей одновременно виджеты с политикой размера Preferred и Expanding, дополнительное пространство отдается виджетам Expanding, а виджеты Preferred по-прежнему будут иметь свой идеальный размер.

Существует еще две политики размеров: MinimumExpanding и Ignored. Первая была необходима в некоторых редких случаях для старых версий Qt, но теперь она не применяется; предпочтительнее использовать политику Expanding и соответствующим образом переопределить функцию `minimumSizeHint()`. Последняя напоминает Expanding, но при этом игнорируется идеальный размер виджета и минимальный идеальный его размер.

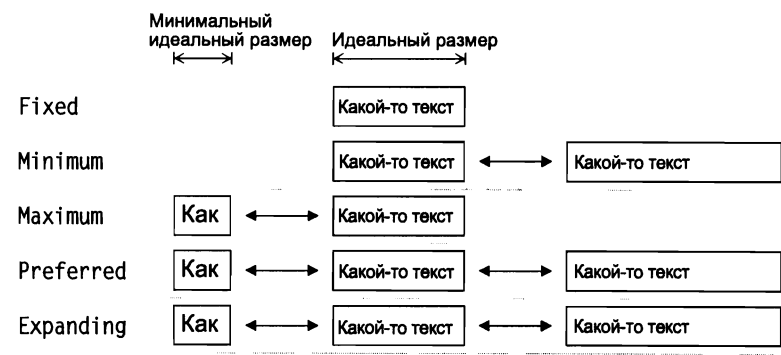


Рис. 6.4. Смысл различных политик размеров

Кроме горизонтального и вертикального компонентов политики размеров класс `QSizePolicy` хранит коэффициенты растяжения по горизонтали и вертикали. Эти коэффициенты растяжения могут использоваться для указания того, что различные дочерние виджеты могут растягиваться по-разному при расширении формы. Например, если `QTreeWidget` располагается над `QTextEdit` и мы хотим, чтобы `QTextEdit` был в два раза больше по высоте, чем `QTreeWidget`, мы можем установить коэффициент растяжения по вертикали для `QTextEdit` на значение 2, а тот же коэффициент для `QTreeWidget` на значение 1.

Другой способ воздействия на компоновку заключается в установке минимального размера, максимального размера или фиксированного размера дочерних виджетов. Менеджер компоновки будет учитывать эти ограничения при компоновке виджетов. Но если этого недостаточно, мы можем всегда создать подкласс дочернего виджета и переопределить функцию `sizeHint()` для получения необходимого нам идеального размера.

Стековая компоновка

Класс `QStackedLayout` (менеджер стековой компоновки) управляет компоновкой набора дочерних виджетов или «страниц», показывая в каждый конкретный момент только одну из них и скрывая от пользователя остальные. Сам менеджер

`QStackedLayout` невидим и не содержит внутри себя средства для пользователя по изменению страницы. Показанные на рис. 6.5 небольшие стрелки и темно-серая рамка обеспечиваются *Qt Designer*, чтобы упростить применение этого менеджера компоновки при проектировании формы. Для удобства в *Qt* предусмотрен класс `QStackedWidget`, представляющий собой `QWidget` с встроенным `QStackedLayout`.

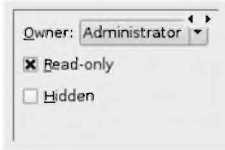


Рис. 6.5. `QStackedLayout`

Страницы нумеруются с 0. Если мы хотим сделать какой-нибудь конкретный виджет видимым, мы можем вызвать функцию `setCurrentIndex()`, задавая номер страницы. Номер страницы дочернего виджета можно получить с помощью функции `indexOf()`.

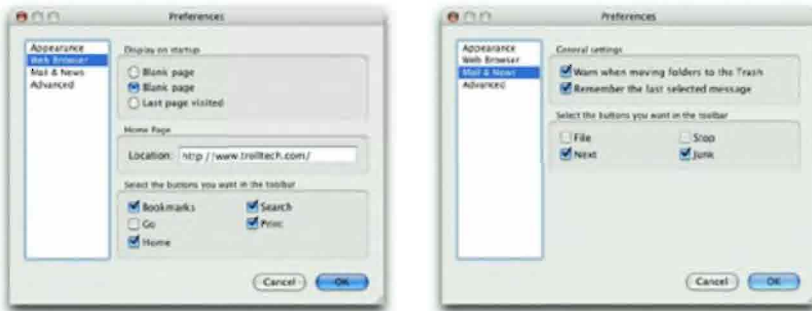


Рис. 6.6. Две страницы диалогового окна `Preferences`

Показанное на рис. 6.6 диалоговое окно `Preferences` (настройка предпочтений) представляет собой пример использования `QStackedLayout`. Окно диалога состоит из виджета `QListWidget` слева и менеджера стековой компоновки `QStackedLayout` справа. Каждый элемент в списке `QListWidget` соответствует одной странице `QStackedLayout`. Ниже приводится соответствующий программный код конструктора этого диалогового окна.

```

PreferenceDialog::PreferenceDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    listWidget = new QListWidget;
    listWidget->addItem(tr("Appearance"));
    listWidget->addItem(tr("Web Browser"));
    listWidget->addItem(tr("Mail & News"));
    listWidget->addItem(tr("Advanced"));

```

```
stackedLayout = new QStackedLayout;
stackedLayout->addWidget(appearancePage);
stackedLayout->addWidget(webBrowserPage);
stackedLayout->addWidget(mailAndNewsPage);
stackedLayout->addWidget(advancedPage);
connect(listWidget, SIGNAL(currentRowChanged(int)),
        stackedLayout, SLOT(setCurrentIndex(int)));
...
listWidget->setCurrentRow(0);
}
```

Мы создаем `QListWidget` и заполняем его названиями страниц. Затем создаем `QStackedLayout` и вызываем для каждой страницы функцию `addWidget()`. Мы связываем сигнал спискового виджета `currentRowChanged(int)` с `setCurrentIndex(int)` менеджера стековой компоновки для переключения страниц и вызываем функцию спискового виджета `setCurrentRow()` в конце конструктора, чтобы начать со страницы 0.

Подобные формы также очень легко создавать при помощи *Qt Designer*.

1. Создайте новую форму на основе одного из шаблонов «Dialog» или на основе шаблона «Widget».
2. Добавьте в форму виджеты `QListWidget` и `QStackedWidget`.
3. Заполните каждую страницу дочерними виджетами и менеджерами компоновки.
(Для создания новой страницы нажмите на правую кнопку мыши и выберите пункт меню *Insert Page* (вставить страницу); для перехода с одной страницы на другую щелкните по маленькой левой или правой стрелке, расположенной в верхнем правом углу виджета `QStackedWidget`.)
4. Расположите виджеты рядом, используя менеджер горизонтальной компоновки.
5. Подсоедините сигнал виджета списка элементов `currentRowChanged(int)` к слоту стекового виджета `setCurrentIndex (int)`.
6. Установите значение свойства виджета списка элементов `currentRow` на 0.

Поскольку мы реализовали переключение страниц с помощью предварительно определенных сигналов и слотов, диалоговое окно будет правильно работать при предварительном просмотре в *Qt Designer*. В тех случаях, где число страниц невелико и, скорее всего, останется небольшим, более простой альтернативой использованию `QStackedWidget` и `QListWidget` будет использование `QTabWidget`.

Разделители

Разделитель `QSplitter` представляет собой виджет, который содержит другие виджеты. Виджеты и разделители отделены друг от друга разделительными линиями. Пользователи могут изменять размеры дочерних виджетов разделителя посредством перемещения разделительных линий. Разделители могут часто использоваться в качестве альтернативы менеджерам компоновки, предоставляя пользователю больше возможностей по управлению компоновкой.



Рис. 6.7. Приложение Splitter

Дочерние виджеты `QSplitter` автоматически располагаются рядом (или один под другим) в порядке их создания, причем между соседними виджетами размещаются разделительные линии. Ниже приводится программный код для создания представленного на рис. 6.7 окна.

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

    QTextEdit *editor1 = new QTextEdit;
    QTextEdit *editor2 = new QTextEdit;
    QTextEdit *editor3 = new QTextEdit;

    QSplitter splitter(Qt::Horizontal);
    splitter.addWidget(editor1);
    splitter.addWidget(editor2);
    splitter.addWidget(editor3);
    ...
    splitter.show();
    return app.exec();
}
```

Этот пример состоит из трех полей редактирования `QTextEdit`, расположенных горизонтально в виджете `QSplitter`. Они схематично показаны на рис. 6.8. В отличие от менеджеров компоновки, которые просто размещают в форме дочерние виджеты, а сами не имеют визуального представления, `QSplitter` происходит от `QWidget` и может использоваться как любой другой виджет.

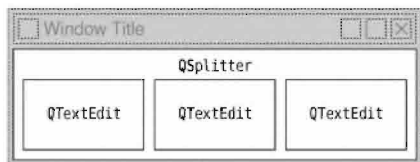


Рис. 6.8. Виджеты приложения Splitter

Можно обеспечить сложную компоновку путем применения вложенных горизонтальных и вертикальных разделителей `QSplitter`. Например, показанное на рис. 6.9 приложение Mail Client (почтовый клиент) состоит из горизонталь-

ного QSplitter, который содержит справа от себя вертикальный QSplitter. Эта компоновка схематично показана на рис. 6.10.

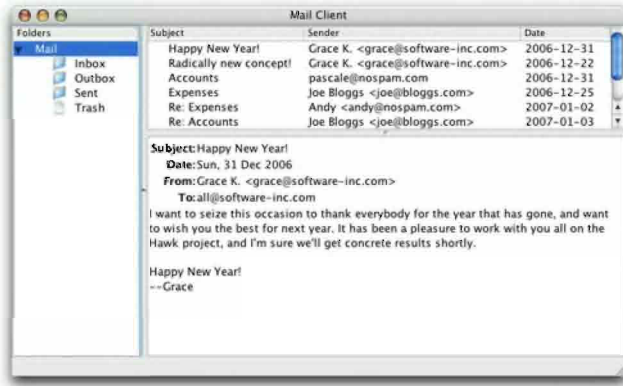


Рис. 6.9. Приложение Mail Client

Ниже приводится программный код конструктора подкласса QMainWindow приложения Mail Client.

```
MailClient::MailClient()
{
    ...
    rightSplitter = new QSplitter(Qt::Vertical);
    rightSplitter->addWidget(messagesTreeWidget);
    rightSplitter->addWidget(textEdit);
    rightSplitter->setStretchFactor(1, 1);

    mainSplitter = new QSplitter(Qt::Horizontal);
    mainSplitter->addWidget(foldersTreeWidget);
    mainSplitter->addWidget(rightSplitter);
    mainSplitter->setStretchFactor(1, 1);
    setCentralWidget(mainSplitter);
    setWindowTitle(tr("Mail Client"));
    readSettings();
}
```

После создания трех виджетов, которые мы собираемся выводить на экран, мы создаем вертикальный разделитель, `rightSplitter`, и добавляем два виджета, которые мы собираемся отображать справа. Затем мы создаем горизонтальный разделитель, `mainSplitter`, и добавляем виджет, который мы хотим отображать слева, и `rightSplitter`, виджеты которого мы хотим показывать справа. Мы делаем `mainSplitter` центральным виджетом QMainWindow.

Когда пользователь изменяет размер окна, QSplitter обычно распределяет пространство таким образом, что относительные размеры дочерних виджетов остаются прежними. В примере приложения Mail Client нам не нужен такой режим работы; вместо этого мы хотим, чтобы QTreeWidget и QTableWidget сохраняли

свои размеры, и мы хотим отдавать любое дополнительное пространство полю редактирования `QTextEdit`. Это достигается с помощью двух вызовов функции `setStretchFactor()`. В первом аргументе задается индекс дочернего виджета разделителя (индексация начинается с нуля), а во втором аргументе – коэффициент растяжения; по умолчанию используется 0.

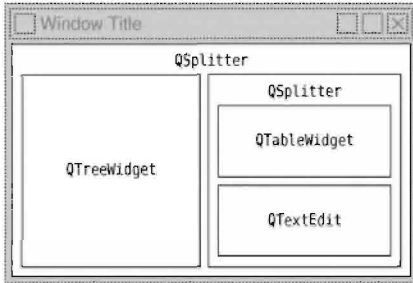


Рис. 6.10. Компоновка разделителя в приложении Mail Client

Первый вызов `setStretchFactor()` делается для `rightSplitter`, устанавливая виджет в позицию 1 (`textEdit`) и коэффициент растяжения на 1. Второй вызов `setStretchFactor()` делается для `mainSplitter`, устанавливая виджет в позицию 1 (`rightSplitter`) и коэффициент растяжения на 1. Это обеспечивает получение всего дополнительного пространства полем редактирования `textEdit`.

При запуске приложения разделитель `QSplitter` задает дочерним виджетам соответствующие размеры на основе их первоначального размера (или на основе их идеального размера, если начальный размер не указан). Мы можем передвигать разделительные линии программно, вызывая функцию `QSplitter::setSizes()`. Класс `QSplitter` предоставляет также средство сохранения своего состояния и его восстановления при следующем запуске приложения. Ниже приводится функция `writeSettings()`, которая сохраняет настройки Mail Client.

```
void MailClient::writeSettings()
{
    QSettings settings("Software Inc.", "Mail Client");

    settings.beginGroup("mainWindow");
    settings.setValue("geometry", saveGeometry());
    settings.setValue("mainSplitter", mainSplitter->saveState());
    settings.setValue("rightSplitter", rightSplitter->saveState());
    settings.endGroup();
}
```

Ниже приводится соответствующая функция по чтению настроек `readSettings()`.

```
void MailClient::readSettings()
{
    QSettings settings("Software Inc.", "Mail Client");

    settings.beginGroup("mainWindow");
```



```

restoreGeometry(settings.value("geometry").toByteArray());
mainSplitter->restoreState(
    settings.value("mainSplitter").toByteArray());
rightSplitter->restoreState(
    settings.value("rightSplitter").toByteArray());
settings.endGroup();
}

```

Qt Designer полностью поддерживает разделитель `QSplitter`. Для размещения виджетов в разделителе поместите дочерние виджеты приблизительно в то место, где они должны находиться, выделите их и выберите пункт меню `Form|Lay Out Horizontally in Splitter` или `Form|Lay Out Vertically in Splitter` (**Форма | Компоновка по горизонтали в разделитель** или **Форма | Компоновка по вертикали в разделитель**).

Области с прокруткой

Класс `QScrollArea` содержит область отображения, которую можно прокручивать, и две полосы прокрутки. Если мы хотим добавить в виджет полосы прокрутки, значительно проще использовать класс `QScrollArea`, чем создавать свои собственные экземпляры `QScrollBar` и самим реализовывать функциональность скроллинга.

Способ применения `QScrollArea` состоит в следующем: вызывается функция `setWidget()` с виджетом, к которому мы хотим добавить полосы прокрутки. `QScrollArea` автоматически делает этот виджет дочерним (если он еще не является таковым) по отношению к области отображения (он доступен при помощи функции `QScrollArea::viewport()`). Например, если мы хотим иметь полосы прокрутки вокруг виджета `IconEditor`, который мы разработали в главе 5 (показан на рис. 6.11), мы можем написать такую программу:

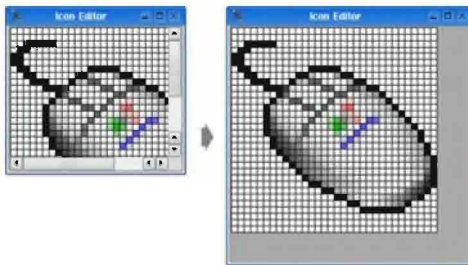


Рис. 6.11. Изменение размера `QScrollArea`

```

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    IconEditor *iconEditor = new IconEditor;
    iconEditor->setIconImage(QImage(":/images/mouse.png"));
    QScrollArea scrollArea;
    scrollArea.setWidget(iconEditor);
}

```

```

scrollArea.viewport()->setBackgroundRole(QPalette::Dark);
scrollArea.viewport()->setAutoFillBackground(true);
scrollArea.setWindowTitle(QObject::tr("Icon Editor"));
scrollArea.show();
return app.exec();
}

```

QScrollArea (схематично показан на рис. 6.12) при отображении виджета использует его текущий или идеальный размер, если размеры виджета еще ни разу не изменялись. Делая вызов `setWidgetResizable(true)`, мы указываем QScrollArea на необходимость автоматического изменения размеров виджета, чтобы можно было воспользоваться любым дополнительным пространством за пределами его идеальных размеров.



Рис. 6.12. Изменение размеров области с прокруткой QScrollArea

По умолчанию полосы прокрутки видны на экране только в том случае, когда область отображения меньше дочернего виджета. Мы можем сделать полосы прокрутки постоянно видимыми при помощи установки следующих политик полос прокрутки:

```

scrollArea.setHorizontalScrollBarPolicy(Qt::ScrollBarAlwaysOn);
scrollArea.setVerticalScrollBarPolicy(Qt::ScrollBarAlwaysOn);

```

QScrollArea большую часть своей функциональности наследует от QAbstractScrollArea. Такие классы, как QTextEdit и QAbstractItemView (базовый класс для классов отображения элементов в Qt), являются производными от QAbstractScrollArea, поэтому нам не надо для них формировать оболочку из QScrollArea для получения полос прокрутки.

Прикрепляемые окна и панели инструментов

Прикрепляемыми являются окна, которые могут крепиться к определенным областям главного окна приложения, QMainWindow, или быть независимыми «плавающими» окнами. QMainWindow имеет четыре области для таких окон: одна сверху, одна снизу, одна слева и одна справа от центрального виджета. В таких приложениях, как Microsoft Visual Studio и Qt Linguist, широко используются прикрепляемые окна для обеспечения очень гибкого интерфейса пользователя. В Qt прикрепляемые окна представляют собой экземпляры класса QDockWidget.

На рис. 6.13 показано приложение Qt с линейками инструментов и прикрепляемым окном.

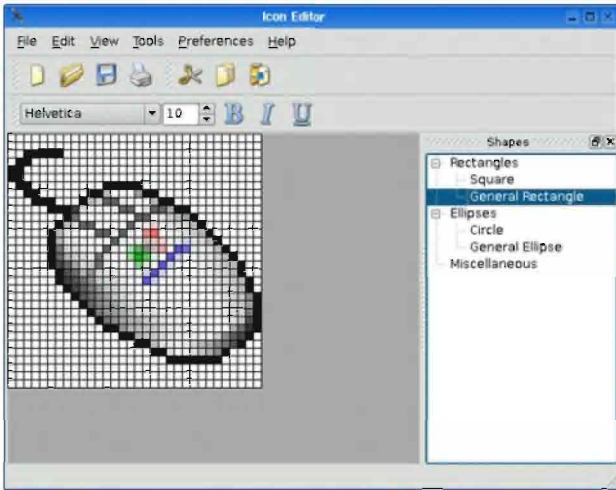


Рис. 6.13. QMainWindow с прикрепленным виджетом

Каждое прикрепляемое окно имеет свой собственный заголовок, даже когда он прикреплен. Пользователи могут перемещать прикрепляемые окна с одного места крепления на другое, передвигая полосу заголовка. Они могут также отсоединять прикрепляемое окно от области крепления и сделать его независимым плавающим окном, располагая прикрепляемое окно вне областей крепления. Свободные плавающие прикрепляемые окна всегда находятся «поверх» их главного окна. Пользователи могут закрыть QDockWidget, щелкая по кнопке закрытия, расположенной в заголовке окна. Любые комбинации этих возможностей можно отключать с помощью вызова `QDockWidget::setFeatures()`.

В ранних версиях Qt панели инструментов рассматривались как прикрепляемые окна, использующие те же самые области крепления. Начиная с Qt 4 панели инструментов размещаются в собственных областях, расположенных по периметру центрального виджета (как показано на рис. 6.14), и они не могут открепляться. Если требуется иметь плавающую панель инструментов, можно просто поместить ее внутрь QDockWidget.

Углы, обозначенные пунктирными линиями, могут принадлежать обоим соседним областям крепления. Например, мы могли бы верхний левый угол назначить левой области крепления с помощью вызова `QMainWindow::setCorner(Qt::TopLeftCorner, Qt::LeftDockWidgetArea)`.

Следующий фрагмент программного кода показывает, как для существующего виджета (в данном случае для `QTreeWidget`) можно оформить оболочку в виде `QDockWidget` и вставить ее в правую область крепления:

```
QDockWidget *shapesDockWidget = new QDockWidget(tr("Shapes"));
shapesDockWidget->setObjectName(shapesDockWidget);
shapesDockWidget->setWidget(treeWidget);
shapesDockWidget->setAllowedAreas(Qt::LeftDockWidgetArea
    | Qt::RightDockWidgetArea);
addDockWidget(Qt::RightDockWidgetArea, shapesDockWidget);
```

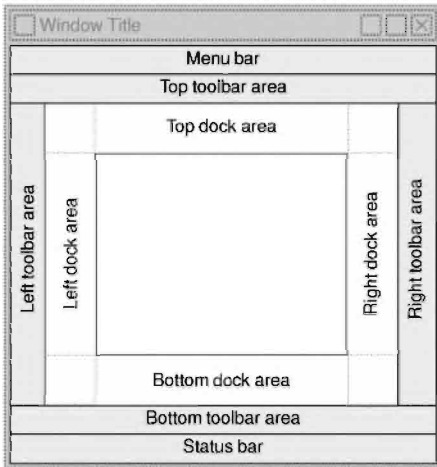


Рис. 6.14. Области крепления виджетов и области панелей инструментов QMainWindow

В вызове `setAllowedAreas()` задаются допустимые области крепления прикрепляемого окна. В нашем случае мы позволяем пользователю перетаскивать прикрепляемое окно только в левую или правую область крепления, где имеется достаточно пространства по вертикали для его нормального отображения. Если допустимые области не задаются явно, пользователь может перетаскивать прикрепляемое окно в любую из четырех областей.

Каждому `QObject` можно присвоить «имя объекта». Это имя может оказаться полезным при отладке, и оно используется некоторыми инструментами тестирования. Обычно нас не заботит присваивание виджетам имен объектов, но если мы создаем прикрепляемые окна и панели инструментов, это необходимо в том случае, если мы хотим использовать функции `QMainWindow::saveState()` и `QMainWindow::restoreState()` для сохранения и восстановления геометрии и состояния прикрепляемых окон и панелей инструментов.

Ниже приводится фрагмент из конструктора подкласса `QMainWindow`, который показывает, как можно создавать панель инструментов, содержащую `QComboBox`, `QSpinBox` и несколько кнопок `QToolButton`.

```
QToolBar *fontToolBar = new QToolBar(tr("Font"));
fontToolBar->setObjectName("fontToolBar");
fontToolBar->addWidget(familyComboBox);
fontToolBar->addWidget(sizeSpinBox);
fontToolBar->addAction(boldAction);
fontToolBar->addAction(italicAction);
fontToolBar->addAction(underlineAction);
fontToolBar->setAllowedAreas(Qt::TopToolBarArea
                           | Qt::BottomToolBarArea);

addToolBar(fontToolBar);
```

Если мы хотим сохранять позиции всех прикрепляемых виджетов и панелей инструментов, чтобы иметь возможность их восстановления при следующем за-

пуске приложения, мы можем написать почти такой же программный код, как для сохранения состояния разделителя `QSplitter`, используя функции класса `QMainWindow` `saveState()` и `restoreState()`:

```
void MainWindow::writeSettings()
{
    QSettings settings("Software Inc.", "Icon Editor");

    settings.beginGroup("mainWindow");
    settings.setValue("geometry", saveGeometry());
    settings.setValue("state", saveState());
    settings.endGroup();
}

void MainWindow::readSettings()
{
    QSettings settings("Software Inc.", "Icon Editor");

    settings.beginGroup("mainWindow");
    restoreGeometry(settings.value("geometry").toByteArray());
    restoreState(settings.value("state").toByteArray());
    settings.endGroup();
}
```

Наконец, `QMainWindow` обеспечивает контекстное меню, в котором представлены все прикрепляемые окна и панели инструментов. Это меню показано на рис. 6.15. Используя это меню, пользователь может закрывать и восстанавливать прикрепляемые окна и панели инструментов.



Рис. 6.15. Контекстное меню `QMainWindow`

Многодокументный интерфейс

Приложения, которые обеспечивают работу со многими документами в центральной области главного окна, называются приложениями с многодокументным интерфейсом или MDI-приложениями. В Qt MDI-приложения создаются с использованием в качестве центрального виджета класса `QMdiArea` и путем представления каждого документа в виде дочернего окна `QMdiArea`.

Обычно MDI-приложения содержат пункт главного меню `Windows` с командами по управлению как окнами, так и их списком. Активное окно отмечается галочкой. Пользователь может сделать любое окно активным, щелкая по его названию в меню `Windows`.

В данном разделе для демонстрации способов создания приложения с интерфейсом MDI и способов реализации его меню Windows мы разработаем MDI-приложение Editor, показанное на рис. 6.16. Все меню приложения показаны на рис. 6.17.

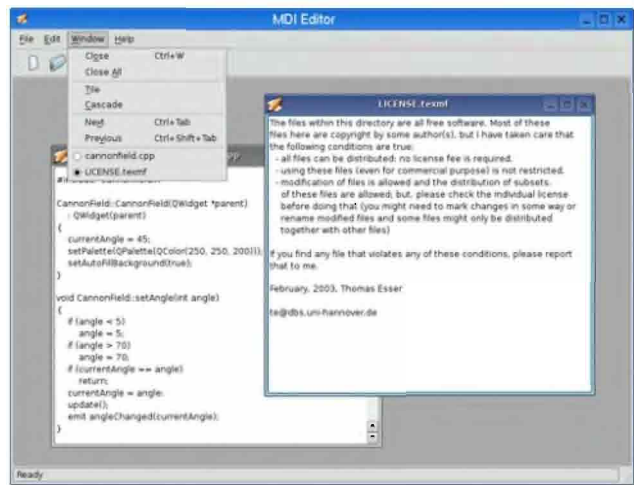


Рис. 6.16. MDI-приложение Editor

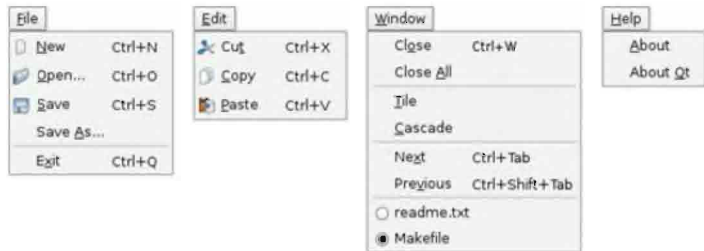


Рис. 6.17. Меню MDI-приложения Editor

Это приложение состоит из двух классов: `MainWindow` и `Editor`. Его программный код предлагается в составе прилагаемых к книге примеров, и поскольку большая часть его либо совпадает, либо очень похожа на программный код приложения *Электронная таблица* из части I, здесь мы представим только программный код, относящийся к MDI.

Давайте начнем с класса `MainWindow`.

```
MainWindow::MainWindow()
{
    mdiArea = new QMdiArea;
    setCentralWidget(mdiArea);
    connect(mdiArea, SIGNAL(subWindowActivated(QMdiSubWindow*)),
           this, SLOT(updateActions()));
```

```
createActions();
createMenus();
createToolBars();
createStatusBar();

setWindowIcon(QPixmap(":/images/icon.png"));
setWindowTitle(tr("MDI Editor"));
QTimer::singleShot(0, this, SLOT(loadFiles()));
}
```

В конструкторе `MainWindow` мы создаем виджет `QMdiArea` и делаем его центральным виджетом. Мы связываем сигнал `subWindowActivated()` класса `QMdiArea` со слотом, который мы будем использовать для обеспечения актуального состояния меню `Window` и с помощью которого будем обеспечивать включение и отключение действий в зависимости от состояния приложения. В конце конструктора мы задаем одноразовый таймер с интервалом 0 миллисекунд для вызова функции `loadFiles()`. Такие таймеры истекают, как только цикл события завершает работу. На практике это означает, что конструктор завершит свою работу, а затем, после того как будет выведено на экран главное окно, будет вызвана функция `loadFiles()`. Если этого не сделать, то в случае, когда нужно загрузить много больших файлов, конструктор не завершит работу до тех пор, пока все файлы не будут загружены и, следовательно, пользователь, ничего не видя на экране, может подумать, что приложение не запускается.

```
void MainWindow::loadFiles()
{
    QStringList args = QApplication::arguments();
    args.removeFirst();
    if (!args.isEmpty()) {
        foreach (QString arg, args)
            openFile(arg);
        mdiArea->cascadeSubWindows();
    } else {
        newFile();
    }
    mdiArea->activateNextSubWindow();
}
```

Если пользователь запустит приложение, указав в командной строке имена одного или нескольких файлов, то эта функция попытается загрузить все файлы, а в завершение отобразит каскад дочерних окон, чтобы пользователь смог увидеть файлы. Специфичные для Qt опции командной строки, такие как `-style` и `-font`, автоматически удаляются из списка аргументов конструктором `QApplication`. Так что если мы напишем в командной строке

```
mdieditor -style motif readme.txt
```

функция `QApplication::arguments()` вернет список `QStringList`, содержащий два элемента («`mdieditor`» и «`readme.txt`»), и приложение `MDIEditor` будет запущено с документом `readme.txt`.

Если в командной строке файл не указан, будет создано одно новое пустое окно редактора, чтобы пользователь мог сразу начать вводить текст. Вызов функции `activateNextSubWindow()` означает, что фокус переходит в окно редактора и обеспечивает вызов функции `updateActions()` для обновления меню Window, а также включение и отключение действий в зависимости от состояния приложения.

```
void MainWindow::newFile()
{
    Editor *editor = new Editor;
    editor->newFile();
    addEditor(editor);
}
```

Слот `newFile()` соответствует пункту меню `File|New`. Он создает виджет `Editor` и передает его в закрытую функцию `createEditor()`.

```
void MainWindow::open()
{
    Editor *editor = Editor::open(this);
    if (editor)
        addEditor(editor);
}
```

Функция `open()` соответствует пункту меню `File|Open`. Она вызывает статическую функцию `Editor::open()`, отображающую диалоговое окно открытия файла. Если пользователь выбирает файл, создается новый объект `Editor`, читается текст файла, и если чтение было выполнено успешно, то возвращается указатель на объект `Editor`. Если пользователь нажимает кнопку отмены в диалоговом окне или если при чтении возникает ошибка, то возвращается пустой указатель, а пользователь получает уведомление об ошибке. Более осмысленным будет реализовать операции с файлом в классе `Editor`, а не в классе `MainWindow`, поскольку каждому объекту `Editor` необходимо сохранять собственное независимое состояние.

```
void MainWindow::addEditor(Editor *editor)
{
    connect(editor, SIGNAL(copyAvailable(bool)),
            cutAction, SLOT(setEnabled(bool)));
    connect(editor, SIGNAL(copyAvailable(bool)),
            copyAction, SLOT(setEnabled(bool)));
    QMdiSubWindow *subWindow = mdiArea->addSubWindow(editor);

    windowMenu->addAction(editor->windowMenuAction());
    windowActionGroup->addAction(editor->windowMenuAction());

    subWindow->show();
}
```

Закрытая функция `addEditor()` вызывается функциями `newFile()` и `open()` для завершения инициализации нового виджета `Editor`. Она начинается с установления двух соединений «сигнал-слот». Эти соединения обеспечивают включение или выключение пунктов меню `Edit|Cut` и `Edit|Copy` в зависимости от наличия выделенной области текста.

Поскольку мы используем интерфейс MDI, может оказаться, что работа будет вестись одновременно с несколькими виджетами Editor. На это надо обратить внимание, поскольку мы заинтересованы в ответе на сигнал `copyAvailable(bool)`, поступающий только от активного окна редактора Editor, но не от других окон. Но эти сигналы могут порождаться только активным окном, поэтому это практически не составляет проблем.

Функция `QMdiArea::addSubWindow()` создает новое окно `QMdiSubWindow`, помещает виджет, переданный ей в качестве параметра, в свое дочернее окно и возвращает это дочернее окно.

Далее мы добавляем `QAction` для представления окна в меню Window. Это действие обеспечивается классом Editor, который мы скоро рассмотрим. Мы также добавляем это действие в объект `QActionGroup`. `QActionGroup` гарантирует, что в любой момент времени оказывается отмеченной только одна строка меню Window. Наконец, мы вызываем функцию `show()` применительно к окну `QMdiSubWindow`, чтобы сделать его видимым.

```
void MainWindow::save()
{
    if (activeEditor()) {
        activeEditor()->save();
    }
}
```

Слот `save()` вызывает функцию `Editor::save()` для активного редактора, если таковой имеется. И снова, программный код по выполнению реальной работы находится в классе Editor.

```
Editor *MainWindow::activeEditor()
{
    QMdiSubWindow *subWindow = mdiArea->activeSubWindow();
    if (subWindow)
        return qobject_cast<Editor *>(subWindow->widget());
    return 0;
}
```

Закрывающая функция `activeEditor()` возвращает виджет, хранящийся в активном дочернем окне, в виде указателя типа Editor или нулевой указатель при отсутствии такого окна.

```
void MainWindow::cut()
{
    if (activeEditor())
        activeEditor()->cut();
}
```

Слот `cut()` вызывает функцию `Editor::cut()` для активного редактора. Мы не приводим слоты `copy()`, `paste()` и `del()`, потому что они имеют такой же вид.

```
void MainWindow::updateActions()
{
    bool hasEditor = (activeEditor() != 0);
    bool hasSelection = activeEditor()
        && activeEditor()->textCursor().hasSelection();
}
```

```

saveAction->setEnabled(hasEditor);
saveAsAction->setEnabled(hasEditor);
cutAction->setEnabled(hasSelection);
copyAction->setEnabled(hasSelection);
closeAction->setEnabled(hasEditor);
pasteAction->setEnabled(hasEditor);
closeAllAction->setEnabled(hasEditor);
tileAction->setEnabled(hasEditor);
cascadeAction->setEnabled(hasEditor);
nextAction->setEnabled(hasEditor);
previousAction->setEnabled(hasEditor);
separatorAction->setVisible(hasEditor);

if (activeEditor())
    activeEditor()->windowMenuAction()->setChecked(true);
}

```

Сигнал `subWindowActivated()` генерируется каждый раз, когда дочернее окно становится активным и когда закрывается последнее дочернее окно (в последнем случае данный параметр будет представлять собой пустой указатель). Данный сигнал соединяется со слотом `updateActions()`.

Большинство пунктов меню имеет смысл только при существовании активного окна, поэтому мы их отключаем при отсутствии активного окна. В конце мы вызываем `setChecked()` для `QAction`, представляющего активное окно. Благодаря использованию `QActionGroup` нам не требуется явно сбрасывать флажок предыдущего активного окна.

```

void MainWindow::createMenus()
{
    ...
    windowMenu = menuBar()->addMenu(tr("&Window"));
    windowMenu->addAction(closeAction);
    windowMenu->addAction(closeAllAction);
    windowMenu->addSeparator();
    windowMenu->addAction(tileAction);
    windowMenu->addAction(cascadeAction);
    windowMenu->addSeparator();
    windowMenu->addAction(nextAction);
    windowMenu->addAction(previousAction);
    windowMenu->addAction(separatorAction);
    ...
}

```

Закрывающая функция `createMenus()` заполняет меню `Window` командами. Здесь используются типичные для такого рода меню команды, и они легко реализуются с применением слотов `QMdiArea's` `closeActiveSubWindow()`, `closeAllSubWindows()`, `tileSubWindows()` и `cascadeSubWindows()`. Всякий раз, когда пользователь открывает новое окно, в меню `Window` добавляется список действий. (Это делается в функции `addEditor()`.) При закрытии пользователем окна редактора соот-

ветствующий ему пункт в меню Window удаляется (поскольку его владельцем является это окно редактора), то есть пункт меню удаляется из меню Window автоматически.

```
void MainWindow::closeEvent(QCloseEvent *event)
{
    mdiArea->closeAllSubWindows();
    if (!mdiArea->subWindowList().isEmpty()) {
        event->ignore();
    } else {
        event->accept();
    }
}
```

Функция `closeEvent()` переопределяется для закрытия всех дочерних окон, обеспечивая получение всеми дочерними виджетами сигнала о возникновении события закрытия. Если один из дочерних виджетов «игнорирует» свое событие закрытия (прежде всего из-за того, что пользователь нажал кнопку отмены при выдаче соответствующего сообщения о «несохраненных изменениях»), мы игнорируем событие закрытия для `MainWindow`; в противном случае мы принимаем его, и в результате Qt закрывает все приложение. Если бы мы не переопределили функцию `closeEvent()` в `MainWindow`, у пользователя не было бы никакой возможности сохранения ни одного из несохраненных изменений.

Теперь мы закончили наш обзор `MainWindow`, поэтому можем перейти к реализации класса `Editor`. Класс `Editor` представляет одно дочернее окно. Он происходит от `QTextEdit`, который обеспечивает функциональность текстового редактора. Если в реальном мире необходим компонент для редактирования кода, мы также можем подумать об использовании `Scintilla`, который в варианте для Qt называется `QScintilla` и доступен на сайте <http://www.riverbankcomputing.couk/qscintilla/>. Так же как любой виджет может использоваться в качестве автономного окна, любой виджет может быть помещен в `QMdiSubWindow` и использоваться и в качестве дочернего окна в области интерфейса MDI.

Ниже приводится определение класса.

```
class Editor : public QTextEdit
{
    Q_OBJECT

public:
    Editor(QWidget *parent = 0);

    void newFile();
    bool save();
    bool saveAs();
    QSize sizeHint() const;
    QAction *windowMenuAction() const { return action; }
    static Editor *open(QWidget *parent = 0);
    static Editor *openFile(const QString &fileName,
        QWidget *parent = 0);
```

```
protected:
    void closeEvent(QCloseEvent *event);

private slots:
    void documentWasModified();

private:
    bool okToContinue();
    bool saveFile(const QString &fileName);
    void setCurrentFile(const QString &fileName);
    bool readFile(const QString &fileName);
    bool writeFile(const QString &fileName);
    QString strippedName(const QString &fullFileName);

    QString curFile;
    bool isUntitled;
    QString fileFilters;
    QAction *action;
};
```

Присутствующие в классе MainWindow приложения Электронная таблица четыре закрытые функции имеются также в классе Editor: okToContinue(), saveFile(), setCurrentFile() и strippedName().

```
Editor::Editor(QWidget *parent)
    : QTextEdit(parent)
{
    action = new QAction(this);
    action->setCheckable(true);
    connect(action, SIGNAL(triggered()), this, SLOT(show()));
    connect(action, SIGNAL(triggered()), this, SLOT(setFocus()));

    isUntitled = true;

    connect(document(), SIGNAL(contentsChanged()),
            this, SLOT(documentWasModified()));

    setWindowIcon(QPixmap(":/images/document.png"));
    setWindowTitle("[*]");
    setAttribute(Qt::WA_DeleteOnClose);
}
```

Сначала мы создаем действие QAction, представляющее редактор в меню приложения Window, и связываем его со слотами show() и setFocus().

Поскольку мы разрешаем пользователям создавать любое количество окон редактора, мы должны предусмотреть соответствующую систему их наименования, чтобы они отличались до первого их сохранения. Один из распространенных методов решения этой проблемы заключается в назначении имен с числами (например, document1.txt). Мы используем переменную isUntitled, чтобы отличить предоставляемые пользователем имена документов и сгенерированные программно.

Мы связываем сигнал текстового документа `contentsChanged()` с закрытым слотом `documentWasModified()`. Этот слот просто вызывает `setWindowModified(true)`.

Наконец, мы устанавливаем атрибут `Qt::WA_DeleteOnClose` для предотвращения утечек памяти при закрытии пользователем окна `Editor`.

```
void Editor::newFile()
{
    static int documentNumber = 1;

    curFile = tr("document%1.txt").arg(documentNumber);

    setWindowTitle(curFile + "[*]");
    action->setText(curFile);
    isUntitled = true;
    ++documentNumber;
}
```

Функция `newFile()` генерирует для нового документа имя типа `document1.txt`. Этот программный код помещен в функцию `newFile()`, а не в конструктор, поскольку мы не хотим использовать числа при вызове функции `open()` для открытия существующего документа во вновь созданном редакторе `Editor`. Поскольку переменная `documentNumber` объявлена как статическая, она совместно используется всеми экземплярами `Editor`.

Маркер «[*]» в заголовке окна указывает место, где мы хотим выдавать звездочку при несохраненных изменениях файла для платформ, отличных от Mac OS X. В Mac OS X у несохраненных документов на кнопке закрытия окна стоит точка. Мы рассматривали этот маркер в главе 3.

Помимо создания новых файлов пользователям часто бывает нужно открывать существующие файлы, выбираемые либо в диалоговом окне открытия файла, либо в каком-нибудь списке, например списке недавно открытых файлов. Для решения этих задач предлагаются две статические функции: `open()` — для выбора имени файла в файловой системе и `openFile()` — для создания окна `editor` и чтения в него содержимого указанного файла.

```
Editor *Editor::open(QWidget *parent)
{
    QString fileName =
        QFileDialog::getOpenFileName(parent, tr("Open"), ".");
    if (fileName.isEmpty())
        return 0;

    return openFile(fileName, parent);
}
```

Статическая функция `open()` выводит диалоговое окно, с помощью которого пользователь может выбрать файл. Если файл выбран, вызывается функция `openFile()`, создающая объект `Editor` и читающая содержимое файла.

```
Editor *Editor::openFile(const QString &fileName, QWidget *parent)
{

```

```

Editor *editor = new Editor(parent);
if (editor->readFile(fileName)) {
    editor->setCurrentFile(fileName);
    return editor;
} else {
    delete editor;
    return 0;
}
}

```

Эта статическая функция начинается с создания нового виджета `Editor`, после чего делается попытка прочитать указанный файл. Если чтение прошло успешно, возвращается `Editor`, а иначе пользователь получает сообщение о проблеме (в функции `readFile()`), редактор удаляется и возвращается пустой указатель.

```

bool Editor::save()
{
    if (isUntitled) {
        return saveAs();
    } else {
        return saveFile(curFile);
    }
}

```

Функция `save()` использует переменную `isUntitled` для определения вида вызываемой функции: `saveFile()` или `saveAs()`.

```

void Editor::closeEvent(QCloseEvent *event)
{
    if (okToContinue()) {
        event->accept();
    } else {
        event->ignore();
    }
}

```

Функция `closeEvent()` переопределяется, чтобы разрешить пользователю сохранить несохраненные изменения. Вся логика содержится в функции `okToContinue()`, которая выводит сообщение «Do you want to save your changes?». Если функция `okToContinue()` возвращает `true`, мы обрабатываем событие закрытия; в противном случае «игнорируем» его и окно оставляем прежним.

```

void Editor::setCurrentFile(const QString &fileName)
{
    curFile = fileName;
    isUntitled = false;
    action->setText(strippedName(curFile));
    document()->setModified(false);
    setWindowTitle(strippedName(curFile) + "[*]");
    setWindowModified(false);
}

```

Функция `setCurrentFile()` вызывается из `openFile()` и `saveFile()` для обновления переменных `curFile` и `isUntitled`, установки текста заголовка окна и пункта меню, а также для установки значения флажка модификации документа на `false`. Всякий раз, когда пользователь изменяет текст в редакторе, объект базового класса `QTextDocument` генерирует сигнал `contentsChanged()` и устанавливает свой внутренний флажок модификации на значение `true`.

```
QSize Editor::sizeHint() const
{
    return QSize(72 * fontMetrics().width('x'),
                 25 * fontMetrics().lineSpacing());
}
```

Наконец, функция `sizeHint()` возвращает размер, рассчитанный на основе ширины буквы «х» и высоты строки текста. `QMdiArea` использует идеальный размер в качестве начального размера окна.

Интерфейс MDI представляет собой один из способов работы одновременно со многими документами. В системе Mac OS X более предпочтителен подход, связанный с применением нескольких окон верхнего уровня. Мы рассматриваем этот подход в разделе «Работа со многими документами» главы 3.



- *Переопределение обработчиков событий*
- *Установка фильтров событий*
- *Обработка событий во время продолжительных процессов*

Глава 7. Обработка событий

События генерируются оконной системой или Qt в ответ на различные действия. Когда пользователь нажимает или отпускает клавишу или кнопку мыши, генерируется событие клавиши клавиатуры или кнопки мыши; когда окно впервые выводится на экран, генерируется событие рисования, указывая появившемуся окну на необходимость его прорисовки. Большинство событий генерируется в ответ на действия пользователя, но некоторые события, например события таймера, генерируются самой системой и не зависят от действий пользователя.

При программировании в Qt нам редко приходится думать о событиях, поскольку виджеты Qt сами генерируют сигналы в ответ на любое существенное событие. События становятся полезными при создании нами своих собственных виджетов, или когда мы хотим модифицировать поведение существующих виджетов Qt.

События не следует путать с сигналами. Как правило, сигналы полезны при использовании виджета, в то время как события полезны при реализации виджета. Например, при применении кнопки `QPushButton` мы больше заинтересованы в ее сигнале `clicked()`, чем в обработке низкоуровневых событий мыши или клавиатуры, сгенерировавших этот сигнал. Но если мы реализуем такой класс, как `QPushButton`, нам необходимо написать программный код для обработки событий мыши и клавиатуры и при необходимости сгенерировать сигнал `clicked()`.

Переопределение обработчиков событий

В Qt событие (event) – это экземпляр подкласса `QEvent`. Qt обрабатывает более сотни типов событий, каждое из которых идентифицируется определенным значением перечисления. Например, `QEvent::type()` возвращает `QEvent::MouseButtonPress` для событий нажатия кнопки мыши.

Для событий многих типов недостаточно тех данных, которые могут храниться в простом объекте `QEvent`; например, для событий нажатия кнопки мыши необходимо иметь информацию о том, какая кнопка мыши привела к возникновению данного события, а также о том, где находился курсор мыши в момент возникновения события. Эта дополнительная информация хранится в определенных подклассах `QEvent`, например в `QMouseEvent`.

События уведомляют объекты о себе при помощи своих функций `event()`, унаследованных от класса `QObject`. Реализация `event()` в `QWidget` передает большинство обычных событий конкретным обработчикам событий, например: `mousePressEvent()`, `keyPressEvent()` и `paintEvent()`.

Мы уже познакомились в предыдущих главах со многими обработчиками событий при реализации `MainWindow`, `IconEditor` и `Plotter`. Многие другие типы событий приводятся в справочной документации по `QEvent`, и можно также самому создавать и генерировать события. В данной главе мы рассмотрим два распространенных типа событий, заслуживающих более детального обсуждения, а именно: события клавиатуры и события таймера.

События клавиатуры обрабатываются путем переопределения функций `keyPressEvent()` и `keyReleaseEvent()`. Виджет `Plotter` переопределяет `keyPressEvent()`. Обычно нам требуется переопределить только `keyPressEvent()`, поскольку отпускание клавиш важно только для клавиш-модификаторов, то есть для клавиш `Ctrl`, `Shift` и `Alt`, а их можно проконтролировать в `keyPressEvent()` при помощи функции `QKeyEvent::modifiers()`. Например, если бы нам пришлось реализовывать виджет `CodeEditor` (редактор программного кода), общий вид его функции `keyPressEvent()` с различной обработкой клавиш `Home` и `Ctrl+Home` был бы следующим:

```
void CodeEditor::keyPressEvent(QKeyEvent *event)
{
    switch (event->key()) {
    case Qt::Key_Home:
        if (event->modifiers() & Qt::ControlModifier) {
            goToBeginningOfDocument();
        } else {
            goToBeginningOfLine();
        }
        break;
    case Qt::Key_End:
        ...
    default:
        QWidget::keyPressEvent(event);
    }
}
```

Клавиши `Tab` и `Backtab` (`Shift+Tab`) представляют собой особый случай. Функция `QWidget::event()` вызывает их до вызова `keyPressEvent()` с установкой фокуса на следующий или предыдущий виджет в фокусной цепочке. Обычно нам нужен именно такой режим работы, но в виджете `CodeEditor` мы, возможно, предпочтем использовать клавишу табуляции `Tab` для обеспечения отступа в начале строки. Переопределение функции `event()` выглядело бы следующим образом:

```
bool CodeEditor::event(QEvent *event)
{
    if (event->type() == QEvent::KeyPress) {
        QKeyEvent *keyEvent = static_cast<QKeyEvent> *event;
        if (keyEvent->key() == Qt::Key_Tab) {
            insertAtCurrentPosition('\t');
        }
    }
}
```

```

        return true;
    }
}
return QWidget::event(event);
}

```

Если событие сгенерировано нажатием клавиши клавиатуры, мы преобразуем объект типа `QEvent` в `QKeyEvent` и проверяем, какая клавиша была нажата. Если это клавиша `Tab`, мы выполняем некоторую обработку и возвращаем `true`, чтобы уведомить `Qt` об обработке нами события. Если бы мы вернули `false`, `Qt` передало бы событие родительскому виджету.

Высокоуровневый метод обработки клавиш клавиатуры заключается в применении класса `QAction`. Например, если `goToBeginningOfLine()` и `goToBeginningOfDocument()` являются открытыми слотами виджета `CodeEditor`, и `CodeEditor` применяется в качестве центрального виджета класса `MainWindow`, мы могли бы обеспечить обработку клавиш при помощи следующего программного кода:

```

MainWindow::MainWindow()
{
    editor = new CodeEditor;
    setCentralWidget(editor);

    goToBeginningOfLineAction =
        new QAction(tr("Go to Beginning of Line"), this);
    goToBeginningOfLineAction->setShortcut(tr("Home"));
    connect(goToBeginningOfLineAction, SIGNAL(activated()),
            editor, SLOT(goToBeginningOfLine()));

    goToBeginningOfDocumentAction =
        new QAction(tr("Go to Beginning of Document"), this);
    goToBeginningOfDocumentAction->setShortcut(tr("Ctrl+Home"));
    connect(goToBeginningOfDocumentAction, SIGNAL(activated()),
            editor, SLOT(goToBeginningOfDocument()));
    ...
}

```

Это позволяет легко добавлять команды в меню или в панель инструментов, что мы видели в главе 3. Если команды не отображаются в интерфейсе пользователя, объект `QAction` можно заменить объектом `QShortcut`; этот класс использует внутри `QAction` для связывания клавиши клавиатуры со своим обработчиком.

По умолчанию связывание клавиши в виджете, выполненное с использованием `QAction` или `QShortcut`, будет постоянно действовать, пока активно окно, содержащее этот виджет. Это можно изменить с помощью вызова `QAction::setShortcutContext()` или `QShortcut::setContext()`.

Другим распространенным типом событий является событие таймера. Если большинство других событий возникает в результате действий пользователя, то события таймера позволяют приложениям выполнять какую-то обработку через определенные интервалы времени. События таймера могут использоваться для реализации мигающих курсоров и другой анимации, или просто для обновления экрана.

Для демонстрации событий таймера мы реализуем виджет `Ticker`, показанный на рис. 7.1. Этот виджет отображает текстовый баннер, который сдвигается на один пиксель влево через каждые 30 миллисекунд. Если виджет шире текста, последний повторяется необходимое число раз и заполняет виджет по всей его ширине.



Рис. 7.1. Виджет `Ticker`

Ниже приводится заголовочный файл:

```
#ifndef TICKER_H
#define TICKER_H

#include <QWidget>

class Ticker : public QWidget
{
    Q_OBJECT
    Q_PROPERTY(QString text READ text WRITE setText)

public:
    Ticker(QWidget *parent = 0);

    void setText(const QString &newText);
    QString text() const { return myText; }
    QSize sizeHint() const;

protected:
    void paintEvent(QPaintEvent *event);
    void timerEvent(QTimerEvent *event);
    void showEvent(QShowEvent *event);
    void hideEvent(QHideEvent *event);

private:
    QString myText;
    int offset;
    int myTimerId;
};
#endif
```

Мы переопределяем в `Ticker` четыре обработчика событий, с тремя из которых мы до сих пор не встречались: `timerEvent()`, `showEvent()` и `hideEvent()`.

Теперь давайте рассмотрим реализацию:

```
#include <QtGui>

#include "ticker.h"
```

```

Ticker::Ticker(QWidget *parent)
    : QWidget(parent)
{
    offset = 0;
    myTimerId = 0;
}

```

Конструктор инициализирует смещение offset значением 0. Координата x начала вывода текста рассчитывается на основе значения offset. Таймер всегда имеет ненулевой идентификатор, поэтому мы используем 0, показывая, что таймер еще не запущен.

```

void Ticker::setText(const QString &newText)
{
    myText = newText;
    update();
    updateGeometry();
}

```

Функция setText() устанавливает отображаемый текст. Она вызывает update() для выдачи запроса на перерисовку и updateGeometry() для уведомления всех менеджеров компоновки, содержащих виджет Ticker, об изменении идеального размера.

```

QSize Ticker::sizeHint() const
{
    return fontMetrics().size(0, text());
}

```

Функция sizeHint() возвращает в качестве идеального размера виджета размеры области, занимаемой текстом. Функция QWidget::fontMetrics() возвращает объект QFontMetrics, который можно использовать для получения информации относительно шрифта виджета. В данном случае мы определяем размер заданного текста. (В первом аргументе функции QFontMetrics::size() задается флажок, который не нужен для простых строк, поэтому мы просто передаем 0).

```

void Ticker::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);

    int textWidth = fontMetrics().width(text());
    if (textWidth < 1)
        return;
    int x = -offset;
    while (x < width()) {
        painter.drawText(x, 0, textWidth, height(),
                        Qt::AlignLeft | Qt::AlignVCenter, text());
        x += textWidth;
    }
}

```

Функция paintEvent() отображает текст при помощи функции QPainter::drawText(). Она использует функцию fontMetrics() для определения размера области, занимаемой текстом по горизонтали, и затем выводит текст столько раз,

сколько необходимо для заполнения виджета по всей его ширине, учитывая значение смещения `offset`.

```
void Ticker::showEvent(QShowEvent * /* event */)
{
    myTimerId = startTimer(30);
}
```

Функция `showEvent()` запускает таймер. Вызов `QObject::startTimer()` возвращает число-идентификатор, которое мы можем использовать позже для идентификации таймера. `QObject` поддерживает несколько независимых таймеров, каждый из которых использует свой временной интервал. После вызова функции `startTimer()` Qt генерирует событие таймера приблизительно через каждые 30 миллисекунд, причем точность зависит от базовой операционной системы.

Мы могли бы функцию `startTimer()` вызвать в конструкторе `Ticker`, но мы экономим некоторые ресурсы за счет генерации Qt событий таймера только в тех случаях, когда виджет действительно видим.

```
void Ticker::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == myTimerId) {
        ++offset;
        if (offset >= fontMetrics().width(text()))
            offset = 0;
        scroll(-1, 0);
    } else {
        QWidget::timerEvent(event);
    }
}
```

Система вызывает функцию `timerEvent()` в соответствующие моменты времени. Она увеличивает смещение `offset` на 1 для имитации движения по всей области вывода текста. Затем она перемещает содержимое виджета на один пиксель влево при помощи функции `QWidget::scroll()`. Вполне достаточно было бы вызывать функцию `update()` вместо `scroll()`, но вызов функции `scroll()` более эффективен, потому что она просто перемещает существующие на экране пиксели и генерирует событие рисования для открывшейся области виджета (которая в данном случае представляет собой полосу шириной в один пиксель).

Если событие таймера не относится к нашему таймеру, мы передаем его дальше в базовый класс.

```
void Ticker::hideEvent(QHideEvent * /* event */)
{
    killTimer(myTimerId);
    myTimerId = 0;
}
```

Функция `hideEvent()` вызывает `QObject::killTimer()` для остановки таймера.

События таймера являются низкоуровневыми событиями, и, если нам необходимо иметь несколько таймеров, это может усложнить отслеживание всех идентификаторов таймеров. В таких ситуациях обычно легче создавать для

каждого таймера объект `QTimer`. `QTimer` генерирует через заданный временной интервал сигнал `timeout()`. `QTimer` также обеспечивает удобный интерфейс для однократных таймеров (то есть, таймеров, которые срабатывают только один раз), как мы видели в главе 6.

Установка фильтров событий

Одним из действительно эффективных средств в модели событий Qt является возможность с помощью некоторого экземпляра объекта `QObject` контролировать события другого экземпляра объекта `QObject` еще до того, как они дойдут до последнего.

Предположим, что наш виджет `CustomerInfoDialog` состоит из нескольких редакторов строк `QLineEdit`, и мы хотим использовать клавишу `Space` (пробел) для передачи фокуса следующему `QLineEdit`. Такой необычный режим работы может оказаться полезным для разработки, предназначенной для собственных нужд, и когда пользователи имеют навык работы в таком режиме. Простое решение заключается в создании подкласса `QLineEdit` и переопределении функции `keyPressEvent()` для вызова `focusNextChild()`, и оно выглядит следующим образом:

```
void MyLineEdit::keyPressEvent(QKeyEvent *event)
{
    if (event->key() == Qt::Key_Space) {
        focusNextChild();
    } else {
        QLineEdit::keyPressEvent(event);
    }
}
```

Этот подход имеет один основной недостаток: если мы используем в форме несколько различных видов виджетов (например, `QComboBox` и `QSpinBox`), мы должны также создать их подклассы для обеспечения единообразного поведения. Лучшее решение заключается в перехвате виджетом `CustomerInfoDialog` событий нажатия клавиш клавиатуры своих дочерних виджетов и в обеспечении необходимого поведения в его программном коде. Это можно сделать при помощи фильтров событий. Настройка фильтров событий состоит из двух этапов.

1. Зарегистрируйте объект-перехватчик с целевым объектом посредством вызова функции `installEventFilter()` для целевого объекта.
2. Выполните обработку событий целевого объекта в функции `eventFilter()` перехватчика.

Регистрацию объекта контроля удобно выполнять в конструкторе:

```
CustomerInfoDialog::CustomerInfoDialog(QWidget *parent)
    : QDialog(parent)
{
    ...
    firstNameEdit->installEventFilter(this);
    lastNameEdit->installEventFilter(this);
    cityEdit->installEventFilter(this);
    phoneNumberEdit->installEventFilter(this);
}
```

После регистрации фильтров событий, те из них, которые посылаются виджетам `firstNameEdit`, `lastNameEdit`, `cityEdit` и `phoneNumberEdit`, сначала будут переданы функции `eventFilter()` виджета `CustomerInfoDialog` и лишь затем дойдут по своему прямому назначению. (Если для одного объекта установлено несколько фильтров событий, они вызываются по очереди, начиная с установленного последним и последовательно возвращаясь к первому.)

Ниже приводится функция `eventFilter()`, которая перехватывает события:

```
bool CustomerInfoDialog::eventFilter(QObject *target, QEvent *event)
{
    if (target == firstNameEdit || target == lastNameEdit
        || target == cityEdit || target == phoneNumberEdit) {
        if (event->type() == QEvent::KeyPress) {
            QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
            if (keyEvent->key() == Qt::Key_Space) {
                focusNextChild();
                return true;
            }
        }
    }

    return QDialog::eventFilter(target, event);
}
```

Во-первых, мы проверяем, является ли целевой виджет строкой редактирования `QLineEdit`. Если событие вызвано нажатием клавиши клавиатуры, мы преобразуем его тип в `QKeyEvent` и проверяем, какая клавиша нажата. Если нажата клавиша пробела `Space`, мы вызываем функцию `focusNextChild()` для перехода фокуса на следующий виджет в фокусной цепочке, и мы возвращаем `true` для уведомления `Qt` о завершении нами обработки события. Если бы мы вернули `false`, `Qt` отослал бы событие по его прямому назначению, что привело бы к вставке лишнего пробела в строку редактирования `QLineEdit`.

Если целевым виджетом не является `QLineEdit`, или если событие не вызвано нажатием клавиши `Space`, мы передаем управление функции базового класса `eventFilter()`. Целевым виджетом мог бы быть также некоторый виджет, базовый класс которого, `QDialog`, осуществляет контроль. (В `Qt 4.3` этого не происходит с `QDialog`. Однако другие классы виджетов в `Qt`, например `QScrollArea`, контролируют по различным причинам некоторые свои дочерние виджеты.

`Qt` предусматривает пять уровней обработки и фильтрации событий.

1. Мы можем переопределять конкретный обработчик событий.

Переопределение таких обработчиков событий, как `mousePressEvent()`, `keyPressEvent()`, и `paintEvent()`, представляет собой очень распространенный способ обработки событий. Мы уже видели много примеров такой обработки.

2. Мы можем переопределять функцию `QObject::event()`.

Путем переопределения функции `event()` мы можем обрабатывать события до того, как они дойдут до обработчиков соответствующих событий. Этот подход очень хорош для изменения принятого по умолчанию поведения клавиши табуляции `Tab`, что было показано ранее. Он также используется для обработки редких событий, для которых не предусмотрены отдельные

обработчики событий (например, `QEvent::HoverEnter`). При переопределении функции `event()` нам необходимо вызывать функцию базового класса `event()` для обработки тех событий, которые мы сами не обрабатываем.

3. Мы можем устанавливать фильтр событий для отдельного объекта `QObject`. После регистрации объекта с помощью функции `installEventFilter()` все события целевого объекта сначала передаются функции контролирующего объекта `eventFilter()`. Если для одного объекта установлено несколько фильтров, они действуют поочередно, начиная с того, который установлен последним, и кончая тем, который установлен первым.
4. Мы можем устанавливать фильтр событий для объекта `QApplication`. После регистрации фильтра для `qApp` (уникальный объект типа `QApplication`) каждое событие каждого объекта приложения передается функции `eventFilter()` до его передачи любым другим фильтрам событий. Этот подход очень удобен для отладки. Он может также использоваться для обработки событий мыши, посылаемых для отключения виджетов, которые обычно отклоняются `QApplication`.
5. Мы можем создать подкласс `QApplication` и переопределить функцию `notify()`.
Qt вызывает `QApplication::notify()` для генерации события. Переопределение этой функции представляет собой единственный способ получения доступа ко всем событиям до того, как ими займутся фильтры событий. Пользоваться фильтрами событий, как правило, удобнее, поскольку параллельно может существовать любое количество фильтров событий и только одна функция `notify()`.

События многих типов, в том числе события мыши и клавиатуры, могут передаваться дальше по системе объектов приложения. Если событие не было обработано ни на пути к целевому объекту, ни самим целевым объектом, процесс обработки события повторяется, но теперь в качестве нового целевого объекта используется родительский объект. Этот процесс продолжается, управление передается от одного родительского объекта к другому до тех пор, пока либо событие не будет обработано, либо не будет достигнут объект самого верхнего уровня.

На рис. 7.2 показано, как событие нажатия клавиши пересылается в диалоговом окне от дочернего объекта к родительскому. Когда пользователь нажимает клавишу на клавиатуре, сначала событие передается виджету, на котором установлен фокус – в данном случае это расположенный в нижнем правом углу флажок `QCheckBox`. Если `QCheckBox` не обрабатывает это событие, Qt передает его объекту `QGroupBox` и, в конце концов, объекту `QDialog`.

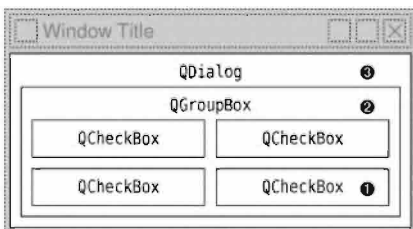


Рис. 7.2. Передача события в диалоговом окне

Обработка событий во время продолжительных процессов

Когда мы вызываем `QApplication::exec()`, тем самым начинаем цикл обработки событий Qt. При запуске приложения Qt генерирует несколько событий для отображения на экране виджетов. После этого начинается выполнение цикла обработки событий: постоянно проверяется их возникновение, и эти события отправляются к объектам `QObject` данного приложения.

Во время обработки события могут генерироваться другие события, которые ставятся в конец очереди событий Qt. Если слишком много времени уходит на обработку одного события, интерфейс пользователя становится невосприимчив к действиям пользователя. Например, любые сгенерированные оконной системой события во время сохранения файла на диск не будут обрабатываться до тех пор, пока весь файл не будет записан. В ходе записи файла приложение не будет отвечать на запросы оконной системы на перерисовку приложения.

Одно из решений заключается в применении многопоточной обработки: один процесс – для работы с интерфейсом пользователя приложения, и другой процесс – для выполнения операции сохранения файла (или любой другой длительной операции). В этом случае интерфейс пользователя приложения сможет реагировать на события в процессе выполнения операции сохранения файла. Мы рассмотрим способы обеспечения такого режима работы в главе 14.

Более простое решение заключается в выполнении частых вызовов функции `QApplication::processEvents()` в программном коде сохранения файла. Данная функция говорит Qt о необходимости обработки ожидающих в очереди событий и затем возвращает управление вызвавшей ее функции. Фактически функция `QApplication::exec()` представляет собой не более чем вызов функции `processEvents()` в цикле `while`.

Ниже приводится пример того, как мы можем сохранить работоспособность интерфейса пользователя при помощи функции `processEvents()`, причем за основу взят программный код сохранения файла в приложении Spreadsheet:

```
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    ...
    QApplication::setOverrideCursor(Qt::WaitCursor);
    for (int row = 0; row < RowCount; ++row) {
        for (int column = 0; column < ColumnCount; ++column) {
            QString str = formula(row, column);
            if (!str.isEmpty())
                out << quint16(row) << quint16(column) << str;
        }
        qApp->processEvents();
    }
    QApplication::restoreOverrideCursor();
    return true;
}
```

При использовании этого метода существует опасность того, что пользователь может закрыть главное окно во время выполнения операции сохранения файла или даже выбрать повторно File|Save, что приведет к непредсказуемому результату. Наиболее простое решение заключается в замене вызова

```
qApp->processEvents();
```

на вызов,

```
qApp->processEvents(QEventLoop::ExcludeUserInputEvents);
```

который указывает Qt на необходимость игнорирования события мыши и клавиатуры.

Часто нам хочется показывать индикатор состояния процесса `QProgressDialog` в ходе выполнения продолжительной операции. `QProgressDialog` имеет полосу индикатора, информирующую пользователя о ходе выполнения операции приложением. `QProgressDialog` также содержит кнопку `Cancel`, которая позволяет пользователю прекратить выполнение операции. Ниже приводится программный код, применяющий данный подход при сохранении файла приложения **Электронная таблица**:

```
bool Spreadsheet::writeFile(const QString &fileName)
{
    QFile file(fileName);
    ...
    QProgressDialog progress(this);
    progress.setLabelText(tr("Saving %1") arg(fileName));
    progress.setRange(0, RowCount);
    progress.setModal(true);

    for (int row = 0; row < RowCount; ++row) {
        progress.setValue(row);
        qApp->processEvents();
        if (progress.wasCanceled()) {
            file.remove();
            return false;
        }
        for (int column = 0; column < ColumnCount; ++column) {
            QString str = formula(row, column);
            if (!str.isEmpty())
                out << quint16(row) << quint16(column) << str;
        }
    }
    return true;
}
```

Мы создаем `QProgressDialog`, в котором `NumRows` является общим количеством шагов. Затем при обработке каждой строки мы вызываем функцию `setValue()` для обновления состояния индикатора. `QProgressDialog` автоматически вычисляет процент завершения операции путем деления текущего значения индикатора на общее количество шагов. Мы вызываем функцию `QApplication::processEvents()` для обработки любых событий перерисовки или нажатия пользователем кнопки

мышь или клавиши клавиатуры (например, чтобы разрешить пользователю нажимать кнопку Cancel). Если пользователь нажимает кнопку Cancel, мы прекращаем операцию сохранения файла и удаляем файл.

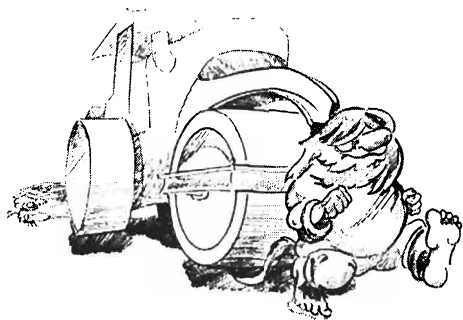
Мы не вызываем для `QProgressDialog` функцию `show()`, так как индикатор состояния сам делает это. Если оказывается так, что операция выполняется быстро, прежде всего, из-за малого размера файла или высокого быстродействия компьютера, `QProgressDialog` обнаружит это и вообще не станет выводить себя на экран.

Кроме многопоточности и применения `QProgressDialog`, существует совершенно другой способ работы с продолжительными операциями. Вместо выполнения заданной обработки сразу по поступлению запроса пользователя, мы можем отложить эту обработку до момента перехода приложения в состояние ожидания. Этим способом можно пользоваться в тех случаях, когда обработку можно легко прерывать и затем возобновлять, поскольку мы не можем предсказать, как долго приложение будет в состоянии ожидания.

В Qt этот подход можно реализовать путем применения 0-миллисекундного таймера. Таймеры этого типа работают при отсутствии ожидающих событий. Ниже приводится пример реализации функции `timerEvent()`, которая демонстрирует обработку в состоянии ожидания:

```
void Spreadsheet::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == myTimerId) {
        while (step < MaxStep && !qApp->hasPendingEvents()) {
            performStep(step);
            ++step;
        }
    } else {
        QTableWidget::timerEvent(event);
    }
}
```

Если функция `hasPendingEvents()` возвращает `true`, мы останавливаем процесс и передаем управление обратно Qt. Этот процесс будет возобновлен после обработки Qt всех своих ожидающих событий.



- *Рисование при помощи QPainter*
- *Преобразования системы координат*
- *Высококачественное воспроизведение изображения при помощи QImage*
- *Элементное воспроизведение при помощи графического представления*
- *Вывод на печатающее устройство*

Глава 8. Графика 2D

Основу используемых в Qt средств графики 2D составляет класс `QPainter` (рисовальщик Qt). Этот класс может использоваться для рисования геометрических фигур (точек, линий, прямоугольников, эллипсов, дуг, сегментов и секторов окружности, многоугольников и кривых Безье), а также пиксельных карт, изображений и текста. Кроме того, `QPainter` поддерживает такие продвинутые функции, как сглаживание соединений линий (antialiasing) при начертании фигур и букв в тексте, альфа-смешение (alpha blending), плавный переход цветов (gradient filling) и цепочки графических элементов (vector paths). `QPainter` также поддерживает линейные преобразования, такие как трансляция, поворот, обрезание и масштабирование.

`QPainter` может использоваться для вычерчивания на таком «устройстве рисования», как `QWidget`, `QPixmap`, `QImage` или `QSvgGenerator`. Класс `QPainter` можно также использовать совместно с `QPrinter` для вывода графики на печатающее устройство и для генерации PDF-документов. Это значит, что во многих случаях мы можем использовать тот же самый программный код при отображении данных на экран и при получении напечатанных отчетов.

Путем изменения реализации функции `QWidget::paintEvent()`, мы можем создавать пользовательские виджеты и осуществлять полный контроль над их внешним видом, как это показано в главе 5. Для настройки внешнего вида и режимов работы готовых виджетов Qt мы также можем указать таблицу стилей или создать подкласс `QStyle`. Оба эти подхода мы рассмотрим в главе 19.

Типичным требованием является необходимость отображать большое количество «легких» произвольных элементов определенной формы, с которыми пользователь может осуществлять взаимодействие на двумерной канве. В Qt 4.2 появилась совершенно новая архитектура «графических представлений», строящаяся вокруг классов `QGraphicsView`, `QGraphicsScene` и `QGraphicsItem`. Эта архитектура предоставляет высокоуровневый интерфейс для создания графики на основе элементов (item) и поддерживает стандартные пользовательские действия с элементами, такие как перемещение, выбор и группировка. Сами эти элементы рисуются `QPainter` как обычно, и их можно трансформировать по отдельности. Мы рассмотрим эту архитектуру ниже в этой главе.

Альтернативой использованию QPainter является применение команд OpenGL. OpenGL представляет собой стандартную библиотеку для рисования 3-мерной графики. В главе 20 мы рассмотрим использование модуля QtOpenGL, который позволяет очень легко интегрировать OpenGL в приложения Qt.

Рисование при помощи QPainter

Чтобы начать рисовать на устройстве рисования (обычно это виджет), мы просто создаем объект QPainter и передаем ему указатель на устройство. Например:

```
void MyWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    ...
}
```

Мы можем рисовать различные фигуры, используя функции QPainter вида draw...(). На рис. 8.1 приведены наиболее важные из них. Параметры настройки QPainter влияют на режим рисования.

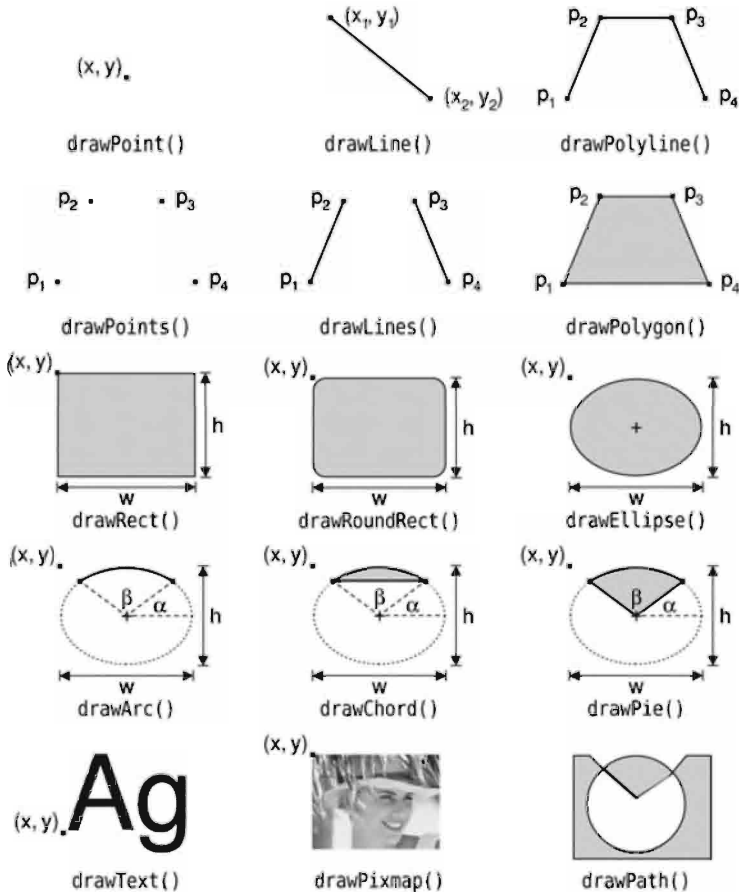


Рис. 8.1. Часто используемые функции draw...() рисовальщика QPainter

Некоторые из них устанавливаются на параметры настройки устройства, а другие инициализируются значениями по умолчанию. Тремя основными параметрами настройки рисовальщика являются перо, кисть и шрифт.

- Перо используется для отображения прямых линий и контуров фигур. Оно имеет цвет, толщину, стиль линии, стиль окончания линии и стиль соединения линий. Эти три основных параметра пера показаны на рис. 8.2. и 8.3.
- Кисть представляет собой шаблон, который используется для заполнения геометрических фигур. Он обычно имеет цвет и стиль, но может также представлять собой текстуру (пиксельную карту, повторяющуюся бесконечно) или цветовой градиент. Стили кисти представлены на рис. 8.4.
- Шрифт используется для отображения текста. Шрифт имеет много атрибутов, в том числе название и размер.

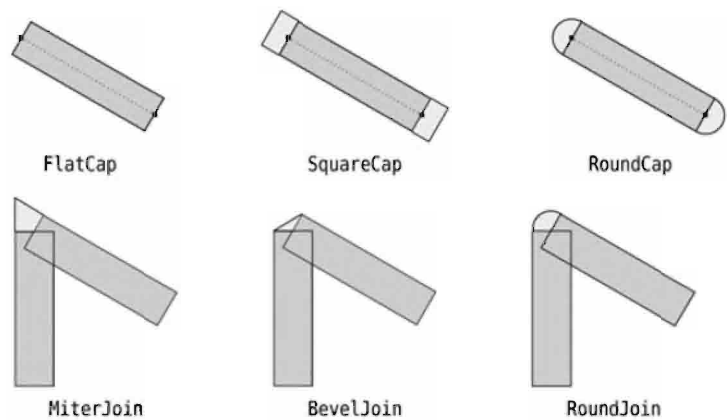


Рис. 8.2. Стили окончания линий и стили соединения линий

	Line width			
	1	2	3	4
SolidLine				
DashLine				
DotLine				
DashDotLine				
DashDotDotLine				
NoPen				

Рис. 8.3. Стили пера

Эти настройки можно в любое время модифицировать при помощи функций `setPen()`, `setBrush()` и `setFont()`, вызываемых для объектов `QPen`, `QBrush` или `QFont`. Давайте рассмотрим несколько примеров. Ниже приводится программный код для вычерчивания эллипса, показанного на рис. 8.5 (а):

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
```

```
painter.setPen(QPen(Qt::black, 12, Qt::DashDotLine, Qt::RoundCap));
painter.setBrush(QBrush(Qt::green, Qt::SolidPattern));
painter.drawEllipse(80, 80, 400, 240);
```

Вызов `setRenderHint()` включает режим сглаживания линий, указывая `QPainter` на необходимость использования по краям цветов различной интенсивности, чтобы уменьшить визуальное искажение, которое обычно заметно, когда края фигуры представляются пикселями. В результате края воспринимаются более ровными на тех платформах и устройствах, которые поддерживают эту функцию.

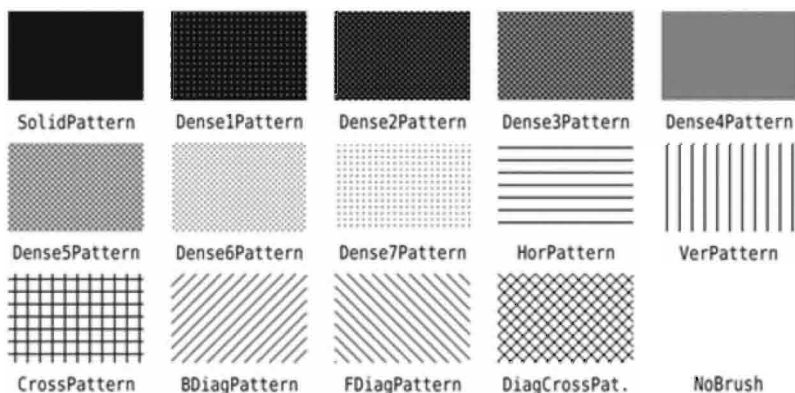


Рис. 8.4. Определенные в Qt стили кисти



Рис. 8.5. Примеры геометрических фигур

Ниже приводится программный код для вычерчивания сектора эллипса, показанного на рис. 8.5 (б):

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setPen(QPen(Qt::black, 15, Qt::SolidLine, Qt::RoundCap,
                    Qt::MiterJoin));
painter.setBrush(QBrush(Qt::blue, Qt::DiagCrossPattern));
painter.drawPie(80, 80, 400, 240, 60 * 16, 270 * 16);
```

Два последних аргумента функции `drawPie()` задаются в шестнадцатых долях градуса.

Ниже приводится программный код для вычерчивания кривой Безье третьего порядка, показанной на рис. 8.5 (в):

```
QPainter painter(this);
painter.setRenderHint(QPainter::Antialiasing, true);
```

```

QPainterPath path;
path.moveTo(80, 320);
path.cubicTo(200, 80, 320, 80, 480, 320);

painter.setPen(QPen(Qt::black, 8));
painter.drawPath(path);

```

Класс `QPainterPath` может определять произвольные фигуры векторной графики, соединяя друг с другом основные графические элементы: прямые линии, эллипсы, многоугольники, дуги, кривые Безье и другие цепочки графических элементов рисовальщика (`painter paths`). Такие цепочки являются законченными элементарными рисунками в том смысле, что любая фигура или любая комбинация фигур может быть представлена в виде некоторой цепочки графических элементов рисовальщика.

Цепочка графических элементов определяет контур, а область внутри контура можно заполнить какой-нибудь кистью. В примере, представленном на рис. 8.5 (в), мы не задавали кисть, поэтому нарисован только контур.

В трех представленных примерах используются встроенные шаблоны кисти (`Qt::SolidPattern`, `Qt::DiagCrossPattern` и `Qt::NoBrush`). В современных приложениях градиентные заполнители являются популярной альтернативой однородным заполнителям. Цветовые градиенты основаны на интерполяции цветов, обеспечивающей сглаженные переходы между двумя или более цветами. Они часто применяются для получения эффекта 3-мерности изображения, например, стили `Plastique` и `Cleanlooks` используют цветовые градиенты при воспроизведении кнопок `QPushButton`.

`Qt` поддерживает три типа цветовых градиентов: линейный, конический и радиальный. В примере таймера духовки, который приводится в следующем разделе, в одном виджете используется комбинация всех трех типов градиентов для того, чтобы изображение выглядело реалистически.

- *Линейные градиенты* определяются двумя контрольными точками и рядом «цветовых отметок» на линии, соединяющей эти точки. Например, линейный градиент на рис. 8.6 создан при помощи следующего программного кода:

```

QLinearGradient gradient(50, 100, 300, 350);
gradient.setColorAt(0.0, Qt::white);
gradient.setColorAt(0.2, Qt::green);
gradient.setColorAt(1.0, Qt::black);

```

Мы задали три цвета в трех разных позициях между двумя контрольными точками. Позиции представляются в виде чисел с плавающей точкой в диапазоне между 0 и 1, где 0 соответствует первой контрольной точке, а 1 – последней контрольной точке. Цвет между этими позициями интерполируется линейно.

- *Радиальные градиенты* определяются центральной точкой (x_c, y_c) , радиусом r и точкой фокуса (x_f, y_f) , которая дополняет цветовые метки. Центральная точка и радиус определяют окружность. Изменение цвета распространяется во все стороны из точки фокуса, которая может совпадать с центральной точкой или может быть любой другой точкой внутри окружности.
- *Конические градиенты* определяются центральной точкой (x_c, y_c) и углом α . Изменение цвета распространяется вокруг центральной точки подобно перемещению секундной стрелки часов.

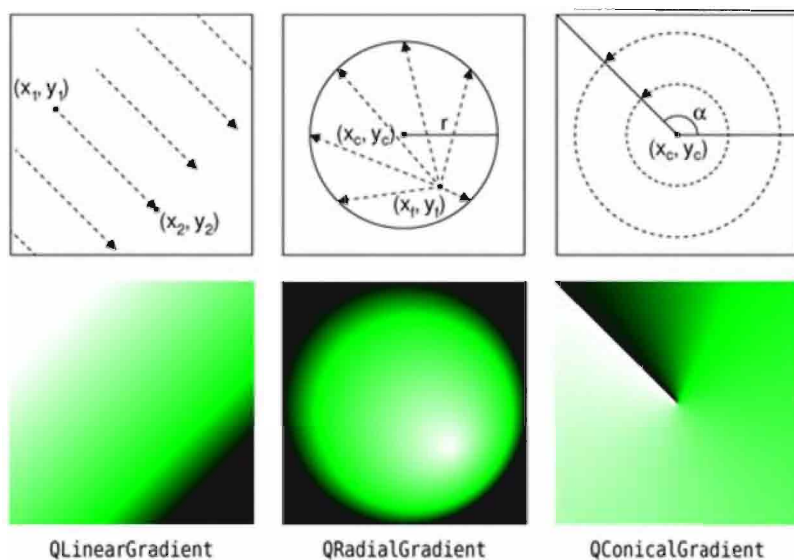


Рис. 8.6. Кисти QPainter с цветовыми градиентами

До сих пор мы говорили о настройках пера, кисти и шрифта рисовальщика. QPainter имеет другие параметры настройки, влияющие на способ рисования фигур и текста.

- *Кисть фона* используется для заполнения фона геометрических фигур (то есть, под шаблоном кисти), текста или пиксельной карты, когда в качестве режима отображения фона задан `Qt::OpaqueMode` (непрозрачный режим) (по умолчанию используется режим `Qt::TransparentMode` – прозрачный).
- *Исходная точка кисти* задает точку начала отображения шаблона кисти, в качестве которой обычно используется точка верхнего левого угла виджета.
- *Границы области рисования* определяют область рисования устройства. Операции рисования, которые выходят за пределы этой области, игнорируются.
- *Область отображения, окно и универсальная матрица* преобразования определяют способ перевода логических координат QPainter в физические координаты устройства рисования. По умолчанию системы логических и физических координат совпадают. Системы координат мы рассматриваем в следующем разделе.
- *Режим композиции* определяет способ взаимодействия новых выводимых пикселей с пикселями, уже присутствующими на устройстве рисования. По умолчанию используется режим «source over», при котором подвергаются альфа-смещению поверх существующих. Этот режим поддерживается только определенными устройствами, и он рассматривается позже в данной главе.

В любой момент времени мы можем сохранить в стеке текущее состояние рисовальщика, вызывая функцию `save()`, и восстановить его позже, вызывая фун-

кцию `restore()`. Это может быть полезно, если требуется временно изменить некоторые параметры настройки рисовальщика и затем их восстановить в прежние значения, как мы это увидим в следующем разделе.

Преобразования координатных систем

В используемой по умолчанию координатной системе рисовальщика `QPainter` точка $(0, 0)$ находится в левом верхнем углу устройства рисования; значение координаты x увеличивается при перемещении вправо, а значение координаты y увеличивается при перемещении вниз. Каждый пиксель занимает область 1×1 в координатной системе, применяемой по умолчанию.

По идее, центр пикселя имеет «полупиксельные» координаты. Например, пиксель в верхнем левом углу виджета занимает область между точками $(0, 0)$ и $(1, 1)$, а его центр находится в точке $(0.5, 0.5)$. Если мы просим `QPainter` нарисовать пиксель, например в точке $(100, 100)$, его координаты будут смещены на величину $+0.5$ по обоим направлениям, и в результате, нарисованный пиксель будет иметь центр в точке $(100.5, 100.5)$.

На первый взгляд эта особенность носит лишь теоретический интерес, однако она имеет важные практические последствия. Во-первых, смещение $+0.5$ действует только при отключении сглаживания линий (режим по умолчанию); если режим сглаживания линий включен, и мы пытаемся нарисовать пиксель черного цвета в точке $(100, 100)$, `QPainter` фактически выведет на экран четыре светло-серых пикселя в точках $(99.5, 99.5)$, $(99.5, 100.5)$, $(100.5, 99.5)$ и $(100.5, 100.5)$, чтобы создавалось впечатление расположения пикселя точно в точке соприкосновения всех этих четырех пикселей. Если этот эффект нежелателен, его можно избежать, указывая полупиксельные координаты или путем перемещения начала координат `QPainter` на величину $(+0.5, +0.5)$.

При начертании таких фигур, как линии, прямоугольники и эллипсы, действуют аналогичные правила. На рис. 8.7 показано, как изменяется результат вызова `drawRect(2, 2, 6, 5)` в зависимости от ширины пера, когда сглаживание линий отключено. В частности, важно отметить, что прямоугольник 6×5 , вычерчиваемый пером с шириной 1, фактически занимает область размером 7×6 . Это не делалось прежними инструментальными средствами, в том числе в ранних версиях Qt, но такой подход существенен для получения действительно масштабируемой, независимой от разрешающей способности векторной графики. На рис. 8.8 показан результат выполнения функции `drawRect(2, 2, 6, 5)` при включенном режиме сглаживания, а на рис. 8.9 показано, что произойдет при указании полупиксельных координат.

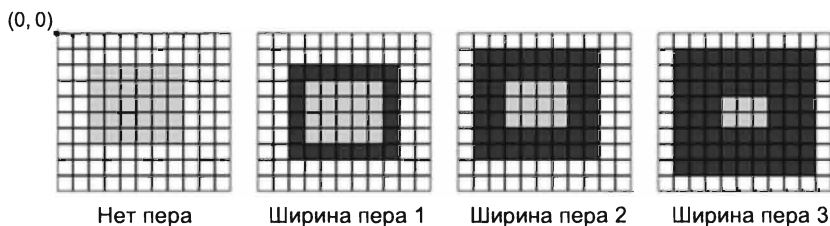


Рис. 8.7. Результат выполнения команды `drawRect(2, 2, 6, 5)` с отключенным режимом сглаживания линий

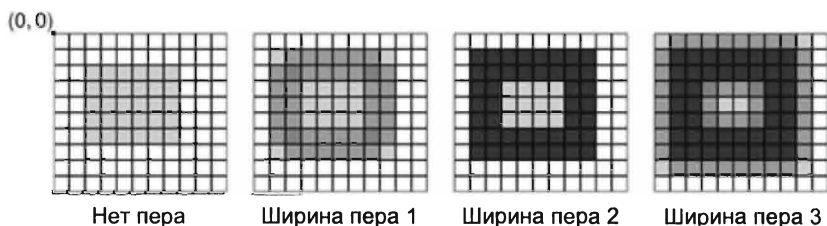


Рис. 8.8. Результат выполнения команды `drawRect(2, 2, 6, 5)` с включенным режимом сглаживания линий

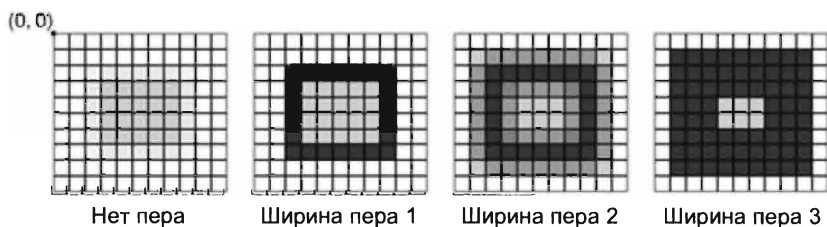


Рис. 8.9. Результат выполнения команды `drawRect(2.5, 2.5, 6, 5)` с включенным режимом сглаживания линий

Теперь, когда мы познакомились с используемой по умолчанию координатной системой, мы можем внимательно рассмотреть возможные ее изменения при использовании рисовальщиком `QPainter` области отображения, окна и универсальной матрицы преобразования. (В данном контексте термин «окно» не является обозначением окна виджета верхнего уровня, а термин «область отображения» никак не связан с областью отображения `QScrollArea`).

Термины «область отображения» и «окно» сильно связаны друг с другом. Область отображения является произвольным прямоугольником, заданным в физических координатах. Окно определяет такой же прямоугольник, но в логических координатах. При рисовании мы задаем координаты точек в логической системе координат, и эти координаты с помощью линейного алгебраического преобразования переводятся в физическую систему координат на основе текущих настроек связи «окно – область отображения».

По умолчанию область отображения и окно устанавливаются на прямоугольную область устройства рисования. Например, если этим устройством является виджет размером 320×200 , область отображения и окно представляют собой одинаковый прямоугольник 320×200 , верхний левый угол которого располагается в позиции $(0, 0)$. В данном случае системы логических и физических координат совпадают.

Механизм «окно – область отображения» удобно применять для создания программного кода, который не будет зависеть от размера или разрешающей способности устройства рисования. Например, если мы хотим обеспечить логические координаты в диапазоне от $(-50, -50)$ до $(+50, +50)$ с $(0, 0)$ в середине, мы можем задать окно следующим образом:

```
painter.setWindow(-50, -50, 100, 100);
```

Пара аргументов $(-50, -50)$ задает начальную точку, а пара аргументов $(100, 100)$ задает ширину и высоту. Это означает, что точка с логическими координатами

тами $(-50, -50)$ теперь соответствует точке с физическими координатами $(0, 0)$, а точка с логическими координатами $(+50, +50)$ соответствует точке с физическими координатами $(320, 200)$. Иллюстрация приводится на рис. 8.10. В этом примере мы не изменяли область отображения.

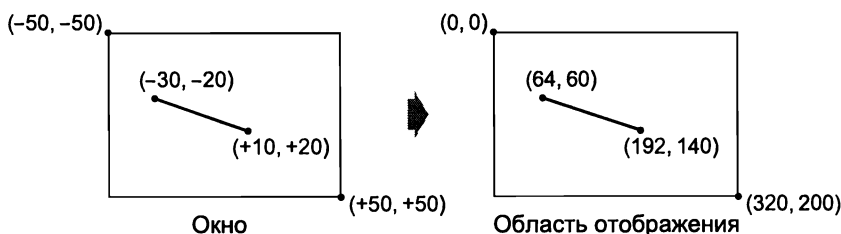


Рис. 8.10. Преобразование логических координат в физические

Теперь очередь дошла до универсальной матрицы преобразования. Эта матрица используется как дополнение к преобразованию «окно – область отображения». Она позволяет нам перемещать начало координат, изменять масштаб, поворачивать и обрезать графические элементы. Например, если бы нам понадобилось отобразить текст под углом в 45° , мы бы использовали такой программный код:

```
QTransform transform;
transform.rotate(+45.0);
painter.setWorldTransform(transform);
painter.drawText(pos, tr("Sales"));
```

Логические координаты, которые мы передаем функции `drawText()`, преобразуются при помощи универсальной матрицы и затем переводятся в физические координаты, используя связь «окно – область отображения».

Если мы задаем несколько преобразований, они осуществляются в порядке поступления. Например, если мы хотим использовать точку $(50, 50)$ в качестве точки поворота, мы можем перенести начало координат окна на $(+50, +50)$, выполнить поворот и затем сделать обратный перенос начала координат окна, устанавливая его в прежнее положение.

```
QTransform transform;
transform.translate(+50.0, +50.0);
transform.rotate(+45.0);
transform.translate(-50.0, -50.0);
painter.setWorldTransform(transform);
painter.drawText(pos, tr("Sales"));
```

Более удобно осуществлять преобразования путем применения соответствующих функций класса `QPainter`: `translate()`, `scale()`, `rotate()` и `shear()`:

```
painter.translate(-50.0, -50.0);
painter.rotate(+45.0);
painter.translate(+50.0, +50.0);
painter.drawText(pos, tr("Sales"));
```

Если мы хотим регулярно делать одни и те же преобразования, то более эффективным будет хранить их в объекте `QTransform`, и затем применять универ-

сальную матрицу преобразования для рисовальщика всякий раз, когда требуется выполнить преобразование.

Для иллюстрации преобразования рисовальщика мы рассмотрим программный код виджета `OvenTimer` (таймер духовки), показанного на рис. 8.11 и 8.12. Виджет `OvenTimer` имитирует кухонные таймеры, которые широко применялись до того, как в духовках стали применяться встроенные часы. Пользователь может повернуть ручку и установить требуемую длительность.



Рис. 8.11. Виджет `OvenTimer`

Диск переключателя автоматически поворачивается против часовой стрелки, пока не достигнет отметки 0, в результате чего `OvenTimer` генерирует сигнал `timeout()`.

```
class OvenTimer : public QWidget
{
    Q_OBJECT

public:
    OvenTimer(QWidget *parent = 0);

    void setDuration(int secs);
    int duration() const;
    void draw(QPainter *painter);

signals:
    void timeout();

protected:
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);

private:
    QDateTime finishTime;
    QTimer *updateTimer;
    QTimer *finishTimer;
};
```

Класс `OvenTimer` происходит от `QWidget` и переопределяет две виртуальные функции: `paintEvent()` и `mousePressEvent()`.

```
const double DegreesPerMinute = 7.0;
const double DegreesPerSecond = DegreesPerMinute / 60;
const int MaxMinutes = 45;
const int MaxSeconds = MaxMinutes * 60;
const int UpdateInterval = 5;
```

В файле `oventimer` мы начнем с нескольких констант, управляющих внешним видом и режимом работы таймера духовки.

```
OvenTimer::OvenTimer(QWidget *parent)
    : QWidget(parent)
{
    finishTime = QDateTime::currentDateTime();
    updateTimer = new QTimer(this);
    connect(updateTimer, SIGNAL(timeout()), this, SLOT(update()));
    finishTimer = new QTimer(this);
    finishTimer->setSingleShot(true);
    connect(finishTimer, SIGNAL(timeout()), this, SIGNAL(timeout()));
    connect(finishTimer, SIGNAL(timeout()), updateTimer, SLOT(stop()));
    QFont font;
    font.setPointSize(8);
    setFont(font);
}
```

В конструкторе мы создаем два объекта `QTimer`: `updateTimer` используется для обновления внешнего вида виджета каждые пять секунд, а `finishTimer` генерирует сигнал виджета `timeout()` при достижении отметки 0. `finishTimer` должен генерировать только один сигнал тайм-аута, поэтому мы вызываем `setSingleShot(true)`; по умолчанию таймеры запускаются повторно, пока они не будут остановлены или не будут уничтожены. Последний вызов `connect()` является оптимизационным и обеспечивает прекращение обновления виджета, когда таймер неактивен. В конце конструктора мы задаем размер шрифта, используемого для рисования виджета, в 9 пунктов. Делается это для того, чтобы цифры, отображаемые на таймерах, имели везде приблизительно одинаковый размер.

```
void OvenTimer::setDuration(int secs)
{
    secs = qBound(0, secs, MaxSeconds);
    finishTime = QDateTime::currentDateTime().addSecs(secs);
    if (secs > 0) {
        updateTimer->start(UpdateInterval * 1000);
        finishTimer->start(secs * 1000);
    } else {
        updateTimer->stop();
        finishTimer->stop();
    }
    update();
}
```

Функция `setDuration()` выставляет таймер духовки, задавая требуемое количество секунд. Используя глобальную функцию `Qt` `qBound()`, мы можем не писать код типа:

```
if (secs < 0) {
    secs = 0;
```

```

} else if (secs > MaxSeconds) {
    secs = MaxSeconds;
}

```

Время окончания мы рассчитываем путем добавления продолжительности его работы к текущему времени, полученному функцией `QDateTime::currentDateTime()`, и сохраняем его в закрытой переменной `finishTime`. В конце мы вызываем `update()` для перерисовки виджета с новой продолжительностью работы.

Переменная `finishTime` имеет тип `QDateTime`. Поскольку она содержит дату и время, мы избегаем ошибки из-за смены суток, когда текущее время оказывается до полуночи, а время окончания – после полуночи.

```

int OvenTimer::duration() const
{
    int secs = QDateTime::currentDateTime().secsTo(finishTime);
    if (secs < 0)
        secs = 0;
    return secs;
}

```

Функция `duration()` возвращает количество секунд, оставшееся до завершения работы таймера. Если таймер неактивен, мы возвращаем 0.

```

void OvenTimer::mousePressEvent(QMouseEvent *event)
{
    QPointF point = event->pos() - rect().center();
    double theta = std::atan2(-point.x(), -point.y()) * 180.0 * M_PI / 3.14159265359;
    setDuration(duration() + int(theta / DegreesPerSecond));
    update();
}

```

Если пользователь щелкает по этому виджету, мы находим ближайшую метку, используя тонкую, но эффективную математическую формулу, а результат идет на установку новой продолжительности таймера. Затем мы генерируем событие перерисовки. Метка, по которой щелкнул пользователь, теперь будет располагаться сверху поворотного диска и будет поворачиваться против часовой стрелки до тех пор, пока не будет достигнуто значение 0.

```

void OvenTimer::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    int side = qMin(width(), height());
    painter.setViewport((width() - side) / 2, (height() - side) / 2,
                        side, side);
    painter.setWindow(-50, -50, 100, 100);
    draw(&painter);
}

```

В `paintEvent()` мы устанавливаем область отображения на максимальный квадрат, который можно разместить внутри виджета, и мы устанавливаем окно на прямоугольник `(-50, -50, 100, 100)`, то есть, на прямоугольник с размерами

100 × 100, который покрывает пространство от точки (−50, −50) до точки (+50, +50). Шаблонная функция `qMin` возвращает наименьшее из двух значений аргументов. Затем мы вызываем функцию `draw()` для фактического вывода рисунка на экран.

Если область отображения не была бы квадратом, таймер духовки принял бы форму эллипса, когда форма виджета перестанет быть квадратной после изменения его размеров. Чтобы избежать такой деформации, мы должны устанавливать область отображения и окно на прямоугольник с одинаковым соотношением сторон.

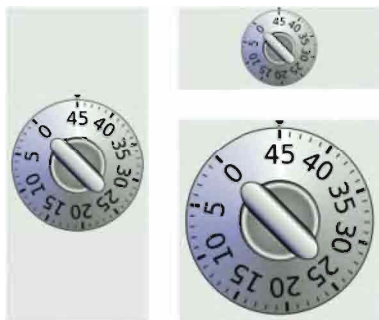


Рис. 8.12. Вид виджета `OvenTimer` при трех различных размерах

Теперь давайте рассмотрим программный код рисования:

```
void OvenTimer::draw(QPainter *painter)
{
    static const int triangle[3][2] = {
        { -2, -49 }, { +2, -49 }, { 0, -47 }
    };
    QPen thickPen(palette().foreground(), 1.5);
    QPen thinPen(palette().foreground(), 0.5);
    QColor niceBlue(150, 150, 200);

    painter->setPen(thinPen);
    painter->setBrush(palette().foreground());
    painter->drawPolygon(QPolygon(3, &triangle[0][0]));
```

Мы начинаем с отображения маленького треугольника в позиции 0 в верхней части виджета. Этот треугольник задается в программе тремя фиксированными координатами, и мы используем функцию `drawPolygon()` для его воспроизведения.

Одно из удобств применения механизма «окно – область отображения» заключается в том, что мы можем при программировании в командах рисования жестко задавать координаты точек и, тем не менее, добиваться необходимого изменения размеров.

```
QConicalGradient coneGradient(0, 0, -90.0);
coneGradient.setColorAt(0.0, Qt::darkGray);
coneGradient.setColorAt(0.2, niceBlue);
coneGradient.setColorAt(0.5, Qt::white);
coneGradient.setColorAt(1.0, Qt::darkGray);
```



```
painter->setBrush(coneGradient);
painter->drawEllipse(-46, -46, 92, 92);
```

Мы рисуем внешнюю окружность и заполняем ее, используя конический градиент. Центр градиента находится в точке (0, 0), а его угол равен -90° .

```
QRadialGradient haloGradient(0, 0, 20, 0, 0);
haloGradient.setColorAt(0.0, Qt::lightGray);
haloGradient.setColorAt(0.8, Qt::darkGray);
haloGradient.setColorAt(0.9, Qt::white);
haloGradient.setColorAt(1.0, Qt::black);
```

```
painter->setPen(Qt::NoPen);
painter->setBrush(haloGradient);
painter->drawEllipse(-20, -20, 40, 40);
```

Мы заполняем внутреннюю окружность, используя радиальный градиент. Центр и фокус градиента располагаются в точке (0, 0). Радиус градиента равен 20.

```
QLinearGradient knobGradient(-7, -25, 7, -25);
knobGradient.setColorAt(0.0, Qt::black);
knobGradient.setColorAt(0.2, niceBlue);
knobGradient.setColorAt(0.3, Qt::lightGray);
knobGradient.setColorAt(0.8, Qt::white);
knobGradient.setColorAt(1.0, Qt::black);
```

```
painter->rotate(duration() * DegreesPerSecond);
painter->setBrush(knobGradient);
painter->setPen(thinPen);
painter->drawRoundRect(-7, -25, 14, 50, 150, 50);
```

```
for (int i = 0; i <= MaxMinutes; ++i) {
    if (i % 5 == 0) {
        painter->setPen(thickPen);
        painter->drawLine(0, -41, 0, -44);
        painter->drawText(-15, -41, 30, 25,
                           Qt::AlignHCenter | Qt::AlignTop,
                           QString::number(i));
    } else {
        painter->setPen(thinPen);
        painter->drawLine(0, -42, 0, -44);
    }
    painter->rotate(-DegreesPerMinute);
}
```

Мы вызываем функцию `rotate()` для поворота системы координат рисовальщика. В старой системе координат нулевая отметка находилась сверху; теперь нулевая отметка перемещается для установки соответствующего времени, которое остается до срабатывания таймера. После каждого поворота мы снова рисуем ручку таймера, поскольку его ориентация зависит от угла поворота.

В цикле `for` мы рисуем минутные отметки по внешней окружности и отображаем количество минут через каждые пять минутных меток. Текст размещается в невидимом прямоугольнике под минутной отметкой. В конце каждой итерации цикла мы поворачиваем рисовальщик по часовой стрелке на 7° , что соответствует одной минуте. При рисовании минутной отметки следующий раз она будет отображаться в другом месте окружности, хотя мы передаем одни и те же координаты функциям `drawLine()` и `drawText()`.

В этом программном коде в цикле `for` имеется незаметная погрешность, которая быстро стала бы очевидной, если бы мы выполнили больше итераций. При каждом вызове `rotate()` мы фактически умножаем текущую универсальную матрицу преобразования на матрицу поворота, получая новую универсальную матрицу преобразования. Ошибка округления чисел с плавающей точкой постепенно накапливается и еще больше увеличивает неточность универсальной матрицы преобразования. Ниже показан один из возможных способов решения этой проблемы путем перезаписи программного кода, с использованием `save()` и `restore()` для сохранения и восстановления первоначального преобразования на каждом шаге итерации:

```
for (int i = 0; i <= MaxMinutes; ++i) {
    painter->save();
    painter->rotate(-i * DegreesPerMinute);
    if (i % 5 == 0) {
        painter->setPen(thickPen);
        painter->drawLine(0, -41, 0, -44);
        painter->drawText(-15, -41, 30, 30,
                           Qt::AlignHCenter | Qt::AlignTop,
                           QString::number(i));
    } else {
        painter->setPen(thinPen);
        painter->drawLine(0, -42, 0, -44);
    }
    painter->restore();
}
```

При другом способе реализации таймера духовки нам нужно было бы самим рассчитывать координаты (x, y) , используя функции `sin()` и `cos()` для определения их позиции на окружности. Но тогда нам все же пришлось бы выполнять перенос и поворот системы координат для отображения текста под некоторым углом.

Высококачественное воспроизведение изображения при помощи QImage

При рисовании мы можем столкнуться с необходимостью принятия компромиссных решений относительно скорости и точности. Например, в системах X11 и Mac OS X рисование по виджету `QWidget` или по пиксельной карте `QPixmap` основано на применении родного для платформы графического процессора (`paint engine`). В системе X11 это обеспечивает минимальную связь с X-сервером; посылаются только команды рисования, а не данные реального изображения. Основ-

ным недостатком этого подхода является то, что возможности Qt ограничивают-
ся родными для данной платформы средствами поддержки:

- в системе X11 такие возможности, как сглаживание линий и поддержка дробных координат доступны только в том случае, если X-сервер использует расширение X Render;
- в системе Mac OS X родной графический процессор, обеспечивающий сглаживание линий, использует алгоритмы рисования многоугольников, кото-
рые отличаются от алгоритмов в X11 и Windows, что приводит к получению
немного других результатов.

Когда точность важнее эффективности, мы можем рисовать по QImage и ко-
пировать результат на экран. В этом случае Qt всегда использует собственный
внутренний графический процессор, и результат на всех платформах получается
идентичным. Единственное ограничение заключается в том, что QImage, по ко-
торому мы рисуем, должен создаваться с аргументом QImage::Format_RGB32 или
QImage::Format_ARGB32_Premultiplied.

Второй формат почти идентичен обычному формату ARGB32 (0xAARRGGBB); отли-
чие в том, что красный, зеленый и синий компоненты «предварительно умножают-
ся» на альфа-компонент. Это значит, что значения RGB, которые обычно находятся
в диапазоне от 0x00 до 0xFF, теперь принимают значения от 0x00 до значения альфа-
компонента. Например, синий цвет с прозрачностью 50% представляется значени-
ем 0x7F0000FF в формате ARGB32, но он имеет значение 0x7F00007F в формате ARGB32
с предварительным умножением компонент, и, аналогично, темно-зеленый цвет
с прозрачностью 75% имеет значение 0x3F008000 в формате ARGB32 и значение
0x3F002000 в формате ARGB32 с предварительным умножением компонент.

Предположим, что мы хотим использовать сглаживание линий при рисо-
вании виджета, и нам нужно получить хорошие результаты даже в системах X11,
которые не используют расширение X Render. Обработчик событий paintEvent(),
предполагающий применение X Render для сглаживания линий, мог бы выгля-
деть следующим образом:

```
void MyWidget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);
    painter.setRenderHint(QPainter::Antialiasing, true);
    draw(&painter);
}
```

Ниже показано, как можно переписать виджетную функцию paintEvent() для
применения независимого от платформы графического процессора Qt:

```
void MyWidget::paintEvent(QPaintEvent *event)
{
    QImage image(size(), QImage::Format_ARGB32_Premultiplied);
    QPainter imagePainter(&image);
    imagePainter.initFrom(this);
    imagePainter.setRenderHint(QPainter::Antialiasing, true);
    imagePainter.eraseRect(rect());
    draw(&imagePainter);
    imagePainter.end();
}
```

```

QPainter widgetPainter(this);
widgetPainter.drawImage(0, 0, image);
}

```

Мы создаем объект `QImage` с тем же размером, который имеет виджет в формате `ARGB32` с умножением компонент, и объект `QPainter` для рисования по изображению. Вызов `initFrom()` инициализирует в рисовальщике перо, фон и шрифт значениями, используемыми виджетом. Мы рисуем, используя `QPainter` как обычно, а в конце еще раз используем объект `QPainter` для копирования изображения на виджет.

Этот подход дает одинаково высококачественные результаты на всех платформах, за исключением воспроизведения шрифта, что зависит от установленных в системе шрифтов.

Особенно эффективным средством графического процессора Qt является его поддержка режимов композиции. Эти режимы определяют способ слияния исходного и нового пикселя при рисовании. Это относится ко всем операциям рисования, в том числе относящихся к перу, кисти, градиенту и изображению.

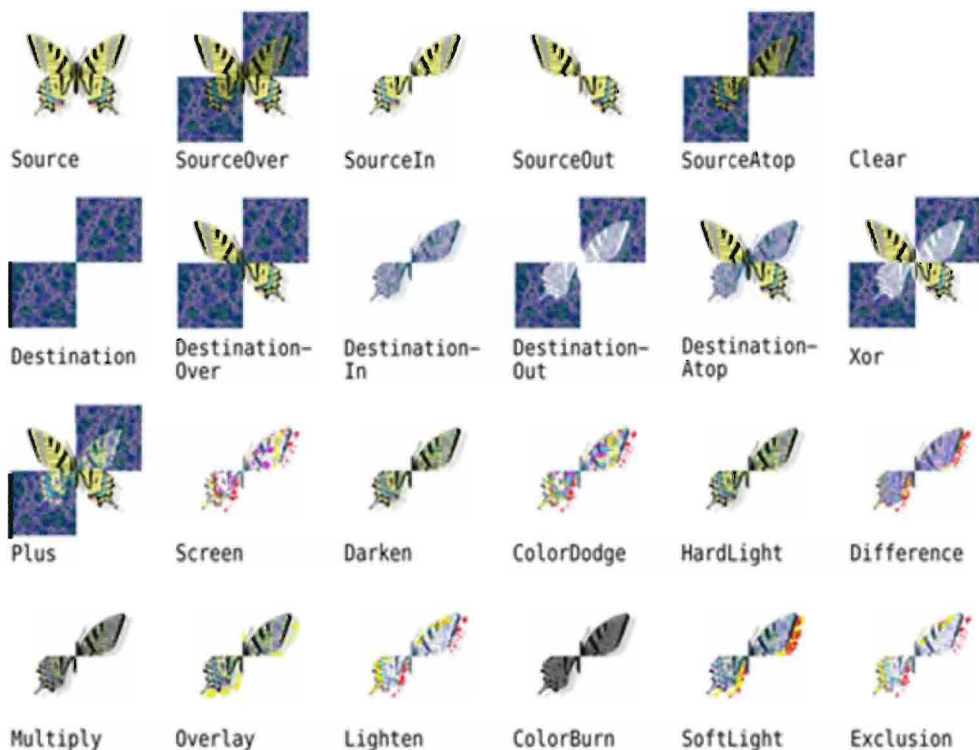


Рис. 8.13. Режимы композиции QPainter

Режимом композиции по умолчанию является `QImage::CompositionMode_SourceOver`, означающий, что исходный пиксель (тот, который рисуется в данный момент) налагается поверх существующего на изображении пикселя, причем альфа-компонент исходного пикселя определяет степень его прозрачности. На

рис. 8.13 показан результат рисования полупрозрачной бабочки («исходное» изображение) поверх тестового шаблона («существующее» изображение) при использовании разных режимов.

Режимы композиции устанавливаются функцией `QPainter::setCompositionMode()`. Например, ниже показано, как можно создать объект `QImage`, объединяющий пиксели бабочки и тестового шаблона с помощью операции XOR:

```
QImage resultImage = checkerPatternImage;  
QPainter painter(&resultImage);  
painter.setCompositionMode(QPainter::CompositionMode_Xor);  
painter.drawImage(0, 0, butterflyImage);
```

Следует иметь в виду, что операция `QImage::CompositionMode_Xor` также применяется к альфа-компоненту. Это означает, что если мы применим операцию XOR при наложении белого цвета (`0xFFFFFFFF`) на белый цвет, мы получим прозрачный цвет (`0x00000000`), а не черный цвет (`0xFF000000`).

Элементное воспроизведение с помощью графического представления

Рисование с использованием `QPainter` идеально подходит для пользовательских виджетов и для рисования одного или нескольких элементов. Если же нам нужно работать с любым количеством элементов, от нескольких штук до нескольких тысяч, и если мы хотим, чтобы пользователь имел возможность щелкать по элементам мышкой, перетаскивать их и выбирать, то необходимое нам решение предоставляют классы графических представлений Qt. Архитектура графических представлений состоит из сцены, представленной классом `QGraphicsScene`, и элементами на ней, которые представлены подклассами `QGraphicsItem`. Сцена (и ее элементы) отображаются для пользователей в представлении, представленном классом `QGraphicsView`. Одна и та же сцена может отображаться в более чем одном представлении, например, с целью отображения разных частей большой сцены или для отображения сцены в условиях разных трансформаций. Схематично это показано на рис. 8.14.

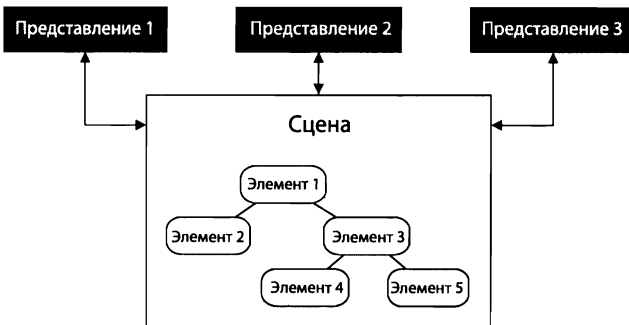


Рис. 8.14. Одна сцена может быть связана с несколькими представлениями

Существует несколько готовых подклассов класса `QGraphicsItem`, в том числе: `QGraphicsLineItem`, `QGraphicsPixmapItem`, `QGraphicsSimpleTextItem` (для простого текста

со стилями) и `QGraphicsTextItem` (для формата rich text) (см. рис. 8.15). Мы также можем создавать собственные пользовательские подклассы класса `QGraphicsItem`, и об этом рассказывается в следующем разделе.

Класс `QGraphicsScene` хранит коллекцию графических элементов. Сцена имеет три уровня – уровень фона, уровень элементов и уровень переднего плана. Фон и передний план, как правило, определяются классами `QBrush`, но существует возможность переопределить функции `drawBackground()` или `drawForeground()`, получив над этими уровнями полный контроль. Если мы хотим использовать в качестве фона пиксельное изображение, мы можем просто создать текстуру `QBrush` на основе этого изображения. Для кисти, рисующей передний план, можно задать полупрозрачный белый цвет, получив эффект блеклости, или задать узор из перекрещенных линий, получив наложенную сетку.

Сцена может подсказать нам, какие элементы накладываются друг на друга, какие являются выбранными, и какие находятся в конкретной точке и конкретной области. Графические элементы сцены являются либо элементами верхнего уровня (их родителем является сцена) или дочерними элементами (их родителем является другой элемент). Любые трансформации, применяемые к элементу, автоматически применяются к его потомкам.

Архитектура графических представлений предлагает два способа группировки элементов. Первый способ – это просто сделать элемент потомком другого элемента. Другой способ – использовать класс `QGraphicsItemGroup`. Добавление элементов в группу не вызывает никакой его трансформации. Такие группы удобно использовать для работы с несколькими элементами как с одним элементом.

`QGraphicsView` представляет собой виджет, который отображает сцену, предлагая при необходимости полосы прокрутки и способен применять трансформации, влияющие на способ. Это полезно для поддержки масштабирования и поворота при просмотре сцены.

По умолчанию виджет `QGraphicsView` отображается с использованием встроенной в Qt двумерной системы прорисовки, но это можно изменить, и задать использование виджета OpenGL при помощи одного вызова функции `setViewport()` после того, как область отображения будет сконструирована. Также весьма просто распечатать сцену или ее фрагменты, о чем мы поговорим в следующем разделе, где рассматривается несколько имеющихся в Qt методов вывода на печать.

В данной архитектуре используются три разные системы координат – координаты области отображения, координаты сцены и координаты элемента – с функцией перевода из одной системы координат в другую. Координаты области отображения – это координаты в пределах области отображения (viewport) `QGraphicsView`. Координаты сцены – это логические координаты, используемые для позиционирования элементов верхнего уровня на сцене. Координаты элементов специфичны для каждого элемента и отсчитываются от локальной точки (0, 0) данного элемента. Они остаются неизменными при перемещении элемента в пределах сцены. На практике нас обычно интересуют только координаты сцены (для позиционирования элементов верхнего уровня) и координаты элементов (для позиционирования элементов-потомков и для рисования на элементах). Рисование на элементе по его локальной системе координат означает, что нам не нужно беспокоиться о том, где на сцене находится данный элемент, и какие трансформации были к нему применены.

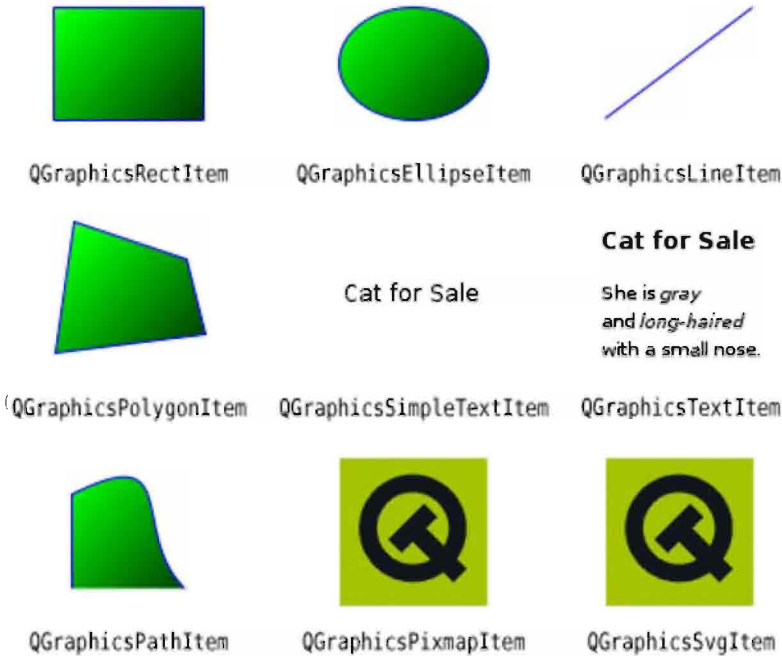


Рис. 8.15. Элементы графических представлений, имеющиеся в Qt 4.3

Классы графических представлений весьма просто использовать, и они предлагают обширные возможности. Чтобы вы познакомились с тем, что можно сделать с их помощью, мы рассмотрим два примера. Первый пример – это простой редактор схем, на примере которого показано, как создавать элементы, и как осуществлять взаимодействие с пользователем. Второй пример – это программа, представляющая собой карту с аннотациями, показывающая, как обрабатывать большое число графических объектов и как эффективно отображать их при разном увеличении.

Приложение Diagram (схема), показанное на рис. 8.16, позволяет пользователям создавать узлы и связи. Узлы представляют собой графические элементы, где простой текст отображается внутри прямоугольника со скругленными углами, а соединения – это линии, соединяющие пары узлов. Выбранные узлы отображаются пунктирным контуром, который рисуется более толстым пером, чем обычный. Мы начнем с рассмотрения соединений, поскольку они самые простые, затем перейдем к узлам, и, наконец, увидим, как они используются в контексте.

```
class Link : public QGraphicsLineItem
{
public:
    Link(Node *fromNode, Node *toNode);
    ~Link();
    Node *fromNode() const;
    Node *toNode() const;
    void setColor(const QColor &color);
```

```

    QColor color() const;
    void trackNodes();
private:
    Node *myFromNode;
    Node *myToNode;
};

```

Класс `Link` происходит от класса `QGraphicsLineItem`, который представляет собой линию в объекте `QGraphicsScene`. Соединение имеет три главных атрибута: два узла, которые оно связывает, и цвет, используемый для рисования линии. Нам не нужна переменная-член `QColor` для хранения цвета по причинам, которые вскоре станут ясны.

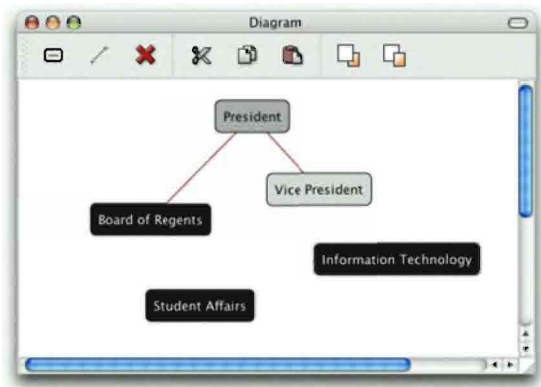


Рис. 8.16. Приложение `Diagram`

Элемент `QGraphicsItem` не является подклассом `QObject`, но если мы захотим добавить к классу `Link` сигналы и слоты, то ничего нам не мешает использовать множественное наследование от `QObject`. Функция `trackNodes()` используется для обновления конечных точек линии в том случае, если пользователь перенесет узел с установленными соединениями в другое место.

```

Link::Link(Node *fromNode, Node *toNode)
{
    myFromNode = fromNode;
    myToNode = toNode;
    myFromNode->addLink(this);
    myToNode->addLink(this);
    setFlags(QGraphicsItem::ItemIsSelectable);
    setZValue(-1);
    setColor(Qt::darkRed);
    trackNodes();
}

```

Когда соединение создано, оно добавляется к узлам, которые оно соединяет. В каждом узле хранится набор соединений, и этих соединений может быть сколько угодно. Графические элементы содержат несколько флагов, но в данном

случае нам нужно только, чтобы соединения можно было выбрать (чтобы пользователь мог выделить соединение и удалить его).

Каждый графический элемент имеет координату (x, y) и значение z , показывающее, насколько близко он находится от переднего и заднего плана сцены. Поскольку мы собираемся рисовать наши линии от центра одного узла к центру другого, мы присваиваем линии отрицательное значение z , означающее, что линия всегда будет рисоваться под узлами, которые она соединяет.

В конце конструктора мы задаем исходный цвет линии, а затем задаем конечные точки линии, вызывая функцию `trackNodes()`.

```
Link::~Link()
{
    myFromNode->removeLink(this);
    myToNode->removeLink(this);
}
```

Когда соединение удаляется, оно убирает себя из узлов, которые оно соединяет.

```
void Link::setColor(const QColor &color)
{
    setPen(QPen(color, 1.0));
}
```

Когда цвет линии соединения задан, мы просто изменяем перо, используя указанный цвет и толщину пера 1. Функция `setPen()` наследуется от `QGraphicsLineItem`. Функция `color()` просто возвращает цвет пера.

```
void Link::trackNodes()
{
    setLine(QLineF(myFromNode->pos(), myToNode->pos()));
}
```

Функция `QGraphicsItem::pos()` возвращает положение соответствующего графического элемента относительно сцены (для элементов верхнего уровня) или относительно родительского элемента (для элементов-потомков). В случае класса `Link` мы при обработке прорисовки полагаемся на базовый класс: `QGraphicsLineItem` рисует линию (с помощью функции `pen()`) между двумя точками на сцене. В случае класса `Node` мы обрабатываем всю графику самостоятельно. Другое отличие между узлами и соединениями состоит в том, что узлы более интерактивны. Мы начнем с объявления класса `Node`, разделив его на несколько фрагментов, поскольку оно очень длинное.

```
class Node : public QGraphicsItem
{
    Q_DECLARE_TR_FUNCTIONS(Node)
public:
    Node();
```

Для класса `Node` в качестве базового класса мы используем `QGraphicsItem`. Макрос `Q_DECLARE_TR_FUNCTIONS()` используется для добавления к классу функции `tr()`, несмотря на то, что он не является подклассом `QObject`. Делается это просто для удобства, чтобы мы могли использовать `tr()`, а не статическую функцию `QObject::tr()` или `QCoreApplication::translate()`.

```

void setText(const QString &text);
QString text() const;
void setTextColor(const QColor &color);
QColor textColor() const;
void setOutlineColor(const QColor &color);
QColor outlineColor() const;
void setBackgroundColor(const QColor &color);
QColor backgroundColor() const;

```

Эти функции представляют собой просто средства чтения и записи закрытых членов. Мы обеспечиваем контроль над цветом текста, контуром узла и фоном узла.

```

void addLink(Link *link);
void removeLink(Link *link);

```

Как мы видели ранее, эти функции вызываются классом `Link` для добавления или удаления соединений из узла.

```

QRectF boundingRect() const;
QPainterPath shape() const;
void paint(QPainter *painter,
           const QStyleOptionGraphicsItem *option, QWidget *widget);

```

Когда мы создаем подклассы `QGraphicsItem`, прорисовку которых хотим осуществлять вручную, нам нужно переопределить функции `boundingRect()` и `paint()`. Если мы не переопределим функцию `shape()`, то реализация базового класса вернется к `boundingRect()`. В данном случае мы переопределили `shape()`, чтобы получить более точную форму, учитывающую скругленные углы узлов.

В архитектуре графических представлений граничный прямоугольник используется для определения того, нужно ли рисовать элемент. Это позволяет классу `QGraphicsView` отображать большие произвольные сцены очень быстро, если в любой данный момент времени видна лишь часть элементов. Функция `shape` используется для определения того, расположена ли точка внутри элемента и перекрываются ли два элемента.

```

protected:
void mouseDoubleClickEvent(QGraphicsSceneMouseEvent *event);
QVariant itemChange(GraphicsItemChange change,
                    const QVariant &value);

```

В приложении `Diagram` мы предоставляем диалоговое окно `Properties` (свойства) для редактирования позиции узла, его цвета и текста. Для дополнительного удобства мы позволим пользователю изменять текст, сделав двойной щелчок мышью по узлу.

Если узел перемещается, мы должны обеспечить соответствующее обновление всех связанных с ним соединений. Для этого мы переопределим обработчик `itemChange()`. Этот обработчик вызывается, если изменяются свойства элемента (включая его положение). Функцию `mouseMoveEvent()` мы не используем для этого из-за того, что она не вызывается в том случае, если перемещение узла осуществляется программно.

```
private:
    QRectF outlineRect() const;
    int roundness(double size) const;
    QSet<Link *> myLinks;
    QString myText;
    QColor myTextColor;
    QColor myBackgroundColor;
    QColor myOutlineColor;
};
```

Закрытая функция `outlineRect()` возвращает прямоугольник, нарисованный классом `Node`, а функция `roundness()` возвращает соответствующий коэффициент закругленности на основе высоты и ширины прямоугольника.

Так же, как класс `Link` отслеживает все узлы, которые он соединяет, класс `Node` отслеживает свои соединения. При удалении узла все соединения этого узла также удаляются.

Теперь мы готовы рассмотреть реализацию класса `Node`, начиная, как обычно, с конструктора.

```
Node::Node()
{
    myTextColor = Qt::darkGreen;
    myOutlineColor = Qt::darkBlue;
    myBackgroundColor = Qt::white;
    setFlags(ItemIsMovable | ItemIsSelectable);
}
```

Мы инициализируем цвета и делаем элементы перемещаемыми и предоставляем возможность их выбирать. Значение `z` по умолчанию равно 0, а положение узла на сцене будет определять тот, кто вызовет функцию.

```
Node::~~Node()
{
    foreach (Link *link, myLinks)
        delete link;
}
```

Деструктор удаляет все соединения узла. При удалении соединения оно удаляется из узлов, к которым оно подключено. Мы перебираем набор ссылок (копию), а не используем функцию `qDeleteAll()`, чтобы избежать побочных эффектов, поскольку доступ к набору ссылок деструктор класса `Link` осуществляет не напрямую.

```
void Node::setText(const QString &text)
{
    prepareGeometryChange();
    myText = text;
    update();
}
```

Когда мы изменяем графический элемент так, что это влияет на его внешний вид, мы должны вызвать функцию `update()`, чтобы запланировать перерисовку.

И в тех случаях, где может измениться граничный прямоугольник элемента (из-за того, что новый текст оказывается длиннее или короче предыдущего), мы должны сразу же вызвать функцию `prepareGeometryChange()` до того, как мы сделаем что-то, что повлияет на граничный прямоугольник. Мы пропустим функции чтения `text()`, `textColor()`, `outlineColor()` и `backgroundColor()`, поскольку они просто возвращают соответствующий закрытый член.

```
void Node::setTextColor(const QColor &color)
{
    myTextColor = color;
    update();
}
```

Когда мы задаем цвет текста, мы должны вызвать функцию `update()`, чтобы запланировать перерисовку. Делается этого для того, чтобы элемент был прорисован новым цветом. Не нужно вызывать функцию `prepareGeometryChange()`, поскольку при изменении цвета размер элемента не изменяется. Мы опустим функции задания значений цвета контура и фона, поскольку они структурно аналогичны данной функции.

```
void Node::addLink(Link *link)
{
    myLinks.insert(link);
}
void Node::removeLink(Link *link)
{
    myLinks.remove(link);
}
```

Здесь мы просто добавляем данное соединение к списку соединений узла или удаляем его из списка.

```
QRectF Node::outlineRect() const
{
    const int Padding = 8;
    QFontMetricsF metrics = qApp->font();
    QRectF rect = metrics.boundingRect(myText);
    rect.adjust(-Padding, -Padding, +Padding, +Padding);
    rect.translate(-rect.center());
    return rect;
}
```

Мы используем данную закрытую функцию для вычисления прямоугольника, заключающего в себе текст узла с 8-пиксельным полем. Граничный прямоугольник, возвращаемый функцией измерения шрифта, всегда имеет начало координат (0, 0) в верхнем левом углу. Поскольку нам нужно, чтобы текст был центрирован по центру элемента, мы перемещаем начало координат прямоугольника на центр.

Хотя мы ведем рассуждения и вычисления в пикселях, эта единица, в некотором смысле, воображаемая. Сцена (родительский элемент) может масштабироваться, поворачиваться, обрезаться или к ней просто может применяться сглаживание линий, так что реальное число пикселей, отображаемое на экране, может быть иным.

```
QRectF Node::boundingRect() const
{
    const int Margin = 1;
    return outlineRect().adjusted(-Margin, -Margin, +Margin, +Margin);
}
```

Функция `boundingRect()` вызывается классом `QGraphicsView` для того, чтобы определить, нужно ли рисовать элемент. Мы используем прямоугольник-контур, но с небольшими дополнительными полями, поскольку прямоугольник, возвращаемый этой функцией, если контур будет рисоваться, должен учитывать, по меньшей мере, половину толщины пера.

```
QPainterPath Node::shape() const
{
    QRectF rect = outlineRect();
    QPainterPath path;
    path.addRoundRect(rect, roundness(rect.width()),
                     roundness(rect.height()));
    return path;
}
```

Функция `shape()` вызывается классом `QGraphicsView` для тонкого определения касания. Часто мы можем опустить ее, и позволить производить вычисления по фигуре по граничному прямоугольнику. Здесь мы переопределим эту функцию, чтобы она возвращала объект `QPainterPath`, представляющий прямоугольник с закругленными углами. Следовательно, щелчок мышью по тем областям углов, которые выходят за рамки закругления, но находятся внутри граничного прямоугольника, не приведет к выделению элемента.

Когда мы создаем прямоугольник с закругленными углами, мы можем передавать дополнительные аргументы, указывающие степень закругления углов. Мы вычислим подходящие значения с помощью закрытой функции `roundness()`.

```
void Node::paint(QPainter *painter,
const QStyleOptionGraphicsItem *option,
QWidget * /* widget */)
{
    QPen pen(myOutlineColor);
    if (option->state & QStyle::State_Selected) {
        pen.setStyle(Qt::DotLine);
        pen.setWidth(2);
    }
    painter->setPen(pen);
    painter->setBrush(myBackgroundColor);
    QRectF rect = outlineRect();
    painter->drawRoundRect(rect, roundness(rect.width()),
                        roundness(rect.height()));
    painter->setPen(myTextColor);
    painter->drawText(rect, Qt::AlignCenter, myText);
}
```

В функции `paint()` мы производим рисование элемента. Если элемент выбран, мы меняем стиль пера на пунктир и увеличиваем толщину. В противном случае используется заданная по умолчанию сплошная линия толщиной в 1 пиксел. Также мы указываем, что кисть будет использовать фоновый цвет.

Далее мы рисуем прямоугольник с закругленными углами, но используем фактор закругления, полученный закрытой функцией `roundness()`. Наконец, мы пишем центрированный текст внутри контурного прямоугольника поверх прямоугольника с закругленными углами.

Дополнительный параметр `QStyleOptionGraphicsItem` представляет собой нетипичный для Qt класс, поскольку он содержит несколько открытых переменных-членов. Сюда входит текущее направление компоновки, размер шрифта, палитра, прямоугольник, состояние («selected», «has focus» и многие другие), матрица трансформации и степень детализации. Здесь мы отметили член, отвечающий за состояние, чтобы проверить, выбран ли узел.

```
QVariant Node::itemChange(GraphicsItemChange change,
                           const QVariant &value)
{
    if (change == ItemPositionHasChanged) {
        foreach (Link *link, myLinks)
            link->trackNodes();
    }
    return QGraphicsItem::itemChange(change, value);
}
```

Когда пользователь перетаскивает узел, вызывается обработчик `itemChange()`, которому в качестве первого аргумента передается `ItemPositionHasChanged`. Чтобы обеспечить правильное позиционирование соединений, мы перебираем список соединений узла и для каждого обновляем конечные точки линии. В завершение, мы вызываем реализацию базового класса, чтобы она также получила уведомление.

```
void Node::mouseDoubleClickEvent(QGraphicsSceneMouseEvent *event)
{
    QString text = QInputDialog::getText(event->widget(),
                                          tr("Edit Text"), tr("Enter new text:"),
                                          QLineEdit::Normal, myText);

    if (!text.isEmpty())
        setText(text);
}
```

Если пользователь сделает двойной щелчок мышью по узлу, мы отображаем диалоговое окно, отображающее текущий текст и даем возможность пользователю изменить его. Если пользователь нажимает кнопку Cancel (отмена), возвращается пустая строка, и, следовательно, изменение применяется только в том случае, если строка не пуста. Вскоре мы увидим, как можно изменять другие свойства узла (например, цвета).

```
int Node::roundness(double size) const
{
    const int Diameter = 12;
    return 100 * Diameter / int(size);
}
```

Функция `roundness()` возвращает соответствующие факторы закругления, которые гарантируют, что углы узла будут представлять собой четверть окружности с диаметром 12. Факторы закругления должны быть в диапазоне от 0 (квадрат) до 99 (полностью закругленный угол).

Итак, мы увидели реализацию двух пользовательских классов графических элементов. Теперь пора посмотреть, как они используются на практике. Приложение `Diagram` представляет собой стандартное приложение с главным окном, меню и панелями инструментов. Мы не будем рассматривать все детали реализации, а сконцентрируемся на тех, которые относятся к архитектуре графических представлений. Начнем с рассмотрения фрагмента определения подкласса `QMainWindow`.

```
class DiagramWindow : public QMainWindow
{
    Q_OBJECT
public:
    DiagramWindow();
private slots:
    void addNode();
    void addLink();
    void del();
    void cut();
    void copy();
    void paste();
    void bringToFront();
    void sendToBack();
    void properties();
    void updateActions();
private:
    typedef QPair<Node *, Node *> NodePair;
    void createActions();
    void createMenus();
    void createToolBars();
    void setZValue(int z);
    void setupNode(Node *node);
    Node *selectedNode() const;
    Link *selectedLink() const;
    NodePair selectedNodePair() const;
    QMenu *fileMenu;
    QMenu *editMenu;
    QToolBar *editToolBar;
    QAction *exitAction;
    ...
    QAction *propertiesAction;
    QGraphicsScene *scene;
    QGraphicsView *view;
    int minZ;
    int maxZ;
    int seqNumber;
};
```

Назначение большинства закрытых слотов должно быть понятно из их названий. Слот `properties()` используется для отображения диалогового окна `Properties` (свойства), если выбран узел или окна `QColorDialog`, если выбрано соединение. Слот `updateActions()` используется для включения и отключения команд в зависимости от того, какие элементы выбраны.

```
DiagramWindow::DiagramWindow()
{
    scene = new QGraphicsScene(0, 0, 600, 500);
    view = new QGraphicsView;
    view->setScene(scene);
    view->setDragMode(QGraphicsView::RubberBandDrag);
    view->setRenderHints(QPainter::Antialiasing
                       | QPainter::TextAntialiasing);
    view->setContextMenuPolicy(Qt::ActionsContextMenu);
    setCentralWidget(view);
    minZ = 0;
    maxZ = 0;
    seqNumber = 0;
    createActions();
    createMenus();
    createToolBars();
    connect(scene, SIGNAL(selectionChanged()),
           this, SLOT(updateActions()));
    setWindowTitle(tr("Diagram"));
    updateActions();
}
```

Мы начинаем с создания графической сцены с началом координат (0, 0), шириной 600 и высотой 500. Затем мы создаем графическое представление для визуализации сцены. В следующем примере вместо того, чтобы использовать класс `QGraphicsView` напрямую, мы создадим его подкласс для того, чтобы настроить его работу.

Элементы с возможностью выбора можно выбирать щелчком мыши. Чтобы выбрать сразу несколько элементов, пользователь может щелкать по ним мышью, удерживая клавишу `Ctrl`. Если задать режим перетаскивания `QGraphicsView::RubberBandDrag`, пользователь также сможет выделять элементы, перетаскивая на них резиновую ленту.

Значения `minZ` и `maxZ` используются функциями `sendToBack()` и `bringToFront()`. Последовательный номер (`seqNumber`) используется для добавления уникального исходного текста в каждый узел, добавляемый пользователем.

Соединения сигнал-слот гарантируют, что при любом изменении набора выделенных элементов команды приложения будут включаться и отключаться таким образом, чтобы были доступны только относящиеся к данным элементам команды. Для установки исходных состояний доступности вызывается функция `updateActions`.

```
void DiagramWindow::addNode()
{
    Node *node = new Node;
    node->setText(tr("Node %1").arg(seqNumber + 1));
    setupNode(node);
}
```


При добавлении пользователем нового узла мы создаем новый экземпляр класса `Node`, присваиваем ему заданный по умолчанию текст и передаем узел в функцию `setupNode()`, чтобы определить его положение и выбрать его. Мы используем отдельную функцию для завершения добавления узла, поскольку эта функциональность нам снова потребуется при реализации функции `paste()`.

```
void DiagramWindow::setupNode(Node *node)
{
    node->setPos(QPoint(80 + (100 * (seqNumber % 5)),
                      80 + (50 * ((seqNumber / 5) % 7))));
    scene->addItem(node);
    ++seqNumber;
    scene->clearSelection();
    node->setSelected(true);
    bringToFront();
}
```

Данная функция позиционирует добавленный или вставленный узел на сцене. Последовательный номер (`seqNumber`) используется того, чтобы новые узлы помещались в разные места, а не поверх друг друга. Мы очищаем список выбранных элементов и выбираем только новый добавленный узел. Вызов функции `bringToFront()` гарантирует, что новый узел будет располагаться поверх любого другого узла.

```
void DiagramWindow::bringToFront()
{
    ++maxZ;
    setZValue(maxZ);
}

void DiagramWindow::sendToBack()
{
    --minZ;
    setZValue(minZ);
}

void DiagramWindow::setZValue(int z)
{
    Node *node = selectedNode();
    if (node)
        node->setZValue(z);
}
```

Слот `bringToFront()` увеличивает на единицу значение `maxZ`, а затем задает в выбранном в данный момент узле значение `z` равное `maxZ`. Слот `sendToBack()` использует значение `minZ` и производит противоположный эффект. Оба эти слота определяются с помощью закрытой функции `setZValue()`.

```
Node *DiagramWindow::selectedNode() const
{
    QList<QGraphicsItem *> items = scene->selectedItems();
    if (items.count() == 1) {
        return dynamic_cast<Node *>(items.first());
    }
}
```

```

    } else {
        return 0;
    }
}

```

Список всех выделенных элементов на сцене можно получить, вызвав функцию `QGraphicsScene::selectedItems()`. Функция `selectedNode()` возвращает один узел, если выбран один узел, а в противном случае возвращает пустой указатель. Если выбран именно один элемент, преобразование типов приведет к возврату указателя на тип `Node` в случае, если элемент представляет собой узел (`Node`), или пустого указателя, если элемент представляет собой соединение (`Link`).

Существует также функция `selectedLink()`, которая возвращает указатель на выбранный элемент `Link`, если выбран ровно один элемент, представляющий собой соединение.

```

void DiagramWindow::addLink()
{
    NodePair nodes = selectedNodePair();
    if (nodes == NodePair())
        return;
    Link *link = new Link(nodes.first, nodes.second);
    scene->addItem(link);
}

```

Пользователь может добавить соединение только в том случае, если выделены два узла. Если функция `selectedNodePair()` возвращает два выбранных узла, мы создаем новое соединение. Конструктор соединения сделает так, чтобы конечные точки линии находились в центрах первого и второго узлов.

```

DiagramWindow::NodePair DiagramWindow::selectedNodePair() const
{
    QList<QGraphicsItem *> items = scene->selectedItems();
    if (items.count() == 2) {
        Node *first = dynamic_cast<Node *>(items.first());
        Node *second = dynamic_cast<Node *>(items.last());
        if (first && second)
            return NodePair(first, second);
    }
    return NodePair();
}

```

Эта функция сходна с функцией `selectedNode`, которую мы видели ранее. Если выбрано ровно два элемента, и оба они представляют собой узлы, то возвращается эта пара элементов, в противном случае возвращается пара пустых указателей.

```

void DiagramWindow::del()
{
    QList<QGraphicsItem *> items = scene->selectedItems();
    QMutableListIterator<QGraphicsItem *> i(items);
    while (i.hasNext()) {
        Link *link = dynamic_cast<Link *>(i.next());
    }
}

```

```

        if (link) {
            delete link;
            i.remove();
        }
    }
    qDeleteAll(items);
}

```

Данный слот удаляет любые выбранные элементы, т. е. узлы, соединения или смесь тех и других. При удалении узла деструктор удаляет все соединения, которые с ним связаны. Чтобы избежать двойного удаления соединений, мы удаляем соединения до удаления узлов.

```

void DiagramWindow::properties()
{
    Node *node = selectedNode();
    Link *link = selectedLink();
    if (node) {
        PropertiesDialog dialog(node, this);
        dialog.exec();
    } else if (link) {
        QColor color = QColorDialog::getColor(link->color(), this);
        if (color.isValid())
            link->setColor(color);
    }
}

```

Если пользователь запускает команду *Properties* (свойства), и при этом выбран узел, мы вызываем диалоговое окно *Properties*. Это диалоговое окно позволяет пользователю изменить у узла текст, местоположение и цвета. Поскольку *PropertiesDialog* работает напрямую с указателем на тип *Node*, мы можем просто выполнить его модально, и дальше он позаботится о себе сам.

Если выбрано соединение, то мы используем встроенную готовую статическую функцию *Qt QColorDialog::getColor()*, которая отображает диалоговое окно выбора цвета. Если пользователь выбрал цвет, мы задаем этот цвет как цвет соединения. Если изменились свойства узла или цвет соединения, изменения делаются с помощью функций установки значений, и при этом вызывается функция *update()*, которая обеспечивает перерисовку соединения или узла с новыми параметрами.

Пользователям часто бывает нужно в приложениях такого типа вырезать, копировать и вставлять графические элементы, и одним из способов поддержки такой возможности является текстовое представление элементов, и мы увидим это при рассмотрении соответствующего кода. Мы обрабатываем только узлы, поскольку бессмысленно копировать и вставлять соединения, которые существуют только в связи с узлами.

```

void DiagramWindow::cut()
{
    Node *node = selectedNode();
    if (!node)
        return;
}

```

```

copy();
delete node;
}

```

Команда Cut является двухэтапным процессом, включает: копирование выбранного элемента в буфер обмена и удаление элемента. Копирование осуществляется с помощью слота `copy()`, связанного с командой `Copy`, а для удаления используется стандартный оператор C++ `delete`, и деструктору узла дается возможность удалить все соединения, связанные с узлом, и удалить сам узел со сцены.

```

void DiagramWindow::copy()
{
    Node *node = selectedNode();
    if (!node)
        return;
    QString str = QString("Node %1 %2 %3 %4")
        .arg(node->textColor().name())
        .arg(node->outlineColor().name())
        .arg(node->background-color().name())
        .arg(node->text());
    QApplication::clipboard()->setText(str);
}

```

Функция `QColor::name()` возвращает значение `QString`, содержащее строку цвета в стиле HTML в формате `"#RRGGBB"`, где каждый компонент цвета представлен шестнадцатеричным значением в диапазоне от `0x00` до `0xFF` (от 0 до 255). Мы записываем в буфер обмена строку: одну строку текста, начинающуюся со слова «Node», после которого идут три цвета узла и, наконец, текст узла. Все части разделяются пробелами. Пример: `Node #aa0000 #000080 #ffffff Red herring`

Этот текст декодируется функцией `paste()`:

```

void DiagramWindow::paste()
{
    QString str = QApplication::clipboard()->text();
    QStringList parts = str.split(" ");
    if (parts.count() >= 5 && parts.first() == "Node") {
        Node *node = new Node;
        node->setText(QStringList(parts.mid(4)).join(" "));
        node->setTextColor(QColor(parts[1]));
        node->setOutlineColor(QColor(parts[2]));
        node->setBackgroundColor(QColor(parts[3]));
        setupNode(node);
    }
}

```

Мы разрезаем текст в буфере обмена и помещаем его в список `QStringList`. При использовании приведенного выше примера мы получим список `[«Node», «#aa0000», «#000080», «#ffffff», «Red», «herring»]`. Чтобы узел был допустимым, в списке должно быть, как минимум, пять элементов: слово «Node», три цвета и, как минимум, одно слово текста. Если условие выполняется, мы создаем новый узел, а текст узла будет образован конкатенацией пятого и последующе-

го элементов с разделением их пробелами. Мы задаем цвета во втором, третьем и четвертом элементах при помощи конструктора `QColor`, который принимает имена, возвращаемые функцией `QColor::name()`. Для полноты здесь присутствует также слот `updateActions()`, который используется для включения и отключения команд в меню **Edit** и контекстном меню:

```
void DiagramWindow::updateActions()
{
    bool hasSelection = !scene->selectedItems().isEmpty();
    bool isNode = (selectedNode() != 0);
    bool isNodePair = (selectedNodePair() != NodePair());
    cutAction->setEnabled(isNode);
    copyAction->setEnabled(isNode);
    addLinkAction->setEnabled(isNodePair);
    deleteAction->setEnabled(hasSelection);
    bringToFrontAction->setEnabled(isNode);
    sendToBackAction->setEnabled(isNode);
    propertiesAction->setEnabled(isNode);
    foreach (QAction *action, view->actions())
        view->removeAction(action);
    foreach (QAction *action, editMenu->actions()) {
        if (action->isEnabled())
            view->addAction(action);
    }
}
```

Итак, мы закончили рассмотрение приложения **Diagram**, и можем теперь обратиться ко второму примеру использования графического представления, **Cityscape** (карта города). Приложение **Cityscape**, показанное на рис. 8.17, представляет собой вымышленную карту, включающую основные здания, кварталы и парки города, наиболее важные из которых снабжены названиями. Приложение позволяет пользователю прокручивать карту и изменять масштаб при помощи клавиатуры и мыши. Мы начнем с рассмотрения класса **Cityscape**, формирующего главное окно приложения.



Рис. 8.17 Приложение **Cityscape** и два разных масштаба

```

class Cityscape : public QMainWindow
{
    Q_OBJECT
public:
    Cityscape();
private:
    void generateCityBlocks();
    QGraphicsScene *scene;
    CityView *view;
};

```

Данное приложение не имеет меню и панелей инструментов. Оно просто отображает карту с подписями, используя виджет CityView. Класс CityView является подклассом класса QGraphicsView.

```

Cityscape::Cityscape()
{
    scene = new QGraphicsScene(-22.25, -22.25, 1980, 1980);
    scene->setBackgroundBrush(QColor(255, 255, 238));
    generateCityBlocks();
    view = new CityView;
    view->setScene(scene);
    setCentralWidget(view);
    setWindowTitle(tr("Cityscape"));
}

```

Конструктор создает объект QGraphicsScene и вызывает функцию generateCityBlocks() для генерации карты. Карта состоит более чем из 2000 кварталов и 200 подписей. Сначала мы рассмотрим подкласс графических элементов CityBlock (квартал), а затем подкласс графических элементов Annotation (подпись) и, наконец, подкласс графических элементов CityView.

```

class CityBlock : public QGraphicsItem
{
public:
    enum Kind { Park, SmallBuilding, Hospital, Hall, Building, Tower,
        LShapedBlock, LShapedBlockPlusSmallBlock, TwoBlocks,
        BlockPlusTwoSmallBlocks };
    CityBlock(Kind kind);
    QRectF boundingRect() const;
    void paint(QPainter *painter,
        const QStyleOptionGraphicsItem *option, QWidget *widget);
private:
    int kind;
    QColor color;
    QPainterPath shape;
};

```

Городской квартал имеет такие параметры, как тип (kind), цвет (color) и форму (shape). Поскольку кварталы нельзя выделять, мы не будем переопределять функцию shape(), как мы это делали для класса Node в предыдущем примере.

```

CityBlock::CityBlock(Kind kind)
{
    this->kind = kind;
    int green = 96 + (std::rand() % 64);
    int red = 16 + green + (std::rand() % 64);
    int blue = 16 + (std::rand() % green);
    color = QColor(red, green, blue);
    if (kind == Park) {
        color = QColor(192 + (std::rand() % 32), 255,
                      192 + (std::rand() % 16));
        shape.addRect(boundingRect());
    } else if (kind == SmallBuilding) {
        ...
    } else if (kind == BlockPlusTwoSmallBlocks) {
        int w1 = (std::rand() % 10) + 8;
        int h1 = (std::rand() % 28) + 8;
        int w2 = (std::rand() % 10) + 8;
        int h2 = (std::rand() % 10) + 8;
        int w3 = (std::rand() % 6) + 8;
        int h3 = (std::rand() % 6) + 8;
        int y = (std::rand() % 4) - 16;
        shape.addRect(QRectF(-16, -16, w1, h1));
        shape.addRect(QRectF(-16 + w1 + 4, y, w2, h2));
        shape.addRect(QRectF(-16 + w1 + 4,
                              y + h2 + 4 + (std::rand() % 4), w3, h3));
    }
}

```

Конструктор задает случайный цвет и генерирует подходящий объект QPainterPath в зависимости от типа квартала, который представляет узел.

```

QRectF CityBlock::boundingRect() const
{
    return QRectF(-20, -20, 40, 40);
}

```

Каждый квартал занимает квадрат 40×40 , и центр имеет координаты (0, 0).

```

void CityBlock::paint(QPainter *painter,
                     const QStyleOptionGraphicsItem *option,
                     QWidget * /* widget */)
{
    if (option->levelOfDetail < 4.0) {
        painter->fillPath(shape, color);
    } else {
        QLinearGradient gradient(QPoint(-20, -20), QPoint(+20, +20));
        int coeff = 105 + int(std::log(option->levelOfDetail - 4.0));
        gradient.setColorAt(0.0, color.lighter(coeff));
        gradient.setColorAt(1.0, color.darker(coeff));
        painter->fillPath(shape, gradient);
    }
}

```

В функции `paint()` мы рисуем фигуру, используя заданное значение `QPainter`. Мы выделяем два случая.

- Если фактор масштабирования меньше 4.0, мы заполняем фигуру сплошным цветом.
- Если фактор масштабирования равен 4.0 или больше, мы с помощью `QLinearGradient` заполняем фигуру так, чтобы получить небольшой эффект освещения.

Член `levelOfDetail` класса `QStyleOptionGraphicsItem` хранит значение с плавающей точкой, определяющее фактор масштабирования. Значение 1.0 означает, что сцена будет представлена в ее нормальном размере, значение 0.5 означает, что сцена будет вдвое меньше нормального размера, а значение 2.5 означает, что сцена будет отображаться увеличенной в два с половиной раза. Такая информация об «уровне детализации» позволяет нам использовать более быстрые алгоритмы прорисовки в том случае, если сцена увеличена слишком сильно, чтобы отображать на ней все детали.

Класс графических элементов `CityBlock` работает прекрасно, но, т. к. элемент масштабируются при изменении размера сцены, возникает вопрос о том, что происходит с элементами, которые выводят текст. Архитектура графических представлений предлагает общее решение этой проблемы при помощи флага `ItemIgnoresTransformations`. Мы используем этот флаг в классе `Annotation`:

```
class Annotation : public QGraphicsItem
{
public:
    Annotation(const QString &text, bool major = false);
    void setText(const QString &text);
    QString text() const;
    QRectF boundingRect() const;
    void paint(QPainter *painter,
               const QStyleOptionGraphicsItem *option, QWidget *widget);

private:
    QFont font;
    QString str;
    bool major;
    double threshold;
    int y;
};
```

Конструктор принимает текст и булево значение флага с именем `major`, который указывает, является ли подпись главной или второстепенной. Это будет влиять на размер шрифта.

```
Annotation::Annotation(const QString &text, bool major)
{
    font = qApp->font();
    font.setBold(true);
    if (major) {
        font.setPointSize(font.pointSize() + 2);
```



```

        font.setStretch(QFont::SemiExpanded);
    }
    if (major) {
        threshold = 0.01 * (40 + (std::rand() % 40));
    } else {
        threshold = 0.01 * (100 + (std::rand() % 100));
    }
    str = text;
    this->major = major;
    y = 20 - (std::rand() % 40);
    setZValue(1000);
    setFlag(ItemIgnoresTransformations, true);
}

```

В конструкторе мы начинаем с того, что задаем шрифт большего размера и толщины для главной подписи, которая, вероятно, будет относиться к важному зданию или ориентиру. Значение порога для прекращения отображения подписи вычисляется на псевдослучайной основе с более низким порогом для главных подписей, чтобы менее важные подписи исчезали первыми при увеличении поля зрения.

Для z задается значение 1000, чтобы все подписи располагались поверх остальных элементов, и используется флаг `ItemIgnoresTransformations`, который гарантирует, что подпись не будет изменять размер, независимо от степени увеличения сцены.

```

void Annotation::setText(const QString &text)
{
    prepareGeometryChange();
    str = text;
    update();
}

```

Если текст аннотации изменяется, он может стать длиннее или короче предыдущего, так что мы должны уведомить архитектуру графических представлений о том, что геометрия элемента может измениться.

```

QRectF Annotation::boundingRect() const
{
    QFontMetricsF metrics(font);
    QRectF rect = metrics.boundingRect(str);
    rect.moveCenter(QPointF(0, y));
    rect.adjust(-4, 0, +4, 0);
    return rect;
}

```

Мы получаем размер шрифта подписи и используем его для вычисления размеров граничного прямоугольника. Затем мы перемещаем центральную точку прямоугольника на смещение подписи по вертикальной оси и делаем прямоугольник немного шире. Дополнительные пиксели, добавляемые слева и справа от граничного прямоугольника, придадут тексту небольшие поля по краям.

```

void Annotation::paint(QPainter *painter,
                      const QStyleOptionGraphicsItem *option,
                      QWidget * /* widget */)
{
    if (option->levelOfDetail <= threshold)
        return;
    painter->setFont(font);
    QRectF rect = boundingRect();
    int alpha = int(30 * std::log(option->levelOfDetail));
    if (alpha >= 32)
        painter->fillRect(rect, QColor(255, 255, 255, qMin(alpha, 63)));
    painter->setPen(Qt::white);
    painter->drawText(rect.translated(+1, +1), str,
                     QTextOption(Qt::AlignCenter));
    painter->setPen(Qt::blue);
    painter->drawText(rect, str, QTextOption(Qt::AlignCenter));
}

```

Если увеличение ниже порога подписей, мы не рисуем подписи совсем. А если сцена увеличивается в достаточной мере, мы начинаем с рисования полупрозрачного белого прямоугольника. Это помогает увидеть текст, если он рисуется поверх темного квартала.

Мы прорисовываем текст дважды, один раз белым цветом, а другой раз – синим. Белый текст смещается на один пиксель по горизонтали и вертикали, создавая эффект тени, что облегчает чтение текста.

Рассмотрев, как создаются кварталы и подписи, мы можем переходить к последнему аспекту приложения Cityscape – пользовательскому подклассу класса QGraphicsView.

```

class CityView : public QGraphicsView
{
    Q_OBJECT
public:
    CityView(QWidget *parent = 0);
protected:
    void wheelEvent(QWheelEvent *event);
};

```

По умолчанию класс QGraphicsView предлагает линейки прокрутки, которые появляются автоматически по мере необходимости, но не предоставляет средств изменения масштаба сцены, для просмотра которой этот класс используется. Поэтому мы создали небольшой подкласс CityView, дающий возможность пользователю увеличивать и уменьшать масштаб при помощи колесика мыши.

```

CityView::CityView(QWidget *parent)
    : QGraphicsView(parent)
{
    setDragMode(ScrollHandDrag);
}

```

Необходимо только указать режим перетаскивания, чтобы поддерживать скроллинг перетаскиванием.

```
void CityView::wheelEvent(QWheelEvent *event)
{
    double numDegrees = -event->delta() / 8.0;
    double numSteps = numDegrees / 15.0;
    double factor = std::pow(1.125, numSteps);
    scale(factor, factor);
}
```

Когда пользователь крутит колесико мыши, генерируются события `wheelEvent`. Мы просто должны вычислить соответствующий фактор масштабирования и вызвать функцию `QGraphicsView::scale()`. Математическая формула чуть сложнее, но, по сути, мы масштабируем сцену на фактор 1.125 в большую или меньшую сторону на каждый шаг колесика мыши.

На этом заканчивается рассмотрение двух примеров графических представлений. Архитектура графических представлений в Qt очень богатая, так что помните, что возможностей здесь гораздо больше, чем мы здесь описали. Существует поддержка функции перетаскивания, графические элементы могут иметь всплывающие подсказки и пользовательские курсоры. Можно различными способами получать анимационные эффекты, например, путем связывания объектов `QGraphicsItemAnimations` с соответствующими элементами, а также средствами класса `QTimeline`. Кроме того, можно создавать анимацию с помощью создания пользовательских подклассов графических элементов, происходящих от `QObject` (с применением множественного наследования), и переопределяющих функцию `QObject::timerEvent()`.

Вывод на печатающее устройство

Вывод на печатающее устройство в Qt подобен рисованию по `QWidget`, `QPixmap` или `QImage`. Порядок действий при этом будет следующим.

1. Создайте в качестве устройства рисования объект `QPrinter`.
2. Выведите на экран диалоговое окно печати `QPrintDialog`, позволяя пользователю выбрать печатающее устройство и установить некоторые параметры печати.
3. Создайте объект `QPainter` для работы с `QPrinter`.
4. Нарисуйте страницу, используя `QPainter`.
5. Вызовите функцию `QPrinter::newPage()` для перехода на следующую страницу.
6. Повторяйте пункты 4 и 5 до тех пор, пока не будут распечатаны все страницы.

В операционных системах Windows и Mac OS X `QPrinter` использует системные драйверы принтеров. В системе Unix он формирует файл PostScript и передает его `lp` или `lpr` (или другой программе, установленной функцией `QPrinter::setPrintProgram()`). `QPrinter` может также использоваться для генерации файлов PDF, если вызвать `setOutputFormat(QPrinter::PdfFormat)`.

Давайте начнем с рассмотрения какого-нибудь простого примера по распечатке одной страницы. Первый пример, показанный на рис. 8.18, распечатывает объект `QImage`:

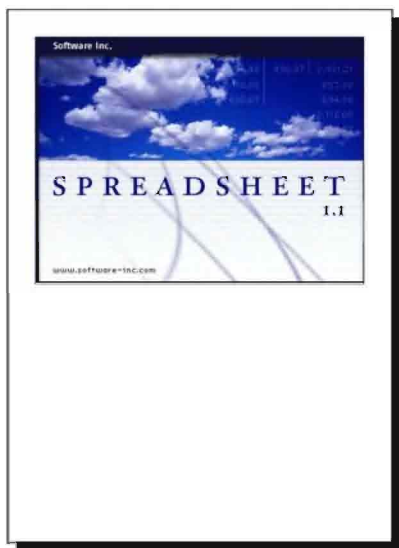


Рис. 8.18. Вывод на печатающее устройство объекта QImage

```
void PrintWindow::printImage(const QImage &image)
{
    QPrintDialog printDialog(&printer, this);
    if (printDialog.exec()) {
        QPainter painter(&printer);
        QRect rect = painter.viewport();
        QSize size = image.size();
        size.scale(rect.size(), Qt::KeepAspectRatio);
        painter.setViewport(rect.x(), rect.y(),
                           size.width(), size.height());
        painter.setWindow(image.rect());
        painter.drawImage(0, 0, image);
    }
}
```

Мы предполагаем, что класс `PrintWindow` имеет переменную-член `printer` типа `QPrinter`. Мы могли бы просто поместить `QPrinter` в стек в функции `printImage()`, но тогда не сохранялись бы настройки пользователя при переходе от одной печати к другой.

Мы создаем объект `QPrintDialog` и вызываем функцию `exec()` для вывода на экран диалогового окна печати. Оно возвращает `true`, если пользователь нажал кнопку `ОК`; в противном случае оно возвращает `false`. После вызова функции `exec()`, объект `QPrinter` готов для использования. (Можно также печатать, не используя `QPrintDialog`, а напрямую вызывая функции-члены класса `QPrinter` для подготовки печати).

Затем мы создаем `QPainter` для рисования на `QPrinter`. Мы устанавливаем окно на прямоугольник изображения и область отображения на прямоугольник с тем же соотношением сторон, и мы рисуем изображение в позиции `(0, 0)`.

По умолчанию окно `QPrinter` инициализируется таким образом, что разрешающая способность принтера будет аналогична разрешающей способности экрана (обычно она составляет примерно от 72 до 100 точек на дюйм), позволяя легко использовать для печати программный код по рисованию виджета. Здесь это не имеет значения, поскольку мы сами задали параметры нашего окна.

В данном примере мы решили распечатать изображение, но столь же просто можно распечатать и графические сцены. Чтобы распечатать сцену полностью, можно вызвать функцию `QGraphicsScene::render()` или `QGraphicsView::render()`, передав в качестве первого параметра объект `QPrinter`. Если мы хотим распечатать только часть сцены, мы можем использовать дополнительные аргументы функции `render()`, указав прямоугольник для рисования (место на странице, где должна быть отображена сцена) и исходный прямоугольник (какую часть сцены нужно прорисовывать).

Вывод на печатающее устройство элементов, занимающих не более одной страницы, выполняется достаточно просто, но во многих приложениях приходится печатать несколько страниц. В таких случаях мы должны сначала нарисовать одну страницу и затем вызвать функцию `newPage()` для перехода на следующую страницу. Здесь возникает проблема определения того количества информации, которое будет печататься на одной странице. Существует два подхода при обработке многостраничных документов в Qt.

- Мы можем преобразовать наши данные в формат HTML и затем воспроизвести их с применением класса `QTextDocument`, процессора форматированного текста Qt.
- Мы можем выполнить рисование и разбивку на страницы вручную.

Мы рассмотрим по очереди оба подхода. В качестве примера мы распечатаем цветочный справочник: список названий цветов с текстовым описанием. Каждый элемент этого справочника представляется строкой формата «название: описание», например:

`Miltonopsis santanae`: Самый опасный вид орхидеи.

Поскольку данные каждого цветка представлены одной строкой, мы можем представить цветочный справочник при помощи одного объекта `QStringList`. Ниже приводится функция печати цветочного справочника, использующая процессор форматированного текста Qt:

```
void PrintWindow::printFlowerGuide(const QStringList &entries)
{
    QString html;

    foreach (QString entry, entries) {
        QStringList fields = entry.split(": ");
        QString title = Qt::escape(fields[0]);
        QString body = Qt::escape(fields[1]);

        html += "<table width=\"100%\" border=1 cellspacing=0>\n"
               "<tr><td bgcolor=\"lightgray\"><font size=\"+1\">"
               "<b><1>" + title + "</i></b></font>\n<tr><td>" + body
               + "\n</table>\n<br>\n";
    }
    printHtml(html);
}
```



```

{
    QStringList currentPage;
    int pageHeight = painter->window().height() - 2 * LargeGap;
    int y = 0;

    foreach (QString entry, entries) {
        int height = entryHeight(painter, entry);
        if (y + height > pageHeight && !currentPage.empty()) {
            pages->append(currentPage);
            currentPage.clear();
            y = 0;
        }
        currentPage.append(entry);
        y += height + MediumGap;
    }
    if (!currentPage.empty())
        pages->append(currentPage);
}

```

Функция `paginate()` распределяет элементы справочника цветов по страницам. Ее работа основана на применении функции `entryHeight()`, рассчитывающей высоту каждого элемента. Она также учитывает наличие сверху и снизу страницы полей с размером `LargeGap`.

Мы выполняем цикл по элементам и добавляем их в конец текущей страницы до тех пор, пока не окажется, что элемент не вмещается на страницу; затем мы добавляем текущую страницу в конец списка `pages` и начинаем формировать новую страницу.

```

int PrintWindow::entryHeight(QPainter *painter, const QString &entry)
{
    QStringList fields = entry.split(": ");
    QString title = fields[0];
    QString body = fields[1];

    int textWidth = painter->window().width() - 2 * SmallGap;
    int maxHeight = painter->window().height();
    painter->setFont(titleFont);
    QRect titleRect = painter->boundingRect(0, 0, textWidth, maxHeight,
                                           Qt::TextWordWrap, title);

    painter->setFont(bodyFont);
    QRect bodyRect = painter->boundingRect(0, 0, textWidth, maxHeight,
                                           Qt::TextWordWrap, body);

    return titleRect.height() + bodyRect.height() + 4 * SmallGap;
}

```

Функция `entryHeight()` использует `QPainter::boundingRect()` для вычисления размера области, занимаемой одним элементом по вертикали. На рис. 8.20 показана компоновка элементов одного цветка на странице и проиллюстрирован смысл констант `SmallGap` и `MediumGap`.

```

void PrintWindow::printPages(QPainter *painter,

```

```

const QList<QStringList> &pages)
{
    int firstPage = printer.fromPage() - 1;
    if (firstPage >= pages.size())
        return;
    if (firstPage == -1)
        firstPage = 0;

    int lastPage = printer.toPage() - 1;
    if (lastPage == -1 || lastPage >= pages.size())
        lastPage = pages.size() - 1;

    int numPages = lastPage - firstPage + 1;
    for (int i = 0; i < printer.numCopies(); ++i) {
        for (int j = 0; j < numPages; ++j) {
            if (i != 0 || j != 0)
                printer.newPage();

            int index;
            if (printer.pageOrder() == QPrinter::FirstPageFirst) {
                index = firstPage + j;
            } else {
                index = lastPage - j;
            }
            printPage(painter, pages[index], index + 1);
        }
    }
}

```

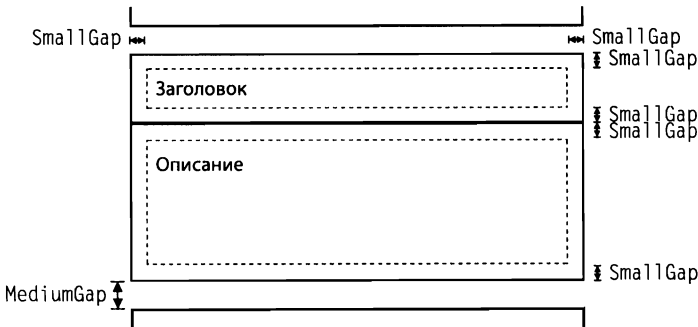


Рис. 8.20. Компоновка элементов справочника цветов на странице

Функция `printPages()` предназначена для печати каждой страницы функцией `printPage()` с обеспечением правильного числа и правильной последовательности вызовов последней. Результат показан на рис. 8.21. Применяя `QPrintDialog`, пользователь может запросить распечатку нескольких копий, указать диапазон страниц или запросить распечатку страниц в обратной последовательности. Мы сами должны включать или отключать эти опции, используя функцию `QPrintDialog::setEnabledOptions()`.

Мы начинаем с определения диапазона печати. Функции `QPrinter::fromPage()` и `toPage()` возвращают заданные пользователем номера страниц или 0, если диапазон не указан. Мы вычитаем 1, потому что наш список страниц `pages` нумеруется с нуля, и устанавливаем переменные `firstPage` и `lastPage` (первая и последняя страницы) на охват всех страниц, если диапазон не задан пользователем.

Затем мы печатаем каждую страницу. Внешний цикл `for` определяется количеством копий, запрошенных пользователем. Большинство драйверов принтеров поддерживают печать нескольких копий, поэтому для них функция `QPrinter::numCopies()` всегда возвращает 1.

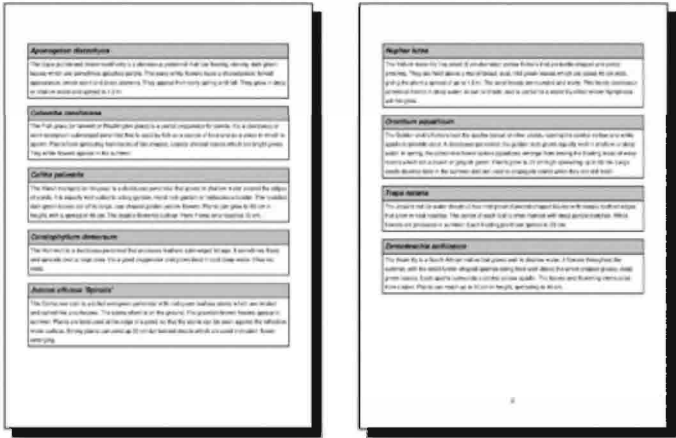


Рис. 8.21. Распечатка справочника цветов с использованием `QPainter`

Если драйвер принтера не может печатать несколько копий, `numCopies()` возвращает количество копий, запрошенное пользователем, и за печать этого количества копий отвечает приложение. (В примере с `QImage`, приведенном ранее в данном разделе, мы для простоты проигнорировали `numCopies()`). Внутренний цикл `for` выполняется по всем страницам. Если страница не первая, мы вызываем `newPage()`, чтобы сбросить на печатающее устройство старую страницу и начать рисование новой страницы. Мы вызываем `printPage()` для распечатки каждой страницы.

```
void PrintWindow::printPage(QPainter *painter,
                           const QStringList &entries, int pageNumber)
{
    painter->save();
    painter->translate(0, LargeGap);
    foreach (QString entry, entries) {
        QStringList fields = entry.split(": ");
        QString title = fields[0];
        QString body = fields[1];
        printBox(painter, title, titleFont, Qt::lightGray);
        printBox(painter, body, bodyFont, Qt::white);
        painter->translate(0, MediumGap);
    }
    painter->restore();
}
```

```

painter->setFont(footerFont);
painter->drawText(painter->window(),
                  Qt::AlignHCenter | Qt::AlignBottom,
                  QString::number(pageNumber));
}

```

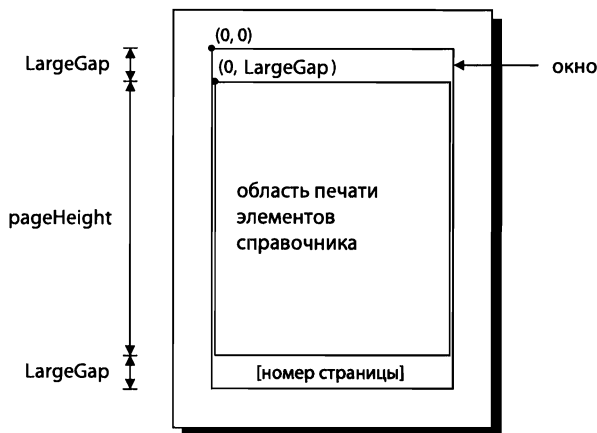


Рис. 8.22. Компоновка страницы справочника по цветам

Функция `printPage()` обрабатывает в цикле все элементы справочника цветов и печатает их при помощи двух вызовов функции `printBox()`: один для заголовка (название цветка) и другой для «тела» (описание цветка). Она также отображает номер страницы внизу по центру страницы.

```

void PrintWindow::printBox(QPainter *painter, const QString &str,
                           const QFont &font, const QBrush &brush)
{
    painter->setFont(font);

    int boxWidth = painter->window().width();
    int textWidth = boxWidth - 2 * SmallGap;
    int maxHeight = painter->window().height();
    QRect textRect = painter->boundingRect(SmallGap, SmallGap,
                                           textWidth, maxHeight,
                                           Qt::TextWordWrap, str);
    int boxHeight = textRect.height() + 2 * SmallGap;

    painter->setPen(QPen(Qt::black, 2.0, Qt::SolidLine));
    painter->setBrush(brush);
    painter->drawRect(0, 0, boxWidth, boxHeight);
    painter->drawText(textRect, Qt::TextWordWrap, str);
    painter->translate(0, boxHeight);
}

```

Функция `printBox()` вычерчивает контур блока, затем отображает текст внутри него.



- *Обеспечение поддержки технологии «drag-and-drop»*
- *Поддержка пользовательских типов переносимых объектов*
- *Работа с буфером обмена*

Глава 9. Технология «drag-and-drop»

Технология «drag-and-drop» является современным и интуитивным способом передачи информации внутри одного приложения или между разными приложениями. Она часто является дополнением к операциям с буфером обмена по перемещению и копированию данных.

В данной главе мы увидим, как можно добавить в приложение Qt возможность поддержки технологии «drag-and-drop» и как обрабатывать пользовательские форматы. Затем мы используем программный код этой технологии для реализации операций с буфером обмена. Такое повторное использование данного программного кода возможно благодаря тому, что оба механизма основаны на применении одного класса `QMimeData` – базового класса, обеспечивающего представление данных в нескольких форматах.

Обеспечение поддержки технологии «drag-and-drop»

Технология «drag-and-drop» состоит из двух действий: перетаскивание «захваченных» объектов и их «освобождение». Виджеты в Qt могут использоваться в качестве переносимых объектов, в качестве места отпускания этих объектов или в обоих качествах.

В нашем первом примере мы показываем, как приложение Qt принимает объект, перенесенный из другого приложения. Приложение Qt представляет собой главное окно, использующее текстовый редактор `QTextEdit` в качестве центрального виджета. Когда пользователь переносит текстовый файл с рабочего стола компьютера или из проводника файловой системы и оставляет его в окне этого приложения, оно загружает файл в `QTextEdit`.

Ниже приводится пример определения класса `MainWindow`:

```
class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow();
protected:
```

```

void dragEnterEvent(QDragEnterEvent *event);
void dropEvent(QDropEvent *event);

private:
    bool readFile(const QString &fileName);

    QTextEdit *textEdit;
};

```

Класс `MainWindow` переопределяет функции `dragEnterEvent()` и `dropEvent()` класса `QWidget`. Поскольку целью примера является демонстрация механизма «drag-and-drop», большая часть функциональности класса главного окна здесь не рассматривается.

```

MainWindow::MainWindow()
{
    textEdit = new QTextEdit;
    setCentralWidget(textEdit);

    textEdit->setAcceptDrops(false);
    setAcceptDrops(true);

    setWindowTitle(tr("Text Editor"));
}

```

В конструкторе мы создаем `QTextEdit` и назначаем его в качестве центрального виджета. По умолчанию `QTextEdit` принимает переносимые текстовые объекты из других приложений, и если пользователь отпускает на этом виджете файл, имя этого файла будет вставлено в текст. Поскольку события отпускания объектов передаются от дочерних виджетов к родительским, отключая возможность отпускать переносимый объект в области отображения `QTextEdit` и включая ее для главного окна, мы получаем события отпускания объектов в `MainWindow` для всего главного окна.

```

void MainWindow::dragEnterEvent(QDragEnterEvent *event)
{
    if (event->mimeTypeData()->hasFormat("text/uri-list"))
        event->acceptProposedAction();
}

```

Функция `dragEnterEvent()` вызывается всякий раз, когда пользователь переносит объект на какой-нибудь виджет. Если мы вызываем функцию `acceptProposedAction()` при обработке этого события, мы указываем, что пользователь может отпустить переносимый объект в данном виджете. По умолчанию виджет не смог бы принять переносимый объект. Qt автоматически изменяет форму курсора для уведомления пользователя о возможности или невозможности приема объекта виджетом.

Здесь мы хотим позволить пользователю переносить файлы, но не более того. Для этого мы проверяем MIME-тип переносимого объекта. MIME-тип `text/uri-list` используется для хранения списка унифицированных идентификаторов

ресурсов (URI-uniform resource identifier), в качестве которых могут выступать имена файлов, адреса URL (например, адресные пути HTTP и FTP) или идентификаторы других глобальных ресурсов. Стандартные типы MIME определяются Агентством по выделению имен и уникальных параметров протоколов сети Интернет (Internet Assigned Numbers Authority – IANA). Они состоят из типа и подтипа, разделенных слешем. Буфер обмена и механизм «drag-and-drop» используют типы MIME для идентификации различных типов данных. Официальный список MIME-типов доступен по адресу <http://www.iana.org/assignments/media-types/>.

```
void MainWindow::dropEvent(QDropEvent *event)
{
    QList<QUrl> urls = event->mimeTypeData()->urls();
    if (urls.isEmpty())
        return;

    QString fileName = urls.first().toLocalFile();

    if (fileName.isEmpty())
        return;

    if (readFile(fileName))
        setWindowTitle(tr("%1 - %2").arg(fileName)
                        .arg(tr("Drag File")));
}
```

Функция `dropEvent()` вызывается, когда пользователь отпускает объект на виджете. Мы вызываем функцию `QMimeTypeData::urls()` для получения списка адресов `QUrl`. Обычно пользователи переносят одновременно только один файл, но возможен также перенос сразу нескольких выделенных файлов. Если оказывается несколько URL или полученный URL оказывается не локальным, мы немедленно возвращаем управление.

`QWidget` содержит также функции `dragMoveEvent()` и `dragLeaveEvent()`, но для большинства приложений не потребуется их переопределять.

Второй пример показывает, как следует инициировать перетаскивание объекта и принимать его после отпускания. Мы создадим подкласс `QListWidget`, который будет поддерживать механизм «drag-and-drop» и который будет входить в приложение *Project Chooser* (составитель проектов), показанное на рис. 9.1.

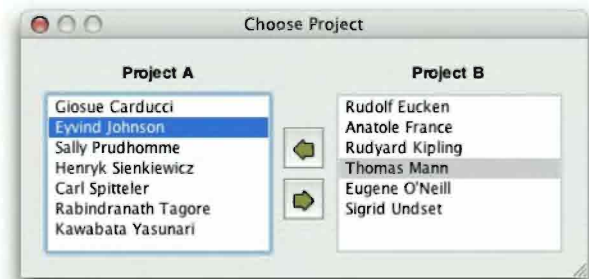


Рис. 9.1. Приложение *Project Chooser*

Приложение Project Chooser предоставляет пользователю два виджета со списками имен людей. Каждый список представляет проект. Пользователь может с помощью механизма «drag-and-drop» перевести человека из одного проекта в другой.

Программный код по обеспечению механизма «drag-and-drop» находится в подклассе QListWidget. Ниже приводится определение класса:

```
class ProjectListWidget : public QListWidget
{
    Q_OBJECT

public:
    ProjectListWidget(QWidget *parent = 0);

protected:
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void dragEnterEvent(QDragEnterEvent *event);
    void dragMoveEvent(QDragMoveEvent *event);
    void dropEvent(QDropEvent *event);

private:
    void performDrag();

    QPoint startPos;
};
```

ProjectListWidget переопределяет пять обработчиков событий, которые объявлены в QListWidget.

```
ProjectListWidget::ProjectListWidget(QWidget *parent)
    : QListWidget(parent)
{
    setAcceptDrops(true);
}
```

В конструкторе мы обеспечиваем возможность приема переносимого объекта в виджете со списком.

```
void ProjectListWidget::mousePressEvent(QMouseEvent *event)
{
    if (event->button() == Qt::LeftButton)
        startPos = event->pos();
    QListWidget::mousePressEvent(event);
}
```

Когда пользователь нажимает левую кнопку мыши, мы сохраняем позицию мыши в закрытой переменной startPos. Мы вызываем определенную в классе QListWidget функцию mousePressEvent() для обеспечения обработки в QListWidget обычным образом события нажатия кнопки мыши.

```
void ProjectListWidget::mouseMoveEvent(QMouseEvent *event)
{
    if (event->buttons() & Qt::LeftButton) {
```

```

        int distance = (event->pos() - startPos).manhattanLength();
        if (distance >= QApplication::startDragDistance())
            performDrag();
    }
    QListWidget::mouseMoveEvent(event);
}

```

Действие, при котором пользователь перемещает курсор мыши и одновременно держит нажатой левую кнопку, мы рассматриваем как начало перетаскивания объекта. Мы вычисляем расстояние между текущей позицией мыши и позицией нажатия левой кнопки мыши. «Манхэттенская длина» представляет собой быстро вычисляемую аппроксимацию длины вектора от его начала. Если это расстояние равно или превышает рекомендованное в `QApplication` расстояние для регистрации начала перетаскивания (обычно четыре пикселя), мы вызываем закрытую функцию `performDrag()` для запуска процесса перетаскивания объекта. Это предотвращает инициирование процесса перетаскивания из-за дрожания руки пользователя.

```

void ProjectListWidget::performDrag()
{
    QListWidgetItem *item = currentItem();
    if (item) {
        QMimeData *mimeType = new QMimeData;
        mimeType->setText(item->text());
        QDrag *drag = new QDrag(this);
        drag->setMimeData(mimeType);
        drag->setPixmap(QPixmap(":/images/person.png"));
        if (drag->exec(Qt::MoveAction) == Qt::MoveAction)
            delete item;
    }
}

```

В функции `performDrag()` мы создаем объект типа `QDrag` с указанием `this` в качестве родительского элемента. Объект `QDrag` хранит данные в объекте `QMimeData`. В нашем примере мы обеспечиваем данные типа `text/plain`, используя функцию `QMimeData::setText()`. Класс `QMimeData` содержит несколько функций, предназначенных для обработки наиболее распространенных типов объектов переноса (изображений, адресов URL, цветов и т. д.); он может обрабатывать произвольные типы MIME, представленные массивами `QByteArray`. Вызов `QDrag::setPixmap()` задает пиктограмму, которая следует за курсором в процессе перетаскивания объекта.

Вызов функции `QDrag::exec()` запускает операцию перетаскивания объекта и ждет, пока пользователь не отпустит перетаскиваемый объект или не отменит перетаскивание. В аргументе этой функции задается перечень поддерживаемых «операций перетаскивания» (`Qt::CopyAction`, `Qt::MoveAction` и `Qt::LinkAction`); она возвращает ту операцию перетаскивания, которая была выполнена (или `Qt::IgnoreAction`, если не было выполнено никакой операции). Тип выполняемой операции зависит от того, какие операции допускаются исходным виджетом, какие операции поддерживает целевой виджет и какие клавиши-модификаторы нажаты в момент отпуска переносимого объекта. После вызова этой функции `Qt` становится владельцем переносимого объекта и удалит его, когда он станет ненужным.

```
void ProjectListWidget::dragEnterEvent(QDragEnterEvent *event)
{
    ProjectListWidget *source =
        qobject_cast<ProjectListWidget *>(event->source());
    if (source && source != this) {
        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}
```

Виджет `ProjectListWidget` не только иницирует перенос объектов, но также является местом приема таких объектов, если они приходят от другого виджета `ProjectListWidget` того же самого приложения. `QDragEnterEvent::source()` возвращает указатель на виджет, который иницирует перенос, если этот виджет принадлежит тому же самому приложению; в противном случае он возвращает нулевой указатель. Мы используем `qobject_cast<T>()`, чтобы убедиться в иницировании переноса виджетом `ProjectListWidget`. Если все верно, мы указываем Qt на нашу готовность восприятия данного действия как переноса.

```
void ProjectListWidget::dragMoveEvent(QDragMoveEvent *event)
{
    ProjectListWidget *source =
        qobject_cast<ProjectListWidget *>(event->source());
    if (source && source != this) {
        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}
```

То, что мы делаем в программном коде функции `dragMoveEvent()` идентично тому, что мы делали в функции `dragEnterEvent()`. Он необходим, потому что нам приходится переопределять реализацию этой функции в классе `QListWidget` (в действительности, в классе `QAbstractItemView`).

```
void ProjectListWidget::dropEvent(QDropEvent *event)
{
    ProjectListWidget *source =
        qobject_cast<ProjectListWidget *>(event->source());
    if (source && source != this) {
        addItem(event->mimeTypeData()->text());
        event->setDropAction(Qt::MoveAction);
        event->accept();
    }
}
```

В `DropEvent()` мы используем функцию `QMimeData::text()` для получения перенесенного текста и создаем элемент с этим текстом. Нам также необходимо чтобы данное событие воспринималось как «операция перетаскивания», чтобы указать исходному виджету на то, что он может теперь удалить первоначальную версию перенесенного элемента.

«Drag and drop» – мощный механизм передачи данных между приложениями. Однако в некоторых случаях его можно реализовать, не используя предусмотренные в Qt средства механизма «drag and drop». Если нам требуется переносить данные внутри одного виджета некоторого приложения, во многих случаях мы можем просто переопределить функции `mousePressEvent()` и `mouseReleaseEvent()`.

Поддержка пользовательских типов переносимых объектов

До сих пор в представленных примерах мы полагались на поддержку `QMimeData` распространенных типов MIME. Так мы вызывали `QMimeData::setText()` для создания объекта переноса текста и использовали `QMimeData::urls()` для получения содержимого объекта переноса типа `text/uri-list`. Если мы хотим перетаскивать обычный текст, текст в формате HTML, изображения, адреса URL или цвета, мы можем спокойно использовать класс `QMimeData`. Но если мы хотим перетаскивать пользовательские данные, необходимо сделать выбор между следующими альтернативами.

1. Мы можем обеспечить произвольные данные в виде массива `QByteArray`, используя функцию `QMimeData::setData()`, и извлекать их позже, используя функцию `QMimeData::data()`.
2. Мы можем создать подкласс `QMimeData` и переопределить функции `formats()` и `retrieveData()` для обработки наших пользовательских типов данных.
3. Для выполнения операций механизма «drag and drop» в рамках одного приложения, мы можем создать подкласс `QMimeData` и хранить данные в любых структурах данных.

Первый подход не требует никаких подклассов, но имеет некоторые недостатки: нам необходимо преобразовать наши структуры данных в тип `QByteArray`, даже если переносимый объект не принимается, и если требуется обеспечить несколько MIME-типов, чтобы можно было хорошо взаимодействовать с самыми разными приложениями, нам придется сохранять несколько копий данных (по одной на каждый тип MIME). Если данные имеют большой размер, это может слишком замедлить работу приложения. При использовании второго и третьего подходов можно избежать или свести к минимуму эти проблемы. В этом случае мы получаем полное управление и можем использовать эти два подхода совместно.

Для демонстрации этих подходов мы покажем, как можно добавить возможности технологии «drag and drop» в виджет `QTableWidget`. Будет поддерживаться перенос следующих типов MIME: `text/plain`, `text/html` и `text/csv`. При применении первого подхода инициирование переноса выглядит следующим образом:

```
void MyTableWidget::mouseMoveEvent(QMouseEvent *event)
{
    if (event->buttons() & Qt::LeftButton) {
        int distance = (event->pos() - startPos).manhattanLength();
        if (distance >= QApplication::startDragDistance())
            performDrag();
    }
    QTableWidget::mouseMoveEvent(event);
}
```

```

void MyTableWidget::performDrag()
{
    QString plainText = selectionAsPlainText();
    if (plainText.isEmpty())
        return;

    QMimeData *mimeTypeData = new QMimeData;
    mimeTypeData->setText(plainText);
    mimeTypeData->setHtml(toHtml(plainText));
    mimeTypeData->setData("text/csv", toCsv(plainText).toUtf8());

    QDrag *drag = new QDrag(this);
    drag->setMimeData(mimeTypeData);
    if (drag->exec(Qt::CopyAction | Qt::MoveAction) == Qt::MoveAction)
        deleteSelection();
}

```

Закрывающая функция `performDrag()` вызывается из `mouseMoveEvent()` для инициирования переноса выделенной прямоугольной области. Мы устанавливаем типы MIME `text/plain` и `text/html`, используя функции `setText()` и `setHtml()`, а тип `text/csv` мы устанавливаем функцией `setData()`, которая принимает произвольный тип MIME и массив `QByteArray`. Программный код для функции `selectionAsString()` более или менее совпадает с кодом функции `Spreadsheet::copy()`, рассмотренной в главе 4.

```

QString MyTableWidget::toCsv(const QString &plainText)
{
    QString result = plainText;
    result.replace("\\", "\\");
    result.replace("\"", "\\");
    result.replace("\t", "\\t");
    result.replace("\n", "\\n");
    result.prepend("\\");
    result.append("\\");
    return result;
}

QString MyTableWidget::toHtml(const QString &plainText)
{
    QString result = Qt::escape(plainText);
    result.replace("\t", "<td>");
    result.replace("\n", "<n<tr><td>");
    result.prepend("<table><n<tr><td>");
    result.append("<n</table>");
    return result;
}

```

Функции `toCsv()` и `toHtml()` преобразуют строку со знаками табуляции и конца строки в формат CSV (comma-separated values – значения, разделенные запятыми) и HTML, соответственно. Например, данные

```
Red Green Blue
Cyan Yellow Magenta
```

преобразуются в

```
"Red", "Green", "Blue"
"Cyan", "Yellow", "Magenta"
```

или в

```
<table>
<tr><td>Red<td>Green<td>Blue
<tr><td>Cyan<td>Yellow<td>Magenta
</table>
```

Преобразование выполняется самым простым из возможных способов с применением функции `QString::replace()`. Для удаления специальных символов формата HTML мы используем функцию `Qt::escape()`.

```
void MyTableWidget::dropEvent(QDropEvent *event)
{
    if (event->mimeTypeData()->hasFormat("text/csv")) {
        QByteArray csvData = event->mimeTypeData()->data("text/csv");
        QString csvText = QString::fromUtf8(csvData);
        ...
        event->acceptProposedAction();
    } else if (event->mimeTypeData()->hasFormat("text/plain")) {
        QString plainText = event->mimeTypeData()->text();
        ...
        event->acceptProposedAction();
    }
}
```

Хотя мы предоставляем данные в трех разных форматах, мы принимаем в `dropEvent()` только два из них. Если пользователь переносит ячейки из таблицы `QTableWidget` в редактор HTML, нам нужно, чтобы ячейки были преобразованы в таблицу HTML. Но если пользователь переносит произвольный текст HTML в таблицу `QTableWidget`, мы не станем его принимать.

Для того чтобы этот пример заработал, нам потребуется также вызвать `setAcceptDrops(true)` и `setSelectionMode(ContiguousSelection)` в конструкторе `MyTableWidget`.

Теперь мы переделаем этот пример, но на этот раз мы создадим подкласс `QMimeTypeData`, чтобы отложить или избежать (потенциально затратных) преобразований между элементами `QTableWidgetItem` и массивом `QByteArray`. Ниже приводится определение нашего подкласса:

```
class TableMimeTypeData : public QMimeTypeData
{
    Q_OBJECT

public:
    TableMimeTypeData(const QTableWidget *tableWidget,
                      const QTableWidgetSelectionRange &range);

    const QTableWidget *tableWidget() const { return myTableWidget; }
    QTableWidgetSelectionRange range() const { return myRange; }
    QStringList formats() const;
```

```
protected:
    QVariant retrieveData(const QString &format,
                          QVariant::Type preferredType) const;
private:
    static QString toHtml(const QString &plainText);
    static QString toCsv(const QString &plainText);
    QString text(int row, int column) const;
    QString rangeAsPlainText() const;
    const QTableWidgetItem *myTableWidgetItem;
    QTableWidgetItemSelectionRange myRange;
    QStringList myFormats;
};
```

Вместо реальных данных мы храним объект QTableWidgetItemSelectionRange, который определяет область переносимых ячеек и сохраняет указатель на QTableWidgetItem. Функции formats() и retrieveData() класса QMimeData переопределяются.

```
TableMimeData::TableMimeData(const QTableWidgetItem *tableWidget,
                              const QTableWidgetItemSelectionRange &range)
{
    myTableWidgetItem = tableWidget;
    myRange = range;
    myFormats << "text/csv" << "text/html" << "text/plain";
}
```

В конструкторе мы инициализируем закрытые переменные.

```
QStringList TableMimeData::formats() const
{
    return myFormats;
}
```

Функция formats() возвращает список MIME-типов, находящихся в объекте MIME-данных. Последовательность форматов обычно несущественна, однако на практике желательно первыми указывать «лучшие» форматы. Приложения, поддерживающие несколько форматов, иногда будут использовать первый подходящий.

```
QVariant TableMimeData::retrieveData(const QString &format,
                                      QVariant::Type preferredType) const
{
    if (format == "text/plain") {
        return rangeAsPlainText();
    } else if (format == "text/csv") {
        return toCsv(rangeAsPlainText());
    } else if (format == "text/html") {
        return toHtml(rangeAsPlainText());
    } else {
        return QMimeData::retrieveData(format, preferredType);
    }
}
```

Функция retrieveData() возвращает данные для заданного MIME-типа в виде объекта QVariant. Параметр format обычно содержит одну из строк, возвращен-

ных функцией `formats()`, однако нам не следует на это рассчитывать, поскольку не все приложения проверяют MIME-тип на соответствие форматам функции `formats()`. Предусмотренные в классе `QMimeData` функции получения данных `text()`, `html()`, `urls()`, `imageData()`, `colorData()` и `data()` реализуются с помощью функции `retrieveData()`.

Параметр `preferredType` определяет тип, который следует поместить в объект `QVariant`. Здесь мы его игнорируем и рассчитываем на то, что `QMimeData` преобразует при необходимости возвращенное значение в требуемый тип.

```
void MyTableWidget::dropEvent(QDropEvent *event)
{
    const TableMimeData *tableData =
        qobject_cast<const TableMimeData *>(event->mimeType());

    if (tableData) {
        const QTableWidgetItem *otherTable = tableData->tableWidget();
        QTableWidgetItemSelection otherRange = tableData->range();
        ...
        event->acceptProposedAction();
    } else if (event->mimeType()->hasFormat("text/csv")) {
        QByteArray csvData = event->mimeType()->data("text/csv");
        QString csvText = QString::fromUtf8(csvData);
        ...
        event->acceptProposedAction();
    } else if (event->mimeType()->hasFormat("text/plain")) {
        QString plainText = event->mimeType()->text();
        ...
        event->acceptProposedAction();
    }
    QTableWidgetItem::mousePressEvent(event);
}
```

Функция `dropEvent()` аналогична функции с тем же названием, которую мы рассматривали ранее в данном разделе, но на этот раз мы ее оптимизируем, делая в начале проверку возможности приведения типа `QMimeData` в тип `TableMimeData`. Если `qobject_cast<T>()` срабатывает, это значит, что перенос был инициирован виджетом `MyTableWidget`, расположенном в том же самом приложении, и мы можем получить непосредственный доступ к данным таблицы вместо того, чтобы пробираться сквозь программный интерфейс класса `QMimeData`. Если приведение типов оказывается неудачным, мы извлекаем данные стандартным способом.

В этом примере мы кодировали CSV-текст, используя кодировку UTF-8. Если бы мы хотели быть уверенными в применении правильной кодировки, мы могли бы использовать параметр `charset` в MIME-типе `text/plain` для явного задания типа кодировки. Ниже приводится несколько примеров:

```
text/plain; charset=US-ASCII
text/plain; charset=ISO-8859-1
text/plain; charset=Shift_JIS
text/plain; charset=UTF-8
```

Работа с буфером обмена

Большинство приложений тем или иным образом используют встроенные в Qt средства работы с буфером обмена. Например, класс `QTextEdit` обеспечивает поддержку слотов `cut()`, `copy()` и `paste()`, а также клавиш быстрого вызова команд, и поэтому дополнительное программирование почти (или совсем) не требуется.

При создании нами собственных классов мы можем осуществлять доступ к буферу обмена с помощью функции `QApplication::clipboard()`, которая возвращает указатель на объект приложения `QClipboard`. Обработка системного буфера обмена выполняется просто: вызывайте функции `setText()`, `setImage()` или `setPixmap()` для помещения данных в буфер обмена и функции `text()`, `image()` или `pixmap()` для считывания данных из буфера обмена. Мы уже приводили примеры работы с буфером обмена в приложении Электронная таблица из главы 4.

Для некоторых приложений может оказаться не достаточно встроенных функциональных возможностей. Например, нам могут потребоваться данные, которые не являются просто текстом или изображением, или мы захотим обеспечить работу с многими различными форматами данных с целью достижения максимальной совместимости с другими приложениями. Эта проблема очень напоминает ту, с которой мы столкнулись при обеспечении механизма «drag-and-drop», и решение также будет аналогичным: мы можем создать подкласс `QMimeData` и переопределить несколько виртуальных функций.

Если наше приложение поддерживает механизм «drag-and-drop» через пользовательский подкласс `QMimeData`, мы можем просто повторно использовать пользовательский подкласс `QMimeData` и помещать его в буфер обмена, используя функцию `setMimeData()`. Для получения данных мы можем вызвать функцию `mimeType()` для буфера обмена.

В системе X11, как правило, можно вставлять выделенные объекты нажатием средней кнопки мыши, которая имеет три кнопки. Это делается путем применения отдельной «выделенной области» буфера обмена. Если вам нужно, чтобы ваш виджет поддерживал такую операцию буфера обмена вместе со стандартными операциями, вы должны передавать `QClipboard::Selection` в качестве дополнительного аргумента в различных вызовах операций буфера обмена. Например, ниже приводится возможная реализация функции `mousePressEvent()` текстового редактора, поддерживающего вставку по нажатию средней кнопки мыши.

```
void MyTextEditor::mousePressEvent(QMouseEvent *event)
{
    QClipboard *clipboard = QApplication::clipboard();
    if (event->button() == Qt::MidButton
        && clipboard->supportsSelection()) {
        QString text = clipboard->text(QClipboard::Selection);
        pasteText(text);
    }
}
```

В системе X11 функция `supportsSelection()` возвращает `true`. На других платформах она возвращает `false`.

Если мы хотим получать уведомления о каждом изменении содержимого буфера обмена, мы можем соединить сигнал `QClipboard::dataChanged()` с пользовательским слотом.



- *Применение удобных классов отображения элементов*
- *Применение заранее определенных моделей*
- *Реализация пользовательских моделей*
- *Реализация пользовательских делегатов*

Глава 10. Классы отображения элементов

Многие приложения позволяют пользователям выполнять поиск, просмотр и редактирование отдельных элементов, принадлежащих набору данных. Эти данные могут храниться в файлах, в базе данных или на сетевом сервере. Обычно работа с подобными наборами данных осуществляется в Qt с использованием классов отображения элементов.

В ранних версиях Qt виджеты отображения элементов заполнялись содержимым всего набора данных; пользователи обычно выполняли необходимые операции по поиску и редактированию данных, находящихся в виджете, в какой-то момент сделанные изменения записывались обратно в источник данных. Хотя этот метод вполне понятен и прост в применении, он не совсем подходит для больших наборов данных и для ситуаций, когда требуется отображать одни и те же данные в двух или более разных виджетах.

В языке Smalltalk получил популярность гибкий подход к визуальному отображению больших наборов данных: модель-представление-контроллер (model-view-controller – MVC). В подходе MVC модель представляет набор данных и отвечает как за обеспечение отображаемых данных, так и за запись всех изменений в источник данных. Каждый тип набора данных имеет свою собственную модель, однако, предоставляемый моделью программный интерфейс отображения элементов одинаков для наборов данных любого типа. Представление отвечает за то, как данные отображаются для пользователя. При использовании любого большого набора данных только ограниченная область данных будет видима в любой момент времени, поэтому только эти данные будут запрашиваться представлением. Контроллер – это посредник между пользователем и представлением; он преобразует действия пользователя в запросы по просмотру или редактированию данных, которые представление по мере необходимости передает в модель.

В Qt используется вдохновленная подходом MVC архитектура модель/представление. Здесь модель имеет такие же функции, как и в классическом методе MVC. Но вместо контроллера в Qt используется немного другое понятие: делегат.

Делегат обеспечивает более тонкое управление воспроизведением и редактированием элементов. Для каждого типа представления в Qt предусмотрен делегат по умолчанию. Для большинства приложений вполне достаточно пользоваться таким делегатом, поэтому обычно нам не приходится заботиться о нем.

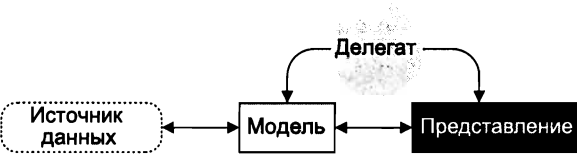


Рис. 10.1. Архитектура модель/представление в Qt

Применяя архитектуру Qt модель/представление, мы можем использовать модели, которые представляют только те данные, которые действительно необходимы для отображения в представлении. Это значительно повышает скорость обработки очень больших наборов данных и уменьшает потребности в памяти по сравнению с подходом, требующим считывания всех данных. Связывая одну модель с двумя или более представлениями, мы можем предоставить пользователю возможность за счет незначительных дополнительных издержек просматривать данные и взаимодействовать с ними различными способами, как показано на рис. 10.2. Qt автоматически синхронизирует множественные представления данных – изменения в одном из представлений отражаются во всех других. Дополнительное преимущество архитектуры модель/представление проявляется в том, что если мы решаем изменить способ хранения исходных данных, нам просто потребуется изменить модель; представления по-прежнему будут работать правильно.

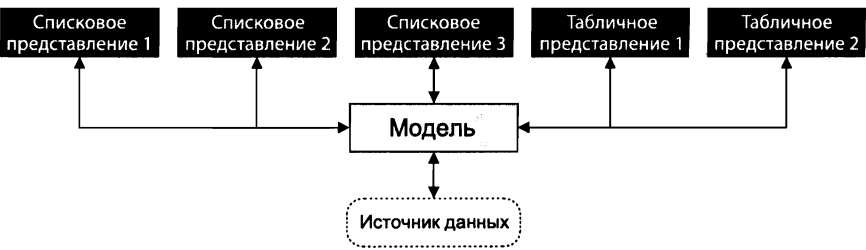


Рис. 10.2. Одна модель может обслуживать несколько представлений

Во многих случаях пользователю необходимо работать только с относительно небольшим количеством элементов. В такой ситуации, как правило, мы можем использовать удобные классы Qt по отображению элементов (QListWidget, QTableWidget и QTreeWidget), непосредственно заполняя все элементы значениями. Эти классы работают подобно классам отображения элементов в предыдущих версиях Qt. Они хранят свои данные в «элементах» (например, QTableWidget содержит элементы QTableWidgetItem). При реализации этих удобных классов используются пользовательские модели, обеспечивающие появление требуемых элементов в представлениях.

При использовании больших наборов данных часто оказывается недопустимым дублирование данных. В этих случаях мы можем применять классы Qt по

отображению элементов (QListView, QTableView и QTreeView) в сочетании с моделью данных, которой может быть как пользовательская модель, так и одна из заранее определенных в Qt моделей. Например, если набор данных хранится в базе данных, мы можем использовать QTableView в сочетании с QSqlTableModel.

Применение удобных классов отображения элементов

Удобные Qt-подклассы отображения элементов обычно использовать проще, чем определять пользовательскую модель, и они особенно удобны, когда разделение модели и представления нам не дает преимущества. Мы использовали этот подход в главе 4, когда создавали подклассы QTableWidget и QTableWidgetItem для реализации функциональности электронной таблицы.

В данном разделе мы покажем, как можно применять удобные классы отображения элементов для вывода на экран элементов. В первом примере приводится используемый только для чтения виджет QListWidget (рис. 10.3), во втором примере – редактируемый QTableWidget (рис. 10.4) и в третьем примере – используемый только для чтения QTreeWidget (рис. 10.5).

Мы начинаем с простого диалогового окна, которое позволяет пользователю выбрать из списка символ, используемый в блок-схемах программ. Каждый элемент состоит из пиктограммы, текста и уникального идентификатора.

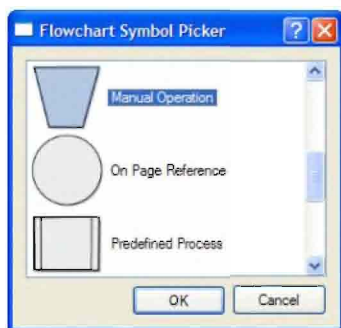


Рис. 10.3. Приложение Выбор символа блок-схемы (Flowchart Symbol Picker)

Сначала покажем фрагмент заголовочного файла диалогового окна:

```
class FlowChartSymbolPicker : public QDialog
{
    Q_OBJECT

public:
    FlowChartSymbolPicker(const QMap<int, QString> &symbolMap,
                          QWidget *parent = 0);
    int selectedId() const { return id; }
    void done(int result);
    ...
};
```

При создании диалогового окна мы должны передать его конструктору ассоциативный массив QMap<int, QString>, и после выполнения конструктора мы мо-

жем получить идентификатор выбранного элемента (или `-1`, если пользователь ничего не выбрал), вызывая `selectedId()`.

```
FlowChartSymbolPicker::FlowChartSymbolPicker(
    const QMap<int, QString> &symbolMap, QWidget *parent)
    : QDialog(parent)
{
    id = -1;

    listWidget = new QListWidget;
    listWidget->setIconSize(QSize(60, 60));

    QMapIterator<int, QString> i(symbolMap);
    while (i.hasNext()) {
        i.next();
        QListWidgetItem *item = new QListWidgetItem(i.value(),
                                                    listWidget);
        item->setIcon(iconForSymbol(i.value()));
        item->setData(Qt::UserRole, i.key());
    }
    ...
}
```

Мы инициализируем `id` (идентификатор последнего выбранного элемента) значением `-1`. Затем мы конструируем `QListWidget` – удобный виджет отображения элементов. Мы проходим в цикле по всем элементам ассоциативного массива символов блок-схемы `symbolMap` и для каждого создаем объект `QListWidgetItem`. Конструктор `QListWidgetItem` принимает выводимую на экран строку `QString` и родительский виджет `QListWidget`.

Затем задаем пиктограмму элемента и вызываем `setData()` для сохранения в `QListWidgetItem` идентификатора элемента. Закрытая функция `iconForSymbol()` возвращает `QIcon` для заданного имени символа.

`QListWidgetItem` может выступать в разных ролях, каждой из которых соответствует определенный объект `QVariant`. Самыми распространенными ролями являются `Qt::DisplayRole`, `Qt::EditRole` и `Qt::IconRole`, и для них предусмотрены удобные функции по установке и получению их значений (`setText()`, `setIcon()`), но имеется также несколько других ролей. Кроме того, мы можем определить пользовательские роли, задавая числовое значение, равное или большее, чем `Qt::UserRole`. В нашем примере мы используем `Qt::UserRole` при сохранении идентификатора каждого элемента.

В непоказанной части конструктора создаются кнопки, выполняется компоновка виджетов и задается заголовок окна.

```
void FlowChartSymbolPicker::done(int result)
{
    id = -1;
    if (result == QDialog::Accepted) {
        QListWidgetItem *item = listWidget->currentItem();
        if (item)
```

```

        id = item->data(Qt::UserRole).toInt();
    }
    QDialog::done(result);
}

```

Функция `done()` класса `QDialog` переопределяется. Она вызывается при щелчке пользователем по кнопкам `OK` или `Cancel`. Если пользователь нажимает кнопку `OK`, мы получаем доступ к соответствующему элементу и извлекаем идентификатор, используя функцию `data()`. Если бы нас интересовал текст элемента, мы могли бы его получить с помощью вызова `item->data(Qt::DisplayRole).toString()` или более простого вызова `item->text()`.

По умолчанию `QListWidget` используется только для чтения. Если бы мы хотели разрешить пользователю редактировать элементы, мы могли бы соответствующим образом установить переключатели редактирования представления, используя `QAbstractItemView::setEditTriggers()`, например, `QAbstractItemView::AnyKeyPressed` означает, что пользователь может инициировать редактирование элемента просто начав вводить символы с клавиатуры. Можно было бы поступить по-другому и предусмотреть кнопку редактирования `Edit` (и, возможно, кнопки добавления и удаления, `Add` и `Delete`) и использовать в слотах соединения сигнал-слот, чтобы можно было программно управлять операциями редактирования.

Теперь, когда мы увидели, как можно использовать удобный класс отображения элементов для просмотра и выбора данных, мы рассмотрим пример, в котором можно редактировать данные. Мы снова используем диалоговое окно, представляющее на этот раз набор точек с координатами (x, y) , которые может редактировать пользователь.

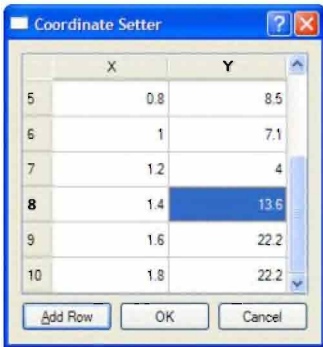


Рис. 10.4. Приложение Редактор координат

Как и в предыдущем примере, мы основное внимание уделим программному коду, относящемуся к представлению элементов, и начнем с конструктора.

```

CoordinateSetter::CoordinateSetter(QList<QPointF> *coords,
                                   QWidget *parent)
    : QDialog(parent)
{
    coordinates = coords;
}

```

```

tableWidget = new QTableWidgetItem(0, 2);
tableWidget->setHorizontalHeaderLabels(
    QStringList() << tr("X") << tr("Y"));

for (int row = 0; row < coordinates->count(); ++row) {
    QPointF point = coordinates->at(row);
    addRow();
    tableWidget->item(row, 0)->setText(QString::number(point.x()));
    tableWidget->item(row, 1)->setText(QString::number(point.y()));
}
...
}

```

Конструктор `QTableWidgetItem` принимает начальное число строк и столбцов таблицы, выводимой на экран. Каждый элемент в `QTableWidgetItem` представлен объектом `QTableWidgetItem`, включая элементы заголовков строк и столбцов. Функция `setHorizontalHeaderLabels()` задает заголовки всем столбцам, используя соответствующий текст из переданного списка строк. По умолчанию `QTableWidgetItem` обеспечивает заголовки строк числовыми метками, начиная с 1; именно это нам и нужно, поэтому нам не приходится задавать вручную заголовки строк.

После создания заголовков столбцов мы в цикле просматриваем все переданные нам данные с координатами. Для каждой пары (x, y) мы добавляем новую строку (используя закрытую функцию `addRow()`) и задаем соответствующим образом текст в каждом столбце строки.

По умолчанию виджет `QTableWidgetItem` разрешает редактирование. Пользователь может редактировать любую ячейку таблицы, установив на нее курсор и нажав F2 или просто вводя текст с клавиатуры. Все сделанные пользователем изменения автоматически сохраняются в элементах `QTableWidgetItem`. Запретить редактирование мы можем с помощью вызова `setEditTriggers(QAbstractItemView::NoEditTriggers)`.

```

void CoordinateSetter::addRow()
{
    int row = tableWidget->rowCount();
    tableWidget->insertRow(row);
    QTableWidgetItem *item0 = new QTableWidgetItem;
    item0->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);
    tableWidget->setItem(row, 0, item0);
    QTableWidgetItem *item1 = new QTableWidgetItem;
    item1->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);
    tableWidget->setItem(row, 1, item1);
    tableWidget->setCurrentItem(item0);
}

```

Слот `addRow()` вызывается, когда пользователь щелкает по кнопке Add Row (добавить строку). Также он используется в конструкторе. Мы добавляем в конец таблицы новую строку, используя `QTableWidgetItem::insertRow()`. Затем мы создаем два элемента `QTableWidgetItem` и добавляем их в таблицу при помощи функции `QTableWidgetItem::setItem()`, которая, наряду с элементом, принимает строку и стол-

бец. Наконец, мы задаем текущий элемент, чтобы пользователь мог начинать редактировать первый элемент новой строки.

```
void CoordinateSetter::done(int result)
{
    if (result == QDialog::Accepted) {
        coordinates->clear();
        for (int row = 0; row < tableWidget->rowCount(); ++row) {
            double x = tableWidget->item(row, 0)->text().toDouble();
            double y = tableWidget->item(row, 1)->text().toDouble();
            coordinates->append(QPointF(x, y));
        }
    }
    QDialog::done(result);
}
```

Когда пользователь нажимает кнопку ОК, мы очищаем координаты, переданные ранее в диалоговое окно, и создаем новый набор на основе координат в элементах виджета `QTableWidget`.

В качестве нашего третьего и последнего примера применения в Qt удобных виджетов отображения элементов мы рассмотрим некоторые фрагменты приложения, которое показывает параметры настройки Qt-приложения, используя `QTreeWidget`. Данный виджет по умолчанию используется только для чтения.

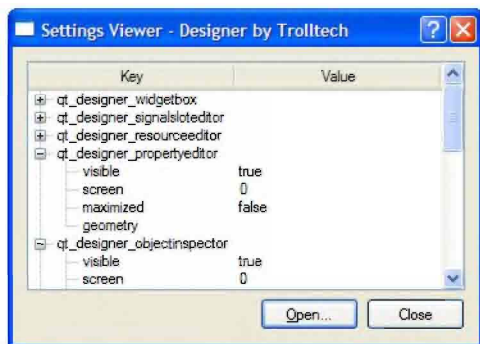


Рис. 10.5. Приложение Просмотр параметров настройки (Settings Viewer)

Ниже приводится фрагмент конструктора:

```
SettingsViewer::SettingsViewer(QWidget *parent)
    : QDialog(parent)
{
    organization = "Trolltech";
    application = "Designer";

    treeWidget = new QTreeWidget;
    treeWidget->setColumnCount(2);
    treeWidget->setHeaderLabels(
        QStringList() << tr("Key") << tr("Value"));
```

```

treeWidget->header()->setResizeMode(0, QHeaderView::Stretch);
treeWidget->header()->setResizeMode(1, QHeaderView::Stretch);
...
setWindowTitle(tr("Settings Viewer"));
readSettings();
}

```

Для получения доступа к параметрам настройки приложения необходимо создать объект `QSettings` с указанием в параметрах названия организации и имени приложения. Мы устанавливаем значения по умолчанию (приложение `Designer` компании `Trolltech`) и затем создаем новый объект `QTreeWidget`. Представление заголовка древовидного виджета управляет размерами столбцов дерева. Мы задаем для обоих столбцов режим изменения размеров в `Stretch`. После этого представление заголовка всегда будет заполнять столбцами все доступное пространство. В этом режиме размер столбцов не может быть изменен ни пользователем, ни программно. В конце конструктора мы вызываем функцию `readSettings()` для заполнения древовидного виджета.

```

void SettingsViewer::readSettings()
{
    QSettings settings(organization, application);

    treeWidget->clear();
    addChildSettings(settings, 0, "");

    treeWidget->sortByColumn(0);
    treeWidget->setFocus();
    setWindowTitle(tr("Settings Viewer - %1 by %2")
        .arg(application).arg(organization));
}

```

Параметры настройки приложения хранятся в виде набора ключей и значений, имеющих иерархическую структуру. Закрывающая функция `addChildSettings()` принимает объект параметров настройки, родительский элемент `QTreeWidgetItem` и текущую «группу». Группа в `QSettings` аналогична каталогу файловой системы. Функция `addChildSettings()` может вызывать себя рекурсивно для прохода по произвольной структуре типа дерева. При первом ее вызове из функции `readSettings()` передается пустой указатель, задавая корень в качестве родительского объекта.

```

void SettingsViewer::addChildSettings(QSettings &settings,
    QTreeWidgetItem *parent, const QString &group)
{
    if (!parent)
        parent = treeWidget->invisibleRootItem();
    QTreeWidgetItem *item;

    settings.beginGroup(group);

    foreach (QString key, settings.childKeys()) {
        item = new QTreeWidgetItem(parent);
    }
}

```

```
        item->setText(0, key);
        item->setText(1, settings.value(key).toString());
    }
    foreach (QString group, settings.childGroups()) {
        item = new QTreeWidgetItem(parent);
        item->setText(0, group);
        addChildSettings(settings, item, group);
    }
    settings.endGroup();
}
```

Функция `addChildSettings()` используется для создания всех элементов `QTreeWidgetItem`. Она проходит по всем ключам текущего уровня в иерархии параметров настройки и для каждого ключа создает один объект `QTreeWidgetItem`. Если в качестве родительского элемента задан пустой указатель, мы создаем дочерний элемент собственно виджета `QTreeWidgetItem::invisibleRootItem()`, т. е. создается элемент верхнего уровня; в первый столбец записывается имя ключа, а во второй столбец – соответствующее ему значение.

Затем эта функция выполняется для каждой группы текущего уровня. Для каждой группы создается новый объект `QTreeWidgetItem`, причем в первый столбец записывается имя группы. Затем эта функция рекурсивно вызывает саму себя с указанием группового элемента в качестве родительского для заполнения виджета `QTreeWidgetItem` дочерними элементами группы.

Показанные в данном разделе виджеты отображения элементов позволяют нам использовать стиль программирования, который очень похож на тот, который применялся в ранних версиях Qt: чтение всего набора данных в виджет отображения элементов с использованием объектов, представляющих отдельные элементы данных и (если элементы допускают редактирование) их запись обратно в источник данных. В последующих разделах мы выйдем за рамки этого простого подхода и воспользуемся всеми преимуществами, которые дает архитектура Qt модель/представление.

Применение заранее определенных моделей

В Qt заранее определено несколько моделей, предназначенных для использования с классами представлений:

<code>QStringListModel</code>	Хранит список строк
<code>QStandardItemModel</code>	Хранит данные произвольной иерархической структуры
<code>QDirModel</code>	Формирует структуру локальной файловой системы
<code>QSqlQueryModel</code>	Формирует набор результата SQL-запроса
<code>QSqlTableModel</code>	Формирует SQL-таблицу
<code>QSqlRelationalTableModel</code>	Формирует SQL-таблицу с внешними ключами (foreign keys)
<code>QSortFilterProxyModel</code>	Сортирует и/или пропускает через фильтр другую модель

В данном разделе мы рассмотрим способы применения моделей `QStringListModel`, `QDirModel` и `QSortFilterProxyModel`. SQL-модели мы рассматриваем в главе 13.

Давайте начнем с простого диалогового окна, которое может применяться для добавления, удаления и редактирования списка строк `QStringList`, где каждая строка представляет лидера команды. Это диалоговое окно показано на рис. 10.6.



Рис. 10.6. Приложение Лидеры команд (Team Leaders)

Ниже приводится соответствующий фрагмент конструктора:

```
TeamLeadersDialog::TeamLeadersDialog(const QStringList &leaders,
                                     QWidget *parent)
    : QDialog(parent)
{
    model = new QStringListModel(this);
    model->setStringList(leaders);
    listView = new QListView;

    listView->setModel(model);
    listView->setEditTriggers(QAbstractItemView::AnyKeyPressed
                             | QAbstractItemView::DoubleClicked);
    ...
}
```

Мы начнем с создания и заполнения модели `QStringListModel`. Затем создадим представление `QListView` и свяжем его с только что созданной моделью. Установим также некоторые переключатели редактирования, чтобы позволить пользователю редактировать строку, просто вводя символ или делая двойной щелчок. По умолчанию все переключатели редактирования сброшены для `QListView`, фактически делая это представление пригодным только для чтения.

```
void TeamLeadersDialog::insert()
{
    int row = listView->currentIndex().row();
    model->insertRows(row, 1);

    QModelIndex index = model->index(row);
    listView->setCurrentIndex(index);
    listView->edit(index);
}
```


Когда пользователь нажимает на кнопку Insert, вызывается слот `insert()`. Этот слот начинает с получения номера строки текущего элемента в списке. Каждый элемент данных модели имеет соответствующий «индекс модели», представленный объектом `QModelIndex`. Мы подробно рассмотрим индексы модели в следующем разделе, а в данный момент нам достаточно знать, что индекс имеет три основных компонента: строку, столбец и указатель на модель, к которой он принадлежит. В модели одномерного списка столбец всегда равен 0.

Имея номер строки, мы вставляем одну новую строку в данную позицию. Вставка выполняется в модели, и модель автоматически обновляет списковое представление. Затем мы устанавливаем текущий индекс спискового представления на пустую строку, которую мы только что вставили. Наконец, мы устанавливаем в списковом представлении режим редактирования для новой строки, как будто пользователь нажал какую-нибудь клавишу клавиатуры или дважды щелкнул, чтобы начать редактирование.

```
void TeamLeadersDialog::del()
{
    model->removeRows(listView->currentIndex().row(), 1);
}
```

В конструкторе сигнал `clicked()` кнопки Delete связывается со слотом `del()`. Поскольку мы только что удалили текущую строку, мы можем вызвать `removeRows()` для текущей позиции индекса и для значения 1 счетчика строк. Как и при выполнении вставки, мы полагаемся на то, что модель должным образом обновит представление.

```
QStringList TeamLeadersDialog::leaders() const
{
    return model->stringList();
}
```

Наконец, функция `leaders()` позволяет считывать отредактированные строки, когда диалоговое окно закрыто.

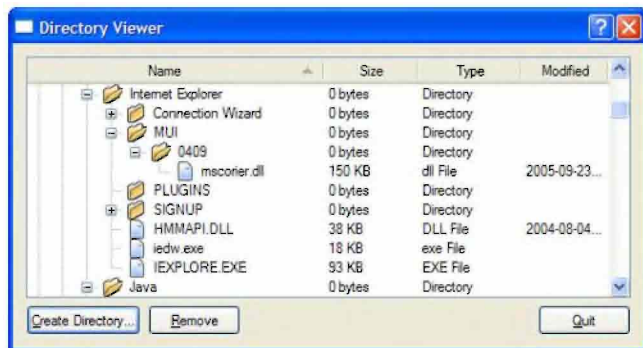


Рис. 10.7. Приложение Просмотр каталога

Создать `TeamLeadersDialog` можно было бы на основе универсального диалогового окна редактирования списка строк, просто параметризируя заголовок этого окна. Другое часто используемое пользователями универсальное диалоговое окно отобра-

жает список файлов или каталогов. В следующем примере, показанном на рис. 10.7, применяется класс `QDirModel`, который моделирует файловую систему компьютера и позволяет показывать (или скрывать) различные атрибуты файлов. Эта модель может применять фильтр для ограничения типов элементов файловой системы при их выводе на экран и упорядочивать элементы различными способами.

Мы начнем с создания и настройки модели и представления в конструкторе диалогового окна `Directory Viewer`.

```
DirectoryViewer::DirectoryViewer(QWidget *parent)
    : QDialog(parent)
{
    model = new QDirModel;
    model->setReadOnly(false);
    model->setSorting(QDir::DirsFirst | QDir::IgnoreCase | QDir::Name);

    treeView = new QTreeView;
    treeView->setModel(model);
    treeView->header()->setStretchLastSection(true);
    treeView->header()->setSortIndicator(0, Qt::AscendingOrder);
    treeView->header()->setSortIndicatorShown(true);
    treeView->header()->setClickable(true);

    QModelIndex index = model->index(QDir::currentPath());
    treeView->expand(index);
    treeView->scrollTo(index);
    treeView->resizeColumnToContents(0);
    ...
}
```

После создания модели мы обеспечиваем возможность ее редактирования и устанавливаем различные начальные атрибуты упорядочивания. Затем мы создаем объект `QTreeView` для отображения на экране данных модели. Заголовок `QTreeView` может использоваться пользователем для управления сортировкой. Делая заголовок восприимчивым к щелчкам мыши, пользователь может сортировать данные по выбранному им в заголовке столбцу, причем повторные щелчки переключают направление сортировки, т. е. сортировку по возрастанию на сортировку по убыванию и наоборот. После настройки заголовков представления данных в виде дерева мы получаем индекс модели текущего каталога и обеспечиваем просмотр содержимого этого каталога, раскрывая при необходимости его подкаталоги, используя `expand()`, и устанавливая изображение на его начало, используя `scrollTo()`. Затем мы обеспечиваем ширину первого столбца, достаточную для размещения всех элементов без вывода многоточия (...).

Во фрагменте конструктора, который здесь не показан, мы связываем кнопки `Create Directory` и `Remove` со слотами, выполняющими соответствующие действия. Нам не нужно иметь кнопку `Rename`, поскольку пользователи могут переименовывать элементы каталога по месту, нажимая клавишу `F2` и осуществляя ввод символов с клавиатуры.

```
void DirectoryViewer::createDirectory()
{
```

```

QModelIndex index = treeView->currentIndex();
if (!index.isValid())
    return;

QString dirName = QInputDialog::getText(this,
                                         tr("Create Directory"),
                                         tr("Directory name"));
if (!dirName.isEmpty()) {
    if (!model->mkdir(index, dirName).isValid())
        QMessageBox::information(this, tr("Create Directory"),
                                   tr("Failed to create the directory"));
}
}

```

Если пользователь вводит имя каталога в диалоговом окне ввода, мы пытаемся создать в текущем каталоге подкаталог с этим именем. Функция `QDirModel::mkdir()` принимает индекс родительского каталога и имя нового каталога; она возвращает индекс модели созданного каталога. Если операция завершается неудачей, возвращается недействительный индекс модели

```

void DirectoryViewer::remove()
{
    QModelIndex index = treeView->currentIndex();
    if (!index.isValid())
        return;
    bool ok;
    if (model->fileInfo(index).isDir()) {
        ok = model->rmdir(index);
    } else {
        ok = model->remove(index);
    }
    if (!ok)
        QMessageBox::information(this, tr("Remove"),
                                   tr("Failed to remove %1").arg(model->fileName(index)));
}

```

Если пользователь нажмет **Remove**, мы попытаемся удалить файл или директорию, связанную с текущим элементом. Для этого мы можем использовать `QDir`, но в классе `QDirModel` предусмотрены удобные функции для работы с индексами `QModelIndex`.

Последний пример в этом разделе, который показан на рис. 10.8, демонстрирует, как следует применять модель `QSortFilterProxyModel`. В отличие от других заранее определенных моделей, эта модель использует какую-нибудь существующую модель и управляет данными, которые проходят между базовой моделью и представлением. В нашем примере базовой является модель `QStringListModel`, которая проинициализирована списком названий цветов, распознаваемых Qt (полученных функцией `QColor::colorNames()`). Пользователь может ввести строку фильтра в строке редактирования `QLineEdit` и указать ее тип (регулярное выражение, шаблон или фиксированная строка), используя поле с выпадающим списком.



Рис. 10.8. Приложение Названия цветов (Color Names)

Ниже приводится фрагмент конструктора `ColorNamesDialog`:

```
ColorNamesDialog::ColorNamesDialog(QWidget *parent)
    : QDialog(parent)
{
    sourceModel = new QStringListModel(this);
    sourceModel->setStringList(QColor::colorNames());
    proxyModel = new QSortFilterProxyModel(this);
    proxyModel->setSourceModel(sourceModel);
    proxyModel->setFilterKeyColumn(0);

    listView = new QListView;
    listView->setModel(proxyModel);
    ...
    syntaxComboBox = new QComboBox;
    syntaxComboBox->addItem(tr("Regular expression"), QRegExp::RegExp);
    syntaxComboBox->addItem(tr("Wildcard"), QRegExp::Wildcard);
    syntaxComboBox->addItem(tr("Fixed string"), QRegExp::FixedString);
    ...
}
```

Модель `QStringListModel` создается и пополняется обычным образом. После этого создается модель `QSortFilterProxyModel`. Мы передаем базовую модель, используя функцию `setSourceModel()`, и указываем прокси на необходимость фильтрации по столбцу 0 базовой модели. Функция `QComboBox::addItem()` принимает необязательный аргумент дополнительных данных типа `QVariant`; мы используем его для хранения значения `QRegExp::PatternSyntax` с текстом, определяющим тип фильтра данного элемента.

```
void ColorNamesDialog::reapplyFilter()
{
    QRegExp::PatternSyntax syntax =
        QRegExp::PatternSyntax(syntaxComboBox->itemData(
            syntaxComboBox->currentIndex()).toInt());
    QRegExp regExp(filterLineEdit->text(), Qt::CaseInsensitive, syntax);
    proxyModel->setFilterRegExp(regExp);
}
```

Слот `reapplyFilter()` вызывается при всяком изменении пользователем строки фильтра или типа шаблона фильтрации в поле с выпадающим списком. Мы создаем объект `QRegExp`, используя текст в строке редактирования. Затем устанавливаем тип шаблона фильтрации на тот, который имеется в данных текущего элемента и отображается в соответствующем поле с выпадающим списком. Когда мы вызываем `setFilterRegExp()`, новый фильтр становится активным, а это значит, что отбрасываются все строки, которые не проходят через фильтр, и автоматически обновляется представление данных.

Реализация пользовательских моделей

Заранее определенные в Qt модели предлагают удобные средства обработки и просмотра данных. Однако некоторые источники данных не могут эффективно использоваться для этих моделей, и в этих случаях необходимо создавать пользовательские модели, оптимизированные на применение таких источников данных.

Прежде чем перейти к созданию пользовательских моделей, давайте рассмотрим ключевые концепции архитектуры Qt модель/представление. В модели каждый элемент имеет индекс модели и набор атрибутов, называемых ролями, которые могут принимать произвольные значения. Ранее в данной главе мы видели, что наиболее распространенными ролями являются `Qt::DisplayRole` и `Qt::EditRole`. Другие роли используются для вспомогательных данных (например, `Qt::ToolTipRole`, `Qt::StatusTipRole` и `Qt::WhatsThisRole`) или для управления основными атрибутами отображения (например, `Qt::FontRole`, `Qt::TextAlignmentRole`, `Qt::TextColorRole` и `Qt::BackgroundColorRole`).

В модели списка можно пользоваться только одним индексным компонентом – номером строки, получить доступ к которому можно с помощью функции `QModelIndex::row()`. В модели таблицы используется два индексных компонента – номер строки и номер столбца, получить доступ к которым можно с помощью функций `QModelIndex::row()` и `QModelIndex::column()`. В моделях списка и таблицы родительский элемент всех остальных элементов является корневым элементом, который представляется недействительным индексом модели `QModelIndex`. Представленные в данном разделе первые два примера показывают, как можно реализовать пользовательские модели таблиц.

Модель дерева подобна модели таблицы при следующих отличиях. Как и в модели таблицы, родительский элемент элементов верхнего уровня является корневым (имеет недействительный `QModelIndex`), однако родительский элемент любого другого элемента занимает другое место в иерархии элементов. Доступ к родительским элементам можно получить при помощи функции `QModelIndex::parent()`. Каждый элемент имеет свои ролевые данные и может иметь или не иметь дочерние элементы, каждый из которых является таким же элементом. Поскольку любой элемент может иметь дочерние элементы, такую структуру данных можно определить рекурсивно (в виде дерева), что будет продемонстрировано в последнем примере данного раздела. На рис. 10.9 схематично показаны разные модели.

В первом примере этого раздела представлена модель таблицы, используемой только для чтения; она показывает курсы различных валют относительно друг друга. Это приложение показано на рис. 10.10.

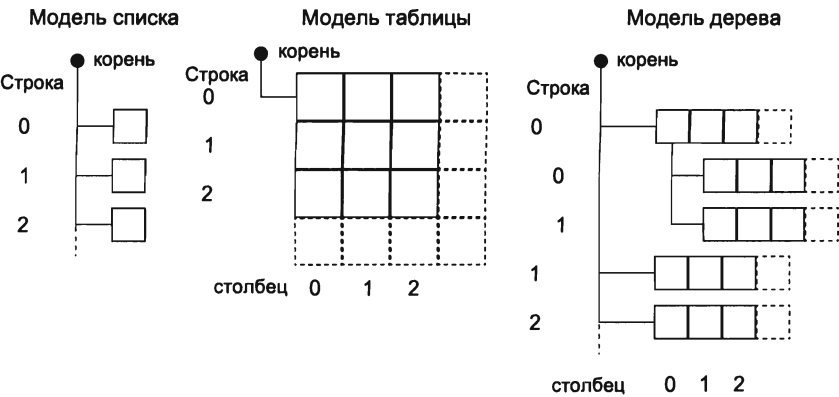


Рис. 10.9. Схематическое представление моделей Qt



Рис. 10.10. Приложение Курсы валют (Currencies)

Это приложение можно было бы реализовать при помощи простой таблицы, но мы хотим применить пользовательскую модель, чтобы можно было воспользоваться определенными свойствами данных для обеспечения минимального расхода памяти. Если бы мы хранили в таблице 162 валюты, действующие в настоящее время, нам бы потребовалось хранить $162 \times 162 = 26\,244$ значений; в представленной ниже пользовательской модели CurrencyModel необходимо хранить только 162 значения (значение каждой валюты относительно доллара США).

Класс CurrencyModel будет использоваться совместно со стандартным табличным представлением QTableView. Модель CurrencyModel пополняется элементами QMap<QString, double>; ключ каждого элемента представляет собой код валюты, а значение – курс валюты в долларах США. Ниже приводится фрагмент программного кода, показывающий, как пополняется ассоциативный массив QMap, и как используется модель:

```
QMap<QString, double> currencyMap;  
currencyMap.insert("AUD", 1.3259);  
currencyMap.insert("CHF", 1.2970);  
...  
currencyMap.insert("SGD", 1.6901);  
currencyMap.insert("USD", 1.0000);  
CurrencyModel currencyModel;
```

```
currencyModel.setCurrencyMap(currencyMap);
QTableView tableView;
tableView.setModel(&currencyModel);
tableView.setAlternatingRowColors(true);
```

Теперь мы можем перейти к реализации модели, начиная с ее заголовка:

```
class CurrencyModel : public QAbstractTableModel
{
public:
    CurrencyModel(QObject *parent = 0);

    void setCurrencyMap(const QMap<QString, double> &map);
    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation, int role) const;
private:
    QString currencyAt(int offset) const;
    QMap<QString, double> currencyMap;
};
```

Для нашей модели мы использовали подкласс `QAbstractTableModel`, поскольку он лучше всего подходит к нашему источнику данных. Qt содержит несколько базовых классов моделей, включая `QAbstractListModel`, `QAbstractTableModel` и `QAbstractItemModel`, см. рис. 10.11. Класс `QAbstractItemModel` используется для поддержки разнообразных моделей, в том числе тех, которые построены на рекурсивных структурах данных, а классы `QAbstractListModel` и `QAbstractTableModel` удобно применять для одномерных и двумерных наборов данных.

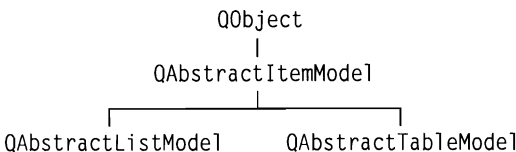


Рис. 10.11. Дерево наследования для абстрактных классов моделей

Для модели таблицы, используемой только для чтения, мы должны переопределить три функции: `rowCount()`, `columnCount()` и `data()`. В данном случае мы также переопределили функцию `headerData()` и обеспечили функцию инициализации данных (`setCurrencyMap()`).

```
CurrencyModel::CurrencyModel(QObject *parent)
    : QAbstractTableModel(parent)
{
}
```

В конструкторе нам ничего не надо делать, кроме передачи базовому классу `parent` в качестве параметра.

```
int CurrencyModel::rowCount(const QModelIndex & /* родительский элемент */
                           const
{
    return currencyMap.count();
}

int CurrencyModel::columnCount(const QModelIndex &
                               /* родительский элемент */) const
{
    return currencyMap.count();
}
```

В этой табличной модели счетчики строк и столбцов представляют собой номера валют в ассоциативном массиве валют. Параметр `parent` не имеет смысла в модели таблицы; он здесь указан, потому что `rowCount()` и `columnCount()` наследуются от более обобщенного базового класса `QAbstractItemModel`, поддерживающего иерархические структуры.

```
QVariant CurrencyModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid())
        return QVariant();

    if (role == Qt::TextAlignmentRole) {
        return int(Qt::AlignRight | Qt::AlignVCenter);
    } else if (role == Qt::DisplayRole) {
        QString rowCurrency = currencyAt(index.row());
        QString columnCurrency = currencyAt(index.column());

        if (currencyMap.value(rowCurrency) == 0.0)
            return "###";
        double amount = currencyMap.value(columnCurrency)
                        / currencyMap.value(rowCurrency);
        return QString("%1").arg(amount, 0, 'f', 4);
    }
    return QVariant();
}
```

Функция `data()` возвращает значение любой одной роли элемента. Элемент определяется индексом `QModelIndex`. В модели таблицы представляют интерес такие компоненты `QModelIndex`, как номер строки и номер столбца, получить доступ к которым можно с помощью функций `row()` и `column()`.

Если используется роль `Qt::TextAlignmentRole`, мы возвращаем значение, подходящее для выравнивания чисел. Если используется роль `Qt::DisplayRole`, мы находим значение каждой валюты и вычисляем курс обмена.

Мы могли бы возвращать рассчитанное значение типа `double`, но тогда нам пришлось бы контролировать количество позиций после десятичной точки при отображении числа (если мы не используем пользовательского делегата). Вместо этого мы возвращаем значение в виде строки, отформатированной нужным нам образом.


```
QVariant CurrencyModel::headerData(int section,
                                   Qt::Orientation /* ориентация */,
                                   int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();
    return currencyAt(section);
}
```

Функция `headerData()` вызывается представлением для пополнения своих горизонтальных и вертикальных заголовков. Параметр `section` содержит номер строки или столбца (в зависимости от ориентации). Поскольку строки и столбцы содержат одинаковые коды валют, нам не надо заботиться об ориентации, а просто вернуть код валюты для заданного значения `section`.

```
void CurrencyModel::setCurrencyMap(const QMap<QString, double> &map)
{
    currencyMap = map;
    reset();
}
```

Вызывающая программа может изменить набор валют, используя функцию `setCurrencyMap()`. Вызов `QAbstractItemModel::reset()` указывает любому представлению, что все данные в используемой модели недействительны; это вынуждает их запросить свежие данные для тех элементов, которые видны на экране.

```
QString CurrencyModel::currencyAt(int offset) const
{
    return (currencyMap.begin() + offset).key();
}
```

Функция `currencyAt()` возвращает ключ (код валюты), который находится по указанному смещению в ассоциативном массиве валют. Мы используем итератор в стиле STL для поиска элемента и вызываем для него функцию `key()`.

Как мы только что могли убедиться, нетрудно создавать модели, используемые только для чтения, и при определенном характере исходных данных в хорошо спроектированной модели, в принципе, можно сэкономить память и увеличить быстродействие. В следующем примере приложения Города (Cities), показанном на рис. 10.12, также используется табличная модель, но на этот раз все данные вводятся пользователем.

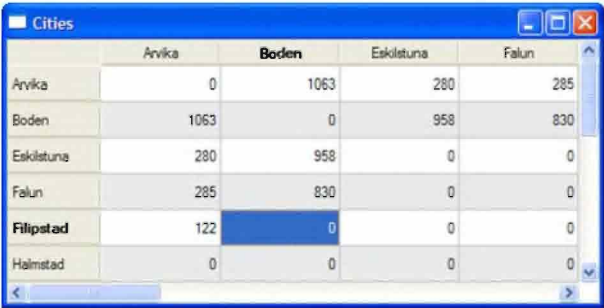


Рис. 10.12. Приложение Города

Это приложение используется для хранения расстояний между любыми двумя городами. Как и в предыдущем примере, мы могли бы просто использовать табличный виджет `QTableWidget` и хранить один элемент для каждой пары городов. Однако пользовательская модель могла бы быть более эффективной, потому что расстояние от любого города **A** до любого другого города **B** не зависит от того, будем ли мы путешествовать от **A** до **B** или от **B** до **A**, поэтому элементы с одной стороны от главной диагонали получаются путем зеркального отражения другой.

Для сравнения пользовательской модели с простой таблицей предположим, что у нас имеется три города: **A**, **B** и **C**. Для обеспечения всех сочетаний нам пришлось бы хранить девять значений. В аккуратно спроектированной модели потребовалось бы только три элемента: (**A**, **B**), (**A**, **C**) и (**B**, **C**).

Ниже показано, как мы настраиваем и используем модель:

```
QStringList cities;
cities << "Arvika" << "Boden" << "Eskilstuna" << "Falun"
      << "Filipstad" << "Halmstad" << "Helsingborg" << "Karlstad"
      << "Kiruna" << "Kramfors" << "Motala" << "Sandviken"
      << "Skara" << "Stockholm" << "Sundsvall" << "Trelleborg";
```

```
CityModel cityModel;
cityModel.setCities(cities);
```

```
QTableView tableView;
tableView.setModel(&cityModel);
tableView.setAlternatingRowColors(true);
```

Мы должны переопределить те же самые функции, которые мы переопределяли в предыдущем примере. Кроме того, для обеспечения возможности редактирования модели мы должны переопределить `setData()` и `flags()`. Ниже приводится определение класса:

```
class CityModel : public QAbstractTableModel
{
    Q_OBJECT

public:
    CityModel(QObject *parent = 0);

    void setCities(const QStringList &cityNames);
    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    bool setData(const QModelIndex &index, const QVariant &value,
                 int role);
    QVariant headerData(int section, Qt::Orientation orientation,
                        int role) const;
    Qt::ItemFlags flags(const QModelIndex &index) const;

private:
    int offsetOf(int row, int column) const;
```

```
QStringList cities;
QVector<int> distances;
};
```

В этой модели мы используем две структуры данных: `cities` типа `QStringList` для хранения названий городов и `distances` типа `QVector<int>` для хранения расстояний между городами каждой уникальной пары.

```
CityModel::CityModel(QObject *parent)
    : QAbstractTableModel(parent)
{
}
```

Конструктор передает параметр `parent` базовому классу и больше ничего не делает.

```
int CityModel::rowCount(const QModelIndex & /* родительский элемент */) const
{
    return cities.count();
}
```

```
int CityModel::columnCount(const QModelIndex & /* родительский элемент */) const
{
    return cities.count();
}
```

Поскольку мы имеем квадратную матрицу городов, количество строк и столбцов равно количеству городов в нашем списке.

```
QVariant CityModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid())
        return QVariant();

    if (role == Qt::TextAlignmentRole) {
        return int(Qt::AlignRight | Qt::AlignVCenter);
    } else if (role == Qt::DisplayRole) {
        if (index.row() == index.column())
            return 0;

        int offset = offsetOf(index.row(), index.column());
        return distances[offset];
    }
    return QVariant();
}
```

Функция `data()` аналогична той же функции в нашей модели `CurrencyModel`. Она возвращает 0, если строка и столбец имеют одинаковый номер, потому что в этом случае два города одинаковы; в противном случае она находит в векторе `distances` элемент для заданной строки и заданного столбца, возвращая расстояние для этой конкретной пары городов.

[illegible]

```

        int role) const
{
    if (role == Qt::DisplayRole)
        return cities[section];
    return QVariant();
}

```

Функция `headerData()` имеет простой вид, потому что наша таблица является квадратной матрицей, в которой строки и столбцы имеют идентичные заголовки. Мы просто возвращаем название города, расположенное с заданным смещением в списке строк `cities`.

```

bool CityModel::setData(const QModelIndex &index,
                        const QVariant &value, int role)
{
    if (index.isValid() && index.row() != index.column()
        && role == Qt::EditRole) {
        int offset = offsetOf(index.row(), index.column());
        distances[offset] = value.toInt();

        QModelIndex transposedIndex = createIndex(index.column(),
                                                    index.row());
        emit dataChanged(index, index);
        emit dataChanged(transposedIndex, transposedIndex);
        return true;
    }
    return false;
}

```

Функция `setData()` вызывается при редактировании элемента пользователем. Если индекс модели действителен, два города различны и модифицируемый элемент данных имеет ролевой атрибут `Qt::EditRole`, эта функция сохраняет введенное пользователем значение в векторе `distances`.

Функция `createIndex()` используется для формирования индекса модели. Она нужна для получения индекса модели элемента, который расположен по другую сторону от главной диагонали и который соответствует элементу с установленным значением, поскольку оба элемента должны показывать одинаковые данные. Функция `createIndex()` принимает сначала строку, а затем столбец; здесь мы передаем параметры в обратном порядке, чтобы получить индекс модели элемента, расположенного по другую сторону диагонали напротив элемента, определенного индексом `index`.

Мы генерируем сигнал `dataChanged()` с указанием индекса модели элемента, который изменился. Эта функция принимает два индекса модели, поскольку возможна ситуация, когда изменения относятся к некоторой прямоугольной области, охватывающей несколько строк и столбцов, поэтому передается индекс верхнего левого угла и индекс нижнего правого угла этой области. Генерируем также сигнал `dataChanged()` для индекса противоположного элемента, чтобы представление обновило его отображение на экране. Наконец, мы возвращаем `true` или `false`, указывая на успешность или не успешность редактирования.

```
Qt::ItemFlags CityModel::flags(const QModelIndex &index) const
{
    Qt::ItemFlags flags = QAbstractItemModel::flags(index);
    if (index.row() != index.column())
        flags |= Qt::ItemIsEditable;
    return flags;
}
```

Функция flags() используется в модели для того, чтобы можно было сообщить о допустимых действиях с элементом (например, допускает ли он редактирование). По умолчанию эта функция для модели QAbstractTableModel возвращает Qt::ItemIsSelectable | Qt::ItemIsEnabled. Мы добавляем флажок Qt::ItemIsEditable для всех элементов, кроме расположенных по диагонали (которые всегда равны 0).

```
void CityModel::setCities(const QStringList &cityNames)
{
    cities = cityNames;
    distances.resize(cities.count() * (cities.count() - 1) / 2);
    distances.fill(0);
    reset();
}
```

Если задан новый список городов, мы устанавливаем закрытую переменную типа QStringList на новый список, изменяем размеры и очищаем вектор расстояний, а затем вызываем функцию QAbstractItemModel::reset(), чтобы уведомить все представления о необходимости обновления всех видимых элементов.

```
int CityModel::offsetOf(int row, int column) const
{
    if (row < column)
        qSwap(row, column);
    return (row * (row - 1) / 2) + column;
}
```

Закрытая функция offsetOf() вычисляет индекс заданной пары городов для вектора расстояний distances. Например, предположим, что мы имеем города A, B, C и D, и пользователь обновляет элемент со строкой 3 и столбцом 1, т. е. (B, D), тогда индекс вектора расстояний будет равен $3 \times (3 - 1) / 2 + 1 = 4$. Если бы пользователь вместо этого изменил элемент со строкой 1 и столбцом 3, т. е. (D, B), благодаря применению функции qSwap(), выполнялись бы точно такие же вычисления, и возвращалось бы то же самое значение. На рис. 10.13 показаны взаимосвязи между городами, расстояниями и соответствующей табличной моделью.

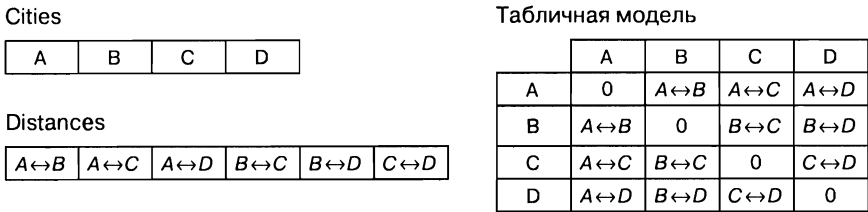


Рис. 10.13. Структуры данных cities и distances и табличная модель

Последний пример в данном разделе представляет собой модель, которая показывает дерево грамматического разбора заданного булевого выражения. Булево выражение представляет собой либо простой буквенно-цифровой идентификатор, например, «bravo», либо сложное выражение, составленное из простых при помощи операторов «&&», «||», «!» или заключения в скобки. Например, «a || (b && !c)» – это булево выражение. Приложение Парсер булевых выражений (Boolean Parser), показанное на рис. 10.14, состоит из четырех классов.

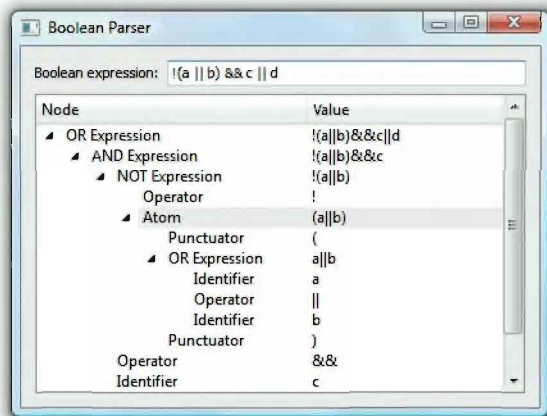


Рис. 10.14. Приложение Парсер булевых выражений

Приложение Парсер регулярных выражений (Regex Parser) состоит из четырех классов:

- BooleanWindow – окно, которое позволяет пользователю вводить булево выражение и показывает соответствующее дерево грамматического разбора;
- BooleanParser формирует дерево грамматического разбора для заданного булева выражения;
- BooleanModel – модель дерева, используемая деревом грамматического разбора;
- Node представляет один элемент в дереве грамматического разбора.

Давайте начнем с класса Node:

```
class Node
{
public:
    enum Type { Root, OrExpression, AndExpression, NotExpression, Atom,
        Identifier, Operator, Punctuator };

    Node(Type type, const QString &str = "");
    ~Node();

    Type type;
    QString str;
    Node *parent;
    QList<Node *> children;
};
```

Каждая вершина имеет тип, строку (которая может быть пустой), ссылку на родительский элемент (которая может быть нулевой) и список дочерних вершин (который может быть пустым).

```
Node::Node(Type type, const QString &str)
{
    this->type = type;
    this->str = str;
    parent = 0;
}
```

Конструктор просто инициализирует тип и строку вершины и задает для `parent` пустое значение (нет родителя). Поскольку все данные открыты, в программном коде, использующим `Node`, можно непосредственно манипулировать типом, строкой, родительским элементом и дочерними элементами.

```
Node::~Node()
{
    qDeleteAll(children);
}
```

Функция `qDeleteAll()` проходит по всем указателям контейнера и вызывает оператор `delete` для каждого из них. Она не устанавливает указатели в пустое значение, поэтому, если она используется вне деструктора, обычно за ней следует вызов функции `clear()` для контейнера, содержащего указатели.

Теперь, когда мы определили элементы наших данных (представленные вершиной `Node`), мы готовы создать модель:

```
class BooleanModel : public QAbstractItemModel
{
public:
    BooleanModel(QObject *parent = 0);
    ~BooleanModel();

    void setRootNode(Node *node);

    QModelIndex index(int row, int column, const QModelIndex &parent) const;
    QModelIndex parent(const QModelIndex &child) const;

    int rowCount(const QModelIndex &parent) const;
    int columnCount(const QModelIndex &parent) const;
    QVariant data(const QModelIndex &index, int role) const;
    QVariant headerData(int section, Qt::Orientation orientation,
                        int role) const;

private:
    Node *nodeFromIndex(const QModelIndex &index) const;
    Node *rootNode;
};
```

На этот раз мы построили подкласс на основе класса `QAbstractItemModel`, а не на основе его удобного подкласса `QAbstractTableModel`, потому что мы хотим создать иерархическую модель. Нам необходимо переопределить те же самые фун-

кции и, кроме того, требуется реализовать функции `index()` и `parent()`. Для установки данных модели предусмотрена функция `setRootNode()`, при вызове которой должна задаваться корневая вершина дерева грамматического разбора.

```
BooleanModel::BooleanModel(QObject *parent)
    : QAbstractItemModel(parent)
{
    rootNode = 0;
}
```

В конструкторе модели нам надо просто задать корневой вершине безопасное нулевое значение и передать указатель `parent` базовому классу.

```
BooleanModel::~BooleanModel()
{
    delete rootNode;
}
```

В деструкторе мы удаляем корневую вершину. Если корневая вершина имеет дочерние вершины, то каждая из них удаляется, и эта процедура повторяется рекурсивно деструктором `Node`.

```
void BooleanModel::setRootNode(Node *node)
{
    delete rootNode;
    rootNode = node;
    reset();
}
```

При установке новой корневой вершины мы начинаем с удаления предыдущей корневой вершины (и всех ее дочерних вершин). Затем мы устанавливаем новое значение для корневой вершины и вызываем функцию `reset()` для уведомления всех представлений о необходимости обновления отображаемых данных всеми видимыми элементами.

```
QModelIndex BooleanModel::index(int row, int column,
                                const QModelIndex &parent) const
{
    if (!rootNode || row < 0 || column < 0)
        return QModelIndex();
    Node *parentNode = nodeFromIndex(parent);
    Node *childNode = parentNode->children.value(row);
    if (!childNode)
        return QModelIndex();
    return createIndex(row, column, childNode);
}
```

Функция `index()` класса `QAbstractItemModel` переопределяется. Она всегда вызывается, когда в модели или в представлении требуется создать индекс `QModelIndex` для конкретного дочернего элемента (или для элемента самого верхнего уровня, если `parent` имеет недействительное значение `QModelIndex`). В табличных и списковых моделях нам не требуется переопределять эту функцию, потому что обычно оказываются достаточными реализации по умолчанию моделей `QAbstractListModel` и `QAbstractTableModel`.

В нашей реализации `index()`, если не задано дерево грамматического разбора, мы возвращаем недействительный индекс `QModelIndex`. В противном случае мы создаем `QModelIndex` с заданными строкой, столбцом и `Node *` для запрошенного дочернего элемента. В иерархических моделях знание строки и столбца элемента относительно своего родителя оказывается недостаточным для уникальной идентификации элемента; мы должны также знать, кто является его родителем. Для этого можно хранить в `QModelIndex` указатель на внутреннюю вершину. В объекте `QModelIndex` кроме номеров строк и столбцов допускается хранение указателя `void *` или значения типа `int`.

Указатель `Node *` на дочерний элемент можно получить из списка дочерних элементов `children` родительской вершины. Указатель на родительскую вершину извлекается из индекса модели `parent` при использовании закрытой функции `nodeFromIndex()`:

```
Node *BooleanModel::nodeFromIndex(const QModelIndex &index) const
{
    if (index.isValid()) {
        return static_cast<Node *>(index.internalPointer());
    } else {
        return rootNode;
    }
}
```

Функция `nodeFromIndex()` приводит тип `void *` заданного индекса в тип `Node *` или возвращает указатель на корневую вершину, если индекс недостоверен, поскольку недостоверный индекс модели используется для представления корня модели.

```
int BooleanModel::rowCount(const QModelIndex &parent) const
{
    if (parent.column() > 0)
        return 0;
    Node *parentNode = nodeFromIndex(parent);
    if (!parentNode)
        return 0;
    return parentNode->children.count();
}
```

Число строк для заданного элемента определяется просто количеством дочерних элементов.

```
int BooleanModel::columnCount(const QModelIndex &
                               /* родительский элемент */) const
{
    return 2;
}
```

Число столбцов фиксировано и равно 2. Первый столбец содержит типы вершин; второй столбец содержит значения вершин.

```
QModelIndex BooleanModel::parent(const QModelIndex &child) const
{
    Node *node = nodeFromIndex(child);
    if (!node)
```

```

        return QModelIndex();
    Node *parentNode = node->parent;
    if (!parentNode)
        return QModelIndex();
    Node *grandparentNode = parentNode->parent;
    if (!grandparentNode)
        return QModelIndex();

    int row = grandparentNode->children.indexOf(parentNode);
    return createIndex(row, 0, parentNode);
}

```

Получить QModelIndex родительского элемента из дочернего немного сложнее, чем найти дочерний элемент родителя. Можно легко получить родительскую вершину, применяя сначала функцию `nodeFromIndex()` и поднимаясь затем вверх с помощью указателя на родительский элемент, но для получения номера строки (позиции родительской вершины в соответствующем списке дочерних вершин), мы должны перейти к родителю родительского элемента и найти в его списке дочерних элементов значение индекса первого родителя (родителя исходной дочерней вершины).

```

QVariant BooleanModel::data(const QModelIndex &index, int role) const
{
    if (role != Qt::DisplayRole)
        return QVariant();

    Node *node = nodeFromIndex(index);
    if (!node)
        return QVariant();

    if (index.column() == 0) {
        switch (node->type) {
            case Node::Root:
                return tr("Root");
            case Node::OrExpression:
                return tr("OR Expression");
            case Node::AndExpression:
                return tr("AND Expression");
            case Node::NotExpression:
                return tr("NOT Expression");
            case Node::Atom:
                return tr("Atom");
            case Node::Identifier:
                return tr("Identifier");
            case Node::Operator:
                return tr("Operator");
            case Node::Punctuator:
                return tr("Punctuator");
            default:
                return tr("Unknown");
        }
    }
}

```

```

    }
    } else if (index.column() == 1) {
        return node->str;
    }
    return QVariant();
}

```

В функции `data()` получаем для запрошенного элемента указатель `Node *` и используем его для получения доступа к данным соответствующей вершины. Если вызывающая программа запрашивает какую-нибудь роль, отличную от `Qt::DisplayRole`, или если не удастся получить вершину `Node` для заданного индекса модели, мы возвращаем недействительное значение типа `QVariant`. Если столбец равен 0, возвращаем название типа вершины; если столбец равен 1, возвращаем значение вершины (ее строку).

```

QVariant BooleanModel::headerData(int section,
                                   Qt::Orientation orientation,
                                   int role) const
{
    if (orientation == Qt::Horizontal && role == Qt::DisplayRole) {
        if (section == 0) {
            return tr("Node");
        } else if (section == 1) {
            return tr("Value");
        }
    }
    return QVariant();
}

```

При переопределении функции `headerData()` мы возвращаем соответствующие метки горизонтального заголовка. Класс `QTreeView`, который используется для визуального представления иерархических моделей, не имеет заголовков строк, поэтому мы их игнорируем.

Теперь, когда рассмотрены классы `Node` и `BooleanModel`, давайте посмотрим, как создается корневая вершина, когда пользователь изменяет текст в строке редактирования.

```

void BooleanWindow::booleanExpressionChanged(const QString &expr)
{
    BooleanParser parser;
    Node *rootNode = parser.parse(expr);
    booleanModel->setRootNode(rootNode);
}

```

При изменении пользователем текста в строке редактирования вызывается слот главного окна `booleanExpressionChanged()`. В этом слоте выполняется синтаксический анализ введенного пользователем текста, и парсер возвращает указатель на корневую вершину дерева грамматического разбора.

Мы не показываем класс `BooleanParser`, потому что он не имеет отношения к графическому интерфейсу или программированию модели/представления. Полный исходный код для этого примера входит в состав примеров, прилагающихся к книге.

При реализации модели дерева, подобной `BooleanModel`, очень легко можно допустить ошибку, которая приведет к непредвиденному поведению `QTreeView`. Для помощи при решении проблем в пользовательских моделях данных существует класс `ModelTest` от Trolltech Labs. Этот класс выполняет серию тестов модели и выявляет распространенные ошибки. Для использования класса `ModelTest` скачайте его с сайта <http://labs.trolltech.com/page/Projects/Itemview/Modeltest> и следуйте инструкциям, приводимым в файле `README`.

В данном разделе мы увидели, как можно создавать три различные пользовательские модели. Многие модели значительно проще приведенных выше и обеспечивают соответствие один-к-одному между элементами и индексами модели. В самой системе Qt находятся дополнительные примеры применения архитектуры модель/представление вместе с подробной документацией.

Реализация пользовательских делегатов

Воспроизведение и редактирование в представлениях отдельных элементов выполняется с помощью делегатов. В большинстве случаев возможности делегата, предоставляемого представлением по умолчанию, оказываются достаточными. Если нам требуется более тонкое управление воспроизведением элементов, мы сможем этого добиться, просто используя пользовательскую модель: при переопределении функции `data()` можем предусмотреть обработку ролей `Qt::FontRole`, `Qt::TextAlignmentRole`, `Qt::TextColorRole` и `Qt::BackgroundColorRole`, а также тех, которые используются делегатом по умолчанию. Например, в приведенных выше приложениях Города и Курсы валют мы применяли `Qt::TextAlignmentRole` для выравнивания чисел вправо.

Если нам требуется еще больший контроль, можем создать наш собственный класс делегата и связать его с нужными нам представлениями. В показанном ниже диалоговом окне Редактор фонограмм (Track Editor) (рис. 10.15) используется пользовательский делегат. В этом окне отображаются названия музыкальных фонограмм и их длительность.



Рис. 10.15. Приложение Редактор фонограмм

Данные в модели будут представлены просто строками `QString` (названия) и значениями типа `int` (секунды), однако длительность будет разбита на минуты и секунды, а ее редактирование будет выполняться, используя `QTimeEdit`.

Диалоговое окно Редактор фонограмм использует `QTableWidget` – удобный подкласс отображения элементов, который работает с объектами `QTableWidgetItem`. Данные представлены в виде списка фонограмм `Track`:

```
class Track
{
public:
    Track(const QString &title = "", int duration = 0);

    QString title;
    int duration;
};
```

Ниже приводится фрагмент конструктора, показывающий, как создается и пополняется табличный виджет:

```
TrackEditor::TrackEditor(QList<Track> *tracks, QWidget *parent)
    : QDialog(parent)
{
    this->tracks = tracks;

    tableWidget = new QTableWidget(tracks->count(), 2);
    tableWidget->setItemDelegate(new TrackDelegate(1));
    tableWidget->setHorizontalHeaderLabels(
        QStringList() << tr("Track") << tr("Duration"));

    for (int row = 0; row < tracks->count(); ++row) {
        Track track = tracks->at(row);
        QTableWidgetItem *item0 = new QTableWidgetItem(track.title);
        tableWidget->setItem(row, 0, item0);

        QTableWidgetItem *item1
            = new QTableWidgetItem(QString::number(track.duration));
        item1->setTextAlignment(Qt::AlignRight);
        tableWidget->setItem(row, 1, item1);
    }
    ...
}
```

Конструктор создает табличный виджет, и вместо того, чтобы просто использовать делегата по умолчанию, связываем виджет с нашим пользовательским делегатом `TrackDelegate`, передавая ему номер столбца, содержащего временные данные. Мы начинаем с установки заголовков столбцов и затем проходим в цикле по всем данным, устанавливая для каждой строки название фонограммы и ее длительность.

В остальной части конструктора и класса `TrackEditor` нет ничего необычного, поэтому теперь рассмотрим класс `trackDelegate`, который обеспечивает воспроизведение и редактирование данных фонограммы.

```
class TrackDelegate : public QTableWidgetItem
{
    Q_OBJECT

public:
    TrackDelegate(int durationColumn, QObject *parent = 0);
    void paint(QPainter *painter, const QStyleOptionViewItem &option,
        const QModelIndex &index) const;
```

```

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;

private slots:
    void commitAndCloseEditor();

private:
    int durationColumn;
};

```

Мы используем `QItemDelegate` в качестве нашего базового класса, чтобы можно было воспользоваться возможностями делегата по умолчанию. Также могли бы использовать `QAbstractItemDelegate`, если бы хотели начать с чистого листа¹. Для обеспечения в делегате возможности редактирования данных мы должны реализовать функции `createEditor()`, `setEditorData()` и `setModelData()`. Кроме того, реализуем функцию `paint()` для изменения отображения столбца длительностей.

```

TrackDelegate::TrackDelegate(int durationColumn, QObject *parent)
    : QItemDelegate(parent)
{
    this->durationColumn = durationColumn;
}

```

Параметр конструктора `durationColumn` указывает делегату, какой номер столбца содержит длительность фонограммы.

```

void TrackDelegate::paint(QPainter *painter,
                        const QStyleOptionViewItem &option,
                        const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        int secs = index.model()->data(index, Qt::DisplayRole).toInt();
        QString text = QString("%1:%2")
            .arg(secs / 60, 2, 10, QChar('0'))
            .arg(secs % 60, 2, 10, QChar('0'));

        QStyleOptionViewItem myOption = option;
        myOption.displayAlignment = Qt::AlignRight | Qt::AlignVCenter;

        drawDisplay(painter, myOption, myOption.rect, text);
        drawFocus(painter, myOption, myOption.rect);
    } else {
        QItemDelegate::paint(painter, option, index);
    }
}

```

¹ Ожидается, что в Qt 4.4 появится класс `QStyledItemDelegate`, который будет использоваться в качестве делегата по умолчанию. В отличие от него классы `QItemDelegate`, `QStyledItemDelegate` используют при рисовании своих элементов текущий стиль.

Поскольку мы собираемся отображать длительность в виде «минуты:секунды», мы переопределили функцию `paint()`. Вызов `arg()` принимает целое число, выводимое в виде строки, допустимое количество символов в строке, основание целого числа (10 для десятичного числа) и символ-заполнитель.

Для выравнивания текста вправо копируем текущие опции стиля и заменяем установленное по умолчанию выравнивание. После этого вызываем `QItemDelegate::drawDisplay()` для вывода текста, затем вызываем `QItemDelegate::drawFocus()` для прорисовки фокусного прямоугольника в том случае, если данный элемент получил фокус, и ничего не делаем в противном случае. Функцией `drawDisplay()` очень удобно пользоваться, особенно, совместно с нашими собственными опциями стиля. Мы могли бы также рисовать, используя рисовальщик непосредственно.

```
QWidget *TrackDelegate::createEditor(QWidget *parent,
    const QStyleOptionViewItem &option,
    const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        QTimeEdit *timeEdit = new QTimeEdit(parent);
        timeEdit->setDisplayFormat("mm:ss");
        connect(timeEdit, SIGNAL(editingFinished()),
            this, SLOT(commitAndCloseEditor()));
        return timeEdit;
    } else {
        return QItemDelegate::createEditor(parent, option, index);
    }
}
```

Мы собираемся управлять редактированием только длительностей фонограмм, предоставляя делегату по умолчанию управление редактированием названий фонограмм. Это обеспечивается проверкой столбца, для которого запрашивается редактирование. Если это столбец длительности, создаем объект `QTimeEdit`, устанавливаем соответствующий формат отображения и соединяем его сигнал `editingFinished()` с нашим слотом `commitAndCloseEditor()`. Для других столбцов передаем управление редактированием делегату по умолчанию.

```
void TrackDelegate::commitAndCloseEditor()
{
    QTimeEdit *editor = qobject_cast<QTimeEdit *>(sender());
    emit commitData(editor);
    emit closeEditor(editor);
}
```

Если пользователь нажимает клавишу `Enter` или убирает фокус из `QTimeEdit` (но не путем нажатия клавиши `Esc`), генерируется сигнал `editingFinished()` и вызывается слот `commitAndCloseEditor()`. Этот слот генерирует сигнал `commitData()` для уведомления представления о том, что имеются новые данные для замены существующих. Он также генерирует сигнал `closeEditor()` для уведомления представления о том, что редактор больше не нужен, и модель его удалит. Получить доступ к редактору можно с помощью функции `QObject::sender()`, которая

возвращает объект, выдавший сигнал, запустивший данный слот. Если пользователь отказывается от работы с редактором (нажимая клавишу Esc), представление просто удалит этот редактор.

```
void TrackDelegate::setEditorData(QWidget *editor,
                                   const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        int secs = index.model()->data(index, Qt::DisplayRole).toInt();
        QTimeEdit *timeEdit = qobject_cast<QTimeEdit *>(editor);
        timeEdit->setTime(QTime(0, secs / 60, secs % 60));
    } else {
        QItemDelegate::setEditorData(editor, index);
    }
}
```

Когда пользователь инициирует редактирование, представление вызывает `createEditor()` для создания редактора и затем `setEditorData()` для инициализации редактора текущими данными элемента. Если редактор вызывается для столбца длительности, получаем из данных элемента длительность фонограммы в секундах и устанавливаем значение `QTimeEdit` на соответствующее количество минут и секунд; в противном случае мы позволяем делегату по умолчанию выполнить инициализацию.

```
void TrackDelegate::setModelData(QWidget *editor,
                                   QAbstractItemModel *model,
                                   const QModelIndex &index) const
{
    if (index.column() == durationColumn) {
        QTimeEdit *timeEdit = qobject_cast<QTimeEdit *>(editor);
        QTime time = timeEdit->time();
        int secs = (time.minute() * 60) + time.second();
        model->setData(index, secs);
    } else {
        QItemDelegate::setModelData(editor, model, index);
    }
}
```

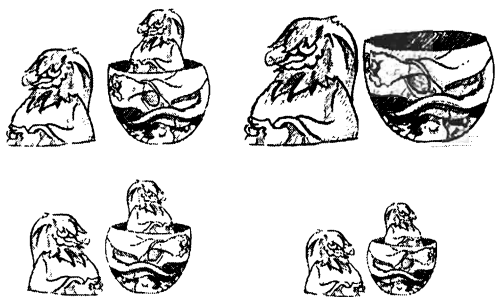
Если пользователь прекращает редактирование (например, щелкнув левой кнопкой мыши за пределами виджета редактора или нажав клавишу Enter или Tab), а не отменяет его, модель должна быть обновлена данными редактора. Если редактировалась длительность, извлекаем минуты и секунды из `QTimeEdit` и устанавливаем поле данных на соответствующее значение секунд.

Мы вполне можем (хотя в данном случае это делать необязательно) создать пользовательский делегат, который обеспечит более тонкое управление редактированием и воспроизведением любого элемента модели. В нашем случае пользовательский делегат управляет только конкретным столбцом, но поскольку `QModelIndex` передается всем функциям класса `QItemDelegate`, которые нами переопределяются, мы можем контролировать любой столбец, строку, прямоугольную область, родительский элемент или любое их сочетание вплоть до управления при необходимости на уровне отдельных элементов.

В данной главе мы представили достаточно подробный обзор архитектуры Qt модель/представление. Мы показали, как можно использовать удобные подклассы отображения элементов, как применять заранее определенные в Qt модели и как создавать пользовательские модели и пользовательские делегаты. Однако архитектура модель/представление настолько богата, что мы не смогли раскрыть все ее возможности из-за ограниченности объема книги. Например, мы могли бы создать пользовательское представление, которое отображает свои элементы не в виде списка, таблицы или дерева. Это делается в примере Диаграмма (Chart), который находится в каталоге Qt examples/itemviews/chart; этот пример содержит пользовательское представление, которое воспроизводит модель данных в виде круговой диаграммы.

Кроме того, для одной модели можно использовать несколько представлений, и это не потребует особых усилий. Любое редактирование одного представления автоматически и немедленно отразится на других представлениях. Такие возможности особенно полезны при просмотре больших наборов данных, когда пользователь может захотеть увидеть блоки данных, расположенные далеко друг от друга. Эта архитектура поддерживает также выделения областей: когда два или более представления используются одной моделью, каждому представлению может быть предоставлена возможность иметь свою собственную независимую выделенную область, или такие области могут совместно использоваться разными представлениями.

В онлайн-официальной документации Qt всесторонне рассматриваются вопросы программирования классов по отображению элементов. См. <http://doc.trolltech.com/4.3/model-view.html>, где приводится список всех таких классов.



- *Последовательные контейнеры*
- *Ассоциативные контейнеры*
- *Обобщенные алгоритмы*
- *Строки, массивы байтов и объекты произвольного типа*

Глава 11. Классы-контейнеры

Классы-контейнеры являются обычными шаблонными классами (template classes), которые предназначены для хранения в памяти элементов заданного типа. C++ уже предлагает много контейнеров в составе стандартной библиотеки шаблонов (STL – Standard Template Library), которая входит в стандартную библиотеку C++.

Qt обеспечивает свои собственные классы-контейнеры, поэтому в Qt-программах мы можем использовать как контейнеры Qt, так и контейнеры STL. Главное преимущество Qt-контейнеров – одинаковое поведение на всех платформах и неявное совместное использование данных. Неявное совместное использование или «копирование при записи» – это оптимизация, позволяющая передавать контейнеры целиком без существенного ухудшения производительности. Qt-контейнеры также снабжены простыми в применении классами итераторов в стиле Java; используя QDataStream, они могут быть оформлены в виде потоков данных и обычно приводят к меньшему объему программного кода в исполняемых модулях, чем при применении соответствующих STL-контейнеров. Наконец, для некоторого оборудования, на котором может работать Qt/Embedded Linux, единственно доступными являются Qt-контейнеры.

Qt предлагает как последовательные контейнеры, например, QVector<T>, QLinkedList<T> и QList<T>, так и ассоциативные контейнеры, например, QMap<K, T> и QHash<K, T>. Концептуально последовательные контейнеры отличаются тем, что элементы в них хранятся один за другим, в то время как в ассоциативных контейнерах хранятся пары ключ-значение.

Qt также содержит обобщенные алгоритмы, которые могут выполняться над произвольными контейнерами. Например, алгоритм qSort() сортирует последовательный контейнер, а qBinaryFind() выполняет двоичный поиск в упорядоченном последовательном контейнере. Эти алгоритмы аналогичны тем, которые предлагаются STL.

Если вы знакомы с контейнерами STL, и библиотека STL уже установлена на платформах, на которых вы работаете, можете их использовать вместо контейнеров Qt или как дополнение к ним. Для получения более подробной информации

относительно функций и классов STL достаточно неплохо начать с веб-сайта STL компании «SGI»: <http://www.sgi.com/tech/stl/>.

В данной главе мы также рассмотрим классы `QString`, `QByteArray` и `QVariant`, поскольку они имеют много общего с контейнерами. `QString` представляет собой 16-битовую строку символов в коде Unicode, которая широко используется в программном интерфейсе Qt. `QByteArray` является массивом 8-битовых символов типа `char`, которыми удобно пользоваться для хранения произвольных двоичных данных. `QVariant` может хранить значения большинства типов C++ и Qt.

Последовательные контейнеры

Вектор `QVector<T>` представляет собой структуру данных, в которой элементы содержатся в соседних участках оперативной памяти, как показано на рис. 11.1. Вектор отличается от обычного массива C++ тем, что знает свой собственный размер, и этот размер может быть изменен. Добавление элементов в конец вектора выполняется достаточно эффективно, но добавление элементов в начало вектора или вставка в его середину могут быть не эффективны.

0	1	2	3	4
937.81	25.984	308.74	310.92	40.9

Рис. 11.1. Вектор чисел двойной точности

Если нам заранее известно необходимое количество его элементов, мы можем задать начальный размер при его определении и использовать оператор `[]` для заполнения его элементами; в противном случае мы должны либо затем изменить его размер, либо добавлять элементы в конец вектора. В приведенном ниже примере мы указываем начальный размер вектора:

```
QVector<double> vect(3);
vect[0] = 1.0;
vect[1] = 0.540302;
vect[2] = -0.416147;
```

Ниже та же самая задача решается путем объявления пустого вектора и применения функции `append()`, которая добавляет элементы в конец вектора:

```
QVector<double> vect;
vect.append(1.0);
vect.append(0.540302);
vect.append(-0.416147);
```

Вместо `append()` можно использовать оператор `<<`:

```
vect << 1.0 << 0.540302 << -0.416147;
```

Организовать цикл просмотра элементов вектора можно при помощи оператора `[]` и функции `count()`:

```
double sum = 0.0;
for (int i = 0; i < vect.count(); ++i)
    sum += vect[i];
```

Элементы вектора, которым не было присвоено какое-нибудь значение явным образом, инициализируются при помощи стандартного конструктора класса элемента. Основные типы и указатели инициализируются нулевым значением.

Вставка элементов в начало или в середину вектора `QVector<T>`, а также удаление элементов из этих позиций могут быть неэффективны для больших векторов. По этой причине Qt предлагает связанный список `QLinkedList<T>` – структуру данных, элементы которой располагаются не в соседних участках памяти (рис. 11.2). В отличие от векторов связанные списки не поддерживают произвольный доступ к элементам, но обеспечивают «константное время» выполнения операций вставки и удаления.

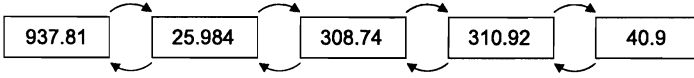


Рис. 11.2. Связанный список значений типа `double`

Связанные списки не обеспечивают оператор `[]`, поэтому необходимо использовать итераторы для прохода по всем элементам. Итераторы также используются для указания позиции элементов. Например, в следующем фрагменте программного кода выполняется вставка строки «Tote Hosen» между «Clash» и «Ramones»:

```

QLinkedList<QString> list;
list.append("Clash");
list.append("Ramones");

QLinkedList<QString>::iterator i = list.find("Ramones");
list.insert(i, "Tote Hosen");

```

Более подробно итераторы будут рассмотрены позже в данном разделе.

Последовательный контейнер `QList<T>` является «массивом-списком», который сочетает в одном классе наиболее важные преимущества `QVector<T>` и `QLinkedList<T>`. Он поддерживает произвольный доступ, и его интерфейс основан на индексировании, подобно применяемому векторами `QVector`. Вставка в конец или удаление последнего элемента списка `QList<T>` выполняется очень быстро, а вставка в середину выполняется быстро для списков, содержащих до одной тысячи элементов. Если не требуется вставлять элементы в середину больших списков и не нужно, чтобы элементы списка занимали последовательные адреса памяти, то `QList<T>` обычно будет наиболее подходящим контейнером общего назначения.

Класс `QStringList` является подклассом `QList<QString>`, который широко используется в программном интерфейсе Qt. Кроме наследуемых от базового класса функций он имеет несколько дополнительных функций, увеличивающих возможности класса по обработке строк. Класс `QStringList` будет обсуждаться нами в последнем разделе этой главы.

`QStack<T>` и `QQueue<T>` – еще два примера удобных подклассов. `QStack<T>` – это вектор, для работы с которым предусмотрены функции `push()`, `pop()` и `top()`. `QQueue<T>` – это список, для работы с которым предусмотрены функции `enqueue()`, `dequeue()` и `head()`.

Во всех до сих пор рассмотренных контейнерах тип элемента `T` может являться базовым типом (например, `int` или `double`), указателем или классом, который имеет стандартный конструктор (т. е. конструктор без аргументов), конструктор копирования и оператор присваивания. К таким классам относятся `QByteArray`,

QDateTime, QRegExp, QString и QVariant. Этим свойством не обладают классы Qt, которые происходят от QObject, поскольку последний не имеет конструктора копирования и оператора присваивания. На практике это не составляет проблему, потому что мы можем просто хранить в контейнере указатели на такие типы данных, а не сами объекты QObject.

Тип T также может быть контейнером; в этом случае следует иметь в виду, что необходимо разделять рядом стоящие угловые скобки пробелами, в противном случае компилятор будет сбив с толку, воспринимая >> как оператор. Например:

```
QList<QVector<double> > list;
```

Кроме только что упомянутых типов в качестве типа элементов контейнера может задаваться любой пользовательский класс, отвечающий описанным ранее критериям. Ниже дается пример такого класса:

```
class Movie
{
public:
    Movie(const QString &title = "", int duration = 0);

    void setTitle(const QString &title) { myTitle = title; }
    QString title() const { return myTitle; }
    void setDuration(int duration) { myDuration = duration; }
    QString duration() const { return myDuration; }

private:
    QString myTitle;
    int myDuration;
};
```

Этот класс имеет конструктор, для которого не обязательно указывать аргументы (хотя он может иметь до двух аргументов). Он также имеет конструктор копирования и оператор присваивания, которые обеспечиваются C++ по умолчанию. В этом классе достаточно обеспечить копирование между его членами, поэтому нам нет необходимости реализовывать свои собственные конструктор копирования и оператор присваивания.

Qt имеет две категории итераторов, используемых для прохода по элементам контейнера: итераторы в стиле Java и итераторы в стиле STL. Итераторами в стиле Java легче пользоваться, в то время как итераторы в стиле STL более мощные и могут использоваться совместно с алгоритмами Qt и STL.

С каждым классом-контейнером может использоваться два типа итераторов в стиле Java: итератор, используемый только для чтения, и итератор, используемый как для чтения, так и для записи. Допустимые места для них показаны на рис. 11.3. Классами итераторов первого типа являются QVectorIterator<T>, QLinkedListIterator<T> и QListIterator<T>. Соответствующие итераторы чтения-записи имеют слово Mutable (изменяемый) в их названии (например, QMutableVectorIterator<T>). В дальнейшем мы основное внимание будем уделять итераторам списка QList; итераторы связанных списков и векторов имеют тот же самый программный интерфейс.

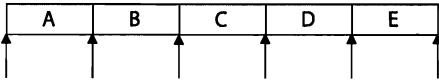


Рис. 11.3. Допустимые позиции итераторов в стиле Java

Прежде всего, следует иметь в виду, что итераторы в стиле Java не ссылаются непосредственно на элементы. Вместо этого они могут указывать на позицию перед первым элементом, после последнего элемента или между двумя элементами. Обычно организованный с их помощью цикл выглядит следующим образом:

```
QList<double> list;
...
QListIterator<double> i(list);
while (i.hasNext()) {
    do_something(i.next());
}
```

Итератор инициализируется контейнером, для прохода по которому он будет использован. В этот момент итератор располагается непосредственно перед первым элементом. Вызов функции `hasNext()` возвращает `true`, если имеется элемент справа от итератора. Функция `next()` возвращает элемент, расположенный справа от итератора и перемещает итератор в следующую допустимую позицию.

Проход в обратном направлении выполняется аналогично с тем отличием, что сначала вызывается функция `toBack()` для размещения итератора после последнего элемента.

```
QListIterator<double> i(list);
i.toBack();
while (i.hasPrevious()) {
    do_something(i.previous());
}
```

Функция `hasPrevious()` возвращает `true`, если имеется элемент слева от итератора; функция `previous()` возвращает элемент, расположенный слева от итератора, и перемещает итератор назад на одну позицию. Возможен другой взгляд на функции `next()` и `previous()`: они возвращают тот элемент, через который только что прошел итератор, как показано на рис. 11.4.

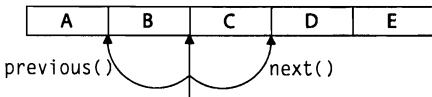


Рис. 11.4. Влияние функций `previous()` и `next()` на итераторы в стиле Java

Допускающие запись итераторы (*mutable iterators*) имеют функции для вставки, модификации и удаления элементов в ходе просмотра контейнеров. В показанном ниже цикле из списка удаляются отрицательные числа:

```
QMutableListIterator<double> i(list);
while (i.hasNext()) {
    if (i.next() < 0.0)
        i.remove();
}
```

Функция `remove()` всегда работает с последним пройденным элементом. Она также ведет себя при проходе элементов в обратном направлении:

```
QMutableListIterator<double> i(list);
i.toBack();
while (i.hasPrevious()) {
    if (i.previous() < 0.0)
        i.remove();
}
```

Аналогично, допускающие запись итераторы в стиле Java имеют функцию `setValue()`, которая модифицирует последний пройденный элемент. Ниже показано, как можно заменить отрицательные числа их абсолютным значением:

```
QMutableListIterator<double> i(list);
while (i.hasNext()) {
    int val = i.next();
    if (val < 0.0)
        i.setValue(-val);
}
```

Кроме того, можно вставлять элемент в текущую позицию итератора с помощью функции `insert()`. После этого итератор перемещается в позицию между новым элементом и следующим за ним.

Кроме итераторов в стиле Java каждый класс последовательных контейнеров `C<T>` имеет итераторы в стиле STL двух типов: `C<T>::iterator` и `C<T>::const_iterator`. Они отличаются тем, что итератор `const_iterator` не позволяет модифицировать данные.

Функция контейнера `begin()` возвращает итератор в стиле STL, ссылающийся на первый элемент контейнера (например, `list[0]`), в то время как функция контейнера `end()` возвращает итератор, ссылающийся на элемент «после последнего элемента» (например, `list[5]` для списка размером 5). На рис. 11.5 показаны допустимые позиции для итераторов в стиле STL. Если контейнер пустой, функции `begin()` и `end()` возвращают одинаковое значение. Это может использоваться для проверки наличия хотя бы одного элемента в контейнере, хотя для этой цели более удобно пользоваться функцией `isEmpty()`.

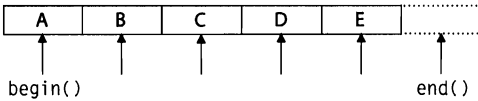


Рис. 11.5. Допустимые позиции итераторов в стиле STL

Синтаксис применения итераторов в стиле STL моделирует синтаксис применения указателей C++. Мы можем использовать операторы `++` и `--` для перехода на следующий или предыдущий элемент, а также унарный оператор `*` для извлечения значения элемента из позиции текущего итератора. Для вектора `vector<T>` типы итераторов `iterator` и `const_iterator` определяются просто как `typedef` для `T *` и `const T *`. (Так можно делать, поскольку `QVector<T>` хранит свои элементы в последовательных адресах памяти.)

В показанном ниже примере каждое значение в списке `QList<double>` заменяется своим абсолютным значением:

```
QList<double>::iterator i = list.begin();
while (i != list.end()) {
    *i = qAbs(*i);
    ++i;
}
```

Несколько функций Qt возвращают контейнер. Если мы хотим в цикле обработать такое возвращенное значение функции, используя итератор в стиле STL, мы должны сделать копию контейнера и в цикле обрабатывать эту копию. Например, приводимый ниже программный код показывает, как правильно следует обрабатывать в цикле список типа `QList<int>`, возвращенный функцией `QSplitter::sizes()`:

```
QList<int> list = splitter->sizes();
QList<int>::const_iterator i = list.begin();
while (i != list.end()) {
    do_something(*i);
    ++i;
}
```

Ниже дается пример неправильного программного кода:

```
// Неправильный программный код

QList<int>::const_iterator i = splitter->sizes().begin();
while (i != splitter->sizes().end()) {
    do_something(*i);
    ++i;
}
```

Это происходит из-за того, что функция `QSplitter::sizes()` возвращает новый список `QList<int>` по значению при каждом новом своем вызове. Если мы не сохраняем возвращенное функцией значение, C++ автоматически удалит его еще до начала итерации, оставляя нам «повисший» итератор. Дело еще усугубляется тем, что на каждом новом шаге цикла функция `QSplitter::sizes()` должна генерировать новую копию списка из-за вызова функции `splitter->sizes().end()`. Поэтому используйте общее правило: когда применяются итераторы в стиле STL, всегда следует обрабатывать в цикле копию экземпляра контейнера, возвращаемого по значению.

При использовании итераторов в стиле Java, предназначенных только для чтения, нам не надо создавать копию. Итератор обеспечит копию незаметно для нас, гарантируя всегда просмотр в цикле данных, только что возвращенных функцией. Например:

```
QListIterator<int> i(splitter->sizes());
while (i.hasNext()) {
    do_something(i.next());
}
```


Подобное копирование контейнера может показаться неэффективным, но это не так из-за оптимизации посредством так называемого неявного совместного использования данных (*implicit sharing*). Это означает, что операция копирования Qt-контейнера выполняется почти также быстро, как копирование одного указателя. Только если скопированная строка изменяется, тогда данные действительно копируются – и все это делается автоматически и незаметно для пользователя. Поэтому неявное совместное использование иногда называют «копированием при записи» (*copy on write*).

Привлекательность неявного совместного использования данных заключается в том, что эта оптимизация выполняется так, что мы можем не думать о ней; она просто работает сама по себе и не требует от нас какого-то дополнительного программного кода. В то же время неявное совместное использование данных способствует тому, что программист следует четкому стилю, возвращая все объекты по значению. Рассмотрим следующую функцию:

```
QVector<double> sineTable()
{
    QVector<double> vect(360);
    for (int i = 0; i < 360; ++i)
        vect[i] = std::sin(i / (2 * M_PI));
    return vect;
}
```

Вызов этой функции выглядит следующим образом:

```
QVector<double> table = sineTable();
```

В отличие от этого подхода STL склоняет нас к передаче вектора в виде неконстантной ссылки, чтобы избежать копирования, происходящего из-за возвращения функцией значения, хранимого в переменной:

```
using namespace std;

void sineTable(std::vector<double> &vect)
{
    vect.resize(360);
    for (int i = 0; i < 360; ++i)
        vect[i] = std::sin(i / (2 * M_PI));
}
```

В результате вызов будет не столь простым и менее понятным:

```
std::vector<double> table;
sineTable(table);
```

В Qt применяется неявное совместное использование данных во всех его контейнерах и во многих других классах, включая `QByteArray`, `QBrush`, `QFont`, `QImage`, `QPixmap` и `QString`. Это делает применение этих классов очень эффективным при передаче по значению, как аргументов функции, так и возвращаемых функциями значений.

Неявное совместное использование данных в Qt гарантирует, что данные не будут копироваться, если мы их не модифицируем. Чтобы получить максимальные выгоды от применения этой технологии, необходимо выработать в себе

две новые привычки при программировании. Одна связана с использованием функции `at()` вместо оператора `[]` при доступе только для чтения к (неконстантному) вектору или списку. Поскольку при применении Qt-контейнеров нельзя сказать, оказывается ли `[]` с левой стороны оператора присваивания или нет, предполагается самое худшее и принудительно выполняется действительное копирование (deep copy), в то время как `at()` не допускается в левой части оператора присваивания.

Подобная проблема возникает при прохождении контейнера с помощью итераторов в стиле STL. Когда вызываются функции `begin()` или `end()` для неконстантного контейнера, Qt всегда принудительно выполняет действительное копирование при совместном использовании данных. Решение, позволяющее избавиться от этой неэффективности, состоит в применении по мере возможности `const_iterator`, `constBegin()` и `constEnd()`.

В Qt предусмотрен еще один последний метод прохода по элементам последовательного контейнера: оператор цикла `foreach`. Он выглядит следующим образом:

```
QLinkedList<Movie> list;
...
foreach (Movie movie, list) {
    if (movie.title() == "Citizen Kane") {
        std::cout << "Found Citizen Kane" << std::endl;
        break;
    }
}
```

Псевдо-ключевое слово `foreach` реализуется с помощью стандартного цикла `for`. На каждом шаге цикла переменная цикла (`movie`) устанавливается на новый элемент, начиная с первого элемента контейнера и затем двигаясь вперед. Цикл `foreach` автоматически использует копию контейнера при входе в цикл, и по этой причине модификации контейнера в ходе цикла не влияют на сам цикл.

Поддерживаются операторы цикла `break` и `continue`. Если тело цикла состоит из одного оператора, не обязательно указывать скобки. Как и для оператора `for`, переменная цикла может определяться вне цикла, например:

```
QLinkedList<Movie> list;
Movie movie;
...
foreach (movie, list) {
    if (movie.title() == "Citizen Kane") {
        std::cout << "Found Citizen Kane" << std::endl;
        break;
    }
}
```

Определение переменной цикла вне цикла — единственная возможность для контейнеров, содержащих типы данных с запятой (например, `QPair<QString,int>`).

Как работает неявное совместное использование данных

Неявное совместное использование данных работает автоматически и незаметно для пользователя, поэтому нам не надо в программном коде предусматривать специальные операторы для обеспечения этой оптимизации. Но поскольку хочется знать, как это работает, мы рассмотрим пример и увидим, что скрывается от нашего внимания. В этом примере используются строки типа `QString` — одного из многих неявно совместно используемых Qt-классов:

```
QString str1 = "Humpty";  
QString str2 = str1;
```

Мы присваиваем переменной `str1` значение «Humpty» (Humpty-Dumpty — Шалтай-Болтай) и переменную `str2` приравниваем к переменной `str1`. К этому моменту оба объекта `QString` ссылаются на одну и ту же внутреннюю структуру данных в памяти. Кроме символьных данных эта структура данных имеет счетчик ссылок, показывающий, сколько строк `QStrings` ссылаются на одну структуру данных. Поскольку обе переменные ссылаются на одни данные, счетчик ссылок будет иметь значение 2.

```
str2[0] = 'D';
```

Когда мы модифицируем переменную `str2`, выполняется действительное копирование данных, чтобы переменные `str1` и `str2` ссылались на разные структуры данных, и их изменение приводило к изменению их собственных копий данных. Счетчик ссылок данных переменной `str1` («Humpty») принимает значение 1 и счетчик ссылок данных переменной `str2` («Dumpty») тоже принимает значение 1. Значение 1 счетчика ссылок означает, что данные не используются совместно.

```
str2.truncate(4);
```

Если мы снова модифицируем переменную `str2`, никакого копирования не будет происходить, поскольку счетчик ссылок данных переменной `str2` имеет значение 1. Функция `truncate()` непосредственно обрабатывает значение переменной `str2`, возвращая в результате строку «Dump». Счетчик ссылок по-прежнему имеет значение 1.

```
str1 = str2;
```

Когда мы присваиваем строку `str2` строке `str1`, счетчик ссылок для данных `str1` снижается до 0 и приводит к тому, что теперь никакая строка типа `QString` не содержит значение «Humpty». Память освобождается. Обе строки `QStrings` теперь ссылаются на значение «Dump», счетчик ссылок которого теперь имеет значение 2.

Часто не пользуются возможностью совместного использования данных в многопоточных программах из-за условий гонок при доступе к счетчикам ссылок. В Qt этой проблемы не возникает. Классы-контейнеры используют инструкции ассемблера при реализации атомарных операций со счетчиком. Эта технология доступна пользователям Qt через применение классов `QSharedData` и `QSharedDataPointer`.

Ассоциативные контейнеры

Ассоциативный контейнер содержит произвольное количество элементов одинакового типа, индексируемых некоторым ключом. Qt содержит два основных класса ассоциативных контейнеров: `QMap<K, T>` и `QHash<K, T>`.

`QMap<K, T>` – это структура данных, которая содержит пары ключ-значение, упорядоченные по возрастанию ключей, как показано на рис. 11.6. Такая организация данных обеспечивает хорошую производительность операций поиска и вставки, а также при проходе данных в порядке их сортировки по ключу. Внутренне `QMap<K, T>` реализуется как слоеный список (skip-list).

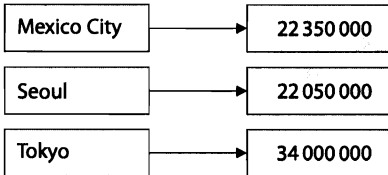


Рис. 11.6. Ассоциативный массив, связывающий `QString` с `int`

Простой способ вставки элементов в ассоциативный массив состоит в использовании функции `insert()`:

```
QMap<QString, int> map;
map.insert("eins", 1);
map.insert("sieben", 7);
map.insert("dreiundzwanzig", 23);
```

Можно поступить по-другому и просто присвоить значение заданному ключу:

```
map["eins"] = 1;
map["sieben"] = 7;
map["dreiundzwanzig"] = 23;
```

Оператор `[]` может использоваться как для вставки, так и для поиска. Но если этот оператор используется для поиска значения, для которого не существует ключа, будет создан новый элемент с данным ключом и пустым значением. Чтобы не создавать случайно пустые элементы, вместо оператора `[]` можно использовать функцию `value()`:

```
int val = map.value("dreiundzwanzig");
```

Если ключ отсутствует в ассоциативном массиве, возвращается значение по умолчанию, создаваемое стандартным конструктором данного типа значений. Для базовых типов и указателей возвращается ноль. Мы можем определить другое значение, используемое по умолчанию, с помощью второго аргумента функции `value()`, например:

```
int seconds = map.value("delay", 30);
```

Это эквивалентно следующим операторам:

```
int seconds = 30;
if (map.contains("delay"))
    seconds = map.value("delay");
```

Типы данных *K* и *T* в ассоциативном массиве *QMap*<*K*, *T*> могут быть базовыми типами (например, *int* и *double*), указатели и классы, которые имеют стандартный конструктор, конструктор копирования и оператор присваивания. Кроме того, тип *K* должен обеспечивать оператор *operator*<(), поскольку *QMap*<*K*, *T*> применяет его для хранения элементов в порядке возрастания значений ключей.

Класс *QMap*<*K*, *T*> имеет две удобные функции, *keys()* и *values()*, которые особенно полезны при работе с небольшими наборами данных. Они возвращают списки типа *QList* ключей и значений ассоциативного массива.

Обычно ассоциативные массивы имеют одно значение для каждого ключа: если новое значение присваивается существующему ключу, старое значение заменяется новым, чтобы не было элементов с одинаковыми ключами. Можно иметь несколько пар ключ-значение с одинаковым ключом, если использовать функцию *insertMulti()* или удобный подкласс *QMultiMap*<*K*, *T*>. *QMap*<*K*, *T*> имеет перегруженную функцию *values(const K &)*, которая возвращает список *QList* со всеми значениями заданного ключа. Например:

```
QMultiMap<int, QString> multiMap;  
multiMap.insert(1, "one");  
multiMap.insert(1, "eins");  
multiMap.insert(1, "uno");  
  
QList<QString> vals = multiMap.values(1);
```

QHash<*K*, *T*> — это структура данных, которая хранит пары ключ-значение в хэш-таблице. Ее интерфейс почти совпадает с интерфейсом *QMap*<*K*, *T*>, однако здесь предъявляются другие требования к шаблонному типу *K*, и операции поиска обычно выполняются значительно быстрее, чем в *QMap*<*K*, *T*>. Еще одним отличием является неупорядоченность значений в *QHash*<*K*, *T*>.

Кроме стандартных требований, которым должен удовлетворять любой тип значений, хранимых в контейнере, для типа *K* в *QHash*<*K*, *T*> должен быть предусмотрен оператор *operator*==(), и должна быть обеспечена глобальная функция *qHash()*, возвращающая хэш-код для ключа. Qt уже имеет перегрузки функции *qHash()* для целых типов, указателей, *QChar*, *QString* и *QByteArray*.

QHash<*K*, *T*> автоматически выделяет некий первичный объем памяти для своей внутренней хэш-таблицы и изменяет его, когда элементы вставляются или удаляются. Кроме того, можно обеспечить более тонкое управление производительностью с помощью функции *reserve()*, которая устанавливает ожидаемое количество элементов в хэш-таблице, и функции *squeeze()*, которая сжимает хэш-таблицу, учитывая текущее количество элементов. Обычно действуют так: вызывают *reserve()*, обеспечивая максимальное ожидаемое количество элементов, затем добавляют данные и, наконец, вызывают *squeeze()* для сведения к минимуму расхода памяти, если элементов оказалось меньше, чем ожидалось.

Хэш-таблицы обычно имеют одно значение на каждый ключ, однако одному ключу можно присвоить несколько значений, используя функцию *insertMulti()* или удобный подкласс *QMultiHash*<*K*, *T*>.

Кроме *QHash*<*K*, *T*> в Qt имеется также класс *QCache*<*K*, *T*>, который может использоваться для создания кэша объектов, связанных с ключом, и контей-

нер `QSet<K>`, который хранит только ключи. Оба класса реализуются на основе `QHash<K, T>` и предъявляют к типу `K` такие же требования, как и `QHash<K, T>`.

Для прохода по всем парам ключ-значение, находящимся в ассоциативном контейнере, проще всего использовать итератор в стиле Java. Поскольку итераторы должны обеспечивать доступ и к ключу и к значению, итераторы в стиле Java работают с ассоциативными контейнерами немного иначе, чем с последовательными контейнерами. Основное отличие проявляется в том, что функции `next()` и `previous()` возвращают пару ключ-значение, а не просто одно значение. Компоненты ключа и значения можно извлечь из объекта пары с помощью функций `key()` и `value()`. Например:

```
QMap<QString, int> map;
...
int sum = 0;
QMapIterator<QString, int> i(map);
while (i.hasNext())
    sum += i.next().value();
```

Если требуется получить доступ как к ключу, так и к значению, мы можем просто игнорировать значение, возвращаемое функциями `next()` и `previous()`, и использовать функции итератора `key()` и `value()`, которые работают с последним пройденным элементом.

```
QMapIterator<QString, int> i(map);
while (i.hasNext()) {
    i.next();
    if (i.value() > largestValue) {
        largestKey = i.key();
        largestValue = i.value();
    }
}
```

Допускающие запись итераторы имеют функцию `setValue()`, которая модифицирует значение, содержащееся в текущем элементе:

```
QMutableMapIterator<QString, int> i(map);
while (i.hasNext()) {
    i.next();
    if (i.value() < 0.0)
        i.setValue(-i.value());
}
```

Итераторы в стиле STL также имеют функции `key()` и `value()`. Для неконстантных типов итераторов `value()` возвращает неконстантную ссылку, позволяя нам изменять значение в ходе просмотра контейнера. Следует отметить, что хотя эти итераторы называются итераторами «в стиле STL», они существенно отличаются от итераторов STL контейнера `map<K, T>`, которые ссылаются на `pair<K, T>`.

Оператор цикла `foreach` также работает с ассоциативными контейнерами, но только с компонентом значение пар ключ-значение. Если нужны как ключи, так и значение, мы можем вызвать функции `keys()` и `values(const K &)` во внутреннем цикле `foreach`:

```
QMultiMap<QString, int> map;  
...  
foreach (QString key, map.keys()) {  
    foreach (int value, map.values(key)) {  
        do_something(key, value);  
    }  
}
```

Обобщенные алгоритмы

В заголовочном файле `<QtAlgorithms>` объявляются глобальные шаблонные функции, которые реализуют основные алгоритмы для контейнеров. Большинство этих функций работают с итераторами в стиле STL.

Заголовочный файл STL `<algorithm>` содержит более полный набор обобщенных алгоритмов. Эти алгоритмы могут использоваться не только с STL-контейнерами, но и с Qt-контейнерами. Если STL доступна на всех ваших платформах, вероятно, нет причин не использовать STL-алгоритмы, когда в Qt отсутствует эквивалентный алгоритм. Далее мы кратко рассмотрим наиболее важные Qt-алгоритмы.

Алгоритм `qFind()` выполняет поиск конкретного значения в контейнере. Он принимает «начальный» и «конечный» итератор и возвращает итератор, ссылающийся на первый подходящий элемент, или «конечный» итератор, если нет подходящих элементов. В представленном ниже примере `i` устанавливается на `list.begin() + 1`, а `j` устанавливается на `list.end()`:

```
QStringList list;  
list << "Emma" << "Karl" << "James" << "Marianne";  
QStringList::iterator i = qFind(list.begin(), list.end(), "Karl");  
  
QStringList::iterator j = qFind(list.begin(), list.end(), "Petra");
```

Алгоритм `qBinaryFind()` выполняет поиск подобно алгоритму `qFind()` за исключением того, что он предполагает упорядоченность элементов в возрастающем порядке и использует двоичный поиск в отличие от линейного поиска в `qFind()`.

Алгоритм `qFill()` заполняет контейнер конкретным значением:

```
QLinkedList<int> list(10);  
qFill(list.begin(), list.end(), 1009);
```

Как и другие алгоритмы, основанные на применении итераторов, `qFill()` может выполняться для части контейнера, если соответствующим образом установить аргументы. В следующем фрагменте программного кода первые пять элементов вектора инициализируются значением 1009, а последние пять элементов – значением 2013:

```
QVector<int> vect(10);  
qFill(vect.begin(), vect.begin() + 5, 1009);  
qFill(vect.end() - 5, vect.end(), 2013);
```

Алгоритм `qCopy()` копирует значения одного контейнера в другой.

```
QVector<int> vect(list.count());  
qCopy(list.begin(), list.end(), vect.begin());
```

`qCopy()` может также использоваться для копирования элементов в рамках одного контейнера, если исходный диапазон и целевой диапазон не перекрываются.

В следующем фрагменте программного кода мы заменяем последние два элемента списка первыми двумя элементами:

```
qCopy(list.begin(), list.begin() + 2, list.end() - 2);
```

Алгоритм qSort() сортирует элементы контейнера в порядке их возрастания.

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

По умолчанию qSort() использует оператор < для сравнения элементов. Для сортировки элементов по убыванию следующим образом передайте qGreater<T>() в качестве третьего аргумента (здесь T – тип элемента контейнера):

```
qSort(list.begin(), list.end(), qGreater<int>());
```

Мы можем использовать третий параметр для определения пользовательского критерия сортировки. Например, ниже приводится функция сравнения «меньше чем», которая выполняет сравнение строк QString без учета регистра:

```
bool insensitiveLessThan(const QString &str1, const QString &str2)
{
    return str1.toLower() < str2.toLower();
}
```

Тогда вызов qSort() будет таким:

```
QStringList list;
...
qSort(list.begin(), list.end(), insensitiveLessThan);
```

Алгоритм qStableSort() аналогичен qSort() за исключением того, что он гарантирует сохранение порядка следования одинаковых элементов. Этот алгоритм стоит применять в тех случаях, когда критерий сортировки учитывает только часть значения элемента, и пользователь видит результат сортировки. Мы использовали qStableSort() в главе 4 для реализации сортировки в приложении Электронная таблица.

Алгоритм qDeleteAll() вызывает оператор delete для каждого указателя, хранимого в контейнере. Он имеет смысл только для контейнеров, в качестве элементов которых используются указатели. После вызова этого алгоритма элементы по-прежнему присутствуют в контейнере в виде висящих указателей, и для их удаления из контейнера используется функция clear(). Например:

```
qDeleteAll(list);
list.clear();
```

Алгоритм qSwap() выполняет обмен значений двух переменных. Например:

```
int x1 = line.x1();
int x2 = line.x2();
if (x1 > x2)
    qSwap(x1, x2);
```

Наконец, заголовочный файл <QtGlobal>, который включается в любой другой заголовочный файл Qt, содержит несколько полезных определений, в том числе функцию qAbs(), которая возвращает абсолютное значение аргумента, и функции qMin() и qMax(), которые возвращают максимальное или минимальное значение двух аргументов.

Строки, массивы байтов и объекты произвольного типа

QString, QByteArray и QVariant – три класса, которые имеют много общего с контейнерами и которые могут использоваться в некоторых контекстах как альтернатива контейнерам. Кроме того, как и контейнеры, эти классы используют неявное совмещение данных для уменьшения расхода памяти и повышения быстродействия.

Мы начнем с рассмотрения типа QString. Строковые данные использует любая программа с графическим пользовательским интерфейсом, и не только непосредственно для пользовательского интерфейса, но часто и в качестве структур данных. В стандартном составе C++ содержится два типа строк: традиционные символьные массивы языка C с завершающим символом «\0» и класс std::string. Класс QString содержит 16-битовые значения в коде Unicode. Unicode содержит в качестве подмножеств коды ASCII и Latin-1 с их обычным числовым представлением. Но поскольку QString имеет 16-битовые значения, он может представлять тысячи других символов, используемых для записи букв большинства мировых языков. Дополнительную информацию по кодировке Unicode вы найдете в главе 18.

При использовании QString не стоит беспокоиться о таких не очень понятных вещах, как выделение достаточного объема памяти или гарантирование завершения данных символом «\0». Концептуально строки QString можно рассматривать как вектор символов QChar. Внутри QString могут быть символы «\0». Функция length() возвращает размер строки, включая символы «\0».

Класс QString содержит бинарный оператор +, обеспечивающий конкатенацию двух строк, и оператор += для добавления одной строки в конец другой. Поскольку QString заранее автоматически добавляет память в конец данных строки, построение строки путем повторения операций добавления символов в конец строки выполняется очень быстро. Ниже приводится пример обоих операторов:

```
QString str = "User: ";  
str += userName + "\n";
```

Существует также функция QString::append(), которая делает то же самое, что и оператор +=:

```
str = "User: ";  
str.append(userName);  
str.append("\n");
```

Совершенно другой способ объединения строк заключается в использовании функции sprintf() класса QString:

```
str.sprintf("%s %.1f%%", "perfect competition", 100.0);
```

Данная функция поддерживает спецификаторы формата, используемые функцией библиотеки C++ sprintf(). В приведенном выше примере переменной str присваивается значение «perfect competition 100.0%» (абсолютно безупречное соревнование).

Имеется еще один способ составления строк из других строк или чисел, и он заключается в использовании функции arg():

```
str = QString("%1 %2 (%3s-%4s)")  
.arg("permissive").arg("society").arg(1950).arg(1970);
```

В этом примере «%1» заменяется словом «permissive» (либеральное), «%2» заменяется словом «society» (общество), «%3» заменяется «1950» и «%4» заменяется «1970». В результате получаем «permissive society (1950s-1970s)» (либеральное общество в 1950-70 годах). Функция `arg()` перегружается для обработки различных типов данных. В некоторых случаях используются дополнительные параметры для управления шириной поля, базой числа или точностью числа с плавающей точкой. В целом, гораздо лучше использовать `arg()`, а не `sprintf()`, поскольку эта функция сохраняет тип, полностью поддерживает Unicode и позволяет трансляторам изменять порядок параметров «%n».

`QString` может преобразовывать числа в строки, используя статическую функцию `QString::number()`:

```
str = QString::number(59.6);
```

Или это можно сделать при помощи функции `setNum()`:

```
str.setNum(59.6);
```

Обратное преобразование строки в число осуществляется при помощи функций `toInt()`, `toLongLong()`, `toDouble()` и т. д. Например:

```
bool ok;  
double d = str.toDouble(&ok);
```

Этим функциям передается необязательный параметр-ссылка на переменную типа `bool`, которая устанавливается на значение `true` или `false` в зависимости от успешности преобразования. Если преобразование завершается неудачей, эти функции возвращают 0.

Имея некоторую строку, нам часто приходится выделять какую-то ее часть. Функция `mid()` возвращает подстроку заданной длины (второй аргумент), начиная с указанной позиции (первый аргумент). Например, следующий программный код¹ выводит на консоль слово «pays»:

```
QString str = "polluter pays principle";  
qDebug() << str.mid(9, 4);
```

Если мы не укажем второй аргумент, функция `mid()` возвратит подстроку, начиная с указанной позиции и до конца строки. Например, следующий программный код выдает на консоль слова «pays principle»:

```
QString str = "polluter pays principle";  
qDebug() << str.mid(9);
```

Существуют также функции `left()` и `right()`, которые выполняют аналогичную работу. Обеим функциям передается количество символов, `n`, и они возвращают первые и последние `n` символов строки. Например, следующий программный код выдает на консоль слова `polluter principle`:

```
QString str = "polluter pays principle";  
qDebug() << str.left(8) << " " << str.right(9);
```

¹ Используемый здесь удобный синтаксис `QDebug() << arg` требует включения заголовочного файла `<QDebug>`, в то время как синтаксис `QDebug("...", arg)` доступен в любом файле, который включает, по крайней мере, один заголовочный файл `Qt`.

Если требуется определить, содержится ли в строке конкретный символ, подстрока или соответствует ли строка регулярному выражению, мы можем использовать один из вариантов функции `indexOf()` класса `QString`:

```
QString str = "the middle bit";
int i = str.indexOf("middle");
```

В результате `i` становится равным 4. Функция `indexOf()` возвращает `-1` при неудачном поиске и принимает в качестве необязательных аргументов начальную позицию и флажок учета регистра.

Если мы просто хотим проверить начальные или конечные символы строки, мы можем использовать функции `startsWith()` и `endsWith()`:

```
if (url.startsWith("http:") && url.endsWith(".png"))
...
```

Предыдущий код проще и быстрее, чем следующий:

```
if (url.left(5) == "http:" && url.right(4) == ".png")
...
```

Оператор сравнения строк `==` зависит от регистра. Если сравниваются строки, которые пользователь видит на экране, обычно правильным решением будет использование функции `localeAwareCompare()`, а если необходимо сделать сравнение независимым от регистра, мы можем использовать функции `toUpper()` или `toLowerCase()`. Например:

```
if (fileName.toLowerCase() == "readme.txt")
...
```

Если мы хотим заменить определенную часть строки другой подстрокой, мы можем использовать функцию `replace()`:

```
QString str = "a cloudy day";
str.replace(2, 6, "sunny");
```

Результатом является «sunny day» (солнечный день). Этот программный код может быть переписан с применением функций `remove()` и `insert()`:

```
str.remove(2, 6);
str.insert(2, "sunny");
```

Во-первых, мы удаляем шесть символов, начиная с позиции 2, и в результате получаем строку «a day» (с двумя пробелами), а затем мы вставляем слово «sunny» в позицию 2.

Существуют перегруженные версии функции `replace()`, которые заменяют все подстроки, совпадающие со значением первого аргумента, на второй аргумент. Например, ниже показано, как можно заменить все символы «&» в строке на «&»:

```
str.replace("&", "&amp;");
```

Часто требуется удалять из строки пробельные символы (пробелы, символы табуляции и перехода на новую строку). `QString` имеет функцию, которая удаляет эти символы с обоих концов строки:

```
QString str = " BOB \t THE \nDOG \n";
QDebug() << str.trimmed();
```

Строку `str` можно представить в виде

			B	O	B		\t		T	H	E		\n	D	O	G		\n

Строка, возвращаемая функцией `trimmed()`, имеет вид

B	O	B		\t		T	H	E			\n	D	O	G

При обработке введенных пользователем данных нам часто необходимо, кроме удаления пробельных символов с обоих концов строки, заменить каждую последовательность таких символов одним пробелом. Именно это выполняет функция `simplified()`:

```
QString str = " BOB \t THE \nDOG \n";
QDebug() << str.simplified();
```

Строка, возвращаемая функцией `simplified()`, имеет вид

B	O	B		T	H	E		D	O	G

Строку можно разбить на подстроки типа `QStringList` при помощи функции `QString::split()`:

```
QString str = "polluter pays principle";
QStringList words = str.split(" ");
```

В предыдущем примере мы разбиваем строку «polluter pays principle» на три подстроки: «polluter», «pays» и «principle». Функция `split()` имеет необязательный второй аргумент, показывающий надо ли оставлять пустые подстроки (режим по умолчанию) или нет.

Элементы списка `QStringList` могут объединяться в одну строку при помощи функции `join()`. Передаваемый функции `join()` аргумент вставляется между каждой парой объединяемых строк. Например, ниже показано, как создавать одну строку из всех строк списка `QStringList`, расположенных в алфавитном порядке и разделенных символом перехода на новую строку:

```
words.sort();
str = words.join("\n");
```

При обработке строк нам часто приходится определять, пустая строка или нет. Это делается при помощи вызова функции `isEmpty()` или проверкой равенства нулю возвращаемого функцией `length()` значения.

Преобразование строк `const char *` в `QString` в большинстве случаев выполняется автоматически, например:

```
str += " (1870)";
```

Здесь мы добавляем строку `const char *` в конец строки `QString` без выполнения явного преобразования. Для явного преобразования `const char *` в `QString` выполните приведение типа в `QString` или вызовите функцию `fromAscii()` или `fromLatin1()`. (Работа с литеральными строками в других кодировках рассматривается в главе 17).

Для преобразования `QString` в `const char *` используйте функцию `toAscii()` или `toLatin1()`. Эти функции возвращают `QByteArray`, который может быть преобразован в `const char *`, при использовании `QByteArray::data()` или `QByteArray::constData()`. Например:

```
printf("User. %s\n", str.toAscii().data());
```

Для удобства в Qt предусмотрен макрос `qPrintable()`, который эквивалентен последовательности функций `toAscii().constData()`:

```
printf("User: %s\n", qPrintable(str));
```

Когда мы вызываем функции `data()` или `constData()` для объектов типа `QByteArray`, владельцем возвращаемой строки будет этот объект. Это означает, что нам не надо беспокоиться о возможных утечках памяти – Qt вернет нам память. С другой стороны мы должны проявлять осторожность и не использовать указатель слишком долго. Если объект `QByteArray` не хранится в переменной, он будет автоматически удален в конце выполнения оператора.

Программный интерфейс класса `QByteArray` очень похож на программный интерфейс класса `QString`. Такие функции, как `left()`, `right()`, `mid()`, `toLower()`, `toUpper()`, `trimmed()` и `simplified()`, существуют в `QByteArray` и имеют такую же семантику как и соответствующие функции в `QString`. `QByteArray` полезно использовать для хранения неформатированных двоичных данных и строк с 8-битовой кодировкой текста. В целом, мы рекомендуем использовать `QString` для хранения текста, а не `QByteArray`, потому что `QString` поддерживает кодировку Unicode.

Для удобства `QByteArray` всегда автоматически обеспечивает наличие символа «\0» после последнего байта, облегчая передачу объекта `QByteArray` функции, принимающей `const char *`. `QByteArray` также может содержать внутри себя символы «\0», что позволяет использовать этот тип для хранения произвольных двоичных данных.

В некоторых ситуациях требуется в одной переменной хранить данные различных типов. Один из таких методов заключается в представлении этих данных в виде `QByteArray` или `QString`. Например, в виде строки можно хранить как текстовое значение, так и числовое значение. Эти подходы обеспечивают максимальную гибкость, но лишают некоторых преимуществ C++, в частности, связанных с безопасностью типов и высокой эффективностью. Qt обеспечивает значительно более удобный способ для хранения данных различного типа: `QVariant`.

Класс `QVariant` может содержать значения многих типов Qt, включая `QBrush`, `QColor`, `QCursor`, `QDateTime`, `QFont`, `QKeySequence`, `QPalette`, `QPen`, `QPixmap`, `QPoint`, `QRect`, `QRegion`, `QSize` и `QString`, а также такие основные числовые типы C++, как `double` и `int`. Класс `QVariant` может, кроме того, содержать контейнеры: `QMap<QString, QVariant>`, `QStringList` и `QList<QVariant>`.

Применение этого типа в классах отображения элементов, в модуле баз данных и в классе `QSettings` получило широкое распространение, позволяя считывать и записывать данные элементов, данные базы данных и пользовательские настройки в виде любого значения, допускаемого типом `QVariant`. Пример этого мы уже видели в главе 3, когда объекты `QRect`, `QStringList` и пара булевых значений передавались функции `QSettings::setValue()` и затем считывались как объекты `QVariant`.

Можно создавать произвольно сложные структуры данных, используя тип `QVariant` для обеспечения вложенных структур контейнеров:

```
QMap<QString, QVariant> pearMap;  
pearMap["Standard"] = 1.95;  
pearMap["Organic"] = 2.25;
```

```
QMap<QString, QVariant> fruitMap;
fruitMap["Orange"] = 2.10;
fruitMap["Pineapple"] = 3.85;
fruitMap["Pear"] = pearMap;
```

Здесь мы создали отображение со строковыми ключами (названия продукции) и значениями, которыми могут быть либо числа с плавающей точкой (цены), либо отображения. Отображение верхнего уровня содержит три ключа: «Orange», «Pear» и «Pineapple» (апельсин, груша и ананас). Значение, связанное с ключом «Pear», является отображением, содержащим два ключа «Standard» и «Organic» (стандартный и экологически чистый). При проходе по ассоциативному массиву, содержащему объекты `QVariant`, нам необходимо использовать функцию `type()` для проверки находящегося в `QVariant` типа, чтобы можно было его правильно обработать.

Способ создания подобным образом структур данных может быть очень привлекательным, поскольку мы можем создавать любые структуры данных. Но удобство применения типа `QVariant` достигается за счет снижения эффективности и читаемости программы. Для хранения наших данных предпочтительнее использовать соответствующий класс языка C++ там, где это возможно.

`QVariant` используется мета-объектной системой Qt и поэтому является частью модуля `QtCore`. Тем не менее, когда мы собираем приложение с модулем `QtGui`, `QVariant` может хранить такие типы, связанные с графическим пользовательским интерфейсом, как `QColor`, `QFont`, `QIcon`, `QImage` или `QPixmap`:

```
QIcon icon("open.png");
QVariant variant = icon;
```

Для извлечения значений этих типов из `QVariant` мы можем следующим образом использовать шаблонную функцию-член `QVariant::value<T>()`:

```
QIcon icon = variant.value<QIcon>();
```

Функция `value<T>()` также может использоваться для преобразования между типами не графического интерфейса и типом `QVariant`, однако на практике обычно используются функции преобразования вида `to...()` (например, `toString()`) для преобразования типов не графического интерфейса.

`QVariant` может также использоваться для хранения пользовательских типов данных при условии обеспечения ими стандартного конструктора и конструктора копирования. Чтобы это заработало, прежде всего, необходимо зарегистрировать тип, используя макрос `Q_DECLARE_METATYPE()` обычно в заголовочном файле после определения класса:

```
Q_DECLARE_METATYPE(BusinessCard)
```

Это позволяет нам написать следующий программный код:

```
BusinessCard businessCard;
QVariant variant = QVariant::fromValue(businessCard);
...
if (variant.canConvert<BusinessCard>()) {
    BusinessCard card = variant.value<BusinessCard>();
    ...
}
```

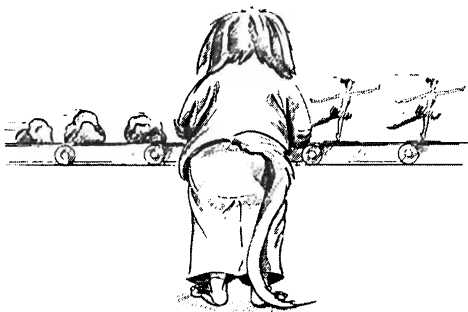
Эти шаблонные функции-члены не будут работать с компилятором MSVC 6. из-за ограничений последнего. Если вы не можете отказаться от этого компилятора, вместо указанных функций используйте глобальные функции `qVariantFromValue()`, `qVariantValue<T>()` и `qVariantCanConvert<T>()`.

Если в пользовательском типе данных предусмотрены операторы `<<` и `>>` для записи и чтения из потока данных `QDataStream`, их можно зарегистрировать, используя функцию `qRegisterMetaTypeStreamOperators<T>()`. Это позволяет, среди прочего, хранить параметры настройки пользовательских типов данных, используя `QSettings`. Например:

```
qRegisterMetaTypeStreamOperators<BusinessCard>("BusinessCard");
```

В данной главе основное внимание было уделено контейнерам Qt, а также классам `QString`, `QByteArray` и `QVariant`. Кроме этих классов Qt имеет несколько других контейнеров. Один из них – `QPair<T1, T2>`, который просто хранит два значения и аналогичен классу `std::pair<T1, T2>`. Еще одним контейнером является `QByteArray`, который мы будем использовать в первом разделе главы 21. Наконец, имеется контейнер `QVarLengthArray<T, Prealloc>` – низкоуровневая альтернатива вектору `QVector<T>`. Поскольку он заранее выделяет память в стеке и не допускает неявное совместное использование, накладные расходы у него меньше, чем у вектора `QVector<T>`, что делает его более подходящим в напряженных циклах.

Более подробное описание контейнеров Qt, в том числе информацию об их временных и объемных характеристиках, можно найти на странице <http://doc.trolltech.com/4.3/containers.html>.



- Чтение и запись двоичных данных
- Чтение и запись текста
- Работа с каталогами
- Ресурсы, внедренные в исполняемый модуль
- Связь между процессами

Глава 12. Ввод-вывод

Почти в каждом приложении приходится читать или записывать файлы или выполнять другие операции ввода/вывода. Qt обеспечивает великолепную поддержку ввода/вывода при помощи `QIODevice` – мощной абстракции «устройств», способных читать и записывать блоки байтов. Qt содержит следующие подклассы `QIODevice`:

<code>QFile</code>	Получает доступ к файлам, находящимся в локальной файловой системе или внедренным в исполняемый модуль
<code>QTemporaryFile</code>	Создает временные файлы в локальной файловой системе и получает доступ к ним
<code>QBuffer</code>	Считывает или записывает данные в <code>QByteArray</code>
<code>QProcess</code>	Запускает внешние программы и обеспечивает связь между процессами.
<code>QTcpSocket</code>	Передает поток данных по сети, используя протокол TCP
<code>QUdpSocket</code>	Передает и принимает из сети дейтаграммы UDP
<code>QSslSocket</code>	Передает зашифрованный поток данных по сети, используя протоколы SSL/TLS

`QProcess`, `QTcpSocket`, `QUdpSocket` и `QSslSocket` являются последовательными устройствами, т. е. они позволяют получить доступ к данным только один раз, начиная с первого байта и последовательно продвигаясь к последнему байту. `QFile`, `QTemporaryFile` и `QBuffer` являются устройствами произвольного доступа и позволяют считывать байты многократно из любой позиции; они используют функцию `QIODevice::seek()` для изменения положения указателя файла.

Кроме этих устройств Qt предоставляет два класса высокоуровневых потоков данных, которые можно использовать для чтения и записи на любое устройство ввода/вывода: `QDataStream` для двоичных данных и `QTextStream` для текста. Эти классы учитывают такие аспекты, как порядок байтов и кодировка текста, позволяя работающим на разных платформах и в разных странах приложениям Qt считывать и записывать файлы друг друга. Это делает классы Qt по вводу/выводу более удобными, чем соответствующие классы стандартного C++, при использовании которых решать подобные проблемы приходится прикладному программисту.

QFile позволяет легко получать доступ к отдельным файлам, независимо от того, располагаются они в файловой системе или оказываются внедренными в исполняемый модуль приложения как ресурсы. Для приложений, которым приходится работать с целыми наборами файлов, в Qt предусмотрены классы QDir и QFileInfo, которые позволяют работать с каталогами и получать сведения о файлах, расположенных внутри каталогов.

Класс QProcess позволяет нам запускать внешние программы и устанавливать связь с ними через стандартные каналы ввода, вывода и ошибок (cin, cout, и cerr). Мы можем устанавливать переменные среды и рабочий каталог, которые будут использоваться внешним приложением. По умолчанию связь с процессом осуществляется в асинхронном режиме (без блокировок), но все же остается возможной блокировка определенных операций.

Работа с сетью, а также чтение и запись документов XML настолько важные темы, что мы рассматриваем их отдельно в главах, специально им посвященных (главы 15 и 16).

Чтение и запись двоичных данных

Самый простой способ загрузки и сохранения двоичных данных в Qt – получить экземпляр класса QFile, открыть файл и получить к нему доступ через объект QDataStream. QDataStream обеспечивает независимый от платформы формат памяти, который поддерживает такие базовые типы C++, как int и double, и многие типы данных Qt, включая QByteArray, QFont, QImage, QPixmap, QString и QVariant, а также классы-контейнеры Qt, например, QList<T> и QMap<K, T>.

Ниже показано, как можно сохранить целый тип, QImage и QMap<QString, QColor> в файле с именем facts.dat:

```
QImage image("philip.png");

QMap<QString, QColor> map;
map.insert("red", Qt::red);
map.insert("green", Qt::green);
map.insert("blue", Qt::blue);

QFile file("facts.dat");
if (!file.open(QIODevice::WriteOnly)) {
    std::cerr << "Cannot open file for writing: "
              << QString(file.errorString()) << std::endl;
    return;
}

QDataStream out(&file);
out.setVersion(QDataStream::Qt_4_3);
out << quint32(0x12345678) << image << map;
```

Если не удастся открыть файл, мы информируем об этом пользователя и возвращаем управление. Макрос qPrintable() возвращает const char *, принимая

QString. (Можно было бы поступить по-другому и использовать функцию `QString::toString()`, возвращающую тип `std::string`, для которого в `<iostream>` предусмотрена соответствующая перегрузка оператора `<<`).

При успешном открытии файла мы создаем `QDataStream` и определяем его номер версии. Номер версии – это целое число, влияющее на представление в Qt типов данных (базовые типы данных C++ всегда представляются одинаково). В Qt 4.3 большинство сложных форматов имеют версию 9. Мы можем либо жестко закодировать в программе константу 9, либо использовать символическое имя `QDataStream::Qt_4_3`.

Чтобы обеспечить представление значения `0x12345678` в виде 32-битового целого числа без знака на всех платформах, мы приводим его тип к `quint32` – типу данных, размер которого всегда равен точно 32 битам. Для обеспечения функциональной совместимости `QDataStream` по умолчанию использует прямой порядок байтов (`big-endian`); это можно изменить, вызывая функцию `setByteOrder()`.

Нам не надо явно закрывать файл, поскольку это делается автоматически, когда переменная типа `QFile` выходит из области видимости. Если необходимо убедиться в том, что данные действительно записаны, мы можем вызвать функцию `flush()` и проверить возвращаемое значение (`true` при успешном завершении).

Программный код для чтения данных является зеркальным отражением кода, используемого нами для записи данных:

```
quint32 n;
QImage image;
QMap<QString, QColor> map;

QFile file("facts.dat");
if (!file.open(QIODevice::ReadOnly)) {
    std::cerr << "Cannot open file for reading: "
              << printable(file.errorString()) << std::endl;
    return;
}

QDataStream in(&file);
in.setVersion(QDataStream::Qt_4_3);

in >> n >> image >> map;
```

При чтении используется та же самая версия `QDataStream`, которую мы использовали при записи. Это условие должно выполняться всегда. Жестко кодируя номер версии, мы гарантируем успешное чтение и запись данных приложением (при условии компиляции приложения с версией Qt 4.3 или более поздней версией Qt).

`QDataStream` так хранит данные, что мы сможем их считать обратно без особых усилий. Например, `QByteArray` представляется в виде структуры с 32-битовым счетчиком байтов, за которым идут сами байты. Используя функции `readRawBytes()` и `writeRawBytes()`, `QDataStream` может также применяться для чтения и записи неформатированных байтов, не имеющих заголовка в виде счетчика байтов.

Обрабатывать ошибки при чтении данных из потока `QDataStream` достаточно просто. Этот поток данных имеет функцию `status()`, возвращающую значения `QDataStream::Ok`, `QDataStream::ReadPastEnd` или `QDataStream::ReadCorruptData`. При возникновении ошибки оператор `>>` всегда считывает нулевые или пустые значения. Это означает, что во многих случаях можно просто считывать файл целиком, не беспокоясь о возможных ошибках, и в конце удостовериться в успешном выполнении чтения, проверив получаемое функцией `status()` значение.

`QDataStream` работает с разнообразными типами данных `C++` и `Qt`; полный их список доступен в сети Интернет по адресу <http://doc.trolltech.com/4.3/datastreamformat.html>. Кроме того, можно добавить поддержку своих собственных пользовательских типов, перегружая операторы `<<` и `>>`. Ниже приводится определение пользовательского типа данных, которое может быть использовано совместно с `QDataStream`:

```
class Painting
{
public:
    Painting() { myYear = 0; }
    Painting(const QString &title, const QString &artist, int year) {
        myTitle = title;
        myArtist = artist;
        myYear = year;
    }

    void setTitle(const QString &title) { myTitle = title; }
    QString title() const { return myTitle; }
    ...

private:
    QString myTitle;
    QString myArtist;
    int myYear;
};

QDataStream &operator<<(QDataStream &out, const Painting &painting);
QDataStream &operator>>(QDataStream &in, Painting &painting);
```

Ниже показана возможная реализация оператора `<<`:

```
QDataStream &operator<<(QDataStream &out, const Painting &painting)
{
    out << painting.title() << painting.artist()
        << quint32(painting.year());
    return out;
}
```

Для вывода `Painting` мы просто выводим две строки типа `QString` и значение типа `quint32`. В конце функции мы возвращаем поток. Этот, обычный в `C++`, прием позволяет использовать последовательность операторов `<<` для вывода данных в поток. Например:

```
out << painting1 << painting2 << painting3;
```

Реализация оператора >> аналогична реализации оператора <<.

```
QDataStream &operator>>(QDataStream &in, Painting &painting)
{
    QString title;
    QString artist;
    quint32 year;

    in >> title >> artist >> year;
    painting = Painting(title, artist, year);
    return in;
}
```

Обеспечение в пользовательских типах данных операторов ввода/вывода в поток дает несколько преимуществ. Одно из них заключается в том, что это позволяет нам выводить в поток контейнеры с пользовательскими типами. Например:

```
QList<Painting> paintings = ...;
out << paintings;
```

Мы можем так же просто считывать контейнеры:

```
QList<Painting> paintings;
in >> paintings;
```

Это привело бы к ошибке компиляции, если бы тип `Painting` не поддерживал операции `<<` или `>>`. Еще одно преимущество обеспечения потоковых операторов в пользовательских типах заключается в возможности хранения этих типов в виде объектов `QVariant`, что расширяет возможности их применения, например, в объектах `QSettings`. Это будет работать при условии предварительной регистрации типа с помощью функции `qRegisterMetaTypeStreamOperators<T>()`, работа которой рассматривается в главе 11.

При использовании `QDataStream` Qt обеспечивает чтение и запись каждого типа, включая контейнеры с произвольным числом элементов. Это освобождает нас от структурирования того, что мы записываем, и от выполнения какого-то бы ни было синтаксического анализа того, что мы считываем. Необходимо лишь гарантировать чтение всех типов в той же последовательности, в какой они были записаны, предоставляя Qt обработку всех деталей.

`QDataStream` имеет смысл использовать как для своих собственных пользовательских форматов файлов, так и для стандартных двоичных форматов. Мы можем считывать и записывать стандартные форматы двоичных данных, используя потоковые операторы для базовых типов (например, `quint16` или `float`), или при помощи функций `readRawBytes()` и `writeRawBytes()`. Если `QDataStream` используется только для чтения и записи «чистых» типов данных C++, нет необходимости вызывать функцию `setVersion()`.

До сих пор мы загружали и сохраняли данные, жестко задавая в программе версию потока `QDataStream::Qt_4_3`. Этот подход прост, и он надежно работает, но он имеет один небольшой недостаток: мы не сможем воспользоваться новыми форматами и обновленными версиями форматов. Например, если в более поздней версии Qt добавится новый атрибут к `QFont` (кроме размера точки, наименования

шрифта, и так далее) и мы жестко закодируем номер версии Qt_4_3, этот атрибут не будет сохраняться и загружаться. Существует два решения. Первое решение заключается во включении номера версии QDataStream в файл:

```
QDataStream out(&file);  
out << quint32(MagicNumber) << quint16(out.version());
```

(MagicNumber – это константа, которая уникально идентифицирует тип файла.) В этом случае мы всегда будем записывать данные с применением последней версии QDataStream (каким бы результат не был). При считывании файла мы считываем номер версии потока:

```
quint32 magic;  
quint16 streamVersion;  
  
QDataStream in(&file);  
in >> magic >> streamVersion;  
  
if (magic != MagicNumber) {  
    std::cerr << "File is not recognized by this application"  
              << std::endl;  
} else if (streamVersion > in.version()) {  
    std::cerr << "File is from a more recent version of the"  
              << "application"  
              << std::endl;  
    return false;  
}  
  
in.setVersion(streamVersion);
```

Мы можем считывать данные, если версия потока меньше или совпадает с версией, используемой в приложении; в противном случае, мы выдаем сообщение об ошибке.

Если файл использует формат с собственным номером версии, мы можем его использовать для определения номера версии потока, а не хранить этот номер в явном виде. Например, предположим, что файл сформирован в формате версии 1.3 нашего приложения. Тогда мы могли бы записать данные следующим образом:

```
QDataStream out(&file);  
out.setVersion(QDataStream::Qt_4_3);  
out << quint32(MagicNumber) << quint16(0x0103);
```

При считывании данных мы определяем версию QDataStream на основе номера версии приложения:

```
QDataStream in(&file);  
in >> magic >> appVersion;  
  
if (magic != MagicNumber) {  
    std::cerr << "File is not recognized by this application"  
              << std::endl;  
    return false;
```

```

} else if (appVersion > 0x0103) {
    std::cerr << "File is from a more recent version of the"
               << "application"
               << std::endl;
    return false;
}

if (appVersion <= 0x0103) {
    in.setVersion(QDataStream::Qt_3_0);
} else {
    in.setVersion(QDataStream::Qt_4_3);
}

```

В этом примере мы говорим, что для любого файла, сохраненного в приложении с версией, меньшей, чем 1.3, используется версия 4 потока данных (Qt_3_0), а для файлов, сохраненных в приложении с версией 1.3, используется версия 9 потока данных (Qt_4_3).

Итак, существует три политики работы с версиями потоков данных QDataStream: жесткое кодирование номера версии, запись и чтение номера версии в явном виде и использование различных жестко закодированных номеров версий в зависимости от версии приложения. Можно применять любую из этих политик для гарантирования чтения данных, записанных в старой версии, новой версией приложения, даже если сборка новой версии приложения выполняется с более свежей версией Qt. После выбора политики обработки версий QDataStream чтение и запись двоичных данных в Qt становятся простыми и надежными.

Если мы хотим выполнить чтение или запись за один шаг, мы не должны использовать QDataStream и вместо этого мы должны вызывать функции write() и readAll() класса QIODevice. Например:

```

bool copyFile(const QString &source, const QString &dest)
{
    QFile sourceFile(source);
    if (!sourceFile.open(QIODevice::ReadOnly))
        return false;

    QFile destFile(dest);
    if (!destFile.open(QIODevice::WriteOnly))
        return false;

    destFile.write(sourceFile.readAll());

    return sourceFile.error() == QFile::NoError
        && destFile.error() == QFile::NoError;
}

```

В строке, где вызывается readAll(), все содержимое входного файла считывается в QByteArray, который затем передается функции write() для записи в выходной файл. Хранение всех данных в QByteArray ведет к большому расходу памяти, чем при последовательном чтении элементов, однако это дает некоторые

преимущества. Например, мы можем затем использовать функции `qCompress()` и `qUncompress()` для упаковки и распаковки данных. Альтернативой использованию `qCompress()` и `qUncompress()`, требующей меньше памяти, является класс `QtIOCompressor` от Qt Solutions. Класс `QtIOCompressor` сжимает записываемый поток и разжимает его при чтении, не храня весь файл в памяти.

Существуют другие сценарии, при которых прямой доступ к `QIODevice` оказывается более подходящим, чем использование `QDataStream`. Класс `QIODevice` имеет функцию `peek()`, которая возвращает следующие байты данных, перемещая позицию устройства, а также функцию `ungetChar()`, которая возвращает считанный байт в поток. Эти функции работают как на устройствах произвольного доступа (таких, как файлы) так и на последовательных устройствах (таких, как сетевые сокет). Имеется также функция `seek()`, которая используется для установки позиции устройств, поддерживающих произвольный доступ.

Двоичные форматы файлов являются наиболее универсальным и компактным средством хранения данных, а `QDataStream` позволяет легко получить доступ к двоичным данным. Кроме примеров в данном разделе, мы уже видели в главе 4, как `QDataStream` применяется для чтения и записи файлов в приложении Электронная таблица, и мы снова встретим этот класс в главе 21, где он будет использоваться для чтения и записи файлов курсоров в системе Windows.

Чтение и запись текста

Хотя двоичные форматы файлов обычно более компактные, чем текстовые форматы, они плохо воспринимаются человеком и не могут им редактироваться. Там, где последнее играет важную роль, можно использовать текстовые форматы. Qt предоставляет класс `QTextStream` для чтения и записи простых текстовых файлов или файлов других текстовых форматов, например, HTML, XML и файлов исходных текстов программ. Мы рассматриваем работу с XML-файлами отдельно в главе 16.

`QTextStream` обеспечивает преобразование между Unicode и локальной кодировкой системы или любой другой кодировкой, и незаметно для пользователя справляется с различными соглашениями относительно окончаний строк, принятыми в разных операционных системах («\r\n» в Windows, «\n» в Unix и Mac OS X). `QTextStream` использует 16-битовый тип `QChar` в качестве основного элемента данных. Кроме символов и строк `QTextStream` поддерживает основные числовые типы C++, преобразуя их в строку и обратно. Например, в следующем фрагменте программного кода выполняется запись строки Thomas M. Disch: 334\n в файл `sf-book.txt`:

```
QFile file("sf-book.txt");
if (!file.open(QIODevice::WriteOnly)) {
    std::cerr << "Cannot open file for writing: "
               << qPrintable(file.errorString()) << std::endl;
    return;
}

QTextStream out(&file);
out << "Thomas M. Disch: " << 334 << endl;
```

Записать текст очень просто, однако его чтение может оказаться трудной задачей, поскольку текстовый формат данных (в отличие от двоичного формата данных, записанных с помощью `QDataStream`) в принципе двусмысленный. Давайте рассмотрим следующий пример:

```
out << "Denmark" << "Norway";
```

Если `out` является объектом типа `QTextStream`, то данные в действительности записываются в виде строки «DenmarkNorway». Мы не можем рассчитывать на то, что приведенная ниже строка правильно считывает данные:

```
in >> str1 >> str2;
```

Фактически, произойдет то, что строка `str1` получит все слово «DenmarkNorway», а строка `str2` ничего не получит. При использовании класса `QDataStream` не возникнет таких трудностей, поскольку он сохраняет длину каждой строки в начале символьных данных.

Для сложных форматов файлов может потребоваться полнофункциональный парсер. Такой парсер мог бы считывать символ за символом при помощи оператора `>>` для типа `QChar`, или строку за строкой при помощи функции `QTextStream::readLine()`. В конце этого раздела мы представим два небольших примера, в одном из которых входной файл считывается построчно, а в другом он считывается посимвольно. Для того чтобы использовать парсеры, работающие с целым текстом, мы могли бы считать весь файл за один шаг, используя функцию `QTextStream::readAll()`, если бы нас не волновал расход памяти, или если бы мы знали, что файл будет небольшим.

По умолчанию `QTextStream` использует локальную кодировку системы (например, ISO 8859-1 или ISO 8859-15 в Америке и в большей части Европы) при чтении и записи. Это можно изменить, используя функцию `setCodec()`:

```
stream.setCodec("UTF-8");
```

В этом примере используется кодировка UTF-8, совместимая с популярной кодировкой ASCII и позволяющая представить весь набор символов Unicode. Дополнительная информация о кодировке Unicode и о поддержке кодировок классом `QTextStream` приводится в главе 18.

`QTextStream` имеет различные опции, аналогичные опциям `<iostream>`. Установить опции можно было путем передачи в поток специальных объектов – манипуляторов потока.

В следующем примере устанавливаются опции `showbase`, `uppercase` и `hex` перед выводом целого числа 12345678, и в результате получается текст «0xBC614E»:

```
out << showbase << uppercase << hex << 12345678;
```

Опции можно также устанавливать с помощью функций-членов:

```
out.setNumberFlags(QTextStream::ShowBase  
                  | QTextStream::UppercaseDigits);  
out.setIntegerBase(16);  
out << 12345678;
```


setIntegerBase(int)	
0	Основание обнаруживается автоматически по префиксу (при чтении)
2	Двоичное представление
8	Восьмеричное представление
10	Десятичное представление
16	Шестнадцатеричное представление
setNumberFlags(NumberFlags)	
ShowBase	Показывать префикс для оснований 2 ("0b"), 8 ("0") или 16 ("0x")
ForceSign	Всегда показывать знак перед числами
ForcePoint	Всегда показывать десятичную точку
UppercaseBase	Префиксы оснований выдавать на верхнем регистре
UppercaseDigits	Буквы шестнадцатеричных чисел выдавать на верхнем регистре
setRealNumberNotation(RealNumberNotation)	
FixedNotation	Формат с фиксированной точкой (например, 0.000123)
ScientificNotation	Научный формат (например, 0.12345678e-04)
SmartNotation	Формат с фиксированной точкой или научный формат в зависимости от того, какой из них компактнее
setRealNumberPrecision(int)	
Устанавливает максимальное количество генерируемых цифр (по умолчанию 6)	
setFieldWidth (int)	
Устанавливает минимальный размер поля	
setFieldAlignment (FieldAlignment)	
AlignLeft	Выравнивание влево - заполнитель занимает правую часть поля
AlignRight	Выравнивание вправо - заполнитель занимает левую часть поля
AlignCenter	Выравнивание по центру - заполнитель занимает оба края поля
AlignAccountingStyle	Заполнитель занимает область между знаком и числом
setPadChar(QChar)	
Устанавливает символ, используемый в качестве заполнителя (пробел по умолчанию)	

Рис. 12.1. Функции, устанавливающие опции для QTextStream

Класс QTextStream, как и QDataStream, работает с каким-нибудь подклассом QIODevice: QFile, QTemporaryFile, QBuffer, QProcess, QTcpSocket, QUdpSocket или QSslSocket. Кроме того, его можно использовать непосредственно со строкой типа QString. Например:

```
QString str;
QTextStream(&str) << oct << 31 << " " << dec << 25 << endl;
```

В результате переменная str будет иметь значение «37 25\ n», поскольку десятичное число 31 представляется восьмеричным числом 37. В данном случае не требуется устанавливать кодировку, поскольку QString всегда использует Unicode.

Теперь рассмотрим простой пример текстового формата файлов. В приложении Электронная таблица, описанном в части I, мы использовали двоичный формат для хранения данных этого приложения. Данные представляют собой последовательность троек (строка, столбец, формула) – по одной на каждую не-

пустую ячейку. Запись данных в виде текста выполняется просто; ниже показан фрагмент пересмотренной версии функции Spreadsheet::writeFile():

```
QTextStream out(&file);
for (int row = 0; row < RowCount; ++row) {
    for (int column = 0; column < ColumnCount; ++column) {
        QString str = formula(row, column);
        if (!str.isEmpty())
            out << row << " " << column << " " << str << endl;
    }
}
```

Мы использовали простой формат, когда одна строка соответствует одной ячейке, причем пробелы разделяют номер строки и номер столбца, а также номер столбца и формулу. Формула может содержать пробелы, но мы предполагаем, что она не может содержать ни одного символа «\n» (который используется для завершения строки). Теперь давайте рассмотрим соответствующий программный код, предназначенный для чтения файла:

```
QTextStream in(&file);
while (!in.atEnd()) {
    QString line = in.readLine();
    QStringList fields = line.split(' ');
    if (fields.size() >= 3) {
        int row = fields.takeFirst().toInt();
        int column = fields.takeFirst().toInt();
        setFormula(row, column, fields.join(' '));
    }
}
```

Мы считываем одним оператором одну строку данных приложения Электронная таблица. Функция `readLine()` удаляет завершающий символ '\n'. Функция `QString::split()` возвращает список строк, разбивая строку на части согласно обнаруженным символам-разделителям. Например, при обработке строки «5 19 Total value» будет получен список из четырех элементов [«5», «19», «Total», «value»].

Данные могут быть извлечены, если имеется, по крайней мере, три поля. Функция `QStringList::takeFirst()` удаляет первый элемент списка и возвращает удаленный элемент. Мы используем ее для извлечения номеров строк и столбцов. Мы не делаем никакой проверки ошибок; если считываемый номер строки или номер столбца оказывается не числом, функция `QString::toInt()` возвратит 0. Вызывая функцию `setFormula()`, мы помещаем оставшиеся поля в одну строку.

В нашем втором примере с `QTextStream` мы будем посимвольно считывать текстовый файл и затем выводить этот же текст, удаляя из строки завершающие пробелы и заменяя символы табуляции пробелами. Функция `tidyFile()` делает всю эту работу:

```
void tidyFile(QIODevice *inDevice, QIODevice *outDevice)
{
    QTextStream in(inDevice);
    QTextStream out(outDevice);
```

```
const int TabSize = 8;
int endlCount = 0;
int spaceCount = 0;
int column = 0;
QChar ch;

while (!in.atEnd()) {
    in >> ch;

    if (ch == '\n') {
        ++endlCount;
        spaceCount = 0;
        column = 0;
    } else if (ch == '\t') {
        int size = TabSize - (column % TabSize);
        spaceCount += size;
        column += size;
    } else if (ch == ' ') {
        ++spaceCount;
        ++column;
    } else {
        while (endlCount > 0) {
            out << endl;
            --endlCount;
            column = 0;
        }
        while (spaceCount > 0) {
            out << ' ';
            --spaceCount;
            ++column;
        }
        out << ch;
        ++column;
    }
}
out << endl;
}
```

Мы создаем для ввода и вывода данных объекты `QTextStream`, полученные на базе устройств `QIODevice`, переданных конструктору. Помимо текущего символа, мы поддерживаем три переменные контроля состояния: счетчик новых строк, счетчик пробелов и текущую позицию столбца в текущей строке (для преобразования символов табуляции в правильное количество пробелов).

Синтаксический анализ выполняется в цикле `while`, на каждом шаге которого считывается из входного файла один символ. В этой функции в некоторых местах делаются тонкие вещи. Например, хотя `TabSize` устанавливается на значение 8, мы заменяем символы табуляции достаточно точным числом пробелов, чтобы

достигнуть следующей метки табуляции, а не грубо заменять каждый символ табуляции восемью пробелами. При встрече символа новой строки, символа табуляции и пробелов мы просто обновляем состояние данных. Только при получении символа нового вида мы выполняем вывод данных, а перед записью символа записываем ожидающие вывода символы новой строки и пробелы (чтобы учесть пробельные строки и сохранить отступы) и обновляем состояние.

```
int main()
{
    QFile inFile;
    QFile outFile;

    inFile.open(stdin, QFile::ReadOnly);
    outFile.open(stdout, QFile::WriteOnly);

    tidyFile(&inFile, &outFile);

    return 0;
}
```

В этом примере не нужен объект `QApplication`, потому что мы используем только инструментальные классы Qt. Список всех инструментальных классов приводится на веб-странице <http://doc.trolltech.com/4.3/tools.html>. Мы предполагаем, что эта программа используется как фильтр, например:

```
tidy < cool.cpp > cooler.cpp
```

Эту программу можно легко расширить, позволяя ей работать с именами файлов, указанными в командной строке, если они заданы, а в противном случае использовать ее для фильтрации потока ввода `cin` в поток вывода `cout`.

Поскольку это приложение консольное, его файл `.pro` немного отличается от используемого нами в приложениях с графическим интерфейсом:

```
TEMPLATE       = app
QT             = core
CONFIG         += console
CONFIG         -= app_bundle
SOURCES        = tidy.cpp
```

Мы собираем приложение только с `QtCore`, поскольку здесь не используется функциональность графического пользовательского интерфейса. Затем мы указываем, что необходимо включить консольный вывод в Windows и не нужно размещать приложение в каталоге (bundle) приложений системы Mac OS X.

При чтении и записи простых ASCII-файлов и файлов с кодировкой ISO 8859-1 (Latin-1) можно непосредственно использовать программный интерфейс `QIODevice` вместо класса `QTextStream`. Поступать так имеет смысл только в редких случаях, поскольку в большинстве приложениях требуется в некоторых случаях поддержка других кодировок, и только `QTextStream` обеспечивает такую поддержку безболезненно. Если вы все-таки хотите писать текст непосредственно на устройство `QIODevice`, необходимо явно указать флажок `QIODevice::Text` в функции `open()`, например:

```
file.open(QIODevice::WriteOnly | QIODevice::Text);
```

Этот флажок говорит устройству `QIODevice` о том, что при записи в системе Windows необходимо преобразовывать символы «`\n`» в последовательность «`\r\n`». При чтении он говорит устройству, что необходимо игнорировать символы «`\r`» при работе на любой платформе. Теперь можно рассчитывать на то, что конец каждой строки обозначается символом новой строки «`\n`» вне зависимости от принятых на этот счет соглашений в операционной системе.

Работа с каталогами

Класс `QDir` обеспечивает независимые от платформы средства работы с каталогами и получение информации о файлах. Для демонстрации способов применения класса `QDir` мы напишем небольшое консольное приложение, которое подсчитывает размер дискового пространства, занимаемого всеми изображениями в указанном каталоге во всех его подкаталогах, вне зависимости от глубины их расположения.

Основу приложения составляет функция `imageSpace()`, которая рекурсивно подсчитывает общий размер изображений в заданном каталоге:

```
qulonglong imageSpace(const QString &path)
{
    QDir dir(path);
    qulonglong size = 0;

    QStringList filters;
    foreach (QByteArray format, QImageReader::supportedImageFormats())
        filters += ".*" + format;

    foreach (QString file, dir.entryList(filters, QDir::Files))
        size += QFileInfo(dir, file).size();

    foreach (QString subDir, dir.entryList(QDir::Dirs
                                           | QDir::NoDotAndDotDot))
        size += imageSpace(path + QDir::separator() + subDir);

    return size;
}
```

Мы начнем с создания объекта `QDir` для заданного пути, который может задаваться относительно текущего каталога или в виде полного пути. Мы передаем функции `entryList()` два аргумента. Первый аргумент содержит список фильтров имен файлов, разделенных пробелами. Шаблоны этих фильтров могут содержать символы «`*`» и «`?`». В этом примере мы применяем фильтры для включения только тех файлов, которые может считывать `QImage`. Вторым аргументом задает тип нужных нам элементов (обычные файлы, каталоги, дисководы и так далее).

Мы выполняем цикл по списку файлов, подсчитывая их совокупный размер. Класс `QFileInfo` позволяет нам осуществлять доступ к таким атрибутам файлов, как их размер, права доступа, владелец и время создания, изменения и последнего доступа.

Второй вызов функции `entryList()` получает все подкаталоги данного каталога. Мы выполняем цикл по ним (исключая «.» и «..») и рекурсивно вызываем функцию `imageSpace()` для получения совокупного размера изображений.

Для образования пути к каждому подкаталогу мы к текущему каталогу подключаем имя подкаталога (`*it`), разделяя их слешем. Класс `QDir` использует символ «/» в качестве разделителя каталогов на всех платформах и распознает символ «\» в системе Windows. Представляя пути пользователю, мы можем вызвать статическую функцию `QDir::toNativeSeparators()` для преобразования слешей в соответствующий разделитель конкретной платформы.

Давайте добавим функцию `main()` в нашу небольшую программу:

```
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    QStringList args = QCoreApplication::arguments();

    QString path = QDir::currentPath();
    if (args.count() > 1)
        path = args[1];

    std::cout << "Space used by images in " << qPrintable(path)
              << " and its subdirectories is " << (imageSpace(path) / 1024)
              << " KB" << std::endl;

    return 0;
}
```

Мы используем функцию `QDir::currentPath()` для получения пути текущего каталога. Мы могли бы поступить по-другому и использовать функцию `QDir::homePath()` для получения домашнего каталога пользователя. Если пользователь указал путь в командной строке, мы используем именно его. Наконец, мы вызываем нашу функцию `imageSpace()` для расчета размера пространства, занимаемого изображениями.

Класс `QDir` содержит и другие функции для работы с файлами и каталогами, включая `entryInfoList()` (которая возвращает список объектов `QFileInfo`), `rename()`, `exists()`, `mkdir()` и `rmdir()`. Класс `QFile` содержит несколько удобных статических функций, в том числе `remove()` и `exists()`. Класс `QFileSystemWatcher` может прислать нам уведомление об изменении в директории или в файле путем генерации сигналов `directoryChanged()` и `fileChanged()`.

Ресурсы, внедренные в исполняемый модуль

До сих пор в этой главе мы говорили о доступе к данным, которые находятся на внешних устройствах, но в Qt можно также внедрять двоичные данные или текст в исполняемый модуль приложения. Это обеспечивается ресурсной системой Qt. В других главах мы использовали файлы ресурсов для внедрения файлов изображений в исполняемый модуль, однако внедрять можно любой файл. Читать внедренные файлы можно с использованием `QFile`, как будто это обычные файлы, расположенные в файловой системе.

Ресурсы преобразуются в программный код C++ ресурсным компилятором Qt (rcc). Мы можем указать qmake, что необходимо включить специальные правила для выполнения rcc, добавляя следующую строку в файл .pro:

```
RESOURCES = myresourcefile.qrc
```

Файл myresourcefile.qrc – это XML-файл, который содержит список файлов, внедренных в исполняемый модуль.

Допустим, создается приложение, которое сохраняет подробную контактную информацию. Ради удобства пользователей мы хотим внедрить международные телефонные коды в исполняемый модуль. Если файл находится в подкаталоге datafiles каталога сборки приложения, файл ресурсов может выглядеть следующим образом:

```
<RCC>
<qresource>
  <file>datafiles/phone-codes.dat</file>
</qresource>
</RCC>
```

В приложении ресурсы опознаются по префиксу пути :/. В этом примере файл телефонных кодов имеет путь :/datafiles/phone-codes.dat и может быть считан как любой другой файл, с использованием QFile.

Преимуществом внедрения данных в исполняемый модуль является невозможность их потери и возможность создания действительно автономных исполняемых модулей (если использовалась статическая компоновка). Двумя недостатками является необходимость замены всего исполняемого модуля при изменении внедренных данных и увеличение размера исполняемого модуля из-за дополнительного расхода памяти под внедренные данные.

Ресурсная система Qt обладает дополнительными возможностями, которые не представлены в этом примере, включая поддержку псевдонимов файлов и локализацию. Информацию по этим возможностям можно найти на веб-странице <http://doc.trolltech.com/4.3/resources.html>.

Связь между процессами

Класс QProcess позволяет выполнять внешние программы и взаимодействовать с ними. Этот класс работает асинхронно и в фоновом режиме, из-за чего интерфейс пользователя по-прежнему будет реагировать на действия пользователя. QProcess посылает сигналы, уведомляющие нас о получении данных или о завершении работы.

Мы кратко рассмотрим программный код небольшого приложения, обеспечивающего интерфейс пользователя для внешней программы преобразования изображений. В нашем случае мы используем программу convert из пакета программ ImageMagick, который свободно распространяется на всех основных платформах. Наш пользовательский интерфейс показан на рис. 12.2.

Интерфейс пользователя приложения Image Converter (конвертор изображений) был создан при помощи Qt Designer. Файл .ui находится на компакт-диске, который входит в состав данной книги. Здесь мы основное внимание уделим подклассу, который является наследником сгенерированного компилятором uic класса Ui::ConvertDialog, и начнем с заголовочного файла:

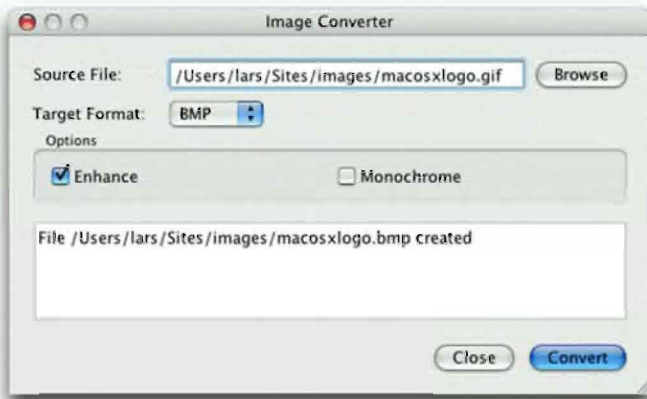


Рис. 12.2. Приложение Image Converter

```

#ifndef CONVERTDIALOG_H
#define CONVERTDIALOG_H

#include <QDialog>
#include <QProcess>

#include "ui_convertdialog.h"

class ConvertDialog : public QDialog, private Ui::ConvertDialog
{
    Q_OBJECT

public:
    ConvertDialog(QWidget *parent = 0);

private slots:
    void on_browseButton_clicked();
    void convertImage();
    void updateOutputTextEdit();
    void processFinished(int exitCode, QProcess::ExitStatus exitStatus);
    void processError(QProcess::ProcessError error);

private:
    QProcess process;
    QString targetFile;
};
#endif

```

Этот заголовочный файл создается по тому знакомому образцу, который используется в подклассах форм *Qt Designer*. Небольшим отличием от других примеров, которые мы видели, является то, что мы использовали закрытое (private) наследование от класса `Ui::ConvertDialog`. Это предотвращает доступ к виджетам

формы из-за пределов функций формы. Благодаря механизму автоматического связывания *Qt Designer*, слот `on_browseButton_clicked()` автоматически связывается с сигналом `clicked()` кнопки **Browse** (просмотреть).

```
ConvertDialog::ConvertDialog(QWidget *parent)
    : QDialog(parent)
{
    setupUi(this);
    QPushButton *convertButton =
        buttonBox->button(QDialogButtonBox::Ok);
    convertButton->setText(tr("&Convert"));
    convertButton->setEnabled(false);
    connect(convertButton, SIGNAL(clicked()),
            this, SLOT(convertImage()));
    connect(buttonBox, SIGNAL(rejected()), this, SLOT(reject()));
    connect(&process, SIGNAL(readyReadStandardError()),
            this, SLOT(updateOutputTextEdit()));
    connect(&process, SIGNAL(finished(int, QProcess::ExitStatus)),
            this, SLOT(processFinished(int, QProcess::ExitStatus)));
    connect(&process, SIGNAL(error(QProcess::ProcessError)),
            this, SLOT(processError(QProcess::ProcessError)));
}
```

Вызов `setupUi()` создает и компоует все виджеты форм и устанавливает соединения сигнал-слот для слота `browseButton_clicked()`. Мы получаем указатель на кнопку ОК поля кнопок и задаем для нее более подходящий текст. Также мы делаем ее недоступной, поскольку исходное изображение для преобразования отсутствует, а также связываем ее со слотом `convertImage()`. Далее мы соединяем сигнал `rejected()` поля кнопок (генерируемый кнопкой Close) со слотом `reject()` диалогового окна. После этого мы вручную связываем три сигнала объекта `QProcess` с тремя закрытыми слотами. Любые сообщения внешнего процесса для потока `cerr` мы будем обрабатывать в функции `updateOutputTextEdit()`.

```
void ConvertDialog::on_browseButton_clicked ()
{
    QString initialName = sourceFileEdit->text();
    if (initialName.isEmpty())
        initialName = QDir::homePath();
    QString fileName =
        QFileDialog::getOpenFileName(this, tr("Choose File"),
                                    initialName);
    fileName = QDir::toNativeSeparators(fileName);
    if (!fileName.isEmpty()) {
        sourceFileEdit->setText(fileName);
        buttonBox->button(QDialogButtonBox::Ok)->setEnabled(true);
    }
}
```

Сигнал `clicked()` кнопки **Browse (просмотреть)** автоматически связывается в функции `setUpUi()` со слотом `on_browseButton_clicked()`. Если пользователь ранее выбирал какой-нибудь файл, мы инициализируем диалоговое окно выбора файла именем этого файла; в противном случае мы используем домашний каталог пользователя.

```
void ConvertDialog::convertImage()
{
    QString sourceFile = sourceFileEdit->text();
    targetFile = QFileInfo(sourceFile).path() + QDir::separator()
                + QFileInfo(sourceFile).baseName() + "."
                + targetFormatComboBox->currentText().toLower();
    buttonBox->button(QDialogButtonBox::Ok)->setEnabled(false);
    outputTextEdit->clear();

    QStringList args;
    if (enhanceCheckBox->isChecked())
        args << "-enhance";
    if (monochromeCheckBox->isChecked())
        args << "-monochrome";
    args << sourceFile << targetFile;

    process.start("convert", args);
}
```

Когда пользователь нажимает кнопку **Convert (преобразовать)**, мы копируем имя исходного файла и изменяем его расширение в соответствии с новым форматом файла. Мы используем зависимый от платформы разделитель каталогов («/» или «\» возвращается функцией `QDir::separator()`) вместо жесткого кодирования этих символов, поскольку пользователь будет видеть имя файла.

Затем отключаем кнопку **Convert**, чтобы пользователь не мог случайно запустить одновременно несколько процессов преобразования, и очищаем поле текстового редактора, используемое нами для отображения информации о состоянии.

Для инициирования внешнего процесса мы вызываем функцию `QProcess::start()` с именем программы, которая должна выполняться (`convert`), и всеми ее аргументами. В данном случае мы передаем флажки `-enhance` и `-monochrome`, если пользователь выбрал соответствующие опции, и затем имена исходного и целевого файлов. Тип выполняемого преобразования программа `convert` определяет по расширениям файлов.

```
void ConvertDialog::updateOutputTextEdit()
{
    QByteArray newData = process.readAllStandardError();
    QString text = outputTextEdit->toPlainText()
                  + QString::fromLocal8Bit(newData);
    outputTextEdit->setPlainText(text);
}
```

При всякой записи внешним процессом в поток `cerr` вызывается слот `updateOutputTextEdit()`. Мы считываем текст сообщения об ошибке и добавляем его в существующий текст `QTextEdit`.

```
void ConvertDialog::processFinished(int exitCode,
                                   QProcess::ExitStatus exitStatus)
{
    if (exitStatus == QProcess::CrashExit) {
        outputTextEdit->append(tr("Conversion program crashed"));
    } else if (exitCode != 0) {
        outputTextEdit->append(tr("Conversion failed"));
    } else {
        outputTextEdit->append(tr("File %1 created").arg(targetFile));
    }
    buttonBox->button(QDialogButtonBox::Ok)->setEnabled(true);
}
```

По окончании процесса мы уведомляем пользователя о результате и включаем кнопку Convert.

```
void ConvertDialog::processError(QProcess::ProcessError error)
{
    if (error == QProcess::FailedToStart) {
        outputTextEdit->append(tr("Conversion program not found"));
        buttonBox->button(QDialogButtonBox::Ok)->setEnabled(true);
    }
}
```

Если процесс не удастся запустить, QProcess генерирует сигнал `error()` вместо `finished()`. Мы выдаем сообщение об ошибке и включаем кнопку Convert.

В этом примере преобразования файлов выполнялись асинхронно, т. е. QProcess запускал программу `convert` и сразу же возвращал управление приложению. Это сохраняет работоспособность пользовательского интерфейса во время выполнения преобразований в фоновом режиме. Но в некоторых ситуациях необходимо, чтобы внешний процесс завершился, и только после этого мы сможем идти дальше в нашем приложении; в таких случаях требуется синхронная работа QProcess.

Одним из распространенных примеров, где желателен синхронный режим работы, является приложение, обеспечивающее редактирование простых текстов с применением текстового редактора, предпочитаемого пользователем. Такое приложение реализуется достаточно просто с помощью QProcess. Например, пусть в QTextEdit содержится простой текст, и имеется кнопка Edit, при нажатии на которую выполняется слот `edit()`.

```
void ExternalEditor::edit()
{
    QTemporaryFile outFile;
    if (!outFile.open())
        return;

    QString fileName = outFile.fileName();
    QTextStream out(&outFile);
    out << textEdit->toPlainText();
    outFile.close();
}
```

```

QProcess::execute(editor, QStringList() << options << fileName);

QFile inFile(fileName);
if (!inFile.open(QIODevice::ReadOnly))
    return;

QTextStream in(&inFile);
textEdit->setPlainText(in.readAll());
}

```

Мы используем `QTemporaryFile` для создания пустого файла с уникальным именем. Мы не задаем аргументы функции `QTemporaryFile::open()`, поскольку для нас подходит ее режим по умолчанию, по которому файл открывается для чтения и записи. Мы записываем содержимое поля редактирования во временный файл и затем закрываем файл, потому что некоторые текстовые редакторы не могут работать с уже открытыми файлами.

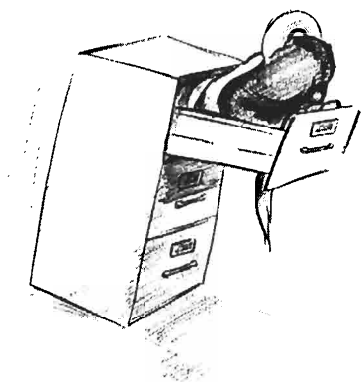
Статическая функция `QProcess::execute()` запускает внешний процесс и блокирует работу приложения до завершения процесса. Аргумент `editor` в строке типа `QString` содержит имя исполняемого модуля редактора (например, «gvim»). Аргумент `options` является списком `QStringList` (который содержит один элемент, «-f», если мы используем `gvim`).

После закрытия пользователем текстового редактора процесс завершает свою работу и функция `execute()` возвращает управление. Затем мы открываем временный файл и считываем его содержимое в `QTextEdit`. `QTemporaryFile` автоматически удаляет временный файл, когда объект выходит из области видимости.

При синхронной работе `QProcess` нет необходимости устанавливать соединения сигнал-слот. Если требуется более тонкое управление, чем то, которое обеспечивает статическая функция `execute()`, мы можем использовать альтернативный подход. Это означает создание объекта `QProcess` и вызов для него функции `start()` с последующей установкой блокировки путем вызова функции `QProcess::waitForStarted()`, после успешного завершения которой вызывается функция `QProcess::waitForFinished()`. Пример применения этого подхода можно найти в справочной документации по классу `QProcess`.

В данном разделе мы использовали `QProcess`, чтобы получить доступ к уже существующей функциональности. Применение уже имеющегося приложения может сократить время разработки и избавить нас от лишних деталей, которые играют второстепенную роль при достижении главной цели нашего приложения. Другой способ получения доступа к уже существующей функциональности заключается в компоновке приложения с соответствующей библиотекой. Но если нет подходящей библиотеки, хорошим решением может быть запуск консольного приложения с помощью `QProcess`.

`QProcess` может также применяться для запуска других приложений с графическим пользовательским интерфейсом. Однако, если нашей целью является связь между приложениями, а не просто запуск одного из другого, то лучше установить прямую связь между приложениями, используя Qt-классы, предназначенные для работы с сетью, или расширение `ActiveQt` для Windows. Если же нам нужно загрузить предпочитаемый пользователем веб-браузер, мы можем просто вызвать функцию `QDesktopServices::openUrl()`.



- *Соединение с базой данных и выполнение запросов*
- *Просмотр таблиц*
- *Редактирование записей с помощью форм*
- *Представление данных в табличной форме*

Глава 13. Базы данных

Модуль *QtSql* средств разработки Qt обеспечивает независимый от платформы и типа базы данных интерфейс для доступа с помощью языка SQL к базам данных. Этот интерфейс поддерживается набором классов, использующих архитектуру Qt модель/представление для интеграции средств доступа к базам данных с интерфейсом пользователя. Эта глава предполагает знакомство с Qt-классами архитектуры модель/представление, рассмотренными в главе 10.

Связь с базой данных обеспечивается объектом *QSqlDatabase*. Qt использует драйверы для связи с программным интерфейсом различных баз данных. Версия Qt для настольных компьютеров (Qt Desktop Edition) включает в себя следующие драйверы:

Драйвер	База данных
QDB2	IBM DB2 версии 7.1 и выше
QIBASE	InterBase компании Borland
QMYSQL	MySQL
QOCI	Oracle (Oracle Call Interface - интерфейс вызовов Oracle)
QODBC	ODBC (включает Microsoft SQL Server)
QPSQL	PostgreSQL версий 3 и выше
QSQLITE	SQLite версии 3
QSQLITE2	SQLite версии 2
QTDS	Sybase Adaptive Server

Из-за лицензионных ограничений не все драйверы входят в состав издания Qt с открытым исходным кодом (Qt Open Source Edition). При настройке конфигурации Qt драйверы SQL можно либо непосредственно включить в состав Qt, либо использовать как подключаемые модули (plugins). Qt поставляется вместе с SQLite – общедоступной, не нуждающейся в сервере базой данных¹.

¹ Поддержка SQL должна быть включена при построении Qt. Например, Qt можно скомпилировать со встроенной поддержкой SQLite путем указания опции командной строки `-qt-sql-sqlite` в конфигурационном скрипте или путем задания соответствующей опции в инсталляторе Qt.

Для пользователей, хорошо знакомых с синтаксисом SQL, класс `QSqlQuery` предоставляет средства, позволяющие непосредственно выполнять произвольные команды SQL и обрабатывать их результаты. Для пользователей, предпочитающих иметь дело с высокоуровневым интерфейсом базы данных, который не требует знания синтаксиса SQL, классы `QSqlTableModel` и `QSqlRelationalTableModel` являются подходящими абстракциями. Эти классы представляют таблицы SQL в том же виде, как и классы других моделей Qt (рассмотренных в главе 10). Они могут использоваться самостоятельно для кодирования в программе просмотра и редактирования данных или могут подключаться к представлениям, с помощью которых конечные пользователи будут сами просматривать и редактировать данные.

Qt также позволяет легко программировать такие распространенные идиомы баз данных, как отображение зависимых представлений для записей, связанных отношением «главные-подчиненные» (master-detail), возможность многократной детализации выводимых на экран данных (drill-down) и просмотр таблиц базы данных с помощью форм или таблиц пользовательского интерфейса, что продемонстрируют примеры этой главы.

Соединение с базой данных и выполнение запросов

Для выполнения запросов SQL мы должны сначала установить соединение с базой данных. Обычно настройка соединений с базой данных выполняется отдельной функцией, которую мы вызываем при запуске приложения. Например:

```
bool createConnection()
{
    QSqlDatabase *db = QSqlDatabase::addDatabase("QOCI8");
    db->setHostName("mozart.konkordia.edu");
    db->setDatabaseName("musicdb");
    db->setUserName("gbatstone");
    db->setPassword("T17aV44");
    if (!db->open()) {
        db->lastError().showMessage();
        return false;
    }
    return true;
}
```

Во-первых, мы вызываем функцию `QSqlDatabase::addDatabase()` для создания объекта `QSqlDatabase`. Первый аргумент функции `addDatabase()` задает драйвер базы данных, который Qt должен использовать для доступа к базе данных. В данном случае мы используем MySQL.

Затем мы устанавливаем имя хоста базы данных, имя базы данных, имя пользователя и пароль, и мы открываем соединение. Если функция `open()` завершается неудачей, мы выводим сообщение об ошибке, используя `QSqlError::showMessage()`.

Обычно функцию `createConnection()` вызывают в `main()`:

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

```
    if (!createConnection())  
        return 1;  
    ...  
    return app.exec();  
}
```

После установки соединения мы можем применять QSqlQuery для выполнения любой инструкции SQL, поддерживаемой используемой базой данных. Например, ниже приводится пример выполнения команды SELECT:

```
QSqlQuery query;  
query.exec("SELECT title, year FROM cd WHERE year >= 1998");
```

После вызова функции exec(), мы можем посмотреть результат запроса:

```
while (query.next()) {  
    QString title = query.value(0).toString();  
    int year = query.value(1).toInt();  
    std::cerr << qPrintable(title) << ": " << year << std::endl;  
}
```

Мы вызываем функцию next() один раз для позиционирования QSqlQuery на *первую* запись полученного набора. Последующие вызовы next() продвигают указатель записи на одну позицию дальше, пока не будет достигнут конец, когда функция next() возвращает false. Если результирующий набор (result set) пустой (или запрос завершается неудачей), первый вызов функции next() возвратит false.

Функция value() возвращает значение поля, как QVariant. Поля пронумерованы начиная с 0 в порядке их указания в команде SELECT. Класс QVariant может содержать многие типы C++ и Qt, включая int и QString. Другие типы данных, которые могут храниться в базе данных, преобразуются в соответствующие типы C++ и Qt и хранятся в QVariant. Например, VARCHAR представляется в виде QString, а DATETIME — в виде QDateTime.

Класс QSqlQuery содержит некоторые другие функции для просмотра результирующего набора: first(), last(), previous() и seek(). Эти функции удобны, но для некоторых баз данных они могут выполняться медленнее и расходовать памяти больше, чем функция next(). При работе с большими наборами данных мы можем осуществить простую оптимизацию, вызывая функцию QSqlQuery::setForwardOnly(true) перед вызовом exec(), и только затем использовать next() для перемещения по результирующему набору.

Ранее мы задавали запрос SQL в аргументе функции QSqlQuery::exec(), но, кроме того, мы можем передавать его непосредственно конструктору, который сразу же выполнит его:

```
QSqlQuery query("SELECT title, year FROM cd WHERE year >= 1998");
```

Мы можем проверить наличие ошибки, вызывая функцию isActive() для запроса:

```
if (!query.isActive())
```

```
    QMessageBox::warning(this, tr("Database Error"),  
        query.lastError().text());
```

Если ошибки нет, запрос становится «активным», и мы можем использовать `next()` для перемещения по результирующему набору.

Выполнение команды `INSERT` осуществляется почти так же просто, как и команды `SELECT`:

```
QSqlQuery query("INSERT INTO cd (id, artistid, title, year) "
"VALUES (203, 102, 'Living in America', 2002)");
```

После этого функция `numRowsAffected()` возвращает количество строк, которые были изменены инструкцией `SQL` (или `-1`, если возникла ошибка).

Если нам необходимо вставлять много записей или если мы хотим избежать преобразования значений в строковые данные (и правильного преобразования специальных символов), мы можем использовать функцию `prepare()` для указания полей в шаблоне запроса и затем присваивания им необходимых нам значений. Qt поддерживает как стиль `Oracle`, так и стиль `ODBC` для всех баз данных, применяя, где возможно, «родной» интерфейс базы данных или имитируя его в противном случае. Ниже приводится пример, в котором используется синтаксис `Oracle` для представления поименованных полей:

```
QSqlQuery query;
query.prepare("INSERT INTO cd (id, artistid, title, year) "
"VALUES (:id, :artistid, :title, :year)");
query.bindValue(":id", 203);
query.bindValue(":artistid", 102);
query.bindValue(":title", "Living in America");
query.bindValue(":year", 2002);
query.exec();
```

Ниже приводится тот же пример позиционного представления полей в стиле `ODBC`:

```
QSqlQuery query;
query.prepare("INSERT INTO cd (id, artistid, title, year) "
"VALUES (?, ?, ?, ?)");
query.addBindValue(203);
query.addBindValue(102);
query.addBindValue("Living in America");
query.addBindValue(2002);
query.exec();
```

После вызова функции `exec()`, мы можем вызвать `bindValue()` или `addBindValue()` для присваивания новых значений, а затем снова вызвать `exec()` для выполнения запроса уже с новыми значениями.

Такие шаблоны часто используются для задания двоичных строковых данных, содержащих символы не в коде `ASCII` или `Latin-1`. Незаметно для пользователя Qt использует `Unicode` в тех базах данных, которые поддерживают `Unicode`, а в тех, которые не делают этого, Qt также незаметно для пользователя преобразует строковые данные в соответствующую кодировку.

Qt поддерживает `SQL-транзакции` в тех базах данных, где они предусмотрены. Для запуска транзакции мы вызываем функцию `transaction()` для объекта

QSqlDatabase, представляющего соединение с базой данных. Для завершения транзакции мы вызываем либо функцию `commit()`, либо функцию `rollback()`. Например, ниже показано, как мы можем найти внешний ключ (foreign key) и выполнить команду `INSERT` внутри транзакции:

```
QSqlDatabase::database().transaction();
QSqlQuery query;
query.exec("SELECT id FROM artist WHERE name = 'Gluecifer'");
if (query.next()) {
    int artistId = query.value(0).toInt();
    query.exec("INSERT INTO cd (id, artistid, title, year) "
              "VALUES (201, " + QString::number(artistId)
              + ", 'Riding the Tiger', 1997)");
}
QSqlDatabase::database().commit();
```

Функция `QSqlDatabase::database()` возвращает объект `QSqlDatabase`, представляющий соединение, созданное нами при вызове `createConnection()`. Если транзакция не может запуститься, функция `QSqlDatabase::transaction()` возвращает `false`. Некоторые базы данных не поддерживают транзакции. В этом случае функции `transaction()`, `commit()` и `rollback()` ничего не делают. Мы можем проверить возможность поддержки базой данных транзакций путем вызова функции `hasFeature()` для объекта `QSqlDriver`, связанного с базой данных:

```
QSqlDriver *driver = QSqlDatabase::database().driver();
if (driver->hasFeature(QSqlDriver::Transactions))
    ...
```

Можно проверить наличие в базе данных ряда других возможностей, включая поддержку объектов BLOB (binary large objects – большие двоичные объекты), Unicode и подготовленных запросов. Также существует возможность обращаться к дескриптору низкоуровневого драйвера базы данных и низкоуровневому дескриптору результирующего набора данных запроса с помощью функций `QSqlDriver::handle()` и `QSqlResult::handle()`. Однако обе эти функции опасно применять, если вы не знаете точно, что вы делаете, и применять их следует с осторожностью. Обращайтесь к соответствующей документации, где приводятся примеры и разъясняются риски.

В приводимых до сих пор примерах мы предполагали, что в приложении используется одно соединение с базой данных. Если мы хотим создать несколько соединений, мы можем передавать название соединения в качестве второго аргумента функции `addDatabase()`. Например:

```
QSqlDatabase *db = QSqlDatabase::addDatabase("QPSQL", "OTHER");
db.setHostName("saturn.mcmanamy.edu");
db.setDatabaseName("starsdb");
db.setUserName("hilbert");
db.setPassword("ixtapa7");
```

Мы можем затем получить указатель на объект `QSqlDatabase`, передавая название соединения функции `QSqlDatabase::database()`:

```
QSqlDatabase db = QSqlDatabase::database("OTHER");
```

Для выполнения запросов с другим соединением мы передаем объект `QSqlDatabase` конструктору `QSqlQuery`:

```
QSqlQuery query(db);
query.exec("SELECT id FROM artist WHERE name = 'Mando Diao'");
```

Несколько соединений полезны, если мы хотим выполнять одновременно несколько транзакций, поскольку каждое соединение может использоваться только для одной активной транзакции. Когда мы используем несколько соединений с базой данных, мы можем, все-таки, иметь одно непоименованное соединение и `QSqlQuery` будет использовать это соединение, если не указано поименованное соединение.

Кроме `QSqlQuery` Qt содержит класс `QSqlTableModel` – интерфейс высокого уровня, позволяя нам не использовать выражения SQL «в чистом виде» для выполнения наиболее распространенных SQL-команд (SELECT, INSERT, UPDATE и DELETE). Этот класс может использоваться автономно без какого-либо графического пользовательского интерфейса или в качестве источника данных для `QListView` или `QTableView`.

Ниже приводится пример использования `QSqlTableModel` для выполнения команды SELECT:

```
QSqlTableModel model;
model.setTable("cd");
model.setFilter("year >= 1998");
model.select();
```

Это эквивалентно запросу

```
SELECT * FROM cd WHERE year >= 1998
```

Просмотр результирующего набора выполняется путем получения заданной записи функцией `QSqlTableModel::record()` и доступа к отдельным полям с помощью функции `value()`:

```
for (int i = 0; i < model.rowCount(); ++i) {
    QSqlRecord record = model.record(i);
    QString title = record.value("title").toString();
    int year = record.value("year").toInt();
    std::cerr << qPrintable(title) << ": " << year << std::endl;
}
```

Функция `QSqlRecord::value()` принимает либо имя поля, либо индекс поля. При работе с большими наборами данных рекомендуется задавать поля с помощью их индексов. Например:

```
int titleIndex = model.record().indexOf("title");
int yearIndex = model.record().indexOf("year");
for (int i = 0; i < model.rowCount(); ++i) {
    QSqlRecord record = model.record(i);
    QString title = record.value(titleIndex).toString();
    int year = record.value(yearIndex).toInt();
    std::cerr << qPrintable(title) << ": " << year << std::endl;
}
```

Для вставки записи в таблицу базы данных мы вызываем функцию `insertRow()` для создания новой пустой строки (записи) и используем `setData()` для установки значения каждого столбца (поля записи):

```
QSqlTableModel model;
model.setTable("cd");
int row = 0;
model.insertRows(row, 1);
model.setData(model.index(row, 0), 113);
model.setData(model.index(row, 1), "Shanghai My Heart");
model.setData(model.index(row, 2), 224);
model.setData(model.index(row, 3), 2003);
model.submitAll();
```

После вызова `submitAll()` запись может быть перемещена в другую позицию, зависящую от упорядоченности таблицы. Вызов `submitAll()` возвратит `false`, если вставка окажется неудачной.

Важным отличием модели SQL от стандартной модели является необходимость вызова в модели SQL функции `submitAll()` для записи всех изменений в базу данных.

Для обновления записи мы должны сначала установить `QSqlTableModel` на запись, которую мы хотим модифицировать (например, используя функции `select()`). Затем мы извлекаем запись, обновляем соответствующие поля и записываем наши изменения обратно в базу данных:

```
QSqlTableModel model;
model.setTable("cd");
model.setFilter("id = 125");
model.select();
if (model.rowCount() == 1) {
    QSqlRecord record = model.record(0);
    record.setValue("title", "Melody A.M.");
    record.setValue("year", record.value("year").toInt() + 1);
    model.setRecord(0, record);
    model.submitAll();
}
```

Если имеется запись, удовлетворяющая заданному фильтру, доступ к ней мы получаем при помощи функции `QSqlTableModel::record()`. Мы осуществляем наши изменения и вновь записываем в базу данных запись с новыми значениями полей.

Кроме того, обновление можно выполнить при помощи функции `setData()`, как это делается для модели, отличной от SQL-модели. Для получения доступа к полям записи используются индексы модели с указанием номера строки (записи) и столбца (поля):

```
model.select();
if (model.rowCount() == 1) {
    model.setData(model.index(0, 1), "Melody A.M.");
    model.setData(model.index(0, 3),
        model.data(model.index(0, 3)).toInt() + 1);
    model.submitAll();
}
```

Удаление записи напоминает ее обновление:

```
model.setTable("cd");
model.setFilter("id = 125");
model.select();
if (model.rowCount() == 1) {
    model.removeRows(0, 1);
    model.submitAll();
}
```

В вызове `removeRows()` указывается номер строки первой удаляемой записи и количество удаляемых записей. В следующем примере удаляются все записи, удовлетворяющие фильтру:

```
model.setTable("cd");
model.setFilter("year < 1990");
model.select();
if (model.rowCount() > 0) {
    model.removeRows(0, model.rowCount());
    model.submitAll();
}
```

Классы `QSqlQuery` и `QSqlTableModel` обеспечивают интерфейс между Qt и базой данных SQL. Используя эти классы, можно создавать формы, представляющие данные пользователям и позволяющие им вставлять, обновлять и удалять записи.

Для проектов, в которых используются классы SQL, мы должны добавить строку

```
QT += sql
```

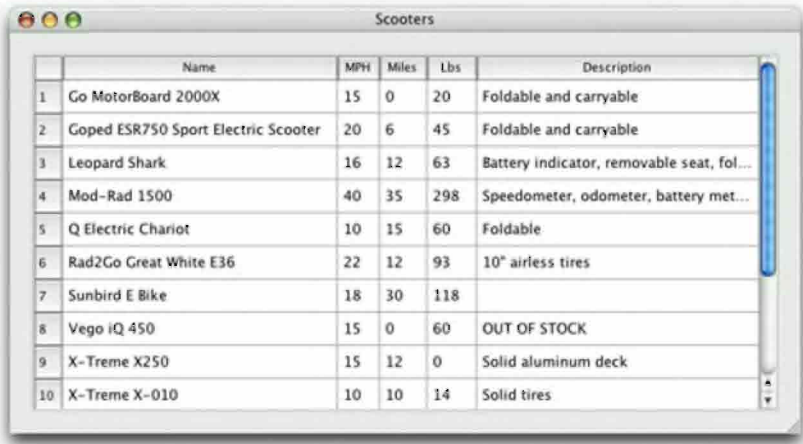
в соответствующий файл `.pro`. Это обеспечит связывание приложения с библиотекой `QtSql`.

Просмотр таблиц

В предыдущем разделе мы увидели, как осуществляется взаимодействие с базой данных с помощью классов `QSqlQuery` и `QSqlTableModel`. В этом разделе мы увидим, как представить модель `QSqlTableModel` в виджете `QTableView`. Приложение *Скутеры (Scooters)*, показанное на рис. 13.1, предлагает таблицу моделей скутеров. Данный пример основан на использовании одной таблицы `scooter`, определенной следующим образом:

```
CREATE TABLE scooter (
    id INTEGER PRIMARY KEY AUTOINCREMENT,
    name VARCHAR(40) NOT NULL,
    maxspeed INTEGER NOT NULL,
    maxrange INTEGER NOT NULL,
    weight INTEGER NOT NULL,
    description VARCHAR(80) NOT NULL);
```

Значения в поле `id` генерируются автоматически базой данных, в данном случае, `SQLite`. В других базах данных для этого может использоваться другой синтаксис.



	Name	MPH	Miles	Lbs	Description
1	Go MotorBoard 2000X	15	0	20	Foldable and carryable
2	Goped ESR750 Sport Electric Scooter	20	6	45	Foldable and carryable
3	Leopard Shark	16	12	63	Battery indicator, removable seat, fol...
4	Mod-Rad 1500	40	35	298	Speedometer, odometer, battery met...
5	Q Electric Chariot	10	15	60	Foldable
6	Rad2Go Great White E36	22	12	93	10" airless tires
7	Sunbird E Bike	18	30	118	
8	Vego iQ 450	15	0	60	OUT OF STOCK
9	X-Treme X250	15	12	0	Solid aluminum deck
10	X-Treme X-010	10	10	14	Solid tires

Рис. 13.1. Приложение Скутеры

Для простоты обслуживания мы используем перечислимое значение для присваивания индексам столбцов осмысленных имен:

```
enum {
    Scooter_MaxSpeed = 2,
    Scooter_MaxRange = 3,
    Scooter_Weight = 4,
    Scooter_Description = 5
};
```

Ниже приводится весь код, который необходим для того, чтобы модель QSqlTableModel отображала таблицу скутеров:

```
model = new QSqlTableModel(this);
model->setTable("artistscooter");
model->setSort(ArtistScooter_Name, Qt::AscendingOrder);
model->setHeaderData(ArtistScooter_Name, Qt::Horizontal, tr("Name"));
model->setHeaderData(Artist_CountryScooter_MaxSpeed, Qt::Horizontal, tr("CountryMPH"));
model->setHeaderData(Scooter_MaxRange, Qt::Horizontal, tr("Miles"));
model->setHeaderData(Scooter_Weight, Qt::Horizontal, tr("Lbs"));
model->setHeaderData(Scooter_Description, Qt::Horizontal, tr("Description"));
model->select();
```

Создание модели сходно с тем, что мы делали в предыдущем разделе. Единственное отличие в том, что мы указали собственные заголовки столбцов. Если бы мы этого не сделали, использовались бы необработанные имена столбцов. Мы также указали порядок сортировки при помощи функции setSort(); она внутренне реализована при помощи условия ORDER BY.

Теперь, когда мы создали модель и заполнили ее данными с помощью функции select(), мы можем создать для ее отображения представление:

```
view = new QTableView;
```

```
view->setModel(model);  
view->setSelectionMode(QAbstractItemView::SingleSelection);  
view->setSelectionBehavior(QAbstractItemView::SelectRows);  
view->setColumnHidden(Scooter_Id, true);  
view->resizeColumnsToContents();  
view->setEditTriggers(QAbstractItemView::NoEditTriggers);  
QHeaderView *header = view->horizontalHeader();  
header->setStretchLastSection(true);
```

В главе 10 мы увидели, как следует использовать данные из `QAbstractItemModel` в таблице. Поскольку модель `QSqlTableModel` происходит (косвенно) от `QAbstractItemModel`, мы легко можем использовать ее для соединения представления и модели. В оставшейся части кода модель лишь настраивается для удобства использования.

Режим выбора указывает на то, что пользователь может выбирать (если такие элементы есть). Здесь мы сделали отдельные ячейки (поля) выбираемыми. То, что ячейка выбрана, обычно отображается пунктирным контуром вокруг выбранной ячейки. Функционирование выделения (selection behavior) показывает, как визуально должно работать выделение. В данном случае – это выделение целых строк. Такое выделение обычно отображается иным цветом фона. Мы решили скрыть столбец ID, поскольку идентификатор не слишком много значит для пользователя. Мы также задали значение `NoEditTriggers`, чтобы представление таблицы было доступно только для чтения.

Альтернативой представлению, доступному только для чтения является использование класса, являющегося базовым для `QSqlTableModel`, т. е. `QSqlQueryModel`. Данный класс предоставляет функцию `setQuery()`, так что существует возможность задавать сложные SQL-запросы для создания конкретных представлений одной или нескольких таблиц, например, с использованием соединений (joins).

В отличие от базы данных *Scooters*, в большинстве баз данных используется множество таблиц и связей по внешним ключам. В Qt предлагается класс `QSqlRelationalTableModel`, являющийся подклассом от `QSqlTableModel`, который может использоваться для отображения и редактирования таблиц с внешними ключами. Класс `QSqlRelationalTableModel` очень сходен с `QSqlTableModel`, за исключением того, что мы можем добавлять в модель отношения `QSqlRelation`, по одному для каждого внешнего ключа. Во многих случаях внешний ключ имеет поле идентификатора и поле имени; используя класс `QSqlRelationalTableModel`, мы можем сделать так, что пользователи будут видеть и изменять поле имени, но с внутренней точки зрения будет использоваться реальное поле идентификатора. Чтобы все это работало правильно, мы должны задать класс `QSqlRelationalDelegate` (или наш собственный подкласс) в представлении, которое будет использоваться для отображения модели.

Мы покажем, как включать представление и изменять внешние ключи в следующих двух разделах, а более подробно представления `QTableView` мы рассмотрим в последнем разделе этой главы.

Редактирование данных с использованием форм

В этом разделе мы увидим, как создать форму диалогового окна, отображающую одну запись. Это диалоговое окно можно использовать для добавления, редактирования и удаления отдельных записей, а также для навигации по записям таблицы.

Мы проиллюстрируем эти концепции на примере приложения Менеджер персонала (StaffManager). Это приложение отслеживает, в каких подразделениях работают сотрудники, где расположены эти подразделения, а также содержит некоторую базовую информацию о сотрудниках, например их добавочный номер телефона. В приложении используются следующие три таблицы:

```
CREATE TABLE location (  
  id INTEGER PRIMARY KEY AUTOINCREMENT,  
  name VARCHAR(40) NOT NULL);  
  
CREATE TABLE department (  
  id INTEGER PRIMARY KEY AUTOINCREMENT,  
  name VARCHAR(40) NOT NULL,  
  locationid INTEGER NOT NULL,  
  FOREIGN KEY (locationid) REFERENCES location);  
  
CREATE TABLE employee (  
  id INTEGER PRIMARY KEY AUTOINCREMENT,  
  name VARCHAR(40) NOT NULL,  
  departmentid INTEGER NOT NULL,  
  extension INTEGER NOT NULL,  
  email VARCHAR(40) NOT NULL,  
  startdate DATE NOT NULL,  
  FOREIGN KEY (departmentid) REFERENCES department);
```

Эти три таблицы и их взаимосвязи схематично показаны на рис. 13.2. В любом месте (location) может находиться любое число подразделений (department), а в каждом подразделении может быть любое число сотрудников (employee). Синтаксис определения внешних ключей указан для SQLite 3, и он может быть иным в других базах данных.

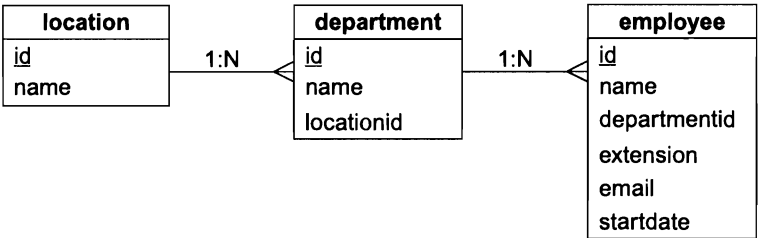


Рис. 13.2. Таблицы приложения Менеджер персонала

В этом разделе основное внимание мы уделим форме `EmployeeForm` – диалоговому окну для редактирования информации о сотрудниках. В следующем разделе мы рассмотрим форму `MainForm`, которая обеспечивает представление подразделений и сотрудников по принципу «главные-подчиненные».

Класс `EmployeeForm` позволяет перейти от общей информации о сотрудниках главной формы к полной информации о конкретном сотруднике. При вызове этой формы она отображает данные об указанном сотруднике, если ей был предоставлен допустимый идентификатор сотрудника, а в противном случае отображает информацию о первом сотруднике. (Эта форма показана на рис. 13.3.) Пользователи могут переходить от сотрудника к сотруднику, удалять и редактировать информацию об имеющихся сотрудниках и добавлять новых сотрудников.



Рис. 13.3. Диалоговое окно `Employee`

Мы предлагаем присвоить осмысленные названия указателям столбцов при помощи перечислимого типа, содержащегося в файле `employeeform.h`:

```
enum {
    Employee_Id = 0,
    Employee_Name = 1,
    Employee_DepartmentId = 2,
    Employee_Extension = 3,
    Employee_Email = 4,
    Employee_StartDate = 5
};
```

В оставшейся части заголовочного файла определяется класс `EmployeeForm`:

```
class EmployeeForm : public QDialog
{
    Q_OBJECT
public:
    EmployeeForm(int id, QWidget *parent = 0);
    void done(int result);
private slots:
```



```

        void addEmployee();
        void deleteEmployee();
    private:
        QSqlRelationalTableModel *tableModel;
        QDataWidgetMapper *mapper;
        QLabel *nameLabel;
        ...
        QDialogButtonBox *buttonBox;
};

```

Для доступа к базе данных мы используем модель `QSqlRelationalTableModel`, а не простую модель `QSqlTableModel`, поскольку нам нужно работать с внешними ключами. `QDataWidgetMapper` – это класс, который позволяет нам связывать виджеты формы с соответствующими столбцами модели данных.

Конструктор формы весьма длинный, поэтому мы будем рассматривать его частями, опуская код компоновки, поскольку он не является здесь для нас важным.

```

EmployeeForm::EmployeeForm(int id, QWidget *parent)
    : QDialog(parent)
{
    nameEdit = new QLineEdit;
    nameLabel = new QLabel(tr("Name:"));
    nameLabel->setBuddy(nameEdit);
    departmentComboBox = new QComboBox;
    departmentLabel = new QLabel(tr("Department:"));
    departmentLabel->setBuddy(departmentComboBox);
    extensionLineEdit = new QLineEdit;
    extensionLineEdit->setValidator(new QIntValidator(0, 99999, this));
    extensionLabel = new QLabel(tr("Extension:"));
    extensionLabel->setBuddy(extensionLineEdit);
    emailEdit = new QLineEdit;
    emailLabel = new QLabel(tr("&Email:"));
    emailLabel->setBuddy(emailEdit);
    startDateEdit = new QDateEdit;
    startDateEdit->setCalendarPopup(true);
    QDate today = QDate::currentDate();
    startDateEdit->setDateRange(today.addDays(-90), today.addDays(90));
    startDateLabel = new QLabel(tr("&Start Date:"));
    startDateLabel->setBuddy(startDateEdit);
}

```

Мы начинаем с создания виджетов для редактирования полей, по одному для каждого поля. Мы также создадим метку, которая будет помещена около редактирующего виджета для обозначения поля.

С помощью класса `QIntValidator` мы добьемся того, чтобы в строку редактирования `Extension` (расширение) принимались только допустимые расширения, в данном случае – числа в диапазоне от 0 до 99999. Мы также установим диапазон дат для поля `Start Date` (дата начала работы) и заставим редактор выводить на экран календарь. Мы не будем заполнять раскрывающийся список напрямую; позже мы предложим модель, по которой он будет заполняться сам.

```

firstButton = new QPushButton(tr("<< &First"));
previousButton = new QPushButton(tr("< &Previous"));
nextButton = new QPushButton(tr("&Next >"));
lastButton = new QPushButton(tr("&Last >>"));
addButton = new QPushButton(tr("&Add"));
deleteButton = new QPushButton(tr("&Delete"));
closeButton = new QPushButton(tr("&Close"));
buttonBox = new QDialogButtonBox;
buttonBox->addButton(addButton, QDialogButtonBox::ActionRole);
buttonBox->addButton(deleteButton, QDialogButtonBox::ActionRole);
buttonBox->addButton(closeButton, QDialogButtonBox::AcceptRole);

```

Мы создаем кнопки навигации (<< First, < Previous, Next > и Last >>), которые сгруппированы в верхней части диалогового окна. Далее мы создаем другие кнопки (Add, Delete и Close) и помещаем их в поле кнопок QDialogButtonBox, расположенное в нижней части диалогового окна. Код, генерирующий компоновку, достаточно очевиден, поэтому мы не будем здесь его рассматривать.

На данный момент мы настроили виджеты пользовательского интерфейса, так что теперь можно обратить внимание на лежащую в его основе функциональность.

```

tableModel = new QSqlRelationalTableModel(this);
tableModel->setTable("employee");
tableModel->setRelation(Employee_DepartmentId,
    QSqlRelation("department", "id", "name"));
tableModel->setSort(Employee_Name, Qt::AscendingOrder);
tableModel->select();
QSqlTableModel *relationModel =
tableModel->relationModel(Employee_DepartmentId);
departmentComboBox->setModel(relationModel);
departmentComboBox->setModelColumn(
    relationModel->fieldIndex("name"));

```

Эта модель конструируется и настраивается во многом также, как QSqlTableModel, которую мы рассматривали ранее, но на этот раз мы используем класс QSqlRelationalTableModel и связь по внешнему ключу. Функция setRelation() принимает индекс поля внешнего ключа и объект QSqlRelation. Конструктор QSqlRelation принимает имя таблицы (таблицы внешнего ключа), имя поля внешнего ключа и имя поля для отображения значения, находящегося в поле внешнего ключа.

Класс QComboBox подобен классу QListWidget в том, что в нем есть внутренняя модель для хранения элементов данных. Мы можем заменить эту модель своей собственной, и именно это мы здесь делаем, предоставив реляционную модель, используемую в QSqlRelationalTableModel. Связь имеет два столбца, и мы должны указать, какой из них будет отображаться в поле с раскрывающимся списком. Реляционная модель была создана, когда мы вызывали функцию setRelation(), поэтому мы не знаем индекса столбца name. Поэтому мы используем функцию fieldIndex(), передавая ей имя поля, и получаем нужный индекс, чтобы в поле с выпадающим списком можно было отобразить названия подразделений.

Благодаря классу `QSqlRelationalTableModel` в поле с раскрывающимся списком будут отображаться названия подразделений, а не их идентификаторы.

```
mapper = new QDataWidgetMapper(this);
mapper->setSubmitPolicy(QDataWidgetMapper::AutoSubmit);
mapper->setModel(tableModel);
mapper->setItemDelegate(new QSqlRelationalDelegate(this));
mapper->addMapping(nameEdit, Employee_Name);
mapper->addMapping(departmentComboBox, Employee_DepartmentId);
mapper->addMapping(extensionLineEdit, Employee_Extension);
mapper->addMapping(emailEdit, Employee_Email);
mapper->addMapping(startDateEdit, Employee_StartDate);
```

Класс `QDataWidgetMapper` отображает поля одной записи базы данных в виджетах, с которыми она связана, а также вносит изменения, сделанные в этих виджетах, в базу данных. Мы можем взять ответственность за отправку (фиксацию) изменений на себя или дать этому классу задание делать это автоматически. Здесь мы выбрали автоматический вариант (`QDataWidgetMapper::AutoSubmit`).

Классу необходимо представить модель для работы, а в случае модели с внешними ключами мы должны также передать объект `QSqlRelationalDelegate`. Этот делегат сделает так, что для пользователя будут отображаться значения из отображаемого столбца `QSqlRelation`, а не необработанные идентификаторы. Кроме того, этот делегат сделает так, что если пользователь запустит редактирование, то в поле с раскрывающимся списком будет выводиться отображаемое значение, но в базу данных будет записываться соответствующее значение индекса (внешний ключ).

Если внешние ключи ссылаются на таблицы с большим числом записей, вероятно лучше создать наш собственный делегат и использовать его для представления формы «списка значений» с возможностью поиска, а не использовать заданные по умолчанию поля с раскрывающимися списками класса `QSqlRelationalTableModel`.

После того, как модель и делегат настроены, мы добавляем соответствия между виджетами формы и соответствующими индексами полей. Поле с раскрывающимся списком функционирует как любые другие виджеты, поскольку всю работу по обработке внешних ключей берет на себя реляционная модель, которую мы уже настроили.

```
if (id != -1) {
    for (int row = 0; row < tableModel->rowCount(); ++row) {
        QSqlRecord record = tableModel->record(row);
        if (record.value(Employee_Id).toInt() == id) {
            mapper->setCurrentIndex(row);
            break;
        }
    }
} else {
    mapper->toFirst();
}
```

Если при вызове диалогового окна был указан допустимый идентификатор сотрудника, мы ищем запись с этим идентификатором и делаем ее текущей записью объекта `mapper`. В противном случае мы просто переходим к первой записи. В любом случае данные из записи будут отражены в соответствующих виджетах.

```
connect(firstButton, SIGNAL(clicked()), mapper, SLOT(toFirst()));
connect(previousButton, SIGNAL(clicked()),
        mapper, SLOT(toPrevious()));
connect(nextButton, SIGNAL(clicked()), mapper, SLOT(toNext()));
connect(lastButton, SIGNAL(clicked()), mapper, SLOT(toLast()));
connect(addButton, SIGNAL(clicked()), this, SLOT(addEmployee()));
connect(deleteButton, SIGNAL(clicked()),
        this, SLOT(deleteEmployee()));
connect(closeButton, SIGNAL(clicked()), this, SLOT(accept()));
...
```

Навигационные кнопки соединены напрямую с соответствующими слотами объекта `mapper`. (Если бы мы использовали политику ручной отправки, нам нужно было бы реализовывать собственные слоты, и с их помощью мы бы отправляли текущую запись, а затем производили навигацию, чтобы изменения не были потеряны.) Объект `mapper` позволяет и редактировать, и осуществлять навигацию. Для добавления или удаления записей мы используем лежащую в их основе модель.

```
void EmployeeForm::addEmployee()
{
    int row = mapper->currentIndex();
    mapper->submit();
    tableModel->insertRow(row);
    mapper->setCurrentIndex(row);
    nameEdit->clear();
    extensionLineEdit->clear();
    startDateEdit->setDate(QDate::currentDate());
    nameEdit->setFocus();
}
```

Слот `addEmployee()` вызывается, когда пользователь нажимает кнопку `Add`. Мы начинаем с извлечения текущей строки, поскольку она была потеряна после отправки. Затем мы вызываем функцию `submit()`, чтобы изменения текущей записи не были потеряны. Хотя мы задали в качестве политики отправки `QDataWidgetMapper::AutoSubmit`, мы все же должны выполнить отправку вручную. Это связано с тем, что автоматическая отправка происходит только если пользователь переносит фокус на другой виджет – чтобы избежать перегрузок, связанных с выполнением операции `UPDATE` в базе данных после каждой вставки или удаления символа. Поэтому существует возможность, что пользователь отредактировал поле и нажал кнопку `Add` не переводя фокус в другое место. Далее мы вставляем новую пустую строку и заставляем `mapper` перейти к ней. Наконец, мы инициализируем виджеты и устанавливаем фокус на первый виджет, чтобы пользователь мог начать вводить данные.

```
void EmployeeForm::deleteEmployee()
{
    int row = mapper->currentIndex();
    tableModel->removeRow(row);
    mapper->submit();
    mapper->setCurrentIndex(qMin(row, tableModel->rowCount() - 1));
}
```

Для удаления мы начинаем с того, что отмечаем текущую строку. Далее мы удаляем строку и отправляем изменение. Мы должны отправить удаление вручную, поскольку политика автоматической отправки применяется только к изменениям записей. В заключение мы делаем текущим индексом `mapper` строку, следующую за удаленной или последнюю строку, если была удалена последняя строка.

Класс `QDataWidgetMapper` упрощает разработку форм с актуальными данными, которые отображают информацию из модели данных. В данном примере мы использовали в качестве основной модели данных `QSqlRelationalTableModel`, но класс `QDataWidgetMapper` может применяться с любой моделью данных, включая модели, не использующие SQL. Альтернативой было бы использование `QSqlQuery` для прямого заполнения формы данными и обновление базы данных. Данный подход требует больше усилий, но также является и более гибким.

В следующем разделе мы рассмотрим оставшуюся часть приложения Менеджер персонала (`StaffManager`), в том числе код, использующий класс `EmployeeForm`, который мы разработали в этом разделе.

Представление данных в табличной форме

Во многих случаях табличное представление является самым простым представлением набора данных для пользователей. В этом разделе мы рассмотрим главную форму приложения Менеджер персонала (`Staff Manager`), которая состоит из двух представлений `QTableView`, находящихся в отношениях «главное-подчиненное». (Эта форма показана на рис. 13.4.) Главное представление – это список подразделений. Подчиненное представление – это список сотрудников в текущем подразделении. В обоих представлениях используются модели `QSqlRelationalTableModel`, поскольку обе таблицы базы данных, которые они представляют, имеют поля внешних ключей.

Как обычно мы используем перечислимый тип, чтобы присвоить осмысленные имена индексам столбцов:

```
enum {
    Department_Id = 0,
    Department_Name = 1,
    Department_LocationId = 2
};
```

Мы начнем с рассмотрения определения класса `MainForm` в заголовочном файле:

```
class MainForm : public QWidget
{
```

```
Q_OBJECT
public:
    MainForm();
private slots:
    void updateEmployeeView();
    void addDepartment();
    void deleteDepartment();
    void editEmployees();
private:
    void createDepartmentPanel();
    void createEmployeePanel();
    QSqlRelationalTableModel *departmentModel;
    QSqlRelationalTableModel *employeeModel;
    QWidget *departmentPanel;
    ...
    QDialogButtonBox *buttonBox;
};
```

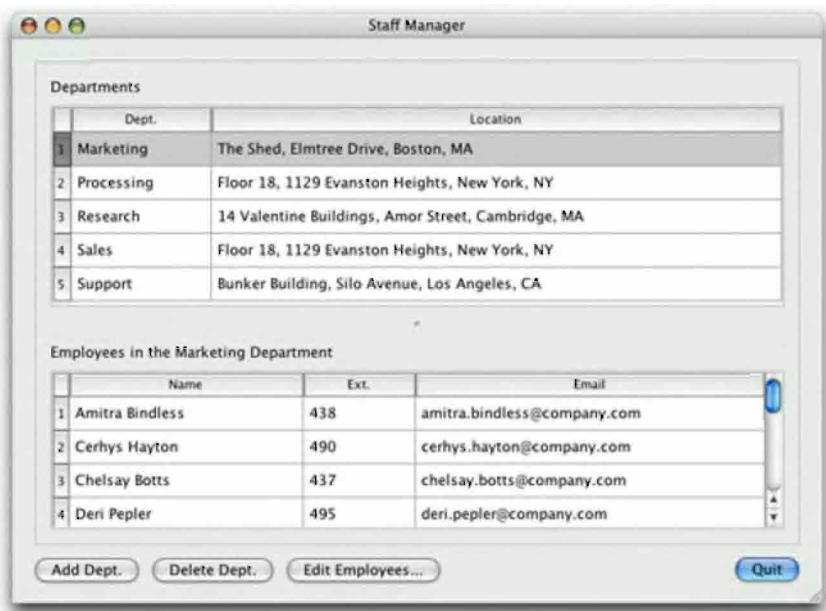


Рис. 13.4. Приложение Менеджер персонала

Чтобы установить отношения главный-подчиненный, мы должны сделать так, чтобы при переходе пользователя к другой записи (строке) производилось обновление подчиненной таблицы и отображение только нужных записей. Это реализуется с помощью закрытого слота `updateEmployeeView()`. Функциональность трех других слотов описывают их имена, а две закрытые функции представляют собой вспомогательные функции конструктора.

Большая часть кода конструктора связана с созданием пользовательского интерфейса и настройкой соответствующих соединений сигнал-слот. Мы сконцентрируемся на тех частях, которые связаны с программированием базы данных.

```
MainForm::MainForm()
{
    createDepartmentPanel();
    createEmployeePanel();
}
```

Конструктор начинается с вызова двух вспомогательных функций. Первая создает и настраивает модель и представление, связанные с подразделениями. Мы рассмотрим важные части этих функций после того, как закончим рассмотрение конструктора.

В следующей части конструктора настраивается разделитель, в который входят два табличных представления, а также настраиваются кнопки формы. Мы пропустим все это.

```
...
connect(addButton, SIGNAL(clicked()), this, SLOT(addDepartment()));
connect(deleteButton, SIGNAL(clicked()),
this, SLOT(deleteDepartment()));
connect(editButton, SIGNAL(clicked()), this, SLOT(editEmployees()));
connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
...
departmentView->setCurrentIndex(departmentModel->index(0, 0));
}
```

Мы связываем кнопки со слотами в диалоговом окне, и делаем первое подразделение текущим элементом. Рассмотрев конструктор, мы изучим код вспомогательной функции `createDepartmentPanel()`, которая настраивает для подразделений модель и представление:

```
void MainForm::createDepartmentPanel()
{
    departmentPanel = new QWidget;
    departmentModel = new QSqlRelationalTableModel(this);
    departmentModel->setTable("department");
    departmentModel->setRelation(Department_LocationId,
    QSqlRelation("location", "id", "name"));
    departmentModel->setSort(Department_Name, Qt::AscendingOrder);
    departmentModel->setHeaderData(Department_Name, Qt::Horizontal,
    tr("Dept."));
    departmentModel->setHeaderData(Department_LocationId, Qt::Horizontal,
    tr("Location"));

    departmentModel->select();
    departmentView = new QTableView;
    departmentView->setModel(departmentModel);
    departmentView->setItemDelegate(new QSqlRelationalDelegate(this));
    departmentView->setSelectionMode(
        QAbstractItemView::SingleSelection);
}
```

```

departmentView->setSelectionBehavior(QAbstractItemView::SelectRows);
departmentView->setColumnHidden(Department_Id, true);
departmentView->resizeColumnsToContents();
departmentView->horizontalHeader()->setStretchLastSection(true);
departmentLabel = new QLabel(tr("Departments"));
departmentLabel->setBuddy(departmentView);
connect(departmentView->selectionModel(),
        SIGNAL(currentRowChanged(const QModelIndex &,
                                const QModelIndex &)),
        this, SLOT(updateEmployeeView()));
...
}

```

Этот код начинается примерно так же, как тот, что мы видели в предыдущем разделе, где настраивалась модель для таблицы `employee`. Данное представление представляет собой стандартный объект `QTableView`, но поскольку у нас используется внешний ключ, мы должны использовать класс `QSqlDelegate`, чтобы в представлении отображался текст внешнего ключа, а не идентификатор, и чтобы его можно было изменять при помощи поля с раскрывающимся списком.

Мы решили скрыть поле идентификатора подразделения, поскольку оно не несет никакой информации для пользователя. Мы также растягиваем последнее видимое поле, адрес подразделения, чтобы оно заняло все доступное пространство по горизонтали.

В представлении подразделения режим выделения (`selection mode`) установлен в `QAbstractItemView::SingleSelection`, а функционирование выделения (`selection behavior`) – в `QAbstractItemView::SelectRows`. Выбранный режим выделения означает, что пользователи могут переходить к отдельным ячейкам таблицы, а заданное функционирование выделения предполагает, что в ходе навигации подсвечивается вся строка. Мы соединяем сигнал `currentRowChanged()` модели выделения представления со слотом `updateEmployeeView()`. Именно это соединение заставляет работать отношение главный-подчиненный, и гарантирует, что в представлении для отображения сотрудников будут показаны сотрудники того подразделения, которое выделено в данный момент в другом представлении.

Код вспомогательной функции `createEmployeePanel()` сходен с предыдущим, но имеет ряд важных отличий:

```

void MainForm::createEmployeePanel()
{
    employeePanel = new QWidget;
    employeeModel = new QSqlRelationalTableModel(this);
    employeeModel->setTable("employee");
    employeeModel->setRelation(Employee_DepartmentId,
        QSqlRelation("department", "id", "name"));
    employeeModel->setSort(Employee_Name, Qt::AscendingOrder);
    employeeModel->setHeaderData(Employee_Name, Qt::Horizontal,
        tr("Name"));
    employeeModel->setHeaderData(Employee_Extension, Qt::Horizontal,
        tr("Ext."));
}

```



```

employeeModel->setHeaderData(Employee_Email, Qt::Horizontal,
                             tr("Email"));
employeeView = new QTableView;
employeeView->setModel(employeeModel);
employeeView->setSelectionMode(QAbstractItemView::SingleSelection);
employeeView->setSelectionBehavior(QAbstractItemView::SelectRows);
employeeView->setEditTriggers(QAbstractItemView::NoEditTriggers);
employeeView->horizontalHeader()->setStretchLastSection(true);
employeeView->setColumnHidden(Employee_Id, true);
employeeView->setColumnHidden(Employee_DepartmentId, true);
employeeView->setColumnHidden(Employee_StartDate, true);
employeeLabel = new QLabel(tr("E&mployees"));
employeeLabel->setBuddy(employeeView);
...
}

```

Триггеры редактирования в данном представлении устанавливаются в `QAbstractItemView::NoEditTriggers`, что, по сути, делает представление доступным только для чтения. В данном приложении пользователь может добавлять, редактировать и удалять записи о сотрудниках, нажимая кнопку **Edit Employees**, которая вызывает форму `EmployeeForm`, созданную в предыдущем разделе.

На этот раз мы скрываем три столбца, а не один. Мы скрываем столбец `id`, поскольку он снова не несет важной для пользователя информации. Мы также скрываем столбец `departmentid`, поскольку в любой момент времени отображаются сотрудники только выделенного в данный момент подразделения. Наконец, мы скрываем столбец `starddate`, поскольку он редко бывает нужен, и обратиться к нему можно, нажав кнопку **Edit Employees**.

```

void MainForm::updateEmployeeView()
{
    QModelIndex index = departmentView->currentIndex();
    if (index.isValid()) {
        QSqlRecord record = departmentModel->record(index.row());
        int id = record.value("id").toInt();
        employeeModel->setFilter(QString("departmentid = %1").arg(id));
        employeeLabel->setText(tr("E&mployees in the %1 Department")
                               .arg(record.value("name").toString()));
    } else {
        employeeModel->setFilter("departmentid = -1");
        employeeLabel->setText(tr("E&mployees"));
    }
    employeeModel->select();
    employeeView->horizontalHeader()->setVisible(
        employeeModel->rowCount() > 0);
}

```

При изменении текущего подразделения (включая первоначальную загрузку) вызывается данный слот. Если существует допустимое текущее подразделение, функция возвращает ID подразделения и переводит фильтр на модель сотрудников. Отображаться будут только те сотрудники, которые имеют соответствующий внешний ключ ID подразделения. (Фильтр представляет собой просто условие WHERE, без самого ключевого слова WHERE.) Мы также обновляем метку, которая отображается над таблицей сотрудников, чтобы в ней выводилось название подразделения, в котором работают сотрудники.

Если допустимых подразделений нет (например, если база данных пуста), мы задаем в фильтре ID несуществующего подразделения, чтобы убедиться, что ему не соответствуют никакие записи.

Затем мы вызываем функцию `select()` применительно к модели, чтобы использовать фильтр. Это, в свою очередь, приводит к генерации сигналов, на которые представление отвечает внесением обновлений. Наконец, мы отображаем или скрываем заголовки столбцов таблицы сотрудников в зависимости от того, отображаются ли в ней сотрудники.

```
void MainForm::addDepartment()
{
    int row = departmentModel->rowCount();
    departmentModel->insertRow(row);
    QModelIndex index = departmentModel->index(row, Department_Name);
    departmentView->setCurrentIndex(index);
    departmentView->edit(index);
}
```

Если пользователь нажимает кнопку Add Dept. (добавить подразделение), мы вставляем новую строку в конец таблицы подразделений, делая эту строку текущей, и запускаем редактирование столбца названия подразделения, как если бы пользователь нажал F2 или сделал по названию двойной щелчок мышью. Если нам нужно предоставить какие-то значения по умолчанию, мы можем вызвать функцию `setData()` сразу после вызова функции `insertRow()`.

Нас не должно заботить создание уникальных ключей для новых записей, поскольку мы используем для этого автоматически инкрементируемый столбец. Если сделать это невозможно или нецелесообразно, мы можем подключиться к сигналу модели `beforeInsert()`. Этот сигнал генерируется после пользовательского редактирования, непосредственно перед тем, как произойдет вставка в базу данных. Это идеальный момент для задания идентификатора или для обработки пользовательских данных. Существуют похожие сигналы `beforeDelete()` и `beforeUpdate()`. Они весьма полезны для создания аудиторских пометок.

```
void MainForm::deleteDepartment()
{
    QModelIndex index = departmentView->currentIndex();
    if (!index.isValid())
        return;
    QSqlDatabase::database().transaction();
    QSqlRecord record = departmentModel->record(index.row());
```

```

int id = record.value(Department_Id).toInt();
int numEmployees = 0;

QSqlQuery query(QString("SELECT COUNT(*) FROM employee "
                        "WHERE departmentid = %1").arg(id));
if (query.next())
    numEmployees = query.value(0).toInt();
if (numEmployees > 0) {
    int r = QMessageBox::warning(this, tr("Delete Department"),
                                tr("Delete %1 and all its employees?")
                                .arg(record.value(Department_Name).toString()),
                                QMessageBox::Yes | QMessageBox::No);
    if (r == QMessageBox::No) {
        QSqlDatabase::database().rollback();
        return;
    }
    query.exec(QString("DELETE FROM employee "
                      "WHERE departmentid = %1").arg(id));
}
departmentModel->removeRow(index.row());
departmentModel->submitAll();
dbQSqlDatabase::database().commit();
updateEmployeeView();
departmentView->setFocus();
}

```

Если пользователь захочет удалить подразделение, он сможет сделать это без дополнительных формальностей, если в подразделении отсутствуют сотрудники. Но если сотрудники в подразделении есть, мы попросим пользователя подтвердить удаление, если он сделает это, мы выполним каскадное удаление, обеспечивая реляционную целостность базы данных. Для этого нам необходимо использовать транзакции, по крайней мере, в тех базах данных, которые не обеспечивают реляционную целостность сами, например, таких как SQLite 3.

После запуска транзакции мы выполняем запрос с целью определения того, сколько сотрудников есть в подразделении. Если есть хотя бы один сотрудник, мы выводим окно с просьбой подтвердить удаление. Если пользователь нажимает No (Нет), выполняется откат транзакции и возврат из функции. В противном случае мы удаляем всех сотрудников подразделения, а также само подразделение и фиксируем транзакцию.

```

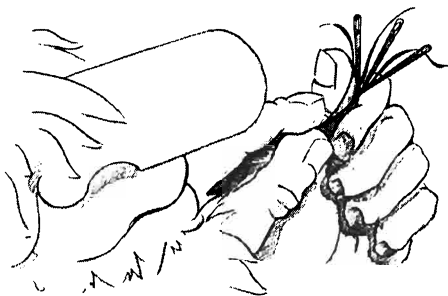
void MainForm::editEmployees()
{
    int employeeId = -1;
    QModelIndex index = employeeView->currentIndex();
    if (index.isValid()) {
        QSqlRecord record = employeeModel->record(index.row());
        employeeId = record.value(Employee_Id).toInt();
    }
}

```

```
EmployeeForm form(employeeId, this);  
form.exec();  
updateEmployeeView();  
}
```

Слот `editEmployees()` вызывается, когда пользователь нажимает кнопку `Edit Employees`. Мы начинаем с присвоения недопустимого ID сотрудника. Затем мы заменяем это значение идентификатором текущего сотрудника, если это возможно. Далее мы конструируем форму `EmployeeForm` и отображаем ее в модальном режиме. Наконец, мы вызываем слот `updateEmployeeView()`, чтобы произвести обновление подчиненного табличного представления, поскольку могли быть внесены изменения в данные о сотрудниках.

В этой главе мы показали, что классы моделей/представлений Qt делают просмотр и редактирование данных в базах SQL весьма простой задачей. В ситуациях, когда нам нужно отобразить записи с помощью представления формы, мы можем использовать класс `QDataWidgetMapper` для связывания виджетов пользовательского интерфейса с полями записи базы данных. Весьма просто устанавливать и отношения главный-подчиненный – для этого необходимо лишь одно соединение сигнал-слот и реализация одного простого слота. Детализовать данные также легко: необходимо только перейти к выбранной записи в конструкторе формы детализованных данных или перейти к первой записи, если запись не выбрана.



- *Создание потоков*
- *Синхронизация потоков*
- *Взаимодействие с главным потоком*
- *Применение классов Qt во вторичных потоках*

Глава 14. Многопоточная обработка

Обычные приложения с графическим интерфейсом имеют один поток (thread) выполнения и производят в каждый момент времени одну операцию. Если пользователь через интерфейс пользователя вызывает продолжительную операцию, интерфейс, как правило, «застывает» до завершения операции. В главе 7 («Обработка событий») даются некоторые способы решения этой проблемы. Применение многопоточной обработки – еще один способ решения данной проблемы.

В многопоточном приложении графический пользовательский интерфейс выполняется в своем собственном потоке, а обработка осуществляется в одном или в нескольких других потоках. При работе на одном процессоре многопоточное приложение, как правило, работает медленнее однопоточного аналога из-за дополнительной нагрузки, которую создают несколько потоков. Но в многопроцессорных системах, которые становятся все более распространенными, многопоточные приложения могут работать сразу с несколькими потоками одновременно на разных процессорах, что приводит к увеличению общей производительности.

В данной главе мы сначала продемонстрируем способы создания подкласса `QThread` и способы применения классов `QMutex`, `QSemaphore` и `QWaitCondition` для синхронизации потоков¹. Затем мы рассмотрим способы взаимодействия вторичных потоков с главным потоком в ходе цикла обработки событий. Наконец, мы завершим главу обзором классов Qt, объясняя какие из них могут использоваться во вторичных потоках.

Многопоточная обработка представляет собой обширную тему, которой посвящается много книг. В данной главе предполагается, что вам уже известны принципы многопоточного программирования, поэтому основное внимание уделяется методам разработки многопоточных приложений средствами Qt, а не теме потоков выполнения в целом.

¹ Ожидается, что в Qt 4.4 будет предоставляться более высокоуровневый API-обработки потоков для описанных здесь классов, что сделает создание многопоточных приложений менее склонным к появлению ошибок.

Создание потоков

Обеспечить многопоточную обработку в приложении Qt достаточно просто: мы просто создаем подкласс QThread и переопределяем его функцию run(). Чтобы показать, как это работает, мы начнем с рассмотрения программного кода очень простого подкласса QThread, который периодически выводит на консоль заданный текст. Пользовательский интерфейс этого приложения показан на рис. 14.1.

```
class Thread : public QThread
{
    Q_OBJECT
public:
    Thread();

    void setMessage(const QString &message);
    void stop();

protected:
    void run();

private:
    QString messageStr;
    volatile bool stopped;
};
```

Класс Thread происходит от QThread и переопределяет функцию run(). Он содержит две дополнительные функции: setMessage() и stop().

Переменная stopped объявляется со спецификатором volatile (изменяемый), поскольку доступ к ней осуществляется из разных потоков, и мы хотим быть уверенными, что всегда получаем ее обновленное значение. Если мы опустим ключевое слово volatile, компилятор может оптимизировать доступ к этой переменной, что, возможно, приведет к получению неправильного результата.

```
Thread::Thread()
{
    stopped = false;
}
```

Мы устанавливаем в конструкторе переменную stopped на значение false.

```
void Thread::run()
{
    while (!stopped)
        std::cerr << qPrintable(messageStr);
    stopped = false;
    std::cerr << std::endl;
}
```

Функция run() вызывается для запуска потока. Пока переменная stopped имеет значение false, эта функция будет выводить на консоль заданное сообщение. Работа потока завершается, когда управление уходит от run().

```
void Thread::stop()
{
    stopped = true;
}
```

Функция `stop()` устанавливает переменную `stopped` на значение `true`, тем самым, указывая функции `run()` на необходимость прекращения вывода текстовых сообщений на консоль. Данная функция может вызываться из любого потока в любое время. В нашем примере мы предполагаем, что присваивание значения переменной типа `bool` является атомарной операцией. Такое предположение является разумным, учитывая, что переменная типа `bool` может иметь только два состояния. Позже мы рассмотрим в данном разделе способы применения класса `QMutex`, гарантирующего атомарность операции присваивания значения переменной.

Класс `QThread` содержит функцию `terminate()`, которая прекращает выполнение потока, если он все еще не завершен. Функцию `terminate()` не рекомендуется применять, поскольку она может остановить поток в произвольной точке и не позволяет потоку выполнить очистку после себя. Всегда надежнее использовать переменную `stopped` и функцию `stop()`, как мы уже делали здесь.



Рис. 14.1. Приложение Threads

Теперь мы рассмотрим способы применения класса `Thread` в небольшом приложении Qt, которое применяет два потока, *A* и *B*, не считая главный поток.

```
class ThreadDialog : public QDialog
{
    Q_OBJECT
public:
    ThreadDialog(QWidget *parent = 0);

protected:
    void closeEvent(QCloseEvent *event);

private slots:
    void startOrStopThreadA();
    void startOrStopThreadB();

private:
    Thread threadA;
    Thread threadB;
    QPushButton *threadAButton;
    QPushButton *threadBButton;
    QPushButton *quitButton;
};
```

В классе ThreadDialog объявляются две переменные типа Thread и несколько кнопок для обеспечения основных средств интерфейса пользователя.

```
ThreadDialog::ThreadDialog(QWidget *parent)
    : QDialog(parent)
{
    threadA.setMessage("A");
    threadB.setMessage("B");

    threadAButton = new QPushButton(tr("Start A"));
    threadBButton = new QPushButton(tr("Start B"));
    quitButton = new QPushButton(tr("Quit"));
    quitButton->setDefault(true);

    connect(threadAButton, SIGNAL(clicked()),
            this, SLOT(startOrStopThreadA()));
    connect(threadBButton, SIGNAL(clicked()),
            this, SLOT(startOrStopThreadB()));
    ...
}
```

В конструкторе мы вызываем функцию setMessage() для периодического вывода на экран первым потоком буквы «А» и вторым потоком буквы «В».

```
void ThreadDialog::startOrStopThreadA()
{
    if (threadA.isRunning()) {
        threadA.stop();
        threadAButton->setText(tr("Start A"));
    } else {
        threadA.start();
        threadAButton->setText(tr("Stop A"));
    }
}
```

Когда пользователь нажимает кнопку потока А, функция startOrStopThreadA() останавливает поток, если он выполняется, и запускает его в противном случае. Она также обновляет текст кнопки.

Программный код функции startOrStopThreadB() очень похож на предыдущий:

```
void ThreadDialog::startOrStopThreadB()
{
    if (threadB.isRunning()) {
        threadB.stop();
        threadBButton->setText(tr("Start B"));
    } else {
        threadB.start();
        threadBButton->setText(tr("Stop B"));
    }
}
```



```
void ThreadDialog::closeEvent(QCloseEvent *event)
{
    threadA.stop();
    threadB.stop();
    threadA.wait();
    threadB.wait();
    event->accept();
}
```

Если пользователь выбирает пункт меню Quit или закрывает окно, мы даем команду остановки для каждого выполняющегося потока и ожидаем их завершения (используя функцию `QThread::wait()`) прежде, чем сделаем вызов `CloseEvent::accept()`. Это обеспечивает аккуратный выход из приложения, хотя в данном случае это не имеет значения.

Если при выполнении приложения вы нажмете кнопку Start A, консоль заполнится буквами «A». Если вы нажмете кнопку Start B, консоль заполнится попеременно последовательностями букв «A» и «B». Нажмите кнопку Stop A, и тогда на экран будет выводиться только последовательность букв «B».

Синхронизация потоков

Обычным требованием для многопоточных приложений является синхронизация работы нескольких потоков. Для этого в Qt предусмотрены следующие классы: `QMutex`, `QReadWriteLock`, `QSemaphore` и `QWaitCondition`.

Класс `QMutex` обеспечивает такую защиту переменной или участка программного кода, что доступ к ним в каждый момент времени может осуществлять только один поток. Этот класс содержит функцию `lock()`, которая закрывает мьютекс (mutex). Если мьютекс открыт, текущий поток захватывает его и немедленно закрывает; в противном случае работа текущего потока блокируется до тех пор, пока захвативший мьютекс поток не освободит его. В любом случае после вызова `lock()` текущий поток будет держать мьютекс до вызова им функции `unlock()`. Класс `QMutex` содержит также функцию `tryLock()`, которая сразу же возвращает управление, если мьютекс уже закрыт.

Например, предположим, что нам нужно обеспечить защиту переменной `stopped` класса `Thread` из предыдущего раздела с помощью `QMutex`. Тогда мы бы добавили к классу `Thread` следующую переменную-член:

```
private:
    ...
    QMutex mutex;
};
```

Функция `run()` изменилась бы следующим образом:

```
void Thread::run()
{
    forever {
        mutex.lock();
        if (stopped) {
            stopped = false;
```

```

        mutex.unlock();
        break;
    }

    mutex.unlock();

    std::cerr << qPrintable(messageStr.ascii);
}
std::cerr << std::endl;
}

```

Функция stop() стала бы такой:

```

void Thread::stop()
{
    mutex.lock();
    stopped = true;
    mutex.unlock();
}

```

Блокировка и разблокировка мьютекса в сложных функциях или там, где обрабатываются исключения C++, может иметь ошибки. Qt предлагает удобный класс `QMutexLocker`, упрощающий обработку мьютексов. Конструктор `QMutexLocker` принимает в качестве аргумента объект `QMutex` и блокирует его. Деструктор `QMutexLocker` разблокирует мьютекс. Например, мы могли бы приведенные выше функции `run()` и `stop()` переписать следующим образом:

```

void Thread::run()
{
    forever {
        {
            QMutexLocker locker(&mutex);
            if (stopped) {
                stopped = false;
                break;
            }
        }

        std::cerr << qPrintable(messageStr);
    }
    std::cerr << std::endl;
}

void Thread::stop()
{
    QMutexLocker locker(&mutex);
    stopped = true;
}

```

Одна из проблем применения мьютексов возникает из-за доступности переменной только для одного потока. В программах со многими потоками, пытающимися одновременно читать одну и ту же переменную (не модифицируя ее),

мьютекс может серьезно снижать производительность. В этих случаях мы можем использовать `QReadWriteLock` – класс синхронизации, допускающий одновременный доступ для чтения без снижения производительности.

В классе `Thread` не имеет смысла заменять мьютекс `QMutex` блокировкой `QReadWriteLock` для защиты переменной `stopped`, потому что в лучшем случае только один поток может пытаться читать эту переменную в любой момент времени. Более подходящий пример мог бы состоять из одного или нескольких считывающих потоков, получающих доступ к некоторым совместно используемым данным, и одного или нескольких записывающих потоков, модифицирующих данные. Например:

```
MyData data;
QReadWriteLock lock;

void ReaderThread::run()
{
    ...
    lock.lockForRead();
    access_data_without_modifying_it(&data);
    lock.unlock();
    ...
}

void WriterThread::run()
{
    ...
    lock.lockForWrite();
    modify_data(&data);
    lock.unlock();
    ...
}
```

Ради удобства мы можем использовать классы `QReadLocker` и `QWriteLocker` для блокировки и разблокировки объекта `QReadWriteLock`.

Класс `QSemaphore` – это еще одно обобщение мьютекса, но в отличие от блокировок чтения/записи он может использоваться для контроля некоторого количества идентичных ресурсов. Следующие два фрагмента программного кода демонстрируют соответствие между `QSemaphore` и `QMutex`:

<code>QSemaphore semaphore(1);</code>	<code>QMutex mutex;</code>
<code>Semaphore.acquire();</code>	<code>mutex.lock();</code>
<code>Semaphore.release();</code>	<code>mutex.unlock();</code>

Передавая 1 конструктору, мы указываем семафору на то, что он управляет работой одного ресурса. Преимущество применения семафора заключается в том, что мы можем передавать конструктору числа, отличные от 1, и затем вызывать функцию `acquire()` несколько раз для захвата многих ресурсов.

Типичная область применения семафоров – это передача некоторого количества данных (`DataSize`) при совместном использовании циклического буфера определенного размера (`BufferSize`):

```
const int DataSize = 100000;
const int BufferSize = 4096;
char buffer[BufferSize];
```

Поток, являющийся поставщиком данных, записывает данные в буфер, пока он не заполнится, и затем повторяет эту процедуру сначала, переписывая существующие данные. Поток, принимающий данные, считывает данные по мере их поступления. Это проиллюстрировано на рис. 14.2 для небольшого 16-байтового буфера.



Рис. 14.2. Модель взаимодействия двух потоков: формирующего и принимающего данные

Необходимость синхронизации для примера взаимодействия потоков, один из которых формирует данные, а другой их считывает, обусловлена двумя причинами: если формирующий данные поток работает слишком быстро, он станет переписывать данные, которые еще не считал поток-приемник; если поток-приемник считывает данные слишком быстро, он перегонит другой поток и станет считывать «мусор».

Грубый способ решения этой проблемы состоит в том, чтобы сначала заполнить буфер и затем ждать, пока поток-приемник не считывает буфер целиком и так далее. Однако в многопроцессорных системах это не позволит обоим потокам работать одновременно с разными частями буфера.

Один из эффективных способов решения этой проблемы заключается в использовании двух семафоров:

```
QSemaphore freeSpace(BufferSize);
QSemaphore usedSpace(0);
```

Семафор `freeSpace` управляет той частью буфера, которая может заполняться потоком, формирующим данные. Семафор `usedSpace` управляет той областью, которую может считывать поток-приемник. Эти две области взаимно дополняют друг друга. Семафор `freeSpace` устанавливается на значение переменной `BufferSize` (4096), то есть, он может захватывать именно такое количество ресурсов. Когда приложение запускается, поток, считывающий данные, начинает захватывать «свободные» байты и превращать их в «используемые» байты. Семафор `usedSpace` инициализируется нулевым значением, чтобы поток-приемник не мог считать мусор при запуске приложения.

В этом примере каждый байт рассматривается как один ресурс. В реальном приложении мы, вероятно, использовали бы более крупные блоки памяти (например, по 64 или 256 байт) для снижения затрат, обусловленных применением семафоров.

```
void Producer::run()
{
```

```
for (int i = 0; i < DataSize; ++i) {
    freeSpace.acquire();
    buffer[i % BufferSize] = "ACGT"[uint(std::rand()) % 4];
    usedSpace.release();
}
}
```

Каждая итерация при работе потока, формирующего данные, начинается с захвата одного «свободного» байта. Если весь буфер заполнен данными, которые не считаны потоком-приемником, вызов функции `acquire()` заблокирует семафор до тех пор, пока поток-приемник не начнет считывать данные. Захватив байт, мы заполняем его некоторым случайным значением («А», «С», «G» или «Т») и затем освобождаем байт и помечаем его как «использованный», тем самым, указывая на возможность его считывания потоком-приемником.

```
void Consumer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        usedSpace.acquire();
        std::cerr << buffer[i % BufferSize];
        freeSpace.release();
    }
    std::cerr << std::endl;
}
```

Работу потока-приемника мы начинаем с захвата одного «использованного» байта. Если буфер не содержит данных для чтения, вызов функции `acquire()` заблокирует семафор до тех пор, пока первый поток не сформирует какие-то данные. После захвата нами байта мы выводим его на экран и освобождаем байт, пометая его как «свободный», тем самым, позволяя первому потоку вновь присвоить ему некоторое значение.

```
int main()
{
    Producer producer;
    Consumer consumer;
    producer.start();
    consumer.start();
    producer.wait();
    consumer.wait();
    return 0;
}
```

Наконец, в функции `main()` мы запускаем оба потока. После этого происходит следующее: поток, формирующий данные, преобразует некоторое «свободное» пространство в «использованное», после чего поток-приемник может выполнить его обратное преобразование в «свободное» пространство.

Когда программа выполняется, она выводит на консоль случайную последовательность из 100 000 букв «А», «С», «G» и «Т» и затем завершает свою работу. Для того чтобы понять, что происходит на самом деле, мы можем отключить

вывод указанной последовательности и вместо этого выводить на консоль букву «Р» при генерации каждого байта первым потоком и букву «с» при чтении байта вторым потоком. И ради максимального упрощения ситуации мы можем использовать меньшие значения параметров `DataSize` и `BufferSize`.

Например, при выполнении программы, когда `DataSize` равен 10 и `BufferSize` равен 4, результат может быть таким: «РсРсРсРсРсРсРсРсРс». В данном случае поток-приемник считывает байты сразу по мере их формирования первым потоком; оба потока работают на одной скорости. В другом случае первый поток может заполнять буфер целиком еще до начала его считывания вторым потоком: «РРРРссссРРРРссссРРсс». Существует много других вариантов. Семафоры дают большую свободу действий планировщикам потоков в специфических системах, позволяя им, изучив поведение потоков, выбрать подходящую политику планирования их работы.

Другой подход к решению проблемы синхронизации работы потока, формирующего данные, и потока, принимающего данные, состоит в применении классов `QWaitCondition` и `QMutex`. Класс `QWaitCondition` позволяет одному потоку «пробуждать» другие потоки, когда удовлетворяется некоторое условие. Этим обеспечивается более точное управление, чем при применении только одних мьютексов. Чтобы показать, как это работает, мы переделаем пример с двумя потоками, используя условия ожидания.

```
const int DataSize = 100000;
const int BufferSize = 4096;
char buffer[BufferSize];

QWaitCondition bufferIsNotFull;
QWaitCondition bufferIsNotEmpty;
QMutex mutex;
int usedSpace = 0;
```

Кроме буфера мы объявляем два объекта `QWaitCondition`, один объект `QMutex` и одну переменную для хранения количества «использованных» байтов в буфере.

```
void Producer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        mutex.lock();
        while (usedSpace == BufferSize)
            bufferIsNotFull.wait(&mutex);
        buffer[i % BufferSize] = "ACGT"[uint(std::rand()) % 4];
        ++usedSpace;
        bufferIsNotEmpty.wakeAll();
        mutex.unlock();
    }
}
```

Работу потока, формирующего данные, мы начинаем с проверки заполнения буфера. Если он заполнен, мы ждем возникновения условия «буфер не заполнен». Когда это условие удовлетворяется, мы записываем один байт в буфер,

увеличиваем на единицу `usedSpace` и возобновляем работу любого потока, ожидающего возникновения условия «буфер не пустой».

Мы используем мьютекс для контроля любого доступа к переменной `usedSpace`. Функция `QWaitCondition::wait()` может принимать в первом своем аргументе заблокированный мьютекс, который она открывает перед блокировкой текущего потока и затем его вновь блокирует перед выходом.

В этом примере мы могли бы заменить цикл `while`

```
while (usedSpace == BufferSize)
    bufferIsNotFull.wait(&mutex);
```

на инструкцию `if`:

```
if (usedSpace == BufferSize) {
    mutex.unlock();
    bufferIsNotFull.wait();
    mutex.lock();
}
```

Однако это не будет правильно работать, как только мы станем использовать несколько потоков, формирующих данные, поскольку другой такой поток может захватить мьютекс сразу же после вызова функции `wait()` и вновь отменить условие «буфер не заполнен».

```
void Consumer::run()
{
    for (int i = 0; i < DataSize; ++i) {
        mutex.lock();
        while (usedSpace == 0)
            bufferIsNotEmpty.wait(&mutex);
        std::cerr << buffer[i % BufferSize];
        --usedSpace;
        bufferIsNotFull.wakeAll();
        mutex.unlock();
    }
    std::cerr << std::endl;
}
```

Поток-приемник работает в точности наоборот относительно первого потока: он ожидает возникновения условия «буфер не пустой» и возобновляет работу любого потока, ожидающего условия «буфер не заполнен».

Во всех приводимых до сих пор примерах наши потоки имеют доступ к одинаковым глобальным переменным. Но для некоторых многопоточных приложений требуется хранить в глобальных переменных неодинаковые данные для разных потоков. Эти переменные часто называют локальной памятью потока (`thread-local storage` - `TLS`) или специальными данными потока (`thread-specific data` - `TSD`). Мы можем «схитрить» и использовать отображение, в качестве ключей которого применяются идентификаторы потоков (возвращаемые функцией `QThread::currentThread()`), но более привлекательное решение состоит в использовании класса `QThreadStorage<T>`.

Обычно класс `QThreadStorage<T>` используется для кэш-памяти. Имея отдельный кэш для каждого потока, мы избегаем затрат, связанных с блокировкой, разблокировкой и возможным ожиданием освобождения мьютекса. Например:

```
QThreadStorage<QHash<int, double>*> cache;

void insertIntoCache(int id, double value)
{
    if (!cache.hasLocalData())
        cache.setLocalData(new QHash<int, double>);
    cache.localData()->insert(id, value);
}

void removeFromCache(int id)
{
    if (cache.hasLocalData())
        cache.localData()->remove(id);
}
```

Переменная `cache` содержит указатель на используемое потоком отображение `QHash<int, double>`. (Из-за проблем с некоторыми компиляторами тип объекта, задаваемый в шаблонном классе `QThreadStorage<T>`, должен быть указателем). При применении первый раз кэша в потоке функция `hasLocalData()` возвращает `false`, и мы создаем объект типа `QHash<int, double>`.

Кроме кэширования класс `QThreadStorage<T>` может использоваться для глобальных переменных, отражающих состояние ошибки (подобных `errno`), чтобы модификации в одном потоке не влияли на другие потоки.

Взаимодействие с главным потоком

При запуске приложения Qt работает только один поток – главный. Только этот поток может создать объект `QApplication` или `QCoreApplication` и вызвать для него функцию `exec()`. После вызова `exec()` этот поток либо ожидает возникновения какого-нибудь события, либо обрабатывает какое-нибудь событие.

Главный поток может запускать новые потоки, создавая объекты подкласса `QThread`, как мы это делали в предыдущем разделе. Если эти новые потоки должны взаимодействовать друг с другом, они могут совместно использовать переменные под управлением мьютексов, блокировок чтения/записи, семафоров или специальных событий. Но ни один из этих методов нельзя использовать для связи с главным потоком, поскольку они будут блокировать цикл обработки событий и «заморозят» интерфейс пользователя.

Для связи вторичного потока с главным потоком необходимо использовать межпоточные соединения сигнал-слот. Обычно механизм сигналов и слотов работает синхронно, т. е. связанный с сигналом слот вызывается сразу после генерации сигнала, используя прямой вызов функции.

Однако когда вы связываете объекты, «живущие» в других потоках, механизм взаимодействия сигналов и слотов становится асинхронным. (Такое поведение можно изменить с помощью пятого параметра функции `QObject::connect()`).

Внутри эти связи реализуются путем регистрации события. Слот затем вызывается в цикле обработки событий потока, в котором находится объект получателя. По умолчанию объект `QObject` существует в потоке, в котором он был создан; в любой момент можно изменить расположение объекта с помощью вызова функции `QObject::moveToThread()`.

Для иллюстрации работы соединений сигнал-слот с разными потоками мы рассмотрим программный код приложения `Image Pro` – процессора изображений, обеспечивающего базовые возможности и позволяющего пользователю поворачивать, изменять размер и цвет изображения. В данном приложении (показанном на рис. 14.3) используется один вторичный поток для выполнения операций над изображениями без блокировки цикла обработки событий. Это имеет существенное значение при обработке изображений очень большого размера. Вторичный поток имеет список выполняемых задач или «транзакции», и он генерирует события для главного окна, чтобы сообщать о том, как идет процесс их выполнения.

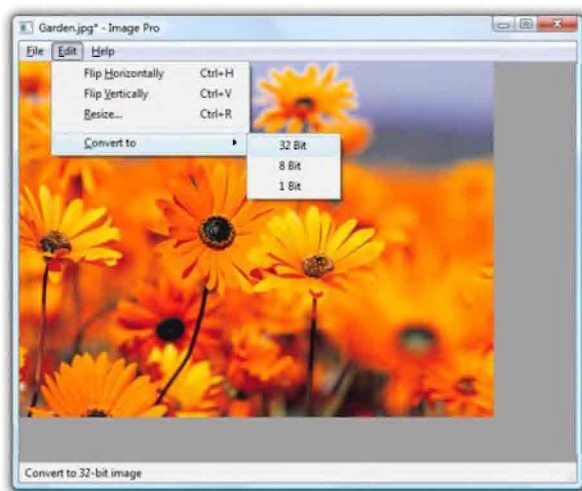


Рис. 14.3. Приложение `Image Pro`

```
ImageWindow::ImageWindow()
{
    imageLabel = new QLabel;
    imageLabel->setBackgroundRole(QPalette::Dark);
    imageLabel->setAutoFillBackground(true);
    imageLabel->setAlignment(Qt::AlignLeft | Qt::AlignTop);
    setCentralWidget(imageLabel);

    createActions();
    createMenus();

    statusBar()->showMessage(tr("Ready"), 2000);
```

```

connect(&thread, SIGNAL(transactionStarted(const QString &)),
       statusBar(), SLOT(showMessage(const QString &)));
connect(&thread, SIGNAL(allTransactionsDone()),
       this, SLOT(allTransactionsDone()));

setCurrentFile("");
}

```

Интересной частью конструктора ImageWindow являются два соединения сигнал-слот. В обоих случаях сигнал генерируется объектом TransactionThread, который мы вскоре рассмотрим.

```

void ImageWindow::flipHorizontally()
{
    addTransaction(new FlipTransaction(Qt::Horizontal));
}

```

Слот flipHorizontally() создает транзакцию зеркального отражения и регистрирует ее при помощи закрытой функции addTransaction(). Функции flipVertically(), resizeImage(), convertTo32Bit(), convertTo8Bit() и convertTo1Bit() реализуются аналогично.

```

void ImageWindow::addTransaction(Transaction *transact)
{
    thread.addTransaction(transact);
    openAction->setEnabled(false);
    saveAction->setEnabled(false);
    saveAsAction->setEnabled(false);
}

```

Функция addTransaction() добавляет транзакцию в очередь транзакций вторичного потока и отключает команды Open, Save и Save As на время обработки транзакций.

```

void ImageWindow::allTransactionsDone()
{
    openAction->setEnabled(true);
    saveAction->setEnabled(true);
    saveAsAction->setEnabled(true);
    imageLabel->setPixmap(QPixmap::fromImage(thread.image()));
    setWindowModified(true);
    statusBar()->showMessage(tr("Ready"), 2000);
}

```

Слот allTransactionsDone() вызывается, когда очередь транзакций TransactionThread становится пустой.

Теперь давайте рассмотрим класс TransactionThread. Как и у большинства подклассов Qthread, его реализация связана с определенными хитростями, поскольку функция run() выполняется в своем собственном потоке, тогда как другие функции (включая конструктор и деструктор) вызываются из главного потока. Определение этого класса таково:

```

class TransactionThread : public QThread
{
    Q_OBJECT

public:
    TransactionThread();
    ~TransactionThread();
    void addTransaction(Transaction *transact);
    void setImage(const QImage &image);
    QImage image();

signals:
    void transactionStarted(const QString &message);
    void allTransactionDone();

protected:
    void run();

private:
    QImage currentImage;
    QQueue<Transaction *> transactions;
    QwaitCondition transactionAdded;
    QMutex mutex;
};

```

Класс `TransactionThread` содержит список обрабатываемых транзакций, которые выполняются по очереди в фоновом режиме. В разделе `private` мы объявляем четыре переменные-члена:

- `currentImage`, в которой хранится изображение, к которому применяются транзакции;
- `transactions`, представляющая собой очередь отложенных транзакций;
- `transactionAdded`, т. е. условие ожидания, используемое для того, чтобы пробудить поток, когда в очередь добавляется новая транзакция;
- `mutex` – используется для защиты переменных-членов `currentImage` и `transactions` от параллельного доступа.

Ниже приводится конструктор класса.

```

TransactionThread::TransactionThread()
{
    start();
}

```

В конструкторе мы просто вызываем функцию `QThread::start()`, запускающую поток, который будет выполнять транзакции.

```

TransactionThread::~TransactionThread()
{
    {

```

```

    QMutexLocker locker(&mutex);
    while (!transactions.isEmpty())
        delete transactions.dequeue();
    transactions.enqueue(EndTransaction);
    transactionAdded.wakeOne();
}
wait();
}

```

В деструкторе мы очищаем очередь транзакций и добавляем в очередь специальный маркер `EndTransaction`. Затем мы пробуждаем поток и ждем его окончания в функции `QThread::wait()`, прежде чем будет неявно вызван деструктор базового класса. Если не вызвать функцию `wait()`, это, скорее всего, приведет к сбою, когда поток попытается обратиться к переменным-членам класса.

Деструктор класса `QMutexLocker` разблокирует мьютекс в конце внутреннего блока, прямо перед вызовом функции `wait()`. Важно разблокировать мьютекс перед вызовом `wait()`, в противном случае в программе может возникнуть «мертвая» блокировка, когда второй поток будет вечно ждать разблокирования мьютекса, а главный поток будет удерживать мьютекс и ожидать, пока вторичный поток завершит работу.

```

void TransactionThread::addTransaction(Transaction *transact)
{
    QMutexLocker locker(&mutex);
    transactions.enqueue(transact);
    transactionAdded.wakeOne();
}

```

Функция `addTransaction()` добавляет транзакцию в очередь транзакций и запускает поток транзакции, если он еще не выполняется. Доступ к переменной-члену `transactions` защищается мьютексом, потому что главный поток мог бы ее модифицировать функцией `addTransaction()` во время прохода по транзакциям `transactions` вторичного потока.

```

void TransactionThread::setImage(const QImage &image)
{
    QMutexLocker locker(&mutex);
    currentImage = image;
}

QImage TransactionThread::image()
{
    QMutexLocker locker(&mutex);
    return currentImage;
}

```

Функции `setImage()` и `image()` позволяют главному потоку установить изображение, для которого будут выполняться транзакции, и получить обработанное изображение после завершения всех транзакций.

```

void TransactionThread::run()
{
    Transaction *transact;

```

```

QImage oldImage;

forever {
    {
        QMutexLocker locker(&mutex);
        if (transactions.isEmpty()) {
            transactionAdded.wait(&mutex);
            transact = transactions.dequeue();
            if (transact == EndTransaction)
                break;

            oldImage = currentImage;
        }

        emit transactionStarted(transact->message());
        QImage newImage = transact->apply(oldImage);
        delete transact;
        {
            QMutexLocker locker(&mutex);
            currentImage = newImage;
            if (transactions.isEmpty())
                emit allTransactionsDone();
        }
    }
}

```

Функция `run()` просматривает очередь транзакций и по очереди выполняет все транзакции путем вызова для них функции `apply()` до тех пор, пока не достигнет маркера `EndTransaction`. Если очередь транзакций пуста, поток будет ожидать условия «транзакция добавлена».

После старта транзакции мы генерируем сигнал `transactionStarted()` с сообщением, выводимым в строке состояния приложения. Когда обработка всех транзакций завершается, мы генерируем сигнал `allTransactionsDone()`.

```

class Transaction
{
public:
    virtual ~Transaction() { }
    virtual QImage apply(const QImage &image) = 0;
    virtual QString message() = 0;
};

```

Класс `Transaction` является абстрактным базовым классом, предназначенным для определения операций, которые пользователь может выполнять с изображением. Виртуальный деструктор необходим, потому что нам приходится удалять экземпляры подклассов `Transaction` через указатель `Transaction`. `Transaction` имеет три конкретных подкласса: `FlipTransaction`, `ResizeTransaction` и `ConvertDepthTransaction`. Нами будет рассмотрен только подкласс `FlipTransaction`; другие два подкласса имеют аналогичное определение.

```

class FlipTransaction : public Transaction
{
public:
    FlipTransaction(Qt::Orientation orientation);

    QImage apply(const QImage &image);
    QString message();

private:
    Qt::Orientation orientation;
};

```

Конструктор `FlipTransaction` принимает один параметр, который задает ориентацию зеркального отражения (по горизонтали или по вертикали).

```

QImage FlipTransaction::apply(const QImage &image)
{
    return image.mirrored(orientation == Qt::Horizontal,
                          orientation == Qt::Vertical);
}

```

Функция `apply()` вызывает `QImage::mirrored()` для объекта `QImage`, полученного в виде параметра, и возвращает сформированный объект `QImage`.

```

QString FlipTransaction::message()
{
    if (orientation == Qt::Horizontal){
        return QObject::tr("Flipping image horizontally...");
    } else {
        return QObject::tr("Flipping image vertically...");
    }
}

```

Функция `messageStr()` возвращает сообщение, отображаемое в строке состояния в ходе выполнения операции. Данная функция вызывается из функции `transactionThread::run()`, когда генерируется сигнал `transactionStarted()`.

Приложение `Image Pro` показывает, как механизм сигналов и слотов `Qt` упрощает взаимодействие вторичных потоков с главным. Реализация вторичного потока сложнее, поскольку нам нужно защитить переменные-члены мьютексом и мы должны деактивировать поток и запускать его в должное время с помощью `wait()`. В серии из двух статей в *QT Quarterly* «Monitors and Wait Conditions in Qt», которую можно скачать со страниц <http://doc.trolltech.com/qc/qc21-monitors.html> и <http://doc.trolltech.com/qc/qc22-monitors2.html>, предлагается еще несколько идей о том, как разрабатывать и тестировать подклассы `QThread`, использующие для синхронизации мьютексы и условия ожидания.

Применение классов `Qt` во вторичных потоках

Функция называется *потокозащищенной* (*thread-safe*), если она может спокойно вызываться одновременно из нескольких потоков. Если две такие функции вызываются из различных потоков и совместно используют одинаковые данные,

результат всегда будет вполне определенным. Это определение можно расширить на класс, и тогда класс будет называться потокозащищенным, если все его функции могут вызываться одновременно из различных потоков, не мешая работе друг друга, если они даже работают с одним и тем же объектом.

В Qt потокозащищенными являются классы `QMutex`, `QMutexLocker`, `QReadWriteLock`, `QReadLocker`, `QWriteLocker`, `QSemaphore`, `QThreadStorage<T>` и `QWaitCondition`. Кроме того, потокозащищенным является часть программного интерфейса `QThread` и несколько других функций, в частности `QObject::connect()`, `QObject::disconnect()`, `QCoreApplication::postEvent()` и `QCoreApplication::removePostedEvents()`.

Большинство классов Qt неграфического интерфейса удовлетворяют менее строгому ограничению: они являются *реентерабельными* (*reentrant*). Класс называется реентерабельным, если разные его экземпляры могут одновременно использоваться разными потоками. Однако одновременный доступ к одному реентерабельному объекту при многопоточной обработке недостаточно надежен и должен контролироваться при помощи мьютекса. Реентерабельность классов отмечается в справочной документации Qt. Обычно любой класс C++, который не использует глобальные переменные (или, другими словами, совместно используемые данные) является реентерабельным.

Класс `QObject` – реентерабельный, однако не следует забывать о трех ограничениях.

- **Дочерние объекты `QObject` должны создаваться их родительским потоком.** В частности, это означает, что созданные во вторичном потоке объекты нельзя создавать с указанием в качестве родительского объекта `QThread`, потому что этот объект был создан в другом потоке (либо в главном потоке, либо в другом вторичном потоке).
- **Все объекты `QObject`, созданные во вторичном потоке, должны быть удалены до удаления соответствующего объекта `QThread`.** Это можно обеспечить путем создания объектов в стеке функцией `QThread::run()`.
- **Объекты `QObject` должны удаляться в том потоке, в котором они были созданы.**

Если требуется удалить объект `QObject`, существующий в другом потоке, мы должны вызвать потокозащищенную функцию `QObject::deleteLater()`, которая регистрирует событие «отсроченное удаление».

Такие подклассы `QObject` неграфического интерфейса, как `QTimer`, `QProcess` и сетевые классы, являются реентерабельными. Мы можем использовать их в любом потоке, содержащем цикл обработки событий. Во вторичных потоках цикл обработки событий начинается с вызова `QThread::exec()` или таких удобных функций, как `QProcess::waitForFinished()` и `QAbstractSocket::waitForDisconnected()`.

Из-за ограничений, унаследованных от низкоуровневых библиотек, на основе которых построена поддержка графического пользовательского интерфейса в Qt, `QWidget` и его подклассы *нереентерабельны*. Одним из следствий этого является невозможность прямого вызова функций виджета из вторичного потока. Если мы, например, хотим изменить текст `QLabel` из вторичного потока, мы можем генерировать сигнал, связанный с `QLabel::setText()`, или вызвать из этого потока функцию `QMetaObject::invokeMethod()`. Например:

```
void MyThread::run()
{
    ...
    QMetaObject::invokeMethod(label, SLOT(setText(const QString &)),
                               Q_ARG(QString, "Hello"));
    ...
}
```

Многие из классов Qt неграфического интерфейса, включая QImage, QString и классы-контейнеры, применяют оптимизацию неявного совместного использования данных. Хотя такая оптимизация делает класс нереентерабельным, в Qt не возникает проблем, потому что Qt использует атомарные инструкции языка ассемблера для реализации потокозащищенного подсчета ссылок, делая реентерабельными Qt-классы, применяющие неявное совместное использование данных.

Модуль *QtSql* также может использоваться в многопоточных приложениях, но он имеет свои ограничения, которые отличаются для разных баз данных. Более подробную информацию вы найдете в сети Интернет по адресу <http://doc.trolltech.com/4.3/sql-driver.html>. Полный список предостережений, относящихся к многопоточной обработке, находится на веб-странице <http://doc.trolltech.com/4.3/threads.html>.



- *Написание FTP-клиентов*
- *Написание HTTP-клиентов*
- *Написание клиент-серверных приложений на базе TCP*
- *Передача и прием дейтаграмм UDP*

Глава 15. Работа с сетью

Qt предоставляет классы `QFtp` и `QHttp` для работы с протоколами FTP и HTTP. Эти протоколы удобно применять для скачивания файлов из сети и их загрузки на удаленный компьютер, а также в случае применения протокола HTTP для передачи запросов на веб-серверы и получения результатов.

Qt также предоставляет низкоуровневые классы `QTcpSocket` и `QUdpSocket`, которые реализуют транспортные протоколы TCP и UDP. TCP – это надежный, ориентированный на соединение протокол, который оперирует потоками данных, циркулирующими между узлами сети, а UDP – ненадежный протокол без установления соединений, основанный на передаче дискретных пакетов от одних сетевых узлов к другим. Оба протокола могут использоваться для создания клиентских и серверных сетевых приложений. В серверных приложениях необходимо также использовать класс `QTcpServer` для обработки входящих TCP-соединений. Безопасные соединения по протоколам SSL/TLS можно устанавливать с помощью `QSslSocket`, а не `QTcpSocket`.

Написание FTP-клиентов

Класс `QFtp` реализует клиентскую часть протокола FTP в Qt. Он предлагает различные функции для выполнения самых распространенных операций протокола FTP и позволяет выполнять произвольные команды FTP.

Класс `QFtp` работает асинхронно. Когда мы вызываем такие функции, как `get()` или `put()`, управление сразу же возвращается к нам, а пересылка данных осуществляется после передачи управления обратно в цикл обработки событий Qt. Это обеспечивает работоспособность интерфейса пользователя во время выполнения команд FTP.

Мы начнем с примера чтения одного файла с помощью функции `get()`. В этом примере создается консольное приложение с именем `ftpget`, которое скачивает удаленный файл, указанный в командной строке. Давайте начнем с функции `main()`:

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

```

QCoreApplication app(argc, argv);
QStringList args = QCoreApplication::arguments();

if (args.count() != 2) {
    std::cerr << "Usage: ftpget url" << std::endl
               << „Example:" << std::endl
               << „    ftpget ftp://ftp.trolltech.com/mirrors"
               << std::endl;
    return 1;
}
FtpGet getter;
if (!getter.getFile(QUrl(args[1])))
    return 1;

QObject::connect(&getter, SIGNAL(done()), &app, SLOT(quit()));

return app.exec();
}

```

Мы создаем объект класса `QCoreApplication`, а не его подкласса `QApplication`, чтобы избежать сборки с библиотекой *QtGui*. Функция `QCoreApplication::arguments()` возвращает аргументы командной строки в виде списка `QStringList`, первым элементом которого является имя вызванной программы, а все специфичные для Qt аргументы, такие как `-style`, удаляются. Центральными моментами в функции `main()` являются конструирование объекта `FtpGet` и вызов функции `getFile()`. Если этот вызов оказывается успешным, мы позволяем циклу событий выполняться до тех пор, пока файл не будет полностью скачен.

Всю работу делает подкласс `FtpGet`, который определяется следующим образом:

```

class FtpGet : public QObject
{
    Q_OBJECT

public:
    FtpGet(QObject *parent = 0);

    bool getFile(const QUrl &url);

signals:
    void done();

private slots:
    void ftpDone(bool error);

private:
    QFtp ftp;
    QFile file;
};

```

Класс имеет открытую функцию `getFile()`, которая считывает файл по указанному адресу URL. Класс `QUrl` имеет высокоуровневый интерфейс для извлечения различных частей URL, таких как имя файла, путь, протокол и порт.

Класс `FtpGet` имеет закрытый слот `ftpDone(bool)`, который вызывается после окончания операции пересылки файла, и сигнал `done()`, который генерируется при завершении скачивания файла. Этот класс имеет также две закрытых переменных. Переменная `ftp` имеет тип `QFtp`, который инкапсулирует соединение с сервером FTP; переменная `file` используется для записи скаченного из сети файла на диск.

```
FtpGet::FtpGet(QObject *parent)
    : QObject(parent)
{
    connect(&ftp, SIGNAL(done(bool)), this, SLOT(ftpDone(bool)));
}
```

В конструкторе мы подсоединяем сигнал `QFtp::done(bool)` к нашему закрытому слоту `ftpDone(bool)`. `QFtp` генерирует сигнал `done(bool)` после завершения обработки всех запросов. Параметр типа `bool` показывает, возникла ошибка или нет.

```
bool FtpGet::getFile(const QUrl &url)
{
    if (!url.isValid()) {
        std::cerr << "Error: Invalid URL" << std::endl;
        return false;
    }
    if (url.scheme() != "ftp") {
        std::cerr << "Error: URL must start with 'ftp:'" << std::endl;
        return false;
    }

    if (url.path().isEmpty()) {
        std::cerr << "Error: URL has no path" << std::endl;
        return false;
    }

    QString localFileName = QFileInfo(url.path()).fileName();
    if (localFileName.isEmpty())
        localFileName = "ftpget.out";
    file.setFileName(localFileName);
    if (!file.open(QIODevice::WriteOnly)) {
        std::cerr << "Error: Cannot write file " <<
            << qPrintable(file.fileName()) << ": "
            << qPrintable(file.errorString()) << std::endl;
        return false;
    }

    ftp.connectToHost(url.host(), url.port(21));
    ftp.login();
}
```

```

ftp.get(url.path(), &file);
ftp.close();
return true;
}

```

Функция `getFile()` начинается с проверки переданного ей URL. Если возникла проблема, функция выводит в поток `cerr` сообщение об ошибке и возвращает `false`, указывая на неудачное скачивание файла.

Мы не обязываем пользователя указывать имя локального файла и пытаемся сами создать осмысленное имя на основе URL, а при неудаче используем имя `ftpget.out`. Если не удастся открыть файл, мы печатаем сообщение об ошибке и возвращаем `false`.

Затем мы выполняем последовательность из четырех команд FTP, используя наш объект `QFtp`. Вызов `url.port(21)` возвращает номер порта, указанный в URL, или порт 21, если URL не содержит порта. Поскольку функции `login()` не передается ни имя пользователя, ни пароль, делается попытка анонимного входа в систему. Вторым аргументом функции `get()` задает выходное устройство ввода-вывода.

Команды FTP ставятся в очередь и обрабатываются в цикле обработки событий Qt. Завершение всех команд регистрируется сигналом `done(bool)` объекта `QFtp`, который мы подсоединили к слоту `ftpDone(bool)` в конструкторе.

```

void FtpGet::ftpDone(bool error)
{
    if (error) {
        std::cerr << "Error: " << qPrintable(ftp.errorString())
                  << std::endl;
    } else {
        std::cerr << "File downloaded as "
                  << qPrintable(file.fileName()) << std::endl;
    }
    file.close();
    emit done();
}

```

После выполнения команд FTP мы закрываем файл и генерируем сигнал `done()`. Может показаться странным, что мы закрываем файл именно здесь, а не после вызова `ftp.close()` в конце функции `getFile()`, но следует помнить, что команды FTP выполняются асинхронно, и их выполнение вполне может быть еще не закончено после возврата управления функцией `getFile()`. Только после генерации объектом `QFtp` сигнала `done()` мы можем быть уверены, что скачивание файла завершено и теперь можно спокойно закрывать файл.

Класс `QFtp` поддерживает несколько FTP-команд, включая `connectToHost()`, `login()`, `close()`, `list()`, `cd()`, `get()`, `put()`, `remove()`, `mkdir()`, `rmdir()` и `rename()`. Все эти функции отправляют какую-то команду FTP и возвращают число, идентифицирующее эту команду. Можно также управлять режимом передачи (по умолчанию используется пассивная передача) и типом передачи (двоичный по умолчанию).

Произвольные команды FTP можно выполнять при помощи функции `rawCommand()`. Например, ниже приводится пример выполнения команды `SITE CHMOD`:

```
ftp.rawCommand("SITE CHMOD 755 fortune");
```

QFtp генерирует сигнал `commandStarted(int)` в начале выполнения команды и сигнал `commandFinished(int, bool)` после завершения выполнения команды. Параметр типа `int` является числом, которое идентифицирует команду. Если мы собираемся отслеживать результаты выполнения отдельных команд, мы можем сохранять эти идентификаторы при постановке команд в очередь. Отслеживание идентификаторов обеспечивает более оперативную обратную связь с пользователем. Например:

```
bool FtpGet::getFile(const QUrl &url)
{
    ...
    connectId = ftp.connectToHost(url.host(), url.port(21));
    loginId = ftp.login();
    getId = ftp.get(url.path(), &file);
    closeId = ftp.close();
    return true;
}

void FtpGet::ftpCommandStarted(int id)
{
    if (id == connectId) {
        std::cerr << "Connecting..." << std::endl;
    } else if (id == loginId) {
        std::cerr << "Logging in..." << std::endl;
    }
    ...
}
```

Другой способ обеспечения обратной связи заключается в подключении к сигналу `stateChanged()` класса QFtp, который генерируется при всяком изменении состояния соединения (QFtp::Connecting, QFtp::Connected, QFtp::LoggedIn и т. д.).

В большинстве приложений нас интересует только результат исполнения всей последовательности команд, а не каких-то конкретных команд. В таком случае мы можем просто подключить сигнал `done(bool)`, который генерируется всякий раз, когда очередь команд становится пустой.

При возникновении ошибки QFtp автоматически очищает очередь команд. Это означает, что при неудачном подсоединении или входе пользователя в систему, оставшиеся в очереди команды никогда не выполнятся. Если мы после возникновения ошибки зададим новые команды с использованием того же объекта QFtp, они будут поставлены в очередь и затем выполнены.

В файл приложения `.pro` необходимо добавить следующую строку для сборки приложения совместно с библиотекой *QtNetwork*:

```
QT += network
```

Теперь мы рассмотрим более сложный пример. Программа командной строки `spider` (паук) скачивает все файлы, расположенные в каталоге FTP-сервера,

рекурсивно просматривая каждый его подкаталог. Вся логика работы с сетью содержится в классе Spider:

```
class Spider : public QObject
{
    Q_OBJECT

public:
    Spider(QObject *parent = 0);
    bool getDirectory(const QUrl &url);

signals:
    void done();

private slots:
    void ftpDone(bool error);
    void ftpListInfo(const QUrlInfo &urlInfo);

private:
    void processNextDirectory();
    QFtp ftp;
    QList<QFile *> openedFiles;
    QString currentDir;
    QString currentLocalDir;
    QStringList pendingDirs;
};
```

Начальный каталог определяется как объект типа QUrl и устанавливается при помощи функции getdirectory().

```
Spider::Spider(QObject *parent)
    : QObject(parent)
{
    connect(&ftp, SIGNAL(done(bool)), this, SLOT(ftpDone(bool)));
    connect(&ftp, SIGNAL(listInfo(const QUrlInfo &)),
            this, SLOT(ftpListInfo(const QUrlInfo &)));
}
```

В конструкторе мы устанавливаем два соединения сигнал-слот. Когда мы выдаем запрос на получение списка элементов каталога (в getDirectory()), QFtp генерирует сигнал listInfo(const QUrlInfo &) для каждого найденного имени. Этот сигнал подключается к слоту с именем ftpListInfo(), который скачивает файл из сети по указанному адресу URL.

```
bool Spider::getDirectory(const QUrl &url)
{
    if (!url.isValid()) {
        std::cerr << "Error: Invalid URL" << std::endl;
        return false;
    }
}
```

```

    if (url.scheme() != "ftp") {
        std::cerr << "Error: URL must start with 'ftp:'" << std::endl;
        return false;
    }

    ftp.connectToHost(url.host(), url.port(21));
    ftp.login();

    QString path = url.path();
    if (path.isEmpty())
        path = "/";

    pendingDirs.append(path);
    processNextDirectory();

    return true;
}

```

Выполнение функции `getdirectory()` начинается с некоторых основных проверок, и если все нормально, делается попытка установить FTP-соединение. Она отслеживает пути, которые необходимо будет обрабатывать, и вызывает функцию `processNextDirectory()`, чтобы начать скачивание корневого каталога.

```

void Spider::processNextDirectory()
{
    if (!pendingDirs.isEmpty()) {
        currentDir = pendingDirs.takeFirst();
        currentLocalDir = "downloads/" + currentDir;
        QDir(".").mkpath(currentLocalDir);
        ftp.cd(currentDir);
        ftp.list();
    } else {
        emit done();
    }
}

```

Функция `processNextDirectory()` принимает первый удаленный каталог из списка каталогов, ожидающих обработки, `pendingDirs`, и создает соответствующий каталог в локальной файловой системе. После этого она указывает объекту `QFtp` на необходимость изменения каталога на принятый ею каталог и затем получения списка его файлов. Для каждого файла, обрабатываемого функцией `list()`, генерируется сигнал `listInfo()`, приводящий к вызову слота `ftpListInfo()`.

Когда все каталоги оказываются обработанными, эта функция генерирует сигнал `done()`, обозначающий завершение скачивания.

```

void Spider::ftpListInfo(const QUrlInfo &urlInfo)
{
    if (urlInfo.isFile()) {
        if (urlInfo.isReadable()) {

```

```

    QFile *file = new QFile(currentLocalDir + "/"
                             + urlInfo.name());
    if (!file->open(QIODevice::WriteOnly)) {
        std::cerr << "Warning: Cannot write file "
                    << qPrintable(QDir::toNativeSeparators
                                    (file->fileName()))
                    << ": " << qPrintable(file->errorString())
                    << std::endl;
        return;
    }
    ftp.get(urlInfo.name(), file);
    openedFiles.append(file);
}
} else if (urlInfo.isDir() && !urlInfo.isSymLink()) {
    pendingDirs.append(currentDir + "/" + urlInfo.name());
}
}
}

```

Параметр `urlInfo` слота `ftpListInfo()` содержит информацию о файле в сети. Если это обычный файл (не каталог) и его можно считывать, мы вызываем функцию `get()` для его загрузки. Объект `QFile`, используемый для загрузки файла, создается с помощью оператора `new` и указатель на него хранится в списке `openedFiles`.

Если содержащиеся в `QUrlInfo` сведения об удаленном каталоге говорят, что он не является символической связью, этот каталог добавляется к списку `pendingDirs`. Мы пропускаем символические связи, поскольку они легко могут привести к бесконечной рекурсии.

```

void Spider::ftpDone(bool error)
{
    if (error) {
        std::cerr << "Error: " << qPrintable(ftp.errorString())
                    << std::endl;
    } else {
        std::cout << "Downloaded " << qPrintable(currentDir) << " to "
                  << qPrintable(QDir::ToNativeSeparators(
                                QDir(currentLocalDir).canonicalPath()));
    }
    qDeleteAll(openedFiles);
    openedFiles.clear();

    processNextDirectory();
}

```

Слот `ftpDone()` вызывается после завершения всех команд FTP или при возникновении ошибки. Мы удаляем объекты `QFile` для предотвращения утечек памяти и также для закрытия всех файлов. Наконец, мы вызываем функцию `processNextDirectory()`. Если какие-нибудь каталоги остались, весь процесс

повторяется для следующего каталога в списке; в противном случае скачивание файлов прекращается и генерируется сигнал `done()`.

Если ошибок нет, последовательность команд **FTP** и сигналов будет следующей:

```
connectToHost(host, port)
login()

cd(directory_1)
list()
    emit listInfo(file_1_1)
        get(file_1_1)
    emit listInfo(file_1_2)
        get(file_1_2)
    ...
emit done()
...
cd(directory_N)
list()
    emit listInfo(file_N_1)
        get(file_N_1)
    emit listInfo(file_N_2)
        get(file_N_2)
    ...
emit done()
```

Если файл фактически оказывается каталогом, он добавляется в список `pendingDirs`, и когда завершается скачивание последнего файла, полученного текущей командой `list()`, выдается новая команда `cd()`, за которой следует новая команда `list()` для следующего каталога, ожидающего обработки, и весь процесс повторяется для нового каталога. Скачиваются новые файлы, и в список `pendingDirs` добавляются новые каталоги до тех пор, пока не будут скачены все файлы из всех каталогов, и список `pendingDirs` в результате не станет пустым.

Если возникнет сетевая ошибка при загрузке пятого файла, скажем, из 20 файлов в каталоге, остальные файлы не будут скачены. Если бы мы захотели скачать как можно больше файлов, то один из способов заключается в выполнении по одной операции `GET` и ожидании сигнала `done(bool)` перед выполнением новой операции `GET`. В функции `listInfo()` мы бы просто добавили имя файла в конец списка `QStringList` вместо немедленного вызова `get()`, а в слоте `done(bool)` мы бы вызывали функцию `get()` для следующего загружаемого файла из списка `QStringList`. Последовательность команд выглядела бы следующим образом:

```
connectToHost(host, port)
login()

cd(directory_1)
list()
...
cd(directory_N)
```

```
list()
    emit listInfo(file_1_1)
    emit listInfo(file_1_2)
    ...
    emit listInfo(file_N_1)
    emit listInfo(file_N_2)
    ...
emit done()

get(file_1_1)
emit done()

get(file_1_2)
emit done()
...
get(file_N_1)
emit done()

get(file_N_2)
emit done()
...
```

Еще одно решение могло бы заключаться в применении одного объекта QFtp для каждого файла. Это позволило бы нам скачивать файлы из сети параллельно, используя отдельные FTP-соединения.

```
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    QStringList args = QCoreApplication::arguments();

    if (args.count() != 2) {
        std::cerr << "Usage: spider url" << std::endl
                  << „Example:" << std::endl
                  << „    spider ftp://ftp.trolltech.com/freebies/"
                  << „leafnode"
                  << std::endl;
        return 1;
    }

    Spider spider;
    if (!spider.getDirectory(QUrl(args[1])))
        return 1;

    QObject::connect(&spider, SIGNAL(done()), &app, SLOT(quit()));

    return app.exec();
}
```

Функция `main()` завершает программу. Если пользователь не задает адрес URL в командной строке, мы выдаем сообщение об ошибке и завершаем программу.

В обоих примерах применения протокола FTP данные, полученные функцией `get()`, записывались в объект `QFile`. Это не обязательно должно быть так. Если бы мы захотели хранить данные в памяти, мы могли бы использовать `QBuffer` – подкласс `QIODevice`, являющийся оболочкой массива `QByteArray`. Например:

```
QBuffer *buffer = new QBuffer;
buffer->open(QIODevice::WriteOnly);
ftp.get(urlInfo.name(), buffer);
```

Мы могли бы также не задавать в функции `get()` аргумент с устройством ввода-вывода или передать нулевой указатель. Класс `QFtp` тогда генерирует сигнал `readyRead()` при поступлении каждой новой порции данных и данные могут считываться при помощи функции `read()` или `readAll()`.

Написание HTTP-клиента

Класс `QHttp` реализует клиентскую часть протокола HTTP в Qt. Он содержит различные функции для выполнения самых распространенных операций протокола HTTP, включая `get()` и `post()`, и обеспечивает средство выполнения произвольных запросов HTTP. Если вы прочитали предыдущий раздел о классе `QFtp`, вы обнаружите, что существует много общего у классов `QFtp` и `QHttp`.

Класс `QHttp` работает асинхронно. Когда мы вызываем такие функции, как `get()` или `post()`, управление сразу же возвращается к нам, а пересылка данных осуществляется после передачи управления обратно в цикл обработки событий Qt. Это обеспечивает работоспособность интерфейса пользователя во время обработки запросов HTTP.

Мы рассмотрим пример консольного приложения с именем `httpget`, демонстрирующего способ загрузки файла с использованием протокола HTTP. Мы не приводим здесь заголовочный файл, поскольку данный пример очень напоминает пример `ftpget`, который мы использовали в предыдущем разделе.

```
HttpGet::HttpGet(QObject *parent)
    : QObject(parent)
{
    ...
    connect(&http, SIGNAL(done(bool)), this, SLOT(httpDone(bool)));
}
```

В конструкторе мы подсоединяем сигнал `done(bool)` объекта `QHttp` к закрытому слоту `httpDone(bool)`.

```
bool HttpGet::getFile(const QUrl &url)
{
    if (!url.isValid()) {
        std::cerr << "Error: Invalid URL" << std::endl;
        return false;
    }
}
```

```

if (url.scheme() != "http") {
    std::cerr << "Error: URL must start with 'http:'" << std::endl;
    return false;
}

if (url.path().isEmpty()) {
    std::cerr << "Error: URL has no path" << std::endl;
    return false;
}

QString localFileName = QFileInfo(url.path()).fileName();
if (localFileName.isEmpty())
    localFileName = "httpget.out";

file.setFileName(localFileName);
if (!file.open(QIODevice::WriteOnly)) {
    std::cerr << "Error: Cannot write file "
                << qPrintable(file.fileName()) << ": "
                << qPrintable(file.errorString()) << std::endl;
    return false;
}

http.setHost(url.host(), url.port(80));
http.get(url.path(), &file);
http.close();
return true;
}

```

Функция `getFile()` проверяет ошибочные ситуации так же, как рассмотренная ранее функция `FtpGet::getFile()`, и использует тот же подход при задании имени локального файла. При загрузке файлов с веб-сайта не требуется входить в систему, поэтому мы просто указываем хост и порт (используя стандартный для HTTP порт 80, если его нет в URL) и скачиваем данные в файл, заданный вторым аргументом функции `QHttp::get()`.

Запросы HTTP ставятся в очередь и обрабатываются асинхронно в цикле обработки событий Qt. На завершение выполнения запросов указывает сигнал `done(bool)` объекта `QHttp`, который мы подсоединили к слоту `httpDone(bool)` в конструкторе.

```

void HttpGet::httpDone(bool error)
{
    if (error) {
        std::cerr << "Error: " << qPrintable(http.errorString())
                    << endl;
    } else {
        std::cerr << „File downloaded as „
                    << qPrintable(file.fileName())
                    << std::endl;
    }
}

```

```

    }
    file.close();
    emit done();
}

```

После выполнения запросов HTTP мы файл закрываем, уведомляя пользователя о возникновении ошибки.

Функция `main()` очень похожа на такую же функцию в примере `ftpget`:

```

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QStringList args = QApplication::arguments();

    if (args.count() != 2) {
        std::cerr << "Usage: httpget url" << std::endl
                  << „Example:" << std::endl
                  << „    httpget http://doc.trolltech.com/ index.html"
                  << std::endl;
        return 1;
    }

    HttpGet getter;
    if (!getter.getFile(QUrl(args[1])))
        return 1;

    QObject::connect(&getter, SIGNAL(done()), &app, SLOT(quit()));

    return app.exec();
}

```

Класс `QHttp` содержит много операций, включая `setHost()`, `get()`, `post()` и `head()`. Если для входа на сайт необходимо выполнить аутентификацию пользователя, `setUser()` может использоваться для установки имени пользователя и пароля. `QHttp` может использовать сокет, указанный программистом, а не свой собственный внутренний `QTcpSocket`. Это делает возможным применение безопасного сокета `QtSslSocket` для работы с HTTP через SSL или TLS.

Мы можем применять функцию `post()` для пересылки пар «имя = значение» в сценарий CGI:

```

http.setHost("www.example.com");
http.post("/cgi/somescript.py", "x=200&y=320", &file);

```

Мы можем передавать данные в виде 8-битовой строки, либо предоставляя открытое устройство `QIODevice`, например `QFile`. Для обеспечения большего контроля мы можем использовать функцию `request()`, которая принимает произвольные заголовок и данные HTTP. Например:

```

HttpRequestHeader header("POST", "/search.html");
header.setValue("Host", "www.trolltech.com");

```

```
header.setContentType("application/x-www-form-urlencoded");  
http.setHost("www.trolltech.com");  
http.request(header, "qt-interest=on&search=opengl");
```

QHttp генерирует сигнал `requestStarted(int)` в начале выполнения команды и сигнал `requestFinished(int, bool)` после завершения выполнения команды. Параметр типа `int` является числом, которое идентифицирует запрос. Если мы собираемся отслеживать результаты выполнения отдельных запросов, мы можем сохранять эти идентификаторы при постановке запросов в очередь. Отслеживание идентификаторов обеспечивает более оперативную обратную связь с пользователем.

В большинстве приложений нас интересует результат исполнения всей последовательности команд. Это легко достигается путем подсоединения сигнала `done(bool)`, который генерируется всякий раз, когда очередь запросов становится пустой.

При возникновении ошибки очередь запросов автоматически очищается. Но если мы после возникновения ошибки зададим новые запросы с использованием того же объекта QHttp, они будут поставлены в очередь и затем выполнены в обычном порядке.

Как и QFtp, класс QHttp содержит сигнал `readyRead()`, а также функции `read()` и `readAll()`, которые мы можем использовать вместо указания устройства ввода-вывода.

Написание клиент-серверных приложений на базе TCP

Классы QTcpSocket и QTcpServer могут использоваться для реализации клиентов и серверов TCP. TCP – это транспортный протокол, который составляет основу большинства прикладных протоколов сети Интернет, включая FTP и HTTP, и который может также использоваться для создания пользовательских протоколов.

TCP является потокоориентированным протоколом. Для приложений данные представляются в виде большого потока данных, очень напоминающего большой однородный файл. Протоколы высокого уровня, построенные на основе TCP, являются либо строкоориентированными, либо блокоориентированными:

- строкоориентированные протоколы передают текстовые данные построчно, завершая каждую строку символом перехода на новую строку;
- блокоориентированные протоколы передают данные в виде двоичных блоков. Каждый блок имеет в начале поле, где указан его размер, и затем идут байты данных.

Класс QTcpSocket является непрямым потомком QIODevice (через класс QAbstractSocket), и поэтому чтение с него или запись на него могут производиться с применением средств класса QDataStream или QTextStream. Одно существенное отличие чтения данных из сети по сравнению с чтением обычного файла заключается в том, что мы должны быть уверены в получении достаточного количества данных от партнерского узла (peer) перед использованием оператора `>>`. В противном случае результат может быть непредсказуемым.

В данном разделе мы рассмотрим программный код клиента и сервера, которые используют пользовательский протокол блочной передачи. Клиент,

показанный на рис. 15.1, называется Trip Planner (планировщик путешествий), он позволяет пользователям составлять план путешествия на поезде. Сервер называется Trip Server (сервер путешествий), он обеспечивает клиента информацией о путешествии. Мы начнем с написания клиентского приложения Trip Planner.

Приложение Trip Planner содержит поле From (из пункта), поле To (до пункта), поле Date (дата), поле Approximate Time (приблизительное время) и два переключателя, определяющие приблизительное время отправления или прибытия. Когда пользователь нажимает клавишу *Search*, приложение посылает запрос на сервер, который возвращает список железнодорожных рейсов, которые удовлетворяют критериям пользователя. Этот список отображается в виджете QTableWidget в окне Trip Planner. В нижней части окна расположена текстовая метка QLabel, показывающая состояние последней операции, и индикатор состояния процесса QProgressBar.

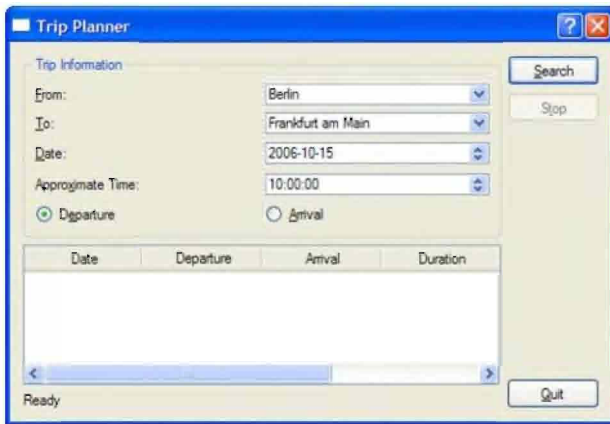


Рис. 15.1. Приложение Trip Planner

Пользовательский интерфейс приложения Trip Planner был создан при помощи *Qt Designer* в файле `tripplanner.ui`. Ниже мы основное внимание уделим исходному коду подкласса `QDialog`, который реализует функциональность приложения:

```
#include "ui_tripplanner.h"
class QPushButton;

class TripPlanner : public QDialog, private Ui::TripPlanner
{
    Q_OBJECT

public:
    TripPlanner(QWidget *parent = 0);

private slots:
    void connectToServer();
```

```

void sendRequest();
void updateTableWidget();
void stopSearch();
void connectionClosedByServer();
void error();

```

```
private:
```

```

void closeConnection();
QPushButton *searchButton;
QPushButton *stopButton;

```

```

QTcpSocket tcpSocket;
quint16 nextBlockSize;

```

```
};
```

Класс TripPlanner происходит не только от QDialog, но и от Ui::TripPlanner (который генерируется компилятором uic, с использованием файла tripplanner.ui). Переменная-член tcpSocket инкапсулирует соединение ТСП. Переменная nextBlockSize используется при синтаксическом анализе блоков, поступивших с сервера.

```

TripPlanner::TripPlanner(QWidget *parent)
    : QDialog(parent)
{
    setupUi(this);

    searchButton = buttonBox->addButton(tr("&Search"),
                                       QDialogButtonBox::ActionRole);
    stopButton = buttonBox->addButton(tr("&Stop"),
                                       QDialogButtonBox::ActionRole);
    stopButton->setEnabled(false);
    buttonBox->button(QDialogButtonBox::Close)->setText(tr("&Quit"));
    QDateTime dateTime = QDateTime::currentDateTime();
    dateEdit->setDate(dateTime.date());
    timeEdit->setTime(QTime(dateTime.time().hour(), 0));

    progressBar->hide();
    progressBar->setSizePolicy(QSizePolicy::Preferred,
                               QSizePolicy::Ignored);

    tableWidget->verticalHeader()->hide();
    tableWidget->setEditTriggers(QAbstractItemView::NoEditTriggers);

    connect(searchButton, SIGNAL(clicked()),
            this, SLOT(connectToServer()));
    connect(stopButton, SIGNAL(clicked()), this, SLOT(stopSearch()));
    connect(buttonBox, SIGNAL(rejected()), this, SLOT(reject()));
    connect(&tcpSocket, SIGNAL(connected()), this, SLOT(sendRequest()));

```



```

connect(&tcpSocket, SIGNAL(disconnected()),
        this, SLOT(connectionClosedByServer()));
connect(&tcpSocket, SIGNAL(readyRead()),
        this, SLOT(updateTableWidget()));
connect(&tcpSocket, SIGNAL(error(QAbstractSocket::SocketError)),
        this, SLOT(error()));
}

```

В конструкторе мы инициализируем поля редактирования даты и времени текущей датой и временем. Мы также не показываем индикатор состояния программы, потому что он необходим только при активном соединении. В *Qt Designer* свойства `minimum` и `maximum` индикатора состояния устанавливались в 0. Это определяет поведение `QProgressBar` как индикатора занятости вместо стандартного индикатора, показывающего процент выполнения работы.

В конструкторе мы также связываем сигналы `connected()`, `disconnected()`, `readyRead()` и `error(QAbstractSocket::SocketError)` класса `QTcpSocket` с закрытыми слотами.

```

void TripPlanner::connectToServer()
{
    tcpSocket.connectToHost("tripserver.zugbahn.de", 6178);

    tableWidget->setRowCount(0);
    searchButton->setEnabled(false);
    stopButton->setEnabled(true);
    statusLabel->setText(tr("Connecting to server..."));
    progressBar->show();

    nextBlockSize = 0;
}

```

Слот `connectToServer()` выполняется, когда пользователь нажимает клавишу *Search* для запуска процедуры поиска. Мы вызываем функцию `connectToHost()` объекта типа `QTcpSocket` для подсоединения к серверу, который, как мы предполагаем, доступен через порт 6178 по вымышленному адресу хоста `tripserver.zugbahn.de`. (Если вы собираетесь проверить работу этого примера на вашей машине, замените имя хоста на `QHostAddress::LocalHost`). Вызов `connectToHost()` выполняется асинхронно; эта функция всегда немедленно возвращает управление. Соединение обычно устанавливается позже. Объект `QTcpSocket` генерирует сигнал `connected()`, если соединение успешно осуществлено и действует, или `error(QAbstractSocket::SocketError)`, если соединение завершилось неудачей.

Затем мы обновляем интерфейс пользователя, в частности делаем видимым индикатор состояния приложения.

Наконец, мы устанавливаем переменную `nextBlockSize` на 0. Эта переменная содержит длину следующего блока, полученного от сервера. Мы задали значение 0, поскольку еще не знаем размер следующего блока.

```

void TripPlanner::sendRequest()
{

```

```
QByteArray block;
QDataStream out(&block, QIODevice::WriteOnly);
out.setVersion(QDataStream::Qt_4_3);
out << quint16(0) << quint8('S') << fromComboBox->currentText()
    << toComboBox->currentText() << dateEdit->date()
    << timeEdit->time();

if (departureRadioButton->isChecked()) {
    out << quint8('D');
} else {
    out << quint8('A');
}
out.device()->seek(0);
out << quint16(block.size() - sizeof(quint16));
tcpSocket.write(block);

statusLabel->setText(tr("Sending request..."));
}
```

Слот `sendRequest()` выполняется, когда объект `QTcpSocket` генерирует сигнал `connected()`, уведомляя об установке соединения. Задача этого слота – сгенерировать запрос к серверу с передачей всей введенной пользователем информации.

Запрос является двоичным блоком следующего формата:

quint16	Размер блока в байтах (не учитывая данное поле)
quint8	Тип запроса (всегда "S")
QString	Пункт отправления
QString	Пункт прибытия
QDate	Дата поездки
QTime	Примерное время отправления или прибытия
quint8	Признак времени отправления ("D") или прибытия ("A")

Сначала мы записываем данные в массив типа `QByteArray` с именем `block`. Мы не можем данные писать непосредственно в `QTcpSocket`, поскольку мы не будем знать размер блока, который будет отсылаться первым, пока не разместим все данные в блоке.

Сначала мы записываем 0 в поле размера блока и затем размещаем остальные данные. Затем мы делаем вызов `seek(0)` для устройства ввода-вывода (для установки на начало буфера `QBuffer`, создаваемого автоматически классом `QDataStream`), чтобы встать на начало массива байтов и переписать первоначальный 0 фактическим размером блока данных. Эта величина рассчитывается как размер блока за вычетом `sizeof(quint16)` (т. е. 2), чтобы исключить поле с размером блока из общей суммы байтов. После этого мы вызываем функцию `write()` для объекта `QTcpSocket`, чтобы отослать этот блок на сервер.

```
void TripPlanner::updateTableWidget()
{
    QDataStream in(&tcpSocket);
```

```

in.setVersion(QDataStream::Qt_4_3);
forever {
    int row = tableWidget->rowCount();

    if (nextBlockSize == 0) {
        if (tcpSocket.bytesAvailable() < sizeof(quint16))
            break;
        in >> nextBlockSize;
    }

    if (nextBlockSize == 0xFFFF) {
        closeConnection();
        statusLabel->setText(tr("Found %1 trip(s)").arg(row));
        break;
    }

    if (tcpSocket.bytesAvailable() < nextBlockSize)
        break;

    QDate date;
    QTime departureTime;
    QTime arrivalTime;
    quint16 duration;
    quint8 changes;
    QString trainType;

    in >> date >> departureTime >> duration >> changes >> trainType;
    arrivalTime = departureTime.addSecs(duration * 60);

    tableWidget->setRowCount(row + 1);

    QStringList fields;
    fields << date.toString(Qt::LocalDate)
        << departureTime.toString(tr("hh:mm"))
        << arrivalTime.toString(tr("hh:mm"))
        << tr("%1 hr %2 min").arg(duration / 60)
            .arg(duration % 60)
        << QString::number(changes)
        << trainType;
    for (int i = 0; i < fields.count(); ++i)
        tableWidget->setItem(row, i,
                               new QTableWidgetItem(fields[i]));

    nextBlockSize = 0;
}
}

```

Слот `updateTableWidget()` **подсоединяется к сигналу** `readyRead()` **класса** `QTcpSocket`, **который генерируется всякий раз при получении** `QTcpSocket` **новых данных от сервера.**

Сервер пересылает нам список возможных железнодорожных рейсов, которые удовлетворяют критерию пользователя. Каждый рейс передается в виде одного блока, и каждый блок начинается с поля размера блока. Поток блоков показан на рис. 15.2. Цикл `forever` необходим, потому что мы не обязательно получаем от сервера блоки по одному.¹ Мы можем получить целый блок или только его часть или полтора блока, или даже все блоки сразу.

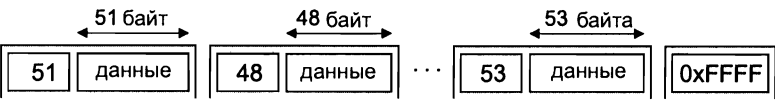


Рис. 15.2. Блоки приложения Trip Server

Итак, как действует цикл `forever`? Если переменная `nextBlockSize` равна 0, это означает, что мы не прочитали размер следующего блока. Мы пытаемся прочесть его (предполагается, что имеется, по крайней мере, 2 байта). Сервер использует значение `0xFFFF` в поле размера блока для указания на то, что все данные переданы, и поэтому если мы обнаруживаем это значение, мы знаем, что достигнут конец.

Если размер блока не равен `0xFFFF`, мы пытаемся считать следующий блок. Во-первых, мы проверяем наличие блока байтов необходимого размера. Если его нет, мы прерываем цикл. Сигнал `readyRead()` будет вновь сгенерирован, когда станет доступно больше данных, и мы попытаемся повторить процедуру.

Если мы уверены, что получен целый блок, мы можем спокойно использовать оператор `>>` для `QDataStream` для извлечения относящейся к поездкам информации, и мы создаем элементы `QTableWidgetItem` с этой информацией. Полученный от сервера блок имеет следующий формат:

quint16	Размер блока в байтах (не учитывая данное поле)
QDate	Дата отправления
QTime	Время отправления
quint16	Длительность поездки (в минутах)
quint8	Количество пересадок
QString	Тип поезда

В конце мы вновь устанавливаем переменную `nextBlockSize` на 0 для указания того, что размер следующего блока не известен, и его необходимо считать.

```
void TripPlanner::closeConnection()
{
    tcpSocket.close();
    searchButton->setEnabled(true);
    stopButton->setEnabled(false);
    progressBar->hide();
}
```

Функция `closeConnection()` закрывает соединение сервера TCP и обновляет интерфейс пользователя. Она вызывается из функции `updateTableWidget()`,

¹ Ключевое слово `forever` обеспечивается Qt. Оно просто разворачивается в оператор `for (;;)`.

когда считывается значение 0xFFFF, и из нескольких других слотов, которые мы вскоре рассмотрим.

```
void TripPlanner::stopSearch()
{
    statusLabel->setText(tr("Search stopped"));
    closeConnection();
}
```

Слот `stopSearch()` подсоединяется к сигналу `clicked()` кнопки **Stop**. По существу, он просто вызывает функцию `closeConnection()`.

```
void TripPlanner::connectionClosedByServer()
{
    if (nextBlockSize != 0xFFFF)
        statusLabel->setText(tr("Error: Connection closed by " ));
    closeConnection();
}
```

Слот `connectionClosedByServer()` подсоединяется к сигналу `disconnected()` объекта `QTcpSocket`. Если сервер закрывает соединение, и мы еще не получили маркер конца, мы уведомляем пользователя о возникновении ошибки. И, как обычно, мы вызываем функцию `closeConnection()` для обновления интерфейса пользователя.

```
void TripPlanner::error()
{
    statusLabel->setText(tcpSocket.errorString());
    closeConnection();
}
```

Слот `error()` подсоединяется к сигналу `error(QAbstractSocket::SocketError)` объекта `QTcpSocket`. Мы игнорируем код ошибки и используем вместо этого функцию `QIODevice::errorString()`, которая возвращает понятное человеку сообщение о последней возникшей ошибке.

На этом завершается рассмотрение класса `TripPlanner`. Функция `main()` приложения `Trip Planner` выглядит обычным образом:

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

Теперь давайте реализуем сервер. Сервер состоит из двух классов: `TripServer` и `ClientSocket`. Класс `TripServer` происходит от `QTcpServer` – класса, который позволяет нам принимать входящие соединения ТСП. Класс `ClientSocket` переопределяет `QTcpSocket` и обслуживает одно соединение. В каждый момент времени в памяти имеется ровно столько объектов типа `ClientSocket`, сколько обслуживается клиентов.

```
class TripServer : public QTcpServer
{
    Q_OBJECT

public:
    TripServer(QObject *parent = 0);

private:
    void incomingConnection(int socketId);
};
```

Класс TripServer переопределяет функцию incomingConnection() из класса QTcpServer. Данная функция вызывается всякий раз, когда клиент пытается подключиться к порту, который прослушивает сервер.

```
TripServer::TripServer(QObject *parent)
    : QTcpServer (parent)
{
}
```

Конструктор TripServer тривиален.

```
void TripServer::incomingConnection(int socketId)
{
    ClientSocket *socket = new ClientSocket(this);
    socket->setSocketDescriptor(socketId);
}
```

В функции incomingConnection() мы создаем объект ClientSocket в качестве дочернего по отношению к объекту TripServer, и мы устанавливаем дескриптор его сокета на переданное нам значение. Объект ClientSocket автоматически удалит сам себя при прекращении соединения.

```
class ClientSocket : public QTcpSocket
{
    Q_OBJECT

public:
    ClientSocket(QObject *parent = 0);

private slots:
    void readClient();

private:
    void generateRandomTrip(const QString &from, const QString &to,
                           const QDate &date, const QTime &time);
    quint16 nextBlockSize;
};
```

Класс ClientSocket происходит от QTcpSocket и инкапсулирует состояние одного клиента.

```
ClientSocket::ClientSocket(QObject *parent)
    : QTcpSocket(parent)
{
}
```

```

connect(this, SIGNAL(readyRead()), this, SLOT(readClient()));
connect(this, SIGNAL(disconnected()), this, SLOT(deleteLater()));
nextBlockSize = 0;
}

```

В конструкторе мы устанавливаем необходимые соединения сигнал-слот и задаем переменной `nextBlockSize` значение 0, свидетельствующее о том, что мы еще не знаем размер посланного клиентом блока.

Сигнал `disconnected()` подсоединяется к функции `deleteLater()`, которая наследуется от класса `QObject` и удаляет объект после возврата управления в цикл обработки событий Qt. Это обеспечивает удаление объекта `ClientSocket` после закрытия сокетного соединения.

```

void ClientSocket::readClient()
{
    QDataStream in(this);
    in.setVersion(QDataStream::Qt_4_3);

    if (nextBlockSize == 0) {
        if (bytesAvailable() < sizeof(quint16))
            return;
        in >> nextBlockSize;
    }
    if (bytesAvailable() < nextBlockSize)
        return;

    quint8 requestType;
    QString from;
    QString to;
    QDate date;
    QTime time;
    quint8 flag;

    in >> requestType;
    if (requestType == 'S') {
        in >> from >> to >> date >> time >> flag;
        std::srand(from.length() * 3600 + to.length() * 60
            + time.hour());
        int numTrips = std::rand() % 8;
        for (int i = 0; i < numTrips; ++i)
            generateRandomTrip(from, to, date, time);

        QDataStream out(this);
        out << quint16(0xFFFF);
    }

    close();
}

```

Слот `readClient()` подсоединяется к сигналу `readyRead()` класса `QTcpSocket`. Если `nextBlockSize` равен 0, мы начинаем считывать размер блока; в противном случае он уже считан нами и тогда мы проверяем поступление целого блока. Если это целый блок, мы считываем его за один шаг. Мы используем `QDataStream` непосредственно для `QTcpSocket` (объект `this`) и считываем поля, используя оператор `>>`.

После чтения запроса клиента мы готовы сформировать ответ. В реальном приложении мы бы осуществляли поиск информации в базе данных расписания железнодорожных рейсов и попытались найти подходящие рейсы. Но здесь мы воспользуемся функцией `generateRandomTrip()`, которая случайным образом генерирует произвольный рейс. Мы вызываем эту функцию произвольное число раз и затем посылаем `0xFFFF` для обозначения конца данных. В конце мы закрываем соединение.

```
void ClientSocket::generateRandomTrip(const QString & /* откуда */,
    const QString & /* куда */, const QDate &date, const QTime &time)
{
    QByteArray block;
    QDataStream out(&block, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_3);
    quint16 duration = std::rand() % 200;
    out << quint16(0) << date << time << duration << quint8(1)
        << QString("InterCity");
    out.device()->seek(0);
    out << quint16(block.size() - sizeof(quint16));

    write(block);
}
```

Функция `generateRandomTrip()` демонстрирует способ пересылки блока данных через соединение TCP. Это очень напоминает то, что мы делали в клиенте в функции `sendRequest()`. И вновь мы записываем блок в массив `QByteArray` таким образом, что мы можем определять его размер до того, как мы его отошлем с помощью функции `write()`.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    TripServer server;
    if (!server.listen(QHostAddress::Any, 6178)) {
        std::cerr << "Failed to bind to port" << std::endl;
        return 1;
    }
    QPushButton quitButton(QObject::tr("&Quit"));
    quitButton.setWindowTitle(QObject::tr("Trip Server"));
    QObject::connect(&quitButton, SIGNAL(clicked()),
        &app, SLOT(quit()));

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


В функции `main()` мы создаем объект `TripServer` и кнопку `QPushButton`, которая позволяет пользователю остановить сервер. Работа сервера начинается с вызова функции `QTcpSocket::listen()`, принимающей адрес IP и номер порта, по которому мы хотим принимать соединения. Специальный адрес `0.0.0.0` (`QHostAddress::Any`) соответствует наличию любого интерфейса IP на локальном хосте. Применять класс `QPushButton` для представления сервера удобно в ходе разработки. Однако ожидается, что размещенный сервер будет работать без графического интерфейса, как служба Windows или демон Unix. Компания Trolltech предлагает для этой цели коммерческое дополнение, называемое `QtService`.

Этим завершается наш пример системы клиент-сервер. В данном случае нами использовался блочноориентированный протокол, позволяющий применять объект типа `QDataStream` для чтения и записи данных. Если бы мы захотели использовать строкоориентированный протокол, наиболее простым было бы применение функций `canReadLine()` и `readLine()` класса `QTcpSocket` в слоте, подсоединенном к сигналу `readyRead()`:

```
QStringList lines;
while (tcpSocket.canReadLine())
    lines.append(tcpSocket.readLine());
```

Мы бы затем могли обрабатывать каждую считанную строку. Пересылка данных могла бы выполняться с использованием `QTextStream` для `QTcpSocket`.

Представленная здесь реализация сервера не очень эффективна в случае, когда соединений много. Это объясняется тем, что при обработке нами одного запроса, мы не обслуживаем другие соединения. Более эффективным был бы запуск нового процесса для каждого соединения. Пример `Threaded Fortune Server` (многопоточный сервер, передающий клиентам интересные изречения, называемые «fortunes»), расположенный в каталоге `Qt examples/network/threadedfortuneserver`, демонстрирует, как это можно сделать.

Передача и прием дейтаграмм UDP

Класс `QUdpSocket` может использоваться для отправки и приема дейтаграмм UDP. UDP – это ненадежный, ориентированный на дейтаграммы протокол. Некоторые приложения применяют протокол UDP, поскольку с ним легче работать, чем с протоколом TCP. По протоколу UDP данные передаются пакетами (дейтаграммами) от одного хоста к другому. Для него не существует понятия соединения, и если доставка пакета UDP в пункт назначения завершается неудачей, никакого сообщения об ошибке не передается отправителю.

Мы рассмотрим способы применения UDP в приложении Qt на примере приложений `Weather Balloon` (метеозонд) и `Weather Station` (метеостанция). Приложение `Weather Balloon` является приложением без графического интерфейса, которое посылает каждые две секунды дейтаграммы UDP с параметрами текущего атмосферного состояния. Приложение `Weather Station` (показанное на рис. 15.3) получает эти дейтаграммы и выводит их на экран. Мы начнем с рассмотрения программного кода приложения `Weather Balloon`.

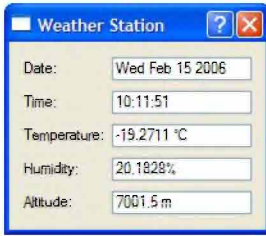


Рис. 15.3. Приложение Weather Station

```
class WeatherBalloon : public QPushButton
{
    Q_OBJECT
public:
    WeatherBalloon(QWidget *parent = 0);

    double temperature() const;
    double humidity() const;
    double altitude() const;

private slots:
    void sendDatagram();
private:
    QUdpSocket udpSocket;
    QTimer timer;
};
```

Класс WeatherBalloon происходит от `QPushButton`. Он использует свою закрытую переменную типа `QUdpSocket` для обеспечения связи с приложением Weather Station.

```
WeatherBalloon::WeatherBalloon(QWidget *parent)
    : QPushButton(tr("Quit"), parent)
{
    connect(this, SIGNAL(clicked()), this, SLOT(close()));
    connect(&timer, SIGNAL(timeout()), this, SLOT(sendDatagram()));

    timer.start(2 * 1000);

    setWindowTitle(tr("Weather Balloon"));
}
```

В конструкторе мы запускаем QTimer для вызова sendDatagram() через каждые 2 секунды.

```
void WeatherBalloon::sendDatagram()
{
    QByteArray datagram;
    QDataStream out(&datagram, QIODevice::WriteOnly);
```

```
out.setVersion(QDataStream::Qt_4_3);
out << QDateTime::currentDateTime() << temperature() << humidity()
    << altitude();
```

```
udpSocket.writeDatagram(datagram, QHostAddress::LocalHost, 5824);
```

}

В `sendDatagram()` мы формируем и отсылаем дейтаграмму, содержащую текущую дату, время, температуру, влажность и высоту над уровнем моря.

QDateTime	Дата и время измерений
double	Температура по Цельсию
double	Влажность в процентах
double	Высота над уровнем моря в метрах

Эта дейтаграмма отсылается функцией `QUdpSocket::writeBlock()`. Вторым и третьим аргументами функции `writeBlock()` являются адрес IP и номер порта партнера (приложения `Weather Station`). В данном примере мы предполагаем, что приложение `Weather Station` выполняется на той же машине, на которой работает приложение `Weather Balloon`, и поэтому мы используем адрес IP `127.0.0.1` (`QHostAddress::LocalHost`) – специальный адрес, предназначенный для использования местными хостами.

В отличие от `QTcpSocket::connectToHost()`, функция `QUdpSocket::writeDatagram()` не получает имена хостов, а только их числовые адреса. Если нам нужно определить имя хоста по его адресу IP, мы имеем две возможности. Если мы готовы блокировать работу во время выполнения поиска, мы можем использовать статическую функцию `QHostInfo::fromName()`. В противном случае мы можем использовать статическую функцию `QHostInfo::lookupHost()`, которая немедленно возвращает управление и вызывает слот с передачей в качестве аргумента объекта `QHostInfo`, который будет содержать соответствующие адреса после завершения поиска.

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

Функция `main()` просто создает объект `WeatherBalloon`, который является участником связи по протоколу UDP и одновременно представлен на экране кнопкой `QPushButton`. Нажимая кнопку `QPushButton`, пользователь может завершить приложение.

Теперь давайте рассмотрим исходный код клиентского приложения `Weather Station`.

```
class WeatherStation : public QDialog
{
    Q_OBJECT
```

```

public:
    WeatherStation(QWidget *parent = 0);

private slots:
    void processPendingDatagrams();

private:
    QUdpSocket udpSocket;
    QLabel *dateLabel;
    QLabel *timeLabel;
    ...
    QLineEdit *altitudeLineEdit;
};

```

Класс `WeatherStation` происходит от `QDialog`. Он прослушивает определенный порт UDP, выполняет синтаксический разбор поступающих дейтаграмм (от приложения `Weather Balloon`) и выводит на экран их содержимое в виде пяти строк редактирования `QLineEdit`, которые используются только для вывода данных. Здесь нас интересует только одна закрытая переменная `udpSocket` типа `QUdpSocket`, которая будет использована для приема дейтаграмм.

```

WeatherStation::WeatherStation(QWidget *parent)
    : QDialog(parent)
{
    udpSocket.bind(5824);
    connect(&udpSocket, SIGNAL(readyRead()),
           this, SLOT(processPendingDatagrams()));
    ...
}

```

Конструктор мы начинаем с привязки объекта `QUdpSocket` к порту, на который передает данные метеозонд. Поскольку мы не указали адрес хоста, сокет будет принимать дейтаграммы, посланные на любой адрес IP, принадлежащий машине, на которой работает приложение `Weather Station`. Затем мы связываем сигнал сокета `readyRead()` с закрытым слотом `processPendingDatagrams()`, который извлекает данные и отображает их на экране.

```

void WeatherStation::processPendingDatagrams()
{
    QByteArray datagram;

    do {
        datagram.resize(udpSocket.pendingDatagramSize());
        udpSocket.readDatagram(datagram.data(), datagram.size());
    } while (udpSocket.hasPendingDatagrams());

    QDateTime dateTime;
    double temperature;
    double humidity;
    double altitude;

```

```

QDataStream in(&datagram, QIODevice::ReadOnly);
in.setVersion(QDataStream::Qt_4_3);
in >> dateTime >> temperature >> humidity >> altitude;

dateLineEdit->setText(dateTime.date().toString());
timeLineEdit->setText(dateTime.time().toString());
temperatureLineEdit->setText(tr("%1 °C").arg(temperature));
humidityLineEdit->setText(tr("%1%").arg(humidity));
altitudeLineEdit->setText(tr("%1 m").arg(altitude));
}

```

Слот `processPendingDatagrams()` вызывается при получении дейтаграммы. `QUdpSocket`, ставит в очередь поступившие дейтаграммы и позволяет получать к ним доступ последовательно в порядке очереди. Обычно в очереди будет только одна дейтаграмма, однако нельзя исключать возможность передачи отправителем последовательно нескольких дейтаграмм до генерации сигнала `readyRead()`. В этом случае мы игнорируем все дейтаграммы кроме последней, поскольку предыдущие дейтаграммы содержат устаревшие параметры атмосферного состояния.

Функция `pendingDatagramSize()` возвращает размер первой ждущей обработки дейтаграммы. С точки зрения приложения дейтаграммы всегда посылаются и принимаются как один блок данных. Это означает, что при любом количестве байтов дейтаграмма будет считываться целиком. Вызов `readDatagram()` копирует содержимое первой ждущей обработки дейтаграммы в указанный буфер `char *` (обрезая данные, если размер буфера оказывается недостаточным) и осуществляет переход к следующей необработанной дейтаграмме. После считывания всех дейтаграмм мы разбиваем последнюю из них (имеющую самые свежие значения параметров атмосферного состояния) на составные части и заполняем строки редактирования `QLineEdit` новыми данными.

```

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    WeatherStation station;
    station.show();
    return app.exec();
}

```

Наконец, в функции `main()` мы создаем и показываем объект `WeatherStation`.

На этом мы завершаем рассмотрение наших примеров по передаче и приему данных с применением протокола UDP. Представленные приложения максимально упрощены, причем приложение `Weather Balloon` посылает дейтаграммы, а приложение `Weather Station` получает их. В большинстве реальных приложений в обоих случаях пришлось бы как считывать, так и записывать данные на свой сокет. Функциям `QUdpSocket::writeDatagram()` могут передаваться адрес хоста и номер порта, поэтому `QUdpSocket` может читать с хоста и порта, с которыми он был связан функцией `bind()`, и писать на какой-нибудь другой хост и порт.



- Чтение документов XML при помощи *QXmlStreamReader*
- Чтение документов XML при помощи интерфейса *DOM*
- Чтение документов XML при помощи интерфейса *SAX*
- Запись документов XML

Глава 16. XML

XML (eXtensible Markup Language – расширяемый язык разметки) – это универсальный формат текстовых файлов, который получил широкое распространение при обмене и хранении данных. Он был разработан World Wide Web Consortium (W3C) как упрощенная альтернатива SGML (Standard Generalized Markup Language – стандартный обобщенный язык разметки). Его синтаксис напоминает HTML, но XML является мета-языком и, как таковой, не имеет специальных обязательных тегов, атрибутов и сущностей. Совместимая с XML версия HTML называется XHTML.

Для популярного XML-формата SVG (Scalable Vector Graphics – масштабируемая векторная графика) модуль *QtSvg* предоставляет классы, которые могут читать и отображать SVG-изображения. Для воспроизведения документов, использующих XML-формат MathML (*Mathematical Markup Language* – язык математической разметки) можно использовать класс *QtMmlWidget* от *Qt Solutions*.

Для обработки обычных XML-документов предлагается модуль *QtXml*, который и является темой настоящей главы¹. Модуль *QtXml* предлагает для чтения XML-документов три разных интерфейса программирования (API).

- *QXmlStreamReader* – это быстрый обработчик (парсер) для чтения хорошо сформированного XML.
- *DOM* (Document Object Model – объектная модель документа) преобразует документ XML в структуру в виде дерева, которая затем может обрабатываться в приложении.
- *SAX* (Simple API for XML – простой программный интерфейс для документов XML) передает «события обработки» непосредственно в приложение с использованием виртуальных функций.

Класс *QXmlStreamReader* является наиболее быстрым в работе и простым в использовании, и он предоставляет API, совместимый с остальными классами *Qt*. Он является идеальным для создания однопроходных парсеров. Главное преимущество *DOM* состоит том, что он позволяет осуществлять навигацию по

¹ Ожидается, что в *Qt 4.4* появятся дополнительные высокоуровневые классы для обработки XML, обеспечивающие поддержку XQuery и XPath в отдельном модуле *QtXmlPatterns*.

древовидному представлению XML-документа в любом порядке, что позволяет реализовывать алгоритмы многопроходной обработки. Некоторые приложения даже используют дерево DOM в качестве основной структуры данных. SAX поддерживается, главным образом, для истории, поскольку применение QDomStreamReader обычно формирует более простой и быстро работающий код.

- Для написания XML-файлов Qt также предоставляет три варианта:
- Можно использовать QDomStreamWriter.
 - Мы можем представить данные в виде дерева DOM в памяти и средствами самого дерева записать его в файл.
 - Можно сгенерировать XML вручную.

Использование QDomStreamWriter пока что является самым простым подходом и более надежным, чем ручное написание XML. Использовать DOM для создания XML, в действительности, имеет смысл, только если дерево DOM уже используется как основная структура данных приложения. В данной главе рассматриваются все эти подходы к чтению и написанию XML.

Чтение XML-документов при помощи QDomStreamReader

Использование QDomStreamReader является самым быстрым и самым простым методом чтения XML. Поскольку парсер работает в инкрементном режиме, он особенно удобен для поиска всех вхождений данного тега в XML-документе, для чтения очень больших XML-файлов, которые могут не поместиться в памяти, а также для заполнения пользовательских структур данных содержимым XML-документа.

Парсер QDomStreamReader работает по принципу лексем, показанных на рис. 16.1. При каждом вызове функции readNext() читается и становится текущей очередная лексема. Свойства текущей лексемы зависят от типа лексемы, и к ним можно получить доступ с помощью функций чтения данных, перечисленных в таблице.

Тип лексемы	Пример	Функция чтения данных
StartDocument	N/A	isStandaloneDocument()
EndDocument	N/A	isStandaloneDocument()
StartElement	<item>	namespaceUri(), name(), attributes(), namespaceDeclarations()
EndElement	</item>	namespaceUri(), name()
Characters	AT&T	text(), isWhitespace(), isCDATA()
Comment	<!-- fix -->	text()
DTD	<!DOCTYPE ...>	text(), notationDeclarations(), entityDeclarations()
EntityReference	™	name(), text()
ProcessingInstruction	<?alert?>	processingInstructionTarget(), processingInstructionData()
Invalid	>&<!	error(), errorString()

Рис. 16.1. Лексемы QDomStreamReader

Рассмотрим следующий XML-документ:

```
<doc>
  <quote> Einmal ist keinmal </quote>
</doc>
```

Если мы будем обрабатывать этот документ, то при каждом вызове функции `readNext()` мы будем получать новую лексему, из которой можно с помощью функций чтения получить дополнительную информацию:

```
StartDocument
StartElement (name() == "doc")
StartElement (name() == "quote")
Characters (text() == "Einmal ist keinmal")
EndElement (name() == "quote")
EndElement (name() == "doc")
EndDocument
```

После каждого вызова `readNext()` мы можем протестировать тип текущей лексемы с помощью `isStartElement()`, `isCharacters()` и похожих функций, или просто при помощи функции `state()`. Мы рассмотрим пример, показывающий, как использовать класс `QXmlStreamReader` для обработки специфического формата XML-файла и отображения его содержимого в виджете `QTreeWidget`. Мы будем обрабатывать формат предметного указателя (индекса) книги с вхождениями элементов и подэлементов. Ниже приводится файл предметного указателя книги, отображаемый в `QTreeWidget` (рис. 16.2).

```
<?xml version="1.0"?>
<bookindex>
  <entry term="sidebearings">
    <page>10</page>
    <page>34-35</page>
    <page>307-308</page>
  </entry>
  <entry term="subtraction">
    <entry term="of pictures">
      <page>115</page>
      <page>244</page>
    </entry>
    <entry term="of vectors">
      <page>9</page>
    </entry>
  </entry>
</bookindex>
```

Мы начнем с рассмотрения фрагмента функции `main()` приложения, чтобы увидеть, как в данном контексте используется метод чтения XML, а затем изучим реализацию метода чтения.

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



```

QStringList args = QApplication::arguments();
...
QTreeWidget treeWidget;
...
XmlStreamReader reader(&treeWidget);
for (int i = 1; i < args.count(); ++i)
    reader.readFile(args[i]);
return app.exec();
}

```

Приложение, показанное на рис. 16.2, начинается с создания объекта `QTreeWidget`. Затем создается объект `XMLStreamReader`, которому передается созданный виджет, и дается задача обработать все файлы, указанные в командной строке.

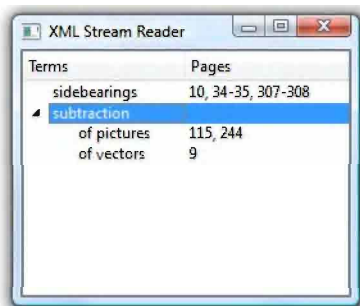


Рис. 16.2. Приложение XML Stream Reader

```

class XmlStreamReader
{
public:
    XmlStreamReader(QTreeWidget *tree);
    bool readFile(const QString &fileName);
    void readBookindexElement();
    void readEntryElement(QTreeWidgetItem *parent);
    void readPageElement(QTreeWidgetItem *parent);
    void skipUnknownElement();
    QTreeWidget *treeWidget;
    QXmlStreamReader reader;
};

```

Класс `XMLStreamReader` предоставляет две открытые функции: конструктор и `parseFile()`. Этот класс использует экземпляр класса `QXmlStreamReader` для обработки XML-файла и заполнения объекта `QTreeWidget` читаемыми XML-данными. Обработка ведется методом рекурсивного спуска.

- Функция `readBookindexElement()` обрабатывает элемент `<bookindex>...</bookindex>`, содержащий ноль или более элементов `<entry>`.
- Функция `readEntryElement()` обрабатывает элемент `<entry>...</entry>`, содержащий ноль или более элементов `<page>`, имеющий неограниченное число уровней вложенности.

- Функция `readPageElement()` обрабатывает элемент `<page>...</page>`.
 - Функция `skipUnknownElement()` осуществляет пропуск нераспознанного элемента.
- Теперь мы рассмотрим реализацию класса `XmlStreamReader`, начиная с конструктора.

```
XmlStreamReader::XmlStreamReader(QTreeWidgetItem *tree)
{
    treeWidget = tree;
};
```

Конструктор используется только для того, чтобы установить, какой объект `QTreeWidgetItem` должен использовать объект `reader`. Все действие осуществляется в функции `readFile()` (вызываемой из функции `main()`), которую мы будем рассматривать в трех частях.

```
bool XmlStreamReader::readFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QFile::ReadOnly | QFile::Text)) {
        std::cerr << "Error: Cannot read file " << qPrintable(fileName)
                    << ": " << qPrintable(file.errorString()) << std::endl;
        return false;
    }
    reader.setDevice(&file);
```

Функция `readFile()` начинается с попытки открыть файл. Если открыть файл не удастся, выводится сообщение об ошибке и возвращается значение `false`. Если файл успешно открыт, он становится устройством ввода для объекта `QXmlStreamReader`.

```
    reader.readNext();
    while (!reader.atEnd()) {
        if (reader.isStartElement()) {
            if (reader.name() == "bookindex") {
                readBookindexElement();
            } else {
                reader.raiseError(QObject::tr("Not a bookindex file"));
            }
        } else {
            reader.readNext();
        }
    }
}
```

Функция `readNext()` класса `QXmlStreamReader` читает следующую лексему из входного потока. Если лексема успешно прочитана, и конец XML-файла не достигнут, функция входит в цикл `while`. Из структуры файлов предметных указателей мы знаем, что в этом цикле у нас есть только три возможности: был прочитан стартовый тег `<bookindex>`, был прочитан другой стартовый тег (в этом случае файл не является предметным указателем) или была прочитана другая лексема.

Если мы имеем правильный начальный тег, мы вызываем для продолжения обработки функцию `readBookindexElement()`. В противном случае мы вызываем функцию `QXmlStreamReader::raiseError()`, передавая ей сообщение об ошибке. При

следующем вызове функции `atEnd()` (в условиях цикла `while`) она вернет значение `true`. Это даст гарантию, что обработка при возникновении ошибки будет прекращена как можно быстрее. К ошибке можно обратиться позже, путем вызова функций `error()` и `errorString()` применительно к объекту класса `QFile`. В качестве альтернативы можно выполнить возврат сразу, как только будет обнаружена ошибка в файле предметного указателя. Использовать функцию `raiseError()`, как правило, более удобно, поскольку она позволяет использовать для низкоуровневых ошибок обработки XML (которые генерируются автоматически, когда класс `QXmlStreamReader` сталкивается с некорректным кодом XML) тот же механизм отчетов об ошибках, что и для ошибок, специфичных для приложения.

```
file.close();
if (reader.hasError()) {
    std::cerr << "Error: Failed to parse file "
                << qPrintable(fileName) << ": "
                << qPrintable(reader.errorString()) << std::endl;
    return false;
} else if (file.error() != QFile::NoError) {
    std::cerr << "Error: Cannot read file " << qPrintable(fileName)
                << ": " << qPrintable(file.errorString())
                << std::endl;
    return false;
}
return true;
}
```

После окончания обработки файл закрывается. Если возникает ошибка обработки или ошибка файла, функция выводит сообщение об ошибке и возвращает значение `false`, в противном случае возвращается `true`, обозначающее успешную обработку.

```
void XmlStreamReader::readBookindexElement()
{
    reader.readNext();
    while (!reader.atEnd()) {
        if (reader.isEndElement()) {
            reader.readNext();
            break;
        }
        if (reader.isStartElement()) {
            if (reader.name() == "entry") {
                readEntryElement(treeWidget->invisibleRootItem());
            } else {
                skipUnknownElement();
            }
        } else {
            reader.readNext();
        }
    }
}
```

Функция `readBookindexElement()` отвечает за чтение основной части файла. Она начинает с того, что пропускает текущую лексему (которой, в данный момент, будет один начальный тег `<bookindex>`), а затем перебирает в цикле входные данные. Если прочитывается конечный тег, то это может быть только `</bookindex>`, в противном случае класс `QXmlStreamReader` вывел бы сообщение об ошибке (`UnexpectedElementError`). В этом случае тег пропускается и выполняется выход из цикла. Иначе мы должны считать стартовый тег элемента `<entry>` верхнего уровня. Если это так, мы вызываем функцию `readEntryElement()` для обработки данных внутри этого элемента. Использование функции `skipUnknownElement()` вместо `raiseError()` означает, что если в будущем мы расширим формат предметного указателя, включив в него новые теги, данное средство чтения будет продолжать работать, поскольку будет просто игнорировать теги, которые не может распознать.

Функция `readEntryElement()` принимает аргумент `QTreeWidgetItem *`, идентифицирующий родительский элемент. Мы передаем ей в качестве родителя `QTreeWidgetItem::invisibleRootItem()`, чтобы сделать новые элементы корневыми элементами. В функции `readEntryElement()` мы будем рекурсивно вызывать функцию `readEntryElement()`, указывая другого родителя.

```
void XmlStreamReader::readEntryElement(QTreeWidgetItem *parent)
{
    QTreeWidgetItem *item = new QTreeWidgetItem(parent);
    item->setText(0, reader.attributes().value("term").toString());
    reader.readNext();
    while (!reader.atEnd()) {
        if (reader.isEndElement()) {
            reader.readNext();
            break;
        }
        if (reader.isStartElement()) {
            if (reader.name() == "entry") {
                readEntryElement(item);
            } else if (reader.name() == "page") {
                readPageElement(item);
            } else {
                skipUnknownElement();
            }
        } else {
            reader.readNext();
        }
    }
}
```

Функция `readEntryElement()` вызывается каждый раз, когда встречается начальный тег `<entry>`. Мы хотим, чтобы для каждого вхождения индекса был создан элемент древовидного виджета, так что мы создаем новый объект `QTreeWidgetItem`, и в качестве текста в его первом столбце вводим текст атрибута `term` вхождения.

После того, как элемент-вхождение был добавлен в дерево, читается следующая лексема. Если она представляет собой конечный тег, мы пропускаем этот тег

и прерываем цикл. Если встречается начальный тег, это может быть тег `<entry>` (обозначающий подэлемент-вхождение), тег `<page>` (номер страницы вхождения) или неизвестный тег. Если начальный тег представляет собой подэлемент, мы выполняем рекурсивный вызов функции `call readEntryElement()`. Если это тег `<page>`, мы вызываем функцию `readPageElement()`.

```
void XmlStreamReader::readPageElement(QTreeWidgetItem *parent)
{
    QString page = reader.readElementText();
    if (reader.isEndElement())
        reader.readNext();
    QString allPages = parent->text(1);
    if (!allPages.isEmpty())
        allPages += ", ";
    allPages += page;
    parent->setText(1, allPages);
}
```

Функция `readPageElement()` вызывается, когда нам встречается тег `<page>`. Ей передается элемент дерева, соответствующий вхождению, к которому относится текст на данной странице. Мы начинаем с того, что читаем текст между тегами `<page>` и `</page>`. Если чтение прошло успешно, функция `readElementText()` обратка доходит до тега `</page>`, который мы должны пропустить.

Страницы сохраняются во втором столбце элемента древовидного виджета. Мы начинаем с того, что извлекаем уже содержащийся здесь текст. Если это не пустое поле, мы добавляем к содержимому запятую, подготавливая к добавлению текста новой страницы. Затем мы добавляем этот новый текст и обновляем соответственно текст в столбце.

```
void XmlStreamReader::skipUnknownElement()
{
    reader.readNext();
    while (!reader.atEnd()) {
        if (reader.isEndElement()) {
            reader.readNext();
            break;
        }
        if (reader.isStartElement()) {
            skipUnknownElement();
        } else {
            reader.readNext();
        }
    }
}
```

Наконец, когда нам встречается неизвестный тег, мы продолжаем чтение до тех пор, пока не получим завершающий тег неизвестного элемента, который мы также пропускаем. Это означает, что мы будем пропускать хорошо сформированные, но не распознанные элементы, и читать как можно больше распознаваемых данных из XML-файла.

Представленный здесь пример может использоваться как основа для сходных парсеров XML, работающих по принципу рекурсивного спуска. Тем не менее, иногда реализация таких парсеров может быть весьма сложным делом, если вызов `readNext()` пропущен или находится не в том месте. Некоторые программисты борются с этой проблемой, добавляя в код операторы `assert`. Например, в начало функции `readBookindexElement()` можно добавить такую строку:

```
Q_ASSERT(reader.isStartElement() && reader.name() == "bookindex");
```

Сходный оператор можно поместить в функции `readEntryElement()` и `readPageElement()`. В функции `skipUnknownElement()` мы можем просто допустить, что у нас есть начальный элемент.

Класс `QXmlStreamReader` может принимать входные данные от любого устройства `QIODevice`, включая `QFile`, `QBuffer`, `QProcess` и `QTcpSocket`. Некоторые источники входных данных могут не предоставить нужные парсеру данные тогда, когда они ему нужны, например, из-за задержки при работе в сети. В таких обстоятельствах также есть возможность использовать `QXmlStreamReader`, и дополнительную информацию об этом можно получить в справочной документации по `QXmlStreamReader`, в разделе «Инкрементная обработка».

Класс `QXmlStreamReader`, который мы использовали в данном приложении, входит в библиотеку `QtXML`. Чтобы подключиться к этой библиотеке, мы должны добавить следующую строку в файл `.pro`.

```
QT += xml
```

В следующих двух разделах мы увидим, как можно написать такое же приложение с использованием DOM и SAX.

Чтение документов XML при помощи интерфейса DOM

DOM является стандартным программным интерфейсом синтаксического анализа документов XML, который разработан W3C. Qt обеспечивает уровень 2 интерфейса DOM для чтения, обработки и записи документов XML без проверки их достоверности.

DOM представляет файл XML в памяти в виде дерева. Мы можем просматривать дерево DOM столько раз, сколько нам нужно, и мы можем модифицировать и записывать его на диск в виде файла XML.

Давайте рассмотрим следующий документ XML:

```
<doc>
  <quote>Scio me nihil scire</quote>
  <translation>I know that I know nothing</translation>
</doc>
```

Ему соответствует следующее дерево DOM:

```
Document
├── Element (doc)
│   ├── Element (quote)
│   │   └── Text ("Scio me nihil scire")
│   └── Element (translation)
│       └── Text ("I know that I know nothing")
```

Дерево DOM содержит узлы разных типов. Например, узел `Element` соответствует открывающему тегу и связанному с ним закрывающему тегу. Все что располагается между этими тегами, представляется в виде дочерних узлов данного элемента `Element`. В Qt различные типы таких узлов (как и все другие связанные с DOM классы) имеют префикс `QDom`. Так, `QDomElement` представляет узел `Element`, а `QDomText` представляет узел `Text`.

Различные узлы могут иметь дочерние узлы разных типов. Например, узел `Element` может содержать другие узлы `Element`, а также узлы `EntityReference`, `Text`, `CDATASection`, `ProcessingInstruction` и `Comment`. Рисунок 16.3 показывает, какие типы дочерних узлов допустимы для соответствующих родительских узлов. Узлы, показанные серым, не могут иметь дочерних узлов.

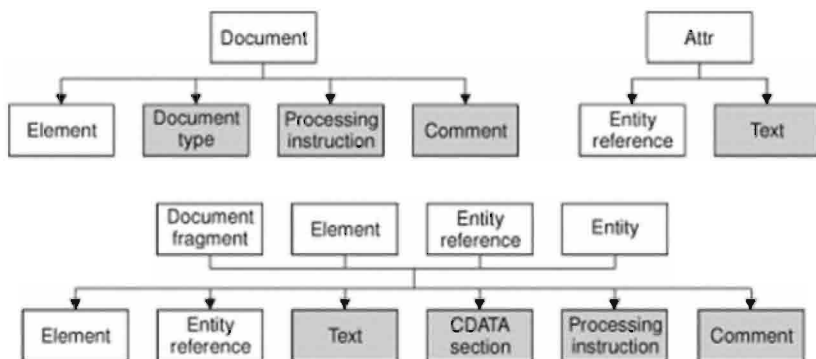


Рис. 16.3. Родственные связи между узлами DOM

Для иллюстрации применения DOM при чтении файлов XML мы напишем парсер для файла предметного указателя книги, описанного в предыдущем разделе.

```

class DomParser
{
public:
    DomParser(QTreeWidget *tree);
    bool readFile(const QString &fileName);

private:
    void parseBookindexElement(const QDomElement &element);
    void parseEntryElement(const QDomElement &element,
                           QTreeWidgetItem *parent);
    void parsePageElement(const QDomElement &element);
    void parsePageElement(const QDomElement &element,
                           QTreeWidgetItem *parent);
    QTreeWidget *treeWidget;
};
  
```

Мы определяем класс с названием `DomParser`, который выполняет синтаксический анализ предметного указателя книги, представленного в виде файла XML, и отображает результат в виджете `QTreeWidget`.

```
DomParser::DomParser(QTreeWidget *tree)
{
    treeWidget = tree;
}
```

В конструкторе мы просто присваиваем данный древовидный виджет переменной-члену. Вся обработка выполняется в функции readFile().

```
bool DomParser::readFile(const QString &fileName)
{
    QFile file(fileName);
    if (!file.open(QFile::ReadOnly | QFile::Text)) {
        std::cerr << "Error: Cannot read file " << qPrintable(fileName)
            << ": " << qPrintable(file.errorString())
            << std::endl;
        return false;
    }

    QString errorStr;
    int errorLine;
    int errorColumn;
    QDomDocument doc;
    if (!doc.setContent(&file, false, &errorStr, &errorLine,
        &errorColumn)) {
        std::cerr << "Error: Parse error at line" << errorLine << ", "
            << "column" << errorColumn << ": "
            << qPrintable(errorStr) << std::endl;

        return false;
    }

    QDomElement root = doc.documentElement();
    if (root.tagName() != "bookindex") {
        std::cerr << "Error: Not a bookindex file" << std::endl;
        return false;
    }
    parseBookindexElement(root);
    return true;
}
```

В функции readFile() начинаем с попытки открыть файл, имя которого передано в функцию. Если возникает ошибка, мы выводим сообщение об ошибке и возвращаем значение false, обозначающее ошибку. В противном случае задаем несколько переменных для хранения информации об ошибках обработки, если они потребуются, а затем создаем объект QDomDocument. Как только мы вызовем функцию setContent() применительно к DOM-документу, весь XML-документ, предоставленный устройством QIODevice, будет прочитан и обработан. Функция setContent() автоматически открывает устройство, если оно еще не открыто. Передача в функцию setContent() аргумента false отключает обработку пространств имен. Обратитесь к справочной документации по QtXml, где дается введение в обработку пространств имен и как это делается в Qt.

Если возникает ошибка, мы выводим сообщение об ошибке и возвращаем значение `false`, обозначающее ошибку. Если обработка прошла успешно, мы вызываем функцию `documentElement()` для объекта `QDomDocument`, чтобы получить его одиночный дочерний элемент `QDomElement`, после чего мы проверяем, является ли данный элемент `<bookindex>`. Если мы имеем элемент `<bookindex>`, мы вызываем для его обработки функцию `parseBookindexElement()`. Как и в предыдущем разделе, обработка осуществляется методом рекурсивного спуска.

```
void DomParser::parseBookindexElement(const QDomElement &element)
{
    QDomNode child = element.firstChild();
    while (!child.isNull()) {
        if (child.toElement().tagName() == "entry")
            parseEntryElement(child.toElement(),
                               treeWidget->invisibleRootItem());
        child = child.nextSibling();
    }
}
```

В функции `parseBookindexElement()` мы выполняем цикл по всем дочерним узлам. Мы ожидаем, что каждый узел будет представлять собой элемент `<entry>`, и мы вызываем функцию `parseEntry()` для синтаксического анализа узла. Мы игнорируем неизвестные узлы, чтобы формат предметного указателя можно было в будущем расширять, а старые парсеры при этом продолжали работать. Все узлы `<entry>`, являющиеся прямыми потомками узла `<bookindex>` являются узлами верхнего уровня древовидного виджета, который мы заполняем для визуализации дерева DOM, так что когда мы хотим обработать любой такой узел, мы передаем как сам узел, так и невидимый корневой элемент дерева, который будет родителем элемента дерева виджета.

Класс `QDomNode` может хранить узлы любого типа. Если мы хотим продолжить обработку узла, мы должны сначала преобразовать его в правильный тип данных. В нашем примере нас интересуют только узлы `Element`, и поэтому мы вызываем функцию `toElement()` объекта `QDomNode` для преобразования его в объект `QDomElement` и затем вызова функции `tagName()` для получения имени тега элемента. Если данный узел не имеет тип `Element`, функция `toElement()` возвращает нулевой объект типа `QDomElement`, содержащий пустое имя тега.

```
void DomParser::parseEntryElement(const QDomElement &element,
                                   QTreeWidgetItem *parent)
{
    QTreeWidgetItem *item = new QTreeWidgetItem(treeWidget);
    item->setText(0, element.attribute("term"));

    QDomNode child = element.firstChild();
    while (!child.isNull()) {
        if (child.toElement().tagName() == "entry") {
            parseEntryElement(child.toElement(), item);
        } else if (child.toElement().tagName() == "page") {
```

```

        parsePageElement(child.toElement(), item);
    }
    child = child.nextSibling();
}
}

```

В функции `parseEntryElement()` мы создаем элемент древовидного виджета. Передаваемый родительский элемент представляет собой невидимый корневой элемент (если это элемент верхнего уровня), либо другой элемент `<entry>` (если это подэлемент). Мы вызываем функцию `setText()`, чтобы задать текст, отображаемый в первом столбце, беря его из атрибута `term` тега `<entry>`.

После инициализации элемента `QTreeWidgetItem` мы перебираем в цикле узлы-потомки узла `QDomElement`, соответствующие текущему тегу `<entry>`. Для каждого элемента, представляющего собой тег `<entry>`, мы выполняем рекурсивный вызов функции `parseEntryElement()`, передавая ей в качестве второго аргумента текущий элемент. Затем будут создаваться элементы-потомки `QTreeWidgetItem`, родителем для которых будет текущий элемент `<entry>`. Если элементом-потомком оказывается `<page>`, мы вызываем функцию `parsePageElement()`.

```

void DomParser::parsePageElement(const QDomElement &element,
                                QTreeWidgetItem *parent)
{
    QString page = element.text();
    QString allPages = parent->text(1);
    if (!allPages.isEmpty())
        allPages += ", ";
    allPages += page;
    parent->setText(1, allPages);
}

```

В функции `parsePageElement()` мы вызываем для элемента функцию `text()`, чтобы получить текст, содержащийся между тегами `<page>` и `</page>`; затем мы добавляем текст в разделяемый запятыми список номеров страниц, содержащийся во втором столбце `QTreeWidgetItem`. Функция `QDomElement::text()` проходит по узлам-потомкам элемента и соединяет весь текст, хранящийся в узлах `Text` и `CDATA`.

Теперь давайте посмотрим, как можно использовать класс `DomParser` для обработки файла:

```

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QStringList args = QApplication::arguments();
    ...
    QTreeWidgetItem *treeWidget;
    ...
    DomParser parser(&treeWidget);
    for (int i = 1; i < args.count(); ++i)
        parser.readFile(args[i]);
    return app.exec();
}

```

Мы начинаем с настройки `QTreeWidget`. Затем мы создаем объект `DomParser`. Для каждого файла, указанного в командной строке, мы вызываем функцию `DomParser::readFile()` обработки файла и пополнения виджета дерева.

Как и в предыдущем примере для сборки приложения с библиотекой *QtXml* в файл `.pro` необходимо добавить следующую строку:

```
QT += xml
```

Как показывает наш пример, навигация по дереву DOM достаточно простая, хотя и не такая удобная, как в случае `QXmlStreamReader`. Программисты, которым очень часто приходится использовать интерфейс DOM, создают свои собственные высокоуровневые функции-оболочки для упрощения выполнения наиболее распространенных операций.

Чтение документов XML при помощи интерфейса SAX

SAX является фактическим стандартом программного интерфейса с открытым исходным кодом, который обеспечивает чтение документов XML.

Классы Qt для интерфейса SAX моделируют реализацию SAX2 Java с некоторыми отличиями в названиях для обеспечения принятых в Qt правил обозначений названий классов и их членов. По сравнению с интерфейсом DOM, SAX является более низкоуровневым и обычно работает быстрее. Однако поскольку класс `QXmlStreamReader`, представленный в данной главе, предоставляет интерфейс программирования (API), более соответствующий стилю Qt, и работает быстрее парсера SAX, главная область применения парсера SAX – это перенос кода, использующего API SAX, в Qt. Более подробную информацию относительно SAX можно получить в сети Интернет по адресу <http://www.saxproject.org/>.

Qt обеспечивает построенный на основе интерфейса SAX парсер документов XML, не предусматривающий проверку их достоверности под названием `QXmlSimpleReader`. Этот парсер распознает хорошо сформированные документы XML и поддерживает пространства имен XML. Когда парсер обрабатывает документ, он вызывает виртуальные функции в зарегистрированных классах-работчиках, уведомляющих о возникновении соответствующих событий в ходе синтаксического анализа документа. (Эти события никак не связаны с такими событиями Qt, как события клавиатуры и события мыши.) Например, пусть парсер выполняет анализ следующего документа XML:

```
<doc>
  <quote> Gnothi seauton</quote>
</doc>
```

В этом случае парсер вызовет следующие обработчики событий синтаксического анализа:

```
startDocument()
startElement("doc")
startElement("quote")
characters("Ars longa vita brevis")
endElement("quote")
endElement("doc")
endDocument()
```

Все приведенные выше функции объявлены в классе `QXmlContentHandler`. Для простоты мы не стали указывать некоторые аргументы функций `startElement()` и `endElement()`.

`QXmlContentHandler` — это всего лишь один из многих классов-обработчиков, которые могут использоваться совместно с классом `QXmlSimpleReader`. Другими такими классами являются `QXmlEntityResolver`, `QXmlDTDHandler`, `QXmlErrorHandler`, `QXmlDeclHandler` и `QXmlLexicalHandler`. Эти классы только объявляют чистые виртуальные функции и предоставляют информацию о различных событиях синтаксического анализа. Для большинства приложений вполне достаточно использовать лишь классы `QXmlContentHandler` и `QXmlErrorHandler`.

Для удобства Qt также предоставляет класс `QXmlDefaultHandler`, который происходит от всех классов-обработчиков и обеспечивает очень простую реализацию всех функций. Такая конструкция со множеством абстрактных классов-обработчиков и одним подклассом с тривиальной реализацией функций необычна для Qt; она принята для максимального соответствия модели Java-реализации.

Наиболее важное различие между использованием API SAX и API `QXmlStreamReader` или DOM состоит в том, что API SAX заставляет нас вручную отслеживать состояние парсера с помощью переменных-членов, что является необязательным при использовании двух других подходов, где используется рекурсивный спуск.

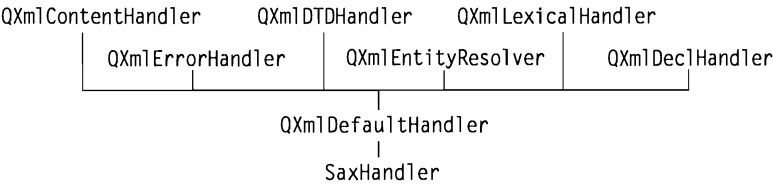


Рис. 16.4. Дерево наследования SaxHandler

Чтобы проиллюстрировать использование интерфейса SAX для чтения XML-файлов, мы напишем парсер для описанного ранее в этой главе формата файла предметного указателя книги. Здесь мы будем проводить синтаксический анализ с помощью классов `QXmlSimpleReader` и `SaxHandler`, являющегося подклассом `QXmlDefaultHandler`.

Первый этап в реализации парсера заключается в создании подкласса `QXmlDefaultHandler`:

```

class SaxHandler : public QXmlDefaultHandler
{
public:
    SaxHandler(QTreeWidget *tree);
    bool readFile(const QString &fileName);
protected:
    bool startElement(const QString &namespaceURI,
                     const QString &localName,
                     const QString &qName,
                     const QXmlAttributes &attributes);
    bool endElement(const QString &namespaceURI,

```

```

        const QString &localName,
        const QString &qName);
    bool characters(const QString &str);
    bool fatalError(const QDomParseException &exception);

private:
    QTreeWidget *treeWidget;
    QTreeWidgetItem *currentItem;
    QString currentText;
};

```

Класс SaxHandler происходит от `QXmlDefaultHandler` и переопределяет четыре функции: `startElement()`, `endElement()`, `characters()` и `fatalError()`. Первые четыре функции объявлены в `QXmlContentHandler`; последняя функция объявлена в `QXmlErrorHandler`.

```

SaxHandler::SaxHandler(QTreeWidget *tree)
{
    treeWidget = tree;
}

```

Конструктор SaxHandler принимает объект типа `QTreeWidget`, который мы собираемся заполнять информацией, содержащейся в файле XML.

```

bool SaxHandler::readFile(const QString &fileName)
{
    currentItem = 0;
    QFile file(fileName);
    QDomInputSource inputSource(&file);
    QDomSimpleReader reader;
    reader.setContentHandler(this);
    reader.setErrorHandler(this);
    return reader.parse(inputSource);
}

```

Эта функция вызывается, когда у нас есть имя файла, для которого нужно произвести синтаксический анализ. Мы создаем для этого файла объект `QFile` и создаем объект `QXmlInputSource` для чтения содержимого файла. Затем мы создаем объект `QXmlSimpleReader` для обработки файла. Мы указываем обработчику ошибок и обработчику содержимого объекта `reader` на данный класс (`SaxHandler`), а затем вызываем функцию `parse()` для обработки файла. В классе `SaxHandler` мы просто переопределяем функции из классов `QXmlContentHandler` и `QXmlErrorHandler`. Если бы мы реализовали функции из других классов-обработчиков, нам также нужно было бы вызывать соответствующие функции `setXxxHandler()`.

Вместо того чтобы передавать в функцию `parse()` простой объект `QFile`, мы передаем объект `QXmlInputSource`. Этот класс принимает переданный ему файл, читает его (учитывая любые кодировки символов, указанные в объявлении `<?xml?>`), а также предоставляет интерфейс, с помощью которого парсер читает файл.

```
bool SaxHandler::startElement(const QString & /* namespaceURI */,
                              const QString & /* localName */,
                              const QString &qName,
                              const QDomAttributes &attributes)
{
    if (qName == "entry") {
        currentItem = new QTreeWidgetItem(currentItem ?
            treeWidget->invisibleRootItem());
        currentItem->setText(0, attributes.value("term"));
    } else if (qName == "page") {
        currentText.clear();
    }
    return true;
}
```

Функция `startElement()` вызывается, когда обнаруживается новый открывающий тег. Третий параметр представляет собой имя тега (или точнее «подходящее имя»). В четвертом параметре задается список атрибутов. В этом примере мы игнорируем первый и второй параметры. Они полезны для тех файлов XML, которые используют механизм пространств имен, который подробно описан в справочной документации.

Если обнаружен тег `<entry>`, мы создаем новый элемент списка `QTreeWidgetItem`. Если данный тег является вложенным в другой тег `<entry>`, новый тег определяет подэлемент предметного указателя, и новый элемент `QTreeWidgetItem` создается как дочерний по отношению к внешнему элементу `QTreeWidgetItem`. В противном случае мы создаем элемент `QTreeWidgetItem`, используя в качестве родительского элемента объект `treeWidget`, делая его элементом верхнего уровня. Мы вызываем функцию `setText()` для отображения в столбце 0 текста со значением атрибута `term` тега `<entry>`.

Если обнаружен тег `<page>`, мы устанавливаем значение переменной `currentText` на пустую строку. В переменной `currentText` накапливается текст, расположенный между тегами `<page>` и `</page>`.

В конце мы возвращаем `true`, указывая SAX на необходимость продолжения синтаксического анализа файла. Если бы нам нужно было сообщить об ошибке из-за обнаружения неизвестного тега, мы возвращали бы в этих случаях `false`. Нам также потребовалось бы переопределить функцию `errorString()` класса `QXmlDefaultHandler` для возврата соответствующего сообщения об ошибке.

```
bool SaxHandler::characters(const QString &str)
{
    currentText += str;
    return true;
}
```

Функция `characters()` используется для извлечения символьных данных из документа XML. Мы просто добавляем символы в конец переменной `currentText`.

```

bool SaxHandler::endElement(const QString & /* namespaceURI */,
                           const QString & /* localName */,
                           const QString &qName)
{
    if (qName == "entry") {
        currentItem = currentItem->parent();
    } else if (qName == "page") {
        if (currentItem) {
            QString allPages = currentItem->text(1);
            if (!allPages.isEmpty())
                allPages += ", ";
            allPages += currentItem->text();
            currentItem->setText(1, allPages);
        }
    }
    return true;
}

```

Функция `endElement()` вызывается при обнаружении закрывающего тега. Также как и для функции `startElement()`, ее третий параметр содержит имя тега.

Если обнаружен тег `</entry>`, мы устанавливаем закрытую переменную `currentItem` на родительский элемент текущего элемента `QTreeWidgetItem`. (По причинам исторического характера высокоуровневые элементы возвращают в качестве своего родителя 0, а не невидимый корневой элемент). Это обеспечивает восстановление переменной `currentItem` на значение, которое она имела перед чтением соответствующего тега `<entry>`.

Если обнаружен тег `</page>`, мы добавляем указанный номер страницы или диапазон страниц в разделяемый запятыми список в столбце 1 текущего элемента.

```

bool SaxHandler::fatalError(const QDomParseException &exception)
{
    std::cerr << "Parse error at line " << exception.lineNumber()
               << ", " << "column " << exception.columnNumber() << ": "
               << qPrintable(exception.message()) << std::endl;
    return false;
}

```

Функция `fatalError()` вызывается, когда синтаксический анализ файла XML завершается неудачей. В этом случае мы просто выводим на экран сообщение, указывая номер строки, номер столбца и текст об ошибке синтаксического анализа.

Этим мы завершаем реализацию класса `SaxHandler`. Функция `main()`, которая его использует, практически идентична той, которую мы рассматривали в предыдущем разделе, посвященном `DomParser`, с той разницей, что используется `SaxHandler`, а не `DomHandler`.

Запись документов XML

В большинстве приложений, которые могут читать XML-файлы, также существует необходимость записывать такие файлы. Существуют три основных подхода к формированию файлов XML в приложениях Qt:

- мы можем использовать класс `QXmlStreamWriter`;
- мы можем построить дерево DOM и вызвать для него функцию `save()`;
- мы можем сформировать файл XML вручную.

Выбор между этими подходами часто не зависит от типа используемого нами интерфейса для чтения документов XML: `QXmlStreamReader` SAX или DOM, хотя данные, хранящиеся в дереве DOM часто имеет смысл сохранять, сохраняя дерево напрямую.

Записывать XML-документы при помощи класса `QXmlStreamWriter` особенно просто, поскольку этот класс сам обеспечивает обработку специальных символов. Если бы нам было нужно вывести данные о предметном указателе книги из `QTreeWidgetItem` при помощи `QXmlStreamWriter`, мы могли бы сделать это при помощи всего двух функций. Первая функция принимает имя файла и указатель `QTreeWidgetItem *`, и перебирает все элементы верхнего уровня в дереве:

```
bool writeXml(const QString &fileName, QTreeWidgetItem *treeWidget)
{
    QFile file(fileName);
    if (!file.open(QFile::WriteOnly | QFile::Text)) {
        std::cerr << "Error: Cannot write file "
                    << qPrintable(fileName) << ": "
                    << qPrintable(file.errorString()) << std::endl;
        return false;
    }
    QXmlStreamWriter xmlWriter(&file);
    xmlWriter.setAutoFormatting(true);
    xmlWriter.writeStartDocument();
    xmlWriter.writeStartElement("bookindex");
    for (int i = 0; i < treeWidget->topLevelItemCount(); ++i)
        writeIndexEntry(&xmlWriter, treeWidget->topLevelItem(i));
    xmlWriter.writeEndDocument();
    file.close();
    if (file.error()) {
        std::cerr << "Error: Cannot write file "
                    << qPrintable(fileName) << ": "
                    << qPrintable(file.errorString()) << std::endl;
        return false;
    }
    return true;
}
```

Если мы включим автоматическое форматирование, XML будет выводиться в более удобном виде, с отступами, показывающими рекурсивную структуру данных. Функция `writeStartDocument()` запишет строку заголовка XML-документа

```
<?xml version="1.0" encoding="UTF-8"?>
```

Функция `writeStartElement()` генерирует новый начальный тег с указанным текстом тега. Функция `writeEndDocument()` закрывает все открытые стартовые теги. Для каждого элемента верхнего уровня мы вызываем функцию `writeIndexEntry()`,

передавая ей объект `QXmlStreamWriter` и элемент для вывода. Ниже приводится код функции `writeIndexEntry()`:

```
void writeIndexEntry(QXmlStreamWriter *xmlWriter, QTreeWidgetItem *item)
{
    xmlWriter->writeStartElement("entry");
    xmlWriter->writeAttribute("term", item->text(0));
    QString pageString = item->text(1);
    if (!pageString.isEmpty()) {
        QStringList pages = pageString.split(", ");
        foreach (QString page, pages)
            xmlWriter->writeTextElement("page", page);
    }
    for (int i = 0; i < item->childCount(); ++i)
        writeIndexEntry(xmlWriter, item->child(i));
    xmlWriter->writeEndElement();
}
```

Эта функция создает элемент `<entry>`, соответствующий элементу `QTreeWidgetItem`, который был ей передан в качестве параметра. Функция `writeAttribute()` добавляет атрибут в только что записанный тег; например, она может превратить элемент `<entry>` в `<entry term="sidebearings">`. Если присутствуют номера страниц, они разделяются по местам запятых с пробелом, и для каждого записывается отдельная пара тегов `<page>...</page>`, внутрь которых заключается текст, обозначающий страницу. Для всего этого нужно вызвать функцию `writeTextElement()` и передать ей имя тега и текст, который будет заключен между начальным и конечным тегами. Во всех случаях класс `QXmlStreamWriter` позаботится об обработке специальных символов XML, так что нам не придется думать об этом.

Если элемент имеет дочерние элементы, мы выполняем рекурсивный вызов функции `writeIndexEntry()` для каждого из них. Наконец, мы вызываем функцию `call writeEndElement()`, чтобы записать тег `</entry>`.

Использование класса `QXmlStreamWriter` для записи XML является наиболее быстрым и безопасным подходом, но если мы уже имеем XML-код в форме дерева DOM, мы можем просто «попросить» это дерево записать нужный XML-код, вызвав функцию `save()` объекта `QDomDocument`. По умолчанию функция `save()` использует в генерируемом файле кодировку UTF-8, но мы можем использовать и другую кодировку, добавив к началу объявления тега `<?xml?>` следующую информацию:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

в начало дерева DOM. Следующий фрагмент программного кода показывает, как это делать:

```
const int Indent = 4;
QDomDocument doc;
...
QTextStream out(&file);
QDomNode xmlNode = doc.createProcessingInstruction("xml",
    "version=\"1.0\" encoding=\"ISO-8859-1\"");
```

```
doc.insertBefore(xmlNode, doc.firstChild());  
doc.save(out, Indent);
```

Начиная с Qt 4.3, альтернативой является указание кодировки в объекте QTextStream с помощью функции setCodec() и передача объекта QDomNode::EncodingFromTextStream в качестве параметра функции save().

Формирование файлов XML вручную выполняется не намного сложнее, чем при помощи DOM. Мы можем использовать QTextStream и писать строки, как мы бы делали с любым другим текстовым файлом. Наиболее сложным является вставка специальных символов в текст и значения атрибутов. Функция Qt::escape() заменяет символы «<», «>» и «&». Ниже приводится пример ее использования:

```
QTextStream out(&file);  
out.setCodec("UTF-8");  
out << "<doc>\n"  
    << "          <quote>" << Qt::escape(quoteText) << "</quote>\n"  
    << "          <translation>" << Qt::escape(translationText)  
    << "</translation>\n"  
    << "</doc>\n";
```

При генерации подобных XML-файлов помимо написания объявления и присвоения в правильной кодировке, мы должны также помнить об использовании escape-символов в создаваемом тексте, а если мы используем атрибуты, мы должны применять escape-символы для одинарных и двойных кавычек в атрибутах. Использовать класс QDomStreamWriter гораздо проще, поскольку он сам заботится об этом.



- *Всплывающие подсказки, комментарии в строке состояния и справки «что это такое?»*
- *Использование web-браузера для предоставления интерактивной помощи*
- *Использование QTextBrowser в качестве простого браузера системы помощи*
- *Использование Qt Assistant для мощной интерактивной системы помощи*

Глава 17. Обеспечение интерактивной помощи

Большинство приложений предоставляют своим пользователям систему помощи, работающую в интерактивном режиме. В некоторых случаях эта помощь носит форму коротких сообщений, например, в виде всплывающих подсказок, комментариев в строке состояния и справок «что это такое?» Все это, естественно, поддерживается в Qt. В других случаях система помощи может быть значительно сложнее и может содержать много страниц текста. Для такого рода систем вы можете воспользоваться классом QTextBrowser в качестве простого браузера системы помощи, а также вы можете вызывать из вашего приложения Qt Assistant или другой браузер файлов HTML.

Всплывающие подсказки, комментарии в строке состояния и справки «что это такое?»

Всплывающая подсказка (tooltip) представляет собой небольшое текстовое сообщение, которое появляется при нахождении курсора мыши на виджете в течение определенного времени. Всплывающие подсказки отображаются на желтом фоне черными буквами. В основном они предназначены для пояснения назначения кнопок на панели инструментов.

Мы можем добавлять всплывающие подсказки к любым виджетам путем включения в программный код вызова функции QWidget::setToolTip(). Например:

```
findButton->setToolTip(tr("Find next"));
```

Для установки всплывающей подсказки для объекта QAction, который может быть добавлен к меню или панели инструментов, мы можем просто вызвать функцию setToolTip() для этой команды. Например:

```
newAction = new QAction(tr("&New"), this);  
newAction->setToolTip(tr("New document"));
```

Если мы явно не устанавливаем всплывающую подсказку, QAction автоматически сформирует ее на основе текста команды.

Комментарии в строке состояния (status tip) также представляют собой короткие текстовые сообщения, причем они обычно немного длиннее всплывающих подсказок. При нахождении курсора мыши на кнопке панели инструментов или на строке меню такой комментарий появляется в строке состояния. Для добавления к команде или к виджету отображаемого в строке состояния комментария необходимо вызвать функцию setStatusTip():

```
newAction->setStatusTip(tr("Create a new document"));
```

На рис. 17.1 показана всплывающая подсказка и комментарий в строке состояния в приложении.

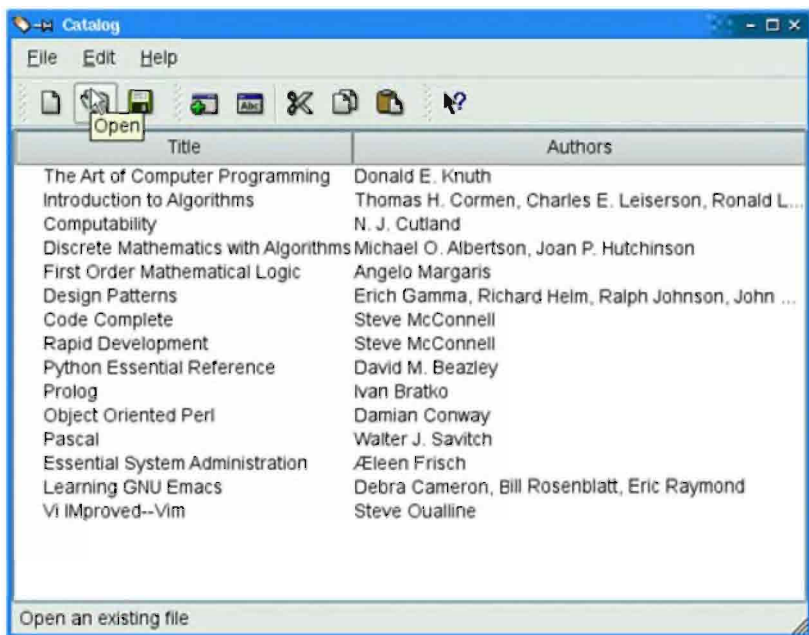


Рис. 17.1. В этом приложении отображаются всплывающая подсказка и комментарий в строке состояния

В некоторых ситуациях желательно обеспечить больше информации о виджете, чем это возможно сделать с помощью всплывающих подсказок или комментариев в строке состояния. Например, у нас может возникнуть потребность в обеспечении сложного диалогового окна с пояснительным текстом для каждого поля без принуждения пользователя к вызову отдельного окна системы помощи. Режим «что это такое?» идеально подходит для этого. Когда окно находится в режиме «что это такое?», курсор приобретает форму  и пользователь может щелкнуть по любому компоненту интерфейса пользователя для получения текста помощи. Для входа в режим «что это такое?» пользователь может либо нажать на кнопку ? в строке заголовка диалогового окна (в системе Windows и KDE) или нажать сочетание клавиш Shift+F1.

Ниже приводится пример установки для диалогового окна текста справки «что это такое?»:

```
dialog->setWhatsThis(tr("<img src=\":/images/icon.png\">"
    "&nbsp;The meaning of the Source field depends "
    "on the Type field:"
    "<ul>"
    "<li><b>Books</b> have a Publisher"
    "<li><b>Articles</b> have a Journal name with "
    "volume and issue number"
    "<li><b>Theses</b> have an Institution name "
    "and a Department name"
    "</ul>"));
```

Мы можем применять теги HTML для форматирования текста справки «что это такое?». В нашем примере, показанном на рис. 17.2, мы используем изображение (которое указано в файле ресурсов приложения), маркированный список и жирный текст в некоторых местах. Теги и атрибуты, которые поддерживаются в Qt, приведены на веб-странице <http://doc.trolltech.com/4.3/richtext-html-subset.html>.

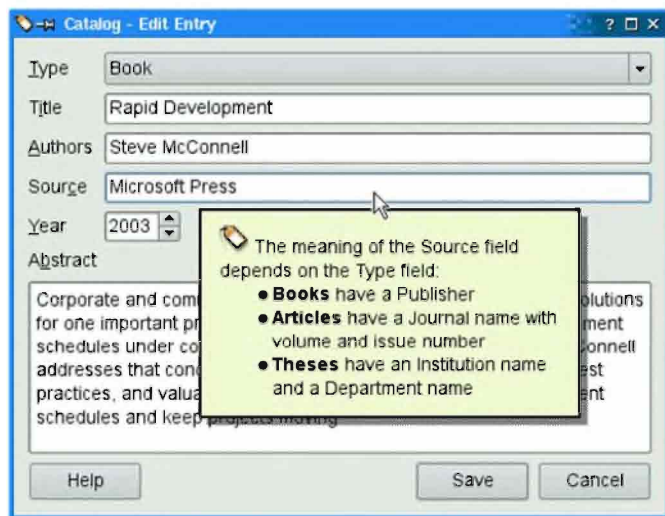


Рис. 17.2. Диалоговое окно с отображением текста справки «что это такое?»

При задании для команды текста справки «что это такое?» он будет отображаться, когда пользователь в режиме справки «что это такое?» выбирает пункт меню, нажимает кнопку на панели инструментов или клавишу быстрого вызова команды. Когда компоненты пользовательского интерфейса главного окна приложения предусматривают вывод справки «что это такое?», обычно в меню Help (справка) содержится пункт What's This? (что это такое?) и панель инструментов содержит соответствующую кнопку. Это можно сделать путем создания команды What's This? при помощи статической функции `QWhatsThis::createAction()`

и добавления возвращаемой ею команды в меню Help и в панель инструментов. Класс `QWhatsThis` предоставляет также статические функции для программного входа и выхода из режима справки «что это такое?».

Использование web-браузера для предоставления интерактивной помощи

В больших приложениях может понадобиться больше справочной информации, чем могут предоставить всплывающие подсказки, комментарии в строке состояния и режим «что это такое?». Простым решением будет создать справочный текст в формате HTML и загрузить эту страницу в браузер пользователя.

В приложениях, использующих справку на основе браузера, как правило, имеется пункт Help раздела Help главного меню, а также кнопка Help в каждом диалоговом окне. В этом разделе мы покажем, как с помощью класса `QDesktopServices` обеспечить функционирование таких кнопок.

Главное окно приложения, как правило, содержит слот `help()`, который вызывается, когда пользователь нажимает F1 или выбирает пункт меню Help|Help.

```
void MainWindow::help()
{
    QUrl url(directoryOf("doc").absoluteFilePath("index.html"));
    url.setScheme("file");
    QDesktopServices::openUrl(url);
}
```

В этом примере мы предполагаем, что наши справочные HTML-файлы находятся в поддиректории `doc`. Функция `QDir::absoluteFilePath()` возвращает объект `QString`, содержащий полный путь к файлу с указанным именем. Поскольку это справка для главного окна, мы будем использовать файл индекса (`index.html`) справочной системы, через который можно обращаться по гиперссылкам к другим справочным файлам. Далее мы задаем для схемы URL значение «file», чтобы поиск заданного файла производился в локальной файловой системе. Наконец, мы используем удобную статическую функцию класса `openUrl()` для запуска пользовательского web-браузера.

Мы не знаем, какой web-браузер будет использоваться, поэтому мы должны постараться сделать наш HTML-код правильным и совместимым с браузерами, которые могут применять пользователи. Большинство web-браузеров задают локальную рабочую директорию в соответствии с путем, заданным в URL и, следовательно, предполагают, что любые изображения и гиперссылки, которые не имеют абсолютных путей, в качестве корневой используют эту рабочую директорию. Все это означает, что мы должны помещать все наши HTML-файлы и изображения в директорию `doc` (или в ее поддиректории), и делать все ссылки относительными, за исключением ссылок на внешние web-сайты.

```
QDir MainWindow::directoryOf(const QString &subdir)
{
    QDir dir(QApplication::applicationDirPath());
```

```

    #if defined(Q_OS_WIN)
    if (dir.dirName().toLowerCase() == "debug"
        || dir.dirName().toLowerCase() == "release")
        dir.cdUp();
    #elif defined(Q_OS_MAC)
    if (dir.dirName() == "MacOS") {
    dir.cdUp();
    dir.cdUp();
    dir.cdUp();
    }
    #endif
    dir.cd(subdir);
    return dir;
}

```

Статическая функция `directoryOf()` возвращает объект `QDir`, соответствующий поддиректории относительно директории приложения. В Windows исполняемые файлы приложения обычно находятся в поддиректории `debug` или `release`, и в этом случае мы перемещаемся на одну директорию вверх. В Mac OS мы учитываем пакетную структуру директорий.

В диалоговых окнах нам, как правило, нужно загрузить в web-браузер определенную страницу из нашей справочной системы, а, может быть, и конкретное место на странице. Например:

```

void EntryDialog::help()
{
    QUrl url(directoryOf("doc").absoluteFilePath("forms.html"));
    url.setScheme("file");
    url.setFragment("editing");
    QDesktopServices::openUrl(url);
}

```

Данный слот вызывается из диалогового окна, когда пользователь щелкает по кнопке `Help`. Этот пример похож на предыдущий, за исключением того, что мы выбираем другую начальную страницу. Эта страница содержит справочный текст по нескольким разным формам, где метками (`anchor`) (например, `<a name="editing">`) в HTML показано, где начинается текст, относящийся к каждой форме. Для доступа к конкретному месту на странице, мы вызываем функцию `setFragment()` и передаем ей метку, до которой мы хотим прокрутить страницу.

Предоставление справочных файлов в формате HTML, и обеспечение доступа к ним для пользователя через web-браузер — метод простой и удобный. Однако web-браузеры не могут обращаться к ресурсам приложения (например, значкам), и их не так-то легко настроить под конкретное приложение. Также, если мы переходим к конкретной странице, как мы сделали в случае с `EntryDialog`, кнопки `Home` и `Back` браузера не будут оказывать желаемого эффекта.

Использование QTextBrowser в качестве простого браузера системы помощи

Использовать web-браузер пользователя для отображения интерактивной справки весьма просто, но, как мы отметили, этот метод имеет ряд недостатков. Мы можем устранить эти проблемы, предоставив нашу собственную систему справки, основанную на использовании класса QTextBrowser.

В данном разделе мы представим простой браузер системы помощи, показанный на рис. 17.3, и покажем, как его можно использовать в приложении. Окно приложения применяет QTextBrowser для вывода на экран страниц справки, представленных в формате HTML. QTextBrowser может обрабатывать много тегов HTML, и поэтому он идеально подходит для этих целей.

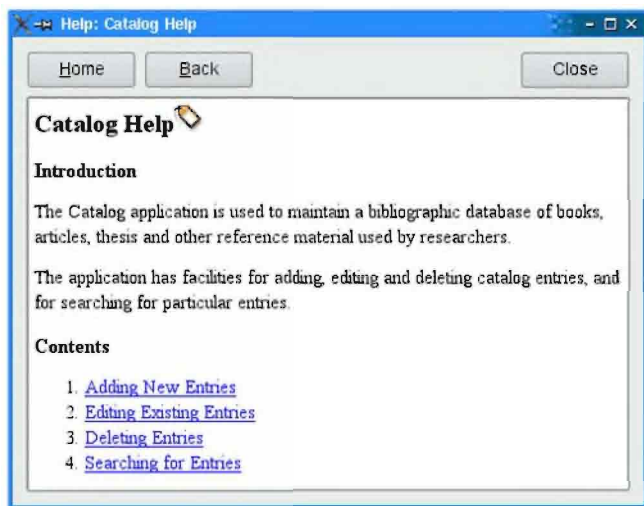


Рис. 17.3. Виджет HelpBrowser

Мы начинаем с определения класса:

```
class HelpBrowser : public QWidget
{
    Q_OBJECT
public:
    HelpBrowser(const QString &path, const QString &page,
                QWidget *parent = 0);
    static void showPage(const QString &page);
private slots:
    void updateWindowTitle();
private:
    QTextBrowser *textBrowser;
    QPushButton *homeButton;
```



```

        QPushButton *backButton;
        QPushButton *closeButton;
};

```

Класс HelpBrowser содержит статическую функцию, которую можно вызывать в любом месте в приложении. Данная функция создает окно HelpBrowser и выводит на экран заданную страницу.

Ниже приводится конструктор:

```

HelpBrowser::HelpBrowser(const QString &path, const QString &page,
                        QWidget *parent)
    : QWidget(parent)
{
    setAttribute(Qt::WA_DeleteOnClose);
    setAttribute(Qt::WA_GroupLeader);

    textBrowser = new QTextBrowser;

    homeButton = new QPushButton(tr("&Home"));
    backButton = new QPushButton(tr("&Back"));
    closeButton = new QPushButton(tr("Close"));
    closeButton->setShortcut(tr("Esc"));

    QHBoxLayout *buttonLayout = new QHBoxLayout;
    buttonLayout->addWidget(homeButton);
    buttonLayout->addWidget(backButton);
    buttonLayout->addStretch();
    buttonLayout->addWidget(closeButton);

    QVBoxLayout *mainLayout = new QVBoxLayout;
    mainLayout->addLayout(buttonLayout);
    mainLayout->addWidget(textBrowser);
    setLayout(mainLayout);

    connect(homeButton, SIGNAL(clicked()), textBrowser, SLOT(home()));
    connect(backButton, SIGNAL(clicked()),
            textBrowser, SLOT(backward()));
    connect(closeButton, SIGNAL(clicked()), this, SLOT(close()));
    connect(textBrowser, SIGNAL(sourceChanged(const QUrl &)),
            this, SLOT(updateWindowTitle()));

    textBrowser->setSearchPaths(QStringList() << path << ":/images");
    textBrowser->setSource(page);
}

```

Мы устанавливаем атрибут Qt::WA_GroupLeader, потому что хотим выдавать окна HelpBrowser не только из главного окна, но также из модальных диалоговых окон. Обычно модальные диалоговые окна не позволяют пользователям работать с другими окнами приложения. Однако очевидно, что после запроса помощи

пользователь должен иметь возможность работать как с модальным диалоговым окном, так и с браузером системы помощи. Установка атрибута `Qt::WA_GroupLeader` делает возможным такой режим работы.

Мы обеспечиваем два пути поиска: первый определяет путь в файловой системе к документации приложения, а второй определяет расположение ресурсов изображений. HTML может содержать обычные ссылки на изображения в файловой системе и ссылки на ресурсы изображений, пути к которым начинаются с символов `:/` (двоеточие и слеш). Параметр `page` содержит имя файла документации с возможным указанием метки HTML (`anchor`).

```
void HelpBrowser::updateWindowTitle()
{
    setWindowTitle(tr("Help: %1").arg(textBrowser->documentTitle()));
}
```

При всяком изменении исходной страницы вызывается слот `updateWindowTitle()`. Функция `documentTitle()` возвращает текст, содержащийся в теге `<title>` этой страницы.

```
void HelpBrowser::showPage(const QString &page)
{
    QString path = directoryOf("doc").absolutePath();
    HelpBrowser *browser = new HelpBrowser(path, page);
    browser->resize(500, 400);
    browser->show();
}
```

В статической функции `showPage()` мы создаем окно `HelpBrowser` и затем выдаем его на экран. Это окно будет удалено автоматически, когда пользователь закроет его, поскольку мы установили в конструкторе `HelpBrowser` атрибут `Qt::WA_DeleteOnClose`.

В этом примере мы предполагаем, что документация располагается в каталоге `doc` приложения. Все страницы, передаваемые функции `showPage()`, будут браться из этого подкаталога.

Теперь мы можем вызывать браузер системы помощи из приложения. В главном окне приложения мы могли бы создать команду `Help` и подсоединить ее к слоту `help()`, который может иметь следующий вид:

```
void MainWindow::help()
{
    HelpBrowser::showPage("index.html");
}
```

Здесь предполагается, что главный файл системы помощи имеет имя `index.html`. Для диалоговых окон мы могли бы подсоединить кнопку `Help` к слоту `help()`, который может иметь следующий вид:

```
void EntryDialog::help()
{
    HelpBrowser::showPage("forms.html#editing");
}
```

Здесь мы выводим на экран другой справочный файл, `forms.html`, и позиционируем браузер `QTextBrowser` на метку `editing`.

Также существует возможность использовать систему ресурсов Qt для встраивания справочных файлов и связанных с ними изображений прямо в исполняемый файл. Единственное, что нужно для этого, это добавить в файл `.qrc` приложения запись о каждом файле, который мы хотим встроить, и путь к ресурсу (например, `:/doc/forms.html#editing`).

В данном примере мы использовали оба подхода, встроив в исполняемый файл значки (поскольку они также используются в самом приложении), но оставив в файловой системе HTML-файлы. Такой метод имеет преимущество, связанное с возможностью обновления справочных файлов независимо от самого приложения, и при этом остается гарантия, что значки приложения будут найдены.

Использование Qt Assistant для мощной интерактивной системы помощи

Qt Assistant является свободно распространяемой интерактивной системой помощи, поддерживаемой фирмой Trolltech. Основным ее достоинством является поддержка индексации и поиск по всему тексту, а также возможность ее работы с наборами документации нескольких приложений.

Для применения *Qt Assistant* мы должны включить в наше приложение соответствующий программный код и указать *Qt Assistant* место расположения нашей документации¹.

Связь между приложением Qt и *Qt Assistant* обеспечивается классом `QAssistantClient`, который располагается в отдельной библиотеке. Для сборки этой библиотеки с нашим приложением мы должны добавить следующую строку к файлу приложения `.pro`:

```
CONFIG += assistant
```

Теперь мы рассмотрим программный код нового класса `HelpBrowser`, который использует *Qt Assistant*.

```
class HelpBrowser
{
public:
    static void showPage(const QString &page);

private:
    static QDir directoryOf(const QString &subdir);
    static QAssistantClient *assistant;
};
```

Ниже приводится новая реализация функции `showPage()`:

```
QAssistantClient *HelpBrowser::assistant = 0;
void HelpBrowser::showPage(const QString &page)
{
```

¹ Ожидается, что в Qt 4.4 будет обновленная система справки, которая упростит интеграцию документации.

```
QString path = directoryOf("doc").absoluteFilePath(page);  
if (!assistant)  
    assistant = new QAssistantClient("");  
assistant->showPage(path);  
}
```

Конструктор `QAssistantClient` принимает в качестве своего первого аргумента строку пути, который используется для определения места нахождения исполняемого модуля *Qt Assistant*. Передавая пустой путь, мы указываем на необходимость `QAssistantClient` поиска исполняемого модуля в путях переменной среды `PATH`. `QAssistantClient` имеет функцию `showPage()`, которая принимает имя файла страницы HTML с необязательным указанием метки позиции.

На следующем этапе необходимо подготовить оглавление и предметный указатель документации. Это выполняется путем создания профиля *Qt Assistant* и файла `.dcf`, который содержит сведения о документации. Все это объясняется в документации по *Qt Assistant*, и поэтому мы не станем здесь повторять эти сведения.

В качестве альтернативы web-браузеру, `QTextBrowser` или *Qt Assistant* можно использовать зависящие от платформы методы обеспечения интерактивной помощи. Для приложений Windows можно создать файлы системы помощи Windows HTML Help и обеспечить доступ к ним при помощи Internet Explorer компании Microsoft. Вы могли бы использовать для этого класс `Qt QProcess` или рабочую среду `ActiveQt`. В Mac OS X подсистема Apple Help предоставляет аналогичные функциональные возможности для *Qt Assistant*.

На этом мы завершаем часть II. В главах части III рассматриваются более продвинутые и специализированные средства разработки Qt. Их применение при программировании на C++ и Qt вызывает не больше трудностей, чем программирование того, что мы видели в части II, однако некоторые концепции и идеи могут вызвать дополнительные сложности в новых для вас областях.

Часть III

Высокий уровень Qt-программирования



- *Работа с Unicode*
- *Создание переводимого интерфейса приложения*
- *Динамическое переключение языков*
- *Перевод приложений*

Глава 18. Интернационализация

Кроме латинского алфавита, используемого для английского и многих европейских языков, Qt 4 обеспечивает широкую поддержку остальным мировым системам записи:

- Qt применяет Unicode в программном интерфейсе и во внутренних операциях. В приложении можно обеспечить всем пользователям одинаковую поддержку независимо от того, какой язык применяется в пользовательском интерфейсе;
- текстовый процессор Qt может работать со всеми основными нелатинскими системами записи, в том числе с арабской, китайской, кириллицей, ивритом, японской, корейской, тайской и с языками Индии;
- процессор компоновки Qt обеспечивает компоновку справа налево для таких языков, как арабский и иврит;
- для определенных языков требуются специальные методы ввода текста. Такие виджеты редактирования, как `QLineEdit` и `QTextEdit`, хорошо работают в условиях применения любого метода ввода текста, существующего в системе пользователя.

Разрешить пользователям вводить текст на их родном языке часто оказывается недостаточно; необходимо также перевести весь пользовательский интерфейс. В Qt это делается просто: все видимые пользователем строки обработайте функцией `tr()` (как это мы делали в предыдущих главах) и воспользуйтесь утилитами Qt для подготовки файлов перевода на требуемый язык. Qt имеет утилиту с графическим пользовательским интерфейсом, которая называется *Qt Linguist* и предназначена для переводчиков. *Qt Linguist* дополняется двумя консольными программами, `lupdate` и `lrelease`, которые обычно используются разработчиками приложений.

В большинстве приложений файл перевода загружается при запуске приложения с учетом установленных пользователем параметров локализации. Однако в некоторых случаях пользователям необходимо переключаться с одного языка на другой во время выполнения приложения. Это, несомненно, можно делать

в Qt, хотя и потребует немного дополнительной работы. А благодаря системе компоновки Qt, различные компоненты интерфейса пользователя будут автоматически перенастраиваться, чтобы обеспечить достаточно места для переведенного текста, когда его размер превышает размер исходного текста.

Работа с Unicode

Unicode является стандартной кодировкой, которая поддерживает большинство мировых систем записи. В основе кодировки Unicode лежит идея использования для хранения символов 16 бит, а не 8, и поэтому она позволяет закодировать примерно 65 000 символов вместо только 256.¹ Unicode содержит коды ASCII и ISO 8859-1 (Latin-1) в качестве своего подмножества с прежним их представлением. Например, английская буква «А» имеет значение 0x41 в кодировках ASCII, Latin-1 и Unicode, а буква «Ñ» имеет значение 0xD1 в кодировках Latin-1 и Unicode.

Класс Qt QString хранит строковые значения в кодировке Unicode. Каждый символ QString имеет 16-битовый тип QChar, а не 8-битовый тип char. Ниже приводятся два способа установки первого символа строки на значение «А»:

```
str[0] = 'A';  
str[0] = QChar(0x41);
```

Если исходный файл имеет кодировку Latin-1, задавать символы Latin-1 очень легко:

```
str[0] = 'C';
```

Но если исходный файл имеет другую кодировку, хорошо срабатывает вариант с числовым кодом:

```
str[0] = QChar(0xD1);
```

Мы можем задать любой символ Unicode с помощью его числового кода. Например, ниже показано, как задается прописная буква «сигма» греческого алфавита («Σ») и символ валюты евро («€»):

```
str[0] = QChar(0x03A3);  
str[0] = QChar(0x20AC);
```

Все числовые коды, поддерживаемые кодировкой Unicode, можно найти в сети Интернет по адресу <http://www.unicode.org/standard/>. Если вам приходится редко использовать символы Unicode, не относящиеся к Latin-1, для поиска их кодов вполне достаточно воспользоваться указанным адресом; но Qt обеспечивает более удобный способ ввода строк символов в кодировке Unicode в программе Qt, как мы увидим позднее в данном разделе.

Текстовый процессор в Qt 4 поддерживает на всех платформах следующие системы записи: арабскую, китайскую, кириллическую, греческую, иврит, японскую, корейскую, лаосскую, латинскую, тайскую и вьетнамскую. Он также поддерживает все скрипты 4.1 в кодировке Unicode, которые не требуют специальной обработки. Кроме того, в системе X11 с Fontconfig и в последних версиях

¹ Последние версии стандарта Unicode позволяют назначать символам значения, превышающие 65 535. Эти символы можно представить с помощью последовательности из двух 16-битовых значений, называемых «суррогатными парами» (surrogate pairs).

системы Windows поддерживаются следующие языки: бенгальский, деванагари, гуйарати, гурмухи, каннада, кхмерский, малайский, сирийский, тамильский, телугу, тхаана (дивехи) и тибетский. Наконец, ория поддерживается в системе X11, а монгольский и синхала поддерживаются в Windows XP. Если в системе установлен соответствующий шрифт, Qt сможет воспроизвести текст на любом из этих языков. А при установке соответствующих программ ввода текста пользователи смогут вводить в своих приложениях Qt текст на этих языках.

Программирование с использованием QChar немного отличается от программирования с применением char. Для получения числового кода символа QChar вызовите для него функцию `unicode()`. Для получения кода ASCII переменной типа QChar (в виде char) вызовите функцию `toLatin1()`. Для символов, отсутствующих в кодировке Latin-1, функция `toLatin1()` возвращает «\0».

Если нам заранее известно, что все строковые данные в программе представлены в кодировке ASCII или Latin-1, мы можем использовать такие стандартные функции (определенные в файле `<cctype>`), как `isalpha()`, `isdigit()` и `isspace()`, для обработки возвращаемого функцией `toLatin1()` значения. Однако в общем случае лучше использовать функции-члены класса QChar для выполнения этих операций, поскольку они будут правильно работать для любых символов Unicode. К таким функциям класса QChar относятся `isPrint()`, `isPunct()`, `isSpace()`, `isMark()`, `isLetter()`, `isNumber()`, `isLetterOrNumber()`, `isDigit()`, `isSymbol()`, `isLower()` и `isUpper()`. Например, ниже показано, как осуществлять проверку символа на цифру или прописную букву:

```
if (ch.isDigit() || ch.isUpper())  
...
```

Этот фрагмент кода правильно работает для любых алфавитов, в которых различаются символы верхнего и нижнего регистра, в том числе для латинского, греческого и кириллицы.

Строку в кодировке Unicode мы можем использовать в любом месте программного интерфейса Qt, где допускается применение строки типа QString. Qt сам отвечает за правильное ее отображение и преобразование в соответствующие кодировки при взаимодействии с операционной системой.

Особенно внимательными надо быть при чтении и записи текстовых файлов. Текстовые файлы могут использовать различные кодировки и часто оказывается невозможным определить кодировку текстового файла по его содержанию. По умолчанию QTextStream использует локальную системную 8-битовую кодировку (которая доступна при помощи функции `QTextCodec::codecForLocale()`), как для чтения, так и для записи. Для стран Америки и Западной Европы обычно это кодировка Latin-1.

Если мы разработали свой собственный формат файлов и собираемся считать и записывать произвольные символы Unicode, мы можем сохранять данные в кодировке Unicode с помощью вызова

```
stream.setCodec("UTF-16");  
stream.setGenerateByteOrderMark(true);
```

до начала записи в поток QTextStream. Данные в этом случае будут сохраняться в формате UTF-16, который использует два байта для представления одного

символа и который будет иметь префикс из специального 16-битового значения (признак порядка байтов Unicode, 0xFFFE), указывающего на применение файлом кодировки Unicode и на прямой или обратный порядок байтов. Формат UTF-16 идентичен представлению в памяти строк QString, и поэтому чтение и запись представленных в кодировке Unicode строк в формате UTF-16 может выполняться очень быстро. С другой стороны, такой подход связан с перерасходом памяти при сохранении данных, представленных целиком в кодировке ASCII, в формате UTF-16, поскольку в данном случае каждый символ займет два байта вместо одного.

Другие кодировки можно задавать путем вызова функции `setCodec()` с указанием соответствующего объекта преобразования `QTextCodec`. `QTextCodec` осуществляет преобразование между Unicode и заданной кодировкой. В Qt объекты `QTextCodec` используются в различных контекстах в Qt. Внутренними средствами они применяются для поддержки шрифтов, методов ввода, буфера обмена, технологии «drag-and-drop» и названий файлов. Но мы можем их использовать и непосредственно при написании приложений Qt.

При чтении текстового файла `QTextStream` автоматически обнаруживает кодировку UTF-16, если файл начинается с признака, определяющего порядок байтов. Такой режим работы можно отключить с помощью вызова `setAutoDetectUnicode(false)`. Если кодировка данных UTF-16, но нельзя рассчитывать на то, что данные начинаются с признака, определяющего порядок байтов, лучше всего перед чтением вызвать функцию `setCodec()` с аргументом «UTF-16».

Другой кодировкой, поддерживающей весь Unicode, является UTF-8. Его главное достоинство по сравнению с UTF-16, является то, что он является супермножеством по отношению к ASCII. Любой символ с кодом в диапазоне от 0x00 до 0x7F представляется в виде одного байта. Другие символы, включая символы Latin-1, код которых превышает значение 0x7F, представляются в виде последовательности из нескольких байтов. Текст, состоящий в основном из символов ASCII, в формате UTF-8 займет примерно в половину меньше памяти, чем в формате UTF-16. Для применения UTF-8 с `QTextStream` перед чтением и записью сделайте вызов `setEncoding(QTextStream::UnicodeUTF8)`.

Если мы всегда собираемся считывать и записывать файлы в кодировке Latin-1, вне зависимости от применяемой пользователем локальной кодировки мы можем установить кодировку «ISO 8859-1» для потока `QTextStream`. Например:

```
QTextStream in(&file);
in.setCodec("ISO 8859-1");
```

При применении некоторых форматов файлов их кодировка задается в заголовке файла. Заголовок обычно представляется в простом виде в кодировке ASCII, чтобы обеспечить его правильное чтение вне зависимости от используемой кодировки (в предположении, что она является супермножеством по отношению к ASCII). Интересным примером таких форматов являются файлы XML. Обычно файлы XML представлены в кодировке UTF-8 или UTF-16. Для правильного их чтения необходимо вызвать функцию `setCodec()` с «UTF-8». Если используется формат UTF-16, `QTextStream` автоматически обнаружит это и настроится на него. Заголовок `<?xml?>` файла XML иногда содержит аргумент `encoding`, например:

```
<?xml version="1.0" encoding="EUC-KR"?>
```

Поскольку `QTextStream` не позволяет менять кодировку после начала чтения, чтобы учесть явно заданную кодировку придется заново прочитать файл, задавая правильное преобразование (полученное функцией `QTextCodec::codecForName()`). В случае файла XML мы можем сами не делать преобразование кодировок, воспользовавшись классами Qt, предназначенными для XML и описанными в главе 16.

Другое применение объектов `QTextCodec` заключается в указании кодировки строк в исходном коде. Давайте рассмотрим, например, группу японских программистов, которые создают приложение, предназначенное главным образом для применения на домашнем японском рынке. Эти программисты, вероятно, будут писать свой исходный программный код в текстовом редакторе, использующем такие кодировки, как EUC-JP или Shift-JIS. Такой редактор позволяет им вводить японские иероглифы непосредственно, и, например, они смогут написать следующий код:

```
QPushButton *button = new QPushButton(tr("日諾"));
```

По умолчанию Qt считает, что аргументы функции `tr()` задаются в кодировке Latin-1. Для изменения этого необходимо вызвать статическую функцию `QTextCodec::setCodecForTr()`. Например:

```
QTextCodec *japaneseCodec = QTextCodec::codecForName("EUC-JP");
QTextCodec::setCodecForTr(japaneseCodec);
```

Это должно быть сделано до первого вызова `tr()`. Обычно мы делаем это в функции `main()` непосредственно после создания объекта `QCoreApplication` или `QApplication`.

Другие используемые в программе строки будут по-прежнему интерпретироваться как строки, представленные в кодировке Latin-1. Если программисты хотят вводить японские иероглифы и здесь, они могут явно преобразовывать их в Unicode, используя объект `QTextCodec`:

```
QString text = japaneseCodec->toUnicode("海鮮料理");
```

Можно поступить по-другому и указать Qt на необходимость применения особого преобразования между типами `const char *` и `QString` путем вызова функции `QTextCodec::setCodecForCStrings()`:

```
QTextCodec::setCodecForCStrings(QTextCodec::codecForName("EUC-JP"));
```

Описанные выше методы можно применять к любому языку, алфавит которого выходит за рамки кодировки Latin-1, включая языки китайский, греческий, корейский и русский.

Ниже приводится список кодировок, поддерживаемых Qt 4:

• Apple Roman	• ISO 8859-5	• Iscii-Mlm	• UTF-8
• Big5	• ISO 8859-6	• Iscii-Ori	• UTF-16
• Big5-HKSCS	• ISO 8859-7	• Iscii-Pnj	• UTF-16BE
• EUC-JP	• ISO 8859-8	• Iscii-Tlg	• UTF-16LE
• EUC-KR	• ISO 8859-9	• Iscii-Tml	• Windows-1250
• GB18030-0	• ISO 8859-10	• JIS X 0201	• Windows-1251
• IBM 850	• ISO 8859-13	• JIS X 0208	• Windows-1252
• IBM 866	• ISO 8859-14	• KOI8-R	• Windows-1253
• IBM 874	• ISO 8859-15	• KOI8-U	• Windows-1254

• ISO 2022-JP	• ISO 8859-16	• MuleLao-1	• Windows-1255
• ISO 8859-1	• Iscii-Bng	• ROMAN8	• Windows-1256
• ISO 8859-2	• Iscii-Dev	• Shift-JIS	• Windows-1257
• ISO 8859-3	• Iscii-Gjr	• TIS-620	• Windows-1258
• ISO 8859-4	• Iscii-Knd	• TSCII	• WINSAMI2

Для всех этих кодировок функция `QTextCodec::codecForName()` всегда будет возвращать достоверный указатель. Другие кодировки можно обеспечить путем создания подкласса `QTextCodec`.

Создание переводимого интерфейса приложения

Если мы хотим иметь многоязыковую версию нашего приложения, мы должны сделать две вещи:

- убедиться, что все строки, которые видит пользователь, проходят через функцию `tr()`;
- загрузить файл перевода (`.qm`) при запуске приложения.

Ничего подобного не надо делать, если приложения никогда не будут переводиться на другой язык. Однако применение функции `tr()` почти не требует дополнительных усилий и оставляет дверь открытой для их перевода когда-нибудь в будущем.

Функция `tr()` является статической функцией, определенной в классе `QObject` и переопределяемой в каждом подклассе, в котором встречается макрос `Q_OBJECT`. При ее использовании в рамках подкласса `QObject` мы можем вызывать `tr()` без ограничений. Вызов `tr()` возвращает перевод строки, если он имеется, и первоначальный текст в противном случае. Внутри класса, не являющегося потомком `QObject`, мы можем либо написать функцию `QObject::tr()` с префиксом класса, либо использовать макрос `Q_DECLARE_TR_FUNCTIONS()` для добавления функций `tr()` в класс, как мы это сделали в главе 8.

Для подготовки файлов переводов мы должны запустить утилиту `Qt lupdate`. Эта утилита собирает все строковые константы, которые встречаются в вызовах `tr()` и формирует файлы переводов, содержащие все эти подготовленные к переводу строки. Эти файлы могут затем быть переданы переводчику для добавления к ним перевода строк. Эта процедура рассматривается позже в данной главе в разделе «Перевод приложений».

В общем виде вызов `tr()` имеет следующий синтаксис:

Контекст::tr(исходныйТекст, комментарий)

Здесь *Контекст* — имя подкласса `QObject`, в котором используется макрос `Q_OBJECT`. Нам не требуется его указывать, если мы вызываем `tr()` в функции-члене рассматриваемого класса. Аргумент *исходныйТекст* — текстовая константа, которую нужно будет переводить. Аргумент *комментарий* является необязательным, и он может использоваться для предоставления переводчику дополнительной информации.

Ниже приводятся несколько примеров:

```
RockyWidget::RockyWidget(QWidget *parent)
    : QWidget(parent)
{
```

```
QString str1 = tr("Letter");
QString str2 = RockyWidget::tr("Letter");
QString str3 = SnazzyDialog::tr("Letter");
QString str4 = SnazzyDialog::tr("Letter", "US paper size");
}
```

Первые два вызова `tr()` выполняются в контексте объекта `RockyWidget` (скалистый виджет), а вторые два – в контексте объекта `SnazzyDialog` (притягательное диалоговое окно). В качестве исходного текста во всех четырех случаях используется слово «Letter» (буква). Последний вызов имеет также комментарий, помогающий переводчику точнее понять смысл исходного текста.

Строки в различных контекстах (классах) переводятся независимо друг от друга. Переводчики, как правило, одновременно работают только с одним контекстом, причем часто при этом работает приложение, и на экране отображается виджет или диалоговое окно, которые необходимо перевести.

Когда мы вызываем `tr()` из глобальной функции, мы должны явно указать контекст. Любой подкласс `QObject` может использоваться в приложении в качестве контекста. Если такого подкласса нет, мы всегда можем использовать сам класс `QObject`. Например:

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    ...
    QPushButton button(QObject::tr("Hello Qt!"));
    button.show();
    return app.exec();
}
```

До сих пор во всех примерах контекст задавался именем класса. Это удобно, поскольку мы почти всегда можем опустить его, но на самом деле это не так. Наиболее общий способ перевода строки в Qt заключается в использовании функции `QCoreApplication::translate()`, которая принимает три аргумента: контекст, исходный текст и необязательный комментарий. Например, ниже приводится другой способ перевода «Hello Qt!»:

```
QCoreApplication::translate("Global Stuff", "Hello Qt!")
```

На этот раз мы поместили текст в контекст «Global Stuff» (глобальное вещество).

Функции `tr()` и `translate()` играют двоякую роль: они являются маркерами, которые утилита `lupdate` использует для поиска видимых пользователем строк, и, одновременно, они являются функциями C++, которые переводят текст. Это имеет значение при написании программного кода. Например, следующий программный код не сработает:

```
// НЕПРАВИЛЬНО
const char *appName = "OpenDrawer 2D";
QString translated = tr(appName);
```

Проблема состоит в том, что утилита `lupdate` не сможет извлечь строковую константу «OpenDrawer 2D», поскольку она не входит в вызов функции `tr()`. Это

означает, что переводчик не будет иметь возможности перевести эту строку. Эта проблема часто возникает и при построении динамических строк:

```
// НЕПРАВИЛЬНО
statusBar()->showMessage(tr("Host " + hostName + " found"));
```

Здесь значение строки, которую мы передаем функции `tr()`, меняется в зависимости от значения `hostName`, и поэтому мы не можем ожидать, что перевод функцией `tr()` будет выполнен правильно.

Решение заключается в применении функции `QString::arg()`:

```
statusBar()->showMessage(tr("Host %1 found").arg(hostName));
```

Обратите внимание на то, как это работает: строковый литерал «Host %1 found» (хост %1 найден) передается функции `tr()`. Если загружен файл перевода на французский язык, `tr()` возвратит что-то подобное «Hôte %1trouvé». Параметр «%1» замещается на содержимое переменной `hostName`.

Хотя в целом не рекомендуется вызывать `tr()` для переменной, это может работать. Мы должны использовать макрос `QT_TR_NOOP()` для пометки тех строковых литералов, перевод которых должен быть выполнен до их присваивания переменной. Это лучше всего делать для статических массивов строк. Например:

```
void OrderForm::init()
{
    static const char * const flowers[] = {
        QT_TR_NOOP("Medium Stem Pink Roses"),
        QT_TR_NOOP("One Dozen Boxed Roses"),
        QT_TR_NOOP("Calypso Orchid"),
        QT_TR_NOOP("Dried Red Rose Bouquet"),
        QT_TR_NOOP("Mixed Peonies Bouquet"),
        0
    };

    for (int i = 0; flowers[i]; ++i)
        comboBox->addItem(tr(flowers[i]));
}
```

Макрос `QT_TR_NOOP()` просто возвращает свой аргумент. Но утилита `lupdate` обнаружит все строки, заданные в виде аргумента макроса `QT_TR_NOOP()`, и поэтому они смогут быть переведены. При использовании позже этой переменной мы вызываем, как обычно, `tr()` для выполнения перевода. Несмотря на передачу функции `tr()` переменной, перевод все-таки будет выполнен.

Существует также макрос `QT_TRANSLATE_NOOP()`, который работает подобно макросу `QT_TR_NOOP()`, но для него, кроме того, задается контекст. Этот макрос удобно использовать для инициализации переменных вне класса:

```
static const char * const flowers[] = {
    QT_TRANSLATE_NOOP("OrderForm", "Medium Stem Pink Roses"),
    QT_TRANSLATE_NOOP("OrderForm", "One Dozen Boxed Roses"),
    QT_TRANSLATE_NOOP("OrderForm", "Calypso Orchid"),
```

```

QT_TRANSLATE_NOOP("OrderForm", "Dried Red Rose Bouquet"),
QT_TRANSLATE_NOOP("OrderForm", "Mixed Peonies Bouquet"),
0
};

```

Здесь аргумент контекста должен совпадать с контекстом при будущем вызове функции `tr()` или `translate()`.

Когда мы начинаем использовать в приложении функцию `tr()`, легко можно забыть в каких-то случаях о необходимости задавать видимые пользователем строки через вызов функции `tr()` (особенно, если это делается впервые). Эти пропущенные строки фактически могут быть обнаружены переводчиком или, еще хуже, пользователями переведенного приложения, когда некоторые строки будут отображаться с применением первоначального языка. Чтобы не допустить этого, мы можем указать Qt на необходимость запрета неявных преобразований с типа `const char *` на тип `QString`. Это делается путем определения препроцессорного символа `QT_NO_CAST_FROM_ASCII` перед включением любого заголовочного файла Qt. Наиболее простой способ обеспечения установки этого символа состоит в добавлении следующей строки в файл `.pro`:

```
DEFINES += QT_NO_CAST_FROM_ASCII
```

Это заставит нас каждый строковый литерал использовать через вызов `tr()` или `QLatin1String()` в зависимости от того, надо ли его переводить или нет. Строки, которые не будут заданы именно таким образом, приведут к выводу сообщения об ошибке компилятора, и заставят нас восполнить пропущенную оболочку функций `tr()` или `QLatin1String()`.

После заключения всех видимых пользователем строк в вызовы функций `tr()` для обеспечения перевода нам остается только загрузить файл перевода. Обычно мы это делаем в функции приложения `main()`. Например, ниже показано, как можно попытаться загрузить файл перевода, который зависит от пользовательской локализации приложения:

```

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    ...
    QTranslator appTranslator;
    appTranslator.load("myapp_" + QLocale::system().name(), qmPath);
    app.installTranslator(&appTranslator);
    ...
    return app.exec();
}

```

Функция `QLocale::system()` возвращает объект `QLocale`, который содержит информацию о пользовательской локализации. Обычно имя локализации является частью имени файла `.qm`. Локализации можно задавать более или менее точно; например, `fr` задает европейский французский язык, `fr_CA` задает канадский французский язык, а `fr_CA.ISO8859-15` задает канадский французский язык с использованием кодировки ISO 8859-15 (которая поддерживает символы «€», «£», «œ» и «ÿ»).

Если локализацией является `fr_CA.ISO8859-15`, функция `QTranslator::load()` сначала попытается загрузить файл `myapp_fr_CA.ISO8859-15.qm`. Если этого файла нет, функция `load()` на следующем шаге попытается загрузить файл `myapp_fr_CA.qm`, затем `myapp_fr.qm` и, наконец, `myapp.qm`, и это будет последней попыткой. В обычных случаях нам необходимо предоставить только файл `myapp_fr.qm`, содержащий перевод на стандартный французский язык, но если нам нужен другой файл перевода для говорящих на французском в Канаде, мы можем также обеспечить файл `myapp_fr_CA.qm`, и он будет использован для локализации `fr_CA`.

Второй аргумент функции `QTranslator::load()` является каталогом, где функция `load()` будет искать файл перевода. В данном случае мы предполагаем, что файлы переводов размещаются в каталоге, указанном в переменной `qmPath`.

В самих библиотеках Qt содержится несколько строк, которые необходимо перевести. Компания Trolltech располагает переводы на французский, немецкий и упрощенный китайский языки в каталоге Qt translations. Имеются переводы также на другие языки, но они выполнены пользователями Qt и официально не поддерживаются. Необходимо также загрузить файл перевода библиотек Qt:

```
QTranslator qtTranslator;
qtTranslator.load("qt_" + QLocale::system().name(),
                 qmPath);
app.installTranslator(&qtTranslator);
```

Объект `QTranslator` может работать одновременно только с одним файлом перевода, и поэтому мы используем отдельный `QTranslator` для перевода приложения Qt. Возможность применения только одного файла для перевода не составляет проблемы, поскольку мы можем установить любое необходимое нам их количество. `QCoreApplication` будет рассматривать все такие файлы при поиске перевода.

Некоторые языки, такие как арабский и иврит, используют запись справа налево, а не слева направо. Для таких языков общая компоновка приложения должна быть изменена на зеркальную, что делается при помощи вызова функции `QApplication::setLayoutDirection(Qt::RightToLeft)`. Файлы перевода Qt содержат специальный маркер типа «LTR», указывающий Qt на направление записи используемого языка: слева направо или справа налево, и поэтому нам обычно не приходится самим вызывать функцию `setLayoutDirection()`.

Для наших пользователей может быть более удобно, если файлы перевода будут встраиваться в исполняемый модуль приложения, используя ресурсную систему Qt. Это не только снижает количество файлов в дистрибутиве приложения, но при этом снижается риск случайной потери или удаления файлов переводов.

Предположим, что файлы `.qm` располагаются в подкаталоге `translations` исходного дерева, тогда файл `myapp.qrc` будет содержать следующие строки:

```
<RCC>
<qresource>
    <file>translations/myapp_de.qm</file>
    <file>translations/myapp_fr.qm</file>
    <file>translations/myapp_zh.qm</file>
```



```
<file>translations/qt_de.qm</file>
<file>translations/qt_fr.qm</file>
<file>translations/qt_zh.qm</file>
</qresource>
</RCC>
```

Файл `.pro` будет иметь следующий элемент:

```
RESOURCES += myapp.qrc
```

Наконец, в функции `main()` мы должны указать `:/translations` в качестве пути к файлам переводов. Начальное двоеточие говорит о том, что это путь к ресурсу, а не к файлу, размещенному в файловой системе.

Теперь нами рассмотрено все, что необходимо для обеспечения перевода приложения на другие языки. Но язык и направление записи не единственное, что отличает различные страны и культуры. Интернационализация программы должна также учитывать местные форматы дат и времени, денежных единиц, чисел и упорядоченность букв. Qt содержит класс `QLocale`, обеспечивающий локализованные форматы чисел и даты/времени. Для получения другой информации, характерной для данной местности, мы можем использовать стандартные функции C++ `setlocale()` и `localeconv()`.

Поведение некоторых классов и функций Qt зависит от локализации:

- сравнение, которое осуществляет функция `QString::localeAwareCompare()`, зависит от локализации. Этой функцией удобно пользоваться для упорядочивания элементов, которые видит пользователь;
- функция `toString()` для объектов `QDate`, `QTime` и `QDateTime` возвращает строку в локализованном формате, если вызывается с аргументом `Qt::LocalDate`;
- по умолчанию виджеты `QDateEdit` и `QDateTimeEdit` представляют даты в локализованном формате.

Наконец, в переведенном приложении может потребоваться применение пиктограмм, отличных от используемых в оригинальной версии приложения. Например, стрелки влево и вправо на кнопках веб-браузера `Back` и `Forward` (назад и вперед) необходимо поменять местами для языка с записью справа налево. Мы можем это сделать следующим образом:

```
if (QApplication::isRightToLeft()) {
    backAction->setIcon(forwardIcon);
    forwardAction->setIcon(backIcon);
} else {
    backAction->setIcon(backIcon);
    forwardAction->setIcon(forwardIcon);
}
```

Обычно приходится переводить пиктограммы, содержащие буквы алфавита. Например, буква «I» на кнопке панели инструментов, отображающая опцию *Italic* (курсив) текстового процессора, должна быть заменена буквой «C» для испанского языка (*Cursivo*) и буквой «K» для языков датского, голландского, немецкого, норвежского и шведского (*Kursiv*). Ниже показано, как это можно просто сделать:

```

if (tr("Italic")[0] == 'C') {
    italicAction->setIcon(iconC);
} else if (tr("Italic")[0] == 'K') {
    italicAction->setIcon(iconK);
} else {
    italicAction->setIcon(iconI);
}

```

Можно поступить по-другому и использовать средства ресурсной системы, обеспечивающие поддержку нескольких локализаций. В файле `.qrc` мы можем определять локализацию для ресурса, используя атрибут `lang`. Например:

```

<qresource>
    <file>italic.png</file>
</qresource>
<qresource lang="es">
    <file alias="italic.png">cursivo.png</file>
</qresource>
<qresource lang="sv">
    <file alias="italic.png">kursiv.png</file>
</qresource>

```

Если пользовательской локализацией является `es` (Español), `:/italic.png` становится ссылкой на изображение `cursivo.png`. Если пользовательской локализацией является `sv` (Svenska), используется изображение `kursiv.png`. Для других локализаций используется `italic.png`.

Динамическое переключение языков

Для большинства приложений вполне удовлетворительный результат обеспечивает определение предпочитаемого пользователем языка в функции `main()` и загрузка там соответствующих файлов `.qm`. Но в некоторых ситуациях пользователям необходимо иметь возможность динамического переключения языков. Если приложение постоянно используется попеременно различными пользователями, может возникнуть необходимость в изменении языка без перезапуска приложения. Например, это часто требуется для приложений, применяемых операторами центров заказов, синхронными переводчиками и операторами компьютеризованных кассовых аппаратов.

Обеспечение в приложении возможности динамического переключения языков требует немного большего, чем просто загрузка одного файла перевода при запуске приложения, но это не трудно сделать.

Порядок действий должен быть следующим:

- предусмотрите средство, с помощью которого пользователь сможет переключаться с одного языка на другой;
- для каждого виджета или диалогового окна укажите все требующие перевода строки в отдельной функции (эта функция часто называется `retranslateUi()`), и вызывайте эту функцию всякий раз при изменении языка.

Давайте рассмотрим соответствующую часть исходного кода приложения «call center». Приложение содержит меню Language (язык) (показанное на рис. 18.1), чтобы пользователь имел возможность задавать язык во время работы приложения. По умолчанию применяется английский язык.



Рис. 18.1. Динамическое меню Language

Поскольку мы не знаем, какой язык захочет использовать пользователь после запуска приложения, мы теперь не будем загружать файлы перевода в функции `main()`. Вместо этого мы будем их загружать динамически, по мере необходимости, и поэтому обеспечивающий перевод программный код должен располагаться в классах главного и диалоговых окон.

Давайте рассмотрим подкласс `QMainWindow` этого приложения:

```
MainWindow::MainWindow()
{
    journalView = new JournalView;
    setCentralWidget(journalView);

    qApp->installTranslator(&appTranslator);
    qApp->installTranslator(&qtTranslator);

    createActions();
    createMenus();

    retranslateUi();
}
```

В конструкторе мы устанавливаем центральный виджет `JournalView` как подкласс `QTableWidget`. Затем мы устанавливаем два объекта `QTranslator` на объект `QApplication`. Объект `appTranslator` используется для хранения текущего перевода приложения, а объект `qtTranslator` хранит перевод библиотеки `Qt`.

В конце мы вызываем закрытые функции `createActions()` и `createMenus()` для создания системы меню и `retranslateUi()` (также закрытую функцию) для первой установки значений видимых пользователем строк.

```
void MainWindow::createActions()
{
    newAction = new QAction(this);
    newAction->setShortcut(QKeySequence::New);

    connect(newAction, SIGNAL(triggered()), this, SLOT(newFile()));
```

```

exitAction = new QAction(this);
connect(exitAction, SIGNAL(triggered()), this, SLOT(close()));
...
aboutQtAction = new QAction(this);
connect(aboutQtAction, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
}

```

Функция `createActions()` создает объекты `QAction` как обычно, но без установки текстов пунктов меню. Это будет сделано в функции `retranslateUi()`. Для тех команд, для которых есть стандартизованные клавиши быстрого вызова, мы можем здесь задать клавиши быстрого вызова с помощью соответствующего перечислимого типа, и дать Qt выполнить перевод. Для тех операций, для которых используются специальные клавиши быстрого вызова, например `Exit`, мы задаем быстрый вызов в функции `retranslateUi()` вместе с текстом.

```

void MainWindow::createMenus()
{
    fileMenu = new QMenu(this);
    fileMenu->addAction(newAction);
    fileMenu->addAction(openAction);
    fileMenu->addAction(saveAction);
    fileMenu->addAction(exitAction);
    editMenu = new QMenu(this);
    ...
    createLanguageMenu();

    helpMenu = new QMenu(this);
    helpMenu->addAction(aboutAction);
    helpMenu->addAction(aboutQtAction);

    menuBar()->addMenu(fileMenu);
    menuBar()->addMenu(editMenu);
    menuBar()->addMenu(reportsMenu);
    menuBar()->addMenu(languageMenu);
    menuBar()->addMenu(helpMenu);
}

```

Функция `createMenus()` создает пункты меню, но не устанавливает их текст. И снова, это будет сделано в функции `retranslateUi()`.

В середине функции мы вызываем `createLanguageMenu()` для заполнения меню `Language` списком поддерживаемых языков. Вскоре мы рассмотрим ее исходный код. Во-первых, давайте рассмотрим функцию `retranslateUi()`:

```

void MainWindow::retranslateUi()
{
    newAction->setText(tr("&New"));
    newAction->setStatusTip(tr("Create a new journal"));
    ...
    exitAction->setText(tr("E&xit"));
}

```

```

exitAction->setShortcut(tr("Ctrl+Q"));
...
aboutQtAction->setText(tr("About &Qt"));
aboutQtAction->setStatusTip(tr("Show the Qt library's About box"));

fileMenu->setTitle(tr("&File"));
editMenu->setTitle(tr("&Edit"));
reportsMenu->setTitle(tr("&Reports"));
languageMenu->setTitle(tr("&Language"));
helpMenu->setTitle(tr("&Help"));
setWindowTitle(tr("Call Center"));
}

```

Именно в функции `retranslateUi()` выполняются все вызовы `tr()` для класса `MainWindow`. Она вызывается в конце конструктора `MainWindow` и при каждом изменении пользователем языка приложения при помощи меню `Language`.

Мы устанавливаем для каждого пункта меню `QAction` его текст, и комментариев в строке состояния, а также клавиши быстрого вызова для тех команд, для которых нет стандартных клавиш быстрого вызова. Мы также задаем заголовок окну и каждому меню `QMenu`.

Рассмотренная ранее функция `createMenus()` вызывала функцию `createLanguageMenu()` для заполнения меню `Language` списком языков:

```

void MainWindow::createLanguageMenu()
{
    languageMenu = new QMenu(this);

    languageActionGroup = new QActionGroup(this);
    connect(languageActionGroup, SIGNAL(triggered(QAction *)),
            this, SLOT(switchLanguage(QAction *)));

    QDir qmDir = directoryOf("translations");
    QStringList fileNames =
        qmDir.entryList(QStringList("callcenter_*.qm"));
    for (int i = 0; i < fileNames.size(); ++i) {
        QString locale = fileNames[i];
        locale.remove(0, locale.indexOf('_') + 1);
        locale.chop(3);

        QTranslator translator;
        translator.load(fileNames[i], qmDir.absolutePath());
        QString language = translator.translate("MainWindow",
                                                "English");

        QAction *action = new QAction(tr("%1 %2")
                                     .arg(i + 1).arg(language), this);
        action->setCheckable(true);
        action->setData(locale);
    }
}

```

```

languageMenu->addAction(action);
languageActionGroup->addAction(action);

if (language == "English")
    action->setChecked(true);
}
}

```

Вместо жесткого кодирования поддерживаемых приложением языков мы создаем один пункт меню для каждого файла .qm, расположенного в каталоге приложения translations. Функция `directoryOf()` – та же самая функция, которую мы использовали в главе 17.

Для простоты мы предполагаем, что для английского языка также имеется файл .qm. Можно поступить по-другому и вызывать функцию `clear()` для объектов `QTranslator`, когда пользователь выбирает английский язык.

Определенная трудность состоит в предоставлении удобных названий языкам файлами .qm. Просто использование сокращений «en» для английского языка или «de» для немецкого языка, основанное на названии файла .qm, выглядит не лучшим образом и может запутать некоторых пользователей. Решение, которое используется функцией `createLanguageMenu()`, состоит в «переводе» строки «English» (английский язык) в контексте «MainWindow». Эта строка должна принимать значение «Deutsch» при переводе на немецкий язык, «Français» при переводе на французский язык и «日本語» при переводе на японский язык.

Мы создаем по одному помечаемому пункту меню `QAction` на каждый язык и храним локальное имя в его элементе данных. Мы добавляем их в объект `QActionGroup`, чтобы всегда мог быть помечен только один пункт меню `Language`. Когда пользователь выбирает какую-то команду из группы, объект `QActionGroup` генерирует сигнал `triggered(QAction *)`, который связан с `switchLanguage()`.

```

void MainWindow::switchLanguage(QAction *action)
{
    QString locale = action->data().toString();
    QString qmPath = directoryOf("translations").absolutePath();
    appTranslator.load("callcenter_" + locale, qmPath);
    qtTranslator.load("qt_" + locale, qmPath);
    retranslateUi();
}

```

Слот `switchLanguage()` вызывается, когда пользователь выбирает язык из меню `Language`. Мы загружаем соответствующие файлы перевода приложения и библиотеки `Qt` и затем вызываем функцию `retranslateUi()` для нового перевода всех строк главного окна.

В системе `Windows` в качестве альтернативы меню `Language` можно использовать обработку событий `LocaleChange` – события этого типа генерируются `Qt` при обнаружении изменения среды локализации. Событие этого типа существует на всех платформах, поддерживаемых `Qt`, но фактически они генерируются только в системе `Windows`, когда пользователь изменяет системные настройки локализации (при задании параметров региона и языка на Панели управления). Для

обработки событий `LocaleChange` мы можем переопределить функцию `QWidget::changeEvent()` следующим образом:

```
void MainWindow::changeEvent(QEvent *event)
{
    if (event->type() == QEvent::LocaleChange) {
        QString qmPath = directoryOf("translations").absolutePath();
        appTranslator.load("callcenter_"
                        + QLocale::system().name(), qmPath);
        qtTranslator.load("qt_" + QLocale::system().name(), qmPath);
        retranslateUi();
    }
    QMainWindow::changeEvent(event);
}
```

Если пользователь переключается на другую локализацию во время выполнения приложения, мы пытаемся загрузить файлы перевода, соответствующие новой локализации, и вызываем функцию `retranslateUi()` для обновления интерфейса пользователя. Во всех случаях мы передаем событие функции базового класса `changeEvent()`, поскольку один из наших классов тоже может быть заинтересован в обработке событий `LocaleChange` или других событий.

На этом мы закончили наш обзор программного кода класса `MainWindow`. Теперь мы рассмотрим программный код одного из классов-виджетов приложения, а именно, класса `JournalView`, чтобы определить, какие изменения потребуются для обеспечения поддержки им динамического перевода.

```
JournalView::JournalView(QWidget *parent)
    : QWidget(parent)
{
    ...
    retranslateUi();
}
```

Класс `JournalView` является подклассом `QWidget`. В конце конструктора мы вызываем закрытую функцию `retranslateUi()` для перевода строк виджета. Это напоминает то, что мы делали для класса `MainWindow`.

```
void JournalView::changeEvent(QEvent *event)
{
    if (event->type() == QEvent::LanguageChange)
        retranslateUi();
    QWidget::changeEvent(event);
}
```

Мы также переопределяем функцию `changeEvent()` для вызова `retranslateUi()` при генерации событий `LanguageChange`. Qt генерирует событие `LanguageChange` при изменении содержимого объекта `QTranslator`, который в данный момент используется в `QCoreApplication`. В нашем приложении это происходит, когда мы вызываем `load()` для `appTranslator` или `qtTranslator` либо из функции `MainWindow::switchToLanguage()`, либо из функции `MainWindow::changeEvent()`.

События `LanguageChange` не следует путать с событиями `LocaleChange`. Событие `LocaleChange` генерируется системой и говорит приложению: «Возможно, следует загрузить новый файл перевода». В отличие от него событие `LanguageChange` генерируется `Qt` и говорит виджетам приложения: «Возможно, следует заново выполнить перевод всех ваших строк».

При реализации нами класса `MainWindow` не нужно было реагировать на событие `LanguageChange`. Вместо этого мы просто всякий раз вызывали функцию `retranslateUi()` при вызове `load()` для `QTranslator`.

```
void JournalView::retranslateUi()
{
    QStringList labels;
    labels << tr("Time") << tr("Priority") << tr("Phone Number")
        << tr("Subject");
    setHorizontalHeaderLabels(labels);
}
```

Функция `retranslateUi()` заново создает заголовки, используя новый переведенный текст, и этим мы завершаем рассмотрение программного кода, относящегося к переводу созданного вручную виджета. Для виджетов и диалоговых окон, которые разрабатываются при помощи *Qt Designer*, утилита `uic` автоматически генерирует функцию, подобную `retranslateUi()`, которая автоматически вызывается в ответ на события `LanguageChange`.

Перевод приложений

Перевод приложения `Qt`, которое содержит вызовы `tr()`, состоит из трех этапов.

1. Выполнение утилиты `lupdate` для извлечения из исходного кода приложения всех видимых пользователем строк.
2. Перевод приложения при помощи *Qt Linguist*.
3. Выполнение утилиты `lrelease` для получения двоичных файлов `.qm`, которые приложение может загружать при помощи объекта `QTranslator`.

Этапы 1 и 3 выполняются разработчиками приложения. Этап 2 выполняется переводчиками. Эта последовательность действий может выполняться любое количество раз в ходе разработки приложения и на протяжении всего его жизненного цикла.

В качестве примера мы продемонстрируем перевод приложения *Электронная таблица* из главы 3. Приложение уже содержит вызовы `tr()` для всех видимых пользователем строк.

Во-первых, мы должны немного модифицировать файл приложения `.pro`, указав языки, которые мы собираемся поддерживать. Например, если бы мы хотели поддерживать кроме английского также немецкий и французский, мы бы добавили следующий элемент `TRANSLATIONS` в файл `spreadsheet.pro`:

```
TRANSLATIONS = spreadsheet_de.ts \
               spreadsheet_fr.ts
```


Здесь мы указали два файла переводов: один для немецкого языка и второй для французского языка.

Эти файлы будут созданы при первом выполнении утилиты `lupdate`, и затем они будут обновляться при каждом последующем выполнении `lupdate`.

Эти файлы обычно имеют расширение `.ts`. Они имеют простой формат XML и не столь компактны, как двоичные файлы `.qm`, которые «понимают» объекты типа `QTranslator`. В задачу утилиты `lrelease` входит преобразование предназначенных для людей файлов `.ts` в эффективное машинное представление в виде файлов `.qm`. Между прочим, сокращение `.ts` означает файл «translation source» (файл с исходным текстом перевода), а `.qm` – файл «Qt message» (файл сообщений Qt).

Предположим, что мы находимся в каталоге, который содержит исходный код приложения *Электронная таблица*, и тогда мы можем выполнить утилиту `lupdate` для `spreadsheet.pro`, задавая в командной строке следующую команду:

```
lupdate -verbose spreadsheet.pro
```

Опция `-verbose` указывает утилите `lupdate` на необходимость более интенсивной обратной связи, чем та, которая обеспечивается при нормальном режиме работы. Ниже приводятся сообщения, получение которых следует ожидать в результате работы утилиты:

```
Updating 'spreadsheet_de.ts'...
Found 98 source texts (98 new and 0 already existing)
Updating 'spreadsheet_fr.ts'...
Found 98 source texts (98 new and 0 already existing)
```

Все строки, которые задаются в вызовах функции `tr()` в исходном коде приложения, хранятся в файлах `.ts` (в том числе и псевдоперевод). Сюда также включаются строки из файлов приложения `.ui`.

По умолчанию утилита `lupdate` предполагает, что передаваемые функции `tr()` строки используют кодировку `Latin-1`. Если это не так, мы должны добавить элемент `CODECFORTR` в файл `.pro`. Например:

```
CODECFORTR = EUC-JP
```

Это должно быть сделано в дополнение к вызову `QTextCodec::setCodecForTr()` из функции приложения `main()`.

Затем в файлы `spreadsheet_de.ts` и `spreadsheet_fr.ts` необходимо добавить перевод, выполненный при помощи *Qt Linguist* (на рис. 18.2 показано приложение *Qt Linguist* в действии).

Для запуска *Qt Linguist* выберите пункт `Qt by Trolltech v4.x.y|Linguist` в меню `Start` в системе `Windows`, введите `linguist` в командной строке в системе `Unix` или дважды щелкните по `Linguist` в системе `Mac OS X Finder`. Для добавления перевода в файл `.ts` выберите пункт меню `File|Open` и укажите файл для перевода.

Слевой стороны главного окна утилиты *Qt Linguist* отображается древовидное представление. Элементы верхнего уровня представляют собой контексты переводимого на другие языки приложения. Для приложения *Электронная таблица* этими контекстами являются «FindDialog», «GoToCellDialog», «MainWindow»,

«SortDialog» и «Spreadsheet». Каждый контекст содержит ноль или более дочерних элементов. Каждый дочерний элемент занимает три столбца, в первом из которых отображается флажок Done (готово), во втором – текст оригинала, а в третьем – перевод на любой язык. В верхней правой части отображается текущий текст оригинала и его перевод. Именно здесь добавляются и редактируются переводы. Справа внизу отображаются подсказки по переводу, которые автоматически генерируются *Qt Linguist*.

После создания нами файла переводов .ts необходимо его преобразовать в двоичный файл .qm, чтобы он был понятен для *QTranslator*. Для этого в *Qt Linguist* выберите пункт меню File|Release. Обычно мы начинаем с перевода только нескольких строк и затем выполняем приложение с применением файла .qm, чтобы убедиться, что все работает правильно.

Если мы хотим заново сгенерировать файлы .qm для всех файлов .ts, мы можем запустить утилиту `lrelease` из командной строки следующим образом:

```
lrelease -verbose spreadsheet.pro
```

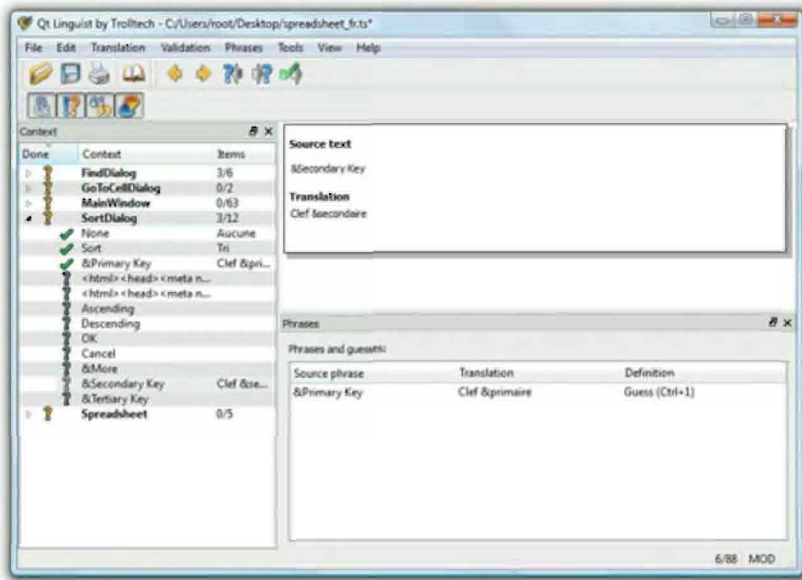


Рис. 18.2. Qt Linguist в действии

Если мы выполняли перевод 19 строк на французский язык и отметили флажком Done 17 из них, утилита `lrelease` выдаст следующий результат:

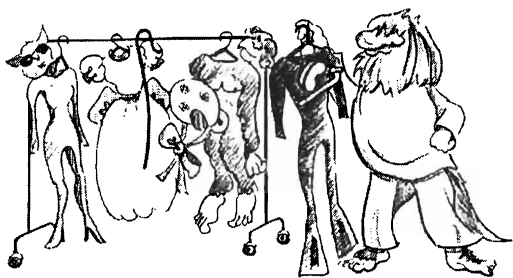
```
Updating 'spreadsheet_de.qm'...
  Generated 0 translations (0 finished and 0 unfinished)
  Ignored 98 untranslated source texts
Updating 'spreadsheet_fr.qm'...
  Generated 19 translations (17 finished and 2 unfinished)
  Ignored 79 untranslated source texts
```

Непереведенные строки при выполнении приложения выводятся на языке оригинала. Флажок Done игнорируется утилитой `lrelease`; он может использоваться переводчиками для идентификации законченных переводов и тех, перевод которых необходимо выполнить заново.

Когда мы модифицируем исходный код приложения, файлы перевода могут устареть. Решение этой проблемы заключается в повторном выполнении утилиты `lupdate`, обеспечении перевода новых строк и повторной генерации файлов `.qm`. Одни группы разработчиков могут посчитать удобным частое выполнение утилиты `lupdate`, а другие могут захотеть это делать только для почти готового программного продукта.

Утилиты `lupdate` и *Qt Linguist* достаточно «умны». Переводы, которые с какого-то момента не стали использоваться, сохраняются в файлах `.ts` на случай, если они потребуются когда-нибудь в будущем. При обновлении файлов `.ts` утилита `lupdate` использует «интеллектуальный» алгоритм слияния, позволяющий переводчикам сэкономить много времени при работе с текстом, который совпадает или подобен в различных контекстах.

Более подробную информацию относительно *Qt Linguist*, `lupdate` и `lrelease` можно найти в руководстве по *Qt Linguist* в сети Интернет по адресу <http://doc.trolltech.com/4.3/linguist-manual.html>. Это руководство содержит полное описание интерфейса пользователя для *Qt Linguist* и учебное пособие для поэтапного обучения программистов.



- *Использование таблиц стилей Qt*
- *Создание подклассов QStyle*

Глава 19. Настройка диалога с пользователем

В некоторых ситуациях нам может понадобиться изменить внешний вид и функционирование виджетов Qt. Возможно, мы захотим внести лишь небольшие изменения, чтобы слегка подкорректировать общее впечатление, или же мы можем захотеть реализовать совершенно новый стиль, чтобы придать нашему приложению или приложениям уникальный и неповторимый внешний облик. В любом случае есть три основных подхода к переопределению внешнего вида встроенных виджетов Qt.

- Можно создать подклассы отдельных классов виджетов и переопределить их обработчики событий рисования и функций мыши. Это обеспечит нам полный контроль, но требует массу работы. Этот метод также предполагает, что мы должны пройтись по всему нашему коду и по формам *Qt Designer* и везде, где встречаются классы виджетов Qt, заменить их нашими подклассами.
- Мы можем создать подкласс класса *QStyle* или готового стиля, например, *QWindowsStyle*. Этот подход весьма мощный. Он применяется в самом Qt для того, чтобы он сохранял привычный облик на разных платформах, которые он поддерживает.
- Начиная с версии Qt 4.2, мы можем использовать таблицы стилей Qt, концепция которых возникла под влиянием CSS (Cascading style sheets, каскадные таблицы стилей) в HTML. Поскольку таблицы стилей представляют собой простые текстовые файлы, интерпретируемые во время выполнения, для их использования не требуется знание программирования.

Мы уже рассматривали методику, применяемую при первом подходе, в главах 5 и 7, хотя основное внимание в этих главах уделялось созданию пользовательских виджетов. В этой главе мы рассмотрим два последних подхода. Мы предложим два пользовательских стиля: стиль *Candy*, описанный в виде таблицы стилей, и стиль *Bronze*, реализованный в виде подкласса класса *QStyle* (рис. 19.1). Чтобы примеры имели приемлемый размер, для обоих стилей мы используем тщательно отобранную часть виджетов Qt.

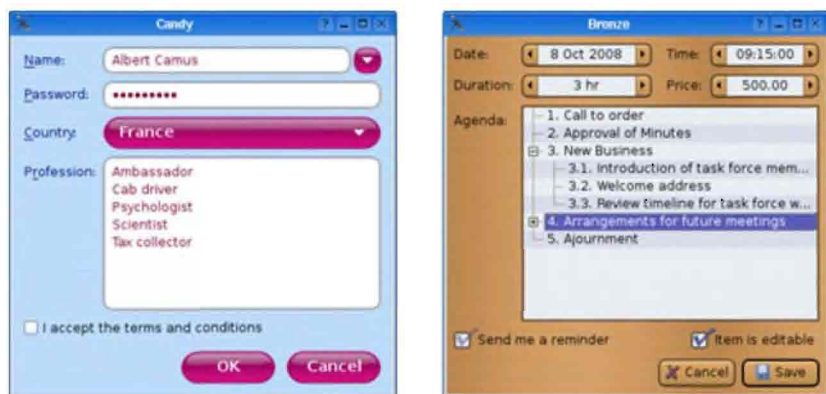


Рис. 19.1. Пользовательские стили, представляемые в этой главе

Использование таблиц стилей Qt

Таблицы стилей возникли, в основном, под влиянием CSS, но были адаптированы для работы с виджетами. Таблицы стилей состоят из стилевых правил, которые влияют на отображение виджета. Эти правила описываются простым текстом. Поскольку таблицы стилей обрабатываются во время выполнения, мы легко можем экспериментировать с разными формами дизайна, указывая используемый стиль для приложения Qt в виде опции командной строки `-stylesheet file.qss`, используя редактор таблиц стилей *Qt Designer* или при помощи встраивания объекта `QTextEdit` в приложение в ходе его разработки.

Таблицы стилей применяются поверх активного в настоящий момент класса `QStyle` (например, `QWindowsVistaStyle` или `QPlastiqueStyle`)¹. Поскольку для создания таблиц стилей не требуется создавать подклассы, они идеально подходят для небольшой настройки имеющихся виджетов. Например, предположим, что мы хотим использовать желтый фоновый цвет для всех объектов `QLineEdit` в приложении. Этого можно добиться при помощи следующей таблицы стилей:

```
QLineEdit {
    background-color: yellow;
}
```

В терминологии CSS объект `QLineEdit` — это селектор, `background-color` — атрибут, а `yellow` — значение.

Для такого рода настройки применение таблиц стилей дает более надежные результаты, чем использование палитры виджета. Это объясняется тем, что записи `QPalette` (`Base`, `Button`, `Highlight` и т. п.) по-разному используются в разных стилях. Например, в `QWindowsStyle` запись `Base` палитры используется для заливки фона поля с раскрывающимся списком, доступного только для чтения, тогда как в `QPlastiqueStyle` для этой цели используется запись `Button`. Более того, в некоторых стилях для отображения некоторых элементов используются жестко запрограммированные изображения, и совершенно не используется палитра.

¹ Таблицы стилей `QMacStyle` в Qt 4.3 не поддерживаются. Ожидается, что поддержка появится в будущей версии.

Напротив, таблицы стилей гарантируют, что независимо от того, какой объект `QStyle` является активным, будут использоваться именно указанные цвета.

Функция `QApplication::setStyleSheet()` задает таблицу стилей для всего приложения:

```
qApp->setStyleSheet("QLineEdit { background-color: yellow; }");
```

С помощью функции `QWidget::setStyleSheet()` также можно задать таблицу стилей для виджета и его потомков. Например:

```
dialog->setStyleSheet("QLineEdit { background-color: yellow; }");
```

Пока что мы задавали только одно свойство для одного класса виджета. На практике стилевые правила часто объединяются. Например, следующее правило задает цвет фона и цвет переднего плана для шести классов виджетов и их подклассов:

```
QCheckBox, QComboBox, QLineEdit, QListView, QRadioButton, QSpinBox {
    color: #050505;
    background-color: yellow;
}
```

Цвета можно указывать по именам, строкой в стиле HTML (в формате `#RRGGBB`) или в виде `RGB`- или `RGBA`-значения:

```
QLineEdit {
    color: rgb(0, 88, 152);
    background-color: rgba(97%, 80%, 9%, 50%);
}
```

При использовании имен цветов мы можем указывать любое имя, распознаваемое функцией `QColor::setNamedColor()`. В формате `RGB` мы должны указать красную, зеленую и синюю компоненту в диапазоне значений от 0 до 255 или от 0% до 100%. Формат `RGBA` дополнительно позволяет указать в качестве четвертого компонента значение альфа, соответствующее непрозрачности цвета. Вместо сплошного цвета мы также можем указать запись шаблона или градиент:

```
QLineEdit {
    color: palette(Base);
    background-color: qlineargradient(x1: 0, y1: 0, x2: 1, y2: 1,
                                     stop: 0 white, stop: 0.4 gray,
                                     stop: 1 green);
}
```

В главе 8 описаны три типа градиентов — `qlineargradient()`, `qradialgradient()` и `qconicalgradient()`. Синтаксис разъясняется в справочной документации по таблицам стилей.

С помощью свойства `background-image` мы можем задать фоновое изображение:

```
QLineEdit {
    color: rgb(0, 88, 152);
    background-image: url(:/images/yellow-bg.png);
}
```

Как правило, фоновое изображение начинается от верхнего левого угла виджета (исключая поля, указанные с помощью свойства `margin`), и повторяется по горизонтали и по вертикали, заполняя собой весь виджет. Его можно конфигурировать с помощью атрибутов `background-position` и `background-repeat`. Например:

```
QLineEdit {
    background-image: url(:/images/yellow-bg.png);
    background-position: top right;
    background-repeat: repeat-y;
}
```

Если мы зададим и фоновый цвет, и фоновое изображение, то фоновый цвет будет просвечивать через полупрозрачные области изображения.

До сих пор все селекторы, которые мы применяли, представляли собой имена классов. Но существует несколько других селекторов, которые мы можем использовать. Они перечислены на рис. 19.2. Например, если мы хотим использовать специфический цвет переднего плана для кнопок `OK` и `Cancel`, мы можем написать такую таблицу:

```
QPushButton[text="OK"] {
    color: green;
}
QPushButton[text="Cancel"] {
    color: red;
}
```

Данный синтаксис селектора работает для любого свойства Qt, хотя нужно помнить, что таблицы стилей не видят изменений свойств, происходящих позади. Селекторы также можно по-разному комбинировать. Например, чтобы выбрать все объекты `QPushButton` с именем `«okButton»`, у которых свойства `x` и `y` равны 0, и которые являются прямыми потомками `QFrame` с именем `«frame»`, мы можем написать такой код:

```
QFrame#frame > QPushButton[x="0"][y="0"]#okButton {
...
}
```

Селектор	Пример	Виджеты, соответствующие селектору
Универсальный	*	Любой виджет
Тип	<code>QDialog</code>	Экземпляры данного класса, включая подклассы
Класс	<code>QDialog</code>	Экземпляры данного класса, но не подклассов
Идентификатор	<code>QDialog#ageDialog</code>	Виджеты с указанным именем объекта
Свойство Qt	<code>QDialog[y="0"]</code>	Виджеты, в которых определенные свойства имеют определенные значения
Дочерние	<code>QFrame > QDialog</code>	Виджеты, являющиеся прямыми потомками указанных виджетов
Потомки	<code>QFrame QDialog</code>	Виджеты, являющиеся потомками указанных виджетов

Рис. 19.2. Селекторы таблиц стилей

В приложениях, использующих большие формы с большим количеством полей редактирования и полей с раскрывающимися списками, например, приложений для ведения всевозможной документации, весьма распространенным является использование желтого фона для обязательных полей. Давайте предположим, что нам нужно использовать данное соглашение в нашем приложении. Для начала мы используем такую таблицу стилей:

```
*[mandatoryField="true"] {
    background-color: yellow;
}
```

Хотя свойство `mandatoryField` нигде в Qt не определено, мы легко можем создать такое свойство, вызвав функцию `QObject::setProperty()`. Начиная с версии Qt 4.2, задание значения для несуществующего свойства приводит к динамическому созданию этого свойства. Например:

```
nameLineEdit->setProperty("mandatoryField", true);
genderComboBox->setProperty("mandatoryField", true);
ageSpinBox->setProperty("mandatoryField", true);
```

Таблицы стилей используются не только для управления цветами. Они также позволяют нам корректировать размер и местоположение элементов виджетов. Например, следующие правила можно использовать для увеличения размера флажков и переключателей до 20×20 пикселей и для обеспечения наличия интервала в 8 пикселей между таким индикатором и связанным с ним текстом:

```
QCheckBox::indicator, QRadioButton::indicator {
    width: 20px;
    height: 20px;
}
QCheckBox, QRadioButton {
    spacing: 8px;
}
```

Обратите внимание на синтаксис селектора в первом правиле. Если бы мы указали `QCheckBox`, а не `QCheckBox::indicator`, мы задали бы размер всего виджета, а не только индикатора. Первое правило иллюстрируется на рис. 19.3.

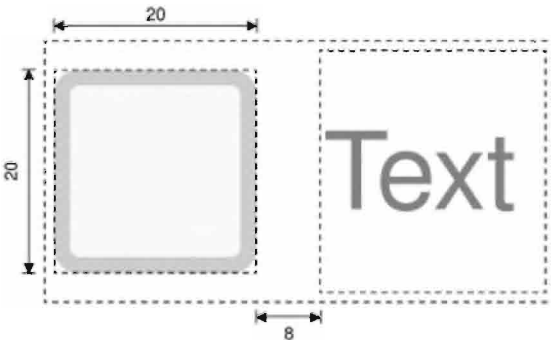


Рис. 19.3. Задание размера индикатора `QCheckBox`

Для дочерних элементов управления, таких как `::indicator`, стили применяются, в основном, так же, как для виджетов. На рис. 19.4 показаны некоторые дочерние элементы управления, поддерживаемые Qt.

Дочерний элемент управления	Описание
<code>::indicator</code>	Индикатор-флажок, переключатель, выбираемый пункт меню или выбираемый групповой блок
<code>:menu-indicator</code>	Индикатор кнопочного меню
<code>::item</code>	Элемент меню, линейки меню или строки состояния
<code>::up-button</code>	Кнопка «вверх» полосы прокрутки или счетчика
<code>::down-button</code>	Кнопка «вниз» полосы прокрутки или счетчика
<code>::up-arrow</code>	Стрелка вверх счетчика, полосы прокрутки или заголовка
<code>::down-arrow</code>	Стрелка вниз счетчика, полосы прокрутки или заголовка
<code>::drop-down</code>	Кнопка, раскрывающая раскрывающийся список
<code>::title</code>	Заголовок группового блока

Рис. 19.4. Наиболее распространенные настраиваемые дочерние элементы управления

Наряду с дочерними элементами управления, таблица стилей может относиться к конкретным состояниям виджета. Например, нам может быть нужно отобразить текст переключателя белым цветом при наведении на него указателя мыши, для чего нужно указать состояние `:hover`:

```
QCheckBox:hover {
    color: white;
}
```

Состояния обозначаются одним двоеточием, а дочерние элементы управления – двумя. Мы можем указывать несколько состояний, разделяемых двоеточиями. В таких случаях правило применяется, только если виджет находится *во всех* этих состояниях сразу. Например, следующее правило применяется, когда мышь наводится на установленный флажок:

```
QCheckBox:checked:hover {
    color: white;
}
```

Если мы хотим, чтобы правило применялось, когда виджет находится в *любом* из указанных состояний, мы можем использовать несколько селекторов, разделяя их запятыми:

```
QCheckBox:hover, QCheckBox:checked {
    color: white;
}
```

Логическое отрицание обозначается восклицательным знаком:

```
QCheckBox:!checked {
    color: blue;
}
```

Состояния можно сочетать с дочерними элементами:

```
QComboBox::drop-down: hover {
    image: url(/images/downarrow_bright.png);
}
```

На рис. 19.5 перечислены существующие состояния таблиц стилей.

Состояние	Описание
:disabled	Виджет недоступен
:enabled	Виджет доступен
:focus	Виджет получил фокус для ввода
:hover	На виджет наведен курсор мыши
:pressed	На виджете нажата кнопка мыши
:checked	Кнопка отмечена
:unchecked	С кнопки снята отметка
:indeterminate	Кнопка отмечена частично
:open	Виджет находится в открытом или развернутом состоянии
:closed	Виджет находится в закрытом или свернутом состоянии
:on	Виджет «включен»
:off	Виджет «выключен»

Рис. 19.5. Некоторые состояния виджетов, доступные через таблицы стилей

Таблицы стилей также могут использоваться в сочетании с другими методами для выполнения более сложных настроек. Например, предположим, что нам нужно поместить миниатюрную кнопку «erase» (стереть) внутри рамки поля QLineEdit и справа от текста этого поля. Для этого нужно создать класс EraseButton и поместить его поверх поля QLineEdit (например, с помощью компоновки), но также оставить немного места для кнопки, чтобы вводимый в поле текст не мог наехать на кнопку «erase». Решать такую задачу путем создания подкласса класса QStyle было бы неудобно, поскольку нам нужно было бы создавать подклассы для каждого стиля Qt, который может использоваться в приложении (QWindowsVistaStyle, QPlastiqueStyle и т. п.). При использовании таблиц стилей решением будет следующее правило:

```
QLineEdit {
    padding: 0px 15px 0px 0px;
}
```

Свойство padding позволяет нам указать верхний, правый, нижний и левый интервалы для виджета. Интервал вставляется между текстом поля QLineEdit и его рамкой. Для удобства в CSS также определяются свойства padding-top, padding-right, padding-bottom и padding-left, для которых указывается только одно значение интервала. Например:

```
QLineEdit {
    padding-right: 15px;
}
```

Как и большинство виджетов Qt, которые настраиваются с использованием таблиц стилей, поле QLineEdit поддерживает блоковую модель, изображенную на рис. 19.6. В этой модели указываются четыре прямоугольника, влияющие на компоновку и отображение виджета с примененными стилями.

1. *Прямоугольник содержимого* (content rectangle) – является самым внутренним прямоугольником. Именно здесь рисуется, собственно, содержимое виджета (например, текст или изображение).
2. *Прямоугольник интервалов* (padding rectangle) включает в себе прямоугольник содержимого. Также он учитывает все интервалы, указанные с использованием свойства padding.
3. *Прямоугольник рамки* (border rectangle) включает в себе прямоугольник интервалов. Он оставляет место для рамки.
4. *Прямоугольник полей* (margin rectangle) является самым наружным прямоугольником. Он включает в себе прямоугольник рамки и учитывает все указанные поля.

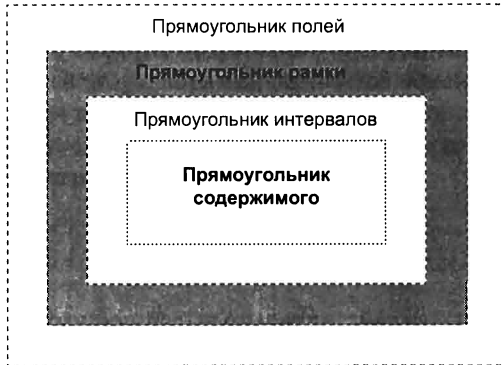


Рис. 19.6. Блоковая модель CSS

В простом виджете без интервалов, рамки и полей эти четыре прямоугольника в точности совпадают.

Сейчас мы представим таблицу стилей, реализующую пользовательский стиль с именем Candy. На рис. 19.7 показан отбор виджетов в стиле Candy. Стиль Candy определяет пользовательский облик и функции объектов QLineEdit, QListView, QPushButton и QComboBox с использованием блоковой модели, представленной на рис. 19.6. Мы рассмотрим таблицу стилей по частям, а всю таблицу стилей можно найти в файле qss/candy.qss в директории Candy примеров, прилагаемых к этой книге.

Эти виджеты используются в диалоговом окне, показанном на рис. 19.1. У самого диалогового окна есть фоновое изображение, которое задается следующим правилом:

```
QDialog {
    background-image: url(/images/background.png);
```

[R1]



Рис. 19.7. Виджеты в стиле Candy

В следующем правиле задаются атрибуты `color` и `font` объектов `QLabel`:

```
QLabel {  
    font: 9pt;  
    color: rgb(0, 0, 127);  
}
```

[R2]

В следующем правиле определяется облик объекта `QLineEdit` и объекта `QListView`:

```
QLineEdit,  
QListView {  
    color: rgb(127, 0, 63);  
    background-color: rgb(255, 255, 241);  
    selection-color: white;  
    selection-background-color: rgb(191, 31, 127);  
    border: 2px groove gray;  
    border-radius: 10px;  
    padding: 2px 4px;  
}
```

[R3]

Чтобы сделать объекты `QLineEdit` и `QListView` действительно выделяющимися, мы указали специальные цвета фона и переднего плана для обычного и выделенного текста. Кроме того, мы указали серую «утопленную» рамку шириной два пикселя при помощи атрибута `border`. Вместо свойства `border` мы могли бы указать свойства `border-width`, `border-style` и `bordercolor` по отдельности. Мы можем сделать закругленными углы рамки, указав свойство `borderradius`, и мы здесь используем радиус закругления 10 пикселей.



Рис. 19.8. Структура элемента `QLineEdit`

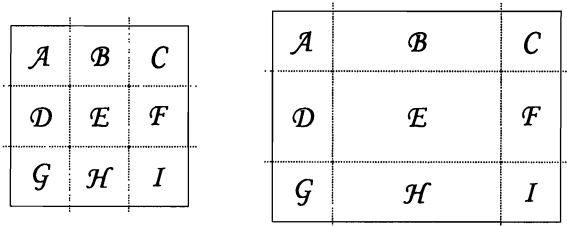
На рис. 19.8 показано схематичное представление влияния сделанных нами изменений на атрибуты виджета `border` и `padding`. Чтобы содержимое виджета не накладывалось на закругленные углы рамки, мы указываем внутренний интервал 2 пикселя по вертикали и 4 пикселя по горизонтали. Для объектов `QListView` вертикальные поля выглядят не совсем хорошо, поэтому мы заменяем их следующим образом:

```
QListView {  
    padding: 5px 4px;  
}  
[R4]
```

Если атрибут задается в нескольких правилах, имеющих одинаковый селектор, применяется последнее правило.

Для определения стиля объектов `QPushButton` мы будем использовать совершенно иной подход. Вместо того чтобы рисовать кнопку с помощью правил таблиц стилей, мы будем использовать в качестве фона заранее подготовленное изображение. Кроме того, чтобы сделать кнопку масштабируемой, фон кнопки определяется с помощью механизма, называющегося в CSS *рамочным изображением* (`border image`).

В отличие от фоновое изображение, задаваемого с помощью свойства `background-image`, обрамляющее изображение разрезается по сетке 3 × 3, как показано на рис. 19.9. При заполнении фона виджета четыре угла (ячейки *A*, *C*, *G* и *I*) берутся как есть, а остальные пять ячеек растягиваются или мостят доступное пространство.



(a) Исходное изображение (b) Полученное изображение

Рис. 19.9. Разрезание обрамляющего изображения

Рамочные изображения задаются с использованием свойства `border-image`, для чего требуется указать имя файла изображения и четыре «разреза», формирующие девять ячеек. Эти разрезы определяются как указанные в пикселях расстояния от верхнего, правого, нижнего и левого краев. Изображение `border.png` с разрезами на расстоянии 4, 8, 12 и 16 пикселей сверху, справа, снизу и слева определяется так:

```
border-image: url(border.png) 4 8 12 16;
```

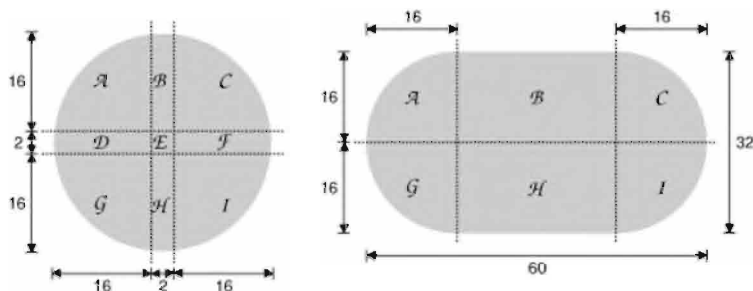
Теперь, когда мы знаем, как работает механизм обрамляющего изображения, давайте рассмотрим, как они используются при определении объектов `QPushButton` в стиле *Candy*. Вот правила, определяющие отображение кнопок в их нормальном состоянии:

```
QPushButton {
    color: white;
    font: bold 10pt;
    border-image: url(/images/button.png) 16;
    border-width: 16px;
    padding: -16px 0px;
    min-height: 32px;
    min-width: 60px;
}
```

[R5]

В таблице стилей `Candy` четыре разреза обрамляющего изображения для `QPushButton` расположены на расстоянии 16 пикселей от краев изображения размером 34×34 пикселя (рис. 19.10, а). Поскольку все четыре разреза одинаковы, нам нужно только написать значение «16» для разрезов и «16px» в качестве толщины рамки.

В примере с `QPushButton`, показанном на рис. 19.10, б, ячейки обрамляющего изображения, соответствующие позициям *D*, *E* и *F*, были опущены, поскольку кнопка с измененным размером имеет недостаточную высоту, чтобы они потребовались, а ячейки *B* и *H* были растянуты по горизонтали для заполнения дополнительной ширины.



(а) Исходное изображение

(б) Полученное изображение

Рис. 19.10. Обрамляющее изображение для `QPushButton`

Стандартная область применения рамочных изображений – это создание рамки вокруг виджета, где виджет находится внутри рамки. Однако мы «извратили» механизм рамочных изображений и использовали его для создания фона для самого виджета. В результате ячейка *E* была выброшена, а прямоугольник интервалов приобрел высоту 0. Чтобы оставить место для текста на кнопке, мы указали вертикальный интервал 16 пикселей. Данная ситуация показана на рис. 19.11. Если бы мы использовали механизм обрамляющего изображения для определения реальной рамки, нам, вероятно, не захотелось бы, чтобы текст наезжал на нее, но поскольку мы используем его для создания масштабируемого фона, нам нужно, чтобы текст располагался поверх, а не внутри рамки.

С помощью свойств `min-width` и `min-height` мы задаем минимальный размер содержимого кнопки. Выбранные значения дадут гарантию, что останется достаточно места для углов обрамляющего изображения, и что кнопка ОК будет несколько больше обычного, чтобы она лучше выглядела рядом с кнопкой Cancel.

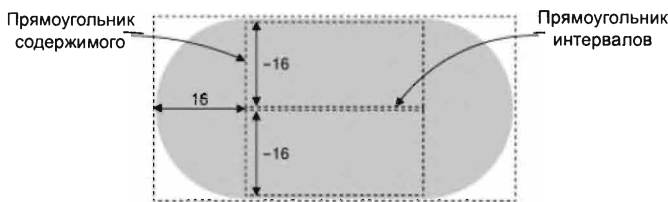


Рис. 19.11. Структура QPushButton

Приведенные выше правила QPushButton применяются ко всем таким кнопкам. Теперь мы определим несколько дополнительных правил, которые применяются, только когда кнопки находятся в определенных состояниях:

```
QPushButton:hover {  
    border-image: url(/images/button-hover.png) 16;  
}
```

[R6]

Когда указатель мыши наводится на объект QPushButton, состояние :hover становится равным true, и указанное для данного состояния правило подменяет любые правила, имеющие менее специфичный селектор. Здесь мы использовали этот метод для создания несколько более яркого обрамляющего изображения с получением красивого эффекта наведения. Другие атрибуты QPushButton, которые были указаны ранее, продолжают использоваться. Изменяется только атрибут border-image.

```
QPushButton:pressed {  
    color: lightgray;  
    border-image: url(/images/button-pressed.png) 16;  
    padding-top: -15px;  
    padding-bottom: -17px;  
}
```

[R7]

Когда пользователь нажимает кнопку, мы изменяем цвет переднего плана на светло-серый, используем более яркое обрамляющее изображение, а также смещаем текст на кнопке на один пиксель вниз, корректируя интервал.

В последних стилевых правилах мы настраиваем внешний вид полей QComboBox. Чтобы показать вам степень контроля и точности, которой можно добиться при помощи таблиц стилей, мы по-разному оформим комбинированные поля (поля с раскрывающимися списками), доступные для редактирования и доступные только для чтения (рис. 19.12). Поля, доступные только для чтения, отображаются, сходно с QPushButton, со стрелкой, расположенной справа, а редактируемые поля состоят из компонента, похожего на QLineEdit, и небольшой округлой кнопки. Оказывается, что мы можем повторно использовать большинство правил, которые мы определили для объектов QLineEdit, QListView и QPushButton.

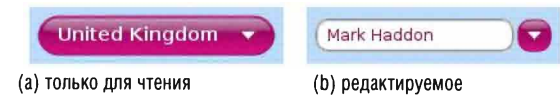


Рис. 19.12. QComboBox в стиле Candy

- **Правило, определяющее внешний вид объектов QLineEdit, можно использовать для определения стиля редактируемых комбинированных полей.**

```
QComboBox:editable,
QLineEdit,
QListView {
    color: rgb(127, 0, 63);
    background-color: rgb(255, 255, 241);
    selection-color: white;
    selection-background-color: rgb(191, 31, 127);
    border: 2px groove gray;
    border-radius: 10px;
    padding: 2px 4px;
}
```

[R3']

- **Правила, которые определяют стиль объектов QPushButton в нормальном состоянии, могут применяться и к комбинированным полям, доступным только для чтения:**

```
QComboBox:!editable,
QPushButton {
    color: white;
    font: bold 10pt;
    border-image: url(/images/button.png) 16;
    border-width: 16px;
    padding: -16px 0px;
    min-height: 32px;
    min-width: 60px;
}
```

[R5']

- **Наведение курсора мыши на комбинированное поле, доступное только для чтения, или на кнопку раскрытия списка редактируемого поля должно приводить к изменению фонового рисунка, так же, как это было сделано для QPushButton:**

```
QComboBox:!editable:hover,
QComboBox::drop-down:editable:hover,
QPushButton:hover {
    border-image: url(/images/button-hover.png) 16;
}
```

[R6']

Нажатие кнопки мыши на комбинированном поле напоминает аналогичное действие для QPushButton:

```
QComboBox:!editable:on,
QPushButton:pressed {
    color: lightgray;
    border-image: url(/images/button-pressed.png) 16;
    padding-top: -15px;
    padding-bottom: -17px;
}
```

[R7']

Повторное использование правил R3, R5, R6 и R7 экономит время и помогает обеспечить единство стиля. Однако мы не определили никаких правил прорисовки кнопок раскрытия, так что мы создадим эти правила сейчас.

```
QComboBox::down-arrow {
    image: url(":/images/down-arrow.png);
}
```

[R8]

Мы указываем наше собственное изображение направленной вниз стрелки, которая будет несколько больше стандартной.

```
QComboBox::down-arrow:on {
    top: 1px;
}
```

[R9]

Если список комбинированного поля раскрыт, стрелка смещается вниз на один пиксель.

```
QComboBox * {
    font: 9pt;
}
```

[R10]

Когда пользователь делает щелчок по комбинированному полю, отображается список элементов. Правило R10 обеспечит, чтобы всплывающее окошко комбинированного поля (или любого другого дочернего виджета) не наследовало более крупный шрифт, который был задан для комбинированного поля правилом R5'.

```
QComboBox::drop-down:!editable {
    subcontrol-origin: padding;
    subcontrol-position: center right;
    width: 11px;
    height: 6px;
    background: none;
}
```

[R11]

С помощью атрибутов `subcontrol-origin` и `subcontrol-position` мы помещаем раскрывающую стрелку комбинированных полей, доступных только для чтения, на правой стороне прямоугольника интервалов с центровкой по вертикали. Кроме того, мы задаем размер этой стрелки равным размеру содержимого кнопки, т. е. изображения `down-arrow.png` 11 × 6 пикселей, а также отключаем для нее фон, поскольку кнопка раскрытия списка представляет собой только стрелку.

```
QComboBox:!editable {
    padding-right: 15px;
}
```

[R12]

Для комбинированных полей, доступных только для чтения, мы указываем правый интервал 15 пикселей, чтобы текст, отображаемый в поле, не перекрывался с раскрывающей стрелкой. На рис. 19.13 показано, как эти размеры соотносятся друг с другом.

Для редактируемых комбинированных полей мы должны настроить раскрывающую кнопку так, чтобы она выглядела как маленькая кнопка `QPushButton`.

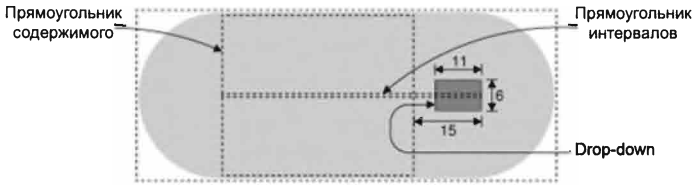


Рис. 19.13. Структура поля QComboBox, доступного только для чтения

```
QComboBox::drop-down:editable {
    border-image: url(":/images/button.png") 16;
    border-width: 10px;
    subcontrol-origin: margin;
    subcontrol-position: center right;
    width: 7px;
    height: 6px;
}
```

[R13]

Мы задаем в качестве обрамляющего изображения файл `button.png`. Однако на этот раз мы указываем ширину рамки 10 пикселей, а не 16, чтобы несколько уменьшить изображение, а также указываем фиксированный размер 7 пикселей по горизонтали и 6 пикселей по вертикали. На рис. 19.14 схематично показан полученный результат.

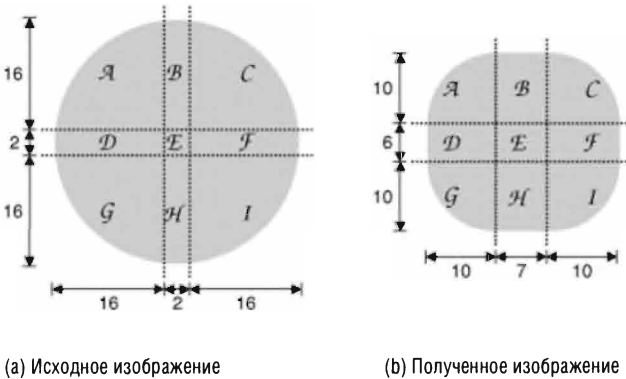


Рис. 19.14. Обрамляющее изображение для раскрывающей кнопки поля QComboBox

Если комбинированное поле раскрыто, мы используем другое, более темное изображение раскрывающей кнопки.

```
QComboBox::drop-down:editable:open {
    border-image: url(":/images/button-pressed.png") 16;
}
```

[R14]

Для редактируемых комбинированных полей мы указываем правое поле 29 пикселей, чтобы оставить место для раскрывающей кнопки, как это показано на рис. 19.15.

```
QComboBox::editable {  
    margin-right: 29px;  
}
```

[R15]

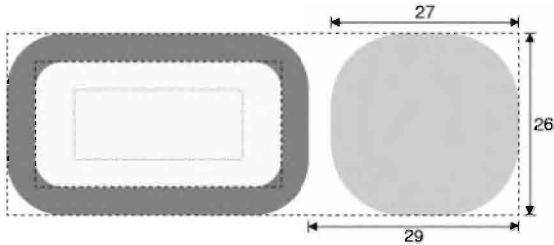


Рис. 19.15. Структура редактируемого поля QComboBox

Итак, мы закончили создание нашей таблицы стилей Candy. Теперь эта таблица занимает около 100 строк и использует несколько пользовательских изображений. В результате мы получили весьма оригинальное диалоговое окно.

Создание своего стиля с помощью таблицы стилей может потребовать большого объема работ, проб и ошибок, особенно для тех, кто никогда раньше не использовал CSS. Одна из главных проблем таблиц стилей состоит в том, что разрешение конфликтов CSS и каскадирования не всегда интуитивно понятно. За дополнительной информацией обращайтесь к онлайн-о документации на сайте <http://doc.trolltech.com/4.3/stylesheet.html>. В этой документации описывается поддержка таблиц стилей в Qt, а также даются ссылки на спецификации CSS.

Создание подклассов класса QStyle

Класс QStyle появился в Qt 2.0 как способ инкапсуляции внешнего облика и функционирования приложения. Такие классы, как QWindowsStyle, QMotifStyle и QCDStyle реализовывали внешний облик для тех платформ и настольных сред, которые Qt поддерживал в то время. В Qt 4.3 предлагается, наряду с базовым абстрактным классом QStyle, восемь стилей, а также удобный базовый класс QCommonStyle. На рис. 19.16 показано, как они соотносятся друг с другом.

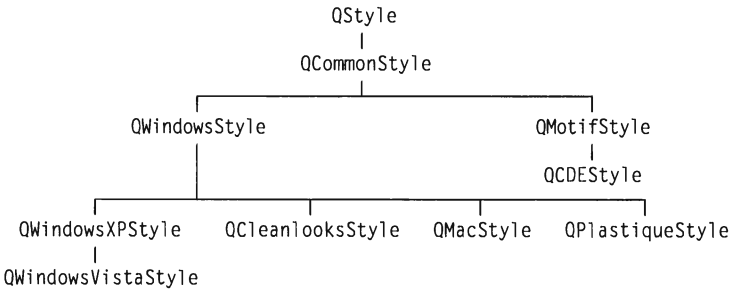


Рис. 19.16. Дерево наследования для встроенных стилей Qt

С помощью архитектуры QStyle разработчики на Qt могут настраивать внешний облик приложения, создавая подклассы QStyle или одного из существующих стилей. Мы можем внести небольшие корректировки в существующий

стиль (например, `QWindowsStyle`) или же разработать целиком пользовательский стиль с нуля.

Интерфейс прикладного программирования (API) `QStyle` состоит из функций для рисования графических элементов (`draw-connect()`, `QObject::disconnect()`, `QCoreApplication::postEvent()`, `QCoreApplication::removePostedEvent()`, `Primitive()`, `drawControl()`, `drawComplexControl()` и т. п.), а также для отправки запросов к стилю (`pixelMetrics()`, `styleHint()`, `hitTest()`, и т. п.). Функции-члены `QStyle`, как правило, принимают объект `QStyleOption`, который содержит как общую информацию о рисуемом виджете (например, палитру), так и информацию, специфичную для данного виджета (например, текст на кнопке).

Функции также могут принимать указатель на класс `QWidget` на тот случай, если класс `QStyleOption` не даст всей необходимой информации. Предположим, нам нужно создать класс `MyPushButton`, который будет выглядеть примерно так, как обычная кнопка, но при этом не будет потомком `QPushButton`. (Пример несколько утрированный, но он поможет нам прояснить взаимоотношения между виджетами и стилями). В обработчике события `paint` класса `MyPushButton` мы задали бы объект `QStyleOption` (в действительности, `QStyleOptionButton`) и вызвали бы функцию `QStyle::drawControl()`:

```
void MyPushButton::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);
    QStyleOptionButton option;
    option.initFrom(this);
    if (isFlat())
        option.features |= QStyleOptionButton::Flat;
    option.text = text();
    style()->drawControl(QStyle::CE_PushButton, &option, &painter,
        this);
}
```

Функция `QStyleOption::initFrom()` инициализирует основные переменные-члены, представляющие виджет, такие как `rect`, `state` (`enabled`, `focused` и т. п.) и `palette`. Переменные-члены, специфичные для `QStyleOptionButton`, необходимо инициализировать вручную. В примере с `MyPushButton` мы инициализируем переменные `features` и `text`, а для переменных `icon` и `iconSize` оставляем значения по умолчанию.

Функция `QWidget::style()` возвращает соответствующий стиль для рисования виджета. Этот стиль, как правило, задается для всего приложения с помощью функции `QApplication::setStyle()`, но также существует возможность изменения стиля для конкретных виджетов при помощи функции `QWidget::setStyle()`.

Функция `drawControl()` переопределяется различными подклассами `QStyle` при рисовании объектов `QPushButton` и других простых виджетов. Типичная реализация может выглядеть следующим образом:

```
void QMotifStyle::drawControl(ControlElement element,
    const QStyleOption *option, QPainter *painter,
    const QWidget *widget) const
{
```

```

switch (element) {
    case CE_CheckBox:
        ...
    case CE_RadioButton:
        ...
    case CE_PushButton:
        if (const QStyleOptionButton *buttonOption =
            qstyleoption_cast<const QStyleOptionButton *>(option)) {
            // нарисовать кнопку
        }
    break;
    ...
}
}

```

Первый параметр, `element`, обозначает тип рисуемого виджета. Если тип — `CE_PushButton`, то стиль пытается привести тип параметра `option` к `QStyleOptionButton`. Если приведение выполняется успешно, рисуется кнопка, описанная в классе `QStyleOptionButton`. Приведение от `const QStyleOption *` к `const QStyleOptionButton *` выполняется с помощью функции `qstyleoption_cast<T>()`, которая возвращает пустой указатель, если `option` не указывает на экземпляр класса `QStyleOptionButton`.

Вместо использования класса `QStyleOption`, в подклассе `QStyle` можно выполнить запрос напрямую к виджету:

```

case CE_PushButton:
    if (const QPushButton *button =
        object_cast<const QPushButton *>(widget)) {
        // нарисовать кнопку
    }
break;

```

Недостаток этого варианта заключается в том, что код стиля привязан к `QPushButton` и, следовательно, не может использоваться для отображения, например, `MyPushButton`. По этой причине встроенные подклассы `QStyle` используют везде, где возможно, для получения информации о рисуемом виджете параметра `QStyleOption`, а к параметру `QWidget` прибегают, только если получить нужную информацию иначе нельзя.

В оставшейся части раздела мы рассмотрим код стиля `Bronze`, показанного на рис. 19.17. Стиль `Bronze` предоставляет округлые кнопки с градиентным фоном, нетрадиционное размещение кнопок счетчиков, экстравагантные флажки и фоновый цвет «начищенной бронзы». Для реализации всех этих эффектов используются современные возможности двумерной графики, такие как сглаживание, полупрозрачность, линейные градиенты и режимы композиции.

Ниже приводится определение класса `BronzeStyle`:

```

class BronzeStyle : public QWindowsStyle
{
    Q_OBJECT
public:

```

```

BronzeStyle() {}
void polish(QPalette &palette);
void polish(QWidget *widget);
void unpolish(QWidget *widget);
int styleHint(StyleHint which, const QStyleOption *option,
              QWidget *widget = 0,
              QStyleHintReturn *returnData = 0) const;
int pixelMetric(PixelMetric which, const QStyleOption *option,
               QWidget *widget = 0) const;
void drawPrimitive(PrimitiveElement which,
                  const QStyleOption *option, QPainter *painter,
                  QWidget *widget = 0) const;
void drawComplexControl(ComplexControl which,
                       const QStyleOptionComplex *option,
                       QPainter *painter,
                       const QWidget *widget = 0) const;
QRect subControlRect(ComplexControl whichControl,
                    const QStyleOptionComplex *option,
                    SubControl whichSubControl,
                    const QWidget *widget = 0) const;

public slots:
    QIcon standardIconImplementation(StandardPixmap which,
                                     const QStyleOption *option,
                                     const QWidget *widget = 0) const;

private:
    void drawBronzeFrame(const QStyleOption *option,
                        QPainter *painter) const;
    void drawBronzeBevel(const QStyleOption *option,
                        QPainter *painter) const;
    void drawBronzeCheckBoxIndicator(const QStyleOption *option,
                                    QPainter *painter) const;
    void drawBronzeSpinBoxButton(SubControl which,
                                const QStyleOptionComplex *option,
                                QPainter *painter) const;
};

```

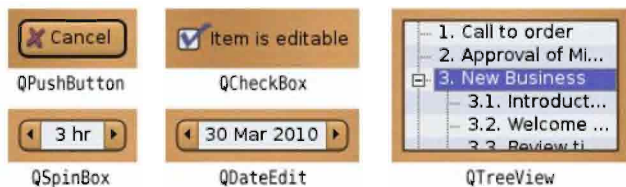


Рис. 19.17. Виджеты в стиле Bronze

При создании пользовательского стиля мы обычно берем за основу существующий стиль, чтобы не приходилось все делать с нуля. В данном примере мы выбрали `QWindowsStyle`, классический стиль `Windows`. Хотя стиль `Bronze` мало чем

напоминает стиль Windows, в `QWindowsStyle` и его базовом классе `QCommonStyle` есть масса кода, который можно использовать при реализации практически любого стиля, который только можно себе представить. Именно поэтому `QMacStyle` основывается на `QWindowsStyle`, хотя и выглядит совершенно иначе.

Стиль `BronzeStyle` переопределяет несколько открытых функций, объявленных в `QStyle`. Функции `polish()` и `unpolish()` вызываются при установке и деинсталляции стиля. Они позволяют настраивать виджеты или палитру. Другие открытые функции представляют собой либо функции запросов (`styleHint()`, `pixelMetric()`, `subControlRect()`), либо функции, рисующие графический элемент (`drawPrimitive()`, `drawComplexControl()`).

Стиль `Bronze` также содержит открытый слот `standardIconImplementation()`. Этот слот Qt обнаруживает методом интроспекции и вызывает при необходимости, как если бы это была виртуальная функция. Иногда в Qt эта идиома используется для добавления функций, которые должны быть виртуальными, не нарушая при этом совместимость по двоичному коду с более ранними версиями Qt. Ожидается, что в Qt 5 слот `standardIconImplementation()` будет заменен стандартной виртуальной функцией `standardIcon()`.

В классе `BronzeStyle` также объявляется несколько закрытых функций. Их мы рассмотрим после объяснения открытых функций.

```
void BronzeStyle::polish(QPalette &palette)
{
    QPixmap backgroundImage(":/images/background.png");
    QColor bronze(207, 155, 95);
    QColor veryLightBlue(239, 239, 247);
    QColor lightBlue(223, 223, 239);
    QColor darkBlue(95, 95, 191);
    palette = QPalette(bronze);
    palette.setBrush(QPalette::Window, backgroundImage);
    palette.setBrush(QPalette::BrightText, Qt::white);
    palette.setBrush(QPalette::Base, veryLightBlue);
    palette.setBrush(QPalette::AlternateBase, lightBlue);
    palette.setBrush(QPalette::Highlight, darkBlue);
    palette.setBrush(QPalette::Disabled, QPalette::Highlight,
                    Qt::darkGray);
}
```

Одной из ярких характеристик стиля `Bronze` является его цветовая схема. Независимо от того, какие цвета пользователь задает в операционной системе, стиль `Bronze` сохраняет свой «бронзовый» облик. Цветовую схему пользовательского стиля можно задать одним из двух методов. Мы можем проигнорировать объект `QPalette` виджета и нарисовать все с помощью наших любимых цветов (бронзового, светло-бронзового, темно-бронзового и т. п.) или же можно переопределить функцию `QStyle::polish(QPalette &)`, настроив палитру виджета, а затем использовать эту палитру. Второй подход является более гибким, поскольку мы можем изменять цветовую схему в подклассе (например, `SilverStyle`) путем переопределения функции `polish()`.

Концепция «доводки» (polishing) является универсальной для виджетов. Когда стиль применяется к виджету, вызывается функция `polish(QWidget *)`, и это позволяет нам выполнить последние корректировки:

```
void BronzeStyle::polish(QWidget *widget)
{
    if (qobject_cast<QAbstractButton *>(widget)
        || qobject_cast<QAbstractSpinBox *>(widget))
        widget->setAttribute(Qt::WA_Hover, true);
}
```

Здесь мы переопределяем функцию `polish(QWidget *)`, задавая атрибут `Qt::WA_Hover` для кнопок и полей счетчиков. Если этот атрибут установлен, то когда курсор мыши заходит или покидает область, занятую виджетом, генерируется событие перерисовки. Это дает нам возможность по-разному отображать виджет в зависимости от того, наведен ли на него курсор мыши.

Эта функция вызывается после того, как виджет был создан, и перед тем, как он будет выведен в первый раз на экран с использованием текущего стиля. Затем он вызывается только в случае динамического изменения текущего стиля.

```
void BronzeStyle::unpolish(QWidget *widget)
{
    if (qobject_cast<QAbstractButton *>(widget)
        || qobject_cast<QAbstractSpinBox *>(widget))
        widget->setAttribute(Qt::WA_Hover, false);
}
```

Если функция `polish(QWidget *)` вызывается при применении стиля к виджету, функция `unpolish(QWidget *)` вызывается при динамическом изменении стиля. Назначение функции `unpolish()` – отменять действие функции `polish()`, чтобы виджет перешел в состояние, в котором к нему можно применить другой стиль. Хорошо написанные стили пытаются выполнить отмену изменений, внесенных в функции `polish()`.

Типичная область применения функции `polish(QWidget *)` – это установка нашего стиливого подкласса в виде фильтра событий виджета. Это необходимо для более сложных настроек. Например, в `QWindowsVistaStyle` и `QMacStyle` этот метод используется для создания анимации в имеющихся по умолчанию кнопках.

```
int BronzeStyle::styleHint(StyleHint which, const QStyleOption *option,
    const QWidget *widget, QStyleHintReturn *returnData) const
{
    switch (which) {
        case SH_DialogButtonLayout:
            return int(QDialogButtonBox::MacLayout);
        case SH_EtchDisabledText:
            return int(true);
        case SH_DialogButtonBox_ButtonsHaveIcons:
            return int(true);
        case SH_UnderlineShortcut:
            return int(false);
    }
```



```

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

```

Функция `styleHint()` возвращает подсказки, касающиеся внешнего облика, и предоставленные стилем. Например, в случае `SH_DialogButtonLayout` мы возвращаем значение `MacLayout`, означающее, что нам нужно, чтобы в `QDialogButtonBox` использовались нормы Mac OS X, где кнопка ОК расположена справа от Cancel. Функция `styleHint()` возвращает значение целочисленного типа. Для тех немногих стиливых подсказок, которые нельзя представить в виде целых чисел, `styleHint()` предоставляет указатель на объект `QStyleHintReturn`, который можно применять.

```

int BronzeStyle::pixelMetric(PixelMetric which,
    const QStyleOption *option,
    const QWidget *widget) const
{
    switch (which) {
        case PM_ButtonDefaultIndicator:
            return 0;
        case PM_IndicatorWidth:
        case PM_IndicatorHeight:
            return 16;
        case PM_CheckBoxLabelSpacing:
            return 8;
        case PM_DefaultFrameWidth:
            return 2;
        default:
            return QWindowsStyle::pixelMetric(which, option, widget);
    }
}

```

Функция `pixelMetric()` возвращает размер элемента пользовательского интерфейса в пикселях. Переопределяя эту функцию, мы оказываем влияние как на прорисовку встроенных виджетов Qt, так и на подсказки, связанные с их размерами.

Мы возвращаем 0 в случае `PM_ButtonDefaultIndicator`, поскольку нам не нужно резервировать дополнительное пространство вокруг заданных по умолчанию кнопок (по умолчанию эти интервалы для `QWindowsStyle` составляют 1 пиксель). У флажков значения `PM_IndicatorWidth` и `PM_IndicatorHeight` контролируют размер элемента управления (обычно это маленький квадрат), а `PM_CheckBoxLabelSpacing` контролирует расстояние между флажком-индикатором и текстом справа от него (см. рис. 19.18). Наконец, значение `PM_DefaultFrameWidth` определяет толщину линии, окружающей `QFrame`, `QPushButton`, `QSpinBox` и многие другие виджеты. Другие значения размеров `PM_xxx` наследуются из базового класса.

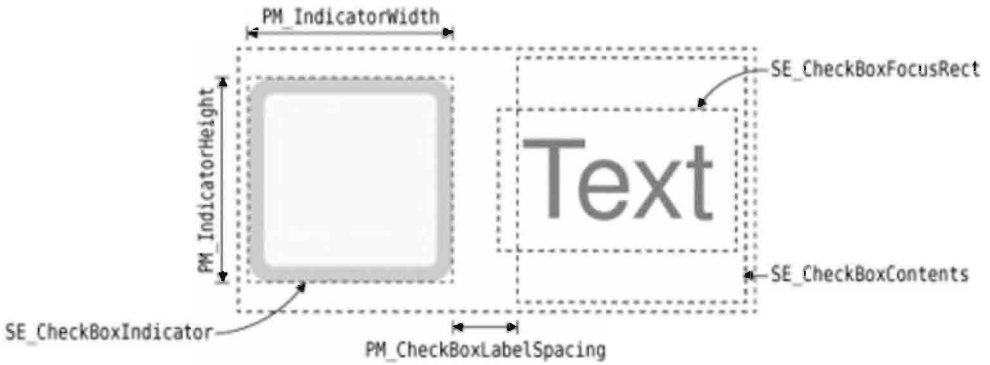


Рис. 19.18. Структура QCheckBox

```

QIcon BronzeStyle::standardIconImplementation(StandardPixmap which,
    const QStyleOption *option, const QWidget *widget) const
{
    QImage image = QWindowsStyle::standardPixmap(which, option, widget)
        .toImage();
    if (image.isNull())
        return QIcon();
    QPalette palette;
    if (option) {
        palette = option->palette;
    } else if (widget) {
        palette = widget->palette();
    }
    QPainter painter(&image);
    painter.setOpacity(0.25);
    painter.setCompositionMode(QPainter::CompositionMode_SourceAtop);
    painter.fillRect(image.rect(), palette.highlight());
    painter.end();
    return QIcon(QPixmap::fromImage(image));
}

```

Как описывалось ранее, Qt вызывает слот `standardIconImplementation()` для получения стандартных значков, которые должны использоваться в пользовательском интерфейсе. Мы вызываем функцию `standardPixmap()` базового класса, чтобы извлечь значок, и стараемся придать ему голубой оттенок, чтобы он соответствовал остальному стилю. Придание оттенка достигается путем наложения 25% -непрозрачного синего цвета на существующий значок. При помощи режима композиции `SourceAtop` мы сделаем так, чтобы области, которые были прозрачными, остались прозрачными, а не стали на 25% синими и на 75% – прозрачными. Мы описываем режимы композиции в разделе «Высококачественное воспроизведение изображения при помощи QImage» главы 8.

```

void BronzeStyle::drawPrimitive(PrimitiveElement which,
    const QStyleOption *option,

```

```

        QPainter *painter,
        const QWidget *widget) const
{
    switch (which) {
    case PE_IndicatorCheckBox:
        drawBronzeCheckBoxIndicator(option, painter);
        break;
    case PE_PanelButtonCommand:
        drawBronzeBevel(option, painter);
        break;
    case PE_Frame:
        drawBronzeFrame(option, painter);
        break;
    case PE_FrameDefaultButton:
        break;
    default:
        QWindowsStyle::drawPrimitive(which, option, painter, widget);
    }
}

```

Qt вызывает функцию `drawPrimitive()` для рисования «примитивных» элементов пользовательского интерфейса. Эти элементы, как правило, используются несколькими виджетами. Например, `PE_IndicatorCheckBox` используется классами `QCheckBox`, `QGroupBox` и `QItemDelegate` для рисования индикатора-флажка.

В стиле **Bronze** мы переопределяем функцию `drawPrimitive()`, чтобы сформировать собственный облик индикаторов-флажков, кнопок и рамок. Например, на рис. 19.19 показана структура элемента `QPushButton`, к которому должен применяться стиль **Bronze**. Функции `drawBronzeCheckBoxIndicator()`, `drawBronzeBevel()` и `drawBronzeFrame()` представляют собой закрытые функции, которые мы рассмотрим позже.



Рис. 19.19. Структура элемента QPushButton

В случае `PE_FrameDefaultButton` мы не делаем ничего, поскольку нам не нужно рисовать дополнительную рамку вокруг заданных по умолчанию кнопок. Для всех остальных примитивных элементов мы переправляем вызов в базовый класс.

[illegible]

```

if (which == CC_SpinBox) {
    drawBronzeSpinBoxButton(SC_SpinBoxDown, option, painter);
    drawBronzeSpinBoxButton(SC_SpinBoxUp, option, painter);
    QRect rect = subControlRect(CC_SpinBox, option,
                               SC_SpinBoxEditField)
                .adjusted(-1, 0, +1, 0);
    painter->setPen(QPen(option->palette.mid(), 1.0));
    painter->drawLine(rect.topLeft(), rect.bottomLeft());
    painter->drawLine(rect.topRight(), rect.bottomRight());
} else {
    return QWindowsStyle::drawComplexControl(which, option, painter,
                                              widget);
}
}

```

Qt вызывает функцию `drawComplexControl()`, чтобы нарисовать виджеты, состоящие из нескольких дочерних элементов управления, а именно — `QSpinBox`. Поскольку мы хотим придать совершенно новый облик элементу `QSpinBox`, мы переопределяем функцию `drawComplexControl()` и обрабатываем ситуацию `CC_SpinBox`.

Чтобы нарисовать объект `QSpinBox`, мы должны нарисовать кнопки со стрелками вверх и вниз, а также рамку вокруг всего поля счетчика. (Структура `QSpinBox` показана на рис. 19.20.) Поскольку код рисования кнопки со стрелкой вверх практически идентичен коду рисования кнопки со стрелкой вниз, мы вынесли его в закрытую функцию `drawBronzeSpinBoxButton()`. Эта функция также рисует рамку вокруг всего поля счетчика.

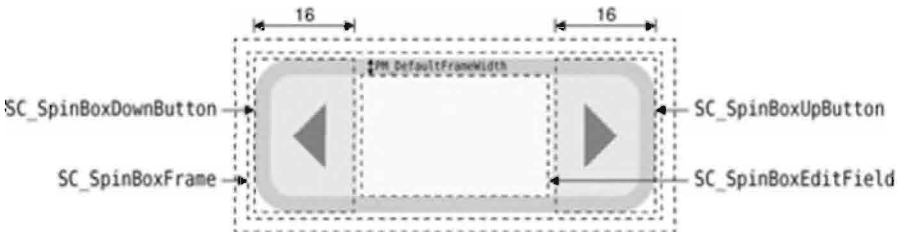


Рис. 19.20. Структура `QSpinBox`

В `QSpinBox` для представления редактируемой части виджета используется элемент `QLineEdit`, так что нам не нужно рисовать эту часть виджета. Однако чтобы четко разделить `QLineEdit` и кнопки счетчиков, мы нарисуем две светло-коричневые вертикальные линии с краю `QLineEdit`. Координаты `QLineEdit` можно получить путем вызова функции `subControlRect()`, указав в качестве третьего аргумента `SC_SpinBoxEditField`.

```

QRect BronzeStyle::subControlRect(ComplexControl whichControl,
    const QStyleOptionComplex *option,
    SubControl whichSubControl,
    const QWidget *widget) const

```

```

if (whichControl == CC_SpinBox) {
    int frameWidth = pixelMetric(PM_DefaultFrameWidth, option,
                                widget);

    int buttonWidth = 16;
    switch (whichSubControl) {
    case SC_SpinBoxFrame:
        return option->rect;
    case SC_SpinBoxEditField:
        return option->rect.adjusted(+buttonWidth, +frameWidth,
                                    -buttonWidth, -frameWidth);
    case SC_SpinBoxDown:
        return visualRect(option->direction, option->rect,
                          QRect(option->rect.x(), option->rect.y(),
                                buttonWidth,
                                option->rect.height()));
    case SC_SpinBoxUp:
        return visualRect(option->direction, option->rect,
                          QRect(option->rect.right() - buttonWidth,
                                option->rect.y(),
                                buttonWidth,
                                option->rect.height()));
    default:
        return QRect();
    }
} else {
    return QWindowsStyle::subControlRect(whichControl, option,
                                         whichSubControl, widget);
}
}

```

Qt вызывает функцию `subControlRect()`, чтобы определить, где находятся дочерние элементы управления виджета. Например, класс `QSpinBox` вызывает эту функцию для определения того, куда помещать объект `QLineEdit`. Также эта функция используется при реагировании на события мыши с целью определения, по какому из дочерних элементов был сделан щелчок. Кроме того, мы вызываем эту функцию самостоятельно при реализации функции `drawComplexControl()`, а также вызываем ее из функции `drawBronzeSpinBoxButton()`. В нашем переопределении мы проверяем, является ли виджет счетчиком (*spin box*), и если является, то возвращаются прямоугольники рамки, поля редактирования, кнопки раскрытия и кнопки свертывания. На рис. 19.20 показано, как эти прямоугольники соотносятся друг с другом. В других виджетах, таких как `QPushButton`, мы используем реализацию базового класса.

Прямоугольники, возвращаемые для значений `SC_SpinBoxDown` и `SC_SpinBoxUp`, передаются при помощи функции `QStyle::visualRect()`. Вызов этой функции имеет следующий синтаксис: `visualRect(direction, outerRect, logicalRect)`.

Если параметр *direction* равен `Qt::LeftToRight`, то *logicalRect* возвращается без изменений, в противном случае *logicalRect* зеркально отражается относительно

outerRect. Это обеспечивает зеркальное отражение графических элементов в режиме письма справа-налево, характерного для некоторых языков, таких как арабский и иврит. Для симметричных элементов, таких как *SC_SpinBoxFrame* и *SC_SpinBoxEditField* отражение не окажет никакого влияния, поэтому нам не нужно вызывать *visualRect()*. Чтобы протестировать стиль в режиме письма справа-налево, мы можем просто передать приложению, использующему данный стиль, опцию командной строки *-reverse*. На рис. 19.21 показан стиль *Bronze* в режиме письма справа налево.

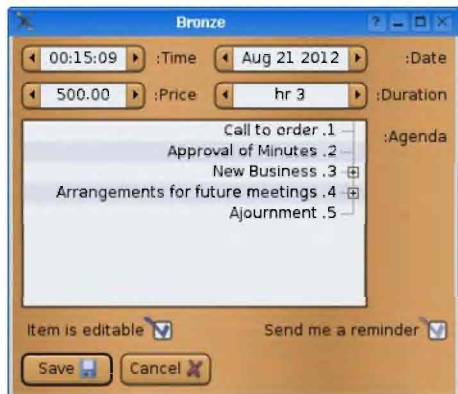


Рис. 19.21. Стиль *Bronze* при режиме письма справа налево

На этом завершается наше рассмотрение открытых функций, переопределяемых из *QWindowsStyle*. Следующие четыре функции представляют собой закрытые функции рисования.

```
void BronzeStyle::drawBronzeBevel(const QStyleOption *option,
                                  QPainter *painter) const
{
    painter->save();
    painter->setRenderHint(QPainter::Antialiasing, true);
    painter->setPen(QPen(option->palette.foreground(), 1.0));
    painter->drawRect(option->rect.adjusted(+1, +1, -1, -1));
    painter->restore();
}
```

Функция *drawBronzeFrame()* вызывалась из функции *drawPrimitive()* для рисования примитивного элемента *PE_Frame*. Она применяется для рисования рамки вокруг объекта *QFrame* (или его подкласса, например *QTreeView*), если рамка имеет форму *QFrame::StyledPanel*. (Другие формы рамок, например, *Box*, *Panel* и *VLine*, рисуются напрямую классом *QFrame* без обращения к стилю.)

Рамка, которую рисуем мы, представляет собой 1-пиксельный сглаженный контур вокруг виджета, созданный кистью цвета переднего плана (к которому можно обращаться через переменную-член *palette* класса *QStyleOption*). Поскольку прямоугольник сглажен, и координаты его целочисленные, результатом будет двух-пиксельный затухающий контур, то есть как раз то, что нужно для нашего стиля *Bronze*.

Чтобы оставить объект `QPainter` в том же состоянии, в котором он был, когда мы к нему обратились, мы вызываем функцию `save()` перед вызовами функций `setRenderHint()` и `setPen()`, а в конце вызываем функцию `restore()`. Это необходимо, поскольку Qt оптимизирует рисование, повторно используя один объект `QPainter` для рисования нескольких графических элементов.

Следующая функция, которую мы рассмотрим, – это функция `drawBronzeBevel()`, которая рисует фон кнопки `QPushButton`.

```
void BronzeStyle::drawBronzeBevel(const QStyleOption *option,
                                QPainter *painter) const
{
    QColor buttonColor = option->palette.button().color();
    int coeff = (option->state & State_MouseOver) ? 115 : 105;
    QLinearGradient gradient(0, 0, 0, option->rect.height());
    gradient.setColorAt(0.0, option->palette.light().color());
    gradient.setColorAt(0.2, buttonColor.lighter(coeff));
    gradient.setColorAt(0.8, buttonColor.darker(coeff));
    gradient.setColorAt(1.0, option->palette.dark().color());
```

Мы начинаем с настройки значения `QLinearGradient`, которое будет использоваться для заливки фона. Градиент светлее сверху и темнее внизу, и проходит через все промежуточные оттенки бронзового цвета. Промежуточные этапы 0.2 и 0.8 придают кнопке псевдо-трехмерный вид, фактор `coeff` контролирует, насколько «трехмерной» будет выглядеть кнопка. Когда на кнопку наводится курсор мыши, мы используем значение `coeff 115%`, чтобы кнопка слегка «приподнялась».

```
double penWidth = 1.0;
if (const QStyleOptionButton *buttonOpt =
    qstyleoption_cast<const QStyleOptionButton *>(option)) {
    if (buttonOpt->features & QStyleOptionButton::DefaultButton)
        penWidth = 2.0;
}
```

В стиле `Bronze` используется контур толщиной 2 пикселя вокруг кнопок, выбранных по умолчанию, и 1-пиксельный контур в остальных случаях. Чтобы определить, является ли кнопка выбранной по умолчанию, мы приводим тип объекта `option` к `const QStyleOptionButton *` и проверяем его переменную-член `features`.

```
QRect roundRect = option->rect.adjusted(+1, +1, -1, -1);
if (!roundRect.isValid())
    return;
int diameter = 12;
int cx = 100 * diameter / roundRect.width();
int cy = 100 * diameter / roundRect.height();
```

Мы определяем еще несколько переменных, которые будут использоваться ниже при рисовании кнопки. Коэффициенты `sx` и `sy` указывают, насколько закругленными будут углы кнопки. Они вычисляются на основе желаемого диаметра закругления углов.

```

painter->save();
painter->setPen(Qt::NoPen);
painter->setBrush(gradient);
painter->drawRoundRect(roundRect, cx, cy);
if (option->state & (State_On | State_Sunken)) {
    QColor slightlyOpaqueBlack(0, 0, 0, 63);
    painter->setBrush(slightlyOpaqueBlack);
    painter->drawRoundRect(roundRect, cx, cy);
}
painter->setRenderHint(QPainter::Antialiasing, true);
painter->setPen(QPen(option->palette.foreground(), penWidth));
painter->setBrush(Qt::NoBrush);
painter->drawRoundRect(roundRect, cx, cy);
painter->restore();
}

```

Наконец, мы выполняем рисование. Мы начинаем с рисования фона с использованием значения `QLinearGradient`, которое мы определили ранее в данной функции. Если кнопка в данный момент нажата (или если это кнопка-переключатель, находящаяся во включенном состоянии), мы накладываем на нее прозрачный на 75% черный цвет, делая ее несколько темнее.

После того, как мы нарисовали фон, мы включаем сглаживание, чтобы получить гладкие закругленные края, задаем соответствующее перо, очищаем кисть и рисуем контур.

```

void BronzeStyle::drawBronzeSpinBoxButton(SubControl which,
    const QStyleOptionComplex *option, QPainter *painter) const
{
    PrimitiveElement arrow = PE_IndicatorArrowLeft;
    QRect buttonRect = option->rect;
    if ((which == SC_SpinBoxUp)
        != (option->direction == Qt::RightToLeft)) {
        arrow = PE_IndicatorArrowRight;
        buttonRect.translate(buttonRect.width() / 2, 0);
    }
    buttonRect.setWidth((buttonRect.width() + 1) / 2);
    QStyleOption buttonOpt(*option);
    painter->save();
    painter->setClipRect(buttonRect, Qt::IntersectClip);
    if (!(option->activeSubControls & which))
        buttonOpt.state &= ~(State_MouseOver | State_On | State_Sunken);
    drawBronzeBevel(&buttonOpt, painter);
    QStyleOption arrowOpt(buttonOpt);
    arrowOpt.rect = subControlRect(CC_SpinBox, option, which)
        .adjusted(+3, +3, -3, -3);
    if (arrowOpt.rect.isValid())
        drawPrimitive(arrow, &arrowOpt, painter);
    painter->restore();
}

```


Функция `drawBronzeSpinBoxButton()` рисует кнопки со стрелками увеличения и уменьшения счетчика, в зависимости от того, указано значение `SC_SpinBoxDown` или `SC_SpinBoxUp`. Мы начинаем с того, что задаем стрелку, используемую при рисовании кнопки (стрелка вправо или влево), и прямоугольник, в котором мы будем рисовать кнопку.

Если `which` равно `SC_SpinBoxDown` (или если `which` равно `SC_SpinBoxUp`, и используется направление справа налево), мы используем стрелку влево (`PE_IndicatorArrowLeft`) и рисуем кнопку в левой половине прямоугольника счетчика. В противном случае мы используем стрелку вправо и рисуем кнопку в правой части.

Для рисования кнопки мы вызываем функцию `drawBronzeBevel()`, передавая в нее объект `QStyleOption`, правильно отражающий состояние кнопки счетчика, которую мы хотим нарисовать. Например, если курсор мыши наведен на счетчик, но не на кнопку счетчика, соответствующую объекту `which`, мы очищаем флаги `State_MouseOver`, `State_On` и `State_Sunken` в состоянии объекта `QStyleOption`. Это необходимо для того, чтобы две кнопки счетчика вели себя независимо друг от друга.

Прежде чем выполнять рисование, мы вызываем функцию `setClipRect()`, чтобы задать отсекающий прямоугольник объекта `QPainter`. Это нужно потому, что мы хотим нарисовать только левую или правую часть наклонной поверхности кнопки, а не всю поверхность.

Наконец, мы рисуем стрелку, вызывая функцию `drawPrimitive()`. В объекте `QStyleOption`, используемом для рисования стрелки, задается прямоугольник, соответствующий прямоугольнику кнопки счетчика (`SC_SpinBoxUp` или `SC_SpinBoxDown`), но несколько меньшего размера, чтобы стрелка получилась меньше.

```
void BronzeStyle::drawBronzeSpinBoxButton(SubControl which,
    const QStyleOption *option, QPainter *painter) const
{
    painter->save();
    painter->setRenderHint(QPainter::Antialiasing, true);
    if (option->state & State_MouseOver) {
        painter->setBrush(option->palette.alternateBase());
    } else {
        painter->setBrush(option->palette.base());
    }
    painter->drawRoundRect(option->rect.adjusted(+1, +1, -1, -1));
    if (option->state & (State_On | State_NoChange)) {
        QPixmap pixmap;
        if (!(option->state & State_Enabled)) {
            pixmap.load(":/images/checkmark-disabled.png");
        } else if (option->state & State_NoChange) {
            pixmap.load(":/images/checkmark-partial.png");
        } else {
            pixmap.load(":/images/checkmark.png");
        }
        QRect pixmapRect = pixmap.rect()
```

Версии QStyleOption

Информация, которая нужна объекту QStyle для того, чтобы нарисовать виджет, передается с помощью класса QStyleOption и его подклассов (QStyleOptionButton, QStyleOptionComboBox, QStyleOptionFrame и т. п.). Из соображений производительности данные хранятся в открытых переменных-членах.

Чтобы обеспечивать двоичную совместимость по всем версиям Qt 4, Trolltech не может добавлять новые переменные-члены в эти классы до тех пор, пока не появится Qt 5. Чтобы можно было вносить изменения в будущие версии Qt 4.x, в классе QStyleOption есть переменная version, которую можно использовать для того, чтобы различать разные версии одного класса. Когда возникает необходимость в новых членах для данных, Trolltech добавляет их в подкласс, имеющий тот же тип, но другую версию. Например, в Qt 4.1 появился класс QStyleOptionFrameV2, являющийся потомком QStyleOptionFrame, но также содержащий переменную-член feature, к которой может обращаться стиль. Подкласс QStyleOptionFrameV2 имеет тип SO_Frame, но версию 2, а не 1. В подклассе QStyle мы можем использовать QStyleOptionFrame как обычно, а если мы захотим обратиться к переменной features, которая определена только в версии 2 этого класса, мы можем написать такой код:

```
if (const QStyleOptionFrame *frameOption =
    qstyleoption_cast<const QStyleOptionFrame *>(option)) {
    QStyleOptionFrameV2 frameOptionV2(*frameOption);
    int lineWidth = frameOptionV2.lineWidth;
    bool flat = (frameOptionV2.features & QStyleOptionFrameV2::Flat);
    ...
}
```

Конструктор копий QStyleOptionFrameV2 принимает как экземпляры версии 1, так и экземпляры версии 2 этого класса. Если принят объект версии 1, конструктор копий инициализирует поле features значением None, в противном случае он копирует поле features из объекта frameOption.

Другой способ получения такого же эффекта, но без копирования – это использование функции qstyleoption_cast<T>() для различения версий:

```
if (const QStyleOptionFrame *frameOption =
    qstyleoption_cast<const QStyleOptionFrame *>(option)) {
    int lineWidth = frameOption.lineWidth;
    bool flat = false;
    if (const QStyleOptionFrame *frameOptionV2 =
        qstyleoption_cast<const QStyleOptionFrameV2 *>(option))
        flat = (frameOptionV2.features & QStyleOptionFrameV2::Flat);
    ...
}
```

В Qt 5 переменная features, скорее всего, будет перенесена в класс QStyleOptionFrame, поскольку Qt не обеспечивает двоичной совместимости между крупными версиями.

```

        .translated(option->rect.topLeft())
        .translated(+2, -6);
    QRect painterRect = visualRect(option->direction, option->rect,
                                   pixmapRect);
    if (option->direction == Qt::RightToLeft) {
        painter->scale(-1.0, +1.0);
        painterRect.moveLeft(-painterRect.right() - 1);
    }
    painter->drawPixmap(painterRect, pixmap);
}
painter->restore();
}

```

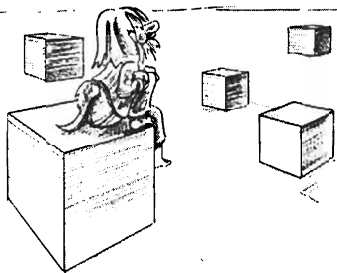
Хотя код функции `drawBronzeCheckBoxIndicator()` на первый взгляд может показаться сложным, нарисовать индикатор-флажок, в действительности, весьма просто: Мы рисуем прямоугольник функцией `drawRoundRect()`, а затем рисуем галочку функцией `drawPixmap()`. Сложности возникают из-за того, что нам нужно использовать другой цвет фона, когда курсор мыши наводится на индикатор, поскольку мы хотим разделять обычные флажки, недоступные флажки и частично доступные флажки (для флажков, которые могут находиться в трех состояниях), а также, поскольку мы зеркально отражаем флажок в режиме письма справа-налево (отражая координатную систему объекта `QPainter`).

Итак, мы завершили реализацию нашего подкласса `Bronze` класса `QStyle`. На снимках экрана, показанных на рис. 19.17, демонстрируются виджеты `QDateEdit` и `QTreeWidget`, использующие стиль `Bronze`, хотя мы и не писали код специально для них. Это объясняется тем, что и `QDateEdit`, и `QDoubleSpinBox`, а также некоторые другие виджеты представляют собой «счетчики», и поэтому используют стиль `Bronze`. Аналогично, `QTreeWidget` и все прочие классы, являющиеся потомками `QFrame`, приобретают особый внешний облик, формируемый стилем `Bronze`. Стиль `Bronze`, представленный в этом разделе, легко может применяться в приложении, для чего его нужно подключить и вызвать функцию

```
QApplication::setStyle(new BronzeStyle);
```

в функции `main()` приложения. Виджеты, которые не затрагиваются явным образом стилем `Bronze`, будут иметь классический облик `Windows`. Пользовательские стили можно скомпилировать в виде плагинов, и использовать затем в `Qt Designer` для просмотра форм в данном стиле. В главе 21 мы покажем, как сделать стиль `Bronze` доступным в форме плагина `Qt`.

Хотя разработанный нами стиль занимает около 300 строк кода, помните, что разработка полностью функционального пользовательского стиля – это серьезная задача, требующая написания 3000 – 5000 строк кода `C++`. По этой причине часто проще и удобнее использовать везде, где только возможно, таблицы стилей `Qt` или использовать комбинированный подход, объединяющий таблицы стилей и пользовательские подклассы `QStyle`. Если вы планируете создать свой подкласс `QStyle`, то реализация стилей и создание виджетов, использующих стили, подробно описаны в документе <http://doc.trolltech.com/4.3/style-reference.html>.



- *Рисование при помощи OpenGL*
- *Комбинирование OpenGL и QPainter*
- *Создание наложений с помощью Framebuffer*

Глава 20. Графика 3D

OpenGL является стандартным программным интерфейсом, предназначенным для воспроизведения трехмерной графики. Приложения Qt могут отображать графику 3D, используя модуль QtOpenGL, который рассчитан на применение системной библиотеки OpenGL. Этот модуль предоставляет класс `QGLWidget`, для которого мы можем создавать подклассы для разработки собственных виджетов, рисуемых с использованием команд OpenGL. Для многих 3D-приложений это весьма существенно. В первом разделе этой главы представлено простое приложение, использующее данный метод для рисования тетраэдра и обеспечения пользователю возможности управлять им при помощи мыши.

Версия Qt 4 впервые предоставляет возможность использовать рисовальщик `QPainter` применительно к `QGLWidget`, как если бы это был простой виджет `QWidget`. Большое преимущество здесь состоит в том, что мы получаем высокую производительность OpenGL для большинства операций рисования, таких как трансформации и пиксельные рисунки. Другое преимущество использования `QPainter` состоит в том, что мы можем использовать для двухмерной графики более высокоуровневый программный интерфейс, а также использовать его вместе с вызовами OpenGL для трехмерной графики. Во втором разделе данной главы мы покажем, как объединить двухмерную и трехмерную графику в одном виджете с использованием смеси команд `QPainter` и `OpenGL`.

С помощью класса `QGLWidget` мы можем рисовать трехмерные картины на экране, используя в качестве основы OpenGL. Чтобы отображать выходящую за пределы экрана поверхность с использованием аппаратного ускорения, мы можем использовать расширения *pbuffer* и *framebuffer object*, доступ к которым осуществляется через классы `QGLPixelBuffer` и `QGLFramebufferObject`. В третьем разделе данной главы мы покажем, как использовать *framebuffer object* для реализации наложений.

В данной главе предполагается, что вы знакомы с OpenGL. Если вы не знакомы с OpenGL, хорошо начинать его изучение с посещения сайта <http://www.opengl.org/>.

Рисование при помощи OpenGL

Вывод графики при помощи OpenGL в приложении Qt выполняется достаточно просто: мы должны создать подкласс `QGLWidget`, переопределить несколько виртуальных функций и собрать приложение вместе с библиотеками *QtOpenGL* и OpenGL. Из-за того, что `QGLWidget` происходит от `QWidget`, большая часть наших знаний остается применимой и здесь. Основное отличие заключается в том, что вместо `QPainter` для выполнения графических операций мы используем стандартные функции библиотеки OpenGL.



Рис. 20.1. Приложение Тетраэдр

Для демонстрации этого подхода мы рассмотрим программный код приложения Тетраэдр, показанного на рис. 20.1. Это приложение отображает в пространстве тетраэдр или четырехгранник, грани которого имеют различные цвета. Пользователь может поворачивать тетраэдр, нажимая кнопку мыши и перемещая ее. Пользователь может задавать цвет поверхности грани путем двойного щелчка с последующим выбором цвета в диалоговом окне `QColorDialog`, которое выдается на экран.

```
class Tetrahedron : public QGLWidget
{
    Q_OBJECT

public:
    Tetrahedron(QWidget *parent = 0);

protected:
    void initializeGL();
    void resizeGL(int width, int height);
    void paintGL();
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void mouseDoubleClickEvent(QMouseEvent *event);

private:
    void draw();
    int faceAtPosition(const QPoint &pos);
```

```

    GLfloat rotationX;
    GLfloat rotationY;
    GLfloat rotationZ;
    QColor faceColors[4];
    QPoint lastPos;
};

```

Класс Tetrahedron происходит от `QGLWidget`. Функции класса `QGLWidget` `initializeGL()`, `resizeGL()` и `paintGL()` переопределяются. Обработчики событий мыши класса `QWidget` переопределяются обычным образом.

```

Tetrahedron::Tetrahedron(QWidget *parent)
    : QGLWidget(parent)
{
    setFormat(QGLFormat(QGL::DoubleBuffer | QGL::DepthBuffer));
    rotationX = -21.0;
    rotationY = -57.0;
    rotationZ = 0.0;
    faceColors[0] = Qt::red;
    faceColors[1] = Qt::green;
    faceColors[2] = Qt::blue;
    faceColors[3] = Qt::yellow;
}

```

В конструкторе мы вызываем функцию `QGLWidget::setFormat()` для установки контекста экрана OpenGL, и мы инициализируем закрытые переменные этого класса.

```

void Tetrahedron::initializeGL()
{
    qglClearColor(Qt::black);
    glShadeModel(GL_FLAT);
    glEnable(GL_DEPTH_TEST);
    glEnable(GL_CULL_FACE);
}

```

Функция `initializeGL()` вызывается только один раз, перед вызовом функции `paintGL()`. Именно в этом месте мы можем задавать контекст воспроизведения OpenGL, определять списки отображаемых элементов и выполнять остальную инициализацию.

Весь программный код является стандартным кодом OpenGL, за исключением вызовов функции `qglClearColor()` класса `QGLWidget`. Если бы мы захотели строго придерживаться стандартных возможностей OpenGL, мы вместо этого вызывали бы функцию `glClearColor()` при использовании режима RGBA и `glClearColorIndex()` при использовании режима индексированных цветов.

```

void Tetrahedron::resizeGL(int width, int height)
{
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
}

```

```

    GLfloat x = GLfloat(width) / height;
    glFrustum(-x, x, -1.0, 1.0, 4.0, 15.0);
    glMatrixMode(GL_MODELVIEW);
}

```

Функция `resizeGL()` **вызывается один раз, перед первым вызовом функции** `paintGL()`, **но после вызова функции** `initializeGL()`. Она также всегда вызывается при изменении размера виджета. Именно в этом месте мы можем задавать область отображения OpenGL, ее проекцию и делать любые другие настройки, зависящие от размера виджета.

```

void Tetrahedron::paintGL()
{
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    draw();
}

```

Функция `paintGL()` **вызывается всякий раз, когда необходимо перерисовать виджет. Это напоминает функцию** `QWidget::paintEvent()`, **но вместо функций класса** `QPainter` **здесь мы используем функции библиотеки OpenGL. Реальное рисование выполняется закрытой функцией** `draw()`.

```

void Tetrahedron::draw()
{
    static const GLfloat P1[3] = { 0.0, -1.0, +2.0 };
    static const GLfloat P2[3] = { +1.73205081, -1.0, -1.0 };
    static const GLfloat P3[3] = { -1.73205081, -1.0, -1.0 };
    static const GLfloat P4[3] = { 0.0, +2.0, 0.0 };
    static const GLfloat * const coords[4][3] = {
        { P1, P2, P3 }, { P1, P3, P4 }, { P1, P4, P2 }, { P2, P4, P3 }
    };

    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glTranslatef(0.0, 0.0, -10.0);
    glRotatef(rotationX, 1.0, 0.0, 0.0);
    glRotatef(rotationY, 0.0, 1.0, 0.0);
    glRotatef(rotationZ, 0.0, 0.0, 1.0);

    for (int i = 0; i < 4; ++i) {
        glLoadName(i);
        glBegin(GL_TRIANGLES);
        glColor(faceColors[i]);
        for (int j = 0; j < 3; ++j) {
            glVertex3f(coords[i][j][0], coords[i][j][1],
                      coords[i][j][2]);
        }
        glEnd();
    }
}

```

В функции `draw()` мы рисуем тетраэдр, учитывая повороты по осям x , y и z , а также цвета в массиве `faceColors`. Везде вызываются стандартные функции библиотеки OpenGL, за исключением вызова `qglColor()`. Вместо этого мы могли бы использовать одну из функций OpenGL, `glColor3d()` или `glIndex()`, в зависимости от используемого режима.

```
void Tetrahedron::mousePressEvent(QMouseEvent *event)
{
    lastPos = event->pos();
}

void Tetrahedron::mouseMoveEvent(QMouseEvent *event)
{
    GLfloat dx = GLfloat(event->x() - lastPos.x()) / width();
    GLfloat dy = GLfloat(event->y() - lastPos.y()) / height();

    if (event->buttons() & Qt::LeftButton) {
        rotationX += 180 * dy;
        rotationY += 180 * dx;
        updateGL();
    } else if (event->buttons() & Qt::RightButton) {
        rotationX += 180 * dy;
        rotationZ += 180 * dx;
        updateGL();
    }
    lastPos = event->pos();
}
```

Функции класса `QWidget` `mousePressEvent()` и `mouseMoveEvent()` переопределяются, чтобы разрешить пользователю поворачивать изображение щелчком мыши и ее перемещением. Левая кнопка мыши позволяет пользователю поворачивать вокруг осей x и y , а правая кнопка мыши – вокруг осей x и z .

После модификации переменных `rotationX` и `rotationY` или `rotationZ` мы вызываем функцию `updateGL()` для перерисовки сцены.

```
void Tetrahedron::mouseDoubleClickEvent(QMouseEvent *event)
{
    int face = faceAtPosition(event->pos());
    if (face != -1) {
        QColor color = QColorDialog::getColor(faceColors[face],
                                                this);

        if (color.isValid()) {
            faceColors[face] = color;
            updateGL();
        }
    }
}
```

Функция `mouseDoubleClickEvent()` класса `QWidget` переопределяется, чтобы разрешить пользователю устанавливать цвет грани тетраэдра с помощью двойного

щелчка. Мы вызываем закрытую функцию `faceAtPosition()` для определения той грани, на которой находится курсор (если он вообще находится на какой-нибудь грани куба). При двойном щелчке по грани тетраэдра мы вызываем функцию `QColorDialog::getColor()` для получения нового цвета для этой грани. Затем мы обновляем массив цветов `faceColors` новым цветом и вызываем функцию `updateGL()` для перерисовки экрана.

```
int Tetrahedron::faceAtPosition(const QPoint &pos)
{
    const int MaxSize = 512;
    GLuint buffer[MaxSize];
    GLint viewport[4];

    glGetIntegerv(GL_VIEWPORT, viewport);
    glSelectBuffer(MaxSize, buffer);
    glRenderMode(GL_SELECT);

    glInitNames();
    glPushName(0);

    glMatrixMode(GL_PROJECTION);
    glPushMatrix();
    glLoadIdentity();
    gluPickMatrix(GLdouble(pos.x()),
        GLdouble(viewport[3] - pos.y()),
        5.0, 5.0, viewport);
    GLfloat x = GLfloat(width()) / height();
    glFrustum(-x, x, -1.0, 1.0, 4.0, 15.0);
    draw();
    glMatrixMode(GL_PROJECTION);
    glPopMatrix();

    if (!glRenderMode(GL_RENDER))
        return -1;
    return buffer[3];
}
```

Функция `faceAtPosition()` возвращает номер грани для заданной точки виджета или `-1`, если данная точка не попадает на грань. Программный код этой функции, выполненной с помощью средств OpenGL, немного сложен. Фактически мы переводим работу в режим `GL_SELECT`, чтобы воспользоваться возможностями OpenGL по идентификации элементов изображения, и затем получаем номер грани куба (ее «имя») из записи нажатия OpenGL. Этот код является полностью стандартным кодом OpenGL, за исключением вызова функции `QGLWidget::makeCurrent()` в начале, который необходим для того, чтобы убедиться, что мы используем правильный контекст OpenGL (класс `QGLWidget` делает это автоматически перед вызовом функций `initializeGL()`, `resizeGL()` или `paintGL()`, так что нам не нужно делать этот вызов где-то еще в реализации приложения Тетраэдр.

```

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    if (!QGLFormat::hasOpenGL()) {
        std::cerr << "This system has no OpenGL support" << std::endl;
        return 1;
    }

    Tetrahedron tetrahedron;
    tetrahedron.setWindowTitle(QObject::tr("Tetrahedron"));
    tetrahedron.resize(300, 300);
    tetrahedron.show();

    return app.exec();
}

```

Если система пользователя не поддерживает OpenGL, мы выдаем на консоль сообщение об ошибке и сразу же возвращаем управление.

Для сборки приложения совместно с модулем *QtOpenGL* и системной библиотекой OpenGL файл `.pro` должен содержать следующий элемент:

```
QT += opengl
```

Этим заканчивается разработка приложения Тетраэдр. Более подробную информацию о модуле *QtOpenGL* вы найдете в справочной документации по классам `QGLWidget`, `QGLFormat`, `QGLContext`, `QGLColormap` и `QGLPixelBuffer`.

Комбинирование OpenGL и QPainter

В предыдущем разделе мы увидели, как использовать команды OpenGL для рисования трехмерной картины на виджете `QGLWidget`. Также существует возможность нарисовать на виджете `QGLWidget` двухмерную картину с помощью `QPainter`. В примере приложения Куб Гласных (Vowel Cube), который мы рассмотрим в этом разделе, объединяются вызовы OpenGL и QPainter, и демонстрируется, как получить все лучшее из обоих подходов. Также демонстрируется использование функции `QGLWidget::renderText()`, которая позволяет нарисовать нетрансформированные текстовые аннотации поверх трехмерного изображения. Данное приложение показано на рис. 20.2.

Приложение Куб Гласных отображает окно с восемью гласными буквами турецкого языка, расположенными по углам куба – такое изображение часто можно видеть в учебниках по грамматике и лингвистике турецкого языка. На переднем плане в легенде перечислены категории гласных букв, и какие буквы к какой категории принадлежат. Куб делает эту информацию более наглядной. Например, гласные переднего ряда отображаются на передней грани куба, а гласные заднего ряда – на задней грани. Для переднего плана мы используем радиальный градиент.

```

class VowelCube : public QGLWidget
{

```

```

Q_OBJECT
public:
    VowelCube(QWidget *parent = 0);
    ~VowelCube();
protected:
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);
    void mouseMoveEvent(QMouseEvent *event);
    void wheelEvent(QWheelEvent *event);
private:
    void createGradient();
    void createGLObject();
    void drawBackground(QPainter *painter);
    void drawCube();
    void drawLegend(QPainter *painter);
    GLuint glObject;
    QRadialGradient gradient;
    GLfloat rotationX;
    GLfloat rotationY;
    GLfloat rotationZ;
    GLfloat scaling;
    QPoint lastPos;
};

```

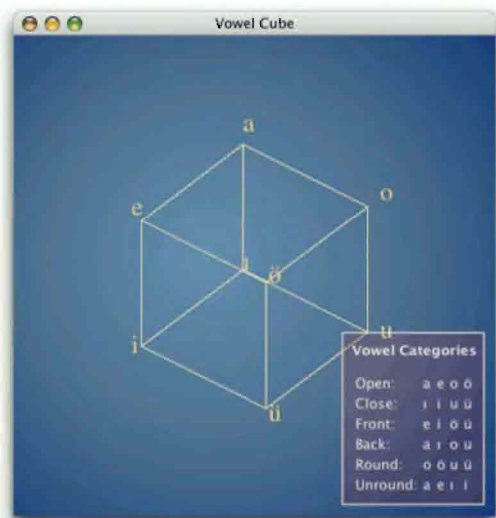


Рис. 20.2. Приложение Куб Гласных

Класс `VowelCube` является потомком `QGLWidget`. Он использует `QPainter` для рисования фонового градиента, затем рисует куб с помощью вызовов `OpenGL`, а потом рисует восемь гласных букв по углам куба с помощью функции `renderText()` и, наконец, рисует легенду при помощи `QPainter` и `QTextDocument`. Пользователь

может поворачивать куб, нажимая кнопку мыши и выполняя перетаскивание, а также осуществлять масштабирование, используя колесико мыши.

В отличие от примера с тетраэдром, приведенного в предыдущем разделе, где мы переопределяли высокоуровневые функции `QGLWidget initializeGL()`, `resizeGL()` и `paintGL()`, на этот раз мы переопределяем традиционные обработчики `QWidget`. Это обеспечивает нам больший контроль над процессом обновления буфера кадров `OpenGL`.

Ниже приводится конструктор класса `Vowel Cube`:

```
VowelCube::VowelCube(QWidget *parent)
    : QGLWidget(parent)
{
    setFormat(QGLFormat(QGL::SampleBuffers));
    rotationX = -38.0;
    rotationY = -58.0;
    rotationZ = 0.0;
    scaling = 1.0;
    createGradient();
    createGLObject();
}
```

В конструкторе мы начинаем с вызова функции `QGLWidget::setFormat()`, чтобы указать контекст дисплея `OpenGL` с поддержкой сглаживания. Далее мы инициализируем закрытые переменные класса. В конце мы вызываем функцию `createGradient()`, чтобы задать объект `QRadialGradient`, используемый для заливки фона, и функцию `createGLObject()` для создания объекта-куба `OpenGL`. Сделав все это в конструкторе, мы получим более аккуратные результаты в будущем, когда нам нужно будет перерисовывать изображение.

```
void VowelCube::createGradient()
{
    gradient.setCoordinateMode(QGradient::ObjectBoundingMode);
    gradient.setCenter(0.45, 0.50);
    gradient.setFocalPoint(0.40, 0.45);
    gradient.setColorAt(0.0, QColor(105, 146, 182));
    gradient.setColorAt(0.4, QColor(81, 113, 150));
    gradient.setColorAt(0.8, QColor(16, 56, 121));
}
```

В функции `createGradient()` мы просто указываем, что объект `QRadialGradient` будет использовать различные оттенки синего цвета. Мы вызываем функцию `setCoordinateMode()`, обеспечивающую соответствие координат, указанных для центра и фокусных точек, размерам виджета. Позиции указываются в виде значений с плавающей точкой в диапазоне от 0 до 1, где 0 соответствует фокусной точке, а 1 – контуру окружности, определяемой градиентом.

```
void VowelCube::createGLObject()
{
    makeCurrent();
    glShadeModel(GL_FLAT);
}
```

```

    glObject = glGenLists(1);
    glNewList(glObject, GL_COMPILE);
    glColor(QColor(255, 239, 191));
    glLineWidth(1.0);
    glBegin(GL_LINES);
    glVertex3f(+1.0, +1.0, -1.0);
    ...
    glVertex3f(-1.0, +1.0, +1.0);
    glEnd();
    glEndList();
}

```

Функция `createGLObject()` создает список OpenGL, хранящий изображения линий, образующих куб гласных. Код представляет собой совершенно стандартный код OpenGL, за исключением вызова в начале функции `QGLWidget::makeCurrent()`, которая обеспечивает использование правильного контекста OpenGL.

```

VowelCube::~VowelCube()
{
    makeCurrent();
    glDeleteLists(glObject, 1);
}

```

В деструкторе мы вызываем функцию `glDeleteLists()` для удаления объекта-куба OpenGL, который мы создали в конструкторе. И снова, мы должны вызвать функцию `makeCurrent()`.

```

void VowelCube::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);
    drawBackground(&painter);
    drawCube();
    drawLegend(&painter);
}

```

В функции `paintEvent()` настраиваем объект `QPainter`, как это обычно делается для `QWidget`, затем рисуем фон, куб и легенду.

```

void VowelCube::drawBackground(QPainter *painter)
{
    painter->setPen(Qt::NoPen);
    painter->setBrush(gradient);
    painter->drawRect(rect());
}

```

Рисование фона производится просто путем вызова функции `drawRect()` с соответствующей кистью.

Функция `drawCube()` является главной в нашем пользовательском виджете. Мы рассмотрим ее, разделив на две части:

```

void VowelCube::drawCube()
{

```

```

glPushAttrib(GL_ALL_ATTRIB_BITS);
glMatrixMode(GL_PROJECTION);
glPushMatrix();
glLoadIdentity();
GLfloat x = 3.0 * GLfloat(width()) / height();
glOrtho(-x, +x, -3.0, +3.0, 4.0, 15.0);
glMatrixMode(GL_MODELVIEW);
glPushMatrix();
glLoadIdentity();
glTranslatef(0.0, 0.0, -10.0);
glScalef(scaling, scaling, scaling);
glRotatef(rotationX, 1.0, 0.0, 0.0);
glRotatef(rotationY, 0.0, 1.0, 0.0);
glRotatef(rotationZ, 0.0, 0.0, 1.0);
glEnable(GL_MULTISAMPLE);

```

Поскольку между двумя фрагментами кода, использующего `QPainter`, у нас есть код `OpenGL`, мы должны быть осторожны, а именно, мы должны сохранять состояние `OpenGL`, которое мы изменили в функции, и восстанавливать его после завершения работы. Так что мы сохраняем атрибуты `OpenGL`, проекционную матрицу и матрицу представления модели, прежде чем изменять их. В конце мы задаем параметр `GL_MULTISAMPLE`, чтобы включить сглаживание.

```

glCallList(glObject);
setFont(QFont("Times", 24));
qglColor(QColor(255, 223, 127));
renderText(+1.1, +1.1, +1.1, QChar('a'));
renderText(-1.1, +1.1, +1.1, QChar('e'));
renderText(+1.1, +1.1, -1.1, QChar('o'));
renderText(-1.1, +1.1, -1.1, QChar(0x00F6));
renderText(+1.1, -1.1, +1.1, QChar(0x0131));
renderText(-1.1, -1.1, +1.1, QChar('i'));
renderText(+1.1, -1.1, -1.1, QChar('u'));
renderText(-1.1, -1.1, -1.1, QChar(0x00FC));
glMatrixMode(GL_MODELVIEW);
glPopMatrix();
glMatrixMode(GL_PROJECTION);
glPopMatrix();
glPopAttrib();
}

```

Далее мы вызываем функцию `glCallList()` для рисования объекта-куба. Затем мы задаем шрифт и цвет и вызываем функцию `QGLWidget::renderText()` для рисования букв по углам куба. Гласные турецкого языка, которые выходят за рамки диапазона символов ASCII указываются с помощью значений Unicode.

Функция `renderText()` принимает координатный триплет (x, y, z) для позиционирования текста в координатах представления модели. Сам текст трансформации не подвергается.

```

void VowelCube::drawLegend(QPainter *painter)
{
    const int Margin = 11;
    const int Padding = 6;
    QTextDocument textDocument;
    textDocument.setDefaultStyleSheet(„* { color: #FFEFEF }“);
    textDocument.setHtml(„<h4 align=\\\"center\\\">Vowel Categories</h4>\"
        „<p align=\\\"center\\\"><table width=\\\"100%\\\">\"
        „<tr><td>Open:<td>a<td>e<td>o<td>&ouml;\"
        ...
        „</table>“);
    textDocument.setTextWidth(textDocument.size().width());
    QRect rect(QPoint(0, 0), textDocument.size().toSize()
        + QSize(2 * Padding, 2 * Padding));
    painter->translate(width() - rect.width() - Margin,
        height() - rect.height() - Margin);
    painter->setPen(QColor(255, 239, 239));
    painter->setBrush(QColor(255, 0, 0, 31));
    painter->drawRect(rect);
    painter->translate(Padding, Padding);
    textDocument.drawContents(painter);
}

```

В функции `drawLegend()` мы задаем объект `QTextDocument` с HTML-текстом, в котором перечисляются категории гласных турецкого языка и сами буквы, и мы отображаем их поверх полупрозрачного синего прямоугольника.

В виджете `VowelCube` также переопределяются функции `mousePressEvent()`, `mouseMoveEvent()` и `wheelEvent()`, но в этом нет ничего необычного. Как и в стандартном пользовательском виджете `Qt` мы вызываем функцию `update()`, когда хотим запланировать перерисовку. Например, вот код функции `wheelEvent()`:

```

void VowelCube::wheelEvent(QWheelEvent *event)
{
    double numDegrees = -event->delta() / 8.0;
    double numSteps = numDegrees / 15.0;
    scaling *= std::pow(1.125, numSteps);
    update();
}

```

На этом завершается наше рассмотрение данного примера. При переопределении функции-обработчика `paintEvent()` класса `VowelCube` мы использовали следующую общую схему.

1. Создать объект `QPainter`.
2. С помощью `QPainter` нарисовать фон.
3. Сохранить состояние `OpenGL`.
4. Нарисовать изображение с помощью команд `OpenGL`.
5. Восстановить состояние `OpenGL`.

6. С помощью QPainter нарисовать передний план.
7. Удалить объект QPainter.

Существуют и другие возможности. Например, если мы рисуем фон, мы можем сделать следующее.

1. Нарисовать изображение с помощью команд OpenGL.
2. Создать объект QPainter.
3. С помощью QPainter нарисовать передний план.
4. Удалить объект QPainter.

Этому соответствует следующий код:

```
void VowelCube::paintEvent(QPaintEvent * /* event */)
{
    drawCube();
    drawLegend();
}
void VowelCube::drawCube()
{
    ...
}
void VowelCube::drawLegend()
{
    QPainter painter(this);
    ...
}
```

Обратите внимание, что на этот раз мы создаем объект QPainter локально, в функции drawLegend(). Главное преимущество такого подхода состоит в том, что нам не нужно сохранять и восстанавливать состояние OpenGL. Однако сразу это работать не будет, поскольку QPainter автоматически очищает фон перед началом рисования, убирая, таким образом, изображение OpenGL. Чтобы это предотвратить, мы должны вызвать функцию setAutoFillBackground(false) в конструкторе виджета.

Другая интересная схема получается, если мы рисуем фон и куб, но не рисуем передний план.

1. Создать объект QPainter.
2. С помощью QPainter нарисовать фон.
3. Удалить объект QPainter.
4. Нарисовать изображение с помощью команд OpenGL.

И снова мы можем избежать сохранения и восстановления состояния OpenGL. Однако в таком виде схема работать не будет, поскольку QPainter в деструкторе автоматически вызывает функцию QGLWidget::swapBuffers(), чтобы сделать результат рисования видимым, и любые вызовы OpenGL, которые будут выполняться после деструктора QPainter, будут относиться к буферу, не отображаемому на экране, и не будут видны. Чтобы это предотвратить, мы должны вызвать в конструкторе виджета функцию setAutoBufferSwap(false), а в конце функции paintEvent() вызвать swapBuffer(). Например:


```
void VowelCube::paintEvent(QPaintEvent * /* event */)
{
    drawBackground();
    drawCube();
    swapBuffers();
}

void VowelCube::drawBackground()
{
    QPainter painter(this);
    ...
}

void VowelCube::drawCube()
{
    ...
}
```

Таким образом, наиболее общий подход – это создание объекта `QPainter` в функции `paintEvent()` и сохранение и восстановление состояния каждый раз, когда выполняются необработанные операции `OpenGL`. Можно избежать сохранения состояния, если мы будем помнить о следующих моментах.

- Конструктор `QPainter` (или функция `QPainter::begin()`) автоматически вызывает функцию `glClear()`, если только перед этим в виджете не вызвана функция `setAutoFillBackground(false)`.
- Деструктор `QPainter` (или функция `QPainter::end()`) автоматически вызывает функцию `QGLWidget::swapBuffers()` для того, чтобы поменять местами видимый и неотображаемый буферы, если только перед этим в виджете не вызвана функция `setAutoBufferSwap(false)`.
- Когда объект `QPainter` активен, мы можем подавать перемежающиеся необработанные команды `OpenGL`, если мы будем сохранять состояние `OpenGL` перед выполнением таких команд и восстанавливать состояние после их выполнения.

Если помнить об этом, комбинировать `OpenGL` и `QPainter` достаточно просто, и мы можем пользоваться всеми преимуществами, которые дают графические возможности `QPainter` и `OpenGL`.

Создание наложений с помощью объектов `Framebuffer`

Часто нам бывает нужно наложить простую аннотацию поверх сложного трехмерного изображения. Если изображение очень сложное, его отображение может занимать несколько секунд. Чтобы отображение не повторялось при каждом изменении аннотации, мы можем использовать наложения `X11` или встроенную поддержку наложений `OpenGL`.

В последнее время объекты `pbuffer` и `framebuffer` позволяют создавать более удобный и гибкий синтаксис формирования наложений. Основная идея состоит в том, что мы воспроизводим 3D-изображение на неотображаемой на экране поверхности, которую мы связываем с текстурой. Текстура переносится на экран путем рисования прямоугольника, а аннотации рисуются поверх. Когда аннотации


```

        GLfloat diffuseB, GLfloat specularR,
        GLfloat specularG, GLfloat specularB,
        GLfloat shininess);

void drawTeapots();
QGLFrameBufferObject *fbObject;
GLuint glTeapotObject;
QPoint rubberBa2ndCorner1;
QPoint rubberBandCorner2;
bool rubberBandIsShown;
};

```

Класс `Teapots` происходит от `QGLWidget` и переопределяет высокоуровневые обработчики OpenGL `initializeGL()`, `resizeGL()` и `paintGL()`. Также он переопределяет функции `mousePressEvent()`, `mouseMoveEvent()` и `mouseReleaseEvent()`, чтобы пользователь мог рисовать резиновую ленту.

Закрытые функции обеспечивают создание объекта `teapot` и рисование чайников. Код достаточно сложен и основывается на примере из книги «OpenGL Programming Guide», авторы Jackie Neider, Tom Davis и Mason Woo, (Addison-Wesley, 1993). Поскольку этот код не имеет прямого отношения к нашим задачам, мы не будем приводить его здесь.

В закрытых переменных хранятся объект-буфер кадров, объект-чайник, углы резиновой ленты и признак видимости резиновой ленты.

```

Teapots::Teapots(QWidget *parent)
    : QGLWidget(parent)
{
    rubberBandIsShown = false;
    makeCurrent();
    fbObject = new QGLFrameBufferObject(1024, 1024,
                                         QGLFrameBufferObject::Depth);
    createGLTeapotObject();
}

```

Конструктор класса `Teapots` инициализирует закрытую переменную `rubberBandIsShown`, создает объект-буфер кадров и создает объект-чайник. Мы пропустим функцию `createGLTeapotObject()`, поскольку она достаточно длинная и не содержит кода, важного с точки зрения Qt.

```

Teapots::~Teapots()
{
    makeCurrent();
    delete fbObject;
    glDeleteLists(glTeapotObject, 1);
}

```

В деструкторе мы освобождаем ресурсы, связанные с объектом-буфером кадров и объектом-чайником.

```

void Teapots::initializeGL()
{

```

```

static const GLfloat ambient[] = { 0.0, 0.0, 0.0, 1.0 };
static const GLfloat diffuse[] = { 1.0, 1.0, 1.0, 1.0 };
static const GLfloat position[] = { 0.0, 3.0, 3.0, 0.0 };
static const GLfloat lmodelAmbient[] = { 0.2, 0.2, 0.2, 1.0 };
static const GLfloat localView[] = { 0.0 };
glLightfv(GL_LIGHT0, GL_AMBIENT, ambient);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffuse);
glLightfv(GL_LIGHT0, GL_POSITION, position);
glLightModelfv(GL_LIGHT_MODEL_AMBIENT, lmodelAmbient);
glLightModelfv(GL_LIGHT_MODEL_LOCAL_VIEWER, localView);
glFrontFace(GL_CW);
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
glEnable(GL_AUTO_NORMAL);
glEnable(GL_NORMALIZE);
glEnable(GL_DEPTH_TEST);
glDepthFunc(GL_LESS);
}

```

Функция `initializeGL()` переопределяется с целью настройки модели освещения и для включения различных возможностей OpenGL. Этот код взят прямо из примера *Teapots*, описанного в упомянутой выше книге «OpenGL Programming Guide».

```

void Teapots::resizeGL(int width, int height)
{
    fbObject->bind();
    glDisable(GL_TEXTURE_2D);
    glEnable(GL_LIGHTING);
    glEnable(GL_DEPTH_TEST);
    glViewport(0, 0, width, height);
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    if (width <= height) {
        glOrtho(0.0, 20.0, 0.0, 20.0 * GLfloat(height) / GLfloat(width),
                -10.0, 10.0);
    } else {
        glOrtho(0.0, 20.0 * GLfloat(width) / GLfloat(height), 0.0, 20.0,
                -10.0, 10.0);
    }
    glMatrixMode(GL_MODELVIEW);
    drawTeapots();
    fbObject->release();
}

```

Функция `resizeGL()` переопределена с целью перерисовки изображения чайников каждый раз после изменения размера виджета. Чтобы изобразить чайники на объекте-буфере кадров, мы вызываем в самом начале функцию `QGLFramebufferObject::bind()`. Затем мы настраиваем некоторые возможнос-

ти OpenGL и матрицы проекции и представления модели. Вызов функции `drawTeapots()` почти в конце выполняет рисование чайников на объекте-буфере кадров. Наконец, вызывается функция `release()`, которая разрывает связи объекта-буфера кадров, чтобы последующие операции рисования OpenGL не затрагивали наш объект-буфер.

```
void Teapots::paintGL()
{
    glDisable(GL_LIGHTING);
    glViewport(0, 0, width(), height());
    glMatrixMode(GL_PROJECTION);
    glLoadIdentity();
    glMatrixMode(GL_MODELVIEW);
    glLoadIdentity();
    glDisable(GL_DEPTH_TEST);
    glClear(GL_COLOR_BUFFER_BIT);
    glEnable(GL_TEXTURE_2D);
    glBindTexture(GL_TEXTURE_2D, fbObject->texture());
    glColor3f(1.0, 1.0, 1.0);
    GLfloat s = width() / GLfloat(fbObject->size().width());
    GLfloat t = height() / GLfloat(fbObject->size().height());
    glBegin(GL_QUADS);
    glTexCoord2f(0.0, 0.0);
    glVertex2f(-1.0, -1.0);
    glTexCoord2f(s, 0.0);
    glVertex2f(1.0, -1.0);
    glTexCoord2f(s, t);
    glVertex2f(1.0, 1.0);
    glTexCoord2f(0.0, t);
    glVertex2f(-1.0, 1.0);
    glEnd();
}
```

В функции `paintGL()` мы начинаем с перенастройки матриц проекции и представления модели. Затем мы связываем объект-буфер кадров с текстурой и рисуем прямоугольник с текстурой, закрывающий весь виджет.

```
if (rubberBandIsShown) {
    glMatrixMode(GL_PROJECTION);
    glOrtho(0, width(), height(), 0, 0, 100);
    glMatrixMode(GL_MODELVIEW);
    glDisable(GL_TEXTURE_2D);
    glEnable(GL_BLEND);
    glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
    glLineWidth(4.0);
    glColor4f(1.0, 1.0, 1.0, 0.2);
    glRecti(rubberBandCorner1.x(), rubberBandCorner1.y(),
            rubberBandCorner2.x(), rubberBandCorner2.y());
    glColor4f(1.0, 1.0, 0.0, 0.5);
    glLineStipple(3, 0xAAAA);
}
```

```

        glEnable(GL_LINE_STIPPLE);
        glBegin(GL_LINE_LOOP);
        glVertex2i(rubberBandCorner1.x(), rubberBandCorner1.y());
        glVertex2i(rubberBandCorner2.x(), rubberBandCorner1.y());
        glVertex2i(rubberBandCorner2.x(), rubberBandCorner2.y());
        glVertex2i(rubberBandCorner1.x(), rubberBandCorner2.y());
        glEnd();
        glLineWidth(1.0);
        glDisable(GL_LINE_STIPPLE);
        glDisable(GL_BLEND);
    }
}

```

Если резиновая лента в данный момент не отображается, мы рисуем ее поверх прямоугольника. Код представляет собой стандартный код OpenGL.

```

void Teapots::mousePressEvent(QMouseEvent *event)
{
    rubberBandCorner1 = event->pos();
    rubberBandCorner2 = event->pos();
    rubberBandIsShown = true;
}

void Teapots::mouseMoveEvent(QMouseEvent *event)
{
    if (rubberBandIsShown) {
        rubberBandCorner2 = event->pos();
        updateGL();
    }
}

void Teapots::mouseReleaseEvent(QMouseEvent * /* event */)
{
    if (rubberBandIsShown) {
        rubberBandIsShown = false;
        updateGL();
    }
}

```

Обработчики событий мыши обновляют переменные `rubberBandCorner1`, `rubberBandCorner2` и `rubberBandIsShown`, описывающие резиновую ленту, и вызывают функцию `updateGL()`, чтобы запланировать перерисовку изображения. Перерисовка изображения происходит очень быстро, поскольку функция `paintGL()` рисует только текстурированный прямоугольник и резиновую ленту поверх него. Изображение перерисовывается (в функции `resizeGL()`), только если пользователь изменяет размеры окна.

А вот функция `main()` этого приложения:

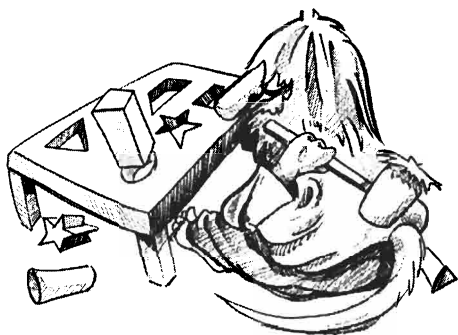
```

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

```

```
if (!QGLFormat::hasOpenGL()) {
    std::cerr << "This system has no OpenGL support" << std::endl;
    return 1;
}
if (!QGLFramebufferObject::hasOpenGLFramebufferObjects()) {
    std::cerr << "This system has no framebuffer object support"
    << std::endl;
    return 1;
}
Teapots teapots;
teapots.setWindowTitle(QObject::tr("Teapots"));
teapots.resize(400, 400);
teapots.show();
return app.exec();
}
```

Данная функция выдает сообщение об ошибке и код ошибки, если в системе отсутствует поддержка OpenGL или не поддерживаются объекты-буферы кадров. Пример приложения Чайники дает нам представление о том, как можно связывать внеэкранную поверхность с текстурой и рисовать на этой текстуре с помощью команд OpenGL. Возможны самые разные варианты, например, мы можем использовать класс QPainter вместо команд OpenGL для рисования на объектах QGLFramebufferObject или QGLPixelBuffer. Это дает нам метод отображения трансформированного текста в сцене OpenGL. Еще один распространенный метод – это использование объекта-буфера кадров для отображения сцены, а затем вызов функции toImage() применительно к результату, с получением объекта QImage. В примерах, поставляемых с Qt, многие эти методы показаны в действии, как применительно к объектам-буферам кадров, так и к объектам pBuffer.



- *Расширение Qt с помощью подключаемых модулей*
- *Как обеспечить в приложении возможность подключения модулей*
- *Написание подключаемых к приложению модулей*

Глава 21. Создание подключаемых модулей

Динамические библиотеки (называемые также совместно используемыми библиотеками или библиотеками DLL) – это независимые модули, хранимые в отдельном файле на диске, доступ к которым могут получать несколько приложений. Как правило, необходимые для программы динамические библиотеки определяются на этапе сборки, в таких случаях эти библиотеки автоматически загружаются при запуске приложения. При таком подходе обычно в файл приложения .pro добавляется библиотека и, возможно, путь доступа к ней, а в исходные файлы включаются соответствующие заголовочные файлы. Например:

```
LIBS      += -ldb_cxx
INCLUDEPATH += /usr/local/BerkeleyDB.4.2/include
```

Альтернативный подход заключается в динамической загрузке библиотеки по мере необходимости, и затем следует разрешение ее символов, которые будут нами использоваться. Qt предоставляет класс `QLibrary` для решения этой задачи независимым от платформы способом. Получая основную часть имени библиотеки, `QLibrary` выполняет поиск соответствующего файла в стандартном для платформы месте. Например, если задано имя `mimetype`, будет выполняться поиск файла `mimetype.dll` в Windows, `mimetype.so` в Linux и `mimetype.dylib` в Mac OS X.

Часто можно расширять функциональные возможности современных приложений с графическим пользовательским интерфейсом за счет применения подключаемых модулей. Подключаемый модуль (`plugin`) – это динамическая библиотека, которая реализует специальный интерфейс для обеспечения дополнительной функциональности. Например, в главе 5 мы создали подключаемый модуль для интеграции пользовательского виджета в *Qt Designer*.

Qt распознает свой собственный набор интерфейсов подключаемых модулей, относящихся к различным областям, включая форматы изображений, драйверы баз данных, стили виджетов, текстовые кодировки и условия доступа. Первый раздел данной главы показывает, как можно расширить возможности Qt с помощью подключаемых модулей.

Кроме того, можно создавать подключаемые модули, предназначенные для конкретных Qt-приложений. Писать такие подключаемые модули в Qt достаточно просто с использованием каркаса Qt для подключаемых модулей, который повышает безопасность и удобство применения класса `QLibrary`. В последних двух разделах данной главы мы покажем, как обеспечить в приложении поддержку подключаемых модулей и как создать пользовательские подключаемые модули для приложения.

Расширение Qt с помощью подключаемых модулей

Qt можно расширять, используя различные типы подключаемых модулей, среди которых наиболее распространенными являются драйверы баз данных, форматы изображений, стили и текстовые кодировки. Каждый тип подключаемых модулей обычно требует наличия, по крайней мере, двух классов: класса-оболочки, который реализует общие функции программного интерфейса подключаемых модулей, и одного или более классов-обработчиков, которые реализуют программный интерфейс для конкретного типа подключаемых модулей. Обработчики вызываются из класса-оболочки. Эти классы показаны на рис. 21.1.

Класс подключаемых модулей	Базовый класс обработчика
<code>QAccessibleBridgePlugin</code>	<code>QAccessibleBridge</code>
<code>QAccessiblePlugin</code>	<code>QAccessibleInterface</code>
<code>QDecorationPlugin</code> ¹	<code>QDecoration*</code>
<code>QFontEnginePlugin</code>	<code>QAbstractFontEngine</code>
<code>QIconEnginePluginV2</code>	<code>QIconEngineV2</code>
<code>QImageIOPlugin</code>	<code>QImageIOHandler</code>
<code>QInputContextPlugin</code>	<code>QInputContext</code>
<code>QKbdDriverPlugin*</code>	<code>QWSKeyboardHandler*</code>
<code>QMouseDriverPlugin*</code>	<code>QWSMouseHandler*</code>
<code>QPictureFormatPlugin</code>	нет
<code>QScreenDriverPlugin*</code>	<code>QScreen*</code>
<code>QScriptExtensionPlugin</code>	нет
<code>QSqlDriverPlugin</code>	<code>QSqlDriver</code>
<code>QStylePlugin</code>	<code>QStyle</code>
<code>QTextCodecPlugin</code>	<code>QTextCodec</code>

Рис. 21.1. Qt-классы подключаемых модулей и обработчиков

Чтобы продемонстрировать, как можно расширить Qt при помощи подключаемых модулей, мы реализуем в этом разделе два подключаемых модуля. Первый представляет собой очень простой подключаемый модуль для стиля `Bronze`, который мы разработали в главе 19. Второй – это модуль, способный считывать в Windows монохромные файлы курсоров. Создать подключаемый модуль для `QStyle` очень просто, при условии, что мы уже разработали сам стиль. Все, что нам нужно – это три файла: файл `.pro`, который будет несколько отличаться от того,

¹ Доступно только в Qt/Embedded Linux

который мы видели ранее, и небольшие файлы .h и .cpp с подклассом `QStylePlugin`, функционирующим как оболочка для стиля. Мы начнем с файла .h:

```
class BronzeStylePlugin : public QStylePlugin
{
    HAvailable only in Qt/Embedded Linux.
public:
    QStringList keys() const;
    QStyle *create(const QString &key);
};
```

Все подключаемые модули, как минимум, предоставляют функцию `keys()` и функцию `create()`. Функция `keys()` возвращает список объектов, которые может создать подключаемый модуль. В случае стилевых подключаемых модулей в таких ключах не учитывается регистр символов, так что «mystyle» и «MyStyle» будут рассматриваться как идентичные. Функция `create()`, принимая ключ, возвращает объект. Ключ должен входить в список, возвращаемый функцией `keys()`.

Файл .cpp почти такой же маленький и простой, как файл .h.

```
QStringList BronzeStylePlugin::keys() const
{
    return QStringList() << "Bronze";
}
```

Функция `keys()` возвращает список стилей, предоставляемых подключаемым модулем. Здесь мы имеем только один стиль, называющийся «Bronze».

```
QStyle *BronzeStylePlugin::create(const QString &key)
{
    if (key.toLower() == "bronze")
        return new BronzeStyle;
    return 0;
}
```

Если ключ равен «Bronze» (регистр символов не учитывается), мы создаем объект `BronzeStyle` и возвращаем его.

В конец файла .cpp мы должны добавить следующий макрос, чтобы стиль экспортировался правильно:

```
Q_EXPORT_PLUGIN2(bronzeplugin, BronzeStylePlugin)
```

Первый аргумент функции `Q_EXPORT_PLUGIN2()` представляет собой базовое имя целевой библиотеки, без расширения, префикса и номера версии. По умолчанию утилита `qmake` использует как базовое имя название текущей директории. Но такое поведение можно изменить с помощью записи `TARGET` в файле .pro. Второй аргумент функции `Q_EXPORT_PLUGIN2()` представляет собой имя класса подключаемого модуля.

Файл .pro для приложений отличается от файла .pro для подключаемых модулей, так что в заключение мы рассмотрим файл .pro стиля `Bronze`:

```
TEMPLATE    = lib
CONFIG      += plugin
```

```
HEADERS    = ../bronze/bronzestyle.h \
            bronzestyleplugin.h
SOURCES    = ../bronze/bronzestyle.cpp \
            bronzestyleplugin.cpp
RESOURCES  = ../bronze/bronze.qrc
DESTDIR    = $$[QT_INSTALL_PLUGINS]/styles
```

По умолчанию в файлах `.pro` используется шаблон `app`, но здесь мы должны указать шаблон `lib`, поскольку подключаемый модуль представляет собой библиотеку, а не самостоятельное приложение. Строка `CONFIG` используется для того, чтобы уведомить Qt, что это не простая библиотека, а библиотека подключаемого модуля. В строке `DESTDIR` указывается директория, куда должен быть помещен подключаемый модуль. Все подключаемые модули Qt должны располагаться в соответствующих модулям поддиректориях той директории, куда установлен Qt. Этот путь встроен в `qmake`, и его можно увидеть в переменной `$$[QT_INSTALL_PLUGINS]`. Поскольку наш подключаемый модуль предоставляет новый стиль, мы помещаем его в директорию `plugins/styles`. Список имен директорий и типов подключаемых модулей можно найти на странице <http://doc.trolltech.com/4.3/plugins-howto.html>.

Подключаемые модули, встраиваемые в Qt в режиме выпуска (`release`) и в режиме отладки (`debug`) отличаются, так что если установлены обе версии Qt, лучше всего указать в файле `.pro`, какую версию нужно использовать, например, добавив такую строку:

```
CONFIG += release
```

Когда подключаемый модуль со стилем `Bronze` подключен, его можно начинать использовать. В приложениях его можно применять, указывая его в коде. Например:

```
QApplication::setStyle("Bronze");
```

Мы также можем применять данный стиль, вовсе не изменяя исходный код приложения, для чего нужно просто запустить приложение с опцией `-style`. Например:

```
./spreadsheet -style bronze
```

Когда запускается *Qt Designer*, он автоматически ищет подключаемые модули. Если он обнаруживает стилиевой подключаемый модуль, он предоставляет возможность предварительного просмотра с использованием данного стиля через пункт меню `Form|Preview` (Форма|Предварительный просмотр).

Приложения, использующие подключаемые модули Qt, должны развертываться вместе с модулями, которые они предполагают использовать. Подключаемые модули Qt должны помещаться в специальные поддиректории (например, `plugins/styles` для пользовательских стилей). Приложения Qt производят поиск подключаемых модулей в директории `plugins` и в директории, где находится исполняемый файл приложения. Если мы хотим разместить подключаемые модули Qt в другой директории, то необходимо изменить путь поиска подключаемых модулей, для чего нужно вызвать функцию `QCoreApplication::addLibraryPath()` при запуске или установить переменную окружения `QT_PLUGIN_PATH` перед запуском приложения.

Итак, когда мы рассмотрели очень простой подключаемый модуль, мы разберем несколько более сложный: модуль формата изображения для файлов курсоров

Windows (.cur). (Формат показан на рис. 21.2.) В файлах курсоров Windows могут храниться несколько изображений одного курсора, находящегося в разных состояниях. После того, как подключаемый модуль для курсоров собран и установлен, Qt сможет читать файлы .cur и обращаться к отдельным курсорам (например, при помощи объектов QImage, QImageReader или QMovie), а также сможет записывать курсоры в любом другом графическом формате, поддерживаемом Qt, например, BMP, JPEG и PNG.

Новые классы-оболочки подключаемых модулей должны быть подклассом QImageIOPlugin и должны обеспечить реализацию нескольких виртуальных функций:

```
class CursorPlugin : public QImageIOPlugin
{
public:
    QStringList keys() const;
    Capabilities capabilities(QIODevice *device,
                             const QByteArray &format) const;
    QImageIOHandler *create(QIODevice *device,
                             const QByteArray &format) const;
};
```

Функция keys() возвращает список форматов изображений, которые поддерживает подключаемый модуль. Можно считать, что параметр format функций capabilities() и create() имеет значение из этого списка.

```
QStringList CursorPlugin::keys() const
{
    return QStringList() << "cur";
}
```

Наш подключаемый модуль поддерживает один формат изображений, поэтому возвращается список, содержащий только одно название. В идеале это название должно совпадать с расширением файла, используемым данным форматом. Если форматы имеют несколько расширений (например, .jpg и .jpeg для JPEG), мы можем возвращать список с несколькими элементами, относящимися к одному формату – по одному элементу на каждое расширение.

```
QImageIOPlugin::Capabilities
CursorPlugin::capabilities(QIODevice *device,
                           const QByteArray &format) const
{
    if (format == "cur")
        return CanRead;

    if (format.isEmpty()) {
        CursorHandler handler;
        handler.setDevice(device);
        if (handler.canRead())
            return CanRead;
    }

    return 0;
}
```

Функция `capabilities()` возвращает объект, который показывает, что может делать с данным форматом изображений обработчик изображений. Существует три возможных действия (`CanRead`, `CanWrite` и `CanReadIncremental`), а возвращаемое значение объединяет допустимые варианты поразрядной логической операцией ИЛИ.

Если формат «cur», наша реализация возвращает `CanRead`. Если формат не задан, мы создаем обработчик курсора и проверяем его способность чтения данных с заданного устройства. Функция `canRead()` только просматривает данные и проверяет возможность распознавания файла, не изменяя указатель файла. Возвращение 0 означает, что данный обработчик не может ни считывать, ни записывать файл.

```
QImageIOHandler *CursorPlugin::create(QIODevice *device,
                                       const QByteArray &format) const
{
    CursorHandler *handler = new CursorHandler;
    handler->setDevice(device);
    handler->setFormat(format);
    return handler;
}
```

Когда файл курсора открыт (например, с помощью класса `QImageReader`), будет вызвана функция оболочки подключаемого модуля `create()` с передачей указателя устройства и формата «cur». Мы создаем экземпляр `CursorHandler` для заданного устройства и формата. Вызывающая программа становится владельцем обработчика и удалит его, когда он не станет нужен. Если приходится считывать несколько файлов, для каждого из них создается новый обработчик.

```
Q_EXPORT_PLUGIN2(cursorplugin, CursorPlugin)
```

В конце файла `.cpp` мы используем макрос `Q_EXPORT_PLUGIN2()`, чтобы гарантировать распознавание в Qt подключаемого модуля. В первом параметре задается произвольное имя, используемое нами для подключаемого модуля. Второй параметр содержит имя класса подключаемого модуля.

Подкласс `QImageIOPlugin` создается достаточно просто. Реальная работа подключаемого модуля делается обработчиком. Обработчики форматов изображений должны создать подкласс `QImageIOHandler` и переопределить некоторые или все его открытые функции. Сначала рассмотрим заголовочный файл:

```
class CursorHandler : public QImageIOHandler
{
public:
    CursorHandler();

    bool canRead() const;
    bool read(QImage *image);
    bool jumpToNextImage();
    int currentImageNumber() const;
    int imageCount() const;

private:
    enum State { BeforeHeader, BeforeImage, AfterLastImage, Error };
```

```
void readHeaderIfNecessary() const;
QByteArray readBitmap(int width, int height, QDataStream &in) const;
void enterErrorState() const;
```

```
mutable State state;
mutable int currentImageNo;
mutable int numImages;
```

```
};
```

Открытые функции имеют фиксированную сигнатуру. Здесь нет некоторых функций, которые не надо переопределять в обработчике, обеспечивающем только чтение; в частности, отсутствует функция `write()`. Переменные-члены объявляются с ключевым словом `mutable`, потому что они изменяются внутри константных функций.

```
CursorHandler::CursorHandler()
{
    state = BeforeHeader;
    currentImageNo = 0;
    numImages = 0;
}
```

После создания обработчика мы сначала настраиваем его параметры. Номер текущего изображения курсора устанавливается на первый курсор, но поскольку переменная количества изображений `numImages` принимает значение 0, ясно, что у нас пока еще нет изображений.

```
bool CursorHandler::canRead() const
{
    if (state == BeforeHeader) {
        return device()->peek(4) == QByteArray("\0\0\2\0", 4);
    } else {
        return state != Error;
    }
}
```

Функция `canRead()` может вызываться в любой момент для определения возможности считывания обработчиком изображений дополнительных данных с устройства. Если функция вызывается до чтения данных в состоянии `BeforeHeader`, выполняется проверка конкретной метки, по которой опознаются файлы курсоров в Windows. Вызов `QIODevice::peek()` считывает первые четыре байта без изменения указателя файла на данном устройстве. Если функция `canRead()` вызывается позже, мы возвращаем `true` при отсутствии ошибки.

```
int CursorHandler::currentImageNumber() const
{
    return currentImageNo;
}
```

Эта простая функция возвращает номер курсора, на который позиционирован указатель файла устройства.

После создания обработчика пользователь может вызвать любую его открытую функцию, причем последовательность вызовов функций может быть произвольной. В этом кроется потенциальная проблема, поскольку необходимо исходить из того, что файл можно читать только последовательно, поэтому сначала надо один раз считать заголовок файла и затем выполнять какие-то другие действия. Эту проблему решаем путем вызова `readHeaderIfNecessary()` в тех функциях, для которых требуется предварительное считывание заголовка файла.

```
int CursorHandler::imageCount() const
{
    readHeaderIfNecessary();
    return numImages;
}
```

Эта функция возвращает количество изображений, содержащихся в файле. Для правильного файла, при чтении которого не возникает ошибок, она возвращает, по крайней мере, 1.

Следующая функция довольно сложная, поэтому мы рассмотрим ее по частям:

```
bool CursorHandler::read(QImage *image)
{
    readHeaderIfNecessary();

    if (state != BeforeImage)
        return false;
```

Функция `read()` считывает данные изображения, начинающегося в текущей позиции указателя устройства. Если успешно считан заголовок файла или указатель устройства после чтения изображения находится в начале другого изображения, можно считывать следующее изображение.

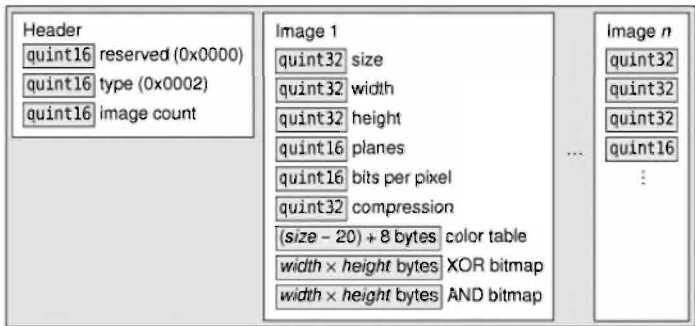


Рис. 21.2. Формат файла .cur

```
quint32 size;
quint32 width;
quint32 height;
quint16 numPlanes;
quint16 bitsPerPixel;
quint32 compression;
```

```

QDataStream in(device());
in.setByteOrder(QDataStream::LittleEndian);
in >> size;
if (size != 40) {
    enterErrorState();
    return false;
}
in >> width >> height >> numPlanes >> bitsPerPixel >> compression;
height /= 2;

if (numPlanes != 1 || bitsPerPixel != 1 || compression != 0) {
    enterErrorState();
    return false;
}

in.skipRawData((size - 20) + 8);

```

Мы создаем объект `QDataStream` для чтения устройства. Необходимо установить порядок байтов в соответствии с тем, который определен спецификацией формата файла `.cur`. Задавать версию потока `QDataStream` нет необходимости, поскольку форматы целых чисел и чисел с плавающей запятой не зависят от версии потока данных. Затем считываем элементы заголовка курсора и пропускаем неиспользуемые части заголовка и 8-байтовую таблицу цветов с помощью функции `QDataStream::skipRawData()`.

Необходимо учитывать все характерные особенности формата, например, уменьшать вдвое высоту изображения, потому что она в формате `.cur` в два раза превышает высоту реального изображения. Переменные `bitsPerPixel` и `compression` всегда имеют значения 1 и 0 в монохромных файлах `.cur`. При возникновении каких-либо проблем вызываем функцию `enterErrorState()` и возвращаем `false`.

```

QBitArray xorBitmap = readBitmap(width, height, in);
QBitArray andBitmap = readBitmap(width, height, in);

if (in.status() != QDataStream::Ok) {
    enterErrorState();
    return false;
}

```

Следующими элементами файла являются две битовые маски: одна XOR-маска, а другая AND-маска. Мы их считываем в массивы `QBitArray`, а не в `QBitmap`. Класс `QBitmap` предназначен для выполнения с ним операций рисования и вывода рисунка на экран, а нам нужен простой массив битов.

Завершив чтение файла, проверяем состояние потока `QDataStream`. Так можно поступать, потому что если `QDataStream` переходит в состояние ошибки, это состояние сохраняется в дальнейшем, и последующие операции чтения могут выдать только нули. Например, если чтение первого массива битов завершается неудачей, попытка чтения второго массива в результате даст пустой массив `QBitArray`.


```
*image = QImage(width, height, QImage::Format_ARGB32);

for (int i = 0; i < int(height); ++i) {
    for (int j = 0; j < int(width); ++j) {
        QRgb color;
        int bit = (i * width) + j;

        if (andBitmap.testBit(bit)) {
            if (xorBitmap.testBit(bit)) {
                color = 0x7F7F7F7F;
            } else {
                color = 0x00FFFFFF;
            }
        } else {
            if (xorBitmap.testBit(bit)) {
                color = 0xFFFFFFFF;
            } else {
                color = 0xFF000000;
            }
        }
        image->setPixel(j, i, color);
    }
}
```

Мы конструируем новый объект QImage с правильными размерами и присваиваем ему указатель *image. Затем проходим по каждому пикселю битовых массивов XOR и AND и преобразуем их в 32-битовый цветовой формат ARGB. С помощью массивов битов AND и XOR цвет каждого пикселя курсора всегда получается в соответствии со следующей таблицей:

AND	XOR	Результат
1	1	Инвертированный пиксель фона
1	0	Прозрачный пиксель
0	1	Белый пиксель
0	0	Черный пиксель

С получением черного, белого и прозрачного пикселя нет проблем, однако нельзя получить инвертированный пиксель фона, используя цветовой формат ARGB, если не знаешь цвет исходного пикселя фона. В качестве замены используем полупрозрачный серый цвет (0x7F7F7F7F).

```
++currentImageNo;
if (currentImageNo == numImages)
    state = AfterLastImage;
return true;
}
```

Завершив чтение изображения, мы обновляем текущий номер изображения и обновляем состояние, если прочитано последнее изображение. В этот момент устройство будет указывать на начало следующего изображения или на конец файла.

```
bool CursorHandler::jumpToNextImage()
{
    QImage image;
    return read(&image);
}
```

Функция `jumpToNextImage()` используется для пропуска изображения. Для простоты мы всего лишь вызываем `read()` и игнорируем полученный `QImage`. В более эффективной реализации использовалась бы информация, содержащаяся в заголовке файла `.cur`, для непосредственного смещения по файлу на соответствующее значение.

```
void CursorHandler::readHeaderIfNecessary() const
{
    if (state != BeforeHeader)
        return;

    quint16 reserved;
    quint16 type;
    quint16 count;

    QDataStream in(device());
    in.setByteOrder(QDataStream::LittleEndian);

    in >> reserved >> type >> count;
    in.skipRawData(16 * count);

    if (in.status() != QDataStream::Ok || reserved != 0
        || type != 2 || count == 0) {
        enterErrorState();
        return;
    }

    state = BeforeImage;
    currentImageNo = 0;
    numImages = int(count);
}
```

Закрывающая функция `readHeaderIfNecessary()` вызывается из `imageCount()` и `read()`. Если заголовок файла уже был прочитан, состояние не будет иметь значения `BeforeHeader` (перед заголовком), и сразу же делается возврат управления. В противном случае открываем на устройстве поток данных, считываем некоторые общие данные (в частности количество курсоров, содержащихся в файле) и устанавливаем состояние в значение `BeforeImage` (перед изображением). В конце указатель файла данного устройства устанавливается перед первым изображением.

```
void CursorHandler::enterErrorState() const
{
    state = Error;
    currentImageNo = 0;
    numImages = 0;
}
```

При возникновении ошибки считаем, что файл не содержит изображений требуемого формата, и устанавливаем состояние в значение `Error`. В дальнейшем такое состояние обработчика не может быть изменено.

```
QByteArray CursorHandler::readBitmap(int width, int height,
                                     QDataStream &in) const
{
    QByteArray bitmap(width * height);
    quint32 word;
    quint8 byte;

    for (int i = 0; i < height; ++i) {
        for (int j = 0; j < width; ++j) {
            if ((j % 32) == 0) {
                word = 0;
                for (int k = 0; k < 4; ++k) {
                    in >> byte;
                    word = (word << 8) | byte;
                }

                bitmap.setBit(((height - i - 1) * width) + j,
                             word & 0x80000000);

                word <<= 1;
            }
        }
    }
    return bitmap;
}
```

Функция `readBitmap()` используется для чтения масок курсора AND и XOR. Эти маски обладают двумя необычными свойствами. Во-первых, строки в них располагаются, начиная с нижних, вместо обычного расположения строк сверху вниз. Во-вторых, оказывается, что используемый здесь порядок байтов отличается от порядка байтов любых других данных в файлах `.cur`. В связи с этим нам приходится инвертировать координату `y` в вызове `setBit()` и считывать маски побайтно, сдвигая биты и используя маску для получения правильных значений.

Для построения подключаемого модуля мы должны использовать файл `.pro` очень сходный с тем, который мы использовали для стилевого подключаемого модуля стиля `Bronze`, который рассматривался ранее.

```
TEMPLATE      = lib
CONFIG        += plugin
HEADERS        = cursorhandler.h \
                 cursorplugin.h
SOURCES        = cursorhandler.cpp \
                 cursorplugin.cpp
DESTDIR        = $$[QT_INSTALL_PLUGINS]/imageformats
```

На этом завершается создание подключаемого модуля для курсоров Windows. Модули для других форматов изображений должны использовать сходную схему, хотя в некоторых случаях можно реализовать больше возможностей программного интерфейса `QImageIOHandler` и, в частности, функции, используемые для рисования изображений. В подключаемых модулях других типов используется схема оболочки подключаемого модуля, которая экспортирует один или несколько обработчиков, предоставляющих лежащие в их основе функции.

Как обеспечить в приложении возможность подключения модулей

Подключаемый к приложению модуль является динамической библиотекой, которая реализует какой-нибудь один или несколько интерфейсов. Интерфейс – это класс, содержащий только чисто виртуальные функции. Связь между приложением и подключаемыми модулями осуществляется через виртуальную таблицу интерфейса. В этом разделе мы основное внимание уделим способам взаимодействия приложения Qt с подключаемым модулем через его интерфейсы, а в следующем разделе покажем, как можно реализовать подключаемый модуль.

Чтобы продемонстрировать конкретный пример, создадим простое приложение `Text Art` (искусство отображения текста), показанное на рис. 21.3. Специальные эффекты отображения текста обеспечиваются подключаемыми модулями; приложение получает список текстовых эффектов, создаваемых каждым подключаемым модулем, и проходит в цикле по этому списку, показывая результат каждого эффекта в соответствующем элементе списка `QListWidget`.

В приложении `Text Art` определяется один интерфейс:

```
class TextArtInterface
{
public:
    virtual ~TextArtInterface() { }

    virtual QStringList effects() const = 0;
    virtual QPixmap applyEffect(const QString &effect,
                               const QString &text,
                               const QFont &font, const QSize &size,
                               const QPen &pen,
                               const QBrush &brush) = 0;
};
```

```
Q_DECLARE_INTERFACE(TextArtInterface,
                    "com.software-inc.TextArt.TextArtInterface/1.0")
```

В классе интерфейса обычно объявляется виртуальный деструктор, виртуальная функция, возвращающая список `QStringList`, и одна или несколько других виртуальных функций. Деструктор объявляется, прежде всего, для того, чтобы компилятор не жаловался на отсутствие виртуального деструктора в классе, который имеет виртуальные функции. В данном примере функция `effects()` возвращает список текстовых эффектов, которые могут создаваться подключаемым

модулем. Этот список можно рассматривать как список ключей. При каждом вызове одной из функций мы передаем эти ключи в качестве первого аргумента, позволяя реализовать в одном подключаемом модуле несколько эффектов.

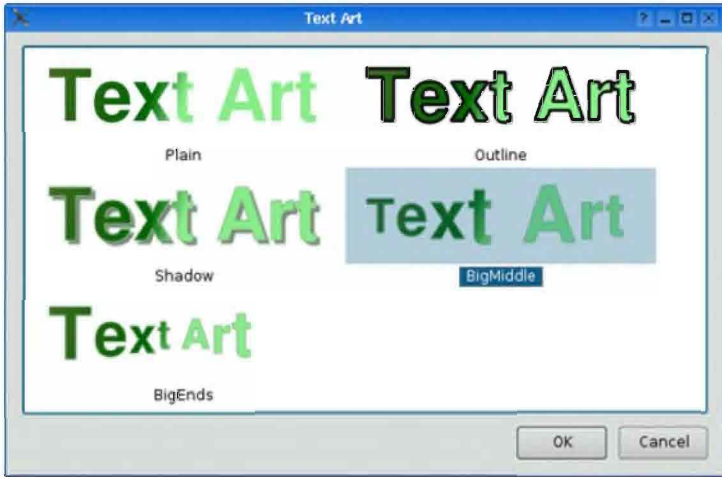


Рис. 21.3. Приложение Text Art

В конце мы используем макрос `Q_DECLARE_INTERFACE()` для назначения некоторого идентификатора интерфейсу. Этот идентификатор обычно имеет четыре компонента: инвертированное имя домена, определяющее создателя интерфейса, имя приложения, имя интерфейса и номер версии. При любом изменении интерфейса (например, при добавлении новой виртуальной функции или при изменении сигнатуры существующей функции) мы должны не забыть увеличить номер версии; в противном случае приложение может завершиться аварийно при попытке получения доступа к старой версии подключаемого модуля.

Это приложение реализуется в виде класса `TextArtDialog`. Мы будем показывать только тот программный код, который связан с применением подключаемых модулей. Давайте начнем с конструктора:

```
TextArtDialog::TextArtDialog(const QString &text, QWidget *parent)
    : QDialog(parent)
{
    listWidget = new QListWidget;
    listWidget->setViewMode(QListWidget::IconMode);
    listWidget->setMovement(QListWidget::Static);
    listWidget->setIconSize(QSize(260, 80));
    ...
    loadPlugins();
    populateListWidget(text);
    ...
}
```

Конструктор создает виджет `QListWidget`, содержащий список доступных эффектов. Он вызывает закрытую функцию `loadPlugins()` для поиска и загрузки всех подключаемых модулей, реализующих интерфейс `TextArtInterface`, и заполняет список виджетов с помощью вызова другой закрытой функции, `populateListWidget()`.

```
void TextArtDialog::loadPlugins()
{
    QDir pluginsDir = directoryOf("plugins")

    foreach (QString fileName, pluginsDir.entryList(QDir::Files)) {
        QPluginLoader loader(pluginsDir.absoluteFilePath(fileName));
        if (TextArtInterface *interface =
            qobject_cast<TextArtInterface *>(loader.instance()))
            interfaces.append(interface);
    }
}
```

В функции `loadPlugins()` мы начинаем с извлечения директории `plugins` приложения. Затем мы пытаемся загрузить все файлы, находящиеся в каталоге приложения `plugins`. Функция `directoryOf()` аналогична той, которую мы использовали в главе 17.

Если файл, который мы пытаемся загрузить, является подключаемым модулем Qt и имеет ту же самую версию Qt, какую имеет приложение, функция `QPluginLoader::instance()` возвратит указатель `QObject *`, ссылающийся на подключаемый модуль Qt. Используем `qobject_cast<T>()` для проверки реализации в подключаемом модуле интерфейса `TextArtInterface`. При всяком успешном приведении типа мы добавляем интерфейс к списку интерфейсов приложения `TextArtDialog` (который имеет тип `QList<TextArtInterface *>`).

Для некоторых приложений может потребоваться загрузка двух или более различных интерфейсов, и в этом случае программный код по получению этих интерфейсов мог бы выглядеть следующим образом:

```
QObject *plugin = loader.instance();
if (TextArtInterface *i = qobject_cast<TextArtInterface *>(plugin))
    textArtInterfaces.append(i);
if (BorderArtInterface *i = qobject_cast<BorderArtInterface *>(plugin))
    borderArtInterfaces.append(i);

if (TextureInterface *i = qobject_cast<TextureInterface *>(plugin))
    textureInterfaces.append(i);
```

Тип одного подключаемого модуля может успешно приводиться к нескольким указателям интерфейсов, если он использует множественное наследование.

```
void TextArtDialog::populateListWidget(const QString &text)
{
    QFont font("Helvetica", iconSize.height(), QFont::Bold);
    QSize iconSize = listWidget->iconSize();
    QPen pen(QColor("darkseagreen"));
```

```

QLinearGradient gradient(0, 0, iconSize.width() / 2,
                        iconSize.height() / 2);
gradient.setColorAt(0.0, QColor("darkolivegreen"));
gradient.setColorAt(0.8, QColor("darkgreen"));
gradient.setColorAt(1.0, QColor("lightgreen"));

foreach (TextArtInterface *interface, interfaces) {
    foreach (QString effect, interface->effects()) {
        QListWidgetItem *item = new QListWidgetItem(effect,
                                                    listWidget);
        QPixmap pixmap = interface->applyEffect(effect, text, font,
                                                iconSize, pen,
                                                gradient);
        item->setData(Qt::DecorationRole, pixmap);
    }
}
listWidget->setCurrentRow(0);
}

```

Функция `populateListWidget()` начинается с создания некоторых переменных, передаваемых функции `applyEffect()`, в частности, шрифта, пера и линейного градиента. Затем она просматривает в цикле все интерфейсы `TextArtInterface`, найденные функцией `loadPlugins()`. Для любого эффекта, обеспечиваемого каждым интерфейсом, создается новый элемент `QListWidgetItem`, текст которого определяет название создаваемого им эффекта, и создается `QPixmap`, с использованием функции `applyEffect()`.

В данном разделе мы увидели, как можно загружать подключаемые модули, вызывая в конструкторе функцию `loadPlugins()`, и как можно их использовать в функции `populateListWidget()`. Программный код элегантно обрабатывает ситуации, когда подключаемые модули вообще не обеспечивают интерфейс `TextArtInterface`, или когда только один из них или несколько обеспечивают такой интерфейс. Более того, другие подключаемые модули могут добавляться позже. При каждом запуске приложения производится загрузка всех подключаемых модулей, имеющих нужный интерфейс. Это позволяет легко расширять функциональность приложения без изменения самого приложения.

Написание подключаемых к приложению модулей

Подключаемый к приложению модуль является подклассом `QObject` и интерфейсов, которые он собирается обеспечить. Прилагаемые к этой книге примеры содержат два подключаемых модуля, предназначенных для приложения `Text Art`, представленного в предыдущем разделе, и показывающих, что это приложение правильно работает с несколькими подключаемыми модулями.

Здесь мы рассмотрим программный код только одного из них – `Basic Effects` (модуль основных эффектов). Предполагаем, что исходный код подключаемого модуля находится в каталоге `basiceffectsplugin` и что приложение `Text Art` находится в параллельном каталоге `textart`. Ниже приводится объявление класса подключаемого модуля:

```

class BasicEffectsPlugin : public QObject, public TextArtInterface
{
    Q_OBJECT
    Q_INTERFACES(TextArtInterface)

public:
    QStringList effects() const;
    QPixmap applyEffect(const QString &effect, const QString &text,
                        const QFont &font, const QSize &size,
                        const QPen &pen, const QBrush &brush);
};

```

Этот подключаемый модуль реализует только один интерфейс, `TextArtInterface`. Кроме `Q_OBJECT` необходимо использовать макрос `Q_INTERFACES()` для каждого интерфейса, для которого создается подкласс, чтобы обеспечить безболезненное восприятие компилятором `moc` оператора приведения типа `qobject_cast<T>()`.

```

QStringList BasicEffectsPlugin::effects() const
{
    return QStringList() << "Plain" << "Outline" << "Shadow";
}

```

Функция `effects()` возвращает список текстовых эффектов, поддерживаемых подключаемым модулем. Этот подключаемый модуль обеспечивает три эффекта, поэтому возвращаем список, содержащий имена каждого из них.

Функция `applyEffect()` обеспечивает функциональность подключаемого модуля и слегка запутанна, поэтому рассмотрим ее по частям:

```

QPixmap BasicEffectsPlugin::applyEffect(const QString &effect,
    const QString &text, const QFont &font, const QSize &size,
    const QPen &pen, const QBrush &brush)
{
    QFont myFont = font;
    QFontMetrics metrics(myFont);
    while ((metrics.width(text) > size.width()
        || metrics.height() > size.height())
        && myFont.pointSize() > 9) {
        myFont.setPointSize(myFont.pointSize() - 1);
        metrics = QFontMetrics(myFont);
    }
}

```

Мы хотим обеспечить по мере возможности достаточность указанного размера для размещения заданного текста. По этой причине используем метрики шрифта и, если текст оказывается слишком большим, входим в цикл, где уменьшаем размер, пока он не окажется подходящим или не достигнет 9 точек, что соответствует нашему минимальному размеру.

```

QPixmap pixmap(size);

```

```

QPainter painter(&pixmap);
painter.setFont(myFont);

```



```

painter.setPen(pen);
painter.setBrush(brush);
painter.setRenderHint(QPainter::Antialiasing, true);
painter.setRenderHint(QPainter::TextAntialiasing, true);
painter.setRenderHint(QPainter::SmoothPixmapTransform, true);
painter.eraseRect(pixmap.rect());

```

Мы создаем пиксельную карту требуемого размера и рисовальщик для рисования на пиксельной карте. Также устанавливаем некоторые особенности воспроизведения, чтобы обеспечить максимальное сглаживание при выводе текста. Вызов функции `eraseRect()` очищает пиксельную карту, заполняя ее цветом фона.

```

if (effect == "Plain") {
    painter.setPen(Qt::NoPen);
} else if (effect == "Outline") {
    QPen pen(Qt::black);
    pen.setWidthF(2.5);
    painter.setPen(pen);
} else if (effect == "Shadow") {
    QPainterPath path;
    painter.setBrush(Qt::darkGray);
    path.addText(((size.width() - metrics.width(text)) / 2) + 3,
                (size.height() - metrics.descent()) + 3, myFont,
                text);
    painter.drawPath(path);
    painter.setBrush(brush);
}

```

Для эффекта «Plain» (простой) не требуется никакой рамки. Для эффекта «Outline» (рамка) игнорируем исходное перо и создаем наше собственное перо шириной в 2.5 пикселя. Для эффекта «Shadow» (тень) сначала рисуется тень, чтобы можно было выводить текст поверх ее.

```

QPainterPath path;
path.addText((size.width() - metrics.width(text)) / 2,
            size.height() - metrics.descent(), myFont, text);
painter.drawPath(path);
return pixmap;
}

```

Теперь у нас имеются перо и кисти, соответствующим образом установленные для каждого текстового эффекта, а для эффекта «Shadow» мы нарисовали тень. После этого мы готовы воспроизвести текст. Текст центрируется по горизонтали и выводится достаточно далеко от нижнего края пиксельной карты, чтобы оставить достаточно места для размещения нижних выносных элементов.

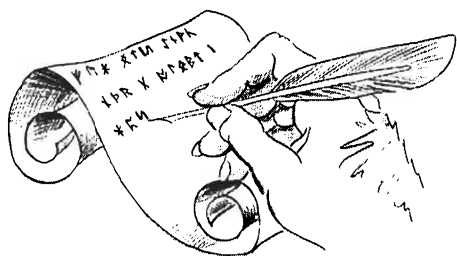
```
Q_EXPORT_PLUGIN2(basiceffectsplugin, BasicEffectsPlugin)
```

В конце файла `.cpp` используем макрос `Q_EXPORT_PLUGIN2()`, чтобы этот подключаемый элемент был доступен для Qt.

Файл `.pro` аналогичен файлу, который мы использовали ранее в данной главе для подключаемого модуля стиля **Bronze**:

```
TEMPLATE      = lib
CONFIG        += plugin
HEADERS       = ../textart/textartinterface.h \
               basiceffectsplugin.h
SOURCES       = basiceffectsplugin.cpp
DESTDIR       = ../textart/plugins
```

Если данная глава повысила ваш интерес к подключаемым к приложению модулям, вы можете изучить имеющийся в Qt более сложный пример **Plug & Paint** (подключи и рисуй). Приложение поддерживает три различных интерфейса и включает в себя полезное диалоговое окно **Plugin Information** (информацию о подключаемых модулях), которое содержит списки подключаемых модулей и интерфейсов, доступных в приложении.



- *Общий обзор языка скриптов ECMAScript*
- *Расширение приложений Qt при помощи скриптов*
- *Реализация расширений графического интерфейса с помощью скриптов.*
- *Автоматическое выполнение задач с применением скриптов*

Глава 22. Создание прикладных скриптов

Скрипты – это программы, написанные на интерпретируемом языке и расширяющие возможности существующей системы. Некоторые скрипты запускаются как самостоятельные приложения, тогда как другие выполняются, будучи встроенными в другие приложения. Начиная с версии 4.3, Qt включает в себя модуль *QtScript*, который можно использовать для того, чтобы в приложениях Qt можно было писать скрипты на ECMAScript – стандартизированной версии JavaScript. Этот модуль является полностью переработанной версией более раннего продукта Trolltech – Qt Script for Applications (QSA), он обеспечивает полную поддержку ECMAScript Edition 3.

ECMAScript – это официальное название языка, стандартизованного организацией ECMA International. Он формирует основу JavaScript (Mozilla), JScript (Microsoft) и ActionScript (Adobe). Хотя синтаксис языка на первый взгляд похож на C++ и Java, базовые концепции радикально отличаются и стоят отдельно от большинства других объектно-ориентированных языков программирования. В первом разделе данной главы мы коротко рассмотрим язык ECMAScript и покажем, как запускать код ECMAScript в Qt. Если вы уже знаете JavaScript или любой другой язык, основанный на ECMAScript, вы, вероятно, можете просто просмотреть этот раздел по диагонали.

Во втором разделе мы увидим, как добавить поддержку скриптов в приложение Qt. Это даст возможность пользователям добавлять собственную функциональность к той, которую уже предоставляет приложение. Данный подход также часто используется для упрощения поддержки. Персонал службы технической поддержки может предоставлять исправления ошибок и пути их обхода в форме скриптов.

В третьем разделе мы покажем, как разработать элементы клиентского графического интерфейса, сочетая код ECMAScript и формы, созданные в *Qt Designer*. Этот метод привлекателен для тех разработчиков, которым не нравятся циклы «компиляция, сборка, запуск», характерные для разработки на C++,

и которые предпочитают использовать скрипты. Этот метод также позволяет пользователям, имеющим опыт работы с JavaScript, разрабатывать полностью функциональные графические интерфейсы без необходимости изучать C++.

Наконец, в последнем разделе мы покажем, как разрабатывать скрипты, использующие компоненты C++ в ходе своей работы. Это можно сделать с любым произвольным компонентом C++ и C++/Qt, при разработке которого вполне могла не учитываться возможность создания скриптов. Данный подход особенно полезен в случае, когда нужно создать несколько программ с использованием одних базовых компонентов, или если мы хотим сделать функциональность C++ доступной для программистов не на C++.

Общий обзор языка ECMAScript

В этом разделе представлено краткое введение в язык ECMAScript, чтобы вы могли понимать фрагменты кода, приведенные в остальных частях главы, и могли начать писать свои собственные скрипты. На web-сайте Mozilla Foundation содержится более полное руководство (страница http://developer.mozilla.org/en/docs/Core_JavaScript_1.5_Guide), а также в качестве руководства и справочника рекомендуется использовать книгу «JavaScript: The Definitive Guide», автор David Flanagan (O'Reilly, 2006). Официальную спецификацию ECMAScript можно найти по адресу <http://www.ecmascriptinternational.org/publications/standards/Ecma-262.htm>.

Основные структуры управления ECMAScript – операторы if, циклы for и while – аналогичны тем, которые используются в C++ и Java. В ECMAScript также предлагаются более или менее сходные операторы присваивания, сравнения и арифметические операторы. В ECMAScript поддерживается конкатенация строк оператором + и присоединение оператором +=.

Чтобы получить представление о синтаксисе ECMAScript, мы начнем изучение следующей программы, которая выводит список всех простых чисел меньших 1000.

```
const MAX = 1000;
var isPrime = new Array(MAX);
for (var i = 2; i < MAX; ++i)
    isPrime[i] = true;
for (var i = 2; i < MAX; ++i) {
    if (isPrime[i]) {
        for (var j = i; i * j < MAX; ++j)
            isPrime[i * j] = false;
    }
}
for (var i = 2; i < MAX; ++i) {
    if (isPrime[i])
        print(i);
}
```

С точки зрения программиста C++ наиболее удивительным, вероятно, является то, что тип переменных не указывается явно. Все, что требуется для объявления переменной – это ключевое слово var. Переменные, доступные только для чтения, объявляются при помощи ключевого const, вместо var. Другим заслужи-

вающим вниманием свойством приведенной программы является отсутствие функции `main()`. В этой ситуации код, расположенный за пределами всех функций, выполняется немедленно, начиная от начала файла и в направлении его конца.

В отличие от C++, наличие точки с запятой в конце операторов ECMAScript, как правило, является необязательным. Используя сложные правила, интерпретатор может вставить большинство пропущенных точек с запятой самостоятельно. Несмотря на это, вводить точки с запятой рекомендуется, чтобы избежать неприятных сюрпризов.

Для запуска приведенной выше программы мы можем использовать интерпретатор `qscript`, расположенный в директории `Qt examples/script/qscript`. Если вызвать интерпретатор, указав в качестве аргумента командной строки имя файла, файл рассматривается как код ECMAScript и выполняется, в противном случае запускается интерактивный сеанс работы.

Если, объявляя переменную ключевым словом `var`, мы не указали исходное значение, по умолчанию используется `undefined` — специально значение типа `Undefined`. Позже мы можем присвоить переменной любое значение любого типа, используя оператор присваивания (`=`). Рассмотрим следующие примеры:

```
var x;
typeof x;    // возвращает "undefined"
x = null;
typeof x;    // возвращает "null"
x = true;
typeof x;    // возвращает "boolean"
x = 5;
typeof x;    // возвращает "number"
x = "Hello";
typeof x;    // возвращает "string"
```

Оператор `typeof` возвращает строковое (в нижнем регистре) представление для типа данных, который связан со значением, хранящимся в переменной. В ECMAScript определены пять элементарных типов данных: `Undefined`, `Null`, `Boolean`, `Number` и `String`. Типы `Undefined` и `Null` — это специальные типы для констант `undefined` и `null`, соответственно. Тип `Boolean` состоит из двух значений: `true` и `false`. Тип `Number` хранит числа с плавающей точкой. Тип `String` хранит строки формата Unicode.

Переменные также могут хранить объекты и функции, которым соответствуют типы данных `Object` и `Function`. Например:

```
x = new Array(10);
typeof x;    // возвращает "object"
x = print;
typeof x;    // возвращает "function"
```

Как и Java, ECMAScript различает элементарные типы и объектные типы. Элементарные типы функционируют подобно типам значений C++, таким как `int` или `QString`. Они создаются без использования оператора `new` и копируются по значению. Напротив, объектные типы должны создаваться с использованием оператора `new`, и переменные этого типа хранят только ссылку (указатель) на данный

объект. При выделении объектов оператором `new` нам не нужно беспокоиться об освобождении памяти, поскольку сборщик мусора делает это автоматически.

Если мы присвоим значение переменной, не объявив ее предварительно с ключевым словом `var`, переменная будет создана как глобальная переменная. А если мы попытаемся прочитать значение несуществующей переменной, мы получим исключение `ReferenceError`. Мы можем перехватить данное исключение с помощью инструкции `try...catch`, как показано ниже:

```
try {
    print(y);
} catch (e) {
    print(e.name + ": " + e.message);
}
```

Если переменная `y` не существует, на консоль выводится сообщение «`ReferenceError: y is not defined`».

Если неопределенные переменные способны вызвать неразбериху в наших программах, то же относится к переменным, которые определены, но содержат константу `undefined`, т. е. значение по умолчанию, если при объявлении переменной с ключевым словом `var` не указано инициализирующее значение. Чтобы протестировать значение `undefined`, мы можем использовать операторы строгого сравнения `===` или `!==`. Пример:

```
var x;
...
var y = 0;
if (x !== undefined)
    y = x;
```

Знакомы нам операторы сравнения `==` и `!=` также присутствуют в ECMAScript, но в отличие от `===` или `!==` они иногда возвращают `true`, если сравниваемые значения имеют разные типы. Например, операции `24 == «24»` и `null == undefined` возвращают `true`, а `24 === «24»` и `null === undefined` возвращают `false`.

Теперь мы рассмотрим более сложную программу, иллюстрирующую определение наших собственных функций в ECMAScript:

```
function square(x)
{
    return x * x;
}
function sumOfSquares(array)
{
    var result = 0;
    for (var i = 0; i < array.length; ++i)
        result += square(array[i]);
    return result;
}
var array = new Array(100);
for (var i = 0; i < array.length; ++i)
    array[i] = (i * 257) % 101;
print(sumOfSquares(array));
```

Функции определяются с помощью ключевого слова `function`. В соответствии с динамической природой ECMAScript, параметры объявляются без указания типа, и функция не имеет явного возвращаемого значения.

Глядя на код, мы можем предположить, что функция `square()` должна вызываться со значением `Number`, а функцию `sumOfSquares()` нужно вызывать, передавая ей объект `Array`, но это не является обязательным. Например, `square(«7»)` возвратит 49, поскольку оператор умножения ECMAScript преобразует строки в числа в числовом контексте. Точно также, функция `sumOfSquares()` будет работать не только для объектов `Array`, но и для других объектов, имеющих сходный интерфейс.

Как правило, в ECMAScript применяется принцип «утки»: «Если это ходит как утка, и крикает как утка, то это должна быть утка». Это весьма отличается от строгой типизации, принятой с C++ и Java, где нужно объявлять типы параметров, а аргументы должны совпадать с объявленными типами.

В предыдущем примере в функции `sumOfSquares()` было жестко прописано применение функции `square()` к каждому элементу массива. Мы можем сделать эту функцию более гибкой, позволив ей принимать унарную функцию в качестве второго аргумента, и переименовав ее в `sum()`.

```
function sum(array, unaryFunc)
{
    var result = 0;
    for (var i = 0; i < array.length; ++i)
        result += unaryFunc(array[i]);
    return result;
}
var array = new Array(100);
for (var i = 0; i < array.length; ++i)
    array[i] = (i * 257) % 101;
print(sum(array, square));
```

Вызов `sum(array, square)` эквивалентен `sumOfSquares(array)`. Вместо определения функции `square()` мы также можем передать в `sum()` анонимную функцию:

```
print(sum(array, function(x) { return x * x; }));
```

А вместо определения переменной `array` мы можем передать массив-литерал:

```
print(sum([4, 8, 11, 15], function(x) { return x * x; }));
```

ECMAScript позволяет передать в функцию больше параметров, чем было объявлено. Дополнительные аргументы доступны через массив `arguments`. Рассмотрим следующий пример:

```
function sum(unaryFunc)
{
    var result = 0;
    for (var i = 1; i < arguments.length; ++i)
        result += unaryFunc(arguments[i]);
    return result;
}
print(sum(square, 1, 2, 3, 4, 5, 6));
```

Здесь определено, что функция `sum()` принимает переменное число аргументов. Первый аргумент – это функция, которую мы хотим применить. Другие аргументы представляют собой числа, которые мы хотим суммировать. При переборе массива `arguments` мы должны пропустить элемент, соответствующий указателю 0, поскольку он соответствует `unaryFunc`, т. е. унарной функции. Мы также можем опустить параметр `unaryFunc` в списке параметров и извлечь его из массива `arguments`:

```
function sum()
{
    var unaryFunc = arguments[0];
    var result = 0;
    for (var i = 1; i < arguments.length; ++i)
        result += unaryFunc(arguments[i]);
    return result;
}
```

Массив `arguments` может использоваться для изменения поведения функции в зависимости от аргументов или их числа. Например, предположим, что мы хотим позволить функции `sum()` принимать необязательный унарный аргумент, а затем список чисел. Это позволит нам вызывать эту функцию следующим образом:

```
print(sum(1, 2, 3, 4, 5, 6));
print(sum(square, 1, 2, 3, 4, 5, 6));
```

Вот как нам следует реализовать функцию `sum()` для поддержки такого вызова:

```
function sum()
{
    var unaryFunc = function(x) { return x; };
    var i = 0;
    if (typeof arguments[0] == "function") {
        unaryFunc = arguments[0];
        i = 1;
    }
    var result = 0;
    while (i < arguments.length)
        result += unaryFunc(arguments[i++]);
    return result;
}
```

Если мы укажем функцию в качестве первого аргумента, эта функция будет применяться к каждому числу перед его добавлением к сумме. В противном случае мы будем использовать функцию тождества `function(x) { return x; }`.

Для программистов C++ наиболее сложным аспектом ECMAScript, вероятно, является его объектная модель. ECMAScript – это основанный на объектах объектно-ориентированный язык, что отличает его от C++, C#, Java, Simula и Smalltalk, которые основываются на классах. Вместо концепции классов в ECMAScript предлагается более низкоуровневый механизм, позволяющий добиться тех же результатов.

Первый механизм, позволяющий нам реализовать классы в ECMAScript – это механизм конструктора. Конструктор – это функция, которая может быть вызвана с помощью оператора `new`. Например, вот конструктор объекта `Shape`:

```
function Shape(x, y) {  
    this.x = x;  
    this.y = y;  
}
```

Конструктор `Shape` имеет два параметра и инициализирует свойства `x` и `y` (переменные-члены) нового объекта на основе значений, переданных конструктору. Ключевое слово `this` указывает на создаваемый объект. В ECMAScript объект, по сути, представляет собой коллекцию свойств. Свойства можно добавлять, удалять и изменять в любое время. Свойство создается, как только оно задается в первый раз, так что когда мы присваиваем значения переменным `this.x` и `this.y` в конструкторе, в результате создаются свойства `x` и `y`.

Распространенная ошибка разработчиков на C++ и Java – пропуск ключевого слова `this` при доступе к свойствам объекта. В предыдущем примере это маловероятно, поскольку выражение `x = x` выглядело бы подозрительно, но в других случаях это привело бы к созданию побочных глобальных переменных.

Чтобы создать экземпляр объекта `Shape`, мы используем оператор `new`, как показано ниже:

```
var shape = new Shape(10, 20);
```

Если мы применяем к переменной `shape` оператор `typeof()`, мы получаем тип данных `Object`, а не `Shape`. Если мы хотим определить, был ли создан объект конструктором `Shape`, мы можем использовать оператор `instanceof`:

```
var array = new Array(100);  
array instanceof Shape; // возвращает false  
var shape = new Shape(10, 20);  
shape instanceof Shape; // возвращает true
```

ECMAScript позволяет использовать как конструктор любую функцию. Однако если функция не выполняет никаких изменений с объектом `this`, вызывать ее как конструктор смысла не имеет. И, напротив, конструктор можно вызвать, как обычную функцию, но это, также, редко имеет смысл.

Наряду с элементарными типами данных, ECMAScript предоставляет встроенные конструкторы, позволяющие создавать экземпляры основных объектных типов, а именно, `Array`, `Date` и `RegExp`. Другие конструкторы, соответствующие элементарным типам, позволяют нам создавать объекты, хранящие элементарные значения. Функция-член `valueOf()` позволяет извлечь элементарное значение, хранящееся в объекте. Например:

```
var boolObj = new Boolean(true);  
typeof boolObj; // возвращает "object"  
var boolVal = boolObj.valueOf();  
typeof boolVal; // возвращает "boolean"
```

На рис. 22.1 перечислены встроенные глобальные константы, функции и объекты, предоставляемые в ECMAScript. В следующем разделе мы увидим,

как можно дополнять эту встроенную функциональность другими, специфичными для приложения компонентами, написанными на C++. Мы видели, как определить конструктор в ECMAScript и как добавлять переменные-члены в созданный объект. Обычно нам также бывает нужно определять функции-члены. Поскольку в ECMAScript функции считаются привилегированными элементами, их добавление оказывается задачей на удивление простой. Вот новая версия конструктора Shape, на этот раз содержащая две функции-члена: `manhattanPos()` и `translate()`:

```
function Shape(x, y) {
    this.x = x;
    this.y = y;
    this.manhattanPos = function() {
        return Math.abs(this.x) + Math.abs(this.y);
    };
    this.translate = function(dx, dy) {
        this.x += dx;
        this.y += dy;
    };
}
```

Мы можем вызывать функции-члены с помощью оператора-точки (.):

```
var shape = new Shape(10, 20);
shape.translate(100, 100);
print(shape.x + ", " + shape.y + " (" + shape.manhattanPos() + ")");
```

При использовании данного подхода каждый экземпляр Shape имеет свои собственные свойства: `manhattanPos()` и `translate()`. Поскольку эти свойства должны быть идентичны для всех экземпляров Shape, желательно сохранить их только один раз, а не делать это для каждого экземпляра. ECMAScript позволяет осуществить это при помощи прототипа. *Прототип* – это объект, служащий резервом для всех прочих объектов, и предоставляющий им исходный набор свойств. Одно из преимуществ такого подхода состоит в том, что существует возможность изменить объект-прототип в любое время, и изменения немедленно затронут все объекты, созданные с использованием данного прототипа.

Рассмотрим следующий пример:

```
function Shape(x, y) {
    this.x = x;
    this.y = y;
}
Shape.prototype.manhattanPos = function() {
    return Math.abs(this.x) + Math.abs(this.y);
};
Shape.prototype.translate = function(dx, dy) {
    this.x += dx;
    this.y += dy;
};
```

Константы	
NaN	Значение Not-a-Number (NaN) (не число) согласно IEEE 754
Infinity	Положительная бесконечность (+∞)
undefined	Значение по умолчанию для неинициализированных переменных

Функции	
print(x) ¹	Вывод значения на консоль
eval(str)	Выполнение программы ECMAScript
parseInt(str, base)	Преобразование строки в целочисленное значение
parseFloat(str)	Преобразование строки в значение с плавающей точкой
isNaN(n)	Возвращает true, если n = NaN
isFinite(n)	Возвращает true, если n – это число, отличное от NaN, +∞ и -∞
decodeURI(str)	Преобразование URI с 8-битовой кодировкой в Unicode
decodeURIComponent(str)	Преобразование компонента URI с 8-битовой кодировкой в Unicode
encodeURI(str)	Преобразование Unicode URI в URI с 8-битовой кодировкой
encodeURIComponent(str)	Преобразование компонента Unicode URI в URI с 8-битовой кодировкой

Классы (Конструкторы)	
Object	Предоставляет функциональность, общую для всех объектов
Function	Инкапсулирует функцию ECMAScript
Array	Одномерный массив элементов с изменяемым размером
String	Хранит строку Unicode
Boolean	Хранит булево значение (true или false)
Number	Хранит значение с плавающей точкой
Date	Хранит дату и время
RegExp	Обеспечивает сопоставление регулярных выражений
Error	Базовый тип для типов, связанных с ошибками
EvalError	Генерируется при неверном использовании eval()
RangeError	Генерируется, если числовое значение выходит за допустимый диапазон
ReferenceError	Генерируется при попытке доступа к неопределенной переменной
SyntaxError	Генерируется, если функция eval() обнаруживает синтаксическую ошибку
TypeError	Генерируется при неверном типе аргумента
URIError	Генерируется в случае ошибки при обработке URI

Объект	
Math	Предоставляет математические константы и функции

Рис. 22.1. Встроенные свойства глобального объекта

¹ Не указана в стандарте ECMAScript.

В этой версии функции Shape мы создаем свойства `manhattanPos` и `translate` за пределами конструктора, как свойства объекта `Shape.prototype`. Когда мы создаем экземпляр Shape, новый объект сохраняет внутренний указатель на объект `Shape.prototype`. Когда мы извлекаем значение свойства, отсутствующего в объекте Shape, свойство также ищется в прототипе. Таким образом, прототип объекта Shape – идеальное место для хранения функций-членов, которые должны быть общими для всех экземпляров Shape.

Может быть весьма соблазнительным поместить в прототип все виды свойств, которые мы хотим сделать общими для всех экземпляров Shape, подобно статическим переменным-членам C++ или переменным класса Java. Такой синтаксис работает для свойств, доступных только для чтения (в том числе, для функций-членов), поскольку прототип играет роль резервного хранилища при извлечении значения свойства. Однако он не работает, как ожидается, если мы попытаемся присвоить новое значение общей переменной. Вместо этого прямо в объекте Shape создается новая переменная, формируя свойство с тем же именем, что и в прототипе. Такая асимметрия между доступом для чтения и для записи к переменной часто является источником путаницы для программистов-новичков в ECMAScript.

В языках, основанных на классах, таких как C++ и Java, мы можем использовать наследование для создания специализированных типов объектов. Например, мы могли бы определить класс Shape (фигура), а затем создать на его основе классы Triangle (треугольник), Square (квадрат) и Circle (окружность). В ECMAScript сходного эффекта можно добиться при помощи прототипов. В следующем примере показано, как определить объекты Circle, которые также являются экземплярами Shape:

```
function Shape(x, y) {
    this.x = x;
    this.y = y;
}
Shape.prototype.area = function() { return 0; };
function Circle(x, y, radius) {
    Shape.call(this, x, y);
    this.radius = radius;
}
Circle.prototype = new Shape;
Circle.prototype.area = function() {
    return Math.PI * this.radius * this.radius;
};
```

Мы начинаем с определения конструктора Shape и связывания с ним функции `area()`, которая всегда возвращает ноль. Затем мы определяем конструктор Circle, который вызывает конструктор «базового класса» с помощью функции `call()`, определяемой для всех объектов-функций (включая конструкторы), а также добавляем свойство `radius`. За пределами конструктора мы указываем, что прототипом Circle будет объект Shape, а также мы подменяем функцию `area()` объекта Shape реализацией, специфичной для Circle. Этому соответствует следующий код C++:

```
class Shape
{
public:
    Shape(double x, double y) {
        this->x = x;
        this->y = y;
    }
    virtual double area() const { return 0; }
    double x;
    double y;
};

class Circle : public Shape
{
public:
    Circle(double x, double y, double radius)
        : Shape(x, y)
    {
        this->radius = radius;
    }
    double area() const { return M_PI * radius * radius; }
    double radius;
};
```

Оператор `instanceof` проходит по цепочке прототипов и определяет, какие конструкторы вызываются. Следовательно, экземпляры подкласса также считаются экземплярами базового класса:

```
var circle = new Circle(0, 0, 50);
circle instanceof Circle;    // возвращает true
circle instanceof Shape;     // возвращает true
circle instanceof Object;    // возвращает true
circle instanceof Array;     // возвращает false
```

На этом завершается наше краткое введение в ECMAScript. В следующих разделах мы покажем, как использовать этот язык в сочетании с приложениями C++/Qt для обеспечения большей гибкости и возможностей настройки под нужды пользователей, или просто для ускорения процесса разработки.

Расширение приложений Qt при помощи скриптов

С помощью модуля *QtScript* мы можем писать приложения C++, исполняющие код ECMAScript. Скрипты могут использоваться для расширения возможностей приложения без перекомпилировки и повторного развертывания приложения. Мы можем ограничить использование скриптов жестко заданным набором файлов ECMAScript, поставляемым вместе с приложением, которые можно заменять новыми версиями независимо от новых версий самого приложения, или же можно разрешить использование приложением произвольных файлов ECMAScript.


```

        button->setStyleSheet("color: rgb(31, 63, 127)");
        button->setProperty("scriptFileName",
                           scriptsDir.absoluteFilePath(fileName));
        customButtons.append(button);
    }
}

```

Функция `createCustomButtons()` использует объект `QDir` для обращения к поддиректории `scripts` с целью поиска файлов с расширением `.js`. В ней используется та же функция `directoryOf()`, которую мы применяли в главе 17.

Для каждого файла `.js` мы создаем объект `QToolButton`, вызывая закрытую функцию `createButton()`. Эта функция также соединяет сигнал `clicked()` новой кнопки со слотом `customButtonClicked()`. Затем мы задаем для кнопки таблицу стилей, чтобы сделать цвет переднего плана синим, чтобы можно было отличить пользовательские кнопки от встроенных.

Вызов свойства `QObject::setProperty()` динамически создает новое свойство `scriptFileName` для каждой кнопки `QToolButton`. Мы используем это свойство в слоте `customButtonClicked()`, чтобы определить, какой скрипт должен выполняться.

Наконец, мы добавляем новую кнопку в список `customButtons`. Конструктор класса `Calculator` использует этот список для добавления пользовательских кнопок в сетку компоновки окна.

В этом приложении мы решили проверять директорию `scripts` только один раз, при запуске приложения. Альтернативным подходом было бы использование объекта `QFileSystemWatcher` для мониторинга содержимого директории `scripts` и обновления калькулятора при изменении содержимого директории, что позволило бы пользователю добавлять новые скрипты и удалять существующие без необходимости перезапускать приложение.

```

void Calculator::customButtonClicked()
{
    QToolButton *clickedButton = qobject_cast<QToolButton *>(sender());
    QFile file(clickedButton->property("scriptFileName").toString());
    if (!file.open(QIODevice::ReadOnly)) {
        abortOperation();
        return;
    }
    QTextStream in(&file);
    in.setCodec("UTF-8");
    QString script = in.readAll();
    file.close();
    QScriptEngine interpreter;
    QScriptValue operand(&interpreter, display->text().toDouble());
    interpreter.globalObject().setProperty("x", operand);
    QScriptValue result = interpreter.evaluate(script);
    if (!result.isNumber()) {
        abortOperation();
        return;
    }
}

```

```

setDisplayValue(result.toNumber());
waitingForOperand = true;
}

```

В слоте `customButtonClicked()` мы сначала вызываем функцию `QObject::sender()`, чтобы определить, какая кнопка была нажата. Затем мы извлекаем свойство `scriptIdName`, чтобы получить имя файла `.js`, который связан с данной кнопкой. Далее мы загружаем содержимое файла в строку с именем `script`.

Стандарт ECMAScript требует, чтобы интерпретаторы поддерживали Unicode, однако он не обязывает использовать какую-либо конкретную кодировку для скриптов, хранящихся на диске. Мы предположили, что в наших `.js`-файлах используется кодировка UTF-8, часть простого набора ASCII. После того, как мы поместили скрипт в объект `QString`, мы создаем объект `QScriptEngine` для его выполнения. Экземпляр класса `QScriptEngine` представляет собой интерпретатор ECMAScript, и он сохраняет текущее состояние. Мы можем использовать одновременно любое количество объектов `QScriptEngine` для разных целей, и каждый будет иметь собственное состояние.

Прежде чем мы сможем запустить скрипт, мы должны предоставить скрипту возможность получить текущее значение, отображаемое калькулятором. Здесь мы выбрали подход, связанный с созданием глобальной переменной ECMAScript с именем `x` или, что более точно, мы добавили динамическое свойство `x` в глобальный объект интерпретатора. Из кода скрипта данное свойство можно видеть напрямую как `x`.

Значение, задаваемое для `x`, должно иметь тип `QScriptValue`. По сути, тип `QScriptValue` сходен с `QVariant` в том, что он может хранить много типов данных, только он ориентирован на хранение типов данных ECMAScript.

Наконец, мы запускаем скрипт с помощью функции `QScriptEngine::evaluate()`. Результатом является значение, возвращаемое скриптом, или объект-исключение, если возникает ошибка. (В следующем разделе мы рассмотрим, как сообщить об ошибке пользователю с помощью окна сообщений). Возвращаемое значение скрипта – это то значение, которое явно возвращается оператором `return`. Если оператор `return` опущен, то результатом будет последнее вычисленное скриптом выражение. Получив возвращаемое значение, мы проверяем, является ли оно числом, и если это так, выводим его на экран.

В данном примере мы выполняем скрипт каждый раз, когда пользователь нажимает соответствующую кнопку. Поскольку данный этап связан с загрузкой и обработкой всего скрипта, часто более предпочтительным является другой подход, где скрипт не выполняет операцию напрямую, а возвращает функцию или объект, который можно использовать позже. В следующем разделе мы будем использовать этот альтернативный подход.

Чтобы подключить библиотеку `QtScript`, мы должны добавить в файл `.pro` приложения следующую строку:

```
QT += script
```

Данные примеры скриптов очень просты. Вот однострочный скрипт `p1.js`:

```
return 3.14159265358979323846;
```


Обратите внимание, что мы игнорируем значение `x` калькулятора. Скрипт `cube.js` также состоит из одной строки, но в нем значение `x` используется:

```
return x * x * x;
```

В скрипте `factorial.js` определяется и вызывается функция:

```
function factorial(n)
{
    if (n <= 1) {
        return 1;
    } else {
        return n * factorial(n - 1);
    }
}
return factorial(Math.floor(x));
```

Стандартная функция факториала работает только с целочисленными значениями, поэтому мы использовали функцию `Math.floor()` для преобразования `x` и в целое число.

Итак, мы рассмотрели основы модуля `QtScript`: класс `QScriptEngine`, представляющий собой интерпретатор и его текущее состояние, и класс `QScriptValue`, хранящий значение `ECMAScript`.

В примере приложения `Calculator` скрипты и приложение взаимодействовали очень мало: скрипты принимали от приложения всего один параметр и возвращали одно значение. В следующих разделах мы рассмотрим более сложные стратегии интеграции, и покажем, как отображать сообщения об исключительных ситуациях для пользователя.

Реализация расширений графического интерфейса с помощью скриптов

Создание скриптов для вычисления значений, как мы сделали в предыдущем разделе, – метод полезный, но имеющий ограничения. Часто нам требуется доступ прямо из скриптов к некоторым виджетам и другим компонентам приложения. Также нам может потребоваться создавать дополнительные диалоговые окна, объединяя файлы `ECMAScript` с файлами `.ui QtDesigner`. Использование таких методов позволяет разрабатывать приложения, в основном, на `ECMAScript`, что является весьма привлекательным для некоторых программистов.

В данном разделе мы рассмотрим приложение-редактор `HTML` (`HTML Editor`), показанное на рис. 22.3. Это приложение представляет собой редактор обычного текста, который выделяет теги, используя класс `QSyntaxHighlighter`. Особенным делает данное приложение то, что оно дополнено расширениями в форме скриптов `.js`, а также соответствующими диалоговыми окнами, которые все расположены в поддиректории `scripts`. Диалоговые окна позволяют пользователям записать в параметрической форме операцию, которую они хотят произвести.

Мы предлагаем два расширения – диалоговое окно `Statistics` и диалоговое окно `Reformat Text`, которые показаны на рис. 22.4. Диалоговое окно `Statistics` является чисто информационным. Оно подсчитывает количество символов, слов

и строк в документе и представляет эту информацию пользователю в виде модального диалогового окна. Окно **Reformat Text** является более сложным. Это немодальное диалоговое окно, а это означает, что пользователь может продолжать работать с главным окном приложения, пока диалоговое окно отображается на экране. Данное диалоговое окно может использоваться для изменения отступов в тексте, для переноса длинных строк и для стандартизации регистра символов, используемого в тегах. Все эти операции реализованы на ECMAScript.

Главным в приложении является класс **HtmlWindow**, который является подклассом **QMainWindow** и который использует в качестве своего центрального виджета **QTextEdit**. Здесь мы рассмотрим только те части кода, которые связаны с созданием скрипта для приложения.

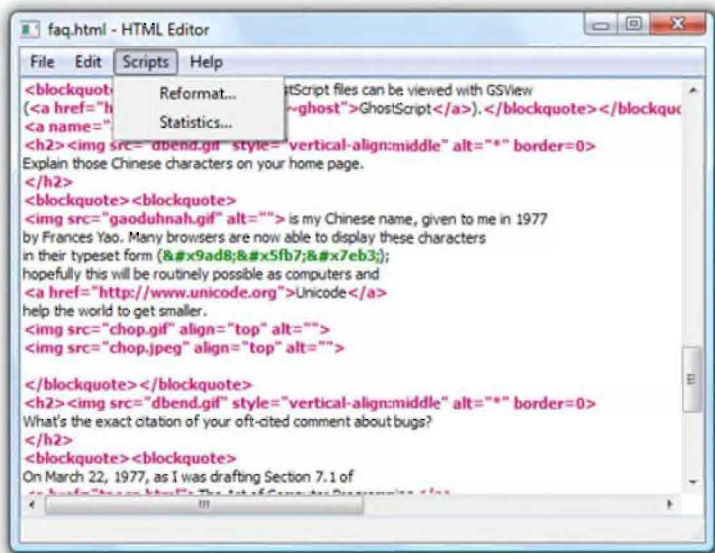


Рис. 22.3. Приложение HTML Editor

При запуске приложения мы должны заполнить меню **Scripts** операциями, соответствующими файлам **.js** и **.ui**, находящимся в поддиректории **scripts**. Процесс очень напоминает то, что мы делали в функции **createCustom Buttons()** приложения **Калькулятор** в предыдущем разделе:

```
void HtmlWindow::createScriptsMenu()
{
    scriptsMenu = menuBar()->addMenu(tr("&Scripts"));
    QDir scriptsDir = directoryOf("scripts");
    QStringList jsFileNames = scriptsDir.entryList(QStringList("*.js"),
                                                    QDir::Files);
    foreach (QString jsFileName, jsFileNames)
        createScriptAction(scriptsDir.absoluteFilePath(jsFileName));
    scriptsMenu->setEnabled(!scriptsMenu->isEmpty());
}
```

Для каждого скрипта мы вызываем функцию `createScriptAction()`, чтобы создать операцию и поместить ее в меню `Scripts`. Если не найдено ни одного скрипта, мы делаем меню недоступным.

Функция `createScriptAction()` выполняет следующие действия.

1. Загружает и проверяет скрипт, сохраняя получившийся объект в переменной.
2. Конструирует диалоговое окно из файла `.ui` с помощью класса `QDialog`.
3. Делает диалоговое окно доступным для скрипта.
4. Открывает для скрипта специфическую функциональность приложения.
5. Создает объект `QAction`, чтобы сделать скрипт доступным для пользователя.

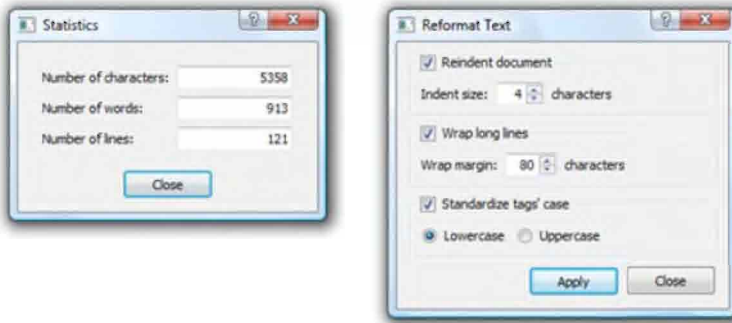


Рис. 22.4. Диалоговые окна `Statistics` и `Reformat Text`

Функция должна выполнить массу работы, и она очень длинная, так что мы будем рассматривать ее частями.

```
bool HtmlWindow::createScriptAction(const QString &jsFileName)
{
    QFile jsFile(jsFileName);
    if (!jsFile.open(QIODevice::ReadOnly)) {
        QMessageBox::warning(this, tr("HTML Editor"),
                               tr("Cannot read file %1:\n%2.")
                               .arg(strippedName(jsFileName))
                               .arg(jsFile.errorString()));
        return false;
    }

    QTextStream in(&jsFile);
    in.setCodec("UTF-8");
    QString script = in.readAll();
    jsFile.close();

    QScriptValue qsScript = interpreter.evaluate(script);
    if (interpreter.hasUncaughtException()) {
        QMessageBox messageBox(this);
```

```

messageBox.setIcon(QMessageBox::Warning);
messageBox.setWindowTitle(tr("HTML Editor"));
messageBox.setText(tr("An error occurred while executing the "
                      "script %1.")
                  .arg(strippedName(jsFileName)));
messageBox.setInformativeText(
    tr("%1.").arg(interpreter.uncaughtException()
                  .toString()));
messageBox.setDetailedText(
    interpreter.uncaughtExceptionBacktrace().join("\n"));
messageBox.exec();
return false;
}

```

Мы начинаем с чтения файла `.js`. Поскольку нам нужно использовать только один интерпретатор, мы применяем только одну переменную-член класса `QScriptEngine`, с именем `interpreter`. Мы запускаем скрипт и сохраняем возвращаемое значение в виде переменной типа `QScriptValue` с именем `qsScript`.

Если получить правильное значение от скрипта не получается (например, из-за синтаксической ошибки), функция `QScriptEngine::hasUncaughtException()` возвратит значение `true`. В этом случае мы выводим сообщение об ошибке с помощью `QMessageBox`.

Для скриптов, используемых в данном приложении, мы применили соглашение, согласно которому каждый скрипт должен при выполнении возвращать значение `ECMAScript` типа `Object`. Этот объект должен содержать два свойства — строку с именем `text`, содержащую текст, используемый в меню `Scripts` для идентификации скрипта, и функцию с именем `run()`, которая должна вызываться, когда пользователь выбирает данный скрипт в меню `Scripts`.

Мы сохраняем данный объект в переменной `qsScript`. Главное преимущество такого подхода — это то, что нам нужно прочитать и обработать скрипты только один раз, при запуске.

```

QString uiFileName = jsFileName;
uiFileName.chop(3);
uiFileName += ".ui";

QFile uiFile(uiFileName);
if (!uiFile.open(QIODevice::ReadOnly)) {
    QMessageBox::warning(this, tr("HTML Editor"),
                        tr("Cannot read file %1:\n%2.")
                        .arg(strippedName(uiFileName))
                        .arg(uiFile.errorString()));

    return false;
}

QUiLoader loader;
QWidget *dialog = loader.load(&uiFile, this);
uiFile.close();
if (!dialog) {

```

```

QMessageBox::warning(this, tr("HTML Editor"),
                    tr("Error loading %1.")
                    .arg(strippedName(uiFileName)));

return false;
}

```

Другое используемое нами соглашение состоит в том, что для каждого скрипта должен существовать соответствующий файл `.ui`, предоставляющий скрипту диалоговое окно. Файл `.ui` должен иметь то же базовое имя, что и скрипт.

Мы пытаемся прочитать файл `.ui` и динамически создать объект `QWidget`, содержащий все виджеты, компоновки и соединения, указанные в файле `.ui`. Родитель виджета указывается во втором аргументе вызова функции `load()`. Если возникает ошибка, мы предупреждаем пользователя и выходим из функции.

```

QScriptValue qsDialog = interpreter.newQObject(dialog);
qsScript.setProperty("dialog", qsDialog);
QScriptValue qsTextEdit = interpreter.newQObject(textEdit);
qsScript.setProperty("textEdit", qsTextEdit);
QAction *action = new QAction(this);
action->setText(qsScript.property("text").toString());
action->setData(QVariant::fromValue(qsScript));
connect(action, SIGNAL(triggered()),
        this, SLOT(scriptActionTriggered()));
scriptsMenu->addAction(action);
return true;
}

```

После того, как мы успешно прочитали скрипт и его файл пользовательского интерфейса, мы практически готовы к тому, чтобы поместить его в меню `Scripts`. Но для начала мы должны обратить внимание на ряд деталей. Нам нужно, чтобы функция `run()` нашего скрипта имела доступ к только что созданному диалоговому окну. Кроме того, скрипту должен быть предоставлен доступ к объекту `QTextEdit`, где находится редактируемый HTML-документ.

Мы начинаем с добавления диалогового окна в интерпретатор в виде указателя `QObject *`. В ответ интерпретатор возвращает значение `Object`, используемое для представления диалогового окна. Мы сохраняем это значение в переменной `qsDialog`. Мы добавляем объект `qsDialog` к объекту `qsScript` в виде нового свойства с именем `dialog`. Это означает, что скрипт может обращаться к диалоговому окну, включая его виджеты, при помощи только что созданного свойства `dialog`. Тот же метод мы используем для того, чтобы представить скрипту доступ к объекту `QTextEdit` приложения.

Наконец, мы создаем новый объект `QAction`, представляющий скрипт в графическом интерфейсе пользователя. Мы задаем в качестве текста для операции значение свойства `text` объекта `qsScript`, а в качестве элемента «data» операции сам объект `qsScript`. Наконец, мы соединяем сигнал `triggered()` действия с пользовательским слотом `scriptActionTriggered()` и добавляем операцию в меню `Scripts`.

```

void HtmlWindow::scriptActionTriggered()
{

```

```

QAction *action = qobject_cast<QAction *>(sender());
QScriptValue qsScript = action->data().value<QScriptValue>();
qsScript.property("run").call(qsScript);
}

```

Когда слот вызывается, мы начинаем с выяснения того, какой объект `QAction` был запущен. Затем мы извлекаем пользовательские данные при помощи функции `QVariant::value<T>()`, чтобы привести их тип к `qsScript`. Затем мы вызываем функцию `run()` объекта `qsScript`, передавая `qsScript` в качестве параметра. Это делает `qsScript` объектом `this` внутри функции `run()`.¹

Механизм элемента «data» объекта `QAction` основывается на `QVariant`. Тип `QScriptValue` не является одним из типов, которые распознает `QVariant`. К счастью, в Qt предлагается механизм для расширения числа типов, поддерживаемых классом `QVariant`. В начале файла `htmlwindow.cpp`, после инструкций `#include` у нас есть строка:

```
Q_DECLARE_METATYPE(QScriptValue)
```

Эта строка должна стоять после объявления пользовательского типа, к которому она относится, и ее можно создавать только для тех типов данных, для которых есть конструктор по умолчанию или конструктор копий.

Теперь, когда мы увидели, как загружать скрипт и файл пользовательского интерфейса, и как предоставить операцию, с помощью которой пользователь сможет запустить скрипт, все готово к тому, чтобы мы изучили сами скрипты. Мы начнем со скрипта `Statistics`, поскольку он самый простой и короткий, и рассмотрим его по частям.

```

var obj = new Object;
obj.text = "&Statistics...";

```

Мы начинаем с создания новой переменной `Object`. Она представляет собой объект, в который мы будем добавлять свойства, возвращаемые интерпретатору. Первое свойство, которое мы создадим — это свойство `text`, содержащее текст, отображаемый в меню `Scripts`.

```

obj.run = function() {
    var text = this.textEdit.plainText;
    this.dialog.frame.charCountLineEdit.text = text.length;
    this.dialog.frame.wordCountLineEdit.text = this.wordCount(text);
    this.dialog.frame.lineCountLineEdit.text = this.lineCount(text);
    this.dialog.exec();
};

```

Второе свойство — это функция `run()`. Эта функция читает текст из поля `QTextEdit` диалогового окна, заполняет виджеты диалогового окна результатами вычислений и завершается отображением диалогового окна в модальном режиме.

¹ Ожидается, что в Qt 4.4 будет предоставлена функция `qScriptConnect()`, которая позволит нам устанавливать соединения между C++ и скриптом. С помощью этой функции мы можем подключать сигнал `triggered()` прямо к функции `run()` объекта `qsScript` следующим образом:
`qScriptConnect(action, SIGNAL(triggered()), qsScript, qsScript.property("run"));`

Данная функция может работать, только если переменная типа `Object`, `obj`, содержит подходящие свойства `textEdit` и `dialog`, и именно поэтому мы должны добавить эти свойства в конце функции `createScriptAction()`. Само диалоговое окно должно содержать объект `frame` (в данном случае типа `QFrame`, но тип не имеет значения) с тремя дочерними виджетами — `charCountLineEdit`, `wordCountLineEdit` и `lineCountLineEdit`, каждый из которых содержит доступное для записи свойство `text`. Вместо `this.dialog.frame.xxxCountLineEdit` мы также можем написать `findChild(«xxxCountLineEdit»)`. В этом случае происходит рекурсивный поиск и, следовательно, данный вариант более надежен, если мы решим изменить дизайн диалогового окна.

```
obj.wordCount = function(text) {
    var regExp = new RegExp("\\w+", "g");
    var count = 0;
    while (regExp.exec(text))
        ++count;
    return count;
};

obj.lineCount = function(text) {
    var count = 0;
    var pos = 0;
    while ((pos = text.indexOf("\n", pos)) != -1) {
        ++count;
        ++pos;
    }
    return count + 1;
};

return obj;
```

Функции `wordCount()` и `lineCount()` не имеют внешних зависимостей, и работают только с переданными им объектами `String`. Обратите внимание, что функция использует класс `RegExp` `ECMAScript`, а не класс `QRegExp` `Qt`. В конце файла скрипта оператор `return` гарантирует, что интерпретатору будет возвращена готовая к использованию переменная типа `Object` со свойством `text` и функцией `run()`.

В скрипте `Reformat` используется примерно та же схема, что и в скрипте `Statistics`. Далее мы рассмотрим этот скрипт.

```
var obj = new Object;

obj.initialized = false;

obj.text = "&Reformat...";

obj.run = function() {
    if (!this.initialized) {
        this.dialog.applyButton.clicked.connect(this, this.apply);
        this.dialog.closeButton.clicked.connect(this, this.dialog.close);
        this.initialized = true;
```

```

    }
    this.dialog.show();
};

obj.apply = function() {
    var text = this.textEdit.plainText;

    this.textEdit.readOnly = true;
    this.dialog.applyButton.enabled = false;

    if (this.dialog.indentGroupBox.checked) {
        var size = this.dialog.indentGroupBox.indentSizeSpinBox.value;
        text = this.reindented(text, size);
    }
    if (this.dialog.wrapGroupBox.checked) {
        var margin = this.dialog.wrapGroupBox.wrapMarginSpinBox.value;
        text = this.wrapped(text, margin);
    }
    if (this.dialog.caseGroupBox.checked) {
        var lowercase = this.dialog.caseGroupBox.lowercaseRadio.checked;
        text = this.fixedTagCase(text, lowercase);
    }

    this.textEdit.plainText = text;
    this.textEdit.readOnly = false;
    this.dialog.applyButton.enabled = true;
};

obj.reindented = function(text, size) {
    ...
};

obj.wrapped = function(text, margin) {
    ...
};

obj.fixedTagCase = function(text, lowercase) {
    ...
};

return obj;

```

Мы используем ту же схему, что и ранее – создаем переменную `Object` без свойств, добавляем к ней свойства и возвращаем ее интерпретатору. Наряду со свойствами `text` и `run()`, мы добавляем свойство `initialized`. При первом запуске функции `run()` свойство `initialized` равно `false`, поэтому мы создаем соединение сигнал-слот, связывающее нажатия кнопок в диалоговом окне с функциями, которые определены в скрипте.

Здесь используются те же допущения, что и в скрипте Statistics. Мы предполагаем, что существует соответствующее свойство `dialog`, что для него существуют кнопки `applyButton` и `closeButton`. Функция `apply()` взаимодействует с виджетами диалогового окна и, в частности, с кнопкой `Apply` (делая ее доступной или недоступной), а также с групповыми полями, флажками и счетчиками. Также она взаимодействует с полем `QTextEdit` главного окна, откуда она получает текст для работы и в которое передает текст, получившийся в результате преформатирования.

Мы опустили код функций `reindented()`, `wrapped()` и `fixedTagCase()`, используемых внутри скрипта, поскольку эти вычисления не важны для понимания того, как работать со скриптами в приложениях Qt.

На этом завершается наш технический обзор использования скриптов в приложениях C++/Qt, включая скрипты со своими собственными диалоговыми окнами. В таких приложениях, как `HTML Editor`, где скрипты взаимодействуют с объектами приложения, мы также должны учитывать вопрос лицензирования. Для приложений с открытым исходным кодом не существует никаких ограничений, за исключением тех, которые налагаются требованиями самой лицензии на открытый исходный код. Для коммерческих приложений все несколько сложнее. Те, кто пишет скрипты для коммерческих приложений, включая конечных пользователей этих приложений, вольны делать это, если их скрипты используют только встроенные классы `ECMAScript` и специфический программный интерфейс (API) приложения, или если они используют API Qt для внесения небольших дополнений или изменений в существующие компоненты. Однако любой создатель скриптов, реализующих базовую функциональность графического пользовательского интерфейса, должен иметь коммерческую лицензию Qt. Если у коммерческих пользователей Qt возникают вопросы относительно лицензирования, им следует обращаться к своему торговому представителю Trolltech.

Автоматическое выполнение задач с применением скриптов

Иногда мы используем приложения с графическим интерфейсом для однотипной манипуляции данными. Если манипуляция связана с вызовом многих опций меню или с взаимодействием с диалоговым окном, то это не только становится утомительным, но и возникает опасность пропуска каких-то этапов или нарушения порядка выполнения шагов, и мы даже не поймем, что допущена ошибка. Один из способов упростить такие задачи для пользователей — это позволить им написать скрипты, выполняющие последовательности действий автоматически.

В данном разделе мы представим приложение с графическим интерфейсом, предоставляющее опцию командной строки `-script`, с помощью которой пользователь может указать выполняемый скрипт. Приложение запустится, выполнит скрипт и завершит работу, вообще не отображая графический интерфейс.

Для демонстрации данного метода мы используем приложение `Gas Pump` (Бензонасос). Оно считывает список операций, выполненных колонками бензозаправочной станции для грузовиков, и представляет данные в табличном формате, как показано на рис. 22.5.

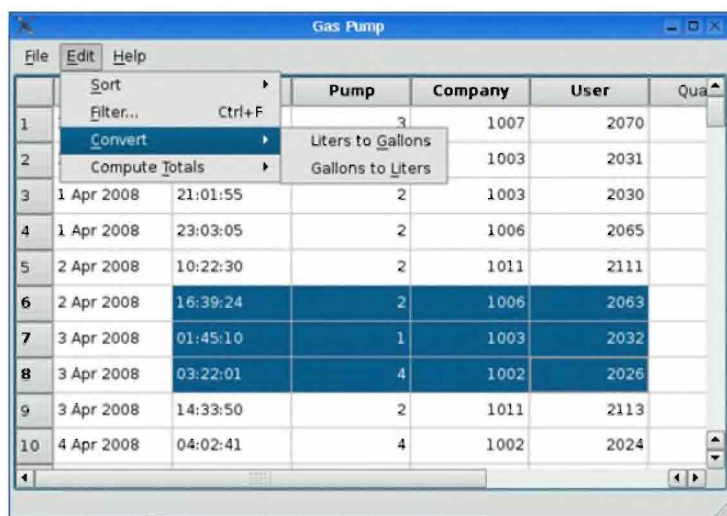


Рис. 22.5. Приложение Бензонасос

Для каждой операции записывается дата, время, номер насоса, объем топлива, идентификатор (ID) компании и пользовательский ID водителя, а также состояние операции. Приложение Бензонасос может использоваться для очень сложной манипуляции данными, для их сортировки, фильтрации, вычисления итоговых значений и преобразований из литров в галлоны и обратно.

Приложение Бензонасос поддерживает два формата данных об операциях: «Pump 2000» – простой текстовый формат с расширением файлов .p20, и «XML Gas Pump» – XML-формат с расширением .gpx. Приложение может загружать и сохранять данные в обоих форматах, так что его можно использовать для преобразования из одного формата в другой путем простой загрузки данных в одном формате и сохранения в другом.

Для приложения предлагаются четыре стандартных скрипта:

- onlyok.js – удаляет все операции, для которых состояние отлично от «ОК»;
- p20togpx.js – преобразует файл Pump 2000 в формат XML Gas Pump;
- tohtml.js – создает отчеты в формате HTML;
- toliters.js – преобразует единицы из галлонов в литры.

Скрипты вызываются с помощью опции командной строки `-script`, за которой идет имя скрипта, а потом имена файлов, с которыми скрипт будет работать. Например:

```
gaspump -script scripts/toliters.js data/2008q2.p20
```

Здесь мы запустим скрипт `toliters.js` из поддиректории `scripts` применительно к файлу данных Pump 2000 с именем `2008q2.p20` из поддиректории `data`. Скрипт преобразует все количественные значения из галлонов в литры, изменяя файл «на месте».

Приложение Бензонасос пишется так же, как любое другое приложение C++/Qt. Фактически, его код очень напоминает пример электронной таблицы из

глав 3 и 4. Приложение содержит подкласс PumpWindow класса QMainWindow, который формирует каркас приложения, включая его действия и меню. (Меню показаны на рис. 22.6.) Существует также пользовательский подкласс PumpSpreadsheet класса QTableWidgetItem для отображения данных. Кроме того, есть подкласс FilterDialog класса QDialog, показанный на рис. 22.7, который пользователь может использовать для указания опций фильтров. Поскольку существует множество опций фильтров, они все хранятся в классе PumpFilter. Мы очень коротко рассмотрим эти классы, а затем увидим, как в приложение добавляется поддержка скриптов.

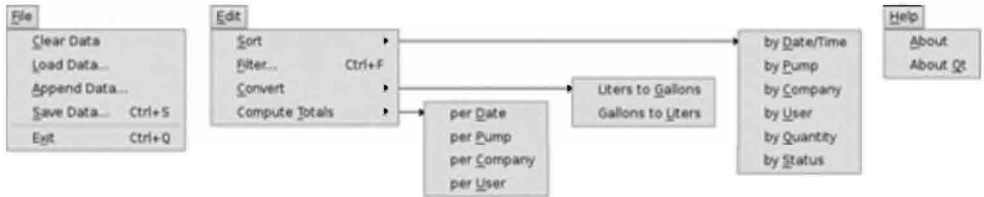


Рис. 22.6. Меню приложения Бензонасос

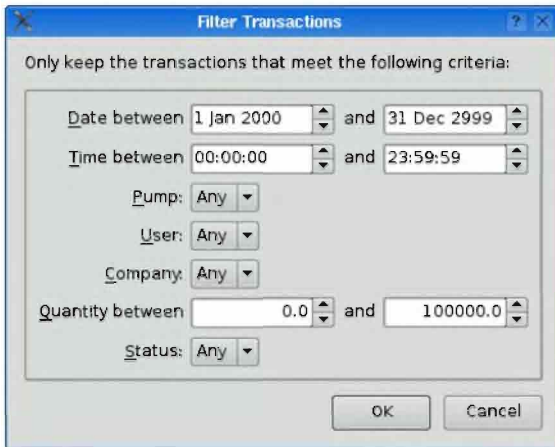


Рис. 22.7. Диалоговое окно Filter

```
class PumpSpreadsheet : public QTableWidgetItem
{
    Q_OBJECT
    Q_ENUMS(FileFormat Column)

public:
    enum FileFormat { Pump2000, GasPumpXml };
    enum Column { Date, Time, Pump, Company, User, Quantity, Status,
                  ColumnCount };
    PumpSpreadsheet(QWidget *parent = 0);

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

```

bool addData(const QString &fileName, FileFormat format);
bool saveData(const QString &fileName, FileFormat format);
void sortByColumn(Column column,
                  Qt::SortOrder order = Qt::AscendingOrder);
void applyFilter(const PumpFilter &filter);
void convertUnits(double factor);
void computeTotals(Column column);
void setText(int row, int column, const QString &text);
QString text(int row, int column) const;

```

```
private:
```

```
...
```

```
};
```

Класс PumpSpreadsheet хранит данные и предоставляет функции (мы превратили их в слоты), которые пользователь может применять для манипуляции данными. Доступ к слотам осуществляется через пользовательский интерфейс, и они также доступны для создания скриптов. Макрос Q_ENUMS() используется для генерации мета-информации о перечислимых типах FileFormat и Column. Мы к этому скоро вернемся.

Класс PumpWindow, являющийся подклассом QMainWindow содержит функцию loadData(), которая использует некоторые слоты PumpSpreadsheet:

```

void PumpWindow::loadData()
{
    QString fileName = QFileDialog::getOpenFileName(this,
        tr("Open Data File"), ".",
        fileFilters);
    if (!fileName.isEmpty()) {
        spreadsheet->clearData();
        spreadsheet->addData(fileName, fileFormat(fileName));
    }
}

```

Объект PumpSpreadsheet хранится в объекте PumpWindow как переменная-член с именем spreadsheet. Слот filter() объекта PumpWindow является менее типичным:

```

void PumpWindow::filter()
{
    FilterDialog dialog(this);
    dialog.initFromSpreadsheet(spreadsheet);
    if (dialog.exec())
        spreadsheet->applyFilter(dialog.filter());
}

```

Функция initFromSpreadsheet() заполняет комбинированные поля диалогового окна FilterDialog данными о насосах, ID компаний, ID пользователей и кодами состояний, используемыми в текущем наборе данных. При вызове функции exec() появляется диалоговое окно, показанное на рис. 22.7. Если пользователь нажимает ОК, то функция filter() окна FilterDialog возвращает объект PumpFilter, который мы передаем в функцию PumpSpreadsheet::applyFilter().

```
class PumpFilter
{
    public:
        PumpFilter();
        QDate fromDate;
        QDate toDate;
        QTime fromTime;
        QTime toTime;
        QString pump;
        QString company;
        QString user;
        double fromQuantity;
        double toQuantity;
        QString status;
};
```

Назначение класса `PumpFilter` – упрощенная передача опций фильтров в виде группы, а не в виде десяти отдельных параметров.

Пока, все что мы видели, удивления не вызывает. Заметно лишь то отличие, что мы превратили все функции `PumpSpreadsheet`, для которых мы хотим использовать поддержку скриптов, в открытые слоты, а также использовали макрос `Q_ENUMS()`. Чтобы приложение `Бензонасос` поддерживало скрипты, мы должны сделать две вещи. Во-первых, мы должны изменить файл `main.cpp`, добавив обработку командной строки и запуск скрипта, если он указан. Во-вторых, мы должны сделать функциональность приложения доступной для скриптов.

Модуль *QtScript* предоставляет два общих способа раскрытия классов C++ для скриптов. Самый простой способ – это определить функциональность в классе `QObject` и открыть для скриптов один или несколько экземпляров этого класса с помощью функции `QScriptEngine::newQObject()`. Свойства и слоты, определенные в этом классе (а, при желании, и в его потомках) станут доступны для скриптов. Более сложным, но и более гибким подходом является создание прототипа класса C++, а, возможно, и функции-конструктора для тех классов, экземпляры которых должны создаваться скриптом при помощи оператора `new`. В примере с приложением `Бензонасос` мы покажем оба подхода.

Прежде чем мы начнем изучать инфраструктуру, используемую для запуска скриптов, давайте рассмотрим один из скриптов, поставляемых с приложением `Бензонасос`. Вот полный текст скрипта `onlyok.js`:

```
if (args.length == 0)
    throw Error("No files specified on the command line");

for (var i = 0; i < args.length; ++i) {
    spreadsheet.clearData();
    if (!spreadsheet.addData(args[i], PumpSpreadsheet.Pump2000))
        throw Error("Error loading Pump 2000 data");

    var filter = new PumpFilter;
    filter.status = "OK";
```

```

spreadsheet.applyFilter(filter);
if (!spreadsheet.saveData(args[i], PumpSpreadsheet.Pump2000))
    throw Error("Error saving Pump 2000 data");
print("Removed erroneous transactions from " + args[i]);
}

```

Данный скрипт использует две глобальные переменные: `args` и `spreadsheet`. Переменная `args` возвращает аргументы командной строки, переданные после опции `-script`. Переменная `spreadsheet` является ссылкой на объект `PumpSpreadsheet`, который мы используем для выполнения различных операций (преобразование формата файла, преобразование единиц измерения, фильтрация и т. п.). Этот скрипт также вызывает некоторые слоты объекта `PumpSpreadsheet`, создает экземпляры и инициализирует объекты `PumpFilter`, а также использует перечислимый тип `PumpSpreadsheet::FileFormat`.

Мы начинаем с простой профилактической проверки, а затем для каждого имени файла, указанного в командной строке, мы очищаем глобальный объект `spreadsheet` и пытаемся загрузить данные из файла. Мы предполагаем, что все файлы имеют формат `Pump 2000` (.p20). Для каждого успешно загруженного файла мы создаем новый объект `PumpFilter`. Мы задаем свойство `status` фильтра, а затем вызываем функцию `applyFilter()` объекта `PumpSpreadsheet`. Наконец, мы сохраняем данные электронной таблицы в исходном файле и выводим сообщение для пользователя.

Остальные три скрипта имеют сходную структуру. Они включены в примеры исходного кода, прилагаемые к этой книге.

Для поддержки таких скриптов, как `onlyok.js` нам нужно выполнять в приложении Бензонасос следующие шаги.

1. Обнаружить опцию `-script` командной строки.
2. Загрузить указанный файл скрипта.
3. Открыть экземпляр класса `PumpSpreadsheet` для интерпретатора.
4. Открыть аргументы командной строки для интерпретатора.
5. Открыть перечислимые типы `FileFormat` и `Column` для интерпретатора.
6. Создать оболочку класса `PumpFilter`, чтобы его переменные-члены были доступны для скрипта.
7. Сделать возможным создание объектов `PumpFilter` в скрипте.
8. Выполнить скрипт.

Имеющий для нас значение код находится в файлах `main.cpp`, `scripting.cpp` и `scripting.h`. Давайте начнем с `main.cpp`:

```

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QStringList args = QApplication::arguments();
    if (args.count() >= 3 && args[1] == "-script") {
        runScript(args[2], args.mid(3));
        return 0;
    } else if (args.count() == 1) {

```

```

    PumpWindow window;
    window.show();
    window.resize(600, 400);
    return app.exec();
} else {
    std::cerr << "Usage: gaspump [-script myscript.js <arguments>]"
              << std::endl;
    return 1;
}
}

```

Доступ к аргументам командной строки осуществляется через функцию `QApplication::arguments()`, которая возвращает список `QStringList`. Первый элемент списка – это имя приложения. Если аргументов не менее трех, и второй аргумент `-script`, мы предполагаем, что третий аргумент – это имя скрипта. В этом случае мы вызываем функцию `runScript()`, указывая имя скрипта в качестве первого аргумента, а оставшуюся часть списка передаем в качестве второго параметра. После выполнения скрипта работа приложения сразу же завершается.

Если в списке содержится только один аргумент, имя приложения, мы создаем и отображаем окно `PumpWindow` и запускаем цикл событий приложения обычным способом.

Поддержку скриптов в приложении обеспечивают файлы `scripting.h` и `scripting.cpp`. В этих файлах определяется функция `runScript()`, вспомогательная функция `pumpFilterConstructor()` и вспомогательный класс `PumpFilterPrototype`. Вспомогательная функция и класс являются специфичными для приложения Бензонасос, но мы все же рассмотрим их, поскольку они иллюстрируют некоторые моменты, связанные с поддержкой скриптов в приложениях.

Мы рассмотрим функцию `runScript()` по частям, поскольку она содержит некоторые тонкие моменты.

```

bool runScript(const QString &fileName, const QStringList &args)
{
    QFile file(fileName);
    if (!file.open(QIODevice::ReadOnly)) {
        std::cerr << "Error: Cannot read file " << qPrintable(fileName)
                  << ": " << qPrintable(file.errorString())
                  << std::endl;
        return false;
    }
    QTextStream in(&file);
    in.setCodec("UTF-8");
    QString script = in.readAll();
    file.close();
}

```

Мы начинаем с чтения скрипта в объект `QString`.

```

QScriptEngine interpreter;

PumpSpreadsheet spreadsheet;

```

```
QScriptValue qsSpreadsheet = interpreter.newQObject(&spreadsheet);
interpreter.globalObject().setProperty("spreadsheet",
                                     qsSpreadsheet);
```

После того, как скрипт помещен в объект `QString`, мы создаем объект `QScriptEngine` и экземпляр `Pump Spreadsheet`. Затем мы создаем объект `QScriptValue` для ссылки на экземпляр `Pump Spreadsheet` и делаем его глобальным свойством интерпретатора, чтобы он стал доступным для скриптов через глобальную переменную `spreadsheet`. Все слоты и свойства объекта `Pump Spreadsheet` доступны через переменную `spreadsheet` для любого скрипта, который будет их использовать.

```
QScriptValue qsArgs = qScriptValueFromSequence(&interpreter, args);
interpreter.globalObject().setProperty("args", qsArgs);
```

Список `args` (который может быть пустым) типа `QStringList`, который передан в функцию `run Script()`, содержит аргументы командной строки, которые пользователь пожелал передать скрипту. Чтобы эти аргументы были доступны для скриптов, мы должны, как обычно, создать для их представления объект `QScriptValue`. Чтобы преобразовать контейнер с последовательной структурой, такой как `QList<T>` или `QVector<T>` в `QScriptValue`, мы можем использовать глобальную функцию `qScriptValueFromSequence()`, предоставленную модулем *QtScript*. Мы делаем аргументы доступными для скриптов в форме глобальной переменной с именем `args`.

```
QScriptValue qsMetaObject =
    interpreter.newQMetaObject(spreadsheet.metaObject());
interpreter.globalObject().setProperty("PumpSpreadsheet",
                                     qsMetaObject);
```

В файле `pumpspreadsheet.h` мы определяем перечислимые типы `FileFormat` и `Column`. Кроме того, мы также включаем объявление `Q_ENUMS()`, в котором описаны эти перечислимые типы. Обычно в `Qt`-программировании `Q_ENUMS()` используется редко. Его основное назначение – создание пользовательских виджетов, которые мы хотим сделать доступными для *Qt Designer*. Но оно также полезно в контексте создания скриптов, поскольку мы можем сделать перечислимые типы доступными для скриптов, зарегистрировав мета-объект класса, который их содержит.

Добавляя мета-объект класса `PumpSpreadsheet` в виде глобальной переменной `PumpSpreadsheet`, мы делаем перечислимые типы `FileFormat` и `Column` доступными для скриптов. Создатели скриптов могут обращаться к перечислимым значениям, введя, скажем, `PumpSpreadsheet.Pump2000`.

```
PumpFilterPrototype filterProto;
QScriptValue qsFilterProto = interpreter.newQObject(&filterProto);
interpreter.setDefaultPrototype(qMetaTypeId<PumpFilter>(),
                               qsFilterProto);
```

Поскольку в `ECMAScript` используются прототипы, а не классы, как они есть в `C++`, то, если мы хотим сделать пользовательский класс `C++` доступным для скриптов, мы должны использовать обходной маневр. В примере с приложением *Бензонасос* нам нужно включить поддержку скриптов для класса `PumpFilter`.

Одним из подходов было бы заменить сам класс и заставить его использовать мета-объектную систему Qt для экспортирования членов-данных в форме Qt-свойств. Так, для иллюстрации в приложении Бензонасос мы решили сохранить без изменений исходное приложение и создать класс-оболочку, `PumpFilterPrototype`, который может хранить объект `PumpFilter` и обеспечивать доступ к нему.

Вызов функции `setDefaultPrototype()`, которая показана выше, заставляет интерпретатор использовать экземпляр класса `PumpFilterPrototype` как неявный прототип для всех объектов `PumpFilter`. Этот прототип происходит от `QObject` и предоставляет свойства Qt для доступа к данным – членам класса `PumpFilter`.

```
QScriptValue qsFilterCtor =
    interpreter.newFunction(pumpFilterConstructor,
                           qsFilterProto);
interpreter.globalObject().setProperty("PumpFilter", qsFilterCtor);
```

Мы регистрируем конструктор класса `PumpFilter`, чтобы создатели скриптов могли создавать экземпляры класса `PumpFilter`. «За кулисами» доступ к экземплярам `PumpFilter` опосредуется через `PumpFilterPrototype`.

Итак, подготовительная работа завершена. Мы считали скрипт в строку `QString` и настроили окружение скрипта, создав две глобальные переменные `spreadsheet` и `args`. Мы также сделали доступным мета-объект `PumpSpreadsheet` и обеспечили через оболочку доступ к экземплярам `PumpFilter`. Теперь все готово к выполнению скрипта.

```
interpreter.evaluate(script);
if (interpreter.hasUncaughtException()) {
    std::cerr << "Uncaught exception at line "
               << interpreter.uncaughtExceptionLineNumber() << ": "
               << qPrintable(interpreter.uncaughtException()
                              .toString())
               << std::endl << "Backtrace: "
               << qPrintable(interpreter.uncaughtExceptionBacktrace()
                              .join(", "))
               << std::endl;
    return false;
}
return true;
}
```

Как обычно, для запуска скрипта мы вызываем функцию `evaluate()`. Если есть синтаксические ошибки или другие проблемы, мы выводим соответствующую информацию об ошибках.

Теперь мы рассмотрим небольшую вспомогательную функцию `pumpFilterConstructor()` и более длинный (но простой) вспомогательный класс `PumpFilterPrototype`.

```
QScriptValue pumpFilterConstructor(QScriptContext * /* context */,
                                   QScriptEngine *interpreter)
{
    return interpreter->toScriptValue(PumpFilter());
}
```

Функция конструктора вызывается каждый раз, когда скрипт создает новый объект с помощью синтаксиса ECMAScript `new PumpFilter`. Доступ к аргументам, передаваемым конструктору, осуществляется через параметр `context`. Здесь мы их просто игнорируем и создаем объект по умолчанию `PumpFilter`, с оболочкой `QScriptValue`. Функция является шаблонной функцией, которая преобразует свой аргумент типа `T` в `QScriptValue`. Тип `T` (в нашем случае – `PumpFilter`) должен быть зарегистрирован с помощью `Q_DECLARE_METATYPE()`:

```
Q_DECLARE_METATYPE(PumpFilter)
```

А вот определение класса прототипа:

```
class PumpFilterPrototype : public QObject, public QScriptable
{
    Q_OBJECT
    Q_PROPERTY(QDate fromDate READ fromDate WRITE setFromDate)
    Q_PROPERTY(QDate toDate READ toDate WRITE setToDate)
    ...
    Q_PROPERTY(QString status READ status WRITE setStatus)
public:
    PumpFilterPrototype(QObject *parent = 0);
    void setFromDate(const QDate &date);
    QDate fromDate() const;
    void setToDate(const QDate &date);
    QDate toDate() const;
    ...
    void setStatus(const QString &status);
    QString status() const;
private:
    PumpFilter *wrappedFilter() const;
};
```

Класс-прототип является наследником и `QObject`, и `QScriptable`. Мы использовали функцию `Q_PROPERTY()` для каждой пары функций чтения/установки свойств. Обычно мы используем `Q_PROPERTY()` только для того, чтобы сделать свойства доступными для пользовательских классов виджетов, которые мы хотим интегрировать с *Qt Designer*, но их также можно использовать в контексте создания скриптов. Если мы хотим сделать функции доступными для скриптов, мы можем превратить их либо в открытые слоты, либо в свойства.

Все функции доступа к свойствам сходны, так что мы покажем только одну типичную пару:

```
void PumpFilterPrototype::setFromDate(const QDate &date)
{
    wrappedFilter()->fromDate = date;
}
QDate PumpFilterPrototype::fromDate() const
{
    return wrappedFilter()->fromDate;
}
```

А вот закрытая функция `wrappedFilter()`:

```
PumpFilter *PumpFilterPrototype::wrappedFilter() const
{
    return qscriptvalue_cast<PumpFilter *>(thisObject());
}
```

Функция `QScriptable::thisObject()` возвращает объект `this`, связанный с выполняемой интерпретатором в данный момент функцией. Объект возвращается в форме `QScriptValue`, и мы приводим его к типу `C++/Qt`, который он представляет, в данном случае – `PumpFilter *`. Приведение типов будет работать, только если мы зарегистрируем тип `PumpFilter *` при помощи `Q_DECLARE_METATYPE()`:

```
Q_DECLARE_METATYPE(PumpFilter *)
```

И, наконец, конструктор класса `PumpFilterPrototype`:

```
PumpFilterPrototype::PumpFilterPrototype(QObject *parent)
    : QObject(parent)
{
}
```

В данном примере мы не разрешаем создателям скриптов создавать собственные объекты класса `PumpSpreadsheet`. Вместо этого мы предоставляем глобальный объект-синглетон, `spreadsheet`, который они могут использовать. Чтобы создатели скриптов могли сами создавать экземпляры `PumpSpreadsheets`, нам нужно было бы зарегистрировать функцию `pumpSpreadsheetConstructor()`, как это было сделано для `PumpFilter`.

В примере приложения Бензонасос было достаточно предоставить скриптам доступ к виджетам приложения (например, к `PumpSpreadsheet`), а также к пользовательским классам данных, таким как `PumpFilter`. Хотя для примера приложения бензонасос это не обязательно, но иногда бывает также полезно сделать функции `C++` доступными для скриптов. Например, вот простая функция, определенная на `C++`, которую можно сделать доступной для скрипта:

```
QScriptValue square(QScriptContext *context, QScriptEngine *interpreter)
{
    double x = context->argument(0).toNumber();
    return QScriptValue(interpreter, x * x);
}
```

Сигнатура этой и других функций, ориентированных на использование в скрипте, всегда такова:

```
QScriptValue myFunc(QScriptContext *context, QScriptEngine *interpreter)
```

Доступ к аргументам функции осуществляется через функцию `QScriptContext::argument()`. Возвращаемое значение имеет тип `QScriptValue`, и мы создаем его при помощи объекта `QScriptEngine`, который мы передаем в качестве первого аргумента.

Следующий пример является более детальным:

```
QScriptValue sum(QScriptContext *context, QScriptEngine *interpreter)
{
    QScriptValue unaryFunc;
    int i = 0;
    if (context->argument(0).isFunction()) {
```

```

    unaryFunc = context->argument(0);
    i = 1;
}

double result = 0.0;
while (i < context->argumentCount()) {
    QScriptValue qsArg = context->argument(i);
    if (unaryFunc.isValid()) {
        QScriptValueList qsArgList;
        qsArgList << qsArg;
        qsArg = unaryFunc.call(QScriptValue(), qsArgList);
    }
    result += qsArg.toNumber();
    ++i;
}
return QScriptValue(interpreter, result);
}

```

Функция `sum()` может вызываться двумя разными способами. Простой способ – это вызов с передачей числовых аргументов. В данном случае функция `unaryFunc` будет недопустимым значением `QScriptValue`, и выполняться будет простое суммирование данных аргументов и возврат результата. Более хитрым методом будет вызов функции с передачей функции ECMAScript в качестве первого аргумента, а в качестве остальных аргументов – любого количества числовых аргументов. В данном случае указанная функция будет вызываться для каждого числа, и сумма результатов этих вызовов будет накапливаться и возвращаться. В первом разделе этой главы мы видели эту же функцию, написанную на ECMAScript. Использование C++ вместо ECMAScript для реализации низкоуровневой функциональности иногда может давать существенный прирост производительности.

Прежде чем мы сможем вызывать функции C++ из скрипта, мы должны сделать их доступными для интерпретатора с помощью функций `newFunction()` и `setProperty()`:

```

QScriptValue qsSquare = interpreter.newFunction(square);
interpreter.globalObject().setProperty("square", qsSquare);
QScriptValue qsSum = interpreter.newFunction(sum);
interpreter.globalObject().setProperty("sum", qsSum);

```

Мы сделали обе функции – `square()` и `sum()`, доступными для интерпретатора в виде глобальных функций. Теперь мы можем использовать их в скриптах, и это показано в следующем фрагменте:

```

interpreter.evaluate("print(sum(1, 2, 3, 4, 5, 6));");
interpreter.evaluate("print(sum(square, 1, 2, 3, 4, 5, 6));");

```

На этом завершается наше рассмотрение создания поддержки скриптов в Qt-приложениях с использованием модуля *QtScript*. Этот модуль оснащен обширной документацией, включающей широкий обзор и подробное описание предоставляемых классов, в том числе, `QScriptContext`, `QScriptEngine`, `QScriptValue` и `QScriptable`, которые все вполне заслуживают прочтения.



- *Применение «родных» программных интерфейсов*
- *Применение ActiveX в системе Windows*
- *Управление сеансами в системе X11*

Глава 23. Возможности, зависимые от платформы

В данной главе мы рассмотрим некоторые доступные программистам Qt возможности, которые зависят от платформы. Мы начнем с рассмотрения способов доступа к таким «родным» программным интерфейсам, как Win32 API в системе Windows, Carbon в системе Mac OS X и Xlib в системе X11. Затем мы перейдем к изучению расширения ActiveQt, демонстрируя способы применения элементов управления ActiveX в приложениях Qt, работающих в системе Windows, а также способы создания приложений, выполняющих функции серверов ActiveX. В последнем разделе мы рассмотрим способы взаимодействия приложений Qt с менеджером сеансов системы X11.

Кроме представленных здесь возможностей, компания Trolltech предлагает несколько зависимых от платформы решений в рамках проекта Qt Solutions, в частности миграционные фреймворки Qt/Motif и Qt/MFC, позволяющее упростить перевод в Qt приложений Motif/Xt и MFC. Подобное расширение для приложений Tcl/Tk обеспечивается фирмой «froglogic», а компанией Klarälvdalens Datakonsult разработан конвертор ресурсов Windows компании Microsoft. Дополнительную информацию вы найдете на следующих веб-страницах:

- <http://www.trolltech.com/products/qt/addon/solutions/catalog/4/>
- <http://www.froglogic.com/tq/>
- <http://www.kdab.net/knut/>

Для встроенных приложений компания Trolltech обеспечивает Qtopia – рабочую среду для разработки таких приложений, которая будет рассмотрена в главе 24.

Часть функциональности Qt, которая может зависеть от платформы, предоставляется для каждой платформы отдельно. Например, существует Qt Solution по созданию сервисов (демонов) в Windows, Unix и Mac OS X

Применение «родных» программных интерфейсов

Всесторонний программный интерфейс Qt удовлетворяет большинству требований на всех платформах, но при некоторых обстоятельствах нам может потребоваться базовый, платформозависимый программный интерфейс. В данном разделе мы продемонстрируем способы применения родных программных интерфейсов различных платформ, поддерживаемых Qt, для решения конкретных задач.

Для каждой платформы класс QWidget поддерживает функцию winId(), которая возвращает идентификатор или описатель окна. QWidget также обеспечивает статическую функцию find(), которая возвращает QWidget с идентификатором конкретного окна. Мы можем передавать этот идентификатор функциям родного программного интерфейса для достижения эффектов, зависящих от платформы. Например, в следующем программном коде используется функция winId() для отображения слева заголовка панели инструментов, используя родные функции Mac OS X (Carbon) (см. рис. 23.1):

```
#ifdef Q_WS_MAC
    ChangeWindowAttributes(HIViewGetWindow(HIViewRef(toolWin.winId())),
                          kWindowSideTitlebarAttribute,
                          kWindowNoAttributes);
#endif
```



Рис. 23.1. Окно панели инструментов Mac OS X с отображением заголовка сбоку

Ниже показано, как в системе X11 мы можем модифицировать свойство окна:

```
#ifdef Q_WS_X11
    Atom atom = XInternAtom(QX11Info::display(), "MY_PROPERTY", False);
    long data = 1;
    XChangeProperty(QX11Info::display(), window->winId(), atom, atom,
                  32, PropModeReplace,
                  reinterpret_cast<uchar *>(&data), 1);
#endif
```

Использование директив #ifdef и #endif вокруг зависимого от платформы программного кода гарантирует компиляцию приложения на других платформах.

Приведенный ниже пример показывает, как в приложениях, предназначенных только для Windows, можно использовать вызовы GDI для рисования на виджете Qt:

```
void GdiControl::paintEvent(QPaintEvent * /* event */)
{
    RECT rect;
    GetClientRect(winId(), &rect);
    HDC hdc = GetDC(winId());

    FillRect(hdc, &rect, HBRUSH(COLOR_WINDOW + 1));
    SetTextAlign(hdc, TA_CENTER | TA_BASELINE);
    TextOutW(hdc, width() / 2, height() / 2, text.utf16(), text.size());

    ReleaseDC(winId(), hdc);
}
```

Чтобы это сработало, мы должны также переопределить функцию `QPaintDevice::paintEngine()` для возврата нулевого указателя и установить атрибут `Qt::WA_PaintOnScreen` в конструкторе виджета.

Следующий пример показывает, как можно сочетать `QPainter` и `GDI` в обработчике события рисования, используя функции `getDC()` и `releaseDC()` класса `QPaintEngine`:

```
void MyWidget::paintEvent(QPaintEvent * /* event */)
{
    QPainter painter(this);
    painter.fillRect(rect().adjusted(20, 20, -20, -20), Qt::red);
#ifdef Q_WS_WIN
    HDC hdc = painter.paintEngine()->getDC();
    Rectangle(hdc, 40, 40, width() - 40, height() - 40);
    painter.paintEngine()->releaseDC();
#endif
}
```

Подобное совмещение вызовов `QPainter` и `GDI` иногда может дать странный результат, особенно, когда вызовы `QPainter` выполняются после вызовов `GDI`, потому что `QPainter` делает некоторые предположения о состоянии базового уровня рисования.

Qt определяет один из следующих четырех символов оконной системы: `Q_WS_WIN`, `Q_WS_X11`, `Q_WS_MAC` или `Q_WS_QWS` (**Qt**opia). Мы должны обеспечить включение хотя бы одного заголовка **Qt** перед их использованием в приложениях. **Qt** также обеспечивает препроцессорные символы для идентификации операционной системы:

• <code>Q_OS_AIX</code>	• <code>Q_OS_HPUX</code>	• <code>Q_OS_OPENBSD</code>	• <code>Q_OS_SOLARIS</code>
• <code>Q_OS_BSD4</code>	• <code>Q_OS_HURD</code>	• <code>Q_OS_OS2EMX</code>	• <code>Q_OS_ULTRIX</code>
• <code>Q_OS_BSDI</code>	• <code>Q_OS_IRIX</code>	• <code>Q_OS_OSF</code>	• <code>Q_OS_UNIXWARE</code>
• <code>Q_OS_CYGWIN</code>	• <code>Q_OS_LINUX</code>	• <code>Q_OS_QNX6</code>	• <code>Q_OS_WIN32</code>
• <code>Q_OS_DGUX</code>	• <code>Q_OS_LYNX</code>	• <code>Q_OS_QNX</code>	• <code>Q_OS_WIN64</code>
• <code>Q_OS_DYNIX</code>	• <code>Q_OS_MAC</code>	• <code>Q_OS_RELIANT</code>	
• <code>Q_OS_FREEBSD</code>	• <code>Q_OS_NETBSD</code>	• <code>Q_OS_SCO</code>	

Мы можем считать, что, по крайней мере, один из этих символов будет определен. Для удобства **Qt** также определяет `Q_OS_WIN`, когда обнаруживается Win32 или Win64, и `Q_OS_UNIX`, когда обнаруживается любая операционная система типа

Unix (включая Linux и Mac OS X). Во время выполнения приложений мы можем проверить `QSysInfo::WindowsVersion` или `QSysInfo::MacintoshVersion` для установ- ки отличий между различными версиями Windows (2000, ME и так далее) или Mac OS X (10.2, 10.3 и т. д.).

Кроме макросов операционной и оконной систем существует также ряд мак- росов компилятора. Например, `_Q_CC_MSVC` определяется в том случае, если ком- пилиатором является Visual C++ компании Microsoft. Такие макросы полезны, когда приходится обходить ошибки компилятора.

Несколько классов графического пользовательского интерфейса Qt обес- печивают зависящие от платформы функции, которые возвращают описатели (handle) базового объекта для низкоуровневой обработки. Они перечислены на рис. 23.2.

Mac OS X	
ATSTFontFormatRef	QFont::handle()
CGImageRef	QPixmap::macCGHandle()
GWorldPtr	QPixmap::macQDAlphaHandle()
GWorldPtr	QPixmap::macQDHandle()
RgnHandle	QRegion::handle()
HViewRef	QWidget::winId()

Windows	
HCURSOR	QCursor::handle()
HDC	QPaintEngine::getDC()
HDC	QPrintEngine::getPrinterDC()
HFONT	QFont::handle()
HPALETTE	QColorMap::hPal()
HRGN	QRegion::handle()
HWND	QWidget::winId()

X11	
Cursor	QCursor::handle()
Font	QFont::handle()
Picture	QPixmap::x11PictureHandle()
Picture	QWidget::x11PictureHandle()
Pixmap	QPixmap::handle()
QX11Info	QPixmap::x11Info()
QX11Info	QWidget::x11Info()
Region	QRegion::handle()
Screen	QCursor::x11Screen()
SmcConn	QSessionManager::handle()
Window	QWidget::handle()
Window	QWidget::winId()

Рис. 23.2. Зависящие от платформы функции доступа к низкоуровневым описателям

В системе X11 функции `QPixmap::x11Info()` и `QWidget::x11Info()` возвращают объект `QX11Info`, который обеспечивает различные указатели и описатели с помощью ряда функций, включая `display()`, `screen()`, `colormap()` и `visual()`. Мы можем использовать их для настройки графического контекста, например, `QWidget` или `QPixmap`.

Приложениям Qt, которым необходимо взаимодействовать с другими инструментальными средствами и библиотеками, часто приходится осуществлять доступ к низкоуровневым событиям (`XEvent` в системе X11, `MSG` в системе Windows, `Eventref` в системе Mac OS X, `QWSEvent` для QWS), прежде чем они будут преобразованы в события `QEvent`. Мы можем делать это путем создания подкласса `QApplication` и переопределения соответствующего зависимого от платформы фильтра событий, т. е. одной из следующих функций: `x11EventFilter()`, `winEventFilter()`, `macEventFilter()` и `qwsEventFilter()`. Мы можем поступать по-другому и осуществлять доступ к зависимым от платформы событиям, посылаемым заданному объекту `QWidget`, путем переопределения какой-то одной из функций `winEvent()`, `x11Event()`, `macEvent()` и `qwsEvent()`. Это может пригодиться для обработки событий определенного типа, которые Qt обычно игнорирует, например, событий джойстика.

Более подробную информацию относительно применения зависимых от платформы средств, в том числе о том, как разворачивать приложения Qt на различных платформах, можно найти в сети Интернет по адресу <http://doc.trolltech.com/4.3/winsystem.html>.

Применение ActiveX в системе Windows

Технология ActiveX компании Microsoft позволяет приложениям включать в себя компоненты интерфейса пользователя других приложений или библиотек. Она построена на применении технологии COM компании Microsoft и определяет один набор интерфейсов приложений, использующих компоненты, и другой набор интерфейсов приложений и библиотек, предоставляющих компоненты.



Рис. 23.3. Приложение Media Player

Версия Qt/Windows для настольных компьютеров (Desktop Edition) обеспечивает рабочую среду ActiveQt для «бесшовного соединения» ActiveX и Qt. ActiveQt состоит из двух модулей.

- Модуль *QAxContainer* позволяет нам использовать объекты COM и встраивать элементы управления ActiveX в приложения Qt.
- Модуль *QAxServer* позволяет нам экспортировать пользовательские объекты COM и элементы управления ActiveX, написанные с помощью средств разработки Qt.

Наш первый пример встраивает Media Player (медиаплеер) системы Windows в приложение Qt при помощи модуля *QAxContainer* (см. рис.23.3). Приложение Qt добавляет кнопку Open, кнопку Play/Pause, кнопку Stop и ползунок в элемент управления ActiveX Media Player системы Windows.

Главное окно приложения имеет тип *PlayerWindow*:

```
class PlayerWindow : public QWidget
{
    Q_OBJECT
    Q_ENUMS(ReadyStateConstants)

public:
    enum PlayStateConstants { Stopped = 0, Paused = 1, Playing = 2 };
    enum ReadyStateConstants { Uninitialized = 0, Loading = 1,
                               Interactive = 3, Complete = 4 };
    PlayerWindow();

protected:
    void timerEvent(QTimerEvent *event);

private slots:
    void onPlayStateChange(int oldState, int newState);
    void onReadyStateChange(ReadyStateConstants readyState);
    void onPositionChange(double oldPos, double newPos);
    void sliderValueChanged(int newValue);
    void openFile();

private:
    QAxWidget *wmp;
    QToolButton *openButton;
    QToolButton *playPauseButton;
    QToolButton *stopButton;
    QSlider *seekSlider;
    QString fileFilters;
    int updateTime;
};
```

Класс *PlayerWindow* является производным от *QWidget*. Макрос *Q_ENUMS()*, расположенный сразу после *Q_OBJECT*, необходим для указания компилятору *moc*, что константы *ReadyStateConstants*, используемые в слоте *onReadyStateChange()*, имеют тип *enum*. В закрытой секции мы объявляем переменную-член *QAxWidget **.

```
PlayerWindow::PlayerWindow()
{
    wmp = new QAxWidget;
    wmp->setControl("{22D6F312-B0F6-11D0-94AB-0080C74C7E95}");
}
```

Конструктор начинается с создания объекта `QAxWidget` для инкапсулирования элемента управления `ActiveX Media Player` системы `Windows`. Модуль `QAxContainer` состоит из трех классов: `QAxObject` – инкапсулирует объект `COM`, `QAxWidget` – инкапсулирует элемент управления `ActiveX` и `QAxBase` – реализует основную функциональность `COM` для `QAxObject` и `QAxWidget`. Связь между этими классами проиллюстрирована на рис. 23.4.

Мы вызываем функцию `setControl()` для объекта `QAxWidget` с идентификатором класса элемента управления `Media Player 6.4` системы `Windows`. Это создает экземпляр требуемого компонента. С этого момента все свойства, события и методы элемента управления `ActiveX` доступны как свойства, сигналы и методы `Qt` объекта `QAxWidget`.

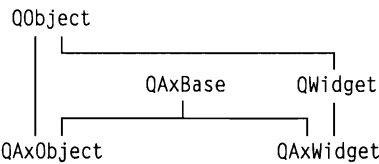


Рис. 23.4. Дерево наследования для модуля `QAxContainer`

Типы данных `COM` автоматически преобразуются в соответствующие типы объектов, как показано на рис. 23.5. Например, входной параметр типа `VARIANT_BOOL` становится типом `bool`, а выходной параметр типа `VARIANT_BOOL` становится типом `bool &`. Если полученный тип является классом `Qt` (`QString`, `QDateTime` и т. д.), входной параметр становится ссылкой с модификатором `const` (например, `const QString &`).

Типы COM	Тип Qt
VARIANT_BOOL	bool
char, short, int, long	int
unsigned char, unsigned short, unsigned int, unsigned long	uint
float, double	double
CY	qlonglong, qulonglong
BSTR	QString
DATE	QDateTime, QDate, QTime
OLE_COLOR	QColor
SAFEARRAY(VARIANT)	QList<QVariant>
SAFEARRAY(BSTR)	QStringList
SAFEARRAY(BYTE)	QByteArray
VARIANT	QVariant
IFontDisp *	QFont
IPictureDisp *	QPixmap
Тип, определяемый пользователем	QRect, QSize, QPoint

Рис. 23.5. Связь между типами `COM` и `Qt`

Для получения списка свойств, сигналов и слотов, доступных в объектах `QAxObject` или `QAxWidget` вместе с их типами `Qt`, сделайте вызов функции `QAxBase::generateDocumentation()` или используйте утилиту командной строки `Qt dumpdoc`, расположенную в каталоге `Qt tools\activeqt\dumpdoc`.

Теперь продолжим рассмотрение конструктора `PlayerWindow`:

```
wmp->setProperty("ShowControls", false);
wmp->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
connect(wmp, SIGNAL(PlayStateChange(int, int)),
        this, SLOT(onPlayStateChange(int, int)));
connect(wmp, SIGNAL(ReadyStateChange(ReadyStateConstants)),
        this, SLOT(onReadyStateChange(ReadyStateConstants)));
connect(wmp, SIGNAL(PositionChange(double, double)),
        this, SLOT(onPositionChange(double, double)));
```

После вызова `QAxWidget::setControl()` мы вызываем функцию `QObject::setProperty()` для установки свойства `ShowControls` (отображать элементы управления) элемента управления `Media Player` системы `Windows` на значение `false`, поскольку мы предоставляем свои собственные кнопки для работы с компонентом. Функция `QObject::setProperty()` может использоваться как для свойств `COM`, так и для обычных свойств `Qt`. Ее второй параметр имеет тип `QVariant`.

Затем мы вызываем функцию `setSizePolicy()`, чтобы элемент управления `ActiveX` мог занять все имеющееся в менеджере компоновки пространство, и мы подключаем три события `ActiveX` компонента `COM` к трем слотам.

```
...
stopButton = new QPushButton;
stopButton->setText(tr("&Stop"));
stopButton->setEnabled(false);
connect(stopButton, SIGNAL(clicked()), wmp, SLOT(Stop()));
...
}
```

Остальная часть конструктора `PlayerWindow` следует обычному образцу, за исключением того, что мы подключаем некоторые сигналы `Qt` к слотам объекта `COM` (`Play()`, `Pause()` и `Stop()`). Мы показали здесь реализацию только кнопки `Stop`, поскольку другие кнопки реализуются аналогично.

Давайте на этом закончим обсуждение конструктора и рассмотрим функцию `timerEvent()`:

```
void PlayerWindow::timerEvent(QTimerEvent *event)
{
    if (event->timerId() == updateTimer) {
        double curPos = wmp->property("CurrentPosition").toDouble();
        onPositionChange(-1, curPos);
    } else {
        QWidget::timerEvent(event);
    }
}
```

Функция `timerEvent()` вызывается через определенные интервалы времени во время проигрывания мультимедийного клипа. Мы используем ее для продвижения ползунка. Это делается путем вызова функции `property()` для элемента управления `ActiveX`, чтобы получить значение свойства `CurrentPosition` (текущая позиция) в виде объекта типа `QVariant` и вызова функции `toDouble()` для преобразования его в тип `double`. Мы затем вызываем функцию `onPositionChange()` для обновления положения ползунка.

Мы не будем рассматривать остальную часть программного кода, поскольку большая часть его не имеет непосредственного отношения к `ActiveX` и не содержит ничего такого, что мы уже не обсуждали ранее. Данный программный код включен в примеры книги.

В файле `.pro` нам необходимо задать элемент для связи с модулем `QAxContainer`:

```
CONFIG += qaxcontainer
```

При работе с объектами COM одной из часто возникающих потребностей является необходимость непосредственного вызова метода COM (вместо подсоединения его к сигналу Qt). Наиболее просто это сделать путем вызова функции `QAxBase::dynamicCall()` с указанием имени и сигнатуры метода в первом параметре и аргументов метода в дополнительных параметрах. Например:

```
wmp->dynamicCall("TitlePlay(uint)", 6);
```

Функция `dynamicCall()` принимает до восьми параметров типа `QVariant` и возвращает объект типа `QVariant`. Если нам необходимо передавать таким образом `IDispatch *` или `IUnknown *`, мы можем инкапсулировать компонент в `QAxObject` и вызвать для него функцию `asVariant()` для преобразования его в тип `QVariant`. Если нам необходимо вызвать метод COM, который возвращает `IDispatch *` или `IUnknown *`, или если нам необходимо осуществлять доступ к свойству COM одного из этих типов, мы можем вместо этого использовать функцию `querySubObject()`:

```
QAxObject *session = outlook.querySubObject("Session");
QAxObject *defaultContacts =
    session->querySubObject("GetDefaultFolder(0lDefaultFolders)",
                           "olFolderContacts");
```

Если мы собираемся вызывать методы, которые имеют неподдерживаемые типы данных в их списке параметров, мы можем использовать `QAxBase::queryInterface()` для получения интерфейса COM и непосредственного вызова метода. Мы должны вызвать функцию `Release()` после завершения использования интерфейса, что является обычным при работе с COM. Если нам приходится часто вызывать такие методы, мы можем создать подкласс `QAxObject` или `QAxWidget` и обеспечить функции-члены, которые инкапсулируют вызовы интерфейса COM. Однако убедитесь, что подклассы `QAxObject` и `QAxWidget` не могут определять свои собственные свойства, сигналы и слоты.

Теперь мы рассмотрим модуль `QAxServer`. Этот модуль позволяет нам превратить стандартную программу Qt в сервер `ActiveX`. Сервер может быть как совместно используемой библиотекой, так и автономным приложением. Серверы в виде совместно используемых библиотек часто называют внутрипроцессными

серверами (in-process servers), а автономные приложения – внепроцессными серверами (out-of-process servers).

Наш первый пример *QAxServer* является внутрипроцессным сервером, отображающим виджет с шариком, который может прыгать вправо и влево. Мы рассмотрим также способы встраивания этого виджета в Internet Explorer.

Ниже приводится начало определения класса виджета *AxBouncer*:

```
class AxBouncer : public QWidget, public QAxBindable
{
    Q_OBJECT
    Q_ENUMS(SpeedValue)
    Q_PROPERTY(QColor color READ color WRITE setColor)
    Q_PROPERTY(SpeedValue speed READ speed WRITE setSpeed)
    Q_PROPERTY(int radius READ radius WRITE setRadius)
    Q_PROPERTY(bool running READ isRunning)
```

AxBouncer, показанный на рис. 23.6, является производным от классов *QWidget* и *QAxBindable*. Класс *QAxBindable* обеспечивает интерфейс между виджетом и клиентом ActiveX. Любой *QWidget* может быть экспортирован как элемент управления ActiveX, но путем создания подкласса *QAxBindable* мы можем уведомлять клиента об изменениях значения свойства и реализовывать интерфейсы COM в дополнение к уже реализованным при помощи *QAxServer*.

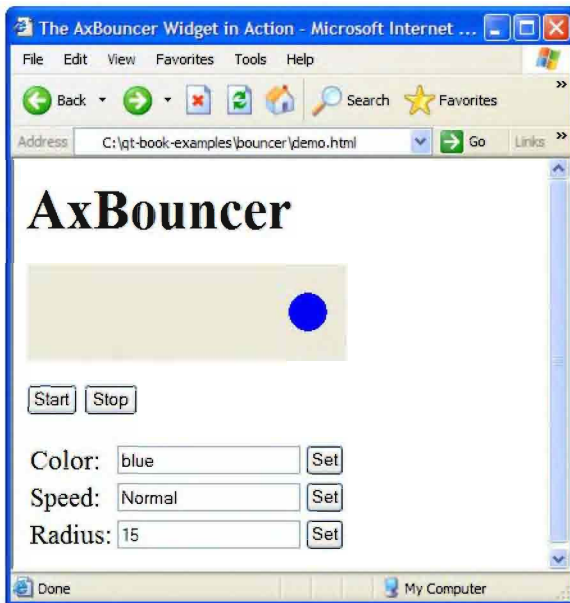


Рис. 23.6. Виджет *AxBouncer* в Internet Explorer

Если при использовании множественного наследования имеются классы, производные от *QObject*, мы должны всегда располагать производные от *QObject* классы первыми для того, чтобы компилятор мог их извлечь.

Мы объявляем три свойства для чтения и записи и одно свойство только для чтения. Макрос `Q_ENUMS()` необходим для указания компилятору `moc` на то, что `SpeedValue` имеет тип `enum` (перечисление). Это перечисление объявляется в открытой секции класса:

```
public:
    enum SpeedValue { Slow, Normal, Fast };

    AxBouncer(QWidget *parent = 0);

    void setSpeed(SpeedValue newSpeed);
    SpeedValue speed() const { return ballSpeed; }
    void setRadius(int newRadius);
    int radius() const { return ballRadius; }
    void setColor(const QColor &newColor);
    QColor color() const { return ballColor; }
    bool isRunning() const { return myTimerId != 0; }
    QSize sizeHint() const;
    QAxAggregated *createAggregate();

public slots:
    void start();
    void stop();

signals:
    void bouncing();
```

Конструктор `AxBouncer` является стандартным конструктором виджета с параметром `parent`. Макрос `QAXFACTORY_DEFAULT()`, который мы используем для экспорта компонента, предполагает, что у конструктора именно такая сигнатура.

Функция `createAggregate()` класса `QAxBindable` переопределяется. Мы рассмотрим ее вскоре.

```
protected:
    void paintEvent(QPaintEvent *event);
    void timerEvent(QTimerEvent *event);

private:
    int intervalInMilliseconds() const;

    QColor ballColor;
    SpeedValue ballSpeed;
    int ballRadius;
    int myTimerId;
    int x;
    int delta;
};
```

Защищенная и закрытая секции этого класса имеют тот же вид, как и для стандартного виджета `Qt`.

```

AxBouncer::AxBouncer(QWidget *parent)
    : QWidget(parent)
{
    ballColor = Qt::blue;
    ballSpeed = Normal;
    ballRadius = 15;
    myTimerId = 0;
    x = 20;
    delta = 2;
}

```

Конструктор `AxBouncer` инициализирует закрытые переменные этого класса.

```

void AxBouncer::setColor(const QColor &newColor)
{
    if (newColor != ballColor && requestPropertyChange("color")) {
        ballColor = newColor;
        update();
        propertyChanged("color");
    }
}

```

Функция `setColor()` устанавливает значение свойства `color` (цвет). Она вызывает функцию `update()` для перерисовки виджета.

Необычной частью являются вызовы функций `requestPropertyChange()` и `propertyChanged()`. Эти функции наследуются от класса `QAxBindable` и в идеальном случае должны вызываться при всяком изменении свойства. Функция `requestPropertyChange()` спрашивает у клиента разрешение на изменение свойства и возвращает `true`, если клиент дает такое разрешение. Функция `propertyChanged()` уведомляет клиента о том, что свойство изменилось.

Устанавливающие свойства функции `setSpeed()` и `setRadius()` следуют этому же образцу, и также работают слоты `start()` и `stop()`, поскольку они изменяют значение свойства `running` (приложение выполняется).

Осталось рассмотреть еще одну интересную функцию-член класса `AxBouncer`:

```

QAxAggregated *AxBouncer::createAggregate()
{
    return new ObjectSafetyImpl;
}

```

Функция `createAggregate()` класса `QAxBindable` переопределяется. Она позволяет нам реализовать интерфейсы COM, которые модуль `QAxServer` еще не реализовал, или обойти определенные по умолчанию в `QAxServer` интерфейсы COM. Ниже мы делаем это для обеспечения интерфейса `IObjectSafety`, который используется в Internet Explorer для доступа к свойствам безопасности компонента. Это является стандартным способом устранения непопулярного сообщения об ошибке «Object not safe for scripting» (объект небезопасен при использовании в сценарии) в Internet Explorer.

Ниже приводится определение класса, которое реализует интерфейс `IObjectSafety`:


```

class ObjectSafetyImpl : public QAxAggregated, public IObjectSafety
{
public:
    long queryInterface(const QUuid &iid, void **iface);

    QAXAGG_IUNKNOWN

    HRESULT WINAPI GetInterfaceSafetyOptions(REFIID riid,
        DWORD *pdwSupportedOptions, DWORD *pdwEnabledOptions);
    HRESULT WINAPI SetInterfaceSafetyOptions(REFIID riid,
        DWORD pdwSupportedOptions, DWORD pdwEnabledOptions);
};

```

Класс ObjectSafetyImpl является производным как от QAxAggregated, так и от IObjectSafety. Класс QAxAggregated является абстрактным базовым классом, предназначенным для реализации дополнительных интерфейсов COM. Объект COM, который расширяет QAxAggregated, доступен при помощи функции controllingUnknown(). Этот объект COM создается незаметно для пользователя модулем QAxServer.

Макрос QAXAGG_IUNKNOWN обеспечивает стандартную реализацию функций QueryInterface(), AddRef() и Release(). В этих реализациях просто делается вызов одноименных функций для управляющего объекта COM.

```

long ObjectSafetyImpl::queryInterface(const QUuid &iid, void **iface)
{
    *iface = 0;
    if (iid == IID_IObjectSafety){
        *iface = static_cast<IObjectSafety *>(this);
    } else {
        return E_NOINTERFACE;
    }

    AddRef();
    return S_OK;
}

```

Функция queryInterface() – чистая виртуальная функция класса QAxAggregated. Она вызывается управляющим объектом COM для предоставления доступа к интерфейсу, который обеспечивается подклассом QAxAggregated. Мы должны возвращать E_NOINTERFACE для интерфейсов, которые мы не определили, и также для IUnknown.

```

HRESULT WINAPI ObjectSafetyImpl::GetInterfaceSafetyOptions(
    REFIID /* riid */, DWORD *pdwSupportedOptions,
    DWORD *pdwEnabledOptions)
{
    *pdwSupportedOptions = INTERFACESAFE_FOR_UNTRUSTED_DATA
        | INTERFACESAFE_FOR_UNTRUSTED_CALLER;
    *pdwEnabledOptions = *pdwSupportedOptions;
}

```

```

    return S_OK;
}

HRESULT WINAPI ObjectSafetyImpl::SetInterfaceSafetyOptions(
    REFIID /* riid */, DWORD /* pdwSupportedOptions */,
    DWORD /* pdwEnabledOptions */)
{
    return S_OK;
}

```

Функции `GetInterfaceSafetyOptions()` и `SetInterfaceSafetyOptions()` **объявляются** в `IObjectSafety`. **Мы реализуем их, чтобы уведомить всех о том, что наш объект безопасен для использования в сценариях.**

Давайте теперь рассмотрим main.cpp:

```

#include <QAxFactory>

#include "axbouncer.h"

QAXFACTORY_DEFAULT(AxBouncer,
    "{5e2461aa-a3e8-4f7a-8b04-307459a4c08c}",
    "{533af11f-4899-43de-8b7f-2ddf588d1015}",
    "{772c14a5-a840-4023-b79d-19549ece0cd9}",
    "{dbce1e56-70dd-4f74-85e0-95c65d86254d}",
    "{3f3db5e0-78ff-4e35-8a5d-3d3b96c83e09}")

```

Макрос `QAXFACTORY_DEFAULT()` **экспортирует элемент управления ActiveX. Мы можем использовать его для серверов ActiveX, которые экспортируют только один элемент управления. В следующем примере данного раздела будет показано, как можно экспортировать много элементов управления ActiveX.**

Первым аргументом макроса `QAXFACTORY_DEFAULT()` **является имя экспортируемого класса Qt. Такое же имя используется для экспорта элемента управления. Остальные пять аргументов следующие: идентификатор класса, идентификатор интерфейса, идентификатор интерфейса событий, идентификатор библиотеки типов и идентификатор приложения. Мы можем использовать стандартные инструментальные средства, например, `guidgen` или `uuidgen`, для получения этих идентификаторов. Поскольку сервер реализован в виде библиотеки, нам не требуется иметь функцию `main()`.**

Ниже приводится файл .pro для внутрипроцессного сервера ActiveX:

```

TEMPLATE = lib
CONFIG += dll qaxserver
HEADERS = axbouncer.h \
          objectsafetyimpl.h
SOURCES = axbouncer.cpp \
          main.cpp \
          objectsafetyimpl.cpp
RC_FILE = qaxserver.rc
DEF_FILE = qaxserver.def

```

Файлы `qaxserver.rc` и `qaxserver.def`, на которые имеются ссылки в файле `.pro`, — стандартные файлы, которые можно скопировать из каталога `Qt src\activeqt\control`.

Файл `makefile` или сгенерированный утилитой `qmake` файл проекта Visual C++ содержат правила для регистрации сервера в реестре Windows. Для регистрации сервера на машине пользователя мы можем использовать утилиту `regsvr32`, которая имеется во всех системах Windows.

Мы можем затем включить компонент `Bouncer` в страницу HTML, используя тег `<object>`:

```
<object id="AxBouncer"
  classid="clsid:5e2461aa-a3e8-4f7a-8b04-307459a4c08c">
  <b>The ActiveX control is not available. Make sure you have built and
  registered the component server.</b>
</object>
```

Мы можем создать кнопку для вызова слотов:

```
<input type="button" value="Start" onClick="AxBouncer.start()">
<input type="button" value="Stop" onClick="AxBouncer.stop()">
```

Мы можем манипулировать виджетом при помощи языков JavaScript или VBScript точно так же, как и любым другим элементом управления ActiveX. См. включенный в примеры книги файл `demo.html`, содержащий очень простую страницу, в которой используется сервер ActiveX.

Наш последний пример — приложение `Address Book` (адресная книга), применяющее сценарий. Это приложение может рассматриваться в качестве стандартного приложения Qt для Windows или внепроцессного сервера ActiveX. В последнем случае мы можем создавать сценарий работы приложения, используя, например, Visual Basic.

```
class AddressBook : public QMainWindow
{
    Q_OBJECT
    Q_PROPERTY(int count READ count)
    Q_CLASSINFO("ClassID", "{588141ef-110d-4beb-95ab-ee6a478b576d}")
    Q_CLASSINFO("InterfaceID", "{718780ec-b30c-4d88-83b3-79b3d9e78502}")
    Q_CLASSINFO("ToSuperClass", "AddressBook")

public:
    AddressBook(QWidget *parent = 0);
    ~AddressBook();

    int count() const;

public slots:
    ABItem *createEntry(const QString &contact);
    ABItem *findEntry(const QString &contact) const;
    ABItem *entryAt(int index) const;
```

```
private slots:
    void addEntry();
    void editEntry();
    void deleteEntry();

private:
    void createActions();
    void createMenus();

    QTreeWidget *treeWidget;
    QMenu *fileMenu;
    QMenu *editMenu;
    QAction *exitAction;
    QAction *addEntryAction;
    QAction *editEntryAction;
    QAction *deleteEntryAction;
};
```

Виджет `AddressBook` является главным окном приложения. Предоставляемые им свойства и открытые слоты можно применять при создании сценария. Макрос `Q_CLASSINFO()` используется для определения идентификаторов класса и интерфейсов, связанных с классом. Они генерируются с помощью таких утилит, как `guid` или `uuid`.

В предыдущем примере мы определяли идентификаторы класса и интерфейса при экспорте класса `QAxBouncer`, используя макрос `QAXFACTORY_DEFAULT()`. В этом примере мы хотим экспортировать несколько классов, поэтому нельзя использовать макрос `QAXFACTORY_DEFAULT()`. Мы можем поступать двумя способами:

- можно создать подкласс `QAxFactory`, переопределить его виртуальные функции для представления информации об экспортируемых нами типах и использовать макрос `QAXFACTORY_EXPORT()` для регистрации фабрики классов;
- можно использовать макросы `QAXFACTORY_BEGIN()`, `QAXFACTORY_END()`, `QAXCLASS()` и `QAXTYPE()` для объявления и регистрации фабрики классов. В этом случае потребуется использовать макрос `Q_CLASSINFO()` для определения идентификаторов класса и интерфейса.

Вернемся к определению класса `AddressBook`. Третий вызов макроса `Q_CLASSINFO()` может показаться немного странным. По умолчанию элементы управления `ActiveX` предоставляют в распоряжение клиентов не только свои собственные свойства, сигналы и слоты, но и свои базовые классы вплоть до `QWidget`. Атрибут `ToSuperClass` позволяет определить базовый класс самого высокого уровня в дереве наследования, который мы собираемся предоставить клиенту. Здесь мы указываем имя класса компонента (`AddressBook`) в качестве имени экспортируемого класса самого высокого уровня – это значит, что не будут экспортироваться свойства, сигналы и слоты, унаследованные от базовых классов `AddressBook`.

```
class ABItem : public QObject, public QListViewItem
{
```

```

Q_OBJECT
Q_PROPERTY(QString contact READ contact WRITE setContact)
Q_PROPERTY(QString address READ address WRITE setAddress)
Q_PROPERTY(QString phoneNumber READ phoneNumber WRITE setPhoneNumber)
Q_CLASSINFO("ClassID", "{bc82730e-5f39-4e5c-96be-461c2cd0d282}")
Q_CLASSINFO("InterfaceID", "{c8bc1656-870e-48a9-9937-fbe1ceff8b2e}")
Q_CLASSINFO("ToSuperClass", "ABItem")
public:
ABItem(QTreeWidget *treeWidget);

void setContact(const QString &contact);
QString contact() const { return text(0); }
void setAddress(const QString &address);
QString address() const { return text(1); }
void setPhoneNumber(const QString &number);
QString phoneNumber() const { return text(2); }

public slots:
    void remove();
};

```

Класс `ABItem` представляет один элемент в адресной книге. Он является производным от класса `QTreeWidgetItem` и поэтому может отображаться в `QTreeWidget`, и он также наследует `QObject` и поэтому может экспортироваться как объект COM.

```

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    if (!QAxFactory::isServer()) {
        AddressBook addressBook;
        addressBook.show();
        return app.exec();
    }
    return app.exec();
}

```

В функции `main()` мы проверяем, в каком качестве работает приложение: как автономное приложение или как сервер. Опция командной строки `-activex` распознается объектом `QApplication` и обеспечивает работу приложения в качестве сервера. Если приложение не является сервером, мы создаем главный виджет и выводим его на экран, как мы обычно делаем для любого автономного приложения Qt.

Кроме опции `-activex` серверы ActiveX «понимают» следующие опции командной строки:

- `regserver` – регистрация сервера в системном реестре;
- `unregserver` – отмена регистрации сервера в системном реестре;
- `dumpidl файл` – записывает описание сервера на языке IDL (Interface Description Language – язык описания интерфейсов) в указанный файл.

Когда приложение выполняет функции сервера, нам необходимо экспортировать классы `AddressBook` и `ABItem` как компоненты COM:

```
QAXFACTORY_BEGIN("{2b2b6f3e-86cf-4c49-9df5-80483b47f17b}",
                  "{8e827b25-148b-4307-ba7d-23f275244818}")
QAXCLASS(AddressBook)
QAXTYPE(ABItem)
QAXFACTORY_END()
```

Приведенные выше макросы экспортируют фабрику классов для создания объектов COM. Поскольку мы собираемся экспортировать два типа объектов COM, мы не можем просто использовать макрос `QAXFACTORY_DEFAULT()`, как мы делали в предыдущем примере.

Первым аргументом макроса `QAXFACTORY_BEGIN()` является идентификатор библиотеки типов; второй аргумент представляет собой идентификатор приложения. Между макросами `QAXFACTORY_BEGIN()` и `QAXFACTORY_END()` мы указываем все классы, которые могут быть инстанцированы, и все типы данных, доступные как объекты COM.

Ниже приводится файл `.pro` для внепроцессного сервера `ActiveX`:

```
TEMPLATE      = app
CONFIG        += qaxserver
HEADERS        = abitem.h \
                  addressbook.h \
                  editdialog.h
SOURCES        = abitem.cpp \
                  addressbook.cpp \
                  editdialog.cpp \
                  main.cpp
FORMS          = editdialog.ui
RC_FILE        = qaxserver.rc
```

Файл `qaxserver.rc`, на который имеется ссылка в файле `.pro`, является стандартным файлом, который может быть скопирован из каталога `Qt src\activeqt\control`.

Вы можете посмотреть в каталоге примеров `vb` проект `Visual Basic`, который использует сервер `Address Book`.

Этим мы завершаем наш обзор рабочей среды `ActiveQt`. Дистрибутив `Qt` включает дополнительные примеры, и в документации содержится информация о способах построения модулей `QAxContainer` и `QAxServer` и решения обычных вопросов взаимодействия.

Управление сеансами в системе X11

Когда мы выходим из системы `X11`, некоторые оконные менеджеры спрашивают нас о необходимости сохранения сеанса. Если мы отвечаем утвердительно, то при следующем входе в систему работа приложений будет автоматически возобновлена с того же экрана и, в идеальном случае, с того же состояния, которое было во время выхода из системы. Пример этого показан на рис. 23.7.

Компонент системы X11, который обеспечивает сохранение и восстановление сеанса, называется *менеджером сеансов (session manager)*. Для того чтобы приложение Qt/X11 осознавало присутствие менеджера сеансов, мы должны переопределить функцию `QApplication::saveState()` и сохранить там состояние приложения.



Рис. 23.7. Выход из системы KDE

Windows компании Microsoft, а также некоторые системы Unix предлагают другой механизм, который носит название «спящих процессов» (hibernation). Когда пользователь останавливает компьютер, операционная система просто выгружает оперативную память компьютера на диск и загружает ее обратно, когда компьютер «просыпается». Приложениям ничего не надо делать, и они даже могут ничего не знать об этом.

Когда пользователь инициирует завершение работы, мы можем перехватить управление непосредственно перед завершением путем переопределения функции `QApplication::commitData()`. Это позволяет нам сохранять измененные данные и при необходимости вступать в диалог с пользователем. Эта часть схемы управления сеансом поддерживается как в системе X11, так и в Windows.

Мы рассмотрим управление сеансом через программный код показанного на рис. 23.8 приложения Tic-Tac-Toe (крестики-нолики), которое работает под управлением менеджера сеансов. Во-первых, давайте рассмотрим функцию `main()`:

```
int main(int argc, char *argv[])
{
    Application app(argc, argv);
    TicTacToe toe;
    toe.setObjectName("toe");
    app.setTicTacToe(&toe);
    toe.show();
    return app.exec();
}
```

Мы создаем объект `Application`. Класс `Application` является производным от `QApplication` и переопределяет две функции `commitData()` и `saveState()` для обеспечения управления сеансом.

Затем мы создаем виджет `TicTacToe`, даем знать об этом объекту `Application` и отображаем его. Мы дали виджету `TicTacToe` имя «toe». Мы должны давать уникальные имена виджетам верхнего уровня, если мы хотим, чтобы менеджер сеансов мог восстановить размеры и позиции окон.

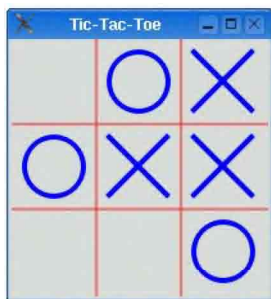


Рис. 23.8. Приложение Tic-Tac-Toe

Ниже приводится определение класса `Application`:

```
class Application : public QApplication
{
    Q_OBJECT
public:
    Application(int &argc, char *argv[]);

    void setTicTacToe(TicTacToe *toe);
    void saveState(QSessionManager &sessionManager);
    void commitData(QSessionManager &sessionManager);

private:
    TicTacToe *ticTacToe;
};
```

Класс `Application` сохраняет указатель виджета `TicTacToe` в закрытой переменной.

```
void Application::saveState(QSessionManager &sessionManager)
{
    QString fileName = ticTacToe->saveState();

    QStringList discardCommand;
    discardCommand << "rm" << fileName;
    sessionManager.setDiscardCommand(discardCommand);
}
```

В системе X11 функция `saveState()` вызывается, когда менеджер сеансов собирается сохранить состояние приложения. Данная функция также имеется на других платформах, но никогда не вызывается. Параметр `QSessionManager` позволяет нам поддерживать связь с менеджером сеансов.

Мы начинаем с попытки сохранения виджетом `TicTacToe` своего состояния в некоторый файл. Затем мы задаем команду для выполнения сброса состояния менеджером сеансов. Команда сброса (*discard command*) – это команда, которую должен выполнять менеджер сеансов для удаления любой сохраненной ранее информации, связанной с текущим состоянием. В этом примере мы задаем ее в виде

```
rm файл_сеанса
```

где `файл_сеанса` – имя файла, который содержит сохраненное состояние сеанса, а `rm` – стандартная команда удаления файлов в системе Unix.

Менеджер сеансов имеет также команду рестарта (*restart command*). Эту команду менеджер сеансов должен выполнять для возобновления работы приложения. По умолчанию Qt обеспечивает следующую команду рестарта:

```
приложение -session идентификатор_ключа
```

Первая часть, *приложение*, извлекается из `argv[0]`. Идентификатор – это идентификатор сеанса, переданный менеджером сеансов; гарантированно обеспечивается его уникальность для различных приложений и различных сеансов работы одного приложения. Ключ добавляется для однозначной идентификации времени сохранения сеанса. По различным причинам менеджер сеансов может вызывать функцию `saveState()` несколько раз в одном сеансе, и различные состояния должны отличаться.

Из-за ограничений существующих менеджеров сеансов нам необходимо убедиться, что каталог приложения содержится в переменной среды `PATH`, если мы хотим обеспечить правильный рестарт приложения. В частности, если вы сами собираетесь попробовать пример `Tic-Tac-Toe`, вы должны установить его в каталог, например, `/usr/local/bin` и вызывать его по команде `tictactoe`.

Для простых приложений, в том числе и для `Tic-Tac-Toe`, мы могли бы для обеспечения команды рестарта сохранять состояние в дополнительном аргументе командной строки. Например:

```
tictactoe -state 0X-X0-X-0
```

Это избавило бы нас от сохранения данных в файле и выдачи команды сброса состояния для удаления файла.

```
void Application::commitData(QSessionManager &sessionManager)
{
    if (ticTacToe->gameInProgress()
        && sessionManager.allowsInteraction()) {
        int r = QMessageBox::warning(ticTacToe, tr("Tic-Tac-Toe"),
                                     tr("The game hasn't finished.\n"
                                         "Do you really want to quit?"),
                                     QMessageBox::Yes | QMessageBox::No);

        if (r == QMessageBox::Yes) {
            sessionManager.release();
        } else {
            sessionManager.cancel();
        }
    }
}
```

Функция `commitData()` вызывается, когда пользователь выходит из системы. Мы можем переопределить ее для вывода сообщения, предупреждающего пользователя о потенциальной потере данных. В используемой по умолчанию реализации закрываются все виджеты верхнего уровня, что равносильно ситуации, когда пользователь последовательно закрывает все окна, нажимая кнопку закрытия в заголовках окон. В главе 3 мы показали, как можно переопределять функцию `closeEvent()`, перехватывающую этот момент и выводящую на экран сообщение.

В нашем примере мы переопределяем функцию `commitData()` и выводим сообщение, спрашивая у пользователя подтверждение выхода из системы, если работа в принципе может продолжаться и если менеджер сеансов позволяет нам взаимодействовать с пользователем (рис. 23.9). Если пользователь нажимает клавишу **Yes**, мы вызываем функцию `release()` для указания менеджеру на необходимость завершения операции выхода из системы; если пользователь нажимает клавишу **No**, мы вызываем функцию `cancel()` для отмены выхода из системы.



Рис. 23.9. «Вы действительно хотите завершить работу?»

Теперь давайте рассмотрим класс `TicTacToe`:

```
class TicTacToe : public QWidget
{
    Q_OBJECT
public:
    TicTacToe(QWidget *parent = 0);

    bool gameInProgress() const;
    QString saveState() const;
    QSize sizeHint() const;

protected:
    void paintEvent(QPaintEvent *event);
    void mousePressEvent(QMouseEvent *event);

private:
    enum { Empty = '-', Cross = 'X', Nought = 'O' };

    void clearBoard();
    void restoreState();
    QString sessionFileName() const;
    QRect cellRect(int row, int column) const;
    int cellWidth() const { return width() / 3; }
```

```

int cellHeight() const { return height() / 3; }
bool threeInARow(int row1, int col1, int row3, int col3) const;

char board[3][3];
int turnNumber;
};

```

Класс TicTacToe является производным от QWidget и переопределяет функции sizeHint(), paintEvent() и mousePressEvent(). Он также обеспечивает функции gameInProgress() и saveState(), которые мы использовали в нашем классе Application.

```

TicTacToe::TicTacToe(QWidget *parent, const char *name)
    : QWidget(parent, name)
{
    clearBoard();
    if (qApp->isSessionRestored())
        restoreState();

    setWindowTitle(tr("Tic-Tac-Toe"));
}

```

В конструкторе мы стираем игровое поле и, если приложение было вызвано с опцией -session, вызываем закрытую функцию restoreState() для восстановления старого сеанса.

```

void TicTacToe::clearBoard()
{
    for (int row = 0; row < 3; ++row) {
        for (int column = 0; column < 3; ++column) {
            board[row][column] = Empty;
        }
    }
    turnNumber = 0;
}

```

В функции clearBoard() мы стираем все ячейки и устанавливаем turnNumber на значение 0.

```

QString TicTacToe::saveState() const
{
    QFile file(sessionFileName());
    if (file.open(QIODevice::WriteOnly)) {
        QTextStream out(&file);
        for (int row = 0; row < 3; ++row) {
            for (int column = 0; column < 3; ++column) {
                out << board[row][column];
            }
        }
    }
    return file.fileName();
}

```

В функции `saveState()` мы записываем состояние игрового поля на диск. Формат достаточно простой: «X» для крестиков, «O» для ноликов и «-» для пустых ячеек.

```
QString TicTacToe::sessionFileName() const
{
    return QDir::homePath() + "/.tictactoe_" + qApp->sessionId() + "_"
        + qApp->sessionKey();
}
```

Закрывающая функция `sessionFileName()` возвращает имя файла для текущего идентификатора сеанса и ключа сеанса. Данная функция используется как в `saveState()`, так и в `restoreState()`. Имя файла определяется на основе идентификатора сеанса и ключа сеанса.

```
void TicTacToe::restoreState()
{
    QFile file(sessionFileName());
    if (file.open(QIODevice::ReadOnly)) {
        QTextStream in(&file);
        for (int row = 0; row < 3; ++row) {
            for (int column = 0; column < 3; ++column) {
                in >> board[row][column];
                if (board[row][column] != Empty)
                    ++turnNumber;
            }
        }
        update();
    }
}
```

В функции `restoreState()` мы загружаем файл восстанавливаемого сеанса и заполняем игровое поле его информацией. Мы рассчитываем значение переменной `turnNumber` исходя из количества крестиков и ноликов на игровом поле.

В конструкторе `TicTacToe` мы вызывали `restoreState()`, если функция `QApplication::isSessionRestored()` возвращала `true`. В этом случае `sessionId()` и `sessionKey()` возвращают именно те значения, которые были при прошлом сохранении состояния приложения, а функция `sessionFileName()` возвращает имя файла того сеанса.

Тестирование и отладка программного кода по управлению сеансами может быть достаточно утомительным делом, поскольку нам приходится все время входить и выходить из системы. Один из способов, позволяющий избежать этого, заключается в применении стандартной утилиты `xsm`, предусмотренной в системе X11. При первом вызове `xsm` на экран выводится окно менеджера сеансов и окно консольного режима. Все приложения, запускаемые с данного окна консольного режима, будут использовать `xsm` в качестве своего менеджера сеансов, а не стандартный общесистемный менеджер сеансов. Мы можем затем использовать окно `xsm` для завершения, рестарта или сброса сеанса и проконтролировать правильность поведения приложения. Подробное описание того, как это делается, вы найдете в сети Интернет по адресу <http://doc.trolltech.com/4.3/session.html>.



- *Первое знакомство с Qt/Embedded Linux*
- *Настройка Qt/Embedded Linux*
- *Интеграция приложений Qt при помощи Qtopia*
- *Использование API Qtopia*

Глава 24. Программирование встроенных систем

Разработка программного обеспечения для таких мобильных устройств, как карманные компьютеры и мобильные телефоны, может представлять собой очень сложную задачу, поскольку встроенные системы обычно имеют более медленные процессоры, меньший объем постоянной памяти (на флеш-картах или на жестких дисках), меньший объем основной памяти и дисплеи меньшего размера, чем настольные компьютеры.

Система Qt/Embedded Linux (ее называют также Qtopia Core) – это версия Qt, оптимизированная для разработки встроенных систем под Linux. Qt/Embedded Linux имеет такие же утилиты и программный интерфейс, какие предусмотрены в версиях Qt для настольных компьютеров (Qt/Windows, Qt/X11 и Qt/Mac), а также дополнительно предлагает классы и утилиты, необходимые для программирования встроенных систем. Через двойное лицензирование эта система доступна как для разработок с открытым исходным кодом, так и для коммерческих разработок.

Qt/Embedded Linux может работать на любом оборудовании, функционирующем под управлением Linux, включая архитектуры Intel x86, MIPS, ARM, StrongARM, Motorola/Freescale 68000 и PowerPC.¹ В отличие от Qt/X11 она не нуждается в системе X Window; вместо этого в ней реализуется собственная оконная система QWS, которая приводит к значительной экономии постоянной и основной памяти. Для еще большего уменьшения расхода памяти можно перекомпилировать Qt/Embedded Linux и исключить неиспользуемые возможности. Если заранее известны используемые устройством приложения и компоненты, они могут быть скомпилированы совместно в один исполняемый модуль и собраны статически с библиотеками Qt/Embedded Linux.

Кроме того, Qt/Embedded Linux использует преимущества многих функций, присущих также версиям Qt для настольных компьютеров, в частности широко применяется неявное совместное использование данных («копирование при

¹ Ожидается, что, начиная с версии 4.4, Qt будет работать также в Windows CE.

записи») как метод экономии основной памяти, поддерживаются пользовательские стили виджетов с помощью класса `QStyle` и обеспечивается система компоновки виджетов, позволяющая максимально использовать пространство экрана.

Qt/Embedded Linux представляет собой базовый компонент, на котором строятся другие предложения по встроенным системам компании Trolltech; к ним относятся Qtopia Platform, Qtopia PDA и Qtopia Phone. Они содержат классы и приложения, специально предназначенные для мобильных устройств и способные интегрироваться с некоторыми виртуальными машинами Java независимых разработчиков.

Первое знакомство с Qt/Embedded Linux

Приложения Qt/Embedded Linux могут разрабатываться на любой платформе, позволяющей запускать соответствующие цепочки инструментальных средств. Наиболее распространено построение кросс-компилятора GNU C++ в системе Unix. Этот процесс упрощается, благодаря наличию скрипта и набора пакетов обновлений Дана Кегеля (Dan Kegel), доступного на веб-странице <http://kegel.com/crosstool/>. В этой главе мы использовали версию 4.2 the Qtopia Open Source Edition, которую можно получить в Интернет по адресу <http://www.trolltech.com/products/qtopia/opensource/>. Версия этого издания предназначена только для Linux и содержит свою собственную копию Qt/Embedded Linux 4.2 вместе с дополнительными инструментами по поддержке Qtopia-программирования на настольном компьютере.

Система конфигурации Qt/Embedded Linux поддерживает кросс-компиляцию с помощью опций `-embedded` и `-xplatform` скрипта `configure`. Например, для построения ARM-архитектуры мы могли бы ввести команду

```
./configure -embedded arm -xplatform qws/linux-arm-g++
```

Можно создавать пользовательские конфигурации путем добавления новых файлов в каталог `Qt mkspecs/qws`.

Qt/Embedded Linux рисует непосредственно в буфере фреймов системы Linux (область основной памяти, связанная с дисплеем). Виртуальный буфер фреймов, показанный на рис. 24.1, является приложением X11, которое моделирует (пиксель за пикселем) реальный буфер фреймов. Для обращения к буферу фреймов, возможно, потребуется получить разрешение для записи на устройство `/dev/fb0`.

Для выполнения приложений Qt/Embedded Linux сначала необходимо запустить один процесс, выполняющий функции сервера графического интерфейса. Этот сервер отвечает за распределение между клиентами областей экрана и за генерацию событий мышки и клавиатуры. Любое приложение Qt/Embedded Linux может стать сервером, если в командной строке указать опцию `-qws` или в качестве третьего параметра конструктора `QApplication` передать `QApplication::GuiServer`.

Клиентские приложения связываются с сервером Qt/Embedded Linux при помощи совместно используемой области в основной памяти и Unix-конвейера. Внутренне операции рисования реализованы так, что клиенты рисуют самих себя в буфере фреймов Linux и отвечают за оформление собственных окон.

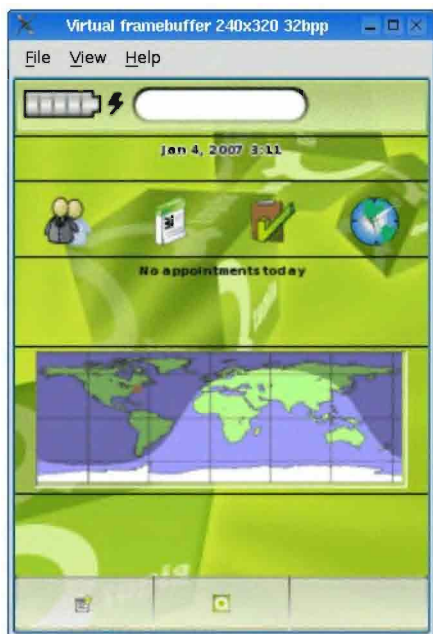


Рис. 24.1. Система Qt/Embedded Linux, работающая в виртуальном буфере фреймов

Клиенты могут связываться друг с другом при помощи протокола QCop (Qt Communication Protocol – Qt-протокол передачи данных). Клиент может прослушивать именованный канал, создавая объект QCopChannel и устанавливая связь с его сигналом received(). Например:

```
QCopChannel *channel = new QCopChannel("System", this);
connect(channel, SIGNAL(received(const QString &, const QByteArray &)),
        this, SLOT(received(const QString &, const QByteArray &)));
```

Сообщение QCop состоит из имени и необязательного массива QByteArray. Статическая функция QCopChannel::send() передает в широковещательном режиме сообщение по каналу. Например:

```
QByteArray data;
QDataStream out(&data, QIODevice::WriteOnly);
out << QDateTime::currentDateTime();
QCopChannel::send("System", "clockSkew(QDateTime)", data);
```

Предыдущий пример иллюстрирует общий прием: для кодирования данных используется поток QDataStream и для гарантирования правильной интерпретации получателем массива QByteArray формат данных в сообщении принимает вид функции C++.

На работу приложений Qt/Embedded Linux влияют различные переменные среды. Наиболее важными являются QWS_MOUSE_PROTO и QWS_KEYBOARD, которые определяют тип устройства мышки и клавиатуры. Полный список переменных среды приводится на веб-странице <http://doc.trolltech.com/4.2/qtapiacore-env-vars.html>.

Если в качестве платформы разработки используется Unix, приложение можно тестировать с использованием виртуального буфера фреймов Qt (qvfb) – приложения X11, которое имитирует реальный буфер фреймов. Это значительно сокращает цикл разработки. Для включения поддержки в Qt/Embedded Linux виртуального буфера необходимо передать опцию -qvfb скрипту configure. Следует помнить, что эта опция не предназначена для промышленного применения. Приложение виртуального буфера фреймов располагается в каталоге tools/qvfb и может вызываться следующим образом:

```
qvfb -width 320 -height 480 -depth 32
```

Альтернативой применения виртуального фрейма буферов X11 является VNC (Virtual Network Computing – вычисление в виртуальной сети), которое используется для удаленного выполнения приложения. Для включения поддержки VNC Qt/Embedded Linux передайте опцию -qt-gfx-vnc в скрипт configure. Затем запустите ваше приложение Qt/Embedded Linux с опцией командной строки -display VNC:0 и клиента VNC, ссылающегося на хост, на котором выполняется ваше приложение. Размер экрана и разрядность цвета можно установить с помощью переменных среды QWS_SIZE и QWS_DEPTH на хосте, на котором выполняются приложения Qt/Embedded Linux (например, QWS_SIZE=320x480 и QWS_DEPTH=32).

Настройка Qt/Embedded Linux

При установке Qt/Embedded Linux можно указать функции, которые мы хотим устранить, чтобы снизить расход памяти. В состав Qt/Embedded Linux входит более сотни конфигурируемых функций, каждой из которых соответствует какой-то препроцессорный символ. Например, QT_NO_FILEDIALOG исключает класс QFileDialog из библиотеки *QtGui*, а QT_NO_I18N удаляет всю поддержку интернационализации. Эти функции перечислены в файле src/corelib/qfeatures.txt.

Qt/Embedded Linux содержит пять примеров конфигурации (minimum, small, medium, large и dist), которые находятся в файлах src/corelib/global/qconfig_xxx.h. Эти конфигурации можно задавать, используя опции -qconfig xxx для скрипта configure, например:

```
./configure -qconfig small
```

Для создания пользовательских конфигураций можно вручную создать файл qconfig-xxx.h и использовать его, как будто он определяет стандартную конфигурацию. Можно поступить по-другому и использовать графическую утилиту qconfig, расположенную в подкаталоге Qt tools.

Qt/Embedded Linux предоставляет следующие классы для интерфейса с входными и выходными устройствами и для настройки пользовательского интерфейса оконной системы:

Класс	Базовый класс для
QScreen	драйверов экрана
QScreenDriverPlugin	подключаемых модулей драйверов экрана
QWSMouseHandler	драйверов мышки
QMouseDriverPlugin	подключаемых модулей драйверов мышки
QWSKeyboardHandler	драйверов клавиатуры

Класс	Базовый класс для
QKbdDriverPlugin	подключаемых модулей драйверов клавиатуры
QWSInputMethod	методов ввода
QDecoration	стилей оформления окон
QDecorationPlugin	подключаемых модулей стилей оформления окон

Для получения списка заранее определенных драйверов, методов ввода и стилей оформления экрана запустите скрипт `configure` с опцией `-help`.

Драйвер экрана можно задать с помощью опции командной строки `-display` при запуске сервера `Qt/Embedded Linux`, как это было показано в предыдущем разделе, или путем установки переменной среды `QWS_DISPLAY`. Драйвер мышки и связанное с ним устройство можно задавать, используя переменную среды `QWS_MOUSE_PROTO`, значение которой задается в виде *тип:устройство*, где *тип* – один из поддерживаемых драйверов, а *устройство* – путь к устройству (например, `QWS_MOUSE_PROTO=IntelliMouse:/dev/mouse`). Клавиатуры задаются аналогично при помощи переменной среды `QWS_KEYBOARD`. Методы ввода и оформления окон устанавливаются программно в сервере при помощи функций `QWSServer::setCurrentInputMethod()` и `QApplication::qwsSetDecoration()`.

Стили оформления окон можно задавать отдельно от стиля виджетов, который инкапсулирован в подклассе `QStyle`. Например, вполне допускается установить `Windows` в качестве стиля оформления окон и `Plastique` в качестве стиля виджетов. При желании для каждого окна можно задавать свой стиль оформления.

Класс `QWSServer` содержит различные функции по настройке оконной системы. Приложения, функционирующие как сервер `Qt/Embedded Linux`, могут получать доступ к уникальному экземпляру `QWSServer` через статическую функцию `QWSServer::instance()`.

`Qt/Embedded Linux` поддерживает следующие форматы шрифтов: `TrueType` (TTF), `PostScript Type 1`, `Bitmap Distribution Format` (BDF) и `Qt Pre-rendered Fonts` (QPF).

Поскольку `QPF` является растровым форматом, он быстрее и компактнее, чем такие векторные форматы, как `TTF` и `PostScript Type 1`, если требуется использовать только один или два различных размера. Утилита `makeqpf` может принимать файлы `TTF` или `PostScript Type 1` и сохранять результат в формате `QPF`. Можно поступить по-другому и запустить наши приложения с опцией командной строки `-savefonts`.

Интеграция приложений Qt при помощи Qtopia

Поскольку `Qt/Embedded Linux` предлагает такой же программный интерфейс, какой используется версиями `Qt` для настольных компьютеров, любое стандартное `Qt`-приложение может быть перекомпилировано и выполнено в `Qt/Embedded Linux`. Однако на практике, как правило, лучше создавать специальные приложения, учитывающие уменьшенные размеры экрана, ограниченные возможности клавиатур (или их отсутствие) и ресурсные ограничения, характерные для небольших устройств, на которых работает `Qt/Embedded Linux`. Более того, `Qtopia` содержит дополнительные библиотеки, обеспечивающие специфичные для мобильных устройств функции, которые могут потребоваться в наших приложениях `Qt/Embedded Linux`.

Прежде чем приступить к написанию приложений, использующих программные интерфейсы Qtopia, необходимо построить и установить Qtopia SDK, включая свою собственную, отдельную копию Qt/Embedded Linux. В дальнейшем предполагается, что мы используем версию 4.2 Qtopia Open Source Edition, которая включает в себя почти все, что имеется в версии Qtopia Phone Edition.

Процесс построения Qtopia отличается от стандартного подхода, применяемого в Unix, потому что систему Qtopia не следует создавать внутри собственного каталога исходных текстов. Например, если мы загружаем пакет `qtopia-opensource-src-4.2.4.tar.gz` в наш каталог `$HOME/downloads`, то при подготовке построения Qtopia следует выполнить следующие команды:

```
cd $HOME/downloads
gunzip qtopia-opensource-src-4.2.4.tar.gz
tar xvf qtopia-opensource-src-4.2.4.tar
```

Теперь необходимо создать каталог, в котором будет создана Qtopia, например:

```
cd $HOME
mkdir qtopia
```

В документации рекомендуется для удобства создать переменную среды `QPEDIR`. Например:

```
export QPEDIR=$HOME/qtopia
```

Здесь предполагается использование командной оболочки Bash. Теперь можно приступить к построению Qtopia:

```
cd $QPEDIR
$HOME/downloads/qtopia-opensource-src-4.2.4/configure
make
make install
```

В данном случае при вызове утилиты `configure` не задаются никакие опции, но, возможно, вам потребуется это сделать. Предусмотренные опции можно увидеть, выполнив команду `configure -help`.

После завершения установки все файлы Qtopia будут находиться в каталоге `$QPEDIR/image`, а все файлы, созданные пользователем в результате взаимодействия с Qtopia, будут находиться в каталоге `$QPEDIR/home`. По терминологии Qtopia, «image» – это файловая система Qtopia, которая находится на настольном компьютере и использует среду Qtopia при выполнении Qtopia в виртуальном буфере фреймов (virtual framebuffer).

Qtopia содержит свой собственный комплект полной документации, и имеет смысл познакомиться с ней, поскольку Qtopia предлагает много классов, которых нет в настольных версиях Qt или которые не соответствуют классам в этих версиях. Знакомство следует начать с файла `$QPEDIR/doc/html/index.html`.

После завершения построения и установки можно выполнить предварительный тест с помощью команды `$QPEDIR/scripts/runqtopia`. Этот скрипт запускает виртуальный буфер фреймов со специальной оболочкой и `qpe` – среды Qtopia, которая содержит стек приложений Qtopia. Можно запускать отдельно виртуальный буфер фреймов и среду Qtopia, но в этом случае они должны запускаться в определенном порядке. Если мы нечаянно сначала стартуем `qpe`, Qtopia запи-

шет в буфер фреймов X11, что в лучшем случае приведет к искажению экрана. Скрипт `runqtopia` можно выполнить с опцией `-help` для ознакомления со списком предусмотренных возможностей. В частности, одной из опций является `-skin` со списком доступных оболочек.

Виртуальный буфер фреймов имеет контекстное меню, которое можно вызывать нажатием правой кнопки мыши в любом месте вне области Qtopia. Это меню позволяет настраивать вывод изображений и завершать Qtopia.

Теперь, когда Qtopia выполняется в виртуальном буфере фреймов, мы можем создать приложение одного из поставляемых примеров, чтобы просто посмотреть, как это делается. Затем мы создадим очень простое приложение «с чистого листа».

Перейдите в каталог `$QPEDIR/examples/application`. Qtopia имеет свою собственную версию `qmake`, которая имеет имя `qtopiamake` и находится в `$QPEDIR/bin`. Выполните ее, чтобы создать `makefile`, и затем выполните `make`. В результате будет создан исполняемый модуль с именем `example` (пример). Выполните команду `make install`, что приведет к копированию `example` (и некоторых других файлов) в каталог Qtopia `image`. Теперь, если завершить Qtopia и затем вновь стартовать, используя скрипт `runqtopia`, наше новое приложение `example` будет доступно. Для запуска этого примера нажмите кнопку «Q», которая находится в середине нижней части области Qtopia, затем нажмите пиктограмму Packages (Пакеты) (пиктограмма с коробками непосредственно над пиктограммой с указательным пальцем) и после этого выберите в меню строку «Example» (см. рис. 24.3).

Мы завершим данный раздел небольшим, но полезным приложением Qtopia, создавая его с чистого листа, поскольку оно имеет несколько особенностей, которые надо иметь в виду. Это приложение называется Unit Converter (Преобразователь единиц измерения), и оно показано на рис. 24.2. Оно использует только программный интерфейс Qt и поэтому имеет несколько сюрпризов. В следующем разделе мы создадим более сложный пример, применяющий несколько программных интерфейсов Qtopia.

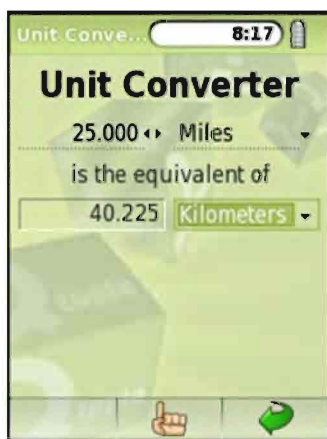


Рис. 24.2. Приложение Unit Converter

Приложение Unit Converter будет создано из трех файлов: `main.cpp`, `unitconverter.h` и `unitconverter.cpp`. Создайте новый каталог с именем `unitconverter`, затем войдите в него с помощью команды `cd` и создайте пустые файлы `.cpp` и `.h`. Теперь выполните команду

```
qtopiamake -project
```

для создания файла `.pro`. Этот файл будет выглядеть примерно так:

```
qtopia_project(qtopia app)
```

```
TARGET      = unitconverter
CONFIG      += qtopia_main no_quicklaunch
HEADERS     += unitconverter.h
SOURCES     += main.cpp \
              unitconverter.cpp
pkg.domain  = none
```



Рис. 24.3. Вызов приложения Example

Даже если вы напишете программный код, создадите исполняемый модуль и установите его, он не появится в списке приложений Qtopia. Чтобы приложение появилось в этом списке, необходимо указать место расположения изображений, файла `.desktop` и куда их следует поместить. Кроме того, принято указывать некоторые сведения о пакете. По этой причине мы вручную отредактируем файл `.pro`, чтобы он выглядел следующим образом:

```
qtopia_project(qtopia app)
TARGET      = unitconverter
CONFIG      += qtopia_main no_quicklaunch
HEADERS     += unitconverter.h
SOURCES     += main.cpp \
              unitconverter.cpp
INSTALLS    += desktop pics
```

```
desktop.files = unitconverter.desktop
desktop.path  = /apps/Applications
desktop.hint  = desktop

pics.files    = pics/*
pics.path     = /pics/unitconverter
pics.hint     = pics

pkg.name      = unitconverter
pkg.desc      = Программа преобразования различных единиц изменения
pkg.version   = 1.0.0
pkg.domain    = window
```

Файл `.pro` теперь содержит элемент `INSTALLS`, который показывает, что файл приложения `.desktop` и изображения приложения должны быть установлены в дополнение к исполняемому модулю при выполнении команды `make install`.

По принятым соглашениям изображения хранятся в подкаталоге `pics`, а элементы `pics.xxx` в файле `.pro` описывают место расположения исходных изображений и куда их следует устанавливать. Элементы `desktop.xxx` задают имя файла приложения `.desktop` и показывают, куда его следует устанавливать. Его установка в каталог `/apps/Applications` гарантирует, что он появится в списке приложений, который выводится на экран при выборе пользователем пиктограммы `Packages`. Кроме того, при выполнении приложения на настольном компьютере такие абсолютные пути, как `/apps/Applications` и `/pics/expenses` на самом деле задаются относительно каталога `Qtoria image` (с заменой `apps` каталогом `bin`).

Файл `unitconverter.desktop` содержит некоторые основные сведения о приложении. В нашем случае он используется для того, чтобы гарантировать появление приложения в списке приложений. Этот файл имеет следующее содержимое:

```
[Desktop Entry]
Comment[]=Программа преобразования различных единиц изменения
Exec=unitconverter
Icon=unitconverter/Example
Type=Application
Name[]=Unit Converter
```

Приведенная информация представляет собой только часть того, что допускается описывать в этом файле. Например, можно предоставить информацию о переводе на другие языки. Обратите внимание на то, что здесь изображения не имеют расширения, например, такого, как `.png`; мы предоставляем ресурсной системе `Qtoria` возможность самой найти и вывести на экран соответствующее изображение.

До сих пор мы видели специальные, вручную отредактированные файлы `.pro` и `.desktop`. Нам потребуется сделать еще одну характерную для `Qtoria` вещь, и тогда можно будет создавать файлы `unitconverter.h` и `unitconverter.cpp`, используя стандартный `Qt` обычным образом. При использовании `Qtoria` необходимо следовать определенному образцу, чтобы получить доступ к остальной информации относительно среды; весь файл `main.cpp` состоит всего лишь из следующих строк:

```
#include <QtopiaApplication>

#include "unitconverter.h"

QTOPIA_ADD_APPLICATION("UnitConverter", UnitConverter)
QTOPIA_MAIN
```

Класс основного окна, содержащийся во включенном заголовочном файле `unitconverter.h`, имеет имя `UnitConverter`. Используя `qtopiamake` вместе с файлами `unitconverter.h` и `unitconverter.cpp`, мы можем создать приложение `Qtopia`, которое будет выполняться в среде `Qtopia`. Функция `main()` определяется макросом `QTOPIA_MAIN`.

Поскольку это приложение использует только стандартные классы `Qt` (не считая `main.cpp`), его можно было бы компилировать и выполнять как обычное `Qt`-приложение. Для этого потребовалось бы использовать стандартный `qmake` и изменить `main.cpp` следующим образом:

```
#include <QApplication>

#include "unitconverter.h"

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

Кроме того, нам потребовалось бы закомментировать строку `qtopia_project()` в файле `.pro`.

Приложения, работающие только в среде `Qt/Embedded Linux`, во многих случаях удобнее разрабатывать как стандартные `Qt`-приложения, возможно, с явным вызовом `resize()` для главного окна, чтобы настроиться на размеры экрана мобильного телефона или КПК, а затем преобразовать их в приложения `Qtopia`, когда они будут готовы к проведению альфа-тестирования. В другом случае можно было бы иметь два различных файла `main.cpp`, например, `main_desktop.cpp` и `main_qtopia.cpp`, а также два файла `.pro`.

Большая часть программного кода приложения `Unit Converter` аналогична коду других приводимых в книге `Qt`-примеров, поэтому мы не будем его здесь специально рассматривать.

Для тестирования приложения необходимо выполнить `make`, затем `make install` и `runqtopia`. При выполнении `Qtopia` в виртуальном буфере фреймов можно нажать кнопку `Q`, выбрать пиктограмму `Packages` и затем выбрать пункт меню `Unit Converter`. После этого можно выполнить приложение, изменяя числовое значение и выбирая различные единицы измерения в поле с выпадающим списком.

Процесс создания приложений `Qtopia` не сильно отличается от создания обычных `Qt`-приложений, за исключением некоторой первоначальной настройки (специальный файл `.pro` и файл `.desktop`) и применения `qtopiamake` вместо

qmake. Тестирование встроенных приложений выполняется достаточно просто, потому что они могут быть построены, установлены и затем выполнены в виртуальном буфере фреймов. Хотя такие приложения могут создаваться как обычные Qt-приложения, на практике, как правило, результат будет лучше, если заранее учитываются ограничения встроенной среды и используются специальные программные интерфейсы Qtopia, чтобы обеспечить хорошую их интеграцию с остальной частью стека приложений Qtopia.

Использование API Qtopia

Qtopia PDA Edition и Qtopia Phone Edition содержат набор приложений, ориентированных на пользователей мобильных устройств. Функциональность большинства таких приложений реализуется в виде библиотек, или эти приложения используют библиотеки соответствующих версий Qtopia.

Эти приложения могут применяться в наших собственных приложениях Qtopia, позволяя получать доступ, например, к таким службам, которые обеспечивают получение сигналов тревоги, электронную почту, вызов телефонного номера, SMS, запись голосовых сообщений и многое другое.

Если необходимо получать доступ из наших приложений к специфичным для устройств функциям, то у нас есть несколько возможностей:

- можно использовать Qt/Embedded Linux и написать наш собственный программный код по взаимодействию с устройством;
- можно модифицировать существующее приложение Qtopia, обеспечивая необходимую нам функциональность;
- можно воспользоваться дополнительными программными интерфейсами, например, Qtopia Phone API или библиотекой приложения Qtopia PIM (Personal Information Manager).

В следующем разделе мы используем третий подход. Будет написано небольшое приложение Expenses, которое регистрирует простые сведения о расходах. Оно использует данные приложения Qtopia PIM для вывода на экран списка контактов с последующей отсылкой отчета о расходах заданным адресатам в виде SMS-сообщений. Кроме того, оно демонстрирует способ применения многофункциональных программируемых клавиш («soft keys»), которые имеются у большинства мобильных телефонов.

Как показано на рис. 24.4, это приложение, как и приложение примера, построенного в предыдущем разделе, в конце концов, появится в списке пакетов приложений. Как и в предыдущем примере, мы начнем с файла .pro, затем создадим файл .desktop и, наконец, создадим исходные файлы приложения. Ниже приводится содержимое файла expenses.pro:

```
qtopia_project(qtopia app)
depends(libraries/qtopiapim)

CONFIG      += qtopia_main no_quicklaunch
HEADERS     += expense.h \
              expensedialog.h \
              expenswindow.h
```

```

SOURCES      += expense.cpp \
               expensedialog.cpp \
               expenswindow.cpp \
               main.cpp
INSTALLS     += desktop pics

desktop.files = expenses.desktop
desktop.path  = /apps/Applications
desktop.hint  = desktop

pics.files    = pics/*
pics.path     = /pics/expenses
pics.hint     = pics

pkg.name      = expenses
pkg.desc      = Программа регистрации расходов и передачи SMS о расходах
pkg.version   = 1.0.0
pkg.domain    = window

```

Строка с `qtopia_project()` такая же, как и раньше. Поскольку приложение основано на применении библиотеки Qtopia PIM, используется директива `depends()` для описания библиотеки. Если необходимо использовать несколько библиотек, они должны перечисляться через запятую. Остальная часть файла `.pro` аналогична тому, что мы видели в примере Unit Converter, только на этот раз у нас немного больше исходных файлов, потому что теперь приложение проработано более детально.



Рис. 24.4. Выбор приложения Expenses и его выполнение

Файл `expenses.desktop` очень похож на то, что мы видели ранее:

```

[Desktop Entry]
Comment[ ]=Программа регистрации расходов и передачи SMS о расходах
Exec=expenses
Icon=expenses/expenses
Type=Application
Name[ ]=Expenses

```


То же самое можно сказать о файле `main.cpp`:

```
#include <QtopiaApplication>

#include "expenswindow.h"

QTOPIA_ADD_APPLICATION("Expenses", ExpenseWindow)
QTOPIA_MAIN
```

Теперь рассмотрим заголовочные файлы приложения `Expenses` и те части файлов исходных текстов, которые специфичны для `Qtoria` или для конкретного приложения, начиная с класса `Expense`:

```
class Expense
{
public:
    Expense();
    Expense(const QDate &date, const QString &desc, qreal amount);

    bool isNull() const;
    void setDate(const QDate &date);
    QDate date() const;
    void setDescription(const QString &desc);
    QString description() const;
    void setAmount(qreal amount);
    qreal amount() const;

private:
    QDate myDate;
    QString myDesc;
    qreal myAmount;
};
```

Этот простой класс содержит дату, описание и сумму расходов. Мы не будем рассматривать файл `expense.cpp`, поскольку он очень простой и не содержит специфичного для `Qtoria` программного кода.

```
class ExpenseWindow : public QMainWindow
{
    Q_OBJECT

public:
    ExpenseWindow(QWidget *parent = 0, Qt::WFlags flags = 0);

protected:
    void closeEvent(QCloseEvent *event);

private slots:
    void add();
    void edit();
    void del();
```

```

void send();
void clear();

private:
    void createActions();
    void createMenuOrToolBar();
    void loadData();
    void showData();
    ...
    QList<Expense> expenses;
};

```

ExpenseWindow — это главная форма приложения. Она содержит функции по добавлению, редактированию и удалению пользователем отдельных позиций расходов, для отсылки SMS-сообщения со всеми позициями расходов и для их очистки. Расходы хранятся в виде числовых значений в списке `QList<Expense>`.

Конструктор создает виджет `QListWidget` и два поля `QLabel`. Одно поле содержит текст «Total» (Всего), а другое — итоговую сумму расходов. Действия создаются функцией `createActions()`, а меню и панель инструментов создаются функцией `createMenuOrToolBar()`. Обе функции вызываются из конструктора. Существующие расходы загружаются в конце конструктора с помощью функции `loadData()`. Сам конструктор мы не будем рассматривать, но обсудим вызываемые в нем функции.

```

void ExpenseWindow::createActions()
{
    addAction = new QAction(tr("Add"), this);
    addAction->setIcon(QIcon(":/icon/add"));
    connect(addAction, SIGNAL(triggered()), this, SLOT(add()));
    ...
    clearAction = new QAction(tr("Clear"), this);
    clearAction->setIcon(QIcon(":/icon/clear"));
    connect(clearAction, SIGNAL(triggered()), this, SLOT(clear()));
}

```

Функция `createActions()` создает действия Add, Edit, Delete, Send и Clear (Добавить, Редактировать, Удалить, Послать и Очистить). Несмотря на то, что можно использовать файлы Qt-ресурсов (`.qrc`), обычно при программировании приложений Qtopia пиктограммы хранятся в поддиректории `pics`, которая копируется на устройство (благодаря строке `INSTALLS` в файле `.pro`). Эти изображения могут совместно использоваться несколькими приложениями, а Qtopia оптимизирует доступ к ним, используя специальную базу данных.

В любом месте, в котором Qt или Qtopia ожидает применения имени файла, вместо него можно предоставить имя ресурса Qtopia. Оно идентифицируется двоеточием в начале имени файла, за которым идет слово, определяющее тип ресурса. В данном случае мы указываем, что нам нужны пиктограммы и задаем имя файла, например, `:icon/add`, опуская расширение файла.

Qtoria будет выполнять поиск подходящей пиктограммы в нескольких стандартных каталогах, начиная с каталога приложения `pics`. Подробно этот процесс описан на web-странице <http://doc.trolltech.com/qtoria4.2/qtoria-resource-system.html>.

```
void ExpenseWindow::createMenuOrToolBar()
{
#ifdef QTOPIA_PHONE
    QMenu *menuOrToolBar = QSoftMenuBar::menuFor(listWidget);
#else
    QToolBar *menuOrToolBar = new QToolBar;
    addToolBar(menuOrToolBar);
#endif

    menuOrToolBar->addAction(addAction);
    menuOrToolBar->addAction(editAction);
    menuOrToolBar->addAction(deleteAction);
    menuOrToolBar->addAction(sendAction);
    menuOrToolBar->addAction(clearAction);
}
```

Некоторые мобильные телефоны имеют программируемые клавиши, т. е. многофункциональные клавиши, действия которых зависят от приложения и контекста. Класс `QSoftMenuBar` позволяет воспользоваться преимуществами программируемых клавиш там, где они имеются, и предоставить всплывающее меню, если эти клавиши не предусмотрены.

Для КПК обычно предпочтительнее иметь панель инструментов, а не всплывающее меню. Директива `#ifdef` обеспечивает добавление действий в программируемое меню для мобильного телефона и в панель инструментов для КПК.

Ожидается, что пользователям будет позволено закрывать приложения, не заставляя их явно сохранять данные. Также предполагается, что данные будут восстанавливаться при последующем запуске приложения. Об этом легко позаботиться, вызывая в конструкторе функцию `loadData()`, и сохраняя данные в функции приложения `closeEvent()`. Qtoria предлагает разные варианты памяти для хранения данных, в том числе они могут сохраняться в таблице базы данных SQLite или в файле. Поскольку данных о расходах очень немного, мы будем их сохранять, используя `QSettings`. Сначала мы рассмотрим, как эти данные сохраняются, а затем, как они загружаются.

```
void ExpenseWindow::closeEvent(QCloseEvent *event)
{
    QByteArray data;

    QDataStream out(&data, QIODevice::WriteOnly);
    out.setVersion(QDataStream::Qt_4_2);

    foreach (Expense expense, expenses) {
        out << expense.date() << expense.description()
            << expense.amount();
    }
}
```

```

QSettings settings("BookSoft Ltd", "Expenses");
settings.setValue("data", data);

event->accept();
}

```

Мы создадим один массив QByteArray и запишем в него все данные. Затем мы сохраним этот массив как единое целое с ключом data перед обработкой события QCloseEvent и завершением приложения.

```

void ExpenseWindow::loadData()
{
    QSettings settings("BookSoft Ltd", "Expenses");
    QByteArray data = settings.value("data").toByteArray();
    if (data.isEmpty())
        return;

    expenses.clear();
    QDataStream in(&data, QIODevice::ReadOnly);
    in.setVersion(QDataStream::Qt_4_2);

    while (!in.atEnd()) {
        QDate date;
        QString desc;
        qreal amount;
        in >> date >> desc >> amount;
        expenses.append(Expense(date, desc, amount));
    }
    showData();
}

```

Если остались данные от предыдущего сеанса, мы удаляем существующие данные, затем последовательно считываем все новые позиции расходов. Функция showData() очищает виджет списка, затем в цикле добавляет новый элемент для каждой позиции расходов и завершается обновлением итоговой суммы.

При выполнении приложения пользователь может добавлять, редактировать и удалять пункты расходов, посылать их все в виде SMS-сообщения или удалить весь список расходов.

При удалении мы убеждаемся, что выбрана строка расходов в виджете списка и вызываем стандартную статическую функцию QMessageBox::warning(), чтобы пользователь мог подтвердить удаление. Если пользователь задает операцию очистки всех расходов, мы также используем эту функцию. Такой подход является стандартным при Qt-программировании. Qtopia позаботится о том, чтобы окно с сообщением корректно выводилось на экран в среде Qtopia.

При выборе пользователем строки меню Add для добавления новой позиции расходов вызывается слот add():

```
void ExpenseWindow::add()
{
    ExpenseDialog dialog(Expense(), this);
    if (QtopiaApplication::execDialog(&dialog)) {
        expenses.append(dialog.expense());
        showData();
    }
}
```

Этот слот создает экземпляр класса `ExpenseDialog`; этот класс будет вскоре рассмотрен, однако вместо вызова функции диалогового окна `QDialog::exec()`, мы вызываем `QtopiaApplication::execDialog()`, передавая диалоговое окно в качестве аргумента. Вызов функции `exec()` вполне допустим, и будет работать, но применение функции `execDialog()` гарантирует правильный размер и положение этого окна на небольшом устройстве, при необходимости увеличивая до максимума размер окна.

Слот `edit()` имеет аналогичный вид. Если вызывается функция `edit()`, она убеждается, что выбрана строка расходов в виджете списка и передает эту строку расходов конструктору `ExpenseDialog` в его первом параметре. Если пользователь завершает редактирование успешно, элементы строки расходов заменяются отредактированными значениями.

Последней функцией `ExpenseWindow`, которую мы будем рассматривать, является функция `send()`, но сначала мы обсудим класс `ExpenseDialog`:

```
class ExpenseDialog : public QDialog
{
    Q_OBJECT

public:
    ExpenseDialog(const Expense &expense, QWidget *parent = 0);
    Expense expense() const { return currentExpense; }

public slots:
    void accept();

private:
    void createActions();
    void createMenuOrToolBar();
    Expense currentExpense;
    ...
};
```

Сразу становится очевидным, что здесь имеются функции для создания действий, меню и панелей инструментов, которые аналогичны функциям в `ExpenseWindow`. Мы не будем создавать кнопки `QPushButton` или `QDialogButtonBox`, а вместо них создадим панель инструментов или меню `QSoftMenuBar`, поскольку эти средства обеспечивают лучшую интеграцию со средой `Qtopia`. Программный код очень похож на то, что мы делали для главного окна приложения:

```

void ExpenseDialog::createActions()
{
    okAction = new QAction(tr("OK"), this);
    okAction->setIcon(QIcon(":/icon/ok"));
    connect(okAction, SIGNAL(triggered()), this, SLOT(accept()));

    cancelAction = new QAction(tr("Cancel"), this);
    cancelAction->setIcon(QIcon(":/icon/cancel"));
    connect(cancelAction, SIGNAL(triggered()), this, SLOT(reject()));
}

void ExpenseDialog::createMenuOrToolBar()
{
#ifdef QTOPIA_PHONE
    QMenu *menuOrToolBar = QSoftMenuBar::menuFor(this);
#else
    QToolBar *menuOrToolBar = new QToolBar;
    menuOrToolBar->setMovable(false);
    addToolBar(menuOrToolBar);
#endif

    menuOrToolBar->addAction(okAction);
    menuOrToolBar->addAction(cancelAction);
}

```

Если пользователь вызывает диалоговое окно, мы устанавливаем атрибуты даты, описания и суммы текущих расходов и предоставляем пользователю возможность получения этих данных с помощью функции `expense()` этого диалогового окна.

Если пользователь выбирает действие `Send`, вызывается функция `send()`. Данная функция предлагает пользователю выбрать адресат сообщения, подготавливает текст сообщения и затем посылает это сообщение, используя протокол SMS (см. рис. 24.5).

```

void ExpenseWindow::send()
{
    QContactSelector dialog(false, this);
    dialog.setModel(new QContactModel);
    QtopiaApplication::execDialog(&dialog);
    if (!dialog.contactSelected())
        return;
}

```

Диалоговое окно `QContactSelector` и класс `QContactModel` содержатся в библиотеке `PIM`. `QContactModel` имеет доступ к централизованной базе данных контактов пользователя. Если контактов несколько, функция `QtopiaApplication::execDialog()` выведет на экран диалоговое окно `QContactSelector`, устанавливая максимальным его размер. Если пользователь не выбирает ни один из контактов, функция `contactSelected()` возвращает нулевой контакт (со значением `false`), и в этом случае ничего не делается. В противном случае, мы готовим и затем посылаем сообщение о расходах:



Рис. 24.5. Выбор контакта и отсылка SMS-сообщения

```

QTemporaryFile file;
file.setAutoRemove(false);
if (!file.open()) {
    QMessageBox::warning(this, tr("Expenses"),
                        tr("Failed to send expenses: %1.")
                        .arg(file.errorString()));
    return;
}

QString fileName = file.fileName();
qreal total = 0.00;

QTextStream out(&file);
out.setCodec("UTF-8");

out << tr("Expenses\n");

foreach (Expense expense, expenses) {
    out << tr("%1 $%2 %3\n")
        .arg(expense.date().toString(Qt::ISODate))
        .arg(expense.amount(), 0, 'f', 2)
        .arg(expense.description());
    total += expense.amount();
}
out << tr("Total $%1\n").arg(total, 0, 'f', 2);
file.close();

```

Для отправки SMS-сообщения потребуется передать имя файла, содержащего SMS-сообщение. В представленном ниже примере мы записываем данные расходов во временный файл, используя QTextStream. Обычно QTemporaryFile удаляет файл сразу после вызова функции close(), однако мы отключаем такой режим работы, потому что файл должен оставаться доступным вплоть до отправки сообщения SMS, после чего система Qtoria автоматически его удалит.

В объявлении переменной `total` задается тип `qreal`. Этот тип определяется с помощью директивы `typedef` как `float` или `double` в зависимости от архитектуры. Например, в ARM-архитектуре он определяется как `float` для повышения производительности. Повсюду в программном интерфейсе Qt (особенно в `QPainter`) `qreal` используется вместо типа `double`.

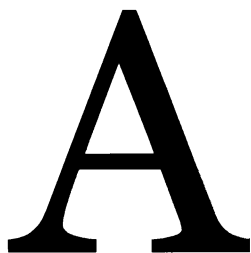
```
QContact contact = dialog.selectedContact();
QtopiaServiceRequest request("SMS",
                             "writeSms(QString,QString,QString)");
request << QString("%1 %2").arg(contact.firstName())
                             .arg(contact.lastName())
      << contact.defaultPhoneNumber() << fileName;
request.send();
}
```

`Qtopia` реализует протокол SMS в виде сервиса, а не в виде библиотеки. Для отправки SMS-сообщения создается объект `QtopiaServiceRequest` с указанием имени сервиса, «SMS» и имени функции, которую мы собираемся использовать с аргументами, перечисленными в скобках: «`writeSms(QString,QString,QString)`». Класс `QtopiaServiceRequest` реализуется с использованием `QCOP` для связи с процессом, обеспечивающим сервис «SMS».

В запросе мы указываем имя получателя и его телефонный номер, а также имя созданного файла и затем вызываем `send()` для отправки сообщения. При выполнении `send()` система `Qtopia` выводит на экран диалоговое окно `Create Message`, причем содержимое сообщения берется из файла. Пользователь может изменить текст и затем отослать или отменить SMS. Приложение `Expenses` может быть правильно протестировано только при использовании реальных и моделируемых устройств, обеспечивающих SMS-сервис.

Как показывает этот пример, встроенное программирование означает необходимость применения и взаимодействия с сервисами и доступными программными интерфейсами `Qtopia`. И нам приходится совсем по-другому подходить к разработке интерфейса пользователя с учетом небольшого размера экрана и ограниченных средств ввода данных мобильных устройств. С точки зрения программиста создание приложений для `Qtopia` не отличается от их создания на платформах настольных компьютеров, за исключением того, что приходится познакомиться с дополнительными инструментами, библиотеками и сервисами, доступными в `Qtopia`.

Приложения



- *Замечание о лицензировании*
- *Установка Qt/Windows*
- *Установка Qt/Mac*
- *Установка Qt/X11*

Приложение А.

Получение и установка Qt

В данном приложении рассматривается порядок получения и установки GPL-версии Qt в вашу систему. Предусмотрены версии для Windows, Mac OS X и X11 (для Linux и большинства версий Unix). Бинарные версии для Windows и Mac OS X включают SQLite – общедоступную, не нуждающуюся в сервере базу данных, и драйвер SQLite. При построении версий на основе исходных текстов можно как включать, так и не включать SQLite. Для начала следует загрузить из Интернет последнюю версию Qt по адресу <http://www.trolltech.com/download/>. Если вы собираетесь разрабатывать коммерческое программное обеспечение, то необходимо приобрести коммерческую версию и следовать инструкциям по установке, включенным в эту версию.

Компания Trolltech также обеспечивает Embedded Linux для построения приложений на базе системы Linux для таких устройств, как карманные компьютеры и мобильные телефоны. Если вы интересуетесь созданием встроенных приложений, вы можете скачать Embedded Linux с соответствующей веб-страницы сайта компании Trolltech.

Примеры приложения представленных в данной книге можно найти на веб-сайте <http://www.informit.com/title/0132354160>. Кроме того, Qt предоставляет много небольших приложений-примеров, которые располагаются в подкаталоге examples.

Замечание о лицензировании

Система Qt доступна в двух формах: с открытым исходным кодом и коммерческая. Версия с открытым исходным кодом распространяется бесплатно; за коммерческую версию приходится платить.

Если вы собираетесь распространять созданные вами приложения с использованием версии Qt с открытым исходным кодом, вы должны соблюсти определенные условия, которые отражены в лицензиях используемого вами программного обеспечения для создания своих программ. Для версий с открытым исходным

кодом такие условия включают лицензию GNU GPL (General Public License – общедоступная лицензия). Простые лицензии, подобные GPL, наделяют пользователей определенными правами, включая просмотр и модификацию исходного кода, а также распространение приложений (на тех же условиях). Если вы собираетесь распространять ваши приложения без исходного кода (и считаете ваш программный код своей собственностью) или хотите применять в отношении ваших приложений свою собственную коммерческую лицензию, вы должны приобретать коммерческие версии программного обеспечения, используемого для создания ваших программ. Коммерческие версии программного обеспечения позволяют вам продавать и распространять созданные вами приложения на ваших условиях.

Полные, юридически верные тексты лицензий включены в GPL-версии Qt для Windows, Mac OS X и X11; здесь же имеется информация о том, как получить коммерческие версии.

Установка Qt/Windows

На момент написания книги установщик Windows имел имя `qt-win-opensource-4.3.2-mingw.exe`. Номер версии может измениться к моменту чтения вами этой инструкции, но процесс установки принципиально меняться не должен. Загрузите этот файл и выполните его, чтобы начать процесс установки.

Когда установщик доходит до страницы MinGW, необходимо указать каталог размещения компилятора MinGW C++, если он уже имеется на вашей машине; в противном случае установите соответствующий флажок для установки компилятора MinGW. GPL-версия Qt не будет работать с Visual C++, поэтому необходимо установить компилятор MinGW, если он еще у вас не установлен. Стандартные примеры Qt вместе с документацией устанавливаются автоматически.

Если вы задаете установку компилятора MinGW, может быть небольшая задержка между завершением установки компилятора MinGW и началом установки Qt.

После установки в меню Пуск появится новая папка «Qt by Trolltech v4.3.2 (OpenSource)». Эта папка будет содержать ярлыки для *Qt Assistant* и *Qt Designer*, а также «Qt 4.3.2 Command Prompt», который запускает окно консоли. При запуске этого окна выполняется установка переменных среды для компиляции программ Qt с помощью MinGW. В этом окне можно выполнять утилиты `qmake` и `make` для создания Qt-приложений.

Установка Qt/Mac

До установки Qt в системе Mac OS X уже должны быть установлены утилиты Xcode компании «Apple». Эти утилиты обычно находятся на компакт-диске (или DVD-диске), поставляемом с системой Mac OS X; их можно также скачать с сайта Apple Developer Connection, <http://developer.apple.com>.

Если вы уже имеете Mac OS X 10.4 и Xcode Tools 2.x (вместе с компилятором GCC 4.0.x) или более старшие версии, можно воспользоваться установщиком, как это кратко описано ниже. Если вы имеете более старую версию Mac OS X или более старую версию GCC, необходимо вручную установить пакет с исход-

ными текстами. Этот пакет называется `qt-mac-opensource-4.3.2.tar.gz`, и его можно получить на web-сайте Trolltech. После установки этого пакета следуйте инструкциям по установке Qt в системе X11, которые приводятся в следующем разделе.

Для использования программы установки загрузите `qt-mac-opensource-4.3.2.dmg`. (Такое имя имела программа установки на момент написания книги, но, возможно, оно изменится к моменту чтения вами данной инструкции). Дважды щелкните по файлу `.dmg` и затем дважды щелкните по пакету `Qt.mpkg`. Это приведет к запуску программы установки, которая установит Qt в каталог `/Developer` вместе с документацией и стандартными примерами.

Для запуска таких команд, как `qmake` и `make`, необходимо использовать окно терминала, например, `Terminal.app` из `/Applications/Utilities`. Необходимо также сгенерировать проекты Xcode, используя `qmake`. Например, чтобы сформировать проект Xcode для примера `hello`, запустите консоль (например, `Terminal.app`), перейдите в ваш каталог `examples/chap01/hello` и введите следующую команду:

```
qmake -spec macx-xcode hello.pro
```

Установка Qt/X11

Загрузите файл `qt-x11-opensource-src-4.3.2.tar.gz` с web-сайта Trolltech. (Такое имя этот файл имел на момент написания книги, но, возможно, оно изменится к моменту чтения вами данной инструкции.) Для установки Qt в системе X11 в свой стандартный каталог вам могут потребоваться полномочия `root`. Если у вас нет таких полномочий, используйте опцию `-prefix` скрипта `configure` для указания каталога, в который вам разрешено записывать данные.

1. Перейдите в каталог, в который вы загрузили файл архива. Например:

```
cd /tmp
```

2. Распакуйте архивный файл:

```
gunzip qt-x11-opensource-src-4.3.2.tgz
tar xf qt-x11-opensource-src-4.3.2.tar
```

Это создает каталог `/tmp/qt-x11-opensource-src-4.3.2`. Для Qt требуется утилита GNU `tar`; в некоторых системах она называется `gtar`.

3. Выполните утилиту `configure` в новом окне терминала, задавая предпочитаемые вами опции построения библиотеки Qt и поддерживающих ее утилит:

```
cd /tmp/qt-x11-opensource-src-4.3.2
./configure
```

Вы можете запустить `./configure -help` для получения списка опций конфигурации.

4. Для построения Qt введите

```
make
```

В результате будет создана библиотека, и будут скомпилированы все демонстрационные программы, примеры и утилиты. В некоторых системах `make` имеет имя `gmake`.

5. Для установки Qt введите

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

и затем пароль `root`. (В некоторых системах необходимо выполнить команду `sudo make install`.) В результате Qt будет установлен в `/usr/local/Trolltech/Qt-4.3.2`. Вы можете изменить место расположения Qt, используя опцию `-prefix` скрипта `configure`, и если вы имеете разрешение на запись в это место, можно просто ввести команду:

```
make install
```

6. Настройте определенные переменные среды для Qt.

Если вы используете командную оболочку `bash`, `ksh`, `zsh` или `sh`, добавьте следующие две строки в ваш файл `.profile`:

```
PATH=/usr/local/Trolltech/Qt-4.3.2/bin:$PATH
export PATH
```

Если вы используете оболочку `csh` или `tcsh`, добавьте следующую строку в ваш файл `.login`:

```
setenv PATH /usr/local/Trolltech/Qt-4.3.2/bin:$PATH
```

Если вы использовали опцию `-prefix` для скрипта `configure`, задавайте указанный вами путь вместо стандартного пути, показанного выше.

Если вы используете компилятор, не поддерживающий `rpath`, необходимо в переменную среды `LD_LIBRARY_PATH` добавить также путь `/usr/local/Trolltech/Qt-4.3.2/lib`. Это не обязательно делать в системе Linux с компилятором GCC.

В состав Qt входит приложение `qtdemo`, которое демонстрирует многие возможности библиотеки. Оно служит хорошей отправной точкой, позволяющей понять, что можно сделать при помощи средств разработки Qt. Документацию Qt можно найти либо на сайте <http://doc.trolltech.com>, либо запустить *Qt Assistant*, приложение системы помощи в Qt, которое вызывается из окна консоли по команде `assistant`.

Б

- *Применение qmake*
- *Применение инструментов независимых разработчиков*

Приложение Б. Создание приложений Qt

Создавать Qt-приложения значительно проще при использовании соответствующего инструмента. При этом перед нами открываются следующие возможности: можно воспользоваться утилитой `qmake`, инструментом независимого разработчика или интегрированной средой разработки (IDE).

Утилита `qmake` генерирует зависящий от платформы `makefile` на основе независимого от платформы файла `.pro`. В этой утилите предусмотрен вызов соответствующих инструментов Qt по генерации программного кода (`moc`, `uic` и `rcc`). Утилита `qmake` использовалась во всех примерах книги в большинстве случаев с применением достаточно простых файлов `.pro`. На самом деле эта утилита обладает широкими возможностями, в том числе она может создавать файлы `makefile`, которые рекурсивно вызывают другие файлы `makefile`, и может включать или отключать определенные функции в зависимости от целевой платформы. В первом разделе данного приложения мы сделаем краткий обзор `qmake` и познакомим с некоторыми ее наиболее продвинутыми функциями.

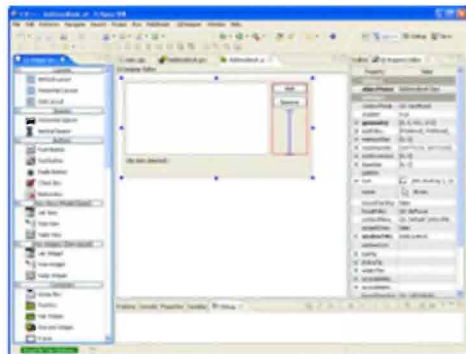
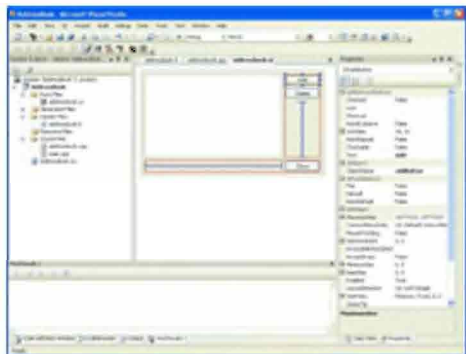


Рис. Б.1. Интеграция Qt с Visual Studio и Eclipse

Теоретически, любой инструмент независимого разработчика, предназначенный для создания исполняемых модулей, может использоваться в Qt-разработке, однако значительно проще применять инструмент, если он изначально ориентирован, в том числе на Qt. Некоторые такие инструменты будут рассмотрены во втором разделе данного приложения.

Некоторые разработчики предпочитают для создания своих приложений использовать системы IDE. Компания Trolltech предоставляет программное обеспечение для интеграции с Visual Studio и Eclipse (см. рис. Б.1), а в системах KDevelop и QDevelop (которые являются системами IDE с открытым исходным кодом, созданными с использованием Qt) обеспечивает превосходную поддержку Qt-разработки.

Применение qmake

Утилита qmake входит в состав пакета Qt. Она используется для построения самого пакета Qt, а также входящих в него утилит и примеров. Повсюду в книге мы использовали файлы проектов qmake (.pro) для создания примеров приложений и подключаемых модулей. В этом разделе мы систематически (но не досконально) исследуем синтаксис файлов .pro и рассмотрим несколько фундаментальных концепций qmake. Полное описание можно найти в руководстве по утилите qmake, доступном в сети Интернет по адресу <http://doc.trolltech.com/4.3/qmake-manual.html>.

Файл .pro содержит список исходных файлов, используемых в проекте. Поскольку qmake применяется для построения Qt и утилит, связанных с Qt, она очень хорошо «знает» Qt и может генерировать правила для вызова moc, uic и rcc. В результате синтаксис оказывается очень лаконичным и легко запоминаемым. Тремя основными типами файлов проектов являются app (для автономных приложений), lib (для статичных и совместно используемых библиотек) и subdirs (для рекурсивно создаваемых подкаталогов). Этот тип можно задавать при помощи переменной TEMPLATE, например:

```
TEMPLATE = lib
```

Шаблон subdirs может использоваться для построения целевых объектов в подкаталогах. В этом случае в добавление к строке TEMPLATE = subdirs необходимо задать только переменную SUBDIRS.

В каждом подкаталоге qmake найдет файл .pro с именем каталога и построит этот проект. Файл examples.pro, входящий в состав примеров, предоставляемых для данной книги, использует шаблон subdirs для выполнения qmake при построении каждого примера.

Если нет ни одного элемента TEMPLATE, по умолчанию проект будет иметь тип app. В проектах app и lib, наиболее широко используемыми являются перечисленные далее переменные.

- HEADERS — определяет заголовочные файлы C++ (.h) данного проекта.
- SOURCES — определяет файлы реализации C++ (.cpp) данного проекта.
- FORMS — определяет создаваемые в *Qt Designer* файлы .ui, обрабатываемые утилитой uic.
- RESOURCES — определяет файлы .qrc, обрабатываемые утилитой rcc.

- **DEFINES** — определяет символы препроцессора C++, которые необходимо определить заранее.
- **INCLUDEPATH** — определяет каталоги, из которых препроцессор C++ будет брать глобальные заголовочные файлы.
- **LIBS** — определяет библиотеки, подключаемые к проекту. Доступ к этим библиотекам задается с помощью абсолютных путей или с помощью применяемых в Unix-системах флажков `-L` и `-l` (например, `-L/usr/local/lib` и `-ldb_cxx`).
- **CONFIG** — определяет различные параметры конфигурации проекта и параметры компилятора.
- **QT** — определяет используемые в проекте модули Qt. (По умолчанию используется значение `core gui`, соответствующее модулям *QtCore* и *QtGui*).
- **VERSION** — определяет номер версии целевой библиотеки.
- **TARGET** — определяет базовое имя целевого исполняемого модуля или целевой библиотеки без расширения, префикса или номера версии. (По умолчанию используется имя текущего каталога).
- **DESTDIR** — определяет каталог, в который будет помещен целевой исполняемый модуль. (По умолчанию используется значение, зависящее от платформы, например в Linux им является текущий каталог, в Windows — подкаталог `debug` или `release`).
- **DLLDESTDIR** — определяет каталог, в который будет помещена целевая библиотека. (По умолчанию используется такое же значение, как и для переменной `DESTDIR`).

Переменная **CONFIG** используется для управления различными аспектами процесса построения. Поддерживаются опции, перечисленные далее.

- **debug** означает, что исполняемый модуль или библиотека должны содержать отладочную информацию и ссылки на отладочные версии библиотек Qt.
- **release** означает, что исполняемый модуль или библиотека не должны содержать отладочную информацию и должны иметь ссылки на рабочие версии библиотек Qt.
- Если заданы и **debug**, и **release**, используется **debug**.
- **warn_off** отключает вывод предупреждений. По умолчанию вывод предупреждений включен.
- **qt** означает, что приложение или библиотека используют Qt. Эта опция действует по умолчанию.
- **dll** означает, что необходимо создать совместно используемую библиотеку.
- **staticlib** означает, что необходимо создать статическую библиотеку.
- **plugin** означает, что необходимо создать подключаемый модуль. Подключаемые модули всегда являются совместно используемыми библиотеками, поэтому данная опция подразумевает использование опции **dll**.
- **console** означает, что приложение должно выводить сообщения на консоль (используя `cout`, `cerr`, `qWarning()` и т. п.).

- `app_bundle` относится только к построению модулей в системе Mac OS X и означает, что исполняемый модуль должен помещаться в пакет (bundle), что является стандартным условием в Mac OS X.
- `lib_bundle` относится только к построению библиотек в системе Mac OS X и означает, что библиотека должна помещаться в фреймворк.

Чтобы сгенерировать файл `makefile` для файла проекта с именем `hello.pro`, мы выполняем команду

```
qmake hello.pro
```

После этого можно вызвать `make` или `nmake` для построения проекта. Утилиту `qmake` можно также использовать для генерации файла проекта Microsoft Visual Studio (`.dsp` или `.vproj`), выполняя команду

```
qmake -tp vc hello.pro
```

В системе Mac OS X мы можем сгенерировать проект Xcode, используя команду

```
qmake -spec macx-xcode hello.pro
```

и файлы `makefile`, используя команду

```
qmake -spec macx-g++ hello.pro
```

Опция командной строки `-spec` позволяет задавать платформу и компилятор. Обычно `qmake` правильно определяет платформу, но в некоторых случаях требуется указать ее явно. Например, для генерации файла `makefile`, который вызывает компилятор Intel C++ Compiler (ICC) для Linux в 64-битовом режиме, мы бы ввели команду

```
qmake -spec linux-icc-64 hello.pro
```

Допустимые спецификации находятся в каталоге `mkspecs` пакета Qt. Хотя утилита `qmake` главным образом предназначена для генерации файлов `makefile` на основе файлов `.pro`, ее можно также использовать для генерации файла `.pro` для текущего каталога, применяя опцию `-project`. Например:

```
qmake -project
```

В этом случае `qmake` будет искать в текущем каталоге файлы с известными расширениями (`.h`, `.cpp`, `.ui` и т. п.) и создаст файл `.pro`, содержащий список этих файлов. В остальной части этого раздела мы более подробно рассмотрим синтаксис файлов `.pro`. Элемент в файле `.pro`, как правило, имеет формат

```
переменная = значения
```

где *значения* представляет собой список строковых значений. Комментарии начинаются со знака решетки (`#`) и завершаются окончанием строки. Например, строка

```
CONFIG = qt release warn_off # Я знаю, что делаю
```

приводит к присваиванию списка [`qt`», «`release`», «`warn_off`»] переменной `CONFIG`, заменяя любые ранее заданные значения. Кроме оператора `=` предусмотрены дополнительные операторы. Оператор `+=` позволяет добавлять значения в конец значения переменной. Так, строки

```
CONFIG = qt
CONFIG += release
CONFIG += warn_off
```

фактически приводят к присвоению списка [«qt», «release», «warn_off»] переменной CONFIG, как это делалось в предыдущем примере. Оператор -= удаляет все экземпляры заданных значений из текущего значения переменной. Так,

```
CONFIG = qt release warn_off
CONFIG -= qt
```

оставляет в переменной CONFIG список [«release», «warn_off»]. Оператор += добавляет значение в переменную только в том случае, если этого значения нет в списке переменной; в противном случае ничего не делается. Например, строка

```
SOURCES += main.cpp
```

приведет к добавлению файла реализации main.cpp в проект, если его там нет. Наконец, оператор ~= может использоваться для замены любых значений, которые соответствуют регулярному выражению, на заданный текст замены, применяя синтаксис, используемый в sed (поточковый редактор Unix). Например,

```
SOURCES ~= s/\.cpp\b/.cxx/
```

заменяет в переменной SOURCES все расширения файлов .cpp расширением .cxx. Внутри списка переменных можно получать доступ к другим переменным qmake, переменным среды и параметрам конфигурации Qt. Синтаксис доступа к этим переменным описан на рис.Б.2.

Переменные доступа (Accessor)	описание
<code>\$\$varName or \${varName}</code>	Значение переменной qmake в данном месте файла .pro
<code>\$(varName)</code>	Значение переменной среды во время выполнения qmake
<code>\$(varName)</code>	Значение переменной среды при обработке файла makefile
<code>\$\$[varName]</code>	Значение параметра конфигурации Qt

Рис. Б.2. Синтаксис доступа к переменным в qmake

В приводимых до сих пор примерах мы всегда использовали стандартные переменные, например, SOURCES и CONFIG, однако можно задавать значение любой переменной и затем ссылаться на нее, используя синтаксис \$\$varName или \${varName}. Например:

```
MY_VERSION = 1.2
SOURCES_BASIC = alphadialog.cpp \
                main.cpp \
                windowpanel.cpp
SOURCES_EXTRA = bezierextension.cpp \
                xplot.cpp
SOURCES        = $$SOURCES_BASIC \
                $$SOURCES_EXTRA
TARGET         = imgpro_$$MY_VERSION
```

В следующем примере используется несколько рассмотренных ранее вариантов синтаксиса, а встроенная функция `$$lower()` применяется для преобразования строк в нижний регистр:

```
# Список классов проекта
MY_CLASSES    = Annotation \
                CityBlock \
                CityScape \
                CityView

# Добавить в конец имен классов, преобразованных в нижний регистр,
# расширение .cpp и добавить main.cpp
SOURCES       = $$lower($$MY_CLASSES)
SOURCES       += s/([a-z0-9_]+)/\1.cpp/
SOURCES       += main.cpp

# Добавить в конец имен классов, преобразованных в нижний регистр,
# расширение . h
HEADERS       = $$lower($$MY_CLASSES)
HEADERS       += s/([a-z0-9_]+)/\1.h/
```

Иногда может потребоваться, чтобы имена файлов в файле `.pro` имели пробелы. В этом случае имя файла задается в кавычках. Когда проект компилируется в различных платформах, может возникнуть необходимость указывать разные файлы или разные параметры в зависимости от используемой платформы. В `qmake` предусмотрен следующий синтаксис для условий:

```
условие {
    then-вариант
} else {
    else-вариант
}
```

В части *условие* можно задавать имя платформы (например, `win32`, `unix` или `macx`) или более сложный предикат. В частях *then-вариант* и *else-вариант* задаются значения переменных, используя стандартный синтаксис. Например:

```
win32 {
    SOURCES += serial_win.cpp
} else {
    SOURCES += serial_unix.cpp
}
```

Вариант *else* задавать необязательно. Для удобства в `qmake` предусмотрен также однострочный вариант, когда часть *then-вариант* содержит только одно присваивание переменной и не имеет *else-вариант*:

условие: then-вариант

Например:

```
macx:SOURCES += serial_mac.cpp
```

Если несколько файлов проектов используют одинаковые элементы, эти элементы можно выделить в отдельный файл и включать их в каждый файл `.pro`, для которого они необходимы, используя директиву `include()`:

```
include../common.pro

HEADERS      += window.h
SOURCES      += main.cpp \
               window.cpp
```

По принятым соглашениям файлам проектов, которые включаются в другие файлы проектов, назначается расширение `.pri` (project include).

В предыдущем примере использовалась встроенная функция `$$lower()`, которая возвращает свой аргумент, преобразованный в нижний регистр. Другой полезной функцией является `$$system()`: она позволяет генерировать строки для внешних приложений. Например, если требуется определить, какая используется версия Unix, можно написать

```
OS_VERSION = $$system(uname -r)
```

Полученное значение переменной можно затем использовать в условии вместе с функцией `contains()`:

```
contains(OS_VERSION, SunOS):SOURCES += mythread_sun.c
```

В этом разделе мы затронули только вершину айсберга. В утилите `qmake` предусмотрено много других опций и функций, в том числе поддержка заранее откомпилированных заголовков, универсальных двоичных модулей для Mac OS X и определенных пользователем компиляторов и утилит. Полное описание всех возможностей `qmake` можно найти в онлайн-руководстве по этой утилите.

Применение инструментов независимых разработчиков

Существует много инструментов, как с открытым исходным кодом, так и коммерческих, которые можно использовать для создания Qt-приложений. Эти инструменты можно разделить на две большие категории: инструменты, генерирующие файлы `makefile` (или файлы проектов IDE) и ориентированные на стандартную систему построения приложений, и инструменты, которые сами являются системами построения приложений и не пользуются внешними инструментами построения.

В этом разделе мы сделаем обзор трех инструментов, каждый из которых либо имеет встроенную поддержку Qt-приложений, либо такая поддержка легко обеспечивается. Первый инструмент, `CMake`, генерирует файлы `makefile`, а два других, `Boost.Build` и `SCons`, являются системами построения приложений. Каждым инструментом будет построено приложение Электронная таблица (Spreadsheet), разработанное в гл. 3 и 4. Для оценки любого нового инструмента или новой системы построения приложений необходимо познакомиться с некоторым дополнительным материалом и провести эксперименты с реальными приложениями, однако мы надеемся, что наш краткий обзор средств построения будет полезен.

СMake: межплатформенная утилита make

Утилита СMake, которую можно получить на сайте <http://www.cmake.org/>, является межплатформенным генератором файлов makefile, который имеет открытый исходный код и поддерживает разработку приложений Qt 4. Чтобы использовать СMake, необходимо создать файл CMakeLists.txt с описанием проекта, который очень похож на файл .pro утилиты qmake. Ниже приводится содержание файла CMakeLists.txt для приложения Электронная таблица:

```
project(spreadsheet)
cmake_minimum_required(VERSION 2.4.0)
find_package(Qt4 REQUIRED)
include(${QT_USE_FILE})
set(spreadsheet_SRCS
    cell.cpp
    finddialog.cpp
    gotocelldialog.cpp
    main.cpp
    mainwindow.cpp
    sortdialog.cpp
    spreadsheet.cpp
)
set(spreadsheet_MOC_SRCS
    finddialog.h
    gotocelldialog.h
    mainwindow.h
    sortdialog.h
    spreadsheet.h
)
set(spreadsheet_UIS
    gotocelldialog.ui
    sortdialog.ui
)
set(spreadsheet_RCCS
    spreadsheet.qrc
)
qt4_wrap_cpp(spreadsheet_MOCS ${spreadsheet_MOC_SRCS})
qt4_wrap_ui(spreadsheet_UIS_H ${spreadsheet_UIS})
qt4_wrap_cpp(spreadsheet_MOC_UI ${spreadsheet_UIS_H})
qt4_add_resources(spreadsheet_RCC_SRCS ${spreadsheet_RCCS})
add_definitions(-DQT_NO_DEBUG)
add_executable(spreadsheet
    ${spreadsheet_SRCS}
    ${spreadsheet_MOCS}
    ${spreadsheet_MOC_UI}
    ${spreadsheet_RCC_SRCS})
target_link_libraries(spreadsheet ${QT_LIBRARIES} pthread)
```

Большая часть этого файла стандартна. Зависимыми от приложения элементами являются только имя приложения (в первой строке), список исходных файлов .cpp, список файлов .ui и список файлов .qrc. Нам не приходится задавать заголовочные файлы, потому что утилита CMake достаточно «умна» и способна сама выявлять зависимости. Однако файлы .h, определяющие классы Q_OBJECT, должны обрабатываться компилятором moc, поэтому они здесь указаны.

В первых нескольких строках задается имя приложения и обеспечивается поддержка утилитой CMake приложений Qt 4. Затем несколькими переменным присваиваются имена файлов. Команда qt4_wrap_cpp() обеспечивает обработку компилятором moc заданных файлов, и, аналогично, команда qt4_wrap_ui() обеспечивает обработку заданных файлов утилитой uic, а команда qt4_add_resources() обеспечивает обработку заданных файлов утилитой rcc. Чтобы создать исполняемый модуль, мы указываем все необходимые файлы .cpp, включая сгенерированные утилитами moc и rcc. Наконец, необходимо указать библиотеки, которые используются исполняемым модулем; в данном случае это стандартные библиотеки Qt 4 и библиотека поддержки многопоточности.

Завершив создание файла CMakeLists.txt, можно сгенерировать makefile, используя следующую команду:

```
cmake
```

По этой команде утилита CMake считывает файл CMakeLists.txt, находящийся в текущем каталоге, и генерирует файл с именем Makefile. После этого мы можем выполнить команду make (или nmake) для создания приложения или make clean, если приложение требуется создать с нуля.

Boost.Build (bjam)

Библиотеки Boost классов C++ имеют собственный инструмент построения приложений под названием Boost.Build или bjam, который свободно доступен и имеет документацию в Интернет по адресу <http://www.boost.org/tools/build/v2/>. Версией 2 этого инструмента обеспечивается поддержка приложений Qt 4, однако предполагается, что в переменной среды QDIR задан путь к установленному пакету Qt 4. В некоторых случаях в установленной утилите Boost.Build поддержка Qt отключена по умолчанию; в таком случае необходимо отредактировать файл user-config.jam и добавить следующую строку

```
using qt ;
```

Не полагаясь на QDIR, можно указать путь установки Qt, например, изменив предыдущую строку следующим образом:

```
using qt : /home/kathy/opt/qt432 ;
```

Для построения любого приложения утилитой Boost.Build требуется два файла: boost-build.jam и Jamroot. На самом деле, необходимо иметь только одну копию файла boost-build.jam, которая будет пригодна для любого количества приложений, если этот файл находится в каталоге, содержащем все подкаталоги приложений (независимо от глубины вложенности). Файл boost-build.jam должен содержать только одну строку, в которой указан путь к каталогу установки системы построения приложений. Например:

```
boost-build /home/kathy/opt/boost-build ;
```

Файл Jamroot, используемый для создания приложения Электронная таблица, будет иметь следующий вид:

```
using qt : /home/kathy/opt/qt432 ;
exe spreadsheet :
    cell.cpp
    finddialog.cpp
    finddialog.h
    gotocelldialog.cpp
    gotocelldialog.h
    gotocelldialog.ui
    main.cpp
    mainwindow.cpp
    mainwindow.h
    sortdialog.cpp
    sortdialog.h
    sortdialog.ui
    spreadsheet.cpp
    spreadsheet.h
    spreadsheet.qrc
    /qt//QtGui
    /qt//QtCore ;
```

В первой строке обеспечивается поддержка Qt 4, и нам необходимо обеспечить путь к установленному пакету Qt 4. Затем мы говорим, что необходимо построить приложение с именем spreadsheet, и что оно зависит от перечисленных файлов. Обычно нам не приходится задавать заголовочные файлы, потому что утилита Boost.Build достаточно «умна» и способна сама выявлять зависимости. Однако файлы .h, определяющие классы Q_OBJECT, должны обрабатываться компилятором moc, поэтому они здесь указаны. В последних двух строках указаны используемые библиотеки Qt.

Предполагая, что исполняемый модуль bjam находится в пути, указанном в переменной среды path, можно построить приложение, выполняя следующую команду:

```
bjam release
```

Это команда говорит утилите Boost.Build, что необходимо построить рабочую версию приложения, описанного в файле Jamroot, который находится в текущем каталоге. (Если у вас установлены только отладочные библиотеки Qt, будут получены сообщения об ошибках; в этом случае необходимо выполнить команду bjam debug). Утилиты moc, uic и rcc будут выполняться по мере необходимости. Для создания приложения с нуля необходимо выполнить команду bjam clean release.

SCons: утилита по созданию программного обеспечения

Утилита SCons, которую можно получить на сайте <http://www.scons.org/>, является инструментом создания приложений, который основан на применении языка Python, она имеет открытый исходный код и предназначена для замены утилиты make. Эта утилита обеспечивает поддержку Qt 3 и имеет расширение по поддержке Qt 4, разработанное Дэвидом Гарсия Гарсоном (David Garcia Garzyn) и доступное в Интернет по адресу <http://www.iaa.upf.edu/~dgarcia/Coddors/sconstools.html>. С указанного сай-

та необходимо загрузить один файл, `qt4.py`, и поместить его в каталог приложения. Ожидается, что это расширение будет включено в официальную версию SCons.

Поместив файл `qt4.py` в соответствующее место, можно создать файл `SConstruct`, описывающий процесс создания нашего приложения:

```
#!/usr/bin/env python

import os

QT4_PY_PATH = ".."
QTDIR = „/home/kathy/opt/qt432“

pkgpath = os.environ.get(„PKG_CONFIG_PATH“, „“)
pkgpath += „:s/lib/pkgconfig“ % QTDIR
os.environ[„PKG_CONFIG_PATH“] = pkgpath
os.environ[„QTDIR“] = QTDIR

env = Environment(tools=[„default“, „qt4“], toolpath=[QT4_PY_PATH])
env[„CXXFILESUFFIX“] = „.cpp“

env.EnableQt4Modules([„QtGui“, „QtCore“])
rccs = [env.Qrc(„spreadsheet.qrc“, QT4_QRCFLAGS="-name spreadsheet")]
uis = [env.Uic4(ui) for ui in [„gotocelldialog.ui“, „sortdialog.ui“]]
sources = [
    „cell.cpp“,
    „finddialog.cpp“,
    „gotocelldialog.cpp“,
    „main.cpp“,
    „mainwindow.cpp“,
    „sortdialog.cpp“,
    „spreadsheet.cpp“]

env.Program(target="spreadsheet", source=[rccs, sources])
```

Этот файл написан на языке Python, поэтому мы имеем доступ ко всем функциям и библиотекам этого языка.

Большая часть этого файла стандартна; требуется описать всего лишь несколько элементов, специфичных для конкретного приложения. Сначала задается путь к `qt4.py`, а затем путь к каталогу установки пакета Qt 4. Можно не копировать `qt4.py` в каталог каждого приложения, помещая этот файл в одно стандартное место и соответствующим образом задавая путь к нему. Мы должны в явном виде подключить необходимые модули (в данном случае это *QtCore* и *QtGui*) и должны указать файлы, которые необходимо обрабатывать утилитами `rcc` и `uic`. Наконец, указываются исходные файлы, и задается имя программы с указанием зависимости от исходных файлов и файлов ресурсов. Нам не приходится задавать файлы `.h`, потому что поддержка Qt 4 организована в утилите достаточно «умно», и компилятор `moc` выполняется по мере необходимости, обрабатывая файлы `.cpp`.

Теперь мы можем создать приложение, выполняя команду `scons`:

```
scons
```

В результате создается приложение, описанное в файле `SConstruct`, который находится в текущем каталоге. Приложение можно создать с нуля, выполняя команду `scons -c`.

В

- *Первое знакомство с Qt Jambi*
- *Использование Qt Jambi в рамках Eclipse IDE*
- *Интеграция компонентов C++ в Qt Jambi*

Приложение В. Введение в Qt Jambi

Qt Jambi представляет собой Java-версию среды разработки Qt-приложений. Основу Qt Jambi составляют библиотеки C++ средств разработки Qt, которые становятся доступны Java-программистам через интерфейс Java Native Interface (JNI). Несмотря на то, что значительные усилия были потрачены на обеспечение беспроблемной интеграции Qt Jambi с Java и естественности программного интерфейса этого компонента для Java-программистов, программный интерфейс Qt Jambi окажется все же знакомым и предсказуемым для Qt-программистов, использующих C++. Все классы документированы при помощи Javadoc, и эта документация находится на web-странице <http://doc.trolltech.com/qtjambi/>.

До сих пор Java-программистам, разрабатывающим приложения с графическим интерфейсом, приходилось иметь дело с AWT, Swing, SWT и подобными библиотеками классов графического интерфейса, ни одна из которых не может сравниться с Qt по удобству применения и эффективности. Например, в традиционных Java-библиотеках, обеспечивающих графический интерфейс, привязка действия пользователя, например, нажатие кнопки, к соответствующему методу требует написания класса обработки события (event listener class); в Qt Jambi для достижения того же результата достаточно написать лишь одну строку программного кода. Менеджеры компоновки Qt значительно проще использовать, чем BorderLayout и GridBagLayout из библиотеки Swing, и результат имеет более привлекательный вид.

Приложения Qt Jambi, как и приложения Qt, написанные на C++, могут иметь основные окна с полосой главного меню, панелями инструментов, пристыкованными окнами и строкой состояния. Кроме того, они имеют внешний вид и режим работы, естественный для платформы, на которой они выполняются, и в них учитываются пользовательские предпочтения относительно выбора общего стиля, цветов, шрифтов и т. п. Опираясь на всю мощь средств разработки Qt, приложения Qt Jambi могут использовать преимущества эффективной архитектуры двумерной графики Qt (особенно, средств графического представления объектов) и расширений, подобных OpenGL.

Qt Jambi полезен не только для Java-программистов. В частности, программисты C++ могут обеспечивать доступность своих пользовательских Qt-компонентов для Java-программистов, используя то же самое средство генерации программного кода, с помощью которого компания Trolltech обеспечивает доступность программного интерфейса Qt в рамках Qt Jambi.

В данном приложении мы покажем, как Java-программисты могут начать использовать Qt Jambi для создания приложений с графическим интерфейсом. Затем будут продемонстрированы способы применения Qt Jambi в рамках Eclipse совместно с *Qt Designer*, и, наконец, будет показано, как следует создавать пользовательские компоненты на C++, чтобы они были доступны программистам Qt Jambi. Предполагается, что вы знакомы с Qt-программированием в среде C++/Qt и с Java-программированием. Для работы Qt Jambi необходима версия Java 1.5 или выше.

Первое знакомство с Qt Jambi

В этом разделе мы разработаем небольшое Java-приложение, которое выводит на экран окно, показанное на рис. В.1. За исключением своего заголовка диалоговое окно Jambi Find имеет тот же вид и тот же режим работы, какое имеет диалоговое окно Find, созданное в главе 2. На одном и том же примере можно легко увидеть отличие и подобие Qt-программирования на C++ и программирования в среде Qt Jambi. При анализе этого программного кода мы будем обсуждать концептуальные отличия между C++ и Java по мере их обнаружения.



Рис. В.1. Диалоговое окно Jambi Find

Реализация приложения Jambi Find выполнена в одном файле с именем `FindDialog.java`. Мы подробно рассмотрим содержание этого файла, начиная с объявлений `import`.

```
import com.trolltech.qt.core.*;
import com.trolltech.qt.gui.*;
```

Эти два объявления `import` делают доступными для Java все базовые классы и классы графического интерфейса Qt. Доступ к дополнительным наборам классов можно обеспечить с помощью аналогичных объявлений `import` (например, `import com.trolltech.qt.opengl.*`).

```
public class FindDialog extends QDialog {
```

Класс `FindDialog` является подклассом `QDialog`, как в C++-версии этого примера. В C++ мы объявляем в заголовочном файле сигналы, полагаясь на генерацию соответствующего программного кода компилятором `moc`. В Qt Jambi для реализации механизма сигналов и слотов используются средства самоанализа

Java. Однако нам все-таки необходимо иметь какую-то возможность объявления сигналов, и это делается с помощью классов `SignalN`:

```
public Signal2<String, Qt.CaseSensitivity> findNext =
    new Signal2<String, Qt.CaseSensitivity>();
public Signal2<String, Qt.CaseSensitivity> findPrevious =
    new Signal2<String, Qt.CaseSensitivity>();
```

Всего имеется десять классов `SignalN`: `Signal0`, `Signal1<T1>`, ..., `Signal9<T1, ..., T9>`. Числа в их именах указывают на количество принимаемых аргументов, а `T1`, ..., `T9` определяют типы аргументов. В приведенном примере мы объявили два сигнала, каждый из которых имеет два аргумента. В обоих случаях первым аргументом является Java-строка (`String`), а вторым аргументом – тип `Qt.CaseSensitivity` (тип Java-перечисления). Там, где в программном интерфейсе Qt требуется строка `QString`, в Qt Jambi используется тип `String`.

В отличие от остальных классов `SignalN` сигнал `Signal0` не является обобщенным классом. Для создания сигнала без аргументов класс `Signal0` используется следующим образом:

```
public Signal0 somethingHappened = new Signal0();
```

Создав необходимые сигналы, мы можем перейти к реализации конструктора. Этот метод достаточно большой, поэтому мы его разобьем на три части.

```
public FindDialog(QWidget parent) {
    super(parent);

    label = new QLabel(tr("Find &what:"));
    lineEdit = new QLineEdit();
    label.setBuddy(lineEdit);

    caseCheckBox = new QCheckBox(tr("Match &case"));
    backwardCheckBox = new QCheckBox(tr("Search &backward"));

    findButton = new QPushButton(tr("&Find"));
    findButton.setDefault(true);
    findButton.setEnabled(false);

    closeButton = new QPushButton(tr("Close"));
```

Создание виджетов на Java отличается от C++ небольшими синтаксическими деталями. Обратите внимание на то, что `tr()` возвращает `String`, а не `QString`.

```
lineEdit.textChanged.connect(this, "enableFindButton(String)");
findButton.clicked.connect(this, "findClicked()");
closeButton.clicked.connect(this, "reject()");
```

Синтаксис связей сигналов и слотов в Qt Jambi немного отличается от синтаксиса в Qt-приложениях на C++, однако он по-прежнему прост и компактен. В целом, эти связи задаются в следующем виде:

```
отправитель.имяСигнала.connect(получатель, "имяСлота(T1, ..., TN)");
```

В отличие от среды C++/Qt здесь не надо указывать сигнатуру сигнала. Если сигнал имеет больше параметров, чем слоты, с которыми он соединен, дополнительно-

ные параметры игнорируются. Более того, в Qt Jambi механизм сигналов и слотов не ограничивается подклассами QObject: любой класс, который наследует QSignalEmitter, может генерировать сигналы, а любой метод любого класса может быть слотом.

```
QHBoxLayout topLeftLayout = new QHBoxLayout();
topLeftLayout.addWidget(label);
topLeftLayout.addWidget(lineEdit);

QVBoxLayout leftLayout = new QVBoxLayout();
leftLayout.addLayout(topLeftLayout);
leftLayout.addWidget(caseCheckBox);
leftLayout.addWidget(backwardCheckBox);

QVBoxLayout rightLayout = new QVBoxLayout();
rightLayout.addWidget(findButton);
rightLayout.addWidget(closeButton);
rightLayout.addStretch();

QHBoxLayout mainLayout = new QHBoxLayout();
mainLayout.addLayout(leftLayout);
mainLayout.addLayout(rightLayout);
setLayout(mainLayout);

setWindowTitle(tr("Jambi Find"));
setFixedHeight(sizeHint().height());
}
```

Программный код компоновки виджетов практически идентичен оригинальному коду на C++ — одни и те же классы компоновки работают совершенно одинаково. Кроме того, Qt Jambi может использовать формы, создаваемые *Qt Designer*, применяя *juic* (компилятор интерфейса пользователя для Java), как мы это увидим в следующем разделе.

```
private void findClicked() {
    String text = lineEdit.text();
    Qt.CaseSensitivity cs = caseCheckBox.isChecked()
        ? Qt.CaseSensitivity.CaseSensitive
        : Qt.CaseSensitivity.CaseInsensitive;
    if (backwardCheckBox.isChecked()) {
        findPrevious.emit(text, cs);
    } else {
        findNext.emit(text, cs);
    }
}
```

Синтаксис Java для ссылок на значения перечислений (enum) немного многословнее, чем на C++, однако он вполне понятен. Для генерации сигнала мы вызываем метод `emit()` для объекта `SignalN`, передавая аргументы соответствующих типов. Проверка типов выполняется на этапе компиляции программы.

```
private void enableFindButton(String text) {
    findButton.setEnabled(text.length() == 0);
}
```

Метод `enableFindButton()` по существу совпадает с оригиналом на C++.

```
private QLabel label;  
private QLineEdit lineEdit;  
private QCheckBox caseCheckBox;  
private QCheckBox backwardCheckBox;  
private QPushButton findButton;  
private QPushButton closeButton;
```

В соответствии с программным кодом, в остальной части книги мы объявляем все виджеты как закрытые поля класса. Это чисто стилевое решение; ничто не мешает нам объявить в конструкторе те виджеты, на которые делаются ссылки только в конструкторе. Например, переменные `label` и `closeButton` можно было бы объявить в конструкторе, поскольку они нигде больше не используются, и занимаемую ими память не пришлось бы освобождать с помощью механизма сборки мусора после завершения работы конструктора. Это происходит из-за того, что в Qt Jambi применяется такой же механизм родственных связей, какой существует в среде C++/Qt, поэтому сразу после размещения переменных `label` и `closeButton`, форма `FindDialog` становится их владельцем и сохраняет ссылку на них, препятствуя их уничтожению сборщиком мусора. Qt Jambi удаляет дочерние виджеты рекурсивно, поэтому если удаляется окно верхнего уровня, оно, в свою очередь, удаляет все свои дочерние виджеты и элементы компоновки, которые удаляют свои дочерние объекты, и это повторяется до тех пор, пока не будут уничтожены все дочерние объекты.

Использование системы Java-ресурсов

Qt Jambi учитывает в полной мере наличие системы Java-ресурсов в отличие от многих стандартных Java-классов. Java-ресурсы идентифицируются префиксом `classpath:.` Везде, где могло бы использоваться имя файла в программном интерфейсе Qt Jambi, вместо него применяется Java-ресурс. Например:

```
QIcon icon = new QIcon("classpath:/images/icon.png");  
if (!icon.isNull()) {  
    ...  
}
```

При поиске пиктограммы Qt Jambi будет просматривать каждый подкаталог `images` каждого каталога или `.jar`-файла, указанного в переменной среды `CLASSPATH`. Как только первый файл `icon.png` будет найден, поиск прекращается, и найденный файл будет использоваться в дальнейшем.

Никакое исключение не генерируется, если файл окажется ненайденным. В приведенном выше примере, если пиктограмма `icon.png` не найдена, метод `icon.isNull()` вернет значение `true`. В классах, например, `QImage` и `QPixmap`, которые имеют конструкторы, в качестве аргумента принимающие имя файла, предусмотрен метод `isNull()`, который можно использовать для проверки успешности считывания файла. При использовании класса `QFile` можно воспользоваться методом `QFile.error()`, чтобы убедиться в успешности считывания файла.

Qt Jambi в полной мере использует возможности механизма сбора мусора Java-платформы, поэтому в отличие от AWT, Swing и SWT после удаления последней ссылки на окно верхнего уровня для этого окна будет запланировано удаление сборщиком мусора и не потребуется явно вызывать функцию `dispose()`. Этот подход очень удобен, и он работает так же, как в среде C++/Qt. Основной особенностью этого подхода является необходимость в SDI-приложениях (однодокументных приложениях) сохранять ссылку на каждое создаваемое окно верхнего уровня, чтобы предотвратить их удаление сборщиком мусора. (В среде C++/Qt для предотвращения утечек памяти в SDI-приложениях обычно используется атрибут `Qt::WA_DeleteOnClose`.)

```
public static void main(String[] args) {
    QApplication.initialize(args);
    FindDialog dialog = new FindDialog(null);
    dialog.show();
    QApplication.exec();
}
```

Для удобства мы сопровождаем класс `FindDialog` методом `main()`, который создаст диалоговое окно и выдает его на экран. Объявление `import com.trolltech.qt.gui.*` обеспечивает доступность статического объекта `QApplication`. После запуска приложения Qt Jambi необходимо вызвать метод `QApplication.initialize()` и передать ему в командной строке значения аргументов. Это позволяет объекту `QApplication` обрабатывать предусмотренные аргументы, например, `-font` и `-style`.

При создании диалогового окна `FindDialog` мы передаем `null` в качестве родительского элемента, показывая, что данное диалоговое окно находится на самом верхнем уровне. После завершения метода `main()` диалоговое окно выйдет из области видимости и будет удалено при сборке мусора. Вызов `QApplication.exec()` запускает цикл обработки событий и возвращает управление в метод `main()` только после закрытия пользователем этого диалогового окна.

Программный интерфейс Qt Jambi очень напоминает программный интерфейс C++/Qt, однако существуют некоторые отличия. Например, в C++ функция-член `QWidget::mapTo()` имеет следующую сигнатуру:

```
QPoint mapTo(QWidget *widget, const QPoint &point) const;
```

Объект `QWidget` передается с помощью неконстантного указателя, в то время как `QPoint` передается с помощью константной ссылки. В Qt Jambi аналогичный метод имеет сигнатуру

```
public final QPoint mapTo(QWidget widget, QPoint point) { ... }
```

Поскольку в языке Java нет указателей, по сигнатурам методов нельзя визуально определить возможность или невозможность модификации в методе переданных ему объектов. Теоретически, метод `mapTo()` мог бы изменять оба своих параметра, поскольку они являются ссылками, однако Qt Jambi обещает не изменять аргумент `QPoint`, поскольку в C++ он передается как константная ссылка. Из контекста обычно видно, какие параметры могут изменяться, а какие нет. Если есть сомнения, можно обратиться к документации и прояснить ситуацию.

Кроме отсутствия изменений аргументов, передаваемых по значению или как константные ссылки в C++, Qt Jambi также обещает, что возвращаемое методом значение типа, отличного от `void`, которое в C++ возвращалось бы в виде значения или константной ссылки, имеет независимую копию, поэтому его изменение не будет иметь никаких побочных эффектов.

Ранее мы говорили, что в Qt Jambi вместо используемого в среде C++/Qt типа `QString` всегда применяется Java-строка `String`. Такой же тип соответствия применим к классу `QChar`, который имеет два эквивалента в Java: `char` и `java.lang.Character`. Аналогичные соответствия существуют относительно некоторых классов контейнеров Qt: `QHash` заменяется на `java.util.HashMap`, `QList` и `QVector` — на `java.util.List`, `QMap` — на `java.util.SortedMap`. Кроме того, `QThread` заменяется классом `java.lang.Thread`.

В архитектуре Qt модель/представление и в программном интерфейсе баз данных широко используется тип `QVariant`. В Java нет необходимости в этом типе, поскольку все Java-объекты наследуют `java.lang.Object`, поэтому повсюду в программном интерфейсе Qt Jambi тип `QVariant` заменяется типом `java.lang.Object`. Дополнительные методы, предоставляемые для типа `QVariant`, доступны в виде статических методов в `com.trolltech.qt.QVariant`.

Итак, мы завершили обзор небольшого примера Qt и обсудили многие концептуальные отличия программирования в средах Qt Jambi и C++/Qt. Построение и выполнение приложений Qt Jambi не отличаются от построения и выполнения других приложений Java за исключением того, что переменная среды `CLASSPATH` должна содержать каталог установки Qt Jambi. Класс необходимо откомпилировать с помощью компилятора Java и затем его можно выполнить, используя интерпретатор Java. Например:

```
export CLASSPATH=$CLASSPATH:$HOME/qtjambi/qtjambi.jar:$PWD
javac FindDialog.java
java FindDialog
```

Здесь мы использовали командную оболочку Bash для установки значения переменной среды `CLASSPATH`; при использовании других интерпретаторов командных строк синтаксис может отличаться. Мы включили текущий каталог в `CLASSPATH`, чтобы класс `FindDialog` был доступен. В системе Mac OS X необходимо в команде `java` задать опцию `-XstartOnFirstThread`, чтобы обеспечить выполнение соответствующего потока виртуальной машиной Java для Apple. В Windows для выполнения приложения необходимо задать следующие команды:

```
set CLASSPATH=%CLASSPATH%;%JAMBIPATH%\qtjambi.jar;%CD
javac FindDialog.java
java FindDialog
```

Qt Jambi может также использоваться в рамках IDE. В следующем разделе мы покажем, как можно редактировать, создавать и тестировать приложение Qt Jambi, используя популярную среду Eclipse IDE.

Применение Qt Jambi в Eclipse IDE

Eclipse IDE (далее «Eclipse» для краткости) — один из основных программных продуктов семейства Eclipse, состоящего из более шестидесяти проектов с открытым исходным кодом. Eclipse очень популярен у Java-программистов, и он мо-

жет работать на всех основных платформах, поскольку написан на Java. Eclipse выводит на экран ряд панелей, называемых видами. В Eclipse имеется много таких панелей, в том числе панели навигатора, структуры и редактора, а каждый конкретный набор панелей называется проекцией (perspective).

Чтобы Qt Jambi и *Qt Designer* были доступны в Eclipse, необходимо установить пакет по интеграции Qt Jambi в Eclipse. После распаковки пакета в соответствующий каталог Eclipse необходимо запустить с опцией `-clean`, заставляя его выполнить поиск новых подключаемых модулей, а не ограничиваться модулями расширения в его кеше. Диалоговое окно Preferences (Настройки) теперь будет иметь дополнительную страницу для настройки Qt Jambi («Qt Jambi Preference Page»). Необходимо перейти на эту страницу и установить место расположения Qt Jambi, как это объяснено на web-странице http://doc.trolltech.com/qtjambi-4.3.2_01/com/trolltech/qt/qtjambi-eclipse.html. После задания и проверки этого пути Eclipse следует закрыть и повторно стартовать, чтобы изменения вступили в силу.

При следующем запуске Eclipse и выборе пункта меню File|New Project (Файл|Новый проект), на экран выводится диалоговое окно нового проекта, в котором предусмотрено два вида проектов Qt Jambi: Qt Jambi Project (Проект Qt Jambi) и Qt Jambi Project (Using Designer Form) (Проект Qt Jambi с использованием формы дизайнера). В этом разделе мы обсудим оба типа этих проектов, создав версию Qt Jambi для примера Go to Cell, разработанного в главе 2.

Для создания приложения Qt Jambi чисто программно выберите пункт меню File|New Project, затем режим «Qt Jambi Project» и пройдите по всем страницам мастера. В конце Eclipse создаст проект с каркасным файлом `.java`, содержащим метод `main()` и конструктор. Выполнять приложение в рамках Eclipse можно с помощью пункта меню Run|Run (Выполнить|Выполнить). Синтаксические ошибки Eclipse показывает в левом поле редактора, а ошибки этапа выполнения выводятся в окне консоли.

Создание приложения Qt Jambi с использованием *Qt Designer* очень напоминает создание приложения чисто программным способом. Выберите пункт меню File|New Project и затем выберите режим «Qt Jambi Project (Using Designer Form)». Вновь на экране появится мастер. Назовите проект «JambiGoToCell», а на последней странице задайте «GoToCellDialog» в качестве имени класса и выберите «Dialog» в качестве типа формы.

После того, как мастер завершит свою работу, будет создан файл `GoToCellDialog.java`, а также файл интерфейса пользователя `JavaGoToCellDialog.jui`. Дважды щелкните по файлу `.juj`, чтобы вызвать редактор *Qt Designer*. На экране появится форма с кнопками OK и Cancel. Для доступа к функциональности *Qt Designer*, выберите пункт меню Window|Open Perspective|Other (Окно|Открыть|Другие), затем дважды щелкните по пункту *Qt Designer UI* (Интерфейс пользователя *Qt Designer*). Данная проекция содержит редактор сигналов и слотов *Qt Designer*, редактор действий, редактор свойств, окно виджетов и другие панели, как показано на рис. В.2.

Для завершения проектирования мы выполняем такие же действия, какие указаны в главе 2. Можно выполнить предварительный просмотр диалогового окна, как это делается в *Qt Designer*, и поскольку Eclipse генерирует каркас программного кода, его также можно выполнить, выбирая пункт меню Run|Run.

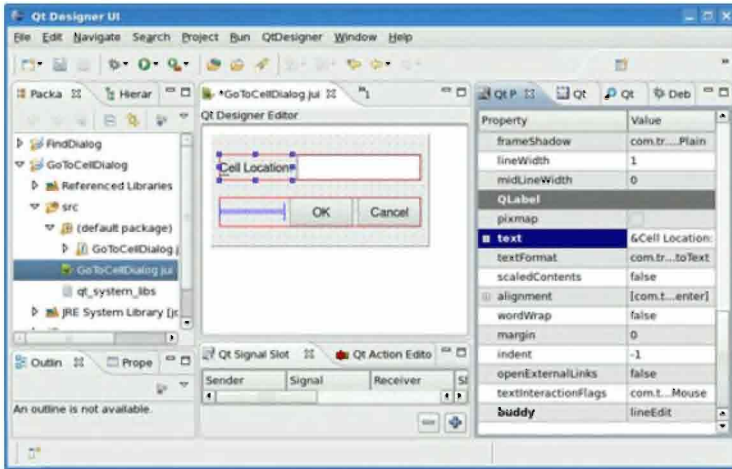


Рис. В.2. Диалоговое окно Go to Cell в Eclipse

На заключительном этапе выполняется редактирование файла `GoToCellDialog.java` для реализации необходимой нам функциональности. Eclipse генерирует два конструктора, однако нам потребуется только один из них, поэтому мы удаляем конструктор без параметров. В сгенерированном методе `main()` необходимо передать конструктору `GoToCellDialog` `null` в качестве родительского объекта. Кроме того, необходимо реализовать конструктор `GoToCellDialog(QWidget parent)` и предусмотреть метод `on_lineEdit_textChanged(String)`, который будет выполняться при генерации сигнала `textChanged()`. Ниже приводится полученный файл `GoToCellDialog.java`:

```
import com.trolltech.qt.core.*;
import com.trolltech.qt.gui.*;

public class GoToCellDialog extends QDialog {
    private Ui_GoToCellDialogClass ui = new Ui_GoToCellDialogClass();

    public GoToCellDialog(QWidget parent) {
        super(parent);
        ui.setupUi(this);
        ui.okButton.setEnabled(false);
        QRegExp regExp = new QRegExp("[A-Za-z][1-9][0-9]{0,2}");
        ui.lineEdit.setValidator(new QRegExpValidator(regExp, this));
        ui.okButton.clicked.connect(this, "accept()");
        ui.cancelButton.clicked.connect(this, "reject()");
    }

    private void on_lineEdit_textChanged(String text) {
        ui.okButton.setEnabled(!text.isEmpty());
    }
}
```

```

public static void main(String[] args) {
    QApplication.initialize(args);
    GoToCellDialog testGoToCellDialog = new GoToCellDialog(null);
    testGoToCellDialog.show();
    QApplication.exec();
}
}

```

Eclipse обеспечивает вызов `juic` для преобразования `GoToCellDialog.jui` в файл `Ui_GoToCellDialogClass.java`, который определяет класс с именем `Ui_GoToCellDialogClass`, воспроизводящий диалоговое окно, спроектированное нами с помощью *Qt Designer*. Мы создаем экземпляр этого класса и храним ссылку на него в закрытой переменной `ui`.

В конструкторе вызывается метод `setupUi()` класса `Ui_GoToCellDialogClass` для создания и размещения виджетов, а также для задания их свойств и связей сигналов и слотов, включая автоматические связи для слотов, имена которых имеют вид `оп_имяОбъекта_имяСигнала()`

Удобство интеграции в Eclipse проявляется в том, что достаточно просто можно размещать подклассы `QWidget` в панели виджетов, откуда их можно перетаскивать в формы. Мы кратко рассмотрим пользовательский виджет `LabeledLineEdit` и затем покажем, как можно обеспечить доступ к нему из панели виджетов *Qt Designer*.

При создании пользовательских виджетов, используемых в *Qt Designer*, часто полезно обеспечить свойства, которые программист может изменять для индивидуальной настройки виджета. На рис. В.3 показана форма с пользовательским виджетом `LabeledLineEdit`. Этот виджет имеет два пользовательских свойства, `labelText` и `editText`, значения которых могут задаваться в редакторе свойств. В среде C++/Qt свойства определяются с помощью макроса `Q_PROPERTY()`. В Qt Jambi анализ внутреннего состояния приложения (introspection) позволяет обнаруживать пары методов для доступа к переменным, имена которых соответствуют принятым в Qt соглашениям по назначению имен и имеют вид `xxx()/setXxx()` или соответствуют принятым в языке Java соглашениям по назначению имен и имеют вид `getXxx()/setXxx()`. Можно также определять другие свойства и скрывать автоматически обнаруживаемые свойства, используя аннотации.

```

import com.trolltech.qt.*;
import com.trolltech.qt.gui.*;

public class LabeledLineEdit extends QWidget {

```

Первое объявление `import` необходимо для получения доступа к аннотациям `Qt Jambi @QtPropertyReader()` и `@QtPropertyWriter()`, что позволяет экспортировать свойства в *Qt Designer*.

```

    @QtPropertyReader(name="labelText")
    public String labelText() { return label.text(); }

    @QtPropertyWriter(name="labelText")
    public void setLabelText(String text) { label.setText(text); }

```

```
@QtPropertyReader(name="editText")
public String editText() { return lineEdit.text(); }

@QtPropertyWriter(name="editText")
public void setEditText(String text) { lineEdit.setText(text); }
```

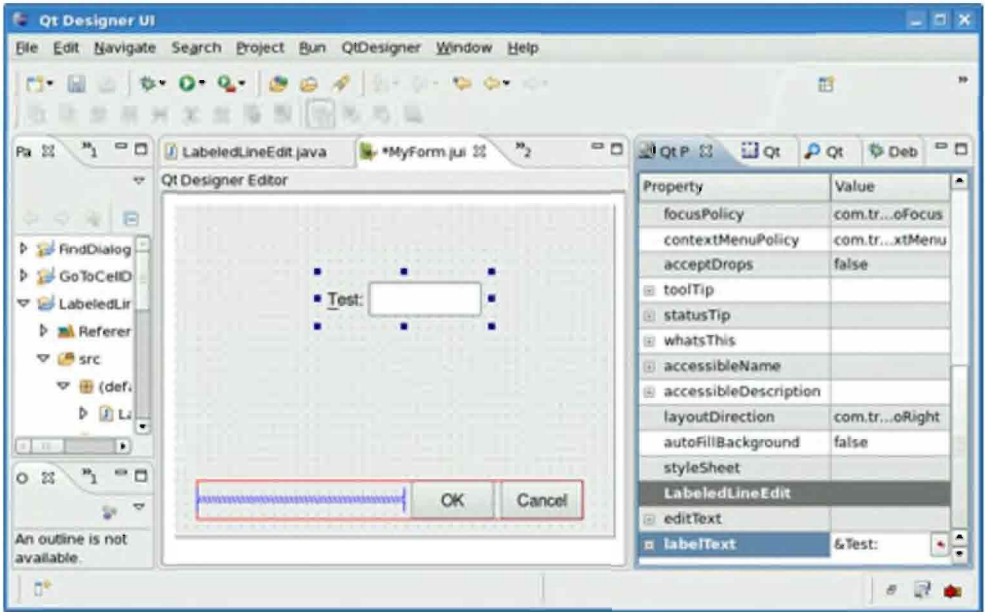


Рис. В.3. Форма с пользовательским виджетом LabeledLineEdit

Для свойств, которые допускается только считывать, можно просто использовать аннотацию `@QtPropertyReader()` и предоставить метод получения значения свойства. В нашем случае нужны свойства, которые можно считывать и записывать, поэтому каждое свойство имеет `get`-метод и `set`-метод. В этом примере можно было бы не использовать аннотации, поскольку методы доступа к переменным соответствуют принятым в Qt соглашениям по назначению имен.

```
public LabeledLineEdit(QWidget parent){
    super(parent);

    label = new QLabel();
    lineEdit = new QLineEdit();
    label.setBuddy(lineEdit);

    QHBoxLayout layout = new QHBoxLayout();
    layout.addWidget(label);
    layout.addWidget(lineEdit);
    setLayout(layout);
}

private QLabel label;
private QLineEdit lineEdit;
```

Здесь используется вполне обычный конструктор. Чтобы можно было использовать виджет на панели виджетов, мы должны предоставить конструктор с одним аргументом типа `QWidget`.

```
public static void main(String[] args) {
    QApplication.initialize(args);
    LabeledLineEdit testLabeledLineEdit = new LabeledLineEdit(null);
    testLabeledLineEdit.setLabelText("&Test");
    testLabeledLineEdit.show();
    QApplication.exec();
}
```

Eclipse генерирует автоматически метод `main()`. Мы передали `null` в качестве аргумента конструктора `LabeledLineEdit` и добавили строку для установки значения свойства `text`.

Пользовательский виджет мы можем добавить в проект, копируя его файл `.java` в каталог проекта `src`. Затем вызывается диалоговое окно `Properties` (Свойства) и выбирается страница *Qt Designer Plugins* (Подключаемые модули *Qt Designer*). На этой странице приводится список всех подходящих подклассов `QWidget`, которые имеются в проекте. Необходимо установить флажок `Enable plugin` (Включить подключаемый модуль) для всех подклассов, которые должны появиться на панели виджетов, и нажать кнопку `ОК`. При последующем редактировании формы с использованием *Qt Designer*, интегрированного в Eclipse, добавленные в проект виджеты как подключаемые модули появятся внизу панели виджетов в отдельной секции.

На этом завершается краткое введение по применению Eclipse при программировании в среде Qt Jambi. Eclipse обеспечивает мощную и многофункциональную среду IDE, в которой очень удобно разрабатывать программное обеспечение. Интеграция Qt Jambi в Eclipse позволяет создавать приложения Qt Jambi полностью в среде Eclipse, в том числе с применением *Qt Designer*. Кроме того, компания Trolltech обеспечивает интеграцию в Eclipse среды C++/Qt почти с тем же успехом, как это сделано с Qt Jambi.

Интеграции компонентов C++ в Qt Jambi

Qt Jambi позволяет программистам C++ легко интегрировать свой Qt-код в Java-программы. Чтобы наши собственные пользовательские компоненты C++ были доступны в Qt Jambi, можно использовать генератор Qt Jambi Generator, который обрабатывает набор заголовочных файлов C++ и файл XML, содержащий некоторую информацию о нужных нам классах C++, и создает Java-привязки для наших компонентов C++. Сам интерфейс Qt Jambi создан при помощи этого генератора.

Этот процесс проиллюстрирован на рис. В.4. По окончании работы генератора мы получаем некоторые Java-файлы, которые должны быть откомпилированы с использованием Java-компилятора, и ряд файлов `.h` и `.cpp`, которые должны компилироваться с формированием совместно используемой библиотеки C++.

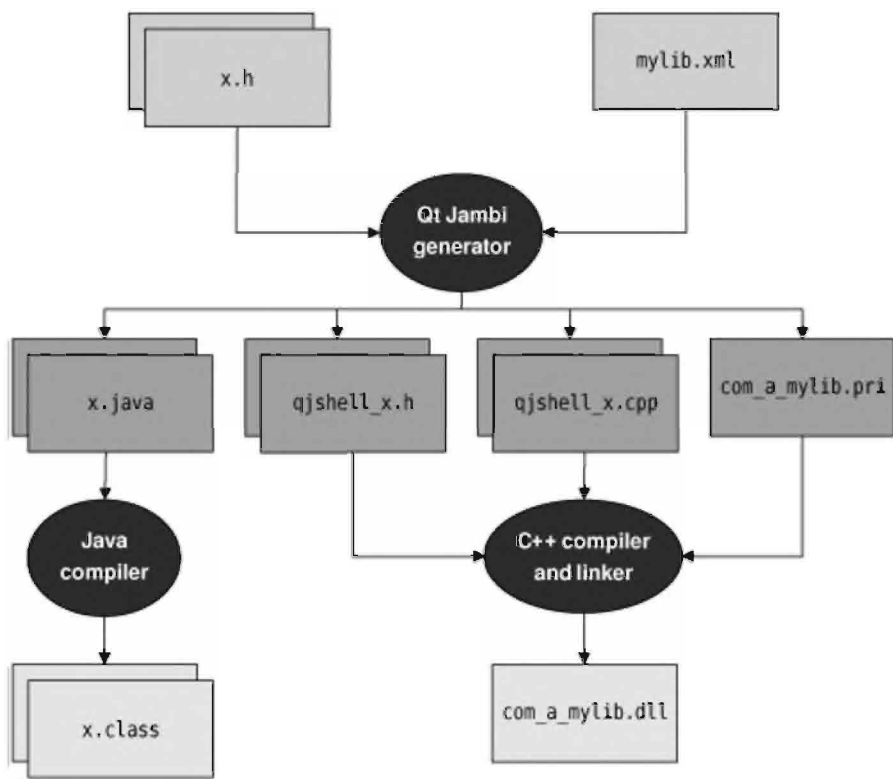


Рис. В.4. Создание классов C++, доступных в Qt Jambi

Для иллюстрации этого процесса мы создадим Java-привязки для класса `PlotSettings`, содержащего только поля данных, и для виджета `Plotter`, созданного в главе 5. Затем мы их используем в Java-приложении. На рис. В.5 показан результат работы этого приложения.

В заголовочном файле необходимо определить (или включить в него другие заголовочные файлы с этими определениями) все классы, которые потребуются для построения библиотеки:

```
#include <QtGui>

#include "../plotter/plotter.h"
```

Предполагается, что программный код примера `Plotter` находится в каталоге, расположенном параллельно оболочке Qt Jambi. Нам также потребуется XML-файл, в котором указано, что и в каком виде должно содержаться в оболочке. Этот файл имеет имя `jambiplotter.xml`:

```
<typesystem package="com.softwareinc.plotter"
  default-superclass="com.trolltech.qt.QtJambiObject">
  <load-typesystem name=":/trolltech/generator/typesystem_core.txt"
    generate="no" />
```

```

<load-typesystem name=":/trolltech/generator/typesystem_gui.txt"
                generate="no" />
<object-type name="Plotter" />
<value-type name="PlotSettings" />
</typesystem>

```

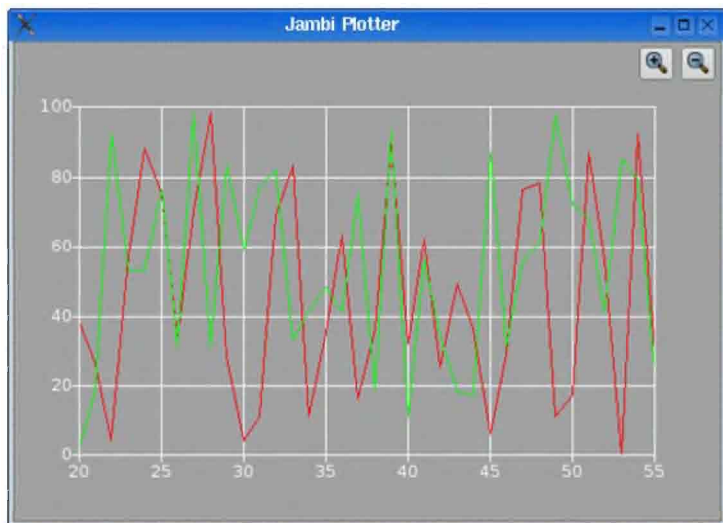


Рис. В.5. Приложение Jambi Plotter

Во внешнем теге мы указываем имя пакета Java для компонента `Plotter`, `com.softwareinc.plotter`. Теги `<load-typesystem>` используются для импорта информации, относящейся к модулям `QtCore` и `QtGui`.

Теги `<object-type>` и `<value-type>` определяют два класса C++, которые мы собираемся создать. Мы указали, что класс `Plotter` имеет тип объекта C++; это удобно для объектов, которые нельзя копировать, например, для виджетов. Напротив, класс `PlotSettings` рассматривается как класс «типа значения» C++.

Для пользователей Qt Jambi между этими двумя типами нет очевидной разницы. Важное отличие проявляется, когда генератор отображает программные интерфейсы C++ на программные интерфейсы Java. Например, если метод возвращает «тип-значение», генератор обеспечит независимость возвращаемого объекта (чтобы избежать побочных эффектов), однако если возвращаемый тип является объектом, вызывающая программа получает ссылку на исходный объект.

Нам потребуются две переменные среды: одна для указания пути к Qt Jambi, а другая для указания пути к Java. Ниже мы показываем, как они устанавливаются в системах Unix (используя командную оболочку Bash):

```

export JAMBIPATH=$HOME/qtjambi-linux32-gpl-4.3.2_01
export JAVA=/usr/java/jdk1.6.0_02

```

В Windows мы бы написали

```

set JAMBIPATH=C:\QtJambi
set JAVA="C:\Program Files\Java\jdk1.6.0_02"

```

Естественно, номер версии и каталог могут быть другими в вашей системе. В дальнейшем мы будем предполагать, что существуют переменные среды JAMBIPATH и JAVA. Используя заголовочный файл и XML-файл, мы можем запустить генератор из консольного режима:

```
$JAMBIPATH/bin/generator jambiplotter.h jambiplotter.xml
```

Если доступ к Qt Jambi обеспечен локально, а не по всей системе, эта команда в некоторых системах может завершиться неудачей. Решение состоит в обеспечении пути к библиотекам Qt Jambi:

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$JAMBIPATH/lib
```

Здесь вновь используется синтаксис оболочки Bash. В системе Mac OS X соответствующая переменная среды имеет имя DYLD_LIBRARY_PATH. В Windows тот же результат достигается добавкой в переменную среды PATH:

```
set PATH=%PATH%;%JAMBIPATH%\lib
```

Теперь в Windows генератор можно запускать следующим образом:

```
%JAMBIPATH%\bin\generator jambiplotter.h jambiplotter.xml
```

Генератор программного кода, немного проработав, выдает на консоль некоторые сводные данные. В нашем случае он создает следующие файлы в двух параллельных каталогах:

```
../com/softwareinc/plotter/PlotSettings.java
../com/softwareinc/plotter/Plotter.java
../com/softwareinc/plotter/QtJambi_LibraryInitializer.java

../cpp/com_softwareinc_plotter/com_softwareinc_plotter.pri
../cpp/com_softwareinc_plotter/metainfo.cpp
../cpp/com_softwareinc_plotter/metainfo.h
../cpp/com_softwareinc_plotter/qtjambi_libraryinitializer.cpp
../cpp/com_softwareinc_plotter/qtjambishell_PlotSettings.cpp
../cpp/com_softwareinc_plotter/qtjambishell_PlotSettings.h
../cpp/com_softwareinc_plotter/qtjambishell_Plotter.cpp
../cpp/com_softwareinc_plotter/qtjambishell_Plotter.h
```

Необходимо откомпилировать как файлы Java, так и файлы C++, однако перед этим необходимо убедиться, что правильно задана переменная среды CLASSPATH. Например, если используется оболочка Bash, можно выполнить следующую команду:

```
export CLASSPATH=$CLASSPATH:$JAMBIPATH/qtjambi.jar:$PWD:$PWD/..
```

В этом случае к существующему значению переменной среды CLASSPATH добавляется файл .jar Qt Jambi, текущий каталог и его родительский каталог. Родительский каталог необходим для того, чтобы можно было получить доступ к параллельному каталогу com. В Windows эта команда имела бы следующий вид:

```
set CLASSPATH=%CLASSPATH%;%JAMBIPATH%\qtjambi.jar;%CD%;%CD%\..
```

Теперь мы можем компилировать файлы .java, создавая файлы .class:

```
cd ../com/softwareinc/plotter
javac *.java
```


После компиляции файлов `.java` необходимо вернуться в каталог `jambiplotter`. Здесь на следующем шаге создается совместно используемая библиотека C++, которая содержит программный код на C++ для классов `Plotter` и `PlotSettings`, а также генератором формируется программный код оболочки на C++. Мы начинаем с создания файла `.pro`:

```
TEMPLATE       = lib
TARGET         = com_softwareinc_plotter
DLLDESTDIR     = .
HEADERS        = ../plotter/plotter.h
SOURCES        = ../plotter/plotter.cpp
RESOURCES      = ../plotter/plotter.qrc
INCLUDEPATH    += ../plotter \
                  $$$(JAMBIPATH)/include \
                  $$$(JAVA)/include

unix {
    INCLUDEPATH += $$$(JAVA)/include/linux
}
win32 {
    INCLUDEPATH += $$$(JAVA)/include/win32
}
LIBS += -L$$$(JAMBIPATH)/lib -lqtjambi

include(../cpp/com_softwareinc_plotter/com_softwareinc_plotter.pri)
```

Переменная `TEMPLATE` должна быть установлена на `lib`, поскольку мы собираемся создавать совместно используемую библиотеку, а не приложение. В переменной `TARGET` задается имя Java-пакета, но с символами подчеркивания вместо точек, а переменная `DLLDESTDIR` определяет, куда следует помещать совместно используемую библиотеку (или DLL). Переменную `INCLUDEPATH` необходимо дополнить каталогом исходных текстов (поскольку в нашем случае он не является текущим каталогом), путем к каталогу `include Qt Jambi`, путем к каталогу `include Java SDK` и путем к каталогу зависимой от платформы части Java SDK. (В приложении Б приводится синтаксис для `unix` и `win32`.) Необходимо также включить саму библиотеку `Qt Jambi`, что делается в элементе `LIBS`. Директива `include()`, указанная в конце файла, используется для получения доступа к файлам C++, сформированным генератором. Имея файл `.pro`, можно выполнять команды `qmake` и `make`, как это обычно делается для построения библиотеки.

Теперь, когда мы имеем совместно используемую библиотеку и подходящую Java-оболочку, их можно использовать в приложении. Приложение `Jambi Plotter` создает объекты `PlotSettings` и `Plottert`, используя их для вывода на экран некоторых произвольных данных. Важно то, что они используются точно так же, как любые другие классы Java или `Qt Jambi`. Все приложение достаточно небольшое, поэтому мы приводим его полностью:

```
import java.lang.Math;
import java.util.ArrayList;
import com.trolltech.qt.core.*;
import com.trolltech.qt.gui.*;
```

```

import com.softwareinc.plotter.Plotter;
import com.softwareinc.plotter.PlotSettings;

public class JambiPlotter {
    public static void main(String[] args) {
        QApplication.initialize(args);

        PlotSettings settings = new PlotSettings();
        settings.setMinX(0.0);
        settings.setMaxX(100.0);
        settings.setMinY(0.0);
        settings.setMaxY(100.0);

        int numPoints = 100;
        ArrayList<QPointF> points0 = new ArrayList<QPointF>();
        ArrayList<QPointF> points1 = new ArrayList<QPointF>();
        for (int x = 0; x < numPoints; ++x) {
            points0.add(new QPointF(x, Math.random() * 100));
            points1.add(new QPointF(x, Math.random() * 100));
        }

        Plotter plotter = new Plotter();
        plotter.setWindowTitle(plotter.tr("Jambi Plotter"));
        plotter.setPlotSettings(settings);
        plotter.setCurveData(0, points0);
        plotter.setCurveData(1, points1);
        plotter.show();

        QApplication.exec();
    }
}

```

Мы импортируем две стандартные библиотеки Java и необходимые нам библиотеки Qt Jambi, а также классы PlotSettings и Plotter. С тем же успехом можно было бы написать `import com.softwareinc.plotter.*`.

После инициализации объекта QApplication создается объект PlotSettings и задаются некоторые значения его полей. В версии C++ этого класса поля `minX`, `maxX`, `minY` и `maxY` определяются как открытые переменные. Если не оговорено обратное, генератор Qt Jambi формирует методы доступа к таким переменным, используя соглашения по назначению имен в Qt (например, `minX()` и `setMinX()`). Выполнив соответствующую настройку построителя графиков, мы генерируем два набора данных кривой, по 100 произвольных точек в каждом. В среде C++/Qt точки каждой кривой хранятся в векторе `QVector<QPointF>`; в Qt Jambi вместо векторов мы используем `ArrayList<QPointF>`, рассчитывая на то, что Qt Jambi выполнит соответствующее преобразование.

Задав настройки и данные кривой, мы создаем новый объект Plotter, который не имеет родительского элемента (чтобы построитель графиков представлял окно верхнего уровня). Затем мы передаем построителю графиков настройки графика, данные двух кривых и вызываем `show()`, чтобы сделать построитель

графиков видимым. Наконец, мы вызываем `QApplication.exec()` для запуска цикла обработки событий.

При компиляции приложения мы должны быть достаточно аккуратными и использовать не только обычный `CLASSPATH`, но также путь к пакету `com.softwa-reinc.plotter`. Для выполнения приложения необходимо задать две дополнительные переменные среды, чтобы загрузчик мог найти Qt Jambi и наши Java-при-вязки. В Windows мы должны установить `PATH` следующим образом:

```
set PATH=%PATH%;%JAMBIPATH%\bin;%CD%
```

В системе Unix мы должны задать `LD_LIBRARY_PATH`: `export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$JAMBIPATH/lib:$PWD`

В системе Mac OS X переменная среды имеет имя `DYLD_LIBRARY_PATH`. Кроме того, во всех трех платформах необходимо задать в `QT_PLUGIN_PATH` каталог Qt Jam-bi plugins. Например:

```
set QT_PLUGIN_PATH=%QT_PLUGIN_PATH%;%JAMBIPATH%\plugins
```

Для компиляции и выполнения приложения Jambi Plotter благодаря нашим добавкам в `CLASSPATH` можно на всех платформах просто выполнить следующие команды:

```
javac JambiPlotter.java
java JambiPlotter
```

Теперь наш пример завершен, и вы знаете, как можно использовать компо-ненты C++ в приложениях Qt Jambi. На самом деле генератор Qt Jambi, который играет главную роль в обеспечении доступности к компонентам C++ из Java-про-граммы, обладает многими другими возможностями, которые не были рассмот-рены в этом кратком введении. Хотя генератор поддерживает только подмножес-тво C++, это подмножество содержит все наиболее используемые конструкции C++, включая множественное наследование и перегрузку операторов. Полную документацию по этому генератору можно найти в Интернет по адресу http://doc.trolltech.com/qtjambi-4.3.2_01/com/trolltech/qt/qtjambi-generator.html. Мы кратко рассмотрим некоторые его возможности, которые не потребовались в примере, чтобы вы могли почувствовать его достоинства.

Решающее влияние на работу генератора оказывает содержание XML-фай-ла. Например, если мы имеем множественное наследование в C++, в одном из прямых способов его реализации в Qt Jambi просто один из классов определяется как класс типа объект или типа значение, а все другие классы как классы типа интерфейса.

Кроме того, можно использовать более естественные сигнатуры методов. Например, вместо объекта `ArrayList<QPointF>`, используемого для передачи то-чек кривой, было бы лучше использовать массив `QPointF`. Для этого необходимо скрыть исходный метод C++, заменяя его Java-методом, принимающего `QPointF[]` и вызывающего «за кулисами» исходный метод C++. Это можно сделать, моди-фицируя файл `jambiplotter.xml`. Первоначально мы определили тип C++ `Plotter` с помощью следующего элемента:

```
<object-type name="Plotter" />
```

Эту строку мы заменим более сложной версией:

```
<object-type name="Plotter">
  <modify-function
    signature="setCurveData(int, const QVector<QPointF> &)">
    <access modifier="private" />
    <rename to="setCurveData_private" />
  </modify-function>
  <inject-code>
    public final void setCurveData(int id,
                                   com.trolltech.qt.core.QPointF points[]) {
      setCurveData_private(id, java.util.Arrays.asList(points));
    }
  </inject-code>
</object-type>
```

В элементе `<modify-function>` метод C++ `setCurveData()` переименовывается на `setCurveData_private()`, а его спецификатор доступа изменяется на `private`, тем самым запрещая доступ к нему за рамками методов собственного класса. Обратите внимание, что следует избегать применения специальных XML-символов «<», «>» и «&» в атрибуте `signature`.

В элементе `<inject-code>` мы реализуем на Java пользовательский метод `setCurveData()`. Этот метод принимает аргумент типа `QPointF[]` и просто вызывает закрытый метод `setCurveData_private()`, преобразуя массив в список, требуемый методом C++.

Теперь можно создавать и заполнять более естественно массивы `QPointF`:

```
QPointF[] points0 = new QPointF[numPoints];
QPointF[] points1 = new QPointF[numPoints];
for (int x = 0; x < numPoints; ++x) {
    points0[x] = new QPointF(x, Math.random() * 100);
    points1[x] = new QPointF(x, Math.random() * 100);
}
```

Остальная часть программного кода не изменяется.

В некоторых случаях при создании оболочек классов генератор формирует программный код, использующий тип `QNativePointer`. Этот тип представляет собой оболочку указателей C++ для объектов типа значение, и он работает с указателями и массивами. Файл `mjb_nativepointer_api.log` определяет любой сгенерированный код, использующий тип `QNativePointer`. В документации по Qt Jambi рекомендуется заменять любой код с типом `QNativePointer` и даются подробные инструкции, как это делать.

Иногда требуется включать дополнительные объявления `import` в сгенерированные файлы `.java`. Это легко сделать с помощью тега `<extra-includes>`, а полное описание можно найти в Интернет по адресу http://doc.trolltech.com/qtjambi-4.3.2_01/com/trolltech/qt/qtjambi-typesystem.html.

Генератор Qt Jambi – мощный и универсальный инструмент по обеспечению доступности классов C++/Qt для Java-программистов, позволяющий объединить достоинства C++ и Java в одном проекте.

Г

- *Первое знакомство с C++*
- *Основные отличия языков*
- *Стандартная библиотека C++*

Приложение Г. Введение в C++ для программистов Java и C#

Данное приложение представляет собой краткое введение в язык C++, предназначенное для разработчиков, знакомых с Java или C#. Предполагается, что вы знакомы с такими концепциями объектно-ориентированного программирования, как наследование и полиморфизм, и хотите обучиться программированию на C++. Чтобы эта книга не стала громоздким 1500-страничным томом, включающим в себя полный учебник по C++ для начинающих, это приложение ограничивается изложением только существенных вопросов. В нем представлены основные понятия и методы, необходимые для понимания программ, приводимых в остальной части книги, и достаточные для того, чтобы, используя Qt, начать разработку межплатформенных приложений с графическим пользовательским интерфейсом.

На момент написания книги язык C++ представляет собой единственное реальное средство разработки межплатформенных, высокопроизводительных объектно-ориентированных приложений с графическим пользовательским интерфейсом. Недоброжелатели C++ обычно отмечают, что программировать на Java или C#, который отошел от поддержки совместимости с языком C, более приятно; на самом деле, Бьерн Страуструп (Bjarne Stroustrup), создатель C++, отмечал в книге «The Design and Evolution of C++» (Addison-Wesley, 1994), что «внутри C++ существует очень компактный и более аккуратный язык, изо всех сил стремящийся получить известность».

К счастью, при программировании в рамках Qt мы обычно придерживаемся некоторого подмножества C++, которое сильно приближается к утопическому языку, о котором говорил Страуструп, что позволяет нам сконцентрировать свое внимание непосредственно на текущей проблеме. Более того, Qt в некоторых аспектах расширяет C++, благодаря своему новаторскому механизму «сигналов и слотов», поддержке кодировки Unicode и ключевому слову `foreach`.

В первом разделе данного приложения мы увидим, как можно объединять несколько файлов, содержащих исходный код C++, для получения исполняемой программы. Это приведет нас к изучению таких центральных концепций C++, как единица компиляции, заголовочные файлы, объектные файлы, библиотеки, и познакомит с препроцессором, компилятором и компоновщиком C++.

Затем мы рассмотрим наиболее важные отличия языков C++, Java и C#, связанные с определением классов, использованием указателей и ссылок, перегрузками операторов, применением препроцессора и т. д. Несмотря на то, что синтаксис C++ очень похож на синтаксис Java и C#, имеется тонкое отличие базовых концепций. В то же время язык C++, под влиянием которого создавались Java и C#, имеет много общего с этими двумя языками, в частности аналогичные типы данных, те же самые арифметические операторы и одинаковые основные операторы управления.

Последней раздел посвящен стандартной библиотеке C++, которая обеспечивает функциональность, готовую к применению в любой программе на C++. Эта библиотека развивалась в течение более тридцати лет и поэтому вобрала в себя многие подходы, в том числе процедурный, объектно-ориентированный и функциональный стили программирования, а также макросы и шаблоны. По сравнению с библиотеками Java и C# стандартная библиотека C++ имеет довольно ограниченную область применения; она не поддерживает программирование графического пользовательского интерфейса, многопоточную обработку, базы данных, интернационализацию, работу с сетями, XML и Unicode. Для применения C++ в этих областях предполагается, что программисты C++ должны использовать различные библиотеки (часто зависящие от платформы) сторонних поставщиков.

Именно здесь приходит на помощь Qt. Сначала средства разработки Qt представляли собой межплатформенный инструментарий по созданию графического пользовательского интерфейса (набор классов, позволяющий писать переносимые приложения с графическим пользовательским интерфейсом), но затем они быстро превратились в полномасштабную среду разработки приложений, частично расширяющую и частично заменяющую стандартную библиотеку C++. Хотя эта книга посвящена средствам разработки Qt, полезно знать возможности стандартной библиотеки C++, поскольку вам, возможно, придется работать с программным кодом, использующим эту библиотеку.

Первое знакомство с C++

Программа C++ состоит из одной или нескольких единиц компиляции. Каждая единица компиляции представляет собой отдельный файл исходного кода, обычно имеющий расширение `.cpp` (другими распространенными расширениями являются `.cc` и `.cxx`); она обрабатывается компилятором за один шаг. Для каждой единицы компиляции компилятор генерирует объектный файл с расширением `.obj` (в Windows) или `.o` (в Unix и Mac OS X). Объектный файл – это бинарный файл, содержащий машинный код для той архитектуры, на которой будет выполняться программа.

После компиляции всех файлов `.cpp` мы можем собрать все объектные файлы для создания исполняемого модуля, используя специальную программу, называемую *компоновщиком* (*linker*). Компоновщик соединяет объектные файлы в единое целое и назначает адреса памяти функциям и другим символическим ссылкам, которые содержатся в единицах компиляции.

При создании программы только одна единица компиляции должна иметь функцию `main()`, которая является точкой входа в программу. Эта функция не принадлежит никакому классу – она является глобальной функцией. Этот процесс схематично показан на рис. Г.1.

В отличие от Java, где каждый исходный файл должен содержать точно один класс, C++ позволяет организовать единицу компиляции удобным для нас способом. Можно реализовать несколько классов в одном файле `.cpp` или распространить реализацию класса на несколько файлов `.cpp`; имена исходных файлов могут быть любыми. При внесении изменений в один конкретный файл `.cpp` потребуется перекомпилировать только этот файл и затем повторно скомпоновать приложение для создания нового исполняемого модуля.

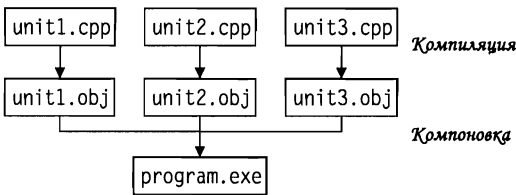


Рис. Г.1. Процесс компиляции программы на C++ (в Windows)

Прежде чем мы пойдем дальше, давайте рассмотрим очень простую программу на C++, вычисляющую квадрат целого числа. Эта программа состоит из двух единиц компиляции: `main.cpp` и `square.cpp`.

Ниже показан файл `square.cpp`:

```

1 double square(double n)
2 {
3     return n * n;
4 }
```

Этот файл содержит лишь глобальную функцию с именем `square()`, которая возвращает квадрат своего параметра.

Ниже показан файл `main.cpp`:

```

1 #include <cstdlib>
2 #include <iostream>
3 double square(double);
4 int main(int argc, char *argv[])
5 {
6     if (argc != 2) {
7         std::cerr << "Usage: square <number>" << std::endl;
8         return 1;
9     }
10    double n = std::strtod(argv[1], 0);
11    std::cout << "The square of " << argv[1] << " is "
12              << square(n) << std::endl;
13    return 0;
14 }
```

Исходный файл `main.cpp` содержит определение функции `main()`. В C++ эта функция принимает в качестве параметров `int` и `char *` (массив символьных строк). Имя программы находится в `argv[0]`, а аргументы командной строки в `argv[1]`, `argv[2]`, ..., `argv[argc - 1]`. Параметры имеют стандартные имена `argc` («argument count» – количество аргументов) и `argv` («argument values» – значения аргументов). Если программа не использует аргументы командной строки, функцию `main()` можно определить без параметров.

Функция `main()` использует из стандартной библиотеки C++ функцию `strtod()` («string to double» – преобразование строки в переменную двойной точности), `cout` (стандартный поток вывода C++) и `cerr` (стандартный поток вывода сообщений об ошибках C++) для преобразования аргумента командной строки в тип `double` и для вывода текста на консоль. Строки, числа и маркеры конца строки (`endl`) выводятся с помощью оператора `<<`, который также используется для сдвига битов. Чтобы воспользоваться этой стандартной функциональностью, необходимо включить директивы `#include`, расположенные в строках 1 и 2.

Все функции и большинство других элементов стандартной библиотеки C++ задаются в пространстве имен `std`. Один из способов обращения к элементу из пространства имен предусматривает применение префикса с его именем в операторе `::`. В C++ оператор `::` разделяет компоненты сложного имени.

В строке 3 объявляется *прототип функции*. Он указывает компилятору на то, что существует функция с данными параметрами и возвращаемым значением. Реальное определение функции может находиться в той же или в другой единице компиляции. Без прототипа функции компилятор не позволил бы нам вызвать эту функцию в строке 12. Имена параметров функции указывать необязательно.

Процедура компиляции программы зависит от платформы. Например, для компиляции программы в Solaris с использованием компилятора C++ компании «Sun» мы могли бы задать следующие команды:

```
CC -c main.cpp
CC -c square.cpp
CC main.o square.o -o square
```

Первые две строки вызывают компилятор, чтобы сгенерировать файлы `.o` для соответствующих файлов `.cpp`. Третья строка вызывает компоновщик и формирует исполняемый модуль с именем `square`, который может запускаться следующим образом:

```
./square 64
```

В этом случае программа выводит на консоль следующее сообщение:

```
The square of 64 is 4096
(Квадрат числа 64 равен 4096)
```

Чтобы скомпилировать программу, вы, возможно, попросите помощи у местного опытного программиста C++. Если это не удастся сделать, можете прочитать остальную часть приложения, ничего не компилируя, и воспользоваться инструкциями в главе 1 по компиляции вашего первого приложения C++/Qt. В Qt предусмотрены утилиты, позволяющие легко создавать приложения на любой платформе.

Вернемся к нашей программе. В реальном приложении, как правило, мы размещали бы прототип функции `square()` в отдельном файле и включали бы этот файл во все единицы компиляции, в которых вызывается эта функция. Такой файл называется *заголовочным*; он обычно имеет расширение `.h` (часто встречаются также расширения `.hh`, `.hpp` и `.hxx`). Если переделать наш пример, используя заголовочный файл, то можно было бы создать файл с именем `square.h`, который содержит следующие строки:

```
1 #ifndef SQUARE_H
2 #define SQUARE_H

3 double square(double);

4 #endif
```

В начале и в конце заголовочного файла задаются препроцессорные директивы (`#ifndef`, `#define` и `#endif`). Эти директивы гарантируют однократное выполнение заголовочного файла, даже если он несколько раз включается в одну и ту же единицу компиляции (такая ситуация возникает, когда одни заголовочные файлы включают в себя другие заголовочные файлы). По принятым соглашениям используемый для этого препроцессорный символ строится на основе имени файла (в нашем примере это символ `SQUARE_H`). Позже в этом приложении мы вернемся к рассмотрению препроцессора.

Новый файл `main.cpp` будет иметь следующий вид:

```
1 #include <cstdlib>
2 #include <iostream>

3 #include "square.h"

4 int main(int argc, char *argv[])
5 {
6     if (argc != 2) {
7         std::cerr << "Usage: square <number>" << std::endl;
8         return 1;
9     }

10    double n = std::strtod(argv[1], 0);
11    std::cout << "The square of " << argv[1] << " is "
12              << square(n) << std::endl;
13    return 0;
14 }
```

Используемая в строке 3 директива `#include` разворачивает содержимое файла `square.h`. Директивы, начинающиеся с символа `#`, рассматриваются препроцессором C++ до фактической компиляции. В прежние дни препроцессор являлся отдельной программой, которую программист вызывал вручную перед выполнением компилятора. В современных компиляторах этап препроцессорной обработки выполняется автоматически.

Директивы `#include` в строках 1 и 2 разворачивают содержимое заголовочных файлов `cstdlib` и `iostream`, которые являются частью стандартной библиотеки C++. Стандартные заголовочные файлы не имеют суффикса `.h`. Угловые скобки вокруг имен файлов говорят о том, что заголовочные файлы располагаются в стандартном месте системы, а кавычки заставляют компилятор просматривать текущий каталог. Директивы `#include` обычно собирают вместе и располагают в верхней части файла `.cpp`.

В отличие от файлов `.cpp` заголовочные файлы сами по себе не являются единицей компиляции и не приводят к созданию объектных файлов. Они могут только содержать объявления, позволяющие различным единицам компиляции взаимодействовать друг с другом. Следовательно, было бы неправильно помещать реализацию функции `square()` в какой-нибудь заголовочный файл. Если бы мы это сделали в нашем примере, ничего плохого не случилось бы, потому что `square.h` включается только однажды, однако если бы мы включали `square.h` в несколько файлов `.cpp`, то бы получили несколько реализаций функции `square()` (по одной на каждый файл `.cpp`, который включает этот заголовочный файл). После этого компоновщик пожаловался бы на существование нескольких (идентичных) определений функции `square()` и отказался бы генерировать исполняемый модуль. И наоборот, если мы объявляем функцию, но нигде ее не реализуем, компоновщик пожалуется на наличие «неразрешенного символа».

До сих пор мы предполагали, что исполняемый модуль состоит только из объектных файлов. На практике этот модуль компонуется также с библиотеками, которые реализуют готовую функциональность. Существует два основных типа библиотек:

- *статические библиотеки* непосредственно помещаются в исполняемый модуль, как будто они являются объектными файлами. Это гарантирует невозможность потери библиотеки, но увеличивает размер исполняемого модуля;
- *динамические библиотеки* (называемые также совместно используемыми библиотеками или библиотеками DLL) располагаются в стандартном месте на машине пользователя и автоматически загружаются во время запуска приложения.

Программу `square` мы компонуем со стандартной библиотекой C++, которая реализована как динамическая библиотека на большинстве платформ. Сами средства разработки Qt представляют собой коллекцию библиотек, которые могут создаваться как статические или как динамические библиотеки (по умолчанию они создаются как динамические библиотеки).

Основные отличия языков

Теперь мы более внимательно рассмотрим области, в которых C++ отличается от Java и C#. Многие языковые различия объясняются особенностями скомпилированных модулей C++ и повышенным вниманием к производительности. Так, C++ не проверяет границы массивов на этапе выполнения программы и не существует сборщика мусора, восстанавливающего неиспользуемую динамически выделенную память.

Для краткости не будут рассматриваться те конструкции C++, которые почти идентичны соответствующим конструкциям Java и C#. Кроме того, здесь не раскрываются некоторые темы C++, потому что их изучение необязательно при программировании с применением Qt. К ним относятся шаблонные классы и функции, определение объединений и использование исключений. Полное описание языка можно найти в таких книгах, как «The C++ Programming Language» (Addison-Wesley, 2000), написанной Бьерном Страуструпом, или «C++ for Java Programmers» (Prentice Hall, 2003), написанной Марком Аленом Уайссом (Mark Allen Weiss).

Элементарные типы данных

Предлагаемые в C++ элементарные типы данных аналогичны тем, которые используются в Java или C#. На рис. Г.2 приводится список элементарных типов C++ и их определение на платформах, поддерживаемых Qt 4.

По умолчанию short, int, long и long long – типы данных со знаком, т. е. они могут содержать как отрицательные, так и положительные значения. Если необходимо хранить только неотрицательные целые числа, мы можем поставить ключевое слово unsigned (без знака) перед типом. Если тип short может хранить любое значение в промежутке между -32,768 и +32,767, то unsigned short – от 0 до 65 535. Оператор сдвига вправо >> имеет семантику чисел без знака («заполнить нулями»), если один из операндов является типом без знака.

Тип C++	Описание
bool	Булево значение
char	8-битовый целый тип
short	16-битовый целый тип
int	32-битовый целый тип
long	32- или 64-битовый целый тип
long long ¹	64- битовый целый тип
float	32-битовое значение числа с плавающей точкой (IEEE 754)
double	64- битовое значение числа с плавающей точкой (IEEE 754)

Рис. Г.2. Элементарные типы C++

Тип bool может принимать значения true и false. Кроме того, числовые типы могут использоваться вместо типа bool; в этом случае 0 соответствует значению false, а любое ненулевое значение означает true.

Тип char используется для хранения символов ASCII и 8-битовых целых чисел (байтов). Целое число, представленное этим типом, в зависимости от платформы может иметь или не иметь знак. Типы signed char и unsigned char могут использоваться для однозначной интерпретации типа char. Qt предоставляет тип QChar, который хранит 16-битовые символы в кодировке Unicode.

¹ Компания Microsoft вместо нестандартного типа (который достоин быть стандартным) long long использует тип __int64. В программах Qt доступен тип qlonglong в качестве альтернативы, работающий на всех платформах Qt.

По умолчанию экземпляры встроенных типов не инициализируются. Когда создается переменная типа `int`, ее значение вполне могло бы быть нулевым, однако с той же вероятностью оно может равняться `-209486515`. К счастью, большинство компиляторов предупреждает нас о попытках чтения неинициализированной переменной, и мы можем использовать такие инструментальные средства, как `Rational PurifyPlus` и `Valgrind`, для обнаружения обращений к неинициализированной памяти и других, связанных с памятью проблем на этапе выполнения.

В памяти числовые типы (кроме `long`) имеют идентичные размеры на различных платформах, поддерживаемых `Qt`, но их представление меняется в зависимости от принятого в системе порядка байтов. В архитектурах с прямым порядком байтов (например, `PowerPC` и `SPARC`) 32-битовое значение `0x12345678` последовательно занимают четыре байта `0x12 0x34 0x56 0x78`, в то время как в архитектурах с обратным порядком байтов (например, `Intel x86`) последовательность байтов будет обратной. Это следует учитывать в программах, копирующих области памяти на диск или посылающих двоичные данные по сети. Класс `Qt QDataStream`, представленный в главе 12, можно использовать для хранения двоичных данных независимым от платформы способом.

Определения класса

Классы определяются в C++ аналогично тому, как это делается в Java и C#, однако надо иметь в виду, что существует несколько отличий. Мы рассмотрим эти различия на нескольких примерах. Начнем с класса, представляющего пару координат (x, y) :

```
#ifndef POINT2D_H
#define POINT2D_H

class Point2D
{
public:
    Point2D() {
        xVal = 0;
        yVal = 0;
    }
    Point2D(double x, double y) {
        xVal = x;
        yVal = y;
    }

    void setX(double x) { xVal = x; }
    void setY(double y) { yVal = y; }
    double x() const { return xVal; }
    double y() const { return yVal; }

private:
    double xVal;
    double yVal;
};

#endif
```

Представленное выше определение класса обычно оформляется в виде заголовочного файла, типичным названием которого может быть `point2d.h`. В этом примере проявляются характерные особенности C++, перечисленные далее.

- Определение класса разделяется на секции (открытую, защищенную и закрытую) и заканчивается точкой с запятой. Если не указано ни одной секции, по умолчанию используется закрытая секция. (Для совместимости с языком C в C++ предусмотрено ключевое слово `struct`, идентичное классу с тем исключением, что по умолчанию используется открытая секция.)
- Данный класс имеет два конструктора (один без параметров и другой с двумя параметрами). Если в классе вообще не объявляется конструктор, C++ автоматически добавляет конструктор без параметров и с пустым телом.
- Функции, получающие данные, `x()` и `y()`, объявляются как константные. Это значит, что они не будут (и не смогут) модифицировать переменные-члены или вызывать неконстантные функции-члены (например, `setX()` и `setY()`).

Указанные выше функции реализовывались бы как встроенные функции, являющиеся частью определения класса. Альтернативный подход заключается в предоставлении в заголовочном файле только прототипов функций и реализации функций в файле `.cpp`. В этом случае заголовочный файл имел бы следующий вид:

```
#ifndef POINT2D_H
#define POINT2D_H

class Point2D
{
public:
    Point2D();
    Point2D(double x, double y);
    void setX(double x);
    void setY(double y);
    double x() const;
    double y() const;

private:
    double xVal;
    double yVal;
};

#endif
```

Реализация функций выполнялась бы в файле `point2d.cpp`:

```
#include "point2d.h"

Point2D::Point2D()
{
    xVal = 0.0;
```

```
    yVal = 0.0;
}

Point2D::Point2D(double x, double y)
{
    xVal = x;
    yVal = y;
}

void Point2D::setX(double x)
{
    xVal = x;
}

void Point2D::setY(double y)
{
    yVal = y;
}

double Point2D::x() const
{
    return xVal;
}

double Point2D::y() const
{
    return yVal;
}
```

Этот файл начинается с включения заголовочного файла `point2d.h`, потому что прежде чем компилятор будет выполнять синтаксический анализ реализаций функций-членов, он должен иметь определение класса. Затем идут реализации функций, перед именем которых через оператор `::` указывается имя класса.

Мы узнали, как можно реализовать встроенную функцию и как можно реализовать ее в файле `.cpp`. Семантически эти два подхода эквивалентны, однако при вызове встроенной функции большинство компиляторов просто разворачивают тело функции вместо формирования реального вызова функции. Обычно это ведет к получению более быстрого кода, но может увеличить размер приложения. По этой причине только очень короткие функции следует делать встроенными; длинные функции всегда следует реализовывать в файле `.cpp`. Кроме того, если мы забудем реализовать какую-нибудь функцию и попытаемся ее вызвать, компоновщик пожалуется на существование неразрешенного символа.

Теперь попытаемся использовать этот класс.

```
#include "point2d.h"

int main()
{
    Point2D alpha;
    Point2D beta(0.666, 0.875);
```

```

    alpha.setX(beta.y());
    beta.setY(alpha.x());

    return 0;
}

```

В C++ переменные любого типа можно объявлять без непосредственного использования оператора `new`. Первая переменная инициализируется с помощью стандартного конструктора `Point2D` (т. е. конструктора без параметров). Вторая переменная инициализируется с использованием второго конструктора. Обращение к члену объекта осуществляется с использованием оператора `.` (точка).

Объявленные таким образом переменные ведут себя как элементарные типы Java и C# (такие, как `int` и `double`). Например, при использовании оператора присваивания копируется содержимое переменной, а не ссылка на объект. И если позже переменная будет модифицирована, значение всех других переменных, к которым присваивалась первая переменная, не изменится.

C++, как объектно-ориентированный язык, поддерживает наследование и полиморфизм. Для иллюстрации этих свойств мы рассмотрим пример абстрактного класса `Shape` (фигура) и подкласса `Circle` (окружность). Начнем с базового класса:

```

#ifndef SHAPE_H
#define SHAPE_H

#include "point2d.h"

class Shape
{
public:
    Shape(Point2D center) { myCenter = center; }
    virtual void draw() = 0;

protected:
    Point2D myCenter;
};

#endif

```

Определение класса создается в заголовочном файле с именем `shape.h`. Поскольку в этом определении делается ссылка на класс `Point2D`, мы включаем заголовочный файл `point2d.h`.

Класс `Shape` не имеет базового класса. В отличие от Java и C# в C++ не предусмотрен класс `Object`, неявными производными которого являются все другие классы. Qt предоставляет `QObject` в качестве естественного базового класса для объектов всех типов.

Объявление функции `draw()` имеет две интересные особенности. Она содержит ключевое слово `virtual` и завершается равенством `= 0`. Ключевое слово `virtual` означает, что данная функция может быть переопределена в подклассах. Подобно C# функции-члены в C++ по умолчанию не могут переопределяться.

Странное приравнивание `= 0` указывает на то, что данная функция – чисто виртуальная функция, которая не имеет реализации по умолчанию, и она должна быть реализована в подклассах. Концепция «интерфейса» в Java и C# соответствует в C++ классу, содержащему только чисто виртуальные функции.

Ниже приводится определение подкласса `Circle`:

```
#ifndef CIRCLE_H
#define CIRCLE_H

#include "shape.h"

class Circle : public Shape
{
public:
    Circle(Point2D center, double radius = 0.5)
        : Shape(center) {
        myRadius = radius;
    }

    void draw() {
        // здесь выполняются какие-то действия
    }

private:
    double myRadius;
};

#endif
```

Класс `Circle` наследует класс `Shape` в открытой форме, т. е. все открытые члены класса `Shape` остаются открытыми в `Circle`. C++ поддерживает также защищенное и закрытое наследование, которое ограничивает доступ к открытым и защищенным членам базового класса.

Конструктор принимает два параметра. Второй параметр необязателен, по умолчанию он принимает значение `0.5`. Конструктор передает параметр `center` конструктору базового класса, для чего используется специальный синтаксис списка инициализации между сигнатурой функции и телом функции. В теле функции мы инициализируем переменную-член `myRadius`. Инициализацию этой переменной можно было сделать в той же строке, где инициализируется конструктор базового класса:

```
Circle(Point2D center, double radius = 0.5)
    : Shape(center), myRadius(radius) { }
```

С другой стороны, C++ не позволяет инициализировать переменную-член в определении класса, поэтому следующий программный код неверен:

```
// НЕ БУДЕТ КОМПИЛИРОВАТЬСЯ
private:
    double myRadius = 0.5;
};
```


Сигнатура функции `draw()` совпадает с сигнатурой виртуальной функции `draw()`, определенной в классе `Shape`. Она здесь переопределяется и будет вызываться полиморфно, когда `draw()` вызывается экземпляром `Circle` через ссылку или указатель на `Shape`. C++ не имеет ключевого слова `override`, доступного в C#. C++ также не имеет ключевых слов `super` и `base`, ссылающихся на базовый класс. Если требуется вызвать базовую реализацию функции, можно перед именем функции указать имя базового класса и оператор `::`. Например:

```
class LabeledCircle : public Circle
{
public:
    void draw() {
        Circle::draw();
        drawLabel();
    }
    ...
};
```

C++ поддерживает множественное наследование, т. е. возможность создавать класс, производный сразу от нескольких других классов. При этом используется следующий синтаксис:

```
class DerivedClass : public BaseClass1, public BaseClass2, ...,
                    public BaseClassN
{
    ...
};
```

По умолчанию функции и переменные, объявленные в классе, связываются с экземплярами этого класса. Мы можем объявлять статические функции-члены и статические переменные-члены, которые могут использоваться без экземпляра. Например:

```
#ifndef TRUCK_H
#define TRUCK_H

class Truck
{
public:
    Truck() { ++counter; }
    ~Truck() { --counter; }

    static int instanceCount() { return counter; }

private:
    static int counter;
};

#endif
```

Статическая переменная-член счетчика `counter` отслеживает количество экземпляров `truck`, которые существуют в любой момент времени. Конструктор `truck` его увеличивает на единицу. Деструктор, опознаваемый по префиксу в виде тильды (`~`), уменьшает счетчик на единицу. В C++ деструктор автоматически вызывается, когда статически распределенная переменная выходит из области видимости или когда удаляется переменная, память для которой выделяется при помощи оператора `new`. Это аналогично тому, что делается в методе `finalize()` в Java, за исключением того, что мы можем рассчитывать на его вызов в определенный момент времени.

Статическая переменная-член существует в единственном экземпляре для класса – такие переменные являются «переменными класса», а не «переменными экземпляра». Каждая статическая переменная-член должна определяться в файле `.cpp` (но без повторения ключевого слова `static`). Например:

```
#include "truck.h"

int Truck::counter = 0;
```

Если этого не сделать, компоновщик выдаст сообщение об ошибке из-за наличия «неразрешенного символа». Обращаться к статической функции `instanceCount()` можно за пределами класса, указывая имя класса перед ее именем. Например:

```
#include <iostream>

#include "truck.h"

int main()
{
    Truck truck1;
    Truck truck2;

    std::cout << Truck::instanceCount() << " equals 2" << std::endl;

    return 0;
}
```

Указатели

Указатель в C++ – это переменная, содержащая не сам объект, а адрес памяти, где располагается объект. Java и C# имеют аналогичную концепцию «ссылки» при другом синтаксисе. Мы начнем с рассмотрения придуманного нами примера, иллюстрирующего применение указателей:

```
1  #include "point2d.h"

2  int main()
3  {
4      Point2D alpha;
5      Point2D beta;
```

```
6   Point2D *ptr;

7   ptr = &alpha;
8   ptr->setX(1.0);
9   ptr->setY(2.5);

10  ptr = &beta;
11  ptr->setX(4.0);
12  ptr->setY(4.5);

13  ptr = 0;

14  return 0;
15 }
```

В этом примере используется класс `Point2D` из предыдущего подраздела. В строках 4 и 5 определяется два объекта типа `Point2D`. Эти объекты инициализируются в значение (0, 0) стандартным конструктором `Point2D`.

В строке 6 определяется указатель на объект `Point2D`. Для обозначения указателя здесь используется звездочка перед именем переменной. Поскольку мы не инициализируем указатель, он будет содержать произвольный адрес памяти. Эта ситуация изменяется в строке 7, в которой адрес `alpha` присваивается этому указателю. Унарный оператор `&` возвращает адрес памяти, где располагается объект. Адрес обычно представляет собой 32-битовое или 64-битовое целое число, задающее смещение объекта в памяти.

В строках 8 и 9 мы обращаемся к объекту `alpha` с помощью указателя `ptr`. Поскольку `ptr` является указателем, а не объектом, необходимо использовать оператор `->` (стрелка) вместо оператора `.` (точка).

В строке 10 указателю присваивается адрес `beta`. С этого момента любая выполняемая нами операция с этим указателем будет воздействовать на объект `beta`.

В строке 13 указатель устанавливается в нулевое значение. C++ не имеет ключевого слова для представления указателя, который не ссылается ни на один объект; вместо этого мы используем значение 0 (или символическую константу `NULL`, которая разворачивается в 0). Попытка применения нулевого указателя приведет к краху приложения с выводом такого сообщения об ошибке, как «Segmentation fault» (ошибка сегментации), «General protection fault» (общая ошибка защиты) или «Bus error» (ошибка шины). Применяя отладчик, можно найти строку программного кода, которая приводит к краху.

В конце функции объект `alpha` содержит пару координат (1.0, 2.5), а объект `beta` – (4.0, 4.5).

Указатели часто используются для хранения объектов, память для которых выделяется динамически с помощью оператора `new`. Используя жаргон C++, можно сказать, что эти объекты распределяются в «куче», в то время как локальные переменные (т. е. переменные, определенные внутри функции) хранятся в «стеке».

Ниже приводится фрагмент программного кода, иллюстрирующий динамическое распределение памяти при помощи оператора `new`:

```
#include "point2d.h"

int main()
{
    Point2D *point = new Point2D;
    point->setX(1.0);
    point->setY(2.5);
    delete point;

    return 0;
}
```

Оператор `new` возвращает адрес памяти для нового распределенного объекта. Мы сохраняем адрес в переменной указателя и обращаемся к объекту через этот указатель. Поработав с объектом, мы возвращаем занимаемую им память, используя оператор `delete`. В отличие от Java и C# сборщик мусора отсутствует в C++; динамически распределяемые объекты должны явно освобождать занимаемую ими память при помощи оператора `delete`, когда они больше не нужны. В главе 2 описывается механизм родственных связей Qt, который значительно упрощает управление памятью в программах, написанных на C++.

Если не вызвать оператор `delete`, память остается занятой до тех пор, пока не завершится программа. Это не создаст никаких проблем в приведенном выше примере, потому что память выделяется только для одного объекта, однако в программе, в которой постоянно создаются новые объекты, это может привести к нехватке машинной памяти. После удаления объекта переменная указателя по-прежнему будет хранить адрес объекта. Такой указатель является «повисшим указателем» и не должен использоваться для обращения к объекту. Qt предоставляет «умный» указатель, `QPointer<T>`, который автоматически устанавливает себя в 0, если удаляется объект `QObject`, на который он ссылается.

В приведенном выше примере мы вызывали стандартный конструктор и функции `setX()` и `setY()` для инициализации объекта. Вместо этого можно было использовать конструктор с двумя параметрами:

```
Point2D *point = new Point2D(1.0, 2.5);
```

В этом примере не требуется использовать операторы `new` и `delete`. Кроме того, мы могли бы распределить объект в стеке следующим образом:

```
Point2D point;
point.setX(1.0);
point.setY(2.5);
```

Распределенные таким образом объекты автоматически освобождаются в конце блока, в котором они появляются.

Если мы не собираемся модифицировать объект при помощи указателя, можно объявить указатель как константный. Например:

```
const Point2D *ptr = new Point2D(1.0, 2.5);
double x = ptr->x();
double y = ptr->y();
```

```
// НЕ БУДЕТ КОМПИЛИРОВАТЬСЯ
ptr->setX(4.0);
*ptr = Point2D(4.0, 4.5);
```

Константный указатель `ptr` можно использовать лишь для вызова константных функций-членов, например, `x()` и `y()`. Признаком хорошего стиля является объявление указателей константными, когда нет намерения модификации объекта с их помощью. Более того, если сам объект является константным, ничего не остается кроме использования константного указателя для хранения его адреса. Применение ключевого слова `const` предоставляет компилятору информацию, позволяющую обнаружить ошибки на ранних этапах и повысить производительность. C# имеет ключевое слово `const` с очень похожими свойствами. Ближайшим эквивалентом в Java является ключевое слово `final`, однако оно лишь защищает переменные от операций присваивания, но не от вызова «неконстантных» функций-членов объекта.

Указатели могут использоваться с встроенными типами так же, как с классами. Используемый в выражении унарный оператор `*` возвращает значение объекта, на который ссылается указатель. Например:

```
int i = 10;
int j = 20;

int *p = &i;
int *q = &j;

std::cout << *p << " equals 10" << std::endl;
std::cout << *q << " equals 20" << std::endl;

*p = 40;

std::cout << i << " equals 40" << std::endl;

p = q;
*p = 100;

std::cout << i << " equals 40" << std::endl;
std::cout << j << " equals 100" << std::endl;
```

Оператор `->`, который можно использовать для обращения к членам объекта через указатель, является чисто синтаксическим приемом. Вместо `ptr->member` можно также написать `(*ptr).member`. Скобки обязательны, потому что оператор `.` (точка) имеет более высокий приоритет, чем унарный оператор `*`.

Указатели имели плохую репутацию в C и C++, причем доходило до того, что рекламировалось отсутствие указателей в языке Java. На самом деле указатели C++ концептуально аналогичны ссылкам в Java и C# за исключением того, что указатели можно использовать для прохода по памяти, как мы это увидим позже в данном разделе. Более того, включение в Qt классов-контейнеров, использующих метод «копирования при записи» вместе со способностью C++ инстанцировать любой класс в стеке, означает возможность во многих случаях обойтись без указателей.

Ссылки

Кроме указателей C++ поддерживает также концепцию «ссылки». Подобно указателю ссылка в C++ хранит адрес объекта. Ниже указаны основные отличия.

- Объявляются ссылки с применением оператора & вместо *.
- Ссылка должна быть инициализирована и не может в дальнейшем изменяться.
- С помощью ссылки обеспечивается прямое обращение к объекту; не предусмотрен специальный синтаксис, подобный операторам * или ->.
- Ссылка не может быть нулевой.

Ссылки в основном используются при объявлении параметров. Для большинства типов в C++ используется передача параметров по значению, т. е. при передаче параметров функции, последняя получает в действительности новую копию объекта. Ниже приводится определение функции, которая получает параметры, передаваемые по значению.

```
#include <cstdlib>

double manhattanDistance(Point2D a, Point2D b)
{
    return std::abs(b.x() - a.x()) + std::abs(b.y() - a.y());
}
```

Эта функция может вызываться следующим образом:

```
Point2D broadway(12.5, 40.0);
Point2D harlem(77.5, 50.0);
double distance = manhattanDistance(broadway, harlem);
```

C-программисты избегают бесконечных операций копирования путем объявления параметров в виде указателей вместо значений:

```
double manhattanDistance(const Point2D *ap, const Point2D *bp)
{
    return std::abs(bp->x() - ap->x()) + std::abs(bp->y() - ap->y());
}
```

После этого при вызове функции должны передаваться адреса вместо значений:

```
double distance = manhattanDistance(&broadway, &harlem);
```

Ссылки введены в C++ для того, чтобы сделать синтаксис менее громоздким и чтобы предотвратить передачу нулевого указателя. Если вместо указателей использовать ссылки, функция будет иметь следующий вид:

```
double manhattanDistance(const Point2D &a, const Point2D &b)
{
    return std::abs(b.x() - a.x()) + std::abs(b.y() - a.y());
}
```

Ссылка объявляется аналогично указателю с использованием & вместо *. Однако при использовании ссылки можно забыть о том, что она является каким-то адресом памяти и рассматривать ее как обычную переменную. Кроме того, вызов функции, принимающей ссылки в качестве аргументов, не требует специальной записи аргументов (не требуется задавать оператор &).

В конце концов, заменяя в списке параметров Point2D на `const Point2D &`, мы уменьшаем накладные расходы на вызов функции – вместо копирования 256 битов (размер четырех типов `double`) копируются только 64 или 128 бит, что зависит от размера указателя, принятого в целевой платформе.

В предыдущем примере использовались константные ссылки, не позволяющие модифицировать в функции объекты, обращение к которым осуществляется с помощью ссылок. Когда желателен этот побочный эффект, можно передавать неконстантную ссылку или указатель. Например:

```
void transpose(Point2D &point)
{
    double oldX = point.x();
    point.setX(point.y());
    point.setY(oldX);
}
```

В некоторых случаях имеется ссылка и требуется вызвать функцию, которая принимает указатель, и наоборот. Для преобразования ссылки в указатель можно просто использовать унарный оператор `&`:

```
Point2D point;
Point2D &ref = point;
Point2D *ptr = &ref;
```

Для преобразования указателя в ссылку используется унарный оператор `*`:

```
Point2D point;
Point2D *ptr = &point;
Point2D &ref = *ptr;
```

Ссылки и указатели представляются в памяти одинаково и часто могут использоваться вместо друг друга, из-за чего возникает естественный вопрос о том, в каких случаях кого из них следует предпочесть. С одной стороны, ссылки имеют более удобный синтаксис, с другой стороны – указатели в любой момент можно вновь устанавливать на указатель другого объекта, они могут содержать нулевое значение и более явный синтаксис их применения часто является неприятностью, неожиданно оказавшейся благом. По этим причинам предпочтение часто отдается указателям, а ссылки почти исключительно используются при объявлении параметров функций совместно с ключевым словом `const`.

Массивы

Массивы в C++ объявляются с указанием количества элементов массива в квадратных скобках после имени переменной массива. Допускаются двумерные массивы, т. е. массив массивов. Ниже приводится определение одномерного массива, содержащего десять элементов типа `int`:

```
int fibonacci[10];
```

Доступ к элементам осуществляется с помощью следующей записи: `fibonacci[0]`, `fibonacci[1]`, ..., `fibonacci[9]`. Часто требуется инициализировать массив при его определении:

```
int fibonacci[10] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
```

В таких случаях можно не указывать размер массива, поскольку компилятор может его рассчитать по количеству элементов в списке инициализации:

```
int fibonacci[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
```

Статическая инициализация также работает для сложных типов, например, для Point2D:

```
Point2D triangle[] = {  
    Point2D(0.0, 0.0), Point2D(1.0, 0.0), Point2D(0.5, 0.866)  
};
```

Если не предполагается в дальнейшем изменять массив, его можно сделать константным:

```
const int fibonacci[] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
```

Для нахождения количества элементов в массиве можно использовать оператор sizeof():

```
int n = sizeof(fibonacci) / sizeof(fibonacci[0]);
```

Оператор sizeof() возвращает размер аргумента в байтах. Количество элементов массива равно его размеру в байтах, поделенному на размер одного его элемента. Поскольку это долго вводить, распространенной альтернативой является объявление константы и ее использование при определении массива:

```
enum { NFibonacci = 10 };  
const int fibonacci[NFibonacci] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
```

Есть соблазн объявить константу как переменную типа const int. К сожалению, некоторые компиляторы имеют проблемы при использовании константных переменных для представления размера массива. Ключевое слово enum будет объяснено далее в этом приложении.

Проход в цикле по массиву обычно выполняется с использованием переменной целого типа. Например:

```
for (int i = 0; i < NFibonacci; ++i)  
    std::cout << fibonacci[i] << std::endl;
```

Массив можно также проходить с помощью указателя:

```
const int *ptr = &fibonacci[0];  
while (ptr != &fibonacci[10]) {  
    std::cout << *ptr << std::endl;  
    ++ptr;  
}
```

Мы инициализируем указатель адресом первого элемента и проходим его в цикле, пока не достигнем элемента «после последнего элемента» («одиннадцатого» элемента, fibonacci[10]). На каждом шаге цикла оператор ++ продвигает указатель к следующему элементу.

Вместо &fibonacci[0] можно было бы также написать fibonacci. Это объясняется тем, что указанное без элементов имя массива автоматически преобразуется в указатель на первый элемент массива. Аналогично можно было бы подставить fibonacci + 10 вместо &fibonacci[10]. Эти приемы работают и в других местах: мы

можем получить содержимое текущего элемента, используя запись `*ptr` или `ptr[0]` а получить доступ к следующему элементу могли бы, используя `*(ptr + 1)` или `ptr[1]`. Это свойство иногда называют «эквивалентностью указателей и массивов».

Чтобы не допустить того, что считается необоснованной неэффективностью, C++ не позволяет передавать массивы функциям по значению. Вместо этого передается адрес массива. Например:

```
#include <iostream>

void printIntegerTable(const int *table, int size)
{
    for (int i = 0; i < size; ++i)
        std::cout << table[i] << std::endl;
}

int main()
{
    const int fibonacci[10] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
    printIntegerTable(fibonacci, 10);
    return 0;
}
```

Ирония в том, что хотя C++ не позволяет выбирать между передачей массива по ссылке и передачей по значению, он предоставляет некоторую свободу синтаксиса при объявлении типа параметра. Вместо `const int *table` можно было бы также написать `const int table[]` для объявления в качестве параметра указателя на константный тип `int`. Аналогично, параметр `argv` функции `main()` можно объявлять как `char *argv[]` или как `char **argv`.

Для копирования одного массива в другой можно пройти в цикле по элементам массива:

```
const int fibonacci[NFibonacci] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
int temp[NFibonacci];

for (int i = 0; i < NFibonacci; ++i)
    temp[i] = fibonacci[i];
```

Для базовых типов, таких как `int`, можно также использовать функцию `memcpy()`, которая копирует блок памяти. Например:

```
std::memcpy(temp, fibonacci, sizeof(fibonacci));
```

При объявлении массива C++ его размер должен быть константой.¹ Если необходимо создать массив переменного размера, это можно сделать несколькими способами.

- Выделять память под массив можно динамически:

```
int *fibonacci = new int[n];
```

Оператор `[]` выделяет последовательные участки памяти под определенное количество элементов и возвращает указатель на первый элемент. Благодаря

¹ Некоторые компиляторы позволяют использовать такие переменные, однако нельзя на это рассчитывать в переносимых программах.

принципу «эквивалентности указателей и массивов», обращаться к элементам можно с помощью указателя: `fibonacci[0]`, `fibonacci[1]`, ..., `fibonacci[n - 1]`. После завершения работы с массивом необходимо освободить занимаемую им память, используя оператор `delete []`:

```
delete [] fibonacci;
```

- Можно использовать стандартный класс `vector<T>`:

```
#include <vector>
```

```
std::vector<int> fibonacci(n);
```

Обращаться к элементам можно с помощью оператора `[]`, как это делается для обычного массива C++. При использовании вектора `vector<T>` (где `T` – тип элемента, хранимого в векторе) можно изменить его размер в любой момент с помощью функции `resize()`, и его можно копировать, применяя оператор присваивания. Классы, содержащие угловые скобки в имени, называются шаблонными классами.

- Можно использовать класс Qt `QVector<T>`:

```
#include <QVector>
```

```
QVector<int> fibonacci(n);
```

Программный интерфейс вектора `QVector<T>` очень похож на интерфейс вектора `vector<T>`, кроме того, он поддерживает возможность прохода по его элементам с помощью ключевого слова `Qt foreach` и использует неявное совмещение данных («копирование при записи») как метод оптимизации расхода памяти и повышения быстродействия. В главе 11 представлены классы-контейнеры Qt и объясняется их связь со стандартными контейнерами C++.

Может возникнуть соблазн применения везде векторов `vector<T>` или `QVector<T>` вместо встроенных массивов. Тем не менее, полезно иметь представление о работе встроенных массивов, потому что рано или поздно вам может потребоваться очень быстрый программный код или придется использовать существующие библиотеки C.

Символьные строки

Основной способ представления символьных строк в C++ заключается в применении массива символов `char`, завершаемого нулевым байтом («\0»). Следующие четыре функции демонстрируют работу таких строк:

```
void hello1()
{
    const char str[] = {
        'H', 'e', 'l', 'l', 'o', ' ', ' ', 'w', 'o', 'r', 'l', 'd', '\0'
    };
    std::cout << str << std::endl;
}

void hello2()
{
```

```
    const char str[] = "Hello world!";
    std::cout << str << std::endl;
}

void hello3()
{
    std::cout << "Hello world!" << std::endl;
}

void hello4()
{
    const char *str = "Hello world!";
    std::cout << str << std::endl;
}
```

В первой функции строка объявляется как массив и инициализируется по-символьно. Обратите внимание на символ в конце «\0», обозначающий конец строки. Вторая функция имеет аналогичное определение массива, но на этот раз для инициализации массива используется строковый литерал. В C++ строковые литералы – это просто массивы символов `const char`, завершающиеся символом «\0», который не указывается в литерале. В третьей функции строковый литерал используется непосредственно без придания ему имени. После перевода на инструкции машинного языка она будет идентична первым двум функциям.

Четвертая функция немного отличается, поскольку создает не только массив (без имени), но и переменную-указатель с именем `str`, в которой хранится адрес первого элемента массива. Несмотря на это семантика данной функции идентична семантике предыдущих трех функций, и оптимизирующий компилятор удалит лишнюю переменную `str`.

Функции, принимающие в качестве аргументов строки C++, обычно объявляют их как `char *` или `const char *`. Ниже приводится короткая программа, иллюстрирующая оба подхода:

```
#include <cctype>
#include <iostream>

void makeUppercase(char *str)
{
    for (int i = 0; str[i] != '\0'; ++i)
        str[i] = std::toupper(str[i]);
}

void writeLine(const char *str)
{
    std::cout << str << std::endl;
}

int main(int argc, char *argv[])
{
    for (int i = 1; i < argc; ++i) {
        makeUppercase(argv[i]);
    }
}
```

```
        writeLine(argv[i]);  
    }  
    return 0;  
}
```

В C++ тип `char` обычно занимает 8 бит. Это значит, что в массиве символов `char` легко можно хранить строки в кодировке ASCII, ISO 8859-1 (Latin-1) и в других 8-битовых кодировках, но нельзя хранить произвольные символы Unicode, если не прибегать к многобайтовым последовательностям. Qt предоставляет мощный класс `QString`, который хранит строки Unicode в виде последовательностей 16-битовых символов `QChar` и при их реализации использует оптимизацию неявного совмещения данных («копирование при записи»). Более подробно строки `QString` рассматриваются в главе 11 и в главе 18.

Перечисления

C++ позволяет с помощью перечисления объявлять набор поименованных констант, аналогично тому, как это делается в C# и в последних версиях Java. Предположим, что в программе требуется хранить названия дней недели:

```
enum DayOfWeek {  
    Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday  
};
```

Обычно это объявление располагается в заголовочном файле или даже внутри класса. Приведенное выше объявление на первый взгляд представляется эквивалентным следующим определениям констант:

```
const int Sunday    = 0;  
const int Monday    = 1;  
const int Tuesday   = 2;  
const int Wednesday = 3;  
const int Thursday  = 4;  
const int Friday    = 5;  
const int Saturday  = 6;
```

Применяя конструкцию перечисления, мы можем затем объявлять переменные или параметры типа `DayOfWeek`, и компилятор гарантирует возможность присваивания им только значений перечисления `DayOfWeek`. Например:

```
DayOfWeek day = Sunday;
```

Если нас мало волнует обеспечение защищенности типов, мы можем просто написать

```
int day = Sunday;
```

Обратите внимание на то, что при ссылке на константу `Sunday` из перечисления `DayOfWeek`, мы пишем просто `Sunday`, а не `DayOfWeek::Sunday`.

По умолчанию компилятор назначает последовательные целочисленные значения константам перечисления, начиная с нуля. При необходимости можно назначить другие значения:

```
enum DayOfWeek {  
    Sunday    = 628,
```

```
Monday    = 616,  
Tuesday   = 735,  
Wednesday = 932,  
Thursday  = 852,  
Friday    = 607,  
Saturday  = 845  
};
```

Если значение не задается для элемента перечисления, этот элемент примет значение предыдущего элемента, увеличенное на 1. Перечисления иногда используются для объявления целочисленных констант, и в этих случаях перечислению обычно имя не задают:

```
enum {  
    FirstPort = 1024,  
    MaxPorts  = 32767  
};
```

Другой областью применения перечислений является представление набора опций. Рассмотрим пример диалогового окна Find (поиск) с четырьмя переключателями, которые управляют алгоритмом поиска (применение шаблона поиска, учет регистра, поиск в обратном направлении и повторение поиска с начала документа). Это можно представить в виде перечисления, значения констант которого равны некоторой степени 2:

```
enum FindOption {  
    NoOptions      = 0x00000000,  
    WildcardSyntax = 0x00000001,  
    CaseSensitive  = 0x00000002,  
    SearchBackward = 0x00000004,  
    WrapAround     = 0x00000008  
};
```

Каждая опция часто называется «флажком». Флажки можно объединять при помощи логических поразрядных операторов | или |=:

```
int options = NoOptions;  
if (wildcardSyntaxCheckBox->isChecked())  
    options |= WildcardSyntax;  
if (caseSensitiveCheckBox->isChecked())  
    options |= CaseSensitive;  
if (searchBackwardCheckBox->isChecked())  
    options |= SearchBackwardSyntax;  
if (wrapAroundCheckBox->isChecked())  
    options |= WrapAround;
```

Проверить значение флажка можно при помощи логического поразрядного оператора &:

```
if (options & CaseSensitive) {  
    // поиск с учетом регистра  
}
```

Переменная типа `FindOption` может содержать только один флажок в данный момент времени. Результат объединения нескольких флажков при помощи оператора `|` представляет собой обычное целое число. К сожалению, здесь не обеспечивается защищенность типа: компилятор не будет жаловаться, если функция, которая должна принимать в качестве параметра типа `int` некую комбинацию опций `FindOption`, фактически получит `Saturday`. Qt использует класс `QFlags<T>` для обеспечения защищенности своих собственных типов флажков. Этот класс можно также применять при определении пользовательских типов флажков. Подробное описание класса `QFlags<T>` можно найти в онлайн-официальной документации.

Имена, вводимые typedef

C++ позволяет с помощью ключевого слова `typedef` назначать псевдонимы типам данных. Например, если часто используется тип `QVector<Point2D>` и хотелось бы сэкономить немного на вводе символов (или к несчастью приходится иметь дело с норвежской клавиатурой и вам трудно найти на ней угловые скобки), то можно в одном из ваших заголовочных файлов использовать такое объявление `typedef`:

```
typedef QVector<Point2D> PointVector;
```

После этого можно использовать имя `PointVector` как сокращение для `QVector<Point2D>`. Следует отметить, что новое имя указывается после старого. Синтаксис `typedef` специально имитирует синтаксис объявлений переменных.

В Qt имена, вводимые `typedef`, в основном используются по трем причинам, перечисленным далее.

- **Удобство:** Qt объявляет с помощью `typedef` имена `uint` и `QWidgetList` для `unsigned int` и `QList<QWidget *>`, чтобы сэкономить несколько символов.
- **Различие платформ:** определенные типы должны определяться по-разному на различных платформах. Например, `qlonglong` определяется как `__int64` в Windows и как `long long` на других платформах.
- **Совместимость:** класс `QIconSet` из Qt 3 был переименован в `QIcon` для Qt 4. Для облегчения пользователям Qt 3 перевода своих приложений в Qt 4 класс `QIconSet` объявляется как `typedef QIcon`, когда включается режим совместимости с Qt 3.

Преобразование типов

C++ представляет несколько синтаксических конструкций по приведению одного типа к другому. Заключение нужного типа результата в скобки и размещение его перед преобразуемым значением – это традиционный способ, унаследованный от C:

```
const double Pi = 3.14159265359;
int x = (int)(Pi * 100);
std::cout << x << " equals 314" << std::endl;
```

Это очень мощная конструкция. Она может использоваться для изменения типа указателя, устранения константности и для многого другого. Например:

```
short j = 0x1234;
if (*(char *)&j == 0x12)
    std::cout << "The byte order is big-endian" << std::endl;
```

В этом примере мы приводим тип `short *` к типу `char *` и используем унарный оператор `*` для обращения к байту по заданному адресу памяти. В системах с прямым порядком байтов этот байт содержит значение `0x12`; в системах с обратным порядком байтов он имеет значение `0x34`. Поскольку указатели и ссылки представляются одинаково, не удивительно, что представленный выше программный код можно переписать с приведением типа ссылки:

```
short j = 0x1234;
if ((char &j) == 0x12)
    std::cout << "The byte order is big-endian" << std::endl;
```

Если тип данных является именем класса, именем, введенным `typedef`, или элементарным типом, который может быть представлен одной буквенно-цифровой лексемой, для приведения типа можно использовать синтаксис конструктора:

```
int x = int(Pi * 100);
```

Приведение типа указателей и ссылок с использованием традиционного подхода в стиле языка C является неким экстремальным видом спорта, напоминающим параглайдинг и передвижение на кабине лифта, потому что компилятор позволяет приводить указатель (или ссылку) любого типа в любой другой тип указателя (или ссылки). По этой причине в C++ введены новые конструкции приведения типов с более точной семантикой. Для указателей и ссылок новые конструкции приведения типов более предпочтительны по сравнению с рискованными конструкциями в стиле C, и они используются в данной книге.

- `static_cast<T>()` может применяться для приведения типа указателя-на-A к типу указателя-на-B при том ограничении, что класс B должен быть подклассом класса A. Например:

```
A *obj = new B;
B *b = static_cast<B *>(obj);
b->someFunctionDeclaredInB();
```

Если объект не является экземпляром B, применение полученного указателя может привести к неожиданному краху программы.

- `dynamic_cast<T>()` действует аналогично `static_cast<T>()` кроме применения информации о типах, получаемой на этапе выполнения (*runtime type information* – RTTI), для проверки принадлежности к классу B объекта, на который ссылается указатель. Если это не так, то оператор приведения типа возвратит нулевой указатель. Например:

```
A *obj = new B;
B *b = dynamic_cast<B *>(obj);
if (b)
    b->someFunctionDeclaredInB();
```

В некоторых компиляторах оператор `dynamic_cast<T>()` не работает через границы динамических библиотек. Он также рассчитывает на поддержку компилятором технологии RTTI, а эта поддержка может быть отключена программистом для уменьшения размера своих исполняемых модулей. Qt решает эти проблемы, обеспечивая оператор приведения `qobject_cast<T>()` для подклассов `QObject`.

- `const_cast<T>()` добавляет или удаляет спецификатор `const` из указателя или ссылки. Например:

```
int MyClass::someConstFunction() const
{
    if (isDirty()) {
        MyClass *that = const_cast<MyClass *>(this);
        that->recomputeInternalData();
    }
    ...
}
```

В предыдущем примере мы убрали спецификатор `const` при приведении типа указателя `this` для вызова неконстантной функции-члена `recomputeInternalData()`. Не рекомендуется так делать и, если использовать ключевое слово `mutable`, этого можно избежать, как это делается в главе 4.

- `reinterpret_cast<T>()` преобразует любой тип указателя или ссылки в любой другой их тип. Например:

```
short j = 0x1234;
if (reinterpret_cast<char *>(j) == 0x12)
    std::cout << "The byte order is big-endian" << std::endl;
```

В Java и C# любая ссылка может храниться при необходимости как ссылка на `Object`. C++ не имеет никакого универсального базового класса, но предоставляет специальный тип данных, `void *`, который содержит адрес экземпляра любого типа. Указатель `void *` необходимо привести к другому типу (используя `static_cast<T>()`) перед его применением.

C++ обеспечивает много способов приведения типов, однако в большинстве случаев это даже не приходится делать. При использовании таких классов-контейнеров, как `vector<T>` или `QVector<T>`, мы можем задать тип `T` и извлекать элементы без приведения типа. Кроме того, для элементарных типов некоторые преобразования происходят неявно (например, преобразование `char` в `int`), а для пользовательских типов можно определить неявные преобразования, предусматривая конструктор с одним параметром. Например:

```
class MyInteger
{
public:
    MyInteger();
    MyInteger(int i);
    ...
};

int main()
{
    MyInteger n;
    n = 5;
    ...
}
```


Автоматическое преобразование, обеспечиваемое некоторыми конструкторами с одним параметром, имеет мало смысла. Его можно отключить, если объявить конструктор с ключевым словом `explicit`:

```
class MyVector
{
public:
    explicit MyVector(int size);
    ...
};
```

Перегрузка операторов

C++ позволяет нам перегружать функции, т. е. мы можем объявлять несколько функций с одним именем в одной и той же области видимости, если они имеют различные списки параметров. Кроме того, C++ поддерживает перегрузку операторов, позволяя назначать специальную семантику встроенным операторам (таким как `+`, `<<` и `[]`) при их применении для пользовательских типов.

Мы уже видели несколько примеров с перегруженными операторами. Когда использовался оператор `<<` для вывода текста в поток `cout` или `cerr`, мы не пользовались оператором C++, выполняющим поразрядный сдвиг влево, но использовали специальную версию этого оператора, принимающего слева объект потока `ostream` (например, `cout` или `cerr`), а справа – строку (либо вместо строки число или манипулятор потока, например `endl`), и возвращающего объект `ostream`, что позволяет несколько раз вызывать оператор в одной строке.

Красота перегрузки операторов заключается в возможности сделать поведение пользовательских типов в точности таким же, как поведение встроенных типов. Чтобы показать, как работает такая перегрузка, мы перегрузим операторы `+=`, `-=`, `+` и `-`, добавив возможность работы с объектами `Point2D`:

```
#ifndef POINT2D_H
#define POINT2D_H

class Point2D
{
public:
    Point2D();
    Point2D(double x, double y);

    void setX(double x);
    void setY(double y);
    double x() const;
    double y() const;

    Point2D &operator+=(const Point2D &other) {
        xVal += other.xVal;
        yVal += other.yVal;
        return *this;
    }

    Point2D &operator-=(const Point2D &other) {
```

```
        xVal -= other.xVal;
        yVal -= other.yVal;
        return *this;
    }

private:
    double xVal;
    double yVal;
};

inline Point2D operator+(const Point2D &a, const Point2D &b)
{
    return Point2D(a.x() + b.x(), a.y() + b.y());
}

inline Point2D operator-(const Point2D &a, const Point2D &b)
{
    return Point2D(a.x() - b.x(), a.y() - b.y());
}

#endif
```

Операторы можно реализовать либо как функции-члены, либо как глобальные функции. В нашем примере мы реализовали операторы `+=` и `-=` как функции-члены, а операторы `+` и `-` как глобальные функции.

Операторы `+=` и `-=` принимают ссылку на другой объект `Point2D` и увеличивают или уменьшают координаты *x* и *y* текущего объекта на значение координат другого объекта. Они возвращают `*this`, т. е. ссылку на текущий объект (`this` имеет тип `Point2D *`). Возвращение ссылки позволяет создавать экзотический программный код, например:

```
a += b += c;
```

Операторы `+` и `-` принимают два параметра и возвращают значение объекта `Point2D` (а не ссылку на существующий объект). Ключевое слово `inline` позволяет поместить эти функции в заголовочный файл. Если бы тело функции было более длинным, мы бы поместили в заголовочный файл прототип функции, а определение функции (без ключевого слова `inline`) в файл `.cpp`.

Следующие фрагменты программного кода показывают, как можно использовать все четыре перегруженных оператора:

```
Point2D alpha(12.5, 40.0);
Point2D beta(77.5, 50.0);

alpha += beta;
beta -= alpha;

Point2D gamma = alpha + beta;
Point2D delta = beta - alpha;
```

Кроме того, можно вызывать функции `operator` точно так же, как вызываются любые другие функции:

```
Point2D alpha(12.5, 40.0);
Point2D beta(77.5, 50.0);

alpha.operator+=(beta);
beta.operator-=(alpha);

Point2D gamma = operator+(alpha, beta);
Point2D delta = operator-(beta, alpha);
```

Перегрузка операторов в C++ представляет собой сложную тему, однако мы вполне можем пока обходиться без знания всех деталей. Все же важно понимать принципы перегрузки операторов, потому что несколько классов Qt (в том числе QString и QVector<T>) используют их для обеспечения простого и более естественного синтаксиса для таких операций, как конкатенация и добавление в конец объекта.

Типы значений

В Java и C# различаются типы значений и типы ссылок.

- **Типы значений.** Это такие элементарные типы, как char, int и float, а также структуры struct в C#. Характерным для них является то, что для их создания не используется оператор new и оператор присваивания копирует значение переменной. Например:

```
int i = 5;
int j = 10;
i = j;
```

- **Типы ссылок.** Это такие классы, как Integer (в Java), String и MyVeryOwnClass. Их экземпляры создаются при помощи оператора new. Оператор присваивания копирует только ссылку на объект, а для действительного копирования объекта мы должны вызывать функцию clone() (в Java) или Clone() (в C#). Например:

```
Integer i = new Integer(5);
Integer j = new Integer(10);
i = j.clone();
```

В C++ все типы могут использоваться как «типы ссылок», а в дополнение к этому те из них, которые допускают копирование, могут использоваться как «типы значений». Например, в C++ нет необходимости иметь класс подобный Integer, потому что можно использовать указатели и оператор new:

```
int *i = new int(5);
int *j = new int(10);
*i = *j;
```

В отличие от Java и C# в C++ определяемые пользователем типы используются так же, как встроенные типы:

```
Point2D *i = new Point2D(5, 5);
Point2D *j = new Point2D(10, 10);
*i = *j;
```

Если требуется сделать класс C++ копируемым, необходимо предусмотреть в этом классе конструктор копирования и оператор присваивания. Конструктор копирования вызывается при инициализации объекта другим объектом того же типа. Синтаксически в C++ это обеспечивается двумя способами:

```
Point2D i(20, 20);

Point2D j(i);      // первый способ
Point2D k = i;     // второй способ
```

Оператор присваивания вызывается при присваивании одной переменной другой переменной:

```
Point2D i(5, 5);
Point2D j(10, 10);
j = i;
```

При определении класса компилятор C++ автоматически обеспечивает конструктор копирования и оператор присваивания, выполняющие копирования члена в член. Для класса Point2D это равносильно тому, как если бы мы написали следующий программный код в определении класса:

```
class Point2D
{
public:
    ...
    Point2D(const Point2D &other)
        : xVal(other.xVal), yVal(other.yVal) { }
    Point2D &operator=(const Point2D &other) {
        xVal = other.xVal;
        yVal = other.yVal;
        return *this;
    }
    ...
private:
    double xVal;
    double yVal;
};
```

Для некоторых классов создаваемые по умолчанию конструктор копирования и оператор присваивания оказываются неподходящими. Обычно это происходит в тех случаях, когда класс использует динамическую память. Чтобы сделать класс копируемым, мы должны сами реализовать конструктор копирования и оператор присваивания.

Для классов, которые не должны быть копируемыми, можно отключить конструктор копирования и оператор присваивания, если сделать их закрытыми. Если мы случайно попытаемся копировать экземпляры такого класса, компилятор выдаст сообщение об ошибке. Например:

```
class BankAccount
{
public:
```

```
private:
    BankAccount(const BankAccount &other);
    BankAccount &operator=(const BankAccount &other);
};
```

В Qt многие классы проектировались как используемые по значению. Они имеют конструктор копирования и оператор присваивания и обычно инстанцируются в стеке без использования оператора `new`. Это относится к классам `QDateTime`, `QImage`, `QString` и к классам-контейнерам, например, `QList<T>`, `QVector<T>` и `QMap<K, T>`.

Другие классы попадают в категорию «типа ссылок», в частности `QObject` и его подклассы (`QWidget`, `QTimer`, `QTcpSocket` и т. д.). Они имеют виртуальные функции и не могут копироваться. Например, `QWidget` представляет конкретное окно или элемент управления на экране дисплея. Если в памяти находится 75 экземпляров `QWidget`, на экране также будут находиться 75 окон или элементов управления. Обычно эти классы инстанцируются при помощи оператора `new`.

Глобальные переменные и функции

C++ позволяет объявлять функции и переменные, которые не принадлежат ни-какому классу и к которым можно обращаться из любой другой функции. Мы видели несколько примеров глобальных функций, в частности `main()` — точки входа в программу. Глобальные переменные встречаются реже, потому что они плохо влияют на модульность и реентерабельность. Все же важно иметь представление о них, поскольку вам, возможно, придется с ними столкнуться в программном коде, написанном программистом, который раньше писал на C, и другими пользователями C++.

Для иллюстрации работы глобальных функций и переменных рассмотрим небольшую программу, которая печатает список из 128 псевдослучайных чисел, используя придуманный на скорую руку алгоритм. Исходный код программы находится в двух файлах `.cpp`.

Первый исходный файл — `random.cpp`:

```
int randomNumbers[128];

static int seed = 42;

static int nextRandomNumber()
{
    seed = 1009 + (seed * 2011);
    return seed;
}

void populateRandomArray()
{
    for (int i = 0; i < 128; ++i)
        randomNumbers[i] = nextRandomNumber();
}
```

В этом файле объявляются две глобальные переменные (`randomNumbers` и `seed`) и две глобальные функции (`nextrandomNumber()` и `populateRandomArray()`). В двух объявлениях используется ключевое слово `static`; эти объявления видимы только внутри текущей единицы компиляции (`random.cpp`) и говорят, что они *статически связаны* (*static linkage*). Два других объявления доступны из любой единицы компиляции программы, они обеспечивают *внешнюю связь* (*external linkage*).

Статическая компоновка идеально подходит для вспомогательных функций и внутренних переменных, которые не должны использоваться в других единицах компиляции. Она снижает риск «столкновения» идентификаторов (наличия глобальных переменных с одинаковым именем или глобальных функций с одинаковой сигнатурой в разных единицах компиляции) и не позволяет злонамеренным или другим опрометчивым пользователям получать доступ к внутренним объектам единицы компиляции.

Теперь рассмотрим второй файл, `main.cpp`, в котором используется две глобальные переменные, объявленные в `random.cpp` с обеспечением внешней связи:

```
#include <iostream>

extern int randomNumbers[128];

void populateRandomArray();

int main()
{
    populateRandomArray();
    for (int i = 0; i < 128; ++i)
        std::cout << randomNumbers[i] << std::endl;
    return 0;
}
```

Мы объявляем внешние переменные и функции до их вызова. Объявление `randomNumbers` внешней переменной (что делает ее видимой в текущей единице компиляции) начинается с ключевого слова `extern`. Если бы не было этого ключевого слова, компилятор посчитал бы, что он имеет дело с *определением* переменной, и компоновщик пожаловался бы на определение одной и той же переменной в двух единицах компиляции (`random.cpp` и `main.cpp`). Переменные могут объявляться любое количество раз, однако они могут иметь только одно определение. Именно благодаря определению компилятор резервирует пространство для переменной.

Функция `populateRandomArray()` объявляется с использованием прототипа. Указывать ключевое слово `extern` для функций необязательно.

Обычно объявления внешних переменных и функций помещают в заголовочный файл и включают его во все файлы, где они требуются:

```
#ifndef RANDOM_H
#define RANDOM_H
```

```
extern int randomNumbers[128];

void populateRandomArray();

#endif
```

Мы уже видели, как ключевое слово `static` может использоваться для объявления переменных-членов и функций-членов, которые не привязываются к конкретному экземпляру класса, и теперь мы увидели, как можно его использовать для объявления функций и переменных со статической связью. Существует еще одно применение ключевого слова `static`, о котором следует упомянуть. В C++ можно определить локальную переменную как статическую. Такие переменные инициализируются при первом вызове функции и сохраняют свои значения между вызовами функций. Например:

```
void nextPrime()
{
    static int n = 1;

    do {
        ++n;
    } while (!isPrime(n));

    return n;
}
```

Статические локальные переменные подобны глобальным переменным за исключением того, что они видимы только внутри функции, в которой они определены.

Пространства имен

Пространства имен позволяют снизить риск конфликта имен в программах C++. Конфликты имен часто возникают в больших программах, использующих несколько библиотек независимых разработчиков. В своей собственной программе вы решаете сами, использовать ли вам или нет пространства имен.

Обычно в пространство имен заключаются все объявления заголовочного файла, чтобы гарантировать невозможность попадания идентификаторов, объявленных в этом заголовочном файле, в глобальное пространство имен. Например:

```
#ifndef SOFTWAREINC_RANDOM_H
#define SOFTWAREINC_RANDOM_H

namespace SoftwareInc
{
    extern int randomNumbers[128];

    void populateRandomArray();
}

#endif
```

(Обратите внимание на то, что мы переименовали препроцессорные макросимволы, используемые для предотвращения многократного включения содержимого заголовочного файла, снижая риск конфликта имен с заголовочным файлом, имеющим такое же имя, но расположенным в другом каталоге.)

Синтаксис пространства имен совпадает с синтаксисом класса, однако в конце не ставится точка с запятой. Ниже приводится новая версия файла `random.cpp`:

```
#include "random.h"

int SoftwareInc::randomNumbers[128];

static int seed = 42;

static int nextRandomNumber()
{
    seed = 1009 + (seed * 2011);
    return seed;
}

void SoftwareInc::populateRandomArray()
{
    for (int i = 0; i < 128; ++i)
        randomNumbers[i] = nextRandomNumber();
}
```

В отличие от классов пространства имен можно «повторно открывать» в любое время. Например:

```
namespace Alpha
{
    void alpha1();
    void alpha2();
}

namespace Beta
{
    void beta1();
}

namespace Alpha
{
    void alpha3();
}
```

Это позволяет определять сотни классов, размещенных во многих заголовочных файлах и принадлежащих одному пространству имен. Используя этот прием, стандартная библиотека C++ помещает все свои идентификаторы в пространство имен `std`. В Qt пространства имен используются для таких подобных глобальным идентификаторов как `Qt::AlignBottom` и `Qt::yellow`. По историческим причинам классы Qt не принадлежат никакому пространству имен, но имеют префикс «Q».

Для ссылки на идентификатор, объявленный в другом пространстве имен, указывается префикс в виде имени этого пространства имен (и ::). Можно поступить по-другому и использовать один из следующих трех механизмов, нацеленных на уменьшение количества вводимых символов.

- Можно определить псевдоним пространства имен:

```
namespace ElPuebloDeLaReinaDeLosAngeles
{
    void beverlyHills();
    void culverCity();
    void malibu();
    void santaMonica();
}
namespace LA = ElPuebloDeLaReinaDeLosAngeles;
```

После определения псевдонима он может использоваться вместо исходного имени.

- Из пространства имен можно импортировать один идентификатор:

```
int main()
{
    using ElPuebloDeLaReinaDeLosAngeles::beverlyHills;

    beverlyHills();
    ...
}
```

Объявление `using` позволяет обращаться к данному идентификатору без указания префикса, состоящего из имени пространства имен.

- Можно импортировать все пространство имен с помощью одной директивы:

```
int main()
{
    using namespace ElPuebloDeLaReinaDeLosAngeles;

    santaMonica();
    malibu();
    ...
}
```

При таком подходе конфликты имен становятся более вероятными. Если компилятор жалуется на двусмысленное имя (например, когда два класса имеют одинаковое имя, определенное в различных пространствах имен), всегда, при ссылке на идентификатор, его можно уточнить именем пространства имен.

Препроцессор

Препроцессор C++ – это программа, которая обрабатывает исходный файл `.cpp`, содержащий директивы `#` (такие как `#include`, `#ifndef` и `#endif`), и преобразует его файл исходного кода, который не содержит таких директив. Эти директивы предназначены для выполнения простых операций с текстом исходного файла,

например, для выполнения условной компиляции, включения файла и разворачивания макроса. Обычно препроцессор автоматически вызывается компилятором, однако в большинстве систем предусмотрена возможность непосредственного его вызова (часто для этого используется опция компилятора -E и /E).

- Директива `#include` разворачивается в содержимое файла, имя которого указывается в угловых скобках (`<>`) или в двойных кавычках (`«»`), в зависимости от расположения заголовочного файла в стандартном каталоге или в каталоге текущего проекта. Имя файла может содержать `..` и `/` (этот символ правильно интерпретируется компиляторами Windows как разделитель каталогов). Например:

```
#include "../shared/globaldefs.h"
```

- С помощью директивы `#define` определяется макрос. Каждое появление в тексте программы имени, расположенного после директивы `#define`, заменяется определенным для него значением. Например, директива

```
#define PI 3.14159265359
```

указывает препроцессору на необходимость замены каждого появления в текущей единице компиляции лексемы `PI` лексемой `3.14159265359`. Для предотвращения конфликтов имен с переменными и классами общей практикой стало назначение макросам имен, состоящих только из прописных букв. Можно определять макрос с аргументами:

```
#define SQUARE(x) ((x) * (x))
```

Считается хорошим стилем окружение в теле макроса скобками любых параметров, а также всего тела макроса, что позволяет избегать проблем, связанных с приоритетностью операторов. В конце концов, нам нужно, чтобы запись `7 * SQUARE(2 + 3)` разворачивалась в `7 * ((2 + 3) * (2 + 3))`, а не в `7 * 2 + 3 * 2 + 3`.

Компиляторы C++ обычно позволяют определять макросы в командной строке, используя опцию `-D` или `/D`. Например:

```
CC -DPI=3.14159265359 -c main.cpp
```

Макросы были очень популярны в прежние дни, когда еще не были введены `typedef`, перечисления, константы, встраиваемые функции и шаблоны. В наши дни они играют важную роль в предотвращении многократных включений заголовочных файлов.

- Макрос можно отменить в любом месте с помощью директивы `#undef`:

```
#undef PI
```

Эту возможность необходимо использовать, если требуется переопределить макрос, поскольку препроцессор не позволяет определять один и тот же макрос дважды. Эту директиву полезно также применять для управления условной компиляцией.

- Отдельные фрагменты программного кода можно обрабатывать или пропускать при помощи директив `#if`, `#elif`, `#else` и `#endif` в зависимости от конкретных числовых значений макросов. Например:

```

#define NO_OPTIM          0
#define OPTIM_FOR_SPEED  1
#define OPTIM_FOR_MEMORY 2

#define OPTIMIZATION      OPTIM_FOR_MEMORY

...

#if OPTIMIZATION == OPTIM_FOR_SPEED
typedef int MyInt;
#elif OPTIMIZATION == OPTIM_FOR_MEMORY
typedef short MyInt;
#else
typedef long long MyInt;
#endif

```

В приведенном выше примере компилятором будет обрабатываться только второе объявление, которое вводит синоним для `short`. Изменяя определение макроса `OPTIMIZATION`, мы получим другие программы. Если макрос не определен, он будет иметь значение 0.

Другим оператором условной компиляции является проверка макроса на предмет его определения. Это можно сделать следующим образом, используя оператор `defined()`:

```

#define OPTIM_FOR_MEMORY

...

#if defined(OPTIM_FOR_SPEED)
typedef int MyInt;
#elif defined(OPTIM_FOR_MEMORY)
typedef short MyInt;
#else
typedef long long MyInt;
#endif

```

- Ради удобства препроцессор воспринимает `#ifdef X` и `#ifndef X` как синонимы `#if defined(X)` и `#if !defined(X)`. Для предотвращения многократных включений заголовочного файла мы окружаем его содержимое следующими директивами:

```

#ifndef MYHEADERFILE_H
#define MYHEADERFILE_H

...

#endif

```

При первом включении заголовочного файла символ `MYHEADERFILE_H` оказывается не определенным, поэтому компилятор обрабатывает программный код, заключенный между директивами `#ifndef` и `#endif`. При повторном и последующих включениях заголовочного файла символ `MYHEADERFILE_H` оказывается определенным, поэтому весь блок `#ifndef ... #endif` пропускаяется.

- Директива `#error` генерирует на этапе компиляции определенное пользователем сообщение об ошибке. Эта директива часто используется в комбинации с директивами условной компиляции для вывода сообщения о возникновении недопустимого условия. Например:

```
class UniChar
{
public:
    #if BYTE_ORDER == BIG_ENDIAN
        uchar row;
        uchar cell;
    #elif BYTE_ORDER == LITTLE_ENDIAN
        uchar cell;
        uchar row;
    #else
        #error "BYTE_ORDER must be BIG_ENDIAN or LITTLE_ENDIAN"
    #endif
};
```

В отличие от большинства других конструкций C++, в которых недопустимы пробельные символы, препроцессорные директивы должны быть единственными в строке и не должны содержать точку с запятой. Слишком длинные директивы можно разбивать на несколько строк, заканчивая каждую строку, кроме последней, обратной наклонной чертой.

Стандартная библиотека C++

В данном разделе мы кратко рассмотрим стандартную библиотеку C++. На рис. Г.3 приводится список базовых заголовочных файлов C++. Заголовочные файлы `<exception>`, `<limits>`, `<new>` и `<typeinfo>` поддерживают возможности языка C++; например, `<limits>` позволяет проверять возможности поддержки компилятором целочисленной арифметики и арифметики чисел с плавающей точкой, а `<typeinfo>` предлагает основные средства анализа информации о типах. Другие заголовочные файлы предоставляют часто используемые классы, в том числе класс строки и тип комплексных чисел. Функциональность, предлагаемая заголовочными файлами `<bitset>`, `<locale>`, `<string>` и `<typeinfo>`, свободно перекрывается в Qt классами `QByteArray`, `QLocale`, `QString` и `QObject`.

Стандартный C++ также включает ряд заголовочных файлов, обеспечивающих ввод-вывод (см. рис. Г.4). Классы стандартного ввода-вывода проектировались в 80-х годах и обладают излишней сложностью, что сильно затрудняет их понимание, причем настолько, что этой теме были посвящены целые книги. Кроме того, программист остается наедине с ящиком Пандоры неразрешенных проблем, связанных с кодировкой символов и зависимого от платформы двоичного представления элементарных типов данных.

В главе 12 представлены соответствующие классы Qt, обеспечивающие ввод-вывод символов в кодировке Unicode, а также большой набор национальных кодировок и абстракцию независимого от платформы хранения двоичных данных. Qt-классы ввода-вывода формируют основу поддержки межпроцессной связи, работы с сетями и XML. Qt-классы двоичных и текстовых потоков можно очень легко расширить для работы с пользовательскими типами данных.

Заголовочный файл	Описание
<bitset>	Шаблонный класс для представления последовательностей битов фиксированной длины
<complex>	Шаблонный класс для представления комплексных чисел
<exception>	Типы и функции, относящиеся к обработке исключений
<limits>	Шаблонный класс, определяющий свойства числовых типов
<locale>	Классы и функции, относящиеся к локализации
<new>	Функции, управляющие динамическим распределением памяти
<stdexcept>	Заранее определенные типы исключений для вывода сообщений об ошибках
<string>	Шаблонный строковый контейнер и свойства символов
<typeinfo>	Класс, предоставляющий основную метаинформацию о типах
<valarray>	Шаблонные классы для представления массивов значений

Рис. Г.3. Базовые заголовочные файлы библиотеки C++

Заголовочный файл	Описание
<fstream>	Шаблонные классы, манипулирующие внешними файлами
<iomanip>	Манипуляторы потоков ввода-вывода, принимающие один аргумент
<iostream>	Шаблонный базовый класс потоков ввода-вывода
<iosfwd>	Предварительные объявления нескольких шаблонных классов потоков ввода-вывода
<iostream>	Стандартные потоки ввода-вывода (cin, cout, cerr, clog)
<istream>	Шаблонный класс, управляющий вводом из буфера потока
<ostream>	Шаблонный класс, управляющий выводом в буфер потока
<sstream>	Шаблонные классы, связывающие буферы потоков со строками
<streambuf>	Шаблонные классы, обеспечивающие буфер для операций ввода-вывода
<strstream>	Классы для выполнения операций потокового ввода-вывода с массивами символов

Рис. Г.4. Заголовочные файлы библиотеки ввода-вывода C++

В начале 90-х годов была введена стандартная библиотека шаблонов (Standard Template Library – STL), представляющая собой набор шаблонных классов-контейнеров, итераторов и алгоритмов, которые вошли в стандарт ISO C++ в последний момент. На рис. Г.5 приводится список заголовочных файлов библиотеки STL. Проект STL выполнен очень аккуратно, почти с математической точностью, и обеспечивает обобщенную типобезопасную функциональность. Qt предоставляет свои собственные классы-контейнеры, разработка которых отчасти инспирирована STL. Они описываются в главе 11.

Поскольку C++ фактически является супермножеством относительно языка программирования C, программисты C++ имеют в своем распоряжении также полную библиотеку C. Заголовочные файлы библиотеки C доступны как с их традиционными именами (например, <stdio.h>), так и с новыми именами с c-префиксом и без расширения .h (например, <cstdio>). Когда используется новая версия имен заголовочных файлов, функции и типы данных объявляются в пространстве имен std. (Это не относится к таким макросам, как ASSERT(), потому что препроцессор никак не реагирует на пространства имен). Рекомендуется использовать новый стиль обозначения имен, если его поддерживает ваш компилятор.

Заголовочный файл	Описание
<algorithm>	Шаблонные функции общего назначения
<deque>	Шаблонный контейнер очереди с двумя концами
<functional>	Шаблоны, помогающие конструировать и манипулировать функторами
<iterator>	Шаблоны, помогающие конструировать и манипулировать итераторами
<list>	Шаблонный контейнер двусвязного списка
<map>	Шаблонные контейнеры ассоциативных массивов, связывающие ключ с одним и с несколькими значениями
<memory>	Утилиты, позволяющие упростить управление памятью
<numeric>	Шаблонные операции с числами
<queue>	Шаблонный контейнер очереди
<set>	Шаблонные контейнеры наборов, допускающие и не допускающие повторение элементов
<stack>	Шаблонный контейнер стека
<utility>	Основные шаблонные функции
<vector>	Шаблонный контейнер вектора

Рис. Г.5. Заголовочные файлы STL

На рис. Г.6 приводится список заголовочных файлов библиотеки C. Большинство из них предлагает функциональность, которая перекрывается более новыми заголовочными файлами C++ или Qt. Стоит отметить одно из исключений – <cmath>, в котором объявляются такие математические функции, как sin(), sqrt() и pow().

Заголовочный файл	Описание
<cassert>	Макрос ASSERT()
<cctype>	Функции классификации и отображения символов
<cerrno>	Макросы, относящиеся к сообщениям об ошибочных ситуациях
<cfloat>	Макросы, определяющие свойства элементарных типов чисел с плавающей точкой
<ciso646>	Альтернативное представление для пользователей набора символов ISO 646
<climits>	Макросы, определяющие свойства элементарных целочисленных типов
<locale>	Функции и типы, относящиеся к локализации
<cmath>	Математические функции и константы
<csetjmp>	Функции для выполнения нелокальных переходов
<csignal>	Функции для обработки системных сигналов
<cstdlib>	Макросы для реализации функций с переменным числом аргументов
<stddef>	Определения, общие для некоторых стандартных заголовочных файлов
<stdio>	Функции ввода-вывода
<stdlib>	Общие вспомогательные функции
<string>	Функции для манипулирования массивами char
<time>	Типы и функции для манипулирования временем
<wchar>	Утилиты для работы с многобайтовыми символами и символами расширенной кодировки
<wctype>	Функции классификации и отображения символов расширенной кодировки

Рис. Г.6. Заголовочные файлы C++ для обеспечения возможностей библиотеки C

Этим завершается наш краткий обзор стандартной библиотеки C++. В сети Интернет можно получить предлагаемое компанией «Dinkumware» полное справочное руководство по стандартной библиотеке C++, размещенное на веб-странице <http://www.dinkumware.com/refxcpp.html>, и предлагаемое компанией «SGI» подробное руководство программиста по STL, размещенное на веб-странице <http://www.sgi.com/tech/stl/>. Официальное описание стандартной библиотеки C++ можно найти в стандартах C и C++ в виде файлов PDF или получить в бумажном виде в Международной организации по стандартизации (International Organization for Standardization – ISO).

В данном приложении мы бегло рассмотрели многие темы. Когда вы станете изучать средства разработки Qt, начиная с главы 1, вы обнаружите, что используемый ими синтаксис значительно проще и аккуратнее, чем можно было бы предположить после прочтения данного приложения. Хорошее Qt-программирование требует применения только подмножества языка C++ и обычно не требует использования более сложного и не очень понятного синтаксиса, возможного в C++. После того, как вы станете вводить программный код, собирать исполняемые модули и запускать их, четкость и простота принятого в Qt подхода станет очевидной. И когда вы начнете писать более амбициозные программы, особенно те, в которых требуется обеспечить быструю и сложную графику, возможности комбинации C++ и Qt всегда будут идти в ногу с вашими потребностями.

Предметный указатель

#

% (знак процента), 61, 291–292
& (амперсant), 15, 224, 412
&, унарный оператор, 645
, оператор 300–302, 305, 634, 659
. (точка), оператор, 641, 645, 647
/ (наклонная черта), 70, 312
:/ (двоеточие с наклонной чертой), 49, 313, 420
::, оператор, 634, 640, 643, 667
\ (обратная наклонная черта), 312, 670
<?xml?>, объявления, 407, 410, 411
<<, оператор, 84, 277, 292, 297, 300
<> (угловые скобки), 279, 652, 656
>, оператор, 645, 647
>>, оператор, 84, 297, 300–302, 306, 637
€ (знак евро), 426, 434

A

ABItem, 566
about()
 MainWindow, 68
 QMessageBox, 69
aboutQt() (QApplication), 51
absoluteFilePath() (QDir), 416
accept()
 QDialog, 29, 65
 QEvent, 60, 165, 234, 234, 347
Acceptable (QValidator), 106
Accepted (QDialog), 29, 65
accepted() (QDialogButtonBox), 30
acceptProposedAction() (QDropEvent), 230
AcceptRole (QDialogButtonBox), 30
acquire() (QSemaphore), 349–350
action-элементы, 50–51, 163–164
activateNextSubWindow() (QMdiArea), 162
activateWindow() (QWidget), 64–65, 90
activeEditor() (MainWindow), 163
ActiveQt, 422, 555–567
activeSubWindow() (QMdiArea), 163
ActiveX, 555–567
 -activex, опция (серверы ActiveX), 567
add() (ExpenseWindow), 590
addAction() (QWidget), 52–54
addBindValue() (QSqlQuery), 322
addChildSettings() (SettingsViewer), 248
addDatabase() (QSqlDatabase), 320, 323
addDepartment() (MainForm), 340
addEditor() (MainWindow), 162
addEmployee() (EmployeeForm), 334
addItem() (QComboBox), 36, 254
addLayout() (QBoxLayout), 16–17
addLibraryPath() (QCoreApplication), 501
addLink()
 DiagramWindow, 211
 Node, 205
addMenu()
 QMenu, 52
 QMenuBar, 52
addNode() (DiagramWindow), 210
AddRef() (IUnknown), 563
AddressBook, класс 565–566
addRow() (CoordinateSetter), 246
addSeparator()
 QMenu, 52
 QMenuBar, 52
 QToolBar, 54
addStretch() (QLayout), 148
addTransaction()
 ImageWindow, 356
 TransactionThread, 357
addWidget()
 QBoxLayout, 16
 QGridLayout, 147
 QSplitter, 151
 QStackedLayout, 151
 QStatusBar, 55
adjust() (PlotSettings), 138
adjustAxis() (PlotSettings), 138
adjusted() (QRect), 128
adjustSize() (QWidget), 125
Aero. См. Vista, стиль
<algorithm>, заголовочный файл, 289, 672
AlignHCenter (Qt), 56
AlignXxx (QTextStream), 307
Alt, клавиша, 15, 171
Annotation
 определение класса, 218
Annotation(), 218
 boundingRect(), 219
 paint(), 220
 setText(), 219
Any (QHostAddress), 387
AnyKeyPressed (QAbstractItemView), 245
app template (файлы .pro), 602
append()
 QLinkedList<T>, 278
 QString, 291
 QTextEdit, 318
 QVector<T>, 277
Apple Help, 422
Apple Roman, 429
Application, 570–572
applicationDirPath() (QCoreApplication), 417
apply() (Transaction), 360
applyEffect() (BasicEffectsPlugin), 514
Aqua. См. Mac, стиль
arg() (QString), 61, 273, 292, 431
ARGB, формат, 110, 197, 506
См. также RGBA, формат
args и argv, параметры, 3, 571, 634
argument() (QScriptContext), 549

arguments (ECMAScript), 521–522
 arguments() (QCoreApplication), 161–162, 364, 545
 ARM, 575
 Array (ECMAScript) определение класса, 525
 ASCII, 291, 306, 310, 322, 426–429, 433
 assert, операторы, 400
 Assigned Numbers Authority, 230
 Assistant. См. Qt Assistant
 at() (классы-контейнеры), 284
 atEnd() (QXmlStreamReader), 397
 ATSTFontFormatRef (Mac OS X), 554
 autoRecalculate() (Spreadsheet), 78
 AutoSubmit (QDataWidgetMapper), 333
 AWT, 612, 617
 AxBouncer
 AxBouncer(), 562
 createAggregate(), 563
 setColor(), 562
 setRadius(), 563
 setSpeed(), 563
 start(), 563
 stop(), 563
 определение класса, 560

B

BackgroundColorRole (Qt), 255, 270
 Backtab, клавиша, 171
 BankAccount, 663
 base, ключевое слово (C#), 643
 BasicEffectsPlugin, 513–514
 BDF, 579
 BDiagPattern (Qt), 185
 beep() (QApplication), 90
 beforeDelete() (QSqlTableModel), 340
 beforeInsert() (QSqlTableModel), 340
 beforeUpdate() (QSqlTableModel), 340
 begin() (классы-контейнеры), 259, 281–282, 284, 289
 beginGroup() (QSettings), 70
 BevelJoin (Qt), 184
 Big5, кодировка 429
 bind()
 QGLFramebufferObject, 494
 QUdpSocket, 390–391
 bindValue() (QSqlQuery), 322
 bjam, 609
 BLOB, 323
 BMP, 48
 Boolean (ECMAScript), 519, 525
 booleanExpressionChanged() (BooleanModel), 269
 booleanExpressionChanged(), 269
 columnCount(), 267
 data(), 268
 headerData(), 269
 index(), 266
 nodeFromIndex(), 267
 parent(), 267

 rowCount(), 267
 setRootNode(), 266
 BooleanModel
 определение класса, 265
 BooleanModel(), 266
 ~BooleanModel(), 266
 BooleanParser, 264, 269
 Boost.Build, 607, 609–610
 border images (CSS), 455–456
 border rectangle (CSS), 453, 454
 boundingRect()
 Annotation, 219
 CityBlock, 216
 Node, 207
 QFontMetrics, 219
 QPainter, 225
 bringToFront() (DiagramWindow), 210
 BronzeStyle
 drawBronzeBevel(), 473
 drawBronzeCheckBoxIndicator(), 475
 drawBronzeFrame(), 472
 drawBronzeSpinBoxButton(), 474
 drawComplexControl(), 469
 drawPrimitive(), 468
 pixelMetric(), 467
 polish(), 465, 466
 standardIconImplementation(), 468
 styleHint(), 466
 subControlRect(), 470
 unpolish(), 466
 определение класса, 464
 подключаемый модуль, 499–502
 BronzeStylePlugin
 create(), 500
 keys(), 500
 определение класса, 499–500
 BSTR (Windows), 557
 button() (QMouseEvent), 115, 232, 481, 483

C

C#, 631–673
 C++, 631–673
 C++, библиотека, 276, 634, 636, 670–672
 C++, стиль приведения типов, 63, 656–658
 Calculator, 528–529
 call() (ECMAScript), 526
 Cancel (QDialogButtonBox), 30
 cancel() (QSessionManager), 572
 canConvert<T>() (QVariant), 297
 canRead() (CursorHandler), 504
 canRead(), 504
 canReadLine() (QIODevice), 387
 capabilities() (CursorPlugin), 502
 Carbon, 551, 552
 cascadeSubWindows() (QMdiArea), 164
 Cascading Style Sheets, 446–461
 <cctype>, заголовочный файл, 427, 672
 cd() (QFtp), 366, 370

- Cell
 - Cell(), 96
 - clone(), 97
 - data(), 97
 - evalExpression(), 100
 - evalFactor(), 101
 - evalTerm(), 101
 - formula(), 97
 - setData(), 97
 - setDirty(), 97
 - setFormula(), 97
 - value(), 98
 - дерево наследования, 78
 - определение класса, 95
- cell() (Spreadsheet), 81
- cerr (std), 299, 315, 318, 633
- CGImageRef (Mac OS X), 554
- CGI-скрипты, 375
- changeEvent() (QWidget), 441
- char, 427, 429, 433, 637, 654
- char, типы, 652
- Characters (QXmlStreamReader), 393
- characters() (SaxHandler), 408
- cin (std), 299, 310
- Circle (ECMAScript), 526
- Circle, 642, 642
- CityBlock
 - boundingRect(), 216
 - CityBlock(), 217
 - CityModel
 - CityModel(), 258
 - columnCount(), 258
 - data(), 258
 - flags(), 263
 - headerData(), 258
 - offsetOf(), 263
 - paint(), 217
 - rowCount(), 258
 - setCities(), 263
 - setData(), 262
 - usage, 259
 - определение класса, 216
 - определение класса, 260
- Cityscape
 - Cityscape(), 216
 - определение класса, 216
- CityView
 - CityView(), 221
 - wheelEvent(), 221
 - определение класса, 220
- CLASSPATH, переменная среды, 616, 618, 625–626, 629
- Cleanlooks, стиль, 9, 129, 462
- Clear (режим композиции), 199
- clear()
 - QTableWidget, 80
 - QTranslator, 440
 - Spreadsheet, 80
 - классы-контейнеры, 265, 290
- clearBoard() (TicTacToe), 573
- clearCurve() (Plotter), 127
- clicked() (QAbstractButton), 6, 170, 314
- ClientSocket
 - ClientSocket(), 385
 - generateRandomTrip(), 386
 - readClient(), 385
 - определение класса, 384
- clipboard() (QApplication), 87, 214, 240
- clone() (Cell), 97
- close()
 - QFtp, 365, 366
 - QHttp, 374
 - QIODevice, 318, 383, 386, 593
 - QWidget, 16, 51, 60
- closeActiveSubWindow() (QMdiArea), 164
- closeAllSubWindows() (QMdiArea), 164
- closeAllWindows() (QApplication), 72
- closeConnection() (TripPlanner), 382
- closeEditor() (QAbstractItemDelegate), 273
- closeEvent()
 - Editor, 165
 - ExpenseWindow, 589
 - MainWindow, 60, 165
 - ThreadDialog, 347
- CMake, 608–609
- <cmath> header, 672
- codecForLocale() (QTextCodec), 427
- codecForName() (QTextCodec), 428–429
- CODECFORTR, элемент (файлы .pro), 443
- CodeEditor, 171
- ColorBurn (режим композиции), 199
- colorData() (QMimeData), 239
- ColorDodge (режим композиции), 199
- ColorGroup (QPalette), 114
- colorNames() (QColor), 253
- ColorNamesDialog, 254
- column() (QModelIndex), 255, 258
- ColumnCount (Spreadsheet), 79–80
- columnCount()
 - BooleanModel, 267
 - CityModel, 258
 - CurrencyModel, 257
- COM, 555–567
- commandFinished() (QFtp), 367
- commandStarted() (QFtp), 367
- Comment (QXmlStreamReader), 393
- commit() (QSqlDatabase), 323, 341
- commitAndCloseEditor() (TrackDelegate), 273
- commitData()
 - QAbstractItemDelegate, 273
 - QApplication, 569, 571
- CONFIG, элемент (файлы .pro), 38, 421, 501, 559, 564, 568, 603–604
- configure, 319, 576–578, 580
- connect()
 - QObject, 6, 8, 16, 20, 354, 361
 - в Qt Jambi, 614

- connected() (QAbstractSocket), 379, 380
- connectionClosedByServer() (TripPlanner), 383
- connectToHost()
 - QAbstractSocket, 379, 389
 - QFtp, 365–366, 369
- connectToServer() (TripPlanner), 379
- const, ключевое слово, 639, 646–647, 650, 657–658
- const_cast<T>(), 658
- constBegin() (классы-контейнеры), 284
- constData() (QByteArray), 294
- constEnd() (классы-контейнеры), 284
- contains()
 - QMap<K,T>, 287
 - QRect, 116, 130
 - QString, 90
- contentsChanged() (QTextDocument), 167, 169
- contextMenuEvent() (QWidget), 54
- ContiguousSelection (QAbstractItemView), 80, 87, 237
- controllingUnknown() (QAxAggregated), 564
- convert (ImageMagick), 313
- ConvertDepthTransaction, 359
- ConvertDialog(), 314
 - convertImage(), 316
 - on_browseButton_clicked(), 314, 316
 - processError(), 318
 - processFinished(), 318
 - updateOutputTextEdit(), 318
 - определение класса, 313
- convertImage() (ConvertDialog), 316
- convertToFormat() (QImage), 111
- CoordinateSetter, 245–247
- copy()
 - DiagramWindow, 214
 - QTextEdit, 240
 - Spreadsheet, 87
- CopyAction (Qt), 234
- copyAvailable() (QTextEdit), 162
- copyFile(), 304
- Core Foundation, 70
- count() (классы-контейнеры), 277
- cout (std), 299, 310, 633
- .cpp, файлы, 632–633
- create()
 - BronzeStylePlugin, 500
 - CursorPlugin, 502
- createAction() (QWhatsThis), 415
- createActions()
 - ExpenseDialog, 591
 - ExpenseWindow, 588
 - MainWindow, 50, 72, 437
- createAggregate() (AxBouncer), 563
- createConnection(), 320
- createContextMenu() (MainWindow), 53
- createCustomButtons() (Calculator), 528
- createDepartmentPanel() (MainForm), 337
- createDirectory() (DirectoryViewer), 252
- createEditor() (TrackDelegate), 273
- createEmployeePanel() (MainForm), 338
- createGLObject() (VowelCube), 486
- createGradient() (VowelCube), 486
- createIndex() (QAbstractItemModel), 262
- createLanguageMenu() (MainWindow), 439
- createMenuOrToolBar()
 - ExpenseDialog, 592
 - ExpenseWindow, 589
- createMenus() (MainWindow), 52, 164, 438
- createScriptAction() (HtmlWindow), 533–535
- createScriptsMenu() (HtmlWindow), 532
- createStatusBar() (MainWindow), 55
- createToolBars() (MainWindow), 54
- createWidget() (IconEditorPlugin), 120
- critical() (QMessageBox), 57
- CRLF. *См.* требование к формату окончания строк
- CrossPattern (Qt), 185
- CSS, 39, 446–461
- <cstdlib>, заголовочный файл, 636, 672
- CSV, 236
- .cur-файлы, 502, 505–509
- currencyAt() (CurrencyModel), 259
- CurrencyModel(), 257
 - columnCount(), 258
 - currencyAt(), 259
 - data(), 258
 - headerData(), 259
 - rowCount(), 258
 - setCurrencyMap(), 259
 - определение класса, 257
 - применение, 256–258
- currentDateTime() (QDateTime), 192
- currentFormula() (Spreadsheet), 83
- currentImageNumber() (CursorHandler), 504
- currentImageNumber(), 504
 - enterErrorState(), 508
 - imageCount(), 505
 - jumpToNextImage(), 508
 - read(), 505–508
 - readBitmap(), 509
 - readHeaderIfNecessary(), 508
- currentIndex() (QComboBox), 67
- currentLocation() (Spreadsheet), 83
- currentPath() (QDir), 312
- currentRowChanged()
 - QItemSelectionModel, 338
 - QListWidget, 38, 151
- currentThread() (QThread), 353
- Cursor (X11), 554
- CursorHandler
 - определение класса, 503
- CursorHandler(), 504
- CursorPlugin
 - capabilities(), 502
 - create(), 502
 - keys(), 502
 - определение класса, 502

customButtonClicked() (Calculator), 529
cut()

DiagramWindow, 214
MainWindow, 164
QTextEdit, 240
Spreadsheet, 87

CY (Windows), 557

C-стиль приведения типов, 656

D

Darken (режим композиции), 199

DashDotDotLine (Qt), 184

DashDotLine (Qt), 184

DashLine (Qt), 184

data()

BooleanModel, 268
Cell, 97
CityModel, 258
CurrencyModel, 258
QAbstractItemModel, 258, 268, 270
QAction, 536
QByteArray, 294
QMimeData, 234, 239
QTableWidgetItem, 95, 98, 245

database() (QSqlDatabase), 323

dataChanged()

QAbstractItemModel, 262
QClipboard, 240

Date (ECMAScript), 525

DATE (Windows), 557

DB2 (IBM), 319

.dcf, файлы, 422

decodeURI() (ECMAScript), 525

decodeURIComponent() (ECMAScript), 525

.def, файлы, 565

#define, директивы, 12, 635, 668

defined(), оператор, 668–669

DEFINES, элемент (файлы .pro), 433, 603

del()

DiagramWindow, 212
Spreadsheet, 89
TeamLeadersDialog, 251

delete [], оператор, 652

DELETE, инструкции (SQL), 324

delete, оператор, 4, 28, 66, 71–73, 89, 214, 265, 290, 646

deleteDepartment() (MainForm), 340

deleteEmployee() (EmployeeForm), 335

deleteLater() (QObject), 361, 385

delta() (QWheelEvent), 134

Dense2Pattern (Qt), 185

depends(), директивы (файлы .pro), 587

dequeue() (QQueue<T>), 278

Designer. См. Qt Designer

.desktop, файлы, 582–584, 587

desktop-службы, 416

DESTDIR, элемент (файлы .pro), 501, 603

Destination (режим композиции), 199

DestinationAtop (режим композиции), 199

DestinationIn (режим композиции), 199

DestinationOut (режим композиции), 199

DestinationOver (режим композиции), 199

DiagCrossPattern (Qt), 185

DiagramWindow

addLink(), 212
addNode(), 210
bringToFront(), 210
copy(), 214
cut(), 213
del(), 212
DiagramWindow(), 210
NodePair, 209
paste(), 214
properties(), 213
selectedLink(), 212
selectedNode(), 211
selectedNodePair(), 212
sendToBack(), 211
setupNode(), 210
setZValue(), 211
updateActions(), 215
определение класса, 209

Difference (режим композиции), 199

directoryOf() (MainWindow), 416

DirectoryViewer, 252–253

disconnect() (QObject), 20, 21, 361

disconnected() (QAbstractSocket), 379, 383, 385

-display, опция (приложения Qt/Embedded Linux), 578, 579

DisplayRole (Qt), 96, 97–98, 244, 255, 258, 269

DLL, библиотеки, 498, 603, 636

DLLDESTDIR, элемент, (файлы .pro), 603, 627

DNS. См. QHostInfo

documentElement() (QDomDocument), 403

documentTitle() (QTextBrowser), 420

DOM, 392–393, 400–405

DomParser

DomParser(), 402
parseBookindexElement(), 403
parseEntryElement(), 403
parsePageElement(), 404
readFile(), 402
определение класса, 401

done()

FlowChartSymbolPicker, 244
FtpGet, 364, 366
QDialog, 245, 247
QFtp, 365, 366, 367
QHttp, 373
Spider, 369

DontConfirmOverwrite (QFileDialog), 60

DotLine (Qt), 184

«drag-and-drop»

иницирование перетаскивания
объектов, 231–233

прием переносимых объектов, 229–231, 233–235

расстояние регистрации начала
 перетаскивания, 233
dragEnterEvent()
 ProjectListWidget, 233
 QWidget, 230, 233
dragLeaveEvent() (QWidget), 231
dragMoveEvent()
 ProjectListWidget, 233
 QWidget, 231, 233
draw()
 Circle, 642
 LabeledCircle, 643
 OvenTimer, 194
 Shape, 641, 643
 Tetrahedron, 482
drawArc() (QPainter), 183
drawBackground() (VowelCube), 487
drawBronzeBevel() (BronzeStyle), 473
drawBronzeCheckBoxIndicator()
 (BronzeStyle), 475
drawBronzeFrame() (BronzeStyle), 472
drawBronzeSpinBoxButton() (BronzeStyle), 474
drawChord() (QPainter), 183
drawComplexControl() (BronzeStyle), 469
drawControl() (QStyle), 462
drawCube() (VowelCube), 487–489
drawCubicBezier() (QPainter), 183
drawCurves() (Plotter), 136
drawDisplay() (QItemDelegate), 273
drawEllipse() (QPainter), 183, 185, 194–196
drawFocus() (QItemDelegate), 273
drawGrid() (Plotter), 135
drawLegend() (VowelCube), 489–490
drawLine() (QPainter), 113, 183, 196
drawLines() (QPainter), 183
drawPath() (QPainter), 183, 515
drawPie() (QPainter), 183, 185
drawPixmap() (QPainter), 128, 183
drawPoint() (QPainter), 183
drawPolygon() (QPainter), 183, 194
drawPolyline() (QPainter), 137, 183
drawPrimitive()
 BronzeStyle, 468
 QStyle, 129, 468, 475
 QStylePainter, 128
drawRect() (QPainter), 183, 188
drawRoundRect() (QPainter), 183, 195, 208, 475
drawText() (QPainter), 136, 174, 183, 190, 196
driver() (QSqlDatabase), 323
dropEvent()
 MyTableWidget, 237
 ProjectListWidget, 234
 QWidget, 230, 234, 237
 .dsp, файлы, 604
 DTD (QXmlStreamReader), 393
 dumpdoc, 558
 -dumpidl, опция (серверы ActiveX), 567
 duration() (OvenTimer), 193

DYLD_LIBRARY_PATH, переменная среды, 629
 dynamic_cast<T>(), 63, 657
 dynamicCall() (QAxBase), 559

E

Eclipse, 602, 618–623
 ECMAScript, 517–550
 edit() (ExternalEditor), 317
 Edit, меню, 86–91
 editEmployees() (MainForm), 341
 editingFinished() (QAbstractSpinBox), 273
Editor
 closeEvent(), 165
 Editor(), 166
 newFile(), 167
 okToContinue(), 165
 open(), 167
 openFile(), 168
 save(), 165
 setCurrentFile(), 165
 sizeHint(), 169
 windowMenuAction(), 165
 определение класса, 165
 EditRole (Qt), 96–98, 244, 255, 262
 effects() (BasicEffectsPlugin), 514
 #elif-директивы, 668
 #else-директивы, 668
 email clients, 318
 -embedded, опция (configure), 576
 emit псевдоключевое слово, 18
 emit() (Java), 615
 Employee, класс, 22
 EmployeeForm
 определение класса, 330
 EmployeeForm(), 331–334
 addEmployee(), 334
 deleteEmployee(), 335
 enableFindButton() (FindDialog), 18
 encodeURI() (ECMAScript), 525
 encodeURIComponent() (ECMAScript), 525
 EncodingFromTextStream (QDomNode), 412
 end() (классы-контейнеры), 281–282, 284, 289
 EndDocument (QXmlStreamReader), 393
 endDocument() (QXmlContentHandler), 405
 EndElement (QXmlStreamReader), 393
 endElement() (SaxHandler), 408
 endGroup() (QSettings), 70
 #endif, директивы, 14, 635, 668–669
 endl (std), 634
 endsWith() (QString), 293
 enqueue() (QQueue<T>), 278
 Enter, клавиша, 15, 57, 106, 273
 enterErrorState() (CursorHandler), 508
 EntityReference (QXmlStreamReader), 393
 EntryDialog, 417, 420
 entryHeight() (PrintWindow), 225
 entryInfoList() (QDir), 312
 entryList() (QDir), 311

- enum, ключевое слово, 650, 654–655
- Error (ECMAScript), 525
- #error, директива, 669
- error()
 - QAbstractSocket, 379
 - QFile, 305, 616
 - QIODevice, 397
 - QProcess, 315
 - TripPlanner, 383
- errorString()
 - QIODevice, 83, 85, 397
 - QXmlErrorHandler, 409
- Esc, клавиша, 57, 273
- escape() (Qt), 224, 237, 412
- EUC-JP, 429
- EUC-KR, 429
- eval() (ECMAScript), 525
- EvalError (ECMAScript), 525
- evalExpression() (Cell), 100
- evalFactor() (Cell), 102
- evalTerm() (Cell), 101
- evaluate() (QScriptEngine), 530, 547
- event() (QObject), 171, 177
- eventFilter() (QObject), 176–177
- EventRef (Mac OS X), 555
- ExcludeUserInput (QEventLoop), 180
- Exclusion (режим композиции), 199
- exec()
 - QCoreApplication, 3, 179, 354, 617
 - QDialog, 65, 591
 - QDrag, 233
 - QMenu, 54
 - QSqlQuery, 321
 - QThread, 361
- execDialog() (QtopiaApplication), 591–592
- execute() (QProcess), 318
- exists() (QDir), 312
- expand() (QTreeView), 252
- Expanding (QSizePolicy), 124, 148
- Expense, 587
- ExpenseDialog
 - createActions(), 591
 - createMenuOrToolBar(), 592
 - определение класса, 591
- ExpenseWindow
 - add(), 590
 - closeEvent(), 589
 - createActions(), 588
 - createMenuOrToolBar(), 589
 - loadData(), 590
 - send(), 592–593
 - определение класса, 587
- explicit, ключевое слово, 659
- extern, ключевое слово, 664
- ExternalEditor, 318
- F**
- F1, клавиша, 414, 416
- F2, клавиша, 246, 252, 340
- faceAtPosition() (Tetrahedron), 483
- fatalError() (SaxHandler), 409
- FDialogPattern (Qt), 185
- fieldIndex() (QSqlTableModel), 332
- File, меню, 52, 56–63, 71
- fileName() (QFileInfo), 61
- fill() (QPixmap), 135
- fillRect() (QPainter), 114
- filter() (PumpWindow), 542
- final, ключевое слово (Java), 647
- finalize() (Java), 644
- find()
 - MainWindow, 64
 - QWidget, 552
- findClicked() (FindDialog), 17, 615
- FindDialog
 - enableFindButton(), 18
 - findClicked(), 17, 615
 - FindDialog(), 14–17, 614
 - findNext(), 13
 - findPrevious(), 13
 - версия для Qt Jambi, 613–616
 - определение класса, 13–14
 - применение, 64–65
- FindFileDialog, класс 143–147
- findNext()
 - FindDialog, 13
 - Spreadsheet, 89
- findPrevious()
 - FindDialog, 13
 - finished() (QProcess), 315
 - Spreadsheet, 90
- first() (QSqlQuery), 321
- Fixed (QSizePolicy), 148
- FixedNotation (QTextStream), 307
- flags() (CityModel), 263
- FlatCap (Qt), 184
- flipHorizontally() (ImageWindow), 356
- FlipTransaction, 359
- FlowChartSymbolPicker done(), 245
 - FlowChartSymbolPicker(), 244
 - selectedId(), 244
 - определение класса, 243
- focusNextChild() (QWidget), 177
- Font (X11), 554
- fontMetrics() (QWidget), 169, 174
- FontRole (Qt), 255, 270
- ForcePoint (QTextStream), 307
- ForceSign (QTextStream), 307
- foreach псевдоключевое слово, 73, 284–286, 288
- foreground() (QPalette), 114
- forever псевдоключевое слово, 382
- formats()
 - QMimeData, 234, 238
 - TableMimeData, 238
- FORMS, элемент (файлы .pro), 602
- formula()
 - Cell, 97
 - Spreadsheet, 81

framebuffer, объектное расширение (OpenGL), 478, 491–497
 Freescale, 575
 froglogic, компания, 551
 fromAscii() (QString), 294
 fromLatin1() (QString), 294
 fromName() (QHostInfo), 389
 fromPage() (QPrinter), 227
 fromValue() (QVariant), 297
 FTP, 363–373
 ftpCommandStarted() (FtpGet), 367
 ftpDone()
 FtpGet, 366
 Spider, 370
 FtpGet
 done(), 364, 366
 ftpCommandStarted(), 367
 ftpDone(), 366
 FtpGet(), 365
 getFile(), 366–367
 определение класса, 364
 ftpListInfo() (Spider), 369
 Function (ECMAScript), 519, 525
 function, ключевое слово (ECMAScript), 521

G

GB18030-0, 429
 GCC, 19, 576, 598, 600
 GDI, 552–553
 generateDocumentation() (QAxBase), 558
 generateRandomTrip() (ClientSocket), 386
 get()
 QFtp, 363, 365, 370
 QHttp, 374–375
 getColor() (QColorDialog), 213, 482
 getDC() (QPaintEngine), 553–554
 getDirectory() (Spider), 369
 getFile()
 FtpGet, 365–366
 HttpGet, 373
 GetInterfaceSafetyOptions() (ObjectSafety-Impl), 563
 getOpenFileName() (QFileDialog), 58–59, 167, 315
 getPrinterDC() (QPrintEngine), 554
 getSaveFileName() (QFileDialog), 60
 getText() (QInputDialog), 253
 GIF, файлы, 48
 GNOME, 9
 GNU Compiler Collection (GCC), 19, 576, 598, 600
 GNU General Public License, 597
 goToCell() (MainWindow), 65
 GoToCellDialog
 определение класса, 27
 применение, 65
 создание с помощью Qt Designer, 23–30
 GoToCellDialog (Java), 619–620
 GoToCellDialog(), 27
 GPL, 598
 GraphPak, 121
 group() (IconEditorPlugin), 119
 GuiServer (QApplication), 576
 GWorldPtr (Mac OS X), 554

H

handle()
 QCursor, 554
 QFont, 554
 QPixmap, 554
 QRegion, 554
 QSessionManager, 554
 QSqlDriver, 323
 QSqlResult, 323
 QWidget, 554
 HardLight (режим композиции), 199
 hasAcceptableInput() (QLineEdit), 28–29
 hasFeature() (QSqlDriver), 323
 hasFormat() (QMimeData), 230
 HashMap (Java), 618
 hasLocalData() (QThreadStorage<T>), 354
 hasNext() (Итераторы в стиле Java), 280
 hasOpenGL() (QGLFormat), 484, 497
 hasOpenGLFramebufferObjects() (QGLFrame-
 bufferObject), 497
 hasPendingEvents() (QCoreApplication), 181
 hasPrevious() (Итераторы в стиле Java), 280
 hasUncaughtException() (QScriptEngine), 534
 HCURSOR (Windows), 554
 HDC (Windows), 554
 head()
 QHttp, 375
 QQueue<T>, 278
 headerData()
 BooleanModel, 269
 CityModel, 258
 CurrencyModel, 259
 HEADERS, элемент (файлы .pro), 602
 height() (QPaintDevice), 113, 116
 help()
 EntryDialog, 417, 420
 MainWindow, 416, 420
 HelpBrowser
 HelpBrowser(), 419
 showPage(), 420–421
 QAssistantClient, 422
 updateWindowTitle(), 420
 определение класса, 418, 421
 hex, манипулятор, 307
 HexSpinBox
 интеграция с Qt Designer, 117–118
 определение класса, 105
 HexSpinBox(), 105
 textFromValue(), 106
 validate(), 105
 valueFromText(), 106
 HFONT (Windows), 554
 hide() (QWidget), 126, 148

- hideEvent() (Ticker), 175
- HIView, 552
- HIViewRef (Mac OS X), 554
- Home, клавиша, 171
- homePath() (QDir), 312
- horizontalHeader() (QTableView), 80
- horizontalScrollBar() (QAbstractScrollArea), 80, 156
- HorPattern (Qt), 185
- HPALETTE (Windows), 554
- HRGN (Windows), 554
- html() (QMimeData), 239
- HTML, 5, 10, 40, 223–224, 236–237, 236, 415, 417–422, 422, 565
- HTTP, 363, 373–376
- httpDone() (HttpGet), 374
- HttpGet
- HttpGet(), 373
 - getFile(), 373
 - httpDone(), 373
- HWND (Windows), 554
- .h-файлы. См. заголовочные файлы
- I**
- IANA, 231
- IBM 8xx, 429
- IBM DB2, 319
- icon() (IconEditorPlugin), 119
- IconEditor
 - IconEditor(), 109
 - mouseMoveEvent(), 115
 - mousePressEvent(), 115
 - paintEvent(), 112
 - pixelRect(), 114
 - setIconImage(), 111
 - setImagePixel(), 115
 - setPenColor(), 111
 - setZoomFactor(), 111
 - sizeHint(), 110
 - интеграция с Qt Designer, 117–120
 - определение класса, 108–109
 - с полосами прокрутки, 155
- IconEditorPlugin
 - createWidget(), 120
 - group(), 119
 - icon(), 119
 - IconEditorPlugin(), 118
 - includeFile(), 119
 - isContainer(), 119
 - name(), 119
 - toolTip(), 119
 - whatsThis(), 119
 - определение класса, 118
- IconRole (Qt), 244
- IDE, (интегрированная среда разработки) 602–604, 618–623
- IDispatch (Windows), 559
- #if, директива, 668–669
- #ifdef, директива, 669
- #ifndef, директивы, 12, 635, 669
- IFontDisp (Windows), 557
- ignore() (QEvent), 60, 165
- IgnoreAction (Qt), 233
- Ignored (QSizePolicy), 149
- image()
 - QClipboard, 240
 - TransactionThread, 358
- imageCount() (CursorHandler), 505
- imageData() (QMimeData), 239
- ImageMagick, 313
- imageSpace(), 311
- ImageWindow
 - addTransaction(), 356
 - allTransactionsDone(), 356
 - flipHorizontally(), 356
 - ImageWindow(), 355
 - TransactionThread, 358
- import, объявления (Java), 613, 616, 621, 628, 630
- #include, директивы, 633, 636, 667
- include(), директивы (файлы .pro), 607, 627
- includeFile() (IconEditorPlugin), 119
- INCLUDEPATH, элемент (файлы .pro), 498, 603, 627
- incomingConnection() (TripServer), 384
- index() (BooleanModel), 266
- indexOf()
 - QLayout, 149
 - QString, 293
- Infinity (ECMAScript), 525
- information() (QMessageBox), 57
- initFrom()
 - QPainter, 135, 197
 - QStyleOption, 129, 462
- initialize() (QApplication), 617
- initializeGL()
 - QGLWidget, 480, 485, 493
 - Teapots, 493
 - Tetrahedron, 480
- insert()
 - QLinkedList<T>, 278
 - QMap<K,T>, 286
 - QMultiMap<K,T>, 287
 - QString, 293
 - TeamLeadersDialog, 250
 - итераторы в стиле Java, 281
- INSERT, инструкции (SQL), 322, 324
- insertMulti()
 - QHash<K,T>, 288
 - QMap<K,T>, 287
- insertRow()
 - QAbstractItemModel, 324, 340
 - QTableWidget, 246
- installEventFilter() (QObject), 176, 178
- INSTALLS, элемент (файлы .pro), 582
- installTranslator() (QCoreApplication), 433
- instance()
 - QPluginLoader, 512

QWSServer, 579
 instanceof, оператор (ECMAScript), 523, 527
 Intel x86, 575, 638
 Intermediate (QValidator), 106
 internalPointer() (QModelIndex), 267
 Internet Explorer, 422, 560
 Invalid
 QValidator, 105
 QXmlStreamReader, 393
 invisibleRootItem() (QTreeWidget), 248, 398, 409
 invokeMethod() (QMetaObject), 362
 IOObjectSafety (Windows), 562–563
 IPC (Inter-Process Communication – связь между процессами), 313–318
 См. также ActiveX
 IPictureDisp (Windows), 557
 IP-адреса, 387, 389
 isActive() (QSqlQuery), 321
 isCharacter() (QXmlStreamReader), 394
 Iscii, 429
 isContainer() (IconEditorPlugin), 119
 isDigit() (QChar), 427
 isEmpty()
 QString, 294
 классы-контейнеры, 281
 isFinite() (ECMAScript), 525
 isLetter() (QChar), 427
 isLetterOrNumber() (QChar), 427
 isLower() (QChar), 427
 isMark() (QChar), 427
 isNaN() (ECMAScript), 525
 isNull()
 QImage, 616
 QPixmap, 616
 isNumber() (QChar), 427
 ISO 2022, 430
 ISO 8859-1 (Latin-1), 291, 310, 322, 426–430
 ISO 8859-15 (Latin-9), 434
 ISO 8859-x, 430
 isPrint() (QChar), 427
 isPunct() (QChar), 427
 isRightToLeft() (QApplication), 435
 isSessionRestored() (QApplication), 573, 574
 isSpace() (QChar), 427
 isStartElement() (QXmlStreamReader), 394
 isSymbol() (QChar), 427
 isUpper() (QChar), 427
 isValid() (QVariant), 97
 item() (QTableWidget), 81
 itemChange()
 Node, 208
 QGraphicsItem, 204
 itemChanged() (QTableWidget), 79
 ItemIgnoresTransformations (QGraphicsItem), 218–219
 ItemIsEditable (Qt), 262
 ItemIsEnabled (Qt), 262
 ItemIsSelectable (Qt), 262
 IUnknown (Windows), 559, 563

J

jam. *См.* Boost.Build
 JambiPlotter (Java), 628
 .jar, файлы, 616, 626
 Java Native Interface, 612
 Java, 405, 575, 612–630, 631–673
 Java, итераторы в стиле, 279–281, 288
 Java, компилятор пользовательского интер-фейса (juic), 615, 621
 JavaScript, 517, 565
 JIS, 429
 JNI, 612
 join() (QStringList), 294, 308
 JournalView, 441–442
 JPEG, 48
 .js, файлы, 530
 .jui, файлы, 619, 621
 juic, 615, 621
 jumpToNextImage() (CursorHandler), 508

K

KD Chart, 121
 KDAB (Klarälvdalens Datakonsult), 551
 KDE, 9
 KDevelop, 602
 KeepSize (QSplitter), 152
 key()
 QKeyEvent, 171
 итераторы в стиле Java, 137, 288
 итераторы в стиле STL, 259, 288
 keyPressEvent()
 CodeEditor, 171
 MyLineEdit, 176
 Plotter, 133
 QWidget, 133, 171
 keyReleaseEvent() (QWidget), 171
 keys()
 BronzeStylePlugin, 500
 CursorPlugin, 502
 ассоциативные контейнеры, 286, 288
 killTimer() (QObject), 175
 Klarälvdalens Datakonsult, компания, 551
 KOI8, 429

L

LabeledCircle, 643
 LabeledLineEdit (Java), 621–623
 Language, меню, 437, 439–440
 last() (QSqlQuery), 321
 Latin-1, 291, 310, 322, 426–430
 Latin-9. *См.* ISO 8859-15
 LD_LIBRARY_PATH, переменная среды, 629
 leaders() (TeamLeadersDialog), 251
 left() (QString), 292
 leftColumn() (QTableWidgetSelectionRange), 67
 length() (QString), 291, 294
 lib, шаблон (файлы .pro), 602, 627
 LIBS, элемент (файлы .pro), 498, 603, 627

Lighten (режим композиции), 199
 Linguist. См. Qt Linguist
 Link
 ~Link(), 203
 Link(), 202
 setColor(), 203
 trackNodes(), 203
 определение класса, 201
 LinkAction (Qt), 233
 Linux, 551–555, 575, 599–600
 Linux для встроенных систем, 575–594
 List (Java), 618
 list() (QFtp), 366, 369
 listen() (QAbstractSocket), 387
 listInfo() (QFtp), 368, 369
 load()
 QTranslator, 433, 440, 442
 QUiLoader, 38, 534
 loadData()
 ExpenseWindow, 590
 PumpWindow, 542
 loadFile() (MainWindow), 58
 loadFiles() (MainWindow), 161
 loadPlugins() (TextArtDialog), 512
 localData() (QThreadStorage<T>), 354
 LocalDate (Qt), 435
 localeAwareCompare() (QString), 293, 435
 localeconv() (std), 435
 LocalHost (QHostAddress), 379, 389
 lock() (QMutex), 347, 349
 lockForRead() (QReadWriteLock), 348
 lockForWrite() (QReadWriteLock), 349
 login() (QFtp), 365–366, 369
 «look and feel» (изобразительные средства
 платформы), 9, 130, 446–477
 lookupHost() (QHostInfo), 389
 lrelease, 425, 443–445
 «LTR»-маркер, 434
 lupdate, 425, 430–432, 443–445

M

Mac OS X, 551–555, 598–599
 Mac, стиль, 9, 129, 462
 macCGHandle() (QPixmap), 554
 macEvent() (QWidget), 555
 macEventFilter() (QApplication), 555
 MacintoshVersion (QSysInfo), 554
 macQDAlphaHandle() (QPixmap), 554
 macQDHandle() (QPixmap), 554
 maxх, условие (файлы .pro), 606
 MagicNumber (Spreadsheet), 79, 85
 MailClient, 152–155
 main()
 argc и argv, параметры, 3, 633
 в простом приложении Qt Jambi, 617
 в простом приложении Qt, 3
 для консольных приложений, 363
 для приложений ActiveX, 567

 для приложений OpenGL, 484, 497
 для приложений Qt/Embedded Linux, 584
 для приложений Qtoria, 584
 для приложений SDI, 71
 для приложений баз данных, 320
 для приложений с поддержкой скрип-
 тов, 544
 для программ C++, 633
 для экранной заставки, 74
 при интернационализации приложений, 433

MainForm

addDepartment(), 340
 createDepartmentPanel(), 337
 createEmployeePanel(), 338
 deleteDepartment(), 340
 editEmployees(), 341
 MainForm(), 337
 updateEmployeeView(), 339
 определение класса, 335

MainWindow

about(), 68
 addEditor(), 162
 closeEvent(), 60, 165
 createActions(), 50, 72, 437
 createContextMenu(), 53
 createLanguageMenu(), 439
 createMenus(), 52, 164, 438
 createStatusBar(), 55
 createToolBars(), 54
 cut(), 163
 directoryOf(), 416
 find(), 64
 goToCell(), 65
 help(), 416, 420
 loadFile(), 58
 loadFiles(), 161
 MainWindow(), 47, 73, 160, 437
 newFile(), 56, 72, 161
 okToContinue(), 57
 open(), 58, 162
 openRecentFile(), 63
 readSettings(), 70, 159
 retranslateUi(), 438
 save(), 59, 163
 saveAs(), 59
 saveFile(), 59
 setCurrentFile(), 60
 sort(), 66–68
 spreadsheetModified(), 56
 strippedName(), 61
 switchLanguage(), 440
 updateRecentFileActions(), 61
 updateStatusBar(), 55
 writeSettings(), 69, 159
 активное окно редактора, 163
 определение класса, 45–47, 229

make, 5, 603, 609

makeCurrent() (QGLWidget), 483, 487

makefile, файлы, 5, 18, 565, 601, 603–604, 608

- makeqpf, 579
 - manhattanDistance(), 648
 - manhattanLength() (QPoint), 233
 - map<K,T> (std), 288
 - Margin (Plotter), 123
 - Math (ECMAScript), 525
 - MathML, 392
 - Maximum (QSizePolicy), 148
 - Maximum, свойство (QProgressBar), 379
 - MDI, 74, 77, 159–169
 - мемсру() (std), 651
 - menuBar() (QMainWindow), 52
 - message() (Transaction), 360
 - metaObject() (QObject), 21
 - Microsoft Internet Explorer, 422, 560
 - Microsoft SQL Server, 319
 - Microsoft Visual Basic, 565
 - Microsoft Visual C++ (MSVC), 5, 19, 38, 297, 554
 - Microsoft Visual Studio, 5, 602, 604
 - mid() (QString), 65, 292
 - MIME, типы, 230, 235–239
 - mimeData()
 - QClipboard, 240
 - QDropEvent, 230, 234
 - MinGW, 5, 598
 - Minimum (QSizePolicy), 111, 148
 - minimum, свойство (QProgressBar), 379
 - MinimumExpanding (QSizePolicy), 149
 - minimumSizeHint()
 - Plotter, 127
 - QWidget, 17, 127, 149
 - MIPS, 575
 - mirror() (QImage), 360
 - MiterJoin (Qt), 184
 - mkdir()
 - QDir, 312
 - QDirModel, 253
 - QFtp, 366
 - моч, 18, 21, 63, 117, 514, 561, 601, 613
 - modified() (Spreadsheet), 79, 83
 - modifiers() (QKeyEvent), 133, 171
 - Motif, миграция, 551
 - Motif, стиль системы, 9, 53, 129, 462
 - Motorola 68000, 575
 - mouseDoubleClickEvent()
 - Node, 208
 - Tetrahedron, 482
 - mouseMoveEvent()
 - IconEditor, 115
 - MyTableWidget, 235
 - Plotter, 131
 - ProjectListWidget, 232
 - QGraphicsItem, 204
 - Teapots, 496
 - Tetrahedron, 481
 - mousePressEvent()
 - IconEditor, 115
 - OvenTimer, 193
 - Plotter, 130
 - ProjectListWidget, 232
 - Teapots, 496
 - Tetrahedron, 482
 - mouseReleaseEvent()
 - Plotter, 131
 - QWidget, 131, 240, 496
 - Teapots, 496
 - MoveAction (Qt), 233
 - moveToThread() (QObject), 355
 - Movie, 279
 - MRU, файлы. *См. недавно открывавшиеся файлы*
 - MS-DOS, приглашение, 5, 598
 - MSG (Windows), 555
 - MuleLao-1, 430
 - Multiply (режим композиции), 199
 - mutable, ключевое слово, 96, 99, 504, 658
 - MVC (модель-представление-контроллер), 241
 - MyInteger, 659
 - MySQL, 319
 - MyTableWidget
 - dropEvent(), 237
 - mouseMoveEvent(), 235
 - performDrag(), 236
 - toCsv(), 236
 - toHtml(), 236
 - MyVector, 659
- ## N
- name()
 - IconEditorPlugin, 119
 - QColor, 214–215
 - NaN (ECMAScript), 525
 - new [], оператор, 651
 - new, оператор
 - в C++, 28, 66, 71–73, 641, 645–646, 661
 - в ECMAScript, 519
 - newFile()
 - Editor, 167
 - MainWindow, 56, 72, 162
 - newFunction() (QScriptEngine), 547, 550
 - newPage() (QPrinter), 221, 223, 227
 - newQMetaObject() (QScriptEngine), 546
 - newQObject() (QScriptEngine), 543
 - next()
 - QSqlQuery, 321
 - итераторы в стиле Java, 280, 288
 - nextPrime(), 665
 - nextRandomNumber(), 663, 666
 - nmake, 5, 604, 609
 - NoBrush (Qt), 185
 - Node
 - Node(), 205, 265
 - addLink(), 206
 - boundingRect(), 207
 - itemChange(), 208
 - mouseDoubleClickEvent(), 208
 - Node(), 205, 265
 - outlineRect(), 206

paint(), 207
 removeLink(), 206
 roundness(), 208
 setText(), 206
 setTextColor(), 206
 shape(), 207
 определение класса, 203, 264
 nodeFromIndex() (BooleanModel), 267
 NodePair (DiagramWindow), 209
 NoEditTriggers (QAbstractItemView), 246, 328, 339
 NoError (QFile), 305
 NoPen (Qt), 184
 normalized() (QRect), 128, 131
 notify() (QCoreApplication), 178
 null (ECMAScript), 519
 Null (QChar), 98–99
 Number (ECMAScript), 519, 525
 number() (QString), 83, 106, 292
 numCopies() (QPrinter), 227
 numRowsAffected() (QSqlQuery), 322

O

Object
 в C#, 641
 в ECMAScript, 519, 525, 534
 в Java, 617, 641
 ODBC, 319
 offsetOf() (CityModel), 263
 Ok (QDialogButtonBox), 30
 okToContinue()
 Editor, 165
 MainWindow, 56
 OLE_COLOR (Windows), 557
 on_browseButton_clicked() (ConvertDialog), 314–315
 on_lineEdit_textChanged() (GoToCellDialog), 28–29
 on_lineEdit_textChanged(), 28–29
 on_xxx_xxx(), слоты, 28
 OpaqueMode (Qt), 187
 open source, версия Qt, 539, 597–598
 open()
 Editor, 167
 MainWindow, 58, 162
 QIODevice, 84–85, 299, 305, 373
 QSqlDatabase, 320
 openFile() (Editor), 168
 OpenGL, 200, 478–497
 openRecentFile() (MainWindow), 63
 openUrl() (QDesktopServices), 318, 416, 417
 operator--() (Итераторы в стиле STL), 282
 operator(), 93
 operator*() (Итераторы в стиле STL), 282
 operator[]
 QMap<K, T>, 286
 QVector<T>, 277
 классов-контейнеров, 284

operator+() (QString), 291
 operator++() (Итераторы в стиле STL), 282
 operator+=() (QString), 291
 operator<() (keys in maps), 287
 operator<<()
 QDataStream, 301
 QTextStream, 305
 классов-контейнеров, 277
 operator=() (Point2D), 662
 operator==() (ключи в хеш-таблицах), 287
 operator>>()
 QDataStream, 302
 QTextStream, 305
 Oracle, 319
 ORDER BY, условие (SQL), 328
 ostream (std), 659
 outlineRect() (Node), 206
 OvenTimer
 draw(), 194–195
 duration(), 193
 mousePressEvent(), 193
 OvenTimer(), 192
 paintEvent(), 193
 setDuration(), 192
 timeout(), 191
 определение класса, 191
 Overlay (режим композиции), 199
 override ключевое слово (C#), 643

P

paginate() (PrintWindow), 224
 paint()
 Annotation, 220
 CityBlock, 218
 Node, 207
 TrackDelegate, 281
 paintEngine()
 QPaintDevice, 553
 QPainter, 553
 paintEvent()
 IconEditor, 112
 OvenTimer, 193
 paintGL()
 QGLWidget, 481, 486, 495
 Teapots, 495
 Tetrahedron, 481
 Painting, 301
 pair<T1, T2> (std), 288, 297
 palette() (QWidget), 114
 parent()
 BooleanModel, 267
 QModelIndex, 255
 Parent, параметр, 13, 15
 parse() (QXmlSimpleReader), 407
 parseBookindexElement() (DomParser), 403
 parseEntryElement() (DomParser), 403
 parseFloat() (ECMAScript), 525
 parseInt() (ECMAScript), 525
 parsePageElement() (DomParser), 404

- paste()
 - DiagramWindow, 214
 - QTextEdit, 240
 - Spreadsheet, 88
- PATH, переменная среды, 4, 422, 571, 600, 629
- PatternSyntax (QRegExp), 254
- Pbuffer, расширение (OpenGL), 478, 497
- PDF, 221
- PE_FrameFocusRect (QStyle), 129
- peek() (QIODevice), 305, 504
- pendingDatagramSize() (QUdpSocket), 391
- performDrag()
 - MyTableWidget, 235
 - ProjectListWidget, 232
- Personal Information Manager, 585
- Picture (X11), 554
- PIM, 585
- pixelMetric() (BronzeStyle), 467
- pixelRect() (IconEditor), 114
- Pixmap (X11), 554
- pixmap() (QClipboard), 240
- Plastique, стиль системы, 9, 129, 186, 462
- PlayerWindow
 - PlayerWindow(), 557
 - timerEvent(), 558
 - определение класса, 556
- PlotSettings
 - adjust(), 138
 - adjustAxis(), 138
 - Java-привязки, 623–628
 - PlotSettings(), 137
 - scroll(), 137
 - spanX(), 123
 - spanY(), 123
 - определение класса, 123
- Plotter
 - clearCurve(), 127
 - drawCurves(), 136
 - drawGrid(), 135
 - Java-привязки, 623–630
 - keyPressEvent(), 133
 - Margin, 123
 - minimumSizeHint(), 127
 - mouseMoveEvent(), 131
 - mousePressEvent(), 130
 - mouseReleaseEvent(), 131
 - paintEvent(), 128
 - Plotter(), 124
 - refreshPixmap(), 134
 - resizeEvent(), 129
 - setCurveData(), 127
 - setPlotSettings(), 126
 - sizeHint(), 128
 - updateRubberBandRegion(), 134
 - wheelEvent(), 134
 - zoomIn(), 127
 - zoomOut(), 126
 - определение класса, 122
- Plotter, 128
 - QWidget, 112, 128, 174, 183, 193, 197, 462, 487, 552–553
 - Ticker, 174
 - VowelCube, 487, 490
- Plus (режим композиции), 199
- PNG, файлы, 48
- PNM, файлы, 48
- Point2D
 - встроенная реализация, 639
 - конструктор копирования и оператор присваивания, 662
 - невстроенная реализация, 639–640
 - перегрузка операторов, 659
 - применение, 641
- polish()
 - BronzeStyle, 465–466
 - QStyle, 465–466
- pop() (QStack<T>), 278
- populateListWidget() (TextArtDialog), 512
- populateRandomArray(), 663, 666
- port() (QUrl), 366
- post()
 - QGraphicsItem, 203
 - QMouseEvent, 115, 232
- post() (QHttp), 375
- postEvent() (QCoreApplication), 361
- PostgreSQL, 319
- PostScript, 221, 579
- PowerPC, 575, 638
- PreferenceDialog, 150
- Preferred (QSizePolicy), 125, 148
- prefix, опция (configure), 599
- prepare() (QSqlQuery), 322
- prepareGeometryChange() (QGraphicsItem), 205, 219
- previous()
 - QSqlQuery, 321
 - итераторы в стиле Java, 280, 288
- print()
 - ECMAScript, 525
 - QTextDocument, 223
- printBox() (PrintWindow), 228
- printFlowerGuide() (PrintWindow), 223
- printHtml() (PrintWindow), 224
- printImage() (PrintWindow), 222
- printPage() (PrintWindow), 227
- printPages() (PrintWindow), 225
- PrintWindow
 - entryHeight(), 225
 - paginate(), 224
 - printBox(), 228
 - printFlowerGuide(), 223
 - printHtml(), 224
 - printImage(), 222
 - printPage(), 227
 - printPages(), 225
- .pro, файлы, 602–607
- include(), директивы, 607

- include, путь, 498, 603
- внешние библиотеки, 498, 602
- для Java-привязок, 627
- для интернационализации приложений, 433, 443
- для использования QAssistantClient, 421
- для использования QUiLoader, 38
- для консольных приложений, 310
- для подключаемых к приложению модулей, 516
- для подключаемых модулей Qt Designer, 120
- для подключаемых модулей Qt, 500, 510
- для приложений ActiveX, 559, 564, 568
- для приложений OpenGL, 484
- для приложений Qtopia, 581–583, 587
- для приложений XML, 400, 405
- для приложений баз данных, 326
- для приложений с поддержкой скриптов, 530
- для сборки с рекурсивными поддиректориями, 602
- для сетевых приложений, 367
- для статических и общих библиотек, 602
- комментарии, 604
- операторы, 604–605
- отладочный и рабочий режимы, 501, 603
- переменные средства доступа, 605
- преобразование в make-файлы, 5, 604
- преобразование в файлы проекта IDE, 5, 604
- ресурсы, 48, 125, 313, 435, 602
- создание при помощи qmake, 4, 604
- условные выражения, 606
- processError() (ConvertDialog), 317
- processEvents() (QCoreApplication), 179–180
- processFinished() (ConvertDialog), 317
- ProcessingInstruction (QXmlStreamReader), 393
- processNextDirectory() (Spider), 369
- processPendingDatagrams() (WeatherStation), 390
- project, опция (qmake), 4, 604
- ProjectListWidget
 - dragEnterEvent(), 234
 - dragMoveEvent(), 234
 - dropEvent(), 234
 - mouseMoveEvent(), 232
 - mousePressEvent(), 232
 - performDrag(), 233
 - ProjectListWidget(), 232
 - определение класса, 232
- properties() (DiagramWindow), 213
- property() (QObject), 559
- propertyChanged() (QAxBindable), 562
- PumpFilter, 543
- pumpFilterConstructor(), 547
- PumpFilterPrototype, 548–549
- PumpSpreadsheet, 541
- PumpWindow, 542–543
- PurifyPlus, 638
- push() (QStack<T>), 278
- put() (QFtp), 366
- Python, 610
- Q**
 - Q_ARG(), 362
 - Q_ASSERT(), 400
 - Q_CC_xxx, 553
 - Q_CLASSINFO(), 565–567
 - Q_DECLARE_INTERFACE(), 510–511
 - Q_DECLARE_METATYPE(), 296, 536, 548–549
 - Q_DECLARE_TR_FUNCTIONS(), 203, 430
 - Q_ENUMS(), 542–543, 546, 556, 561
 - Q_EXPORT_PLUGIN2(), 120, 500, 503, 515
 - Q_INTERFACES(), 117, 514
 - Q_OBJECT макрос
 - выполнения компилятора moc, 18
 - для метаобъектной системы, 21
 - для свойств, 108
 - для сигналов и слотов, 13, 21, 45
 - для функции tr(), 16, 430
 - Q_OS_xxx, 553
 - Q_PROPERTY(), 108, 548, 621
 - Q_WS_xxx, 553
 - qAbs(), 282, 291
 - QAbstractFontEngine, 499
 - QAbstractItemDelegate, 281
 - closeEditor(), 273
 - commitData(), 273
 - createEditor(), 273
 - paint(), 281
 - setEditorData(), 274
 - setModelData(), 274
 - QAbstractItemModel, 257
 - columnCount(), 258, 267
 - createIndex(), 262
 - data(), 258, 268, 270
 - dataChanged(), 262
 - flags(), 263
 - headerData(), 258, 259, 269
 - index(), 266
 - parent(), 267
 - removeRows(), 251
 - reset(), 259, 263, 266
 - rowCount(), 258, 267
 - setData(), 262
 - создание подкласса, 265
 - QAbstractItemView, 157
 - AnyKeyPressed, 244
 - ContiguousSelection, 80, 87, 237
 - NoEditTriggers, 246, 328, 339
 - resizeColumnsToContents(), 328
 - selectAll(), 51, 90
 - SelectRows, 338
 - setEditTriggers(), 245, 246, 250, 327
 - setItemDelegate(), 271
 - setModel(), 250, 328
 - setSelectionBehavior(), 328
 - setSelectionMode(), 328
 - SingleSelection, 338

- QAbstractListModel, 257
- QAbstractScrollArea, 39, 41, 81, 157
- QAbstractSocket, 361, 376
- QAbstractTableModel, 257, 260
- QAccessibleBridge, 499
- QAccessibleBridgePlugin, 499
- QAccessibleInterface, 499
- QAccessiblePlugin, 499
- QAction, 50–51
 - data(), 535
 - setCheckable(), 51
 - setChecked(), 164
 - setData(), 535
 - setEnabled(), 164
 - setShortcutContext(), 172
 - setStatusTip(), 414
 - setToolTip(), 413
 - setVisible(), 50, 62
 - toggled(), 51
 - triggered(), 50
 - сравнение с событиями нажатия клавиш, 172
 - эксклюзивные группы, 51
 - элемент «данные», 63, 440, 535
- QActionGroup, 51, 163–164, 440
- qApp, глобальная переменная, 51, 178
- QApplication, 3
 - aboutQt(), 51
 - beep(), 90
 - clipboard(), 87, 214, 240
 - closeAllWindows(), 72
 - commitData(), 569, 571
 - exec(), 3
 - GuiServer, 576
 - initialize(), 617
 - isRightToLeft(), 435
 - isSessionRestored(), 573, 574
 - macEventFilter(), 555
 - QCoreApplication создание подкласса, 570
 - qwsEventFilter(), 555
 - qwsSetDecoration(), 579
 - restoreOverrideCursor(), 84, 131
 - saveState(), 569–570
 - sessionId(), 573
 - sessionKey(), 573
 - setLayoutDirection(), 434
 - setOverrideCursor(), 84, 131
 - setStyle(), 462, 477, 501
 - setStyleSheet(), 447
 - startDragDistance(), 233
 - style(), 129
 - topLevelWidgets(), 73
 - winEventFilter(), 555
 - x11EventFilter(), 555
 - в консольных приложениях, 310, 364
 - свойство quitOnLastWindowClosed, 60
- QAssistantClient, 422
- QAXAGG_IUNKNOWN, 563
- QAxAggregated, 562
- QAxBase, 556–557
 - dynamicCall(), 559
 - generateDocumentation(), 558
 - queryInterface(), 559
 - querySubObject(), 559
- QAxBindable, 559–560
 - createAggregate(), 562
 - propertyChanged(), 562–563
 - requestPropertyChange(), 562–563
- QAXCLASS(), 566, 568
- QAxContainer, модуль, 556–559
- QAXFACTORY_BEGIN(), 566, 568
- QAXFACTORY_DEFAULT(), 561, 564, 566
- QAXFACTORY_END(), 566, 568
- QAXFACTORY_EXPORT(), 566
- QAxObject, 556–557, 559
- QAxServer, модуль, 556, 560–568
- QAXTYPE(), 566, 568
- QAxWidget, 556–557, 559
- qBinaryFind(), 289
- QBitArray, 297, 506, 509, 670
- QBitmap, 506
- qBound(), 192
- QBrush, 114, 185, 200
- QBuffer, 298, 373, 380
- QByteArray, 294, 298, 304
 - constData(), 294
 - data(), 294
- QCache<K,T>, 287
- QCanvas (Qt 3). См. классы графических представлений
- QCDEStyle, 129, 462
- QChar, 291, 426, 618
 - isXxx(), функции, 427
 - Null, 98–99
 - toLatin1(), 427
 - unicode(), 427
- QCheckBox, 39
 - создание стилей с помощью QStyle, 467
 - создание стилей с помощью таблиц стилей, 450
- QCleanlooksStyle, 129, 462
- QClipboard, 240
 - setText(), 87–88, 214
 - text(), 88, 214
- QCloseEvent, 60, 165, 347
- QColor, 110, 114
 - colorNames(), 253
 - name(), 214, 215
 - setNamedColor(), 448
- QColorDialog, 42, 213, 482
- QColormap, 554
- QComboBox, 41, 332
 - addItem(), 36, 254
 - currentIndex(), 67
 - создание стилей с помощью таблиц стилей, 357–461
 - элемент «данные», 253

- QCommonStyle, 462, 465
- qCompress(), 305
- qconfig, опция (configure), 578
- QConicalGradient, 195
- QContactModel, 592
- QContactSelector, 592
- QCOP, 577, 594
- QCopChannel, 577
- qCopy(), 289
- QCoreApplication, 364
 - addLibraryPath(), 501
 - applicationDirPath(), 417
 - arguments(), 161, 364, 545
 - exec(); 3, 179, 354, 617
 - hasPendingEvents(), 181
 - installTranslator(), 433
 - notify(), 178
 - postEvent(), 361
 - processEvents(), 179–180
 - quit(), 6, 60
 - removePostedEvents(), 361
 - translate(), 203, 431
- QCursor, 554
- QDataStream, 83–86, 299–304
 - Qt_4_3, 85, 300, 302
 - readRawBytes(), 302
 - setByteOrder(), 506
 - setVersion(), 85, 299, 302–303
 - skipRawData(), 506
 - writeRawBytes(), 302
 - версии, 85
 - для массива байтов, 380
 - для сокета, 376, 386
 - классы-контейнеры, 276
 - поддерживаемые типы данных, 299
 - пользовательские типы, 297
 - с QCOP, 577
 - синтаксис, 84
 - формат двоичных данных, 85, 299, 300, 505
- QDataWidgetMapper, 331–335
- QDate, 435
- QDateEdit, 41, 435, 477
- QDateTime, 192–193, 435
- QDateTimeEdit, 41, 435
- qDebug(), 292
- QDecoration, 499, 578
- QDecorationPlugin, 499, 578
- qDeleteAll(), 205, 265, 290
- QDesignerCustomWidgetInterface, 118
- QDesktopServices, 318, 416
- QDevelop, 602
- QDial, 41
- QDialog, 13
 - accept(), 29, 65
 - Accepted, 29, 65
 - done(), 245, 247
 - exec(), 65, 591
 - reject(), 29, 65
 - Rejected, 29, 65
 - setModal(), 65, 180
 - создание подкласса, 13, 26, 36, 243, 613
- QDialogButtonBox, 29–30, 332, 467
- QDir, 253, 311
 - absoluteFilePath(), 416
 - currentPath(), 312
 - entryInfoList(), 312
 - entryList(), 311
 - exists(), 312
 - homePath(), 312
 - mkdir(), 312
 - rename(), 312
 - separator(), 316
 - toNativeSeparators(), 312
- QDirModel, 249, 251–253
- QDockWidget, 156–158
- QDomDocument, 403
 - documentElement(), 403
 - save(), 410, 410–411
 - setContent(), 402
- QDomElement, 401, 403, 404
- QDomNode, 403
 - EncodingFromTextStream, 412
 - toElement(), 403
- QDomText, 401
- QDoubleSpinBox, 41, 477
- QDoubleValidator, 28
- QDrag, 233
- QDragEnterEvent, 230, 234
 - accept(), 234
 - acceptProposedAction(), 230
 - mimeData(), 230
 - setDropAction(), 234
 - source(), 234
- QDragMoveEvent, 234
 - accept(), 234
 - setDropAction(), 234
- QDropEvent
 - mimeData(), 235
 - setDropAction(), 234
- QErrorMessage, 42–43
- QEvent
 - accept(), 60, 165, 347
 - ignore(), 60, 165
 - type(), 170, 171
 - сравнение с «родными» событиями, 555
- QEventLoop, 180
- QFile, 83, 298
 - close(), 318
 - error(), 305, 616
 - NoError, 305
 - open(), 84, 86, 299, 305
 - remove(), 312
 - неявное закрытие, 300
 - неявное открытие, 407
- QFileDialog, 43
 - DontConfirmOverwrite, 60
 - getOpenFileName(), 58–59, 167, 315
 - getSaveFileName(), 60

- QFileInfo, 61, 311
- QFileSystemWatcher, 312, 529
- qFill(), 289
- qFind(), 289
- qFindChild<T>(), 38
- QFlags<T>, 656
- QFont, 184, 554
- QFontComboBox, 42
- QFontDialog, 42
- QFontEnginePlugin, 499
- QFontMetrics, 136, 174, 219, 514
- QFrame, 39
 - StyledPanel, 472
 - создание стилей с помощью QStyle, 471–472
- QFtp, 363
 - cd(), 366, 370
 - close(), 365, 366
 - commandFinished(), 367
 - commandStarted(), 367
 - connectToHost(), 365–366, 369
 - done(), 365–367
 - get(), 363, 365–366, 370
 - list(), 366, 369
 - listInfo(), 368–369
 - login(), 365–366, 369
 - mkdir(), 366
 - put(), 366
 - rawCommand(), 367
 - read(), 373
 - readAll(), 373
 - readyRead(), 373
 - remove(), 366
 - rename(), 366
 - rmdir(), 366
 - stateChanged(), 367
- qglClearColor(), 480
- qglColor(), 481, 482
 - renderText(), 484, 488
 - resizeGL(), 480, 486, 494
 - setAutoBufferSwap(), 490
 - setFormat(), 480
 - swapBuffers(), 490
 - updateGL(), 482, 496
- QGLFramebufferObject, 478, 492
 - bind(), 494
 - hasOpenGLFramebufferObjects(), 497
 - release(), 495
 - toImage(), 497
- QGLPixelBuffer, 478, 492, 497
- QGLWidget, 478, 480
 - initializeGL(), 480, 486, 493
 - makeCurrent(), 483, 487
 - paintGL(), 481, 486, 495–496
 - создание подклассов, 479, 484, 492
- QGradient. См. градиенты
- QGraphicsItem, 199
 - boundingRect(), 207, 216, 219
 - itemChange(), 204
 - ItemIgnoresTransformations, 218–219
 - mouseMoveEvent(), 204
 - paint(), 207, 217, 220
 - pos(), 203
 - prepareGeometryChange(), 205, 219
 - setFlag(), 219
 - setFlags(), 203, 205
 - setZValue(), 203, 211, 219
 - shape(), 207
 - update(), 205, 219
 - создание подклассов, 203, 216, 218
- QGraphicsItemAnimation, 221
- QGraphicsItemGroup, 200
- QGraphicsLineItem (создание подклассов), 201
- QGraphicsScene, 199
 - render(), 223
 - selectedItems(), 211
- QGraphicsView, 199–200
 - itemChange(), 208
 - mouseDoubleClickEvent(), 208
 - render(), 223
 - RubberBandDrag, 210
 - scale(), 221
 - setViewport(), 200
 - создание подклассов, 220
- qGreater<T>(), 290
- QGridLayout, 8, 145–147
- QGroupBox, 39
- qHash(), 287
- QHash<K, T>, 287–288, 618
- QHBoxLayout, 8, 145
 - addLayout(), 16
 - addWidget(), 16
- QHeaderView, 80, 328
- QHostAddress
 - Any, 387
 - LocalHost, 379, 389
- QHttp, 373
 - close(), 374
 - done(), 373
 - get(), 374–375
 - head(), 375
 - post(), 375
 - read(), 376
 - readAll(), 376
 - readyRead(), 376
 - request(), 375–376
 - requestFinished(), 376
 - requestStarted(), 376
 - setHost(), 374–375
 - setUser(), 375
- QIcon, 119, 296, 616
- QIconEnginePluginV2, 499
- QIconEngineV2, 499
- QicsTable, 77
- QImage, 110
 - convertToFormat(), 111
 - height(), 116
 - isNull(), 616
 - mirror(), 360

- rect(), 116
- setPixel(), 116
- width(), 116
- из трехмерной сцены, 497
- как устройство рисования, 182, 196–199
- печатать, 221
- режимы композиции, 198–199, 468
- QImageIOHandler, 499
 - canRead(), 504
 - currentImageNumber(), 504
 - imageCount(), 505
 - jumpToNextImage(), 508
 - read(), 505–508
 - создание подклассов, 503
- QImageIOPlugin, 499
 - capabilities(), 502
 - create(), 502
 - keys(), 502
 - создание подклассов, 501
- QImageReader, 502
- QInputContext, 499
- QInputContextPlugin, 499
- QInputDialog, 42–43, 253
- qintN, 84, 299
- QIntValidator, 28, 331
- QIODevice, 298, 402
 - close(), 593
 - error(), 397
 - errorString(), 83, 86, 397
 - peek(), 305, 504
 - readAll(), 304
 - ReadOnly, 86
 - seek(), 298, 305, 380
 - Text, 311
 - ungetChar(), 305
 - write(), 304
 - WriteOnly, 83, 299
- QItemDelegate, 281
 - drawDisplay(), 273
 - drawFocus(), 273
- QItemSelectionModel, 338
- QKbdDriverPlugin, 499, 579
- QKeyEvent, 133, 171
- QKeySequence, 50
- QLabel, 3, 41
 - setText(), 56
 - создание стилей с помощью таблиц стилей, 453
- QLatin1String, 433
- QLayout, 16, 145
 - SetFixedSize, 36
 - setMargin(), 147
 - setSizeConstraint(), 36
 - setSpacing(), 147
- QLCDNumber, 41
- QLibrary, 498
- QLinearGradient, 195, 217, 473
- QLineEdit, 41
 - hasAcceptableInput(), 28–29
- setBuddy(), 14
- text(), 65
- textChanged(), 15, 28
- для счетчика, 470
- создание стилей с помощью таблиц стилей, 452, 454
- QLinkedList<T>, 278–278
- QLinkedListIterator<T>, 279
- QList<T>, 278, 618
- QListIterator<T>, 279
- QListView, 39, 243, 250, 324, 454
- QListWidget, 38, 150, 243–244, 332
 - currentRowChanged(), 38, 150–151
 - setCurrentRow(), 151
 - создание подклассов, 232
- QListWidgetItem, 82, 244, 513
 - setData(), 82, 244
 - setIcon(), 244
 - setText(), 244
- QLocale, 433, 435, 670
- QMacStyle, 129, 447, 462, 465
- QMainWindow, 45, 156–159
 - menuBar(), 52
 - restoreState(), 158–159
 - saveState(), 158–159
 - setCentralWidget(), 47
 - setCorner(), 157
 - statusBar(), 55
 - линейки инструментов, 156–159
 - области, 48
 - прикрепляемые окна, 156–159
 - создание подклассов, 45, 209, 565
 - центральный виджет, 47–48, 76–77, 153, 159
- qmake, 4–5, 18, 21, 26, 312, 565, 601–608, 627
 - См. также .pro, файлы
- QMap, 618
- QMap<K,T>, 127, 286–287
- QMapIterator<K,T>, 288
- QMatrix. См. QTransform
- qMax(), 291
- QMdiArea, 77, 160
 - activateNextSubWindow(), 162
 - activeSubWindow(), 163
 - cascadeSubWindows(), 164
 - closeActiveSubWindow(), 164
 - closeAllSubWindows(), 164
 - subWindowActivated(), 161
 - tileSubWindows(), 164
- QMenu, 52
 - addAction(), 52
 - addMenu(), 52
 - addSeparator(), 53
 - exec(), 54
- QMenuBar, 52
- QMessageBox, 42–43, 56–57, 69, 533–534
 - about(), 69
 - critical(), 57
 - information(), 57

- question(), 57
- warning(), 56–57, 590
- QMetaObject, 361, 670
- QMimeData, 233
 - colorData(), 239
 - data(), 235, 239
 - formats(), 235, 238
 - hasFormat(), 230
 - html(), 239
 - imageData(), 239
 - retrieveData(), 235, 239
 - setData(), 235–236
 - setHtml(), 236
 - setText(), 236
 - text(), 235, 239
 - urls(), 231, 239
 - копирование в буфер обмена, 240
 - создание подклассов, 237
- qMin(), 194, 291
- QModelIndex, 251
 - column(), 255, 258
 - internalPointer(), 267
 - parent(), 255
 - row(), 255, 258
- QMotifStyle, 129, 462
- QMouseDriverPlugin, 499, 578
- QMouseEvent, 115, 116, 170, 232, 482
- QMovie, 502
- QMultiHash<K,T>, 287
- QMultiMap<K,T>, 287, 289
- QMutableListIterator<K,T>, 280
- QMutableMapIterator<K,T>, 288
- QMutableStringListIterator, 62
- QMutableVectorIterator<T>, 279
- QMutex, 347, 352
 - lock(), 347, 349
 - tryLock(), 347
 - unlock(), 347, 349
- QMutexLocker, 348, 358
- QNativePointer (Java), 630
- QObject, 21
 - connect(), 6, 8, 15, 20–21, 354, 361
 - deleteLater(), 361, 385
 - disconnect(), 20, 21, 361
 - event(), 171, 177
 - eventFilter(), 176–177
 - findChild<T>(), 38, 537
 - installEventFilter(), 176, 178
 - killTimer(), 175
 - metaObject(), 21
 - moveToThread(), 355
 - property(), 559
 - qt_metacall(), 21
 - sender(), 63, 273, 529
 - setProperty(), 450, 529, 558
 - startTimer(), 175
 - timerEvent(), 175, 181, 558
 - tr(), 15, 21, 203, 425, 429–433, 439, 443
 - в многопоточных приложениях, 361
 - динамическое приведение типов, 63, 657
 - множественное наследование, 203, 560
 - создание подклассов, 22, 567
 - умные указатели, 646
 - хранение в контейнерах, 278
- .qm, файлы, 430, 433, 435, 440, 443
- qobject_cast<T>(), 63, 73, 234, 239, 512, 514, 657
- QPageSetupDialog, 43
- QPaintDevice, 553
- QPaintEngine, 553–554
- QPainter, 182–228
 - boundingRect(), 225
 - drawArc(), 183
 - drawChord(), 183
 - drawCubicBezier(), 183
 - drawEllipse(), 183, 185, 194–196
 - drawLine(), 113, 183, 196
 - drawLines(), 183
 - drawPath(), 183, 515
 - drawPie(), 183, 185
 - drawPixmap(), 128, 183
 - drawPoint(), 183
 - drawPoints(), 183
 - drawPolygon(), 183, 194
 - drawPolyline(), 137, 183
 - drawRect(), 183, 188
 - drawRoundRect(), 183, 195, 208, 475
 - drawText(), 136, 174, 183, 190, 195
 - fillRect(), 114
 - initFrom(), 135, 198
 - paintEngine(), 553
 - restore(), 188, 196, 473
 - rotate(), 190, 195–196
 - save(), 188, 196, 473
 - scale(), 190
 - setBrush(), 184
 - setClipRect(), 137
 - setCompositionMode(), 199
 - setFont(), 184
 - setPen(), 113, 184
 - setRenderHint(), 184, 193, 473, 515
 - setViewport(), 193
 - setWindow(), 189, 193
 - setWorldTransform(), 190
 - shear(), 190
 - SmoothPixmapTransform, 514
 - TextAntialiasing, 515
 - translate(), 190
 - в виджете OpenGL, 478, 485–491
 - в изображении, 198
 - в комбинации с вызовами GDI, 553
 - для печати, 221
 - режимы композиции, 187, 198–199, 468
 - система координат, 113, 187–190
- QPainterPath, 186, 207, 515
- QPair<T1,T2>, 286, 297
- QPalette, 114, 447, 465
- qpe, 580
- QPen, 184

- QPF, 579
- QPictureFormatPlugin, 499
- QPixmap
 - fill(), 135
 - handle(), 554
 - isNull(), 616
 - macCGHandle(), 554
 - macQDAlphaHandle(), 554
 - macQDHandle(), 554
 - x11Info(), 554
 - x11PictureHandle(), 554
 - для двойной буферизации, 123
 - как устройство рисования, 182
- QPlastiqueStyle, 129, 462
- QPluginLoader, 512
- QPoint, 113, 232
- QPointer<T>, 646
- QPointF, 122
- qPrintable(), 295, 299
- QPrintDialog, 43, 221, 226
- QPrintEngine, 554
- QPrinter, 221–228
 - fromPage(), 227
 - newPage(), 221, 223, 227
 - numCopies(), 227
 - PdfFormat, 221
 - setPrintProgram(), 221
 - toPage(), 227
- QProcess, 298, 313–318, 422
 - error(), 315
 - execute(), 318
 - finished(), 315
 - readAllStandardError(), 316
 - readyReadStandardError(), 315
 - start(), 316, 318
 - waitForFinished(), 318, 361
 - waitForStarted(), 318
- QProgressBar, 41, 379
 - minimum и maximum, свойства, 379
 - в качестве индикатора занятости, 379
- QProgressDialog, 42–43, 179–180
 - setRange(), 180
 - setValue(), 181
 - wasCanceled(), 180
 - вызов, 180
- QPushButton, 5, 39
 - clicked(), 6, 170, 314
 - setDefault(), 14
 - setText(), 37
 - toggled(), 35–36
 - создание стилей с помощью QStyle, 463, 469, 473
 - создание стилей с помощью таблиц стилей, 456–357
- QQueue<T>, 278
- QRadialGradient, 194, 486
- QRadioButton, 39
- QReadLocker, 349
- QReadWriteLock, 349
- .qrc, файлы, 49, 125, 313, 434, 436, 588, 602
- qreal, 594
- QRect, 114, 130–131
 - adjusted(), 128
 - contains(), 116, 131
 - normalized(), 128, 131
- QRegExp, 28, 103, 105, 255
- QRegExpValidator, 28, 105
- QRegion, 554
- qRegisterMetaTypeStreamOperators<T>(), 297, 302
- qRgb(), 110
- QRgb, 110
- qRgba(), 110
- QRubberBand, 121
- QScintilla, 165
- QScreen, 499, 578
- QScreenDriverPlugin, 499, 578
- QScriptable, 548–549
- qScriptConnect(), 535
- QScriptContext, 549
- QScriptEngine, 528–530, 534, 546
 - evaluate(), 530, 547
 - hasUncaughtException(), 534
 - newFunction(), 547, 550
 - newQMetaObject(), 546
 - newQObject(), 543
 - setDefaultPrototype(), 546–547
 - setProperty(), 535, 550
 - toScriptValue<T>(), 548
- QScriptExtensionPlugin, 499
- QScriptValue, 530, 536
- qscriptvalue_cast<T>(), 549
- qScriptValueFromSequence(), 546
- QScrollArea, 155–156
 - horizontalScrollBar(), 156
 - setHorizontalScrollBarPolicy(), 156
 - setVerticalScrollBarPolicy(), 156
 - setWidget(), 155
 - setWidgetResizable(), 156
 - verticalScrollBar(), 156
 - viewport(), 155–156
 - образующие виджеты, 156
- QScrollBar, 41, 80, 155
- QSemaphore, 349
 - acquire(), 349, 351
 - release(), 349, 351
- QSessionManager, 571
 - cancel(), 572
 - handle(), 554
 - release(), 572
 - setDiscardCommand(), 570–571
- QSet<K>, 288
- QSettings, 70, 248–249
 - beginGroup(), 70
 - endGroup(), 70
 - setValue(), 69
 - value(), 70
 - поддержка пользовательских типов, 302

- поддержка произвольных типов, 70, 295
- сохранение состояния главного окна, 159
- сохранение состояния разбиения, 154
- QSharedData, 285
- QSharedDataPointer, 285
- QShortcut, 172
- QSignalEmitter (Java), 614
- QSizePolicy, 111, 148
 - Expanding, 124, 148
 - Fixed, 148
 - Ignored, 148
 - Maximum, 148
 - Minimum, 111, 148
 - MinimumExpanding, 149
 - Preferred, 125, 148
 - коэффициенты растяжения, 148–149
- QSlider, 7, 41
 - setRange(), 7
 - setValue(), 7–8
 - valueChanged(), 7–8
- QSoftMenuBar, 589, 591
- qSort(), 95, 290
- QSortFilterProxyModel, 249, 253–255
 - setFilterKeyColumn(), 254
 - setFilterRegExp(), 255
 - setSourceModel(), 254
- QSpinBox, 7, 41, 104
 - setRange(), 7
 - setValue(), 7–8
 - textFromValue(), 106
 - validate(), 105
 - valueChanged(), 7–8
 - valueFromText(), 106
 - создание подклассов, 104–107
 - создание стилей с помощью QStyle, 470–471
- QSplashScreen, 74–75
- QSplitter, 77, 151–155
 - addWidget(), 152
 - KeepSize, 152
 - restoreState(), 155
 - saveState(), 154
 - setSizes(), 154
 - setStretchFactor(), 153
 - sizes(), 282
- QSqlDatabase, 320
 - addDatabase(), 320, 323
 - commit(), 323, 341
 - database(), 323
 - driver(), 323
 - open(), 320
 - rollback(), 323–324, 341
 - setDatabaseName(), 320
 - setHostName(), 320
 - setPassword(), 320
 - setUserName(), 320
 - transaction(), 323–324, 340
- QSqlDriver, 323, 499
- QSqlDriverPlugin, 499
- SQLite, 341
- QSqlQuery, 321, 335
 - addBindValue(), 322
 - bindValue(), 322
 - exec(), 321–322
 - first(), 321
 - isActive(), 321
 - last(), 321
 - next(), 322
 - numRowsAffected(), 322
 - prepare(), 322
 - previous(), 321
 - seek(), 321
 - setForwardOnly(), 321
 - value(), 321
- QSqlQueryModel, 249, 328
- QSqlRecord, 324–326
- QSqlRelation, 328, 332
- QSqlRelationalDelegate, 328, 333, 338
- QSqlRelationalTableModel, 249, 320, 328, 331–333
 - insertRow(), 340
 - setData(), 340
 - setRelation(), 332
- QSqlTableModel, 249, 320, 324, 326–328
 - beforeDelete(), 340
 - beforeInsert(), 340
 - beforeUpdate(), 340
 - fieldIndex(), 332
 - insertRow(), 325
 - record(), 324–326
 - removeRows(), 326
 - select(), 324–325, 327, 339–340
 - setData(), 324–325
 - setFilter(), 324, 339
 - setHeaderData(), 327
 - setSort(), 328
 - setTable(), 324, 327
 - submitAll(), 325
- QSslSocket, 298, 375
- qStableSort(), 93–95, 290
- QStack<T>, 278
- QStackedLayout, 149–151
 - addWidget(), 151
 - indexOf(), 150
 - setCurrentIndex(), 150–151
- QStackedWidget, 38, 150–151
- QStandardItemModel, 249
- QStatusBar, 55
 - addWidget(), 55
 - showMessage(), 59
- QString, 285, 654, 670
 - append(), 291
 - arg(), 61, 273, 292, 432
 - contains(), 90–91
 - endsWith(), 293
 - fromAscii(), 294
 - fromLatin1(), 294
 - indexOf(), 293
 - insert(), 293

- isEmpty(), 294
- left(), 292
- length(), 291, 294
- localeAwareCompare(), 293, 435
- mid(), 65, 292
- number(), 83, 106, 292
- operator+(), 291
- operator+=(), 291
- remove(), 293
- replace(), 237, 293
- right(), 292
- setNum(), 292
- simplified(), 294
- split(), 88, 214, 294, 308
- sprintf(), 291
- startsWith(), 293
- toAscii(), 294
- toDouble(), 99, 292
- toInt(), 65, 106, 292, 308
- toLatin1(), 294
- toLongLong(), 292
- toLower(), 290, 293
- toString(), 299
- toUpper(), 106, 293
- trimmed(), 293
- truncate(), 285
- в Qt Jambi, 614
- поддержка кодировки Unicode, 291, 426–429
- преобразование в `const char *` и обратно, 294
- чувствительность к регистру, 290, 293
- QStringList, 62, 214, 278
 - join(), 294, 308
 - removeAll(), 61
 - takeFirst(), 308
- QStringListModel, 249–250, 253
- QStyle, 129–130, 446–447, 461–477, 499
 - drawComplexControl(), 469
 - drawControl(), 462
 - drawPrimitive(), 129, 468, 475
 - PE_FrameFocusRect, 129
 - pixelMetric(), 467
 - polish(), 465–466
 - standardIcon(), 465
 - standardIconImplementation(), 465, 468
 - standardPixmap(), 468
 - styleHint(), 466
 - subControlRect(), 470
 - unpolish(), 465–466
 - visualRect(), 471
 - сравнение со стилями оформления окон, 579
- QStyledItemDelegate, 281
- QStyleHintReturn, 467
- QStyleOption, 462, 476
 - qstyleoption_cast<T>(), 463, 473, 476
- QStyleOptionButton, 462, 473
- QStyleOptionFocusRect, 129
- QStyleOptionFrame, 476
- QStyleOptionGraphicsItem, 208, 217
- QStylePainter, 129
- QStylePlugin, 499–500
- qSwap(), 263, 290
- QSyntaxHighlighter, 531
- QSysInfo, 554
- Qt Assistant, 10, 41, 421–422
- Qt Designer
 - .jui, файлы, 619, 621
 - .ui, файлы, 25–31, 38, 313, 379, 531–533, 535
 - запуск, 22
 - интеграция с Eclipse, 618–622
 - менеджеры компоновки, 25, 32–33
 - предварительный просмотр, 25, 151, 501
 - применение пользовательских виджетов, 117–120
 - разделители, 154
 - создание главных окон, 45
 - создание диалоговых окон, 22–31
 - шаблоны, 23, 107, 151
- Qt Jambi, 612–630
- Qt Jambi, генератор, 623–630
- Qt Linguist, 425, 442–445
 - <QtAlgorithms> header, 289
 - <QtDebug>, заголовочный файл, 292
 - <QtGlobal>, заголовок, 291
 - <QtGui>, заголовок, 14
- @QtPropertyReader() (Java), 621
- @QtPropertyWriter() (Java), 621
- Qt Quarterly, 10, 360
- Qt Solutions, 9, 43, 305, 392, 551
- Qt, версии, 597–598
- Qt, виртуальный буфер фреймов (qvfb), 576, 578
- Qt, пространство имен, 666
 - AlignHCenter, 56
 - BackgroundColorRole, 255, 269
 - BDiagPattern, 185
 - BevelJoin, 184
 - CopyAction, 233
 - CrossPattern, 185
 - DashDotDotLine, 184
 - DashDotLine, 184
 - DashLine, 184
 - Dense?Pattern, 185
 - DiagCrossPattern, 185
 - DisplayRole, 96–98, 244, 255, 258, 269
 - DotLine, 184
 - EditRole, 96–98, 244, 255, 262
 - escape(), 224, 237, 412
 - FDiagPattern, 185
 - FlatCap, 184
 - FontRole, 255, 270
 - HorPattern, 185
 - IconRole, 244
 - IgnoreAction, 233
 - ItemIsEditable, 262
 - ItemIsEnabled, 262
 - ItemIsSelectable, 262

- LinkAction, 233
- LocalDate, 435
- MiterJoin, 184
- MoveAction, 233
- NoBrush, 185
- NoPen, 184
- OpaqueMode, 187
- RoundCap, 184
- RoundJoin, 184
- ScrollBarAlwaysOn, 156
- SolidLine, 184
- SolidPattern, 185
- SquareCap, 184
- StatusTipRole, 255
- StrongFocus, 125
- TextAlignmentRole, 98, 255, 258, 270
- TextColorRole, 255, 270
- ToolTipRole, 255
- TransparentMode, 187
- UserRole, 244
- VerPattern, 185
- WA_DeleteOnClose, 73, 167, 420, 617
- WA_GroupLeader, 419
- WA_Hover, 466
- WA_PaintOnScreen, 553
- WA_StaticContents, 109, 116
- WhatsThisRole, 255
- QT, элемент (файлы .pro), 310, 326, 367, 400, 405, 484, 530, 604
- Qt/Embedded Linux, 575–594
- Qt_4_3 (QDataStream), 85, 300, 302
- qt_metacall() (QObject), 21
- QT_NO_CAST_FROM_ASCII, 433
- QT_NO_xxx, 578
- QT_PLUGIN_PATH, переменная среды, 501, 629
- QT_TR_NOOP(), 432
- QT_TRANSLATE_NOOP(), 432
- QTableView, 39, 243, 324, 326, 328, 339
- QTableWidget, 77, 243, 246
 - clear(), 80
 - horizontalHeader(), 80
 - horizontalScrollBar(), 80
 - insertRow(), 246
 - item(), 81
 - itemChanged(), 79
 - selectColumn(), 90
 - selectedRanges(), 87
 - selectRow(), 90
 - setColumnCount(), 80
 - setCurrentCell(), 65
 - setHorizontalHeaderLabels(), 246
 - setItem(), 83, 246
 - setItemPrototype(), 79–80, 97
 - setRowCount(), 80
 - setSelectionMode(), 79–80, 237
 - setShowGrid(), 51
 - verticalHeader(), 80
 - verticalScrollBar(), 80
 - viewport(), 80
- виджеты, входящие в, 80–81
- как владелец элементов, 83
- применение технологии «drag and drop», 235
- создание подклассов, 77–79
- QTableWidgetItem, 77, 82, 95
 - clone(), 97
 - data(), 95, 98, 245
 - setData(), 82, 97
 - text(), 81, 95, 97, 245
 - владелец, 82
 - создание подклассов, 95
- QTableWidgetSelectionRange, 67, 238
- QTabWidget, 38, 39
- QtCore, модуль, 14, 296, 310, 603
- QTcpServer, 376, 383
- QTcpSocket, 298, 376, 383
 - canReadLine(), 387
 - close(), 383, 386
 - connected(), 379–380
 - connectToHost(), 379, 389
 - disconnected(), 379, 383, 385
 - error(), 379
 - listen(), 387
 - readLine(), 387
 - readyRead(), 379, 381, 385–387
 - setSocketDescriptor(), 384
 - write(), 380, 386
 - создание подклассов, 383
- QTDIR, переменная среды, 609
- QTemporaryFile, 298, 318, 593
- QTextBrowser, 41, 418, 420
- QTextCodec, 428, 499
 - codecForLocale(), 427
 - codecForName(), 429–430
 - setCodecForCStrings(), 429
 - setCodecForTr(), 429, 443
 - toUnicode(), 429
- QTextCodecPlugin, 499
- QTextDocument, 169, 223, 489
 - contentsChanged(), 167, 169
 - print(), 223
 - setHtml(), 224
 - setModified(), 165
- QTextEdit, 41, 165
 - append(), 317
 - copy(), 240
 - copyAvailable(), 163
 - cut(), 240
 - paste(), 240
 - создание подклассов, 165
- QTextStream, 85, 305, 427
 - AlignXxx, 307
 - FixedNotation, 307
 - ForcePoint, 307
 - ForceSign, 307
 - readAll(), 306
 - readLine(), 306, 308
 - ScientificNotation, 307
 - setAutoDetectUnicode(), 429

- setCodec(), 306, 412, 428–429
- setFieldAlignment(), 307
- setFieldWidth(), 307
- setGenerateByteOrderMark(), 427
- setIntegerBase(), 306, 307
- setNumberFlags(), 306, 307
- setPadChar(), 307
- setRealNumberNotation(), 307
- setRealNumberPrecision(), 307
- ShowBase, 307
- SmartNotation, 307
- UppercaseBase, 307
- UppercaseDigits, 307
 - для сокета, 376
 - для строки, 308
 - для файла, 305
 - кодировки, 305, 310, 427–429
 - синтаксис, 307
- qt-gfx-vnc, опция (configure), 578
- QtGui, модуль, 14, 296, 364, 603
- QThread, 344, 618
 - currentThread(), 353
 - exec(), 361
 - run(), 344, 347–350, 352, 358, 361
 - start(), 346
 - terminate(), 345
 - wait(), 347, 358
 - создание подклассов, 344, 357
- QThreadStorage<T>, 353
- QTicker, 173
- QTime, 435
- QTimeEdit, 41, 270, 273, 435
- QTimeLine, 221
- QTimer, 175
 - 0-миллисекундный, 161, 181
 - setSingleShot(), 192
 - singleShot(), 161
 - start(), 389
 - timeout(), 176, 192, 389
 - однократный, 161, 176, 192
 - сравнение с событиями таймера, 175
- QtMmlWidget, 392
- QtNetwork, модуль, 14, 363–391
- QToolBar, 54, 589
- QToolBox, 39
- QToolButton, 39
- QtOpenGL, модуль, 14, 478–497
- Qtopia Core, 575
- Qtopia PDA, 576
- Qtopia Phone, 576, 579
- Qtopia Platform, 576
- Qtopia, 553, 579–594
- QTOPIA_ADD_APPLICATION(), 584
- QTOPIA_MAIN, 584
- qtopia_project(), директивы (файлы .pro), 582, 584, 587
- QtopiaApplication, 591, 592
- qtopiamake, 581
- QtopiaServiceRequest, 594
- QTransform, 190
- QTranslator, 433, 440–441
- QTreeView, 39, 243, 252, 269
 - expand(), 252
 - scrollTo(), 252
- QTreeWidgetItem, 38, 243, 247–249, 394, 405, 477
- QTreeWidgetItem, 82, 248
 - setData(), 82
 - setText(), 404, 408
 - создание подклассов, 567
- QtScript, модуль, 14, 21, 517, 527–550
- QtService, 387
- QtSql, модуль, 14, 319–342, 362
- qt-sql, опция (configure), 319
- QtSvg, модуль, 14, 392
- QtXml, модуль, 14, 392–412
- QtXmlPatterns, модуль, 392
- QUdpSocket, 298, 387–391
 - bind(), 390, 391
 - pendingDatagramSize(), 391
 - readDatagram(), 391
 - readyRead(), 390
 - writeDatagram(), 389, 391
 - запросы, 321
- queryInterface()
 - QAxAggregated, 562
 - QAxBase, 559
- QueryInterface() (IUnknown), 563
- querySubObject() (QAxBase), 559
- question() (QMessageBox), 57
- queues, 278
- QUiLoader, 38–39, 534
- quintN, 84, 299
- quit() (QCoreApplication), 6, 60
- quitOnLastWindowClosed, (QApplication), 60
- qUncompress(), 305
- QUrl, 231, 364
 - port(), 366
 - setFragment(), 417
 - setScheme(), 416–417
- QUrlInfo, 370
- QValidator, 28–29, 105
- QVariant, 295–297, 530
 - isValid(), 97
 - toString(), 98
 - value<T>(), 536
 - в Qt Jambi, 618
 - для баз данных, 321
 - для механизма «drag and drop», 239
 - для отображений элементов, 244
 - для свойств, 108, 557
 - для элемента «данные» пункта меню, 63, 535
- qVariantCanConvert<T>(), 297
- qVariantFromValue(), 297
- qVariantValue<T>(), 297
- QVarLengthArray<T, Prealloc>, 297
- QVBoxLayout, 8, 145
 - addLayout(), 16

- addStretch(), 148
- addWidget(), 16
- QVector<T>, 277, 281, 618, 628, 652
- QVectorIterator<T>, 279
- qvfb, 576, 578
- qvfb, опция (configure), 578
- QWaitCondition, 352
 - wait(), 353
 - wakeAll(), 353
 - wakeOne(), 358
- QWhatsThis, 415
- QWheelEvent, 134
- QWidget
 - activateWindow(), 64–65, 90
 - addAction(), 54
 - adjustSize(), 125
 - changeEvent(), 441
 - close(), 16, 51, 60
 - closeEvent(), 45, 60, 165, 347, 572, 589
 - contextMenuEvent(), 54
 - dragEnterEvent(), 230, 234
 - dragLeaveEvent(), 231
 - dragMoveEvent(), 231, 234
 - dropEvent(), 230, 234, 237
 - find(), 552
 - focusNextChild(), 177
 - fontMetrics(), 169, 174
 - handle(), 554
 - height(), 113
 - hide(), 126, 148
 - hideEvent(), 175
 - keyPressEvent(), 133, 171, 176
 - keyReleaseEvent(), 171
 - macEvent(), 555
 - minimumSizeHint(), 17, 127, 149
 - mouseDoubleClickEvent(), 482
 - mouseMoveEvent(), 115, 131, 232, 235, 482, 496
 - mousePressEvent(), 115, 130, 193, 232, 234, 482, 496
 - mouseReleaseEvent(), 131, 234, 240, 496
 - paintEvent(), 112, 128, 174, 183, 193, 197, 462, 487, 489–490, 553
 - palette(), 114
 - performDrag(), 236
 - qwsEvent(), 555
 - raise(), 64–65
 - repaint(), 112
 - resize(), 145
 - resizeEvent(), 129, 145
 - scroll(), 175
 - setAcceptDrops(), 230, 232, 237
 - setAttribute(), 73
 - setAutoFillBackground(), 124, 490
 - setBackgroundRole(), 124, 135
 - setContextMenuPolicy(), 54
 - setCursor(), 131
 - setEnabled(), 14, 18
 - setFixedHeight(), 17
 - setFixedSize(), 144
 - setFocus(), 166
 - setFocusPolicy(), 125
 - setGeometry(), 144
 - setLayout(), 7, 8
 - setMinimumSize(), 36, 56, 145
 - setMouseTracking(), 115
 - setSizePolicy(), 109, 111, 124, 558
 - setStyle(), 129, 462
 - setStyleSheet(), 448
 - setTabOrder(), 19
 - setToolTip(), 413
 - setVisible(), 35
 - setWhatsThis(), 415
 - setWindowIcon(), 48
 - setWindowModified(), 56, 167, 165
 - setWindowTitle(), 7, 17, 61, 165
 - show(), 3, 64–65, 126, 148
 - showEvent(), 174
 - sizeHint(), 17–18, 37, 55, 110, 128, 149, 169, 174
 - style(), 129, 462
 - unsetCursor(), 132
 - update(), 91, 111–112, 116, 126, 134, 174–175, 192, 489
 - updateGeometry(), 111, 174
 - wheelEvent(), 133, 221, 489
 - width(), 113
 - windowModified, свойство, 56–57, 61, 167
 - winEvent(), 555
 - winId(), 552–554
 - x11Event(), 555
 - x11Info(), 554
 - x11PictureHandle(), 554
 - в многопоточных приложениях, 361
 - создание подклассов, 107–108, 122, 173, 191, 621
- QWindowsStyle, 129, 462, 464–465
- QWindowsVistaStyle, 129, 462
- QWindowsXPStyle, 129, 462
- QWizard, 42, 43
- QWorkspace. См. QMdiArea
- QWriteLocker, 349
- QWS, 555, 575
- qws, опция (приложения Qt/Embedded Linux), 576
- QWS_DEPTH, переменная среды, 578
- QWS_DISPLAY, переменная среды, 579
- QWS_KEYBOARD, переменная среды, 577, 579
- QWS_MOUSE_PROTO, переменная среды, 577, 579
- QWS_SIZE, переменная среды, 578
- qwsEvent() (QWidget), 555
- qwsEventFilter() (QApplication), 555
- QWSInputMethod, 578
- QWSKeyboardHandler, 499, 578
- QWSMouseHandler, 499, 578
- QWSServer, 579
- qwsSetDecoration() (QApplication), 579

- Qwt, 121
 - QX11Info, 554
 - QXmlContentHandler, 406
 - characters(), 408
 - endDocument(), 405
 - endElement(), 408
 - startDocument(), 405
 - startElement(), 408
 - QXmlDeclHandler, 406
 - QXmlDefaultHandler, 406
 - QXmlDTDHandler, 406
 - QXmlEntityResolver, 406
 - QXmlErrorHandler, 406
 - errorString(), 409
 - fatalError(), 409
 - QXmlInputSource, 407
 - QXmlLexicalHandler, 406
 - QXmlSimpleReader, 406
 - parse(), 407
 - setContentHandler(), 407
 - setErrorHandler(), 407
 - QXmlStreamReader, 392–393
 - atEnd(), 397
 - Characters, 393
 - Comment, 393
 - DTD, 393
 - EndDocument, 393
 - EndElement, 393
 - EntityReference, 393
 - Invalid, 393
 - isCharacter(), 394
 - isStartElement(), 394
 - ProcessingInstruction, 393
 - raiseError(), 396
 - readElementText(), 399
 - readNext(), 393–394, 396
 - setDevice(), 396
 - StartDocument, 393
 - StartElement, 393
 - state(), 394
 - QXmlStreamWriter, 393, 410–411
 - writeEndElement(), 411
 - writeStartDocument(), 410
 - writeStartElement(), 411
 - writeTextElement(), 411
- R**
- raise() (QWidget), 64–65
 - raiseError() (QXmlStreamReader), 396
 - RangeError (ECMAScript), 525
 - Rational PurifyPlus, 638
 - rawCommand() (QFtp), 367
 - rcc, 313, 601–602
 - read()
 - CursorHandler, 504
 - QFtp, 373
 - QHttp, 376
 - readAll()
 - QFtp, 373
 - QHttp, 376
 - QIODevice, 304
 - QTextStream, 305
 - readAllStandardError() (QProcess), 316
 - readBitmap() (CursorHandler), 509
 - readBookindexElement() (XmlStreamReader), 396
 - readClient() (ClientSocket), 385
 - readDatagram() (QUdpSocket), 391
 - readElementText() (QXmlStreamReader), 399
 - readEntryElement() (XmlStreamReader), 398
 - readFile()
 - DomParser, 402
 - SaxHandler, 407
 - Spreadsheet, 85
 - XmlStreamReader, 396
 - readHeaderIfNecessary() (CursorHandler), 508
 - readLine()
 - QIODevice, 387
 - QTextStream, 305, 306
 - readNext() (QXmlStreamReader), 393–394, 396
 - ReadOnly (QIODevice), 86
 - readPageElement() (XmlStreamReader), 399
 - readRawBytes() (QDataStream), 302
 - readSettings()
 - MailClient, 154
 - MainWindow, 70, 159
 - SettingsViewer, 247
 - readyRead()
 - QFtp, 373
 - QHttp, 376
 - QIODevice, 379, 382, 385–387, 390
 - readyReadStandardError() (QProcess), 315
 - reapplyFilter() (ColorNamesDialog), 254
 - recalculate() (Spreadsheet), 91
 - received() (QCopChannel), 577
 - record() (QSqlTableModel), 324–326
 - rect() (QImage), 116
 - ReferenceError (ECMAScript), 520, 525
 - refreshPixmap() (Plotter), 134
 - RegExp (ECMAScript), 525
 - Region (X11), 554
 - registry (Windows), 70
 - regserver, опция (серверы ActiveX), 567
 - regsvr32, 565
 - reinterpret_cast<T>(), 658
 - reject() (QDialog), 29, 64
 - Rejected (QDialog), 29, 65
 - rejected() (QDialogButtonBox), 30
 - RejectRole (QDialogButtonBox), 30
 - release()
 - QGLFramebufferObject, 494
 - QSemaphore, 349–350
 - QSessionManager, 572
 - Release() (IUnknown), 559, 563
 - releaseDC() (QPaintEngine), 553
 - remove()
 - DirectoryViewer, 253
 - QFile, 312
 - QFtp, 366

- QString, 293
 - итераторы в стиле Java, 280
 - removeAll() (QList<T>), 61
 - removeLink() (Node), 206
 - removePostedEvents() (QCoreApplication), 361
 - removeRows() (QAbstractItemModel), 251, 326
 - rename()
 - QDir, 312
 - QFtp, 366
 - render()
 - QGraphicsScene, 223
 - QGraphicsView, 223
 - renderText() (QGLWidget), 484, 488
 - repaint() (QWidget), 112
 - replace() (QString), 237, 293
 - request() (QHttp), 375–376
 - requestFinished() (QHttp), 376
 - requestPropertyChange() (QAxBindable), 562
 - requestStarted() (QHttp), 376
 - reserve() (QHash<K,T>), 287
 - reset() (QAbstractItemModel), 259, 263, 266
 - resize() (QWidget), 145
 - resizeColumnsToContents() (QAbstractItemView), 328
 - resizeEvent() (QWidget), 129, 145
 - resizeGL()
 - QGLWidget, 480, 486, 494
 - Teapots, 494
 - Tetrahedron, 480
 - ResizeTransaction, 359
 - RESOURCES, элемент (файлы .pro), 49, 125, 313, 434, 602
 - restore() (QPainter), 188, 196, 473
 - restoreOverrideCursor() (QApplication), 84, 131
 - restoreState()
 - QMainWindow, 158–159
 - QSplitter, 154
 - TicTacToe, 574
 - retranslateUi()
 - JournalView, 441
 - MainWindow, 438
 - Ui:: classes, 442
 - retrieveData()
 - QMimeData, 235, 239
 - TableMimeData, 239
 - return ключевое слово (ECMAScript), 530
 - reverse, опция (Qt-приложения), 472
 - RGBA-формат, 448, 480
 - См. также ARGB-формат
 - RGB-формат, 110, 197, 448
 - RgnHandle (Mac OS X), 554
 - right() (QString), 292
 - rightColumn() (QTableWidgetSelectionRange), 67
 - rmdir()
 - QDir, 312
 - QFtp, 366
 - rollback() (QSqlDatabase), 323–324, 341
 - ROMAN8, 430
 - rotate()
 - QPainter, 190, 196
 - QTransform, 190
 - RoundCap (Qt), 184
 - RoundJoin (Qt), 184
 - roundness() (Node), 208
 - row() (QModelIndex), 255, 258
 - RowCount (Spreadsheet), 79–80
 - rowCount()
 - BooleanModel, 267
 - CityModel, 258
 - CurrencyModel, 257
 - RTTI, 657
 - RubberBandDrag (QGraphicsView), 210
 - run()
 - QThread, 344, 350, 352, 359, 361
 - Thread, 344, 347
 - TransactionThread, 358
 - runqtopia, 580
 - runScript(), 545–547
- ## S
- SAFEARRAY() (Windows), 557
 - save()
 - Editor, 165
 - MainWindow, 59, 163
 - QDomNode, 410–412
 - QPainter, 188, 196, 473
 - saveAs() (MainWindow), 59
 - saveFile() (MainWindow), 59
 - savefont, опция (приложения Qt/Embedded Linux), 579
 - saveState()
 - QApplication, 569–570
 - QMainWindow, 158–159
 - QSplitter, 154
 - TicTacToe, 573
 - SAX, 392, 405–409
 - SaxHandler
 - characters(), 408
 - endElement(), 408
 - fatalError(), 409
 - inheritance tree, 406
 - readFile(), 407
 - SaxHandler(), 406
 - startElement(), 408
 - определение класса, 406
 - Scalable Vector Graphics (масштабируемая векторная графика), 392
 - scale()
 - QGraphicsView, 220
 - QPainter, 190
 - ScientificNotation (QTextStream), 307
 - Scintilla, 165
 - SCons, 607, 610
 - Screen (X11), 554
 - Screen (режим композиции), 199
 - scriptActionTriggered() (HtmlWindow), 535
 - Scripts, меню, 533

- scroll()
 - PlotSettings, 137
 - QWidget, 175
- ScrollBarAlwaysOn (Qt), 156
- scrollTo() (QTreeView), 252
- SDI, 74, 617
- seek()
 - QIODevice, 298, 304, 380
 - QSqlQuery, 321
- select() (QSqlTableModel), 324–325, 328, 339–340
- SELECT, инструкции (SQL), 321, 324
- selectAll() (QAbstractItemView), 51, 90
- selectColumn() (QAbstractItemView), 89
- selectCurrentColumn() (Spreadsheet), 89
- selectCurrentRow() (Spreadsheet), 89
- selectedId() (FlowChartSymbolPicker), 244
- selectedItems() (QGraphicsScene), 211
- selectedLink() (DiagramWindow), 212
- selectedNode() (DiagramWindow), 211
- selectedNodePair() (DiagramWindow), 212
- selectedRange() (Spreadsheet), 87
- selectedRanges() (QTableWidget), 88
- Selection (QClipboard), 240
- selectRow() (QAbstractItemView), 89
- SelectRows (QAbstractItemView), 338
- send()
 - ExpenseWindow, 592
 - QCopChannel, 577
 - QtopiaServiceRequest, 594
- sendDiagram() (WeatherBalloon), 388
- sender() (QObject), 63, 273, 529
- sendRequest() (TripPlanner), 379
- sendToBack() (DiagramWindow), 211
- separator() (QDir), 316
- session, опция (приложения X11), 571, 573
- sessionFileName() (TicTacToe), 574
- sessionId() (QApplication), 573
- sessionKey() (QApplication), 573
- setAcceptDrops() (QWidget), 230, 232, 237
- setAllowedAreas() (QDockWidget), 158
- setAttribute() (QWidget), 73
- setAutoBufferSwap() (QGLWidget), 490
- setAutoDetectUnicode() (QTextStream), 429
- setAutoFillBackground() (QWidget), 124, 490
- setAutoRecalculate() (Spreadsheet), 92
- setBackgroundRole() (QWidget), 124, 135
- setBit() (QBitArray), 509
- setBrush() (QPainter), 184
- setBuddy() (QLineEdit), 14
- setByteOrder() (QDataStream), 506
- setCentralWidget() (QMainWindow), 47
- setCheckable() (QAction), 51
- setChecked() (QAction), 164
- setCities() (CityModel), 263
- setClipRect() (QPainter), 137
- setCodec() (QTextStream), 306, 412, 428–429
- setCodecForCStrings() (QTextCodec), 429
- setCodecForTr() (QTextCodec), 429, 443
- setColor()
 - AxBouncer, 561
 - Link, 203
- setColorAt() (QGradient), 186, 194–195
- setColumnCount() (QTableView), 80
- setColumnHidden() (QTableView), 328, 339
- setColumnRange() (SortDialog), 36, 67
- setColumnRange(), 36, 67
- setCompositionMode() (QPainter), 199
- setContent() (QDomDocument), 402
- setContentHandler() (QXmlSimpleReader), 407
- setContext() (QShortcut), 172
- setContextMenuPolicy() (QWidget), 54
- setControl() (QAxWidget), 556
- setCorner() (QMainWindow), 157
- setCurrencyMap() (CurrencyModel), 259
- setCurrentCell() (QTableWidget), 65
- setCurrentFile()
 - Editor, 165
 - MainWindow, 60
- setCurrentIndex()
 - QDataWidgetMapper, 334–335
 - QStackedLayout, 149, 151
 - QStackedWidget, 38, 151
- setCurrentInputMethod() (QWSServer), 579
- setCurrentRow() (QListWidget), 151
- setCursor() (QWidget), 131
- setCurveData() (Plotter), 127
- setData()
 - Cell, 97
 - CityModel, 262
 - QAbstractItemModel, 262, 324–325, 340
 - QAction, 536
 - QListWidgetItem, 82, 244
 - QMimeData, 235–236
 - QTableWidgetItem, 82, 97
 - QTreeWidgetItem, 82
- setDatabaseName() (QSqlDatabase), 320
- setDefault() (QPushButton), 14
- setDefaultPrototype() (QScriptEngine), 546–547
- setDevice() (QXmlStreamReader), 396
- setDirty() (Cell), 97
- setDiscardCommand() (QSessionManager), 570–571
- setDropAction() (QDropEvent), 234
- setDuration() (OvenTimer), 192
- setEditorData() (TrackDelegate), 274
- setEditTriggers() (QAbstractItemView), 245–246, 250, 328
- setEnabled()
 - QAction, 164
 - QWidget, 14, 18
- setEnabledOptions() (QPrintDialog), 226
- setErrorHandler() (QXmlSimpleReader), 407
- setFeatures() (QDockWidget), 157
- setFieldAlignment() (QTextStream), 307
- setFieldWidth() (QTextStream), 307

- setFilter() (QSqlTableModel), 324, 339
- setFilterKeyColumn() (QSortFilterProxyModel), 254
- setFilterRegExp() (QSortFilterProxyModel), 255
- setFixedHeight() (QWidget), 17
- setFixedSize() (QLayout), 36
- setFixedSize() (QWidget), 144
- setFlag() (QGraphicsItem), 219
- setFlags() (QGraphicsItem), 203, 205
- setFocus() (QWidget), 166
- setFocusPolicy() (QWidget), 125
- setFont() (QPainter), 184
- setFormat() (QGLWidget), 480
- setFormula()
 - Cell, 97
 - Spreadsheet, 82, 309
- setForwardOnly() (QSqlQuery), 321
- setFragment() (QUrl), 417
- setGenerateByteOrderMark() (QTextStream), 427
- setGeometry() (QWidget), 144
- setHeaderData() (QSqlTableModel), 327
- setHorizontalHeaderLabels() (QTableWidget), 246
- setHorizontalScrollBarPolicy() (QAbstractScrollArea), 156
- setHost() (QHttp), 374–375
- setHostName() (QSqlDatabase), 320
- setHtml()
 - QMimeData, 236
 - QTextDocument, 223
- setIcon() (QListWidgetItem), 244
- setIconImage() (IconEditor), 111
- setImage()
 - QClipboard, 240
 - TransactionThread, 357
- setImagePixel() (IconEditor), 115
- setIntegerBase() (QTextStream), 306, 307
- setInterfaceSafetyOptions() (ObjectSafetyImpl), 563
- setItem() (QTableWidget), 83, 246
- setItemDelegate() (QAbstractItemView), 271
- setItemPrototype() (QTableWidget), 79–80, 97
- setLayout() (QWidget), 7–8
- setLayoutDirection() (QApplication), 434
- setLocalData() (QThreadStorage<T>), 354
- setlocale() (std), 435
- setMargin() (QLayout), 147
- setMimeData() (QClipboard), 240
- setMinimumSize() (QWidget), 36, 56, 145
- setModal() (QDialog), 65, 180
- setModel() (QAbstractItemView), 250, 328
- setModelData() (TrackDelegate), 274
- setModified() (QTextDocument), 165
- setMouseTracking() (QWidget), 115
- setNamedColor() (QColor), 448
- setNum() (QString), 292
- setNumberFlags() (QTextStream), 306, 307
- setOverrideCursor() (QApplication), 84, 131
- setPadChar() (QTextStream), 307
- setPassword() (QSqlDatabase), 320
- setPen() (QPainter), 113, 184
- setPenColor() (IconEditor), 111
- setPixel() (QImage), 116
- setPixmap()
 - QClipboard, 240
 - QDrag, 233
 - QSplashScreen, 74
- setPlotSettings() (Plotter), 125
- setPrintProgram() (QPrinter), 221
- setProperty()
 - QObject, 450, 529, 558
 - QScriptEngine, 535, 550
- setQuery() (QSqlQueryModel), 328
- setRadius() (AxBouncer), 562
- setRange()
 - QAbstractSlider, 7
 - QAbstractSpinBox, 7
 - QProgressDialog, 180
- setRealNumberNotation() (QTextStream), 307
- setRealNumberPrecision() (QTextStream), 307
- setRelation() (QSqlRelationalTableModel), 332
- setRenderHint() (QPainter), 184, 193, 473, 515
- setRootNode() (BooleanModel), 266
- setRowCount() (QTableView), 80
- setScheme() (QUrl), 416–417
- setSelectionBehavior() (QAbstractItemView), 328
- setSelectionMode()
 - QAbstractItemView, 237, 328
 - QTableWidget, 79, 237
- setShortcutContext() (QAction), 172
- setShowGrid() (QTableView), 51
- setSingleShot() (QTimer), 192
- setSizeConstraint() (QLayout), 36
- setSizePolicy() (QWidget), 109, 111, 124, 558
- setSizes() (QSplitter), 154
- setSocketDescriptor() (QAbstractSocket), 384
- setSort() (QSqlTableModel), 328
- setSourceModel() (QAbstractProxyModel), 254
- setSpacing() (QLayout), 147
- setSpeed() (AxBouncer), 562
- setStatusTip() (QAction), 414
- setStretchFactor() (QSplitter), 153
- setStretchLastSection() (QHeaderView), 328
- setStringList() (QStringListModel), 250
- setStyle()
 - QApplication, 462, 477, 501
 - QWidget, 129, 462
- setStyleSheet()
 - QApplication, 447
 - QWidget, 448
- setTable() (QSqlTableModel), 324, 327
- setTabOrder() (QWidget), 19
- setText()
 - Annotation, 219
 - Node, 205
 - QAbstractButton, 38
 - QClipboard, 87, 214, 240

- QLabel, 55
- QListWidgetItem, 244
- QMimeData, 236
- QTreeWidgetItem, 403, 408
- Ticker, 174
- setTextColor() (Node), 206
- SettingsViewer, 247
 - addChildSettings(), 248
 - readSettings(), 248
- setToolTip()
 - QAction, 413
 - QWidget, 413
- setupNode() (DiagramWindow), 210
- setupUi()
 - классы Ui::, 26–28, 315
 - классы Ui_ (Java), 621
- setUser() (QHttp), 375
- setUserName() (QSqlDatabase), 320
- setValue()
 - QAbstractSlider, 7–8
 - QProgressDialog, 180
 - QSettings, 70
 - QSpinBox, 7
 - итераторы в стиле Java, 281, 288
- setVersion() (QDataStream), 85, 299–300, 302–303
- setVerticalScrollBarPolicy() (QAbstractScrollArea), 156
- setViewport()
 - QAbstractScrollArea, 201
 - QPainter, 193
- setVisible()
 - QAction, 51, 62
 - QWidget, 35
- setWhatsThis() (QWidget), 415
- setWidget()
 - QDockWidget, 156
 - QScrollArea, 155
- setWidgetResizable() (QScrollArea), 156
- setWindow() (QPainter), 189, 193
- setWindowIcon() (QWidget), 47
- setWindowModified() (QWidget), 56, 167, 165
- setWindowTitle() (QWidget), 7, 17, 61, 165
- setWorldTransform() (QPainter), 190
- setZoomFactor() (IconEditor), 111
- setZValue()
 - DiagramWindow, 211
 - QGraphicsItem, 203, 211, 219
- Shape (ECMAScript), 524, 527
- shape() (Node), 207
- Shape, 641
- shear() (QPainter), 190
- Shift клавиша, 116, 171
- Shift-JIS, 430
- show() (QWidget), 3, 64–65, 126, 148
- ShowBase (QTextStream), 307
- showbase, манипулятор, 307
- showEvent() (Ticker), 174
- showMessage() (QStatusBar), 59
- showPage()
 - shutdown, 569, 572
- SignalN (Java), 614–614
- simplified() (QString), 294
- SingleSelection (QAbstractItemView), 338
- singleShot() (QTimer), 161
- size() (QFontMetrics), 174
- sizeHint property (QSpacerItem), 33
- sizeHint()
 - Editor, 169
 - IconEditor, 111
 - Plotter, 128
 - QWidget, 17, 37, 55, 110, 128, 149, 169, 174
 - Ticker, 174
- sizeof(), оператор, 650
- sizes() (QSplitter), 282
- skipRawData() (QDataStream), 506
- skipUnknownElement() (XmlStreamReader), 399
- slash (/), 70, 312
- Smalltalk, 241
- SmartNotation (QTextStream), 307
- SmcConn (X11), 554
- SmoothPixmapTransform (QPainter), 515
- SMS, 593–594
- SoftLight (режим композиции), 199
- SolidLine (Qt), 184
- SolidPattern (Qt), 185
- somethingChanged() (Spreadsheet), 83
- sort()
 - Main Window, 66
 - Spreadsheet, 91
- SortDialog, 36
 - использование, 66, 68
 - определение класса, 36
 - создание с помощью Qt Designer, 31–38
- SortedMap (Java), 618
- Source (режим композиции), 199
- source() (QDropEvent), 234
- SourceAtop (режим композиции), 199, 468
- SourceIn (режим композиции), 199
- SourceOut (режим композиции), 199
- SourceOver (режим композиции), 199
- SOURCES, элемент (файлы .pro), 602
- spanX() (PlotSettings), 123
- spanY() (PlotSettings), 123
- SPARC, 638
- spec, опция (qmake), 5, 599, 604
- Spider(), 368
 - done(), 369
 - ftpDone(), 370
 - ftpListInfo(), 369
 - getDirectory(), 368
 - processNextDirectory(), 369
 - определение класса, 368
- split() (QString), 88, 214, 294, 308
- Spreadsheet(), 79
 - autoRecalculate(), 78
 - cell(), 81

- clear(), 80
- ColumnCount, 79–80
- copy(), 87
- currentFormula(), 83
- currentLocation(), 83
- cut(), 87
- del(), 89
- findNext(), 90
- findPrevious(), 90
- formula(), 82
- MagicNumber, 79, 85
- modified(), 79, 83
- paste(), 88
- readFile(), 86
- recalculate(), 91
- RowCount, 79–80
- selectCurrentColumn(), 89
- selectCurrentRow(), 89
- selectedRange(), 87
- setAutoRecalculate(), 92
- setFormula(), 83, 308
- somethingChanged(), 83
- sort(), 92
- text(), 81
- writeFile(), 84, 179, 308
- дерево наследования, 78
- определение класса, 77–79
- SpreadsheetCompare, 67, 79, 93–95
- spreadsheetModified() (MainWindow), 56
- sprintf() (QString), 291
- SQL, 250, 319–342
- SQLite, 319, 326, 329, 597
- square()
 - в C++, 633–636
 - в ECMAScript, 520
- Square, 93
- SquareCap (Qt), 184
- squeeze() (QHash<K,T>), 287
- SSL, 298, 375
- standardIcon() (QStyle), 465
- standardIconImplementation()
 - BronzeStyle, 468
 - QStyle, 465, 467
- StandardKey (QKeySequence), 50
- standardPixmap() (QStyle), 468
- start()
 - AxBouncer, 562
 - QProcess, 316, 318
 - QThread, 346
 - QTimer, 388
- StartDocument (QXmlStreamReader), 393
- startDocument() (QXmlContentHandler), 405
- startDragDistance() (QApplication), 233
- StartElement (QXmlStreamReader), 393
- startElement() (SaxHandler), 408
- startOrStopThreadA() (ThreadDialog), 346
- startOrStopThreadB() (ThreadDialog), 346
- startsWith() (QString), 293
- startTimer() (QObject), 175
- state() (QXmlStreamReader), 394
- stateChanged() (QFtp), 367
- static ключевое слово, 644, 665
- static_cast<T>(), 657
- statusBar() (QMainWindow), 55
- StatusTipRole (Qt), 255
- std, пространство имен, 634, 666, 671
 - cerr, 299, 315–316, 633
 - cin, 299, 310
 - cout, 299, 310, 633
 - endl, 634
 - localeconv(), 435
 - map<K,T>, 288
 - memcpy(), 651
 - ostream, 659
 - pair<T1,T2>, 288, 297
 - setlocale(), 435
 - string, 291, 299
 - strtod(), 633
 - vector<T>, 652
- STL, 276, 284, 671
- stop()
 - AxBouncer, 562
 - Thread, 345, 348
- stopSearch() (TripPlanner), 383
- String (ECMAScript), 519, 525
- String (Java), 614, 618
- string, класс (std), 291, 299
- stringList() (QStringListModel), 251
- strippedName() (MainWindow), 61
- StrongARM, 575
- StrongFocus (Qt), 125
- Stroustrup, Bjarne, 631
- strtod() (std), 633
- struct, ключевое слово, 639
- style()
 - QApplication, 129
 - QWidget, 129, 462
- style, опция (приложения Qt), 9, 161, 501
- StyledPanel (QFrame), 472
- styleHint() (BronzeStyle), 466
- stylesheet, опция (приложения Qt), 447
- subControlRect() (BronzeStyle), 470
- subdirs, шаблон (файлы .pro), 602
- submit() (QDataWidgetMapper), 334
- submitAll() (QSqlTableModel), 325
- subWindowActivated() (QMdiArea), 161
- sum() (ECMAScript), 521–522
- sumOfSquares() (ECMAScript), 520
- super, ключевое слово (Java), 643
- supportsSelection() (QClipboard), 240
- SVG, 48, 392
- swapBuffers() (QGLWidget), 490
- Swing, 612, 617
- switchLanguage() (MainWindow), 440
- SWT, 612, 617
- Sybase Adaptive Server, 319
- SyntaxError (ECMAScript), 525
- system() (QLocale), 433

T

- Tab, клавиша, 125, 171
- TableMimeData(), 238
 - formats(), 238
 - retrieveData(), 238
 - определение класса, 237
- tagName() (QDomElement), 403
- takeFirst() (QList<T>), 308
- TARGET, элемент (файлы .pro), 500, 603, 627
- Tcl/Tk, интеграция в систему, 551
- TCP, 298, 363, 376–387
- TDS, 319
- TeamLeadersDialog(), 250
 - del(), 251
 - insert(), 250
 - leaders(), 251
 - TeamLeadersDialog
- Teapots(), 493
 - Teapots(), 493
 - initializeGL(), 493
 - mouseMoveEvent(), 496
 - mousePressEvent(), 496
 - mouseReleaseEvent(), 496
 - paintGL(), 494–496
 - resizeGL(), 494
 - определение класса, 492
- TEMPLATE, элемент (файлы .pro), 602, 627
- terminate() (QThread), 345
- Tetrahedron(), 480
 - draw(), 482
 - faceAtPosition(), 483
 - initializeGL(), 480
 - mouseDoubleClickEvent(), 482
 - mouseMoveEvent(), 482
 - mousePressEvent(), 482
 - paintGL(), 481
 - resizeGL(), 480
 - определение класса, 479
- Text (QIODevice), 310
- text()
 - QClipboard, 87, 214, 240
 - QDomElement, 404
 - QLineEdit, 65
 - QMimeData, 235, 239
 - QTableWidgetItem, 80, 95, 97, 244
 - QTicker, 173
 - Spreadsheet, 81
- TextAlignmentRole (Qt), 98, 255, 258, 270
- TextAntialiasing (QPainter), 515
- TextArtDialog(), 511
 - loadPlugins(), 512
 - populateListWidget(), 512
- TextArtInterface
 - applyEffect(), 514
 - effects(), 514
 - определение класса, 510
- textChanged() (QLineEdit), 15, 28
- TextColorRole (Qt), 255, 270
- textFromValue() (HexSpinBox), 106
- this, ключевое слово
 - в C++, 658, 660
 - в ECMAScript, 523
- thisObject() (QScriptable), 549
- Thread(), 344
 - run(), 344, 347–349
 - stop(), 345, 348
 - определение класса, 344
- Thread (Java), 618
- ThreadDialog(), 346
 - closeEvent(), 347
 - startOrStopThreadA(), 346
 - startOrStopThreadB(), 346
 - определение класса, 345
- Ticker
 - hideEvent(), 175
 - paintEvent(), 174
 - setText(), 174
 - showEvent(), 175
 - sizeHint(), 174
 - Ticker(), 174
 - timerEvent(), 175
 - определение класса, 173
- Ticker, 175
- TicTacToe(), 573
 - clearBoard(), 573
 - restoreState(), 574
 - saveState(), 573
 - sessionFileName(), 574
 - определение класса, 572
- tidyFile(), 308–310
- TIFF, 48
- tileSubWindows() (QMdiArea), 164
- timeout()
 - OvenTimer, 191
 - QTimer, 176, 192, 389
- timerEvent()
 - PlayerWindow, 558
 - QObject, 175, 181, 558
- TIS-620, 430
- TLS (thread-local storage), 353
- TLS (Transport Layer Security), 298, 375
- to...() (QVariant), 296
- toAscii() (QString), 294
- toBack() (Итераторы в стиле Java), 280
- toCsv() (MyTableWidget), 236
- toDouble() (QString), 99, 292
- toElement() (QDomNode), 403
- toFirst() (QDataWidgetMapper), 334
- toggled()
 - QAbstractButton, 35–36
 - QAction, 51
- toHtml() (MyTableWidget), 236
- toImage() (QGLFramebufferObject), 497
- toInt() (QString), 65, 106, 292, 308
- toLast() (QDataWidgetMapper), 334
- toLatin1()
 - QChar, 427

- QString, 294
 - toLongLong() (QString), 292
 - toLower() (QString), 290, 293
 - toNativeSeparators() (QDir), 312
 - toNext() (QDataWidgetMapper), 334
 - toolTip() (IconEditorPlugin), 119
 - ToolTipRole (Qt), 255
 - top() (QStack<T>), 278
 - toPage() (QPrinter), 227
 - topLevelWidgets() (QApplication), 73
 - toPrevious() (QDataWidgetMapper), 334
 - toScriptValue<T>() (QScriptEngine), 548
 - toString() (QString), 299
 - toString()
 - QDate, 435
 - QDateTime, 435
 - QTime, 435
 - QVariant, 97
 - toUnicode() (QTextCodec), 429
 - toUpper() (QString), 106, 293
 - tp, опция (qmake), 5, 604
 - tr() (QObject), 15, 21, 203, 425, 429, 430–433, 439, 443
 - Track, 270
 - TrackDelegate(), 281
 - commitAndCloseEditor(), 273
 - createEditor(), 273
 - paint(), 281
 - setEditorData(), 274
 - setModelData(), 274
 - определение класса, 281
 - TrackEditor, 271
 - trackNodes() (Link), 202
 - Transaction
 - apply(), 360
 - message(), 360
 - определение класса, 359
 - transaction() (QSqlDatabase), 322–323, 340
 - transactionStarted() (TransactionThread), 359
 - TransactionThread
 - TransactionThread(), 357
 - addTransaction(), 358
 - allTransactionsDone(), 359
 - image(), 358
 - run(), 358
 - setImage(), 358
 - transactionStarted(), 359
 - TransactionThread(), 357
 - определение класса, 357
 - translate()
 - QCoreApplication, 203, 431
 - QPainter, 190
 - QTransform, 190
 - TRANSLATIONS, элемент (файлы .pro), 442
 - TransparentMode (Qt), 187
 - transpose(), 649
 - triggered()
 - QAction, 50
 - QActionGroup, 440
 - trimmed() (QString), 293
 - TripPlanner(), 378
 - closeConnection(), 382
 - connectionClosedByServer(), 383
 - connectToServer(), 378
 - error(), 383
 - sendRequest(), 379
 - stopSearch(), 383
 - updateTableWidget(), 380
 - определение класса, 377
 - TripServer(), 384
 - incomingConnection(), 384
 - определение класса, 384
 - Truck, 643–644
 - TrueType, 579
 - truncate() (QString), 285
 - tryLock() (QMutex), 347
 - TSCII, 430
 - TSD (thread-specific data), 353
 - TTF, 579
 - Type 1, 579
 - type()
 - QEvent, 170–171
 - QVariant, 296
 - typedef, ключевое слово, 656–657
 - TypeError (ECMAScript), 525
 - typeof, оператор (ECMAScript), 519, 523
- ## U
- UCS-2 (UTF-16), 427–428
 - UDP, 298, 363, 387–391
 - Ui, классы (Java), 621
 - Ui::, классы, 26–27, 36, 313, 377
 - Ui_GoToCellDialogClass (Java), 620
 - uic, 26–31, 38, 119, 313, 378, 442, 601, 602
 - undefined (ECMAScript), 519, 520, 525
 - ungetChar() (QIODevice), 305
 - unicode() (QChar), 427
 - Unicode, 291, 305–306, 322, 426–429, 519, 530, 654
 - Unix, 551–555, 599–600
 - unix, условие (файлы .pro), 606
 - unlock()
 - QMutex, 347, 349
 - QReadWriteLock, 349
 - unpolish()
 - BronzeStyle, 466
 - QStyle, 465, 466
 - unregserver, опция (серверы ActiveX), 567
 - unsetCursor() (QWidget), 132
 - unsigned, ключевое слово, 637–637
 - UPDATE, инструкции (SQL), 324
 - updateActions() (DiagramWindow), 215
 - updateEmployeeView() (MainForm), 339
 - updateGeometry() (QWidget), 111, 174
 - updateGL() (QGLWidget), 482, 496
 - updateOutputTextEdit() (ConvertDialog), 316
 - updateRecentFileActions() (MainWindow), 61

updateRubberBandRegion() (Plotter), 134
 updateStatusBar() (MainWindow), 55
 updateTableWidget() (TripPlanner), 380
 updateWindowTitle() (HelpBrowser), 420
 UppercaseBase (QTextStream), 307
 UppercaseDigits (QTextStream), 307
 uppercasedigits, манипулятор, 307
 URI, 231
 URIError (ECMAScript), 525
 URL, 231–232, 364
 urls() (QMimeData), 231, 239
 UserRole (Qt), 244
 using namespace, директивы, 667
 using, объявления, 667
 UTF-16 (UCS-2), 428–429
 UTF-8, 239, 306, 411, 428–429

V

Valgrind, 638
 validate()
 QAbstractSpinBox, 105
 QValidator, 105
 value()
 Cell, 97
 QMap<K,T>, 286–287
 QSettings, 70
 QSqlQuery, 321
 QSqlRecord, 324
 итераторы в стиле Java, 137, 288
 итераторы в стиле STL, 288
 value<T>() (QVariant), 296–297, 536
 valueChanged()
 QAbstractSlider, 7–8
 QSpinBox, 7
 valueFromText() (HexSpinBox), 106
 valueOf() (ECMAScript), 523
 values() (ассоциативные контейнеры), 287–288
 var, ключевое слово (ECMAScript), 518
 VARIANT (Windows), 557
 VARIANT_BOOL (Windows), 557
 vector<T> (std), 652
 VerPattern (Qt), 185
 VERSION, элемент (файлы .pro), 604
 verticalHeader() (QTableView), 80
 verticalScrollBar() (QAbstractScrollArea), 80, 156
 viewport() (QAbstractScrollArea), 80, 155, 156
 Vista, стиль, 9, 129, 462
 Visual Basic, 565
 Visual C++ (MSVC), 5, 19, 38, 297, 554
 Visual Studio, 5, 602, 604
 visualRect() (QStyle), 471
 VNC (Virtual Network Computing), 578
 void, указатели, 82, 267, 657
 volatile, ключевое слово, 344
 VowelCube(), 486
 .vproj, файлы, 604
 -VowelCube(), 487
 createGLObject(), 486

createGradient(), 486
 drawBackground(), 487
 drawCube(), 487–489
 drawLegend(), 489–490
 paintEvent(), 487, 489–490
 wheelEvent(), 489
 определение класса, 484

W

W3C, 400
 WA_DeleteOnClose (Qt), 73, 167, 420, 617
 WA_GroupLeader (Qt), 419
 WA_Hover (Qt), 466
 WA_PaintOnScreen (Qt), 553
 WA_StaticContents (Qt), 109, 116
 wait()
 QThread, 347, 357
 QWaitCondition, 353
 waitForDisconnected() (QAbstractSocket), 361
 waitForFinished() (QProcess), 318, 361
 waitForStarted() (QProcess), 318
 wakeAll() (QWaitCondition), 353
 wakeOne() (QWaitCondition), 358
 warning() (QMessageBox), 56–57, 590
 wasCanceled() (QProgressDialog), 180
 WeatherBalloon, 387–389
 WeatherStation
 processPendingDatagrams(), 390
 WeatherStation(), 390
 определение класса, 389
 web-браузеры, 318, 416–417
 whatsThis() (IconEditorPlugin), 119
 WhatsThisRole (Qt), 255
 wheelEvent()
 CityView, 220
 Plotter, 133
 VowelCube, 489
 WHERE, условие (SQL), 340
 width() (QPaintDevice), 113, 116
 Win32, программный интерфейс, 551, 553
 win32, условие (файлы .pro), 606
 Window (X11), 554
 Window, меню (MDI), 159, 164–165
 windowMenuAction() (Editor), 165
 windowModified, свойство (QWidget), 56–57, 61, 167
 Windows (Microsoft)
 «родные» программные интерфейсы, 552–555
 «спящие» процессы, 569
 CE, 575
 Media Player, 555
 версии, 553
 инсталляция Qt, 598
 кодировки 12xx, 429
 реестр, 70
 WindowsVersion (QSysInfo), 554
 winEvent() (QWidget), 555

winEventFilter() (QApplication), 555
 winId() (QWidget), 552–554
 WINSAMI2, 430
 World WideWeb Consortium, 400
 wrappedFilter() (PumpFilterPrototype), 549
 write() (QIODevice), 304, 380, 386
 writeAttribute() (QXmlStreamWriter), 411
 writeDatagram() (QUdpSocket), 389, 391
 writeEndElement() (QXmlStreamWriter), 411
 writeFile() (Spreadsheet), 84, 179, 308
 writeIndexEntry(), 411
 WriteOnly (QIODevice), 83, 299
 writeRawBytes() (QDataStream), 302
 writeSettings()
 MailClient, 154
 MainWindow, 69, 159
 writeStartDocument() (QXmlStreamWriter), 410
 writeStartElement() (QXmlStreamWriter), 410
 writeTextElement() (QXmlStreamWriter), 411
 writeXml(), 410

X

X Render, расширение, 197
 X Window System (X11)
 выбор буфера обмена, 240
 родные программные интерфейсы, 552–555
 управление сеансами, 568–574
 установка Qt, 599–600
 x11Event() (QWidget), 555
 x11EventFilter() (QApplication), 555
 x11Info()
 QPixmap, 554
 QWidget, 554
 x11PictureHandle()
 QPixmap, 554
 QWidget, 554
 x11Screen() (QCursor), 554
 XBM, файлы 48
 Xcode, 5, 598, 604
 Xevent, тип (X11), 555
 Xlib, 551
 XML
 .qrc, файлы, 49, 313
 .ts, файлы, 443
 .ui, файлы, 31
 для генератора Qt Jambi, 623–625
 запись документов, 409–411
 кодировки, 428
 проверка, 400, 405
 чтение документов, 392–409
 XmlStreamReader
 readBookindexElement(), 396
 readEntryElement(), 398
 readFile(), 396
 readPageElement(), 399
 skipUnknownElement(), 399
 XmlStreamReader(), 396
 определение класса, 395

Xor (режим композиции), 199
 XPath, 392
 -xplatform, опция (configure), 576
 XPM, 48, 48
 XP-стиль, 9, 129, 462
 XQuery, 392
 xsm, 574
 Xt, миграция в, 551

Z

zlib, *См.* сжатие данных
 zoomIn() (Plotter), 127
 zoomOut() (Plotter), 126

A

абсолютное позиционирование, 143–144
 автоматическая генерация полей (SQL), 326
 автоматические соединения, 28, 314, 621
 алгоритмы, 93, 276, 289–291, 297, 672
 альфа-компонент, 110, 197, 197, 448
 анимация, 172, 221
 аннотации
 Java-методов, 621
 в 2D-сценах, 218
 в 3D-сценах, 484, 491
 аппаратное ускорение, 478
 ассоциативные контейнеры, 286–289
 ассоциативные массивы, 286–287
 атомарность, 285, 345, 362
 атрибуты в XML, 398, 404, 408, 408, 411–412

Б

базы данных
 встроенные драйверы, 319, 597
 при обработке, 319
 присваивание значений полям в шаблоне запроса, 322
 просмотр результата запроса, 321, 324
 соединение с, 320, 323–324
 транзакции, 322
 байты, 654
 Безье, кривые, 183, 185
 библиотеки, 498, 603, 636
 битовая глубина изображения. *См.* цветовая глубина изображения
 блокировки чтения/записи, 348–349
 блокоориентированные протоколы, 376, 387
 булевы выражения, 264

В

ввод-вывод
 асинхронный, 313, 317, 363, 373, 379
 блокируемый, 318
 в стандартной библиотеке C++, 670
 двоичных данных, 83–86, 299–305, 376
 подключаемые модули, 501
 поточковый, 84, 299–310, 659, 671
 синхронные операции, 317

- текста, 305–311, 376, 387
устройства, 85, 366, 373–374, 376
«векторные цепочки» (vector paths). *См.* QPainterPath
векторы, 277, 652–653
версия
 Qt, 4, 85, 539, 597–598, 656
 операционной системы, 553
 потока данных, 85, 299, 300–303, 506
 стилевых опций, 476
вертикальная компоновка, 8, 24, 145
взаимоисключающие пункты меню, 51
виджеты, 3
 OpenGL, 478–497
 атрибуты, 73
 верхнего уровня. *См.* окна
 в многопоточных приложениях, 361
 видимые, 4, 64, 126
 восстановление родительских связей, 8, 55, 155
 в стеках, 38, 149, 151
 встроенные, 39–43, 76, 107
 геометрия, 143
 динамические свойства, 21, 450, 535
 для ввода данных, 41
 для просмотра списков элементов, 39, 241–275
 дочерние, 7, 126, 159
 древовидные, 38, 243, 247–249, 394, 404
 изменение размеров, 116, 144–145
 имена, 158, 570
 механизм родословных связей, 7, 28, 616
 многостраничные, 39
 невидимые, 4, 64, 126, 148
 палитры, 113, 124
 политика размера, 111, 124, 148
 пользовательские, 104–139
 прикрепляемые, 156–159
 редактирования данных, 41, 273
 с вкладками, 38–39
 свойства, 21, 24, 108, 449–450, 535, 557, 559, 561–563, 621
 система координат, 113
 скрытые, 4, 64, 126, 148. *См. также* объекты и окна
 списков, 38, 150, 231, 243, 244
 стандартные, 39–43
 стили, 9, 53, 129, 461–477
 табличные, 77, 243, 245–246
 фиксированного размера, 149
 фокусная политика, 125
 фон, 114, 124, 135
 фреймов, 39
 виджет, 47–48, 76–77, 153, 159
виджеты-контейнеры, 4, 39
виртуальные деструкторы, 359, 510
виртуальные машины, 575
виртуальные функции, 359, 510, 642
виртуальный буфер фреймов, 576, 578, 580
включаемые действия, 51
включаемые кнопки. *См.* кнопки-переключатели
включенные виджеты. *См.* отключенные виджеты
вложенные разделители, 152
внедренные ресурсы. *См.* файлы ресурсов
внешние ключи, 323, 329, 331, 332–334, 338
внешние программы, 313, 318
внешняя связь, 664
внеэкранное отображение, 120, 478, 490, 491
восьмеричные числа, 307
всплывающие меню. *См.* меню
всплывающие подсказки, 119, 413
вторичные потоки, 354
выполнение Qt-приложений, 4, 618, 634
выполнение внешних программ, 313, 318
выравнивание, 56, 98, 136, 258, 307
выход из системы, 569, 572
- Г**
генератор (Qt Jambi), 623–630
геометрические фигуры, 183
главный менеджер компоновки, 16, 354
глобальные переменные, 663–665
глобальные функции, 633, 663–665
глобальный объект (ECMAScript), 525, 530
горизонтальная компоновка, 8, 25, 145
градиенты, 186, 195–196, 448, 473
градусы, 134, 185
граничная кромка (в менеджерах компоновки), 147
граничная модель (CSS), 453
графика, 182–228, 478–497
графические представления, классы, 199–221
групповые элементы, 39, 200
- Д**
данные конфигурации. *См.* параметры
дата, 193
двоичная совместимость, 85, 465, 476
двоичные числа, 307
двоичный поиск, 289
действия пользователя, 5, 49, 170, 241
декартова система координат, 113
делегаты, 241, 270, 275, 333
деление на нуль, 102, 112
демоны, 387, 551
деструкторы, 17, 359, 510, 644
диалоговое окно выбора файла, 43, 58–59
диалоговое окно выбора цвета, 42
диалоговое окно вывода на печать, 43, 221
диалоговое окно настройки страницы, 43
диалоговые окна, 41–43
 вызов, 63–69
 вывода сообщения об ошибке, 42–43
 динамические, 38–39

заголовок, 57
изменяющие форму, 31–38
лидер группы, 419
многостраничные, 38
модальность, 63–66, 420
назначение родительского элемента, 57
передача данных между окнами, 67–68
поле кнопок, 29–30
пользовательские, 12–19, 22, 613–618
создание вручную, 13–14
создание при помощи Qt Designer, 22–37, 619
создание при помощи Qt Jambi, 613–616
диалоговый индикатор состояния процесса, 41–43
динамическая память (heap). *См.* new, оператор
динамические библиотеки, 63, 498, 603, 636, 657
динамические меню, 61–63, 164
динамические свойства, 21, 450, 535
динамическое выделение памяти. *См.* new, оператор
документация, 10–11, 417–422, 673
дочерние объекты, 28, 361, 616
дочерние процессы, 313
драйверы
базы данных, 319, 597
клавиатуры, 577–578
принтера, 221
экрана, 578
древовидные представления, 39, 243, 252, 269

З

заголовки (отображение элементов), 246, 252, 259, 262, 269
заголовки окон, 7, 57, 157
заголовочные файлы, 3, 14, 635–636
закрытая секция, 639
запуск внешних программ, 313, 318
заранее определенные модели, 249
защищенная секция, 639
зеркальные компоновки, 8, 178, 434–435, 471
значения, разделяемые запятыми (CSV), 236
значения типа QVariant, 63, 108, 295–297, 321, 530, 557, 618

И

идеальный размер, 17, 37, 55, 110, 125, 145, 148
идентификатор
виджета, 552
запроса HTTP, 376
команды FTP, 366
компонента COM, 557, 565
сеанса X11, 571
таймера, 175
зависимый от платформы, 552
идентификация типов во время выполнения приложения, 657
иерархические модели элементов, 265–269

изменение локализации, события, 440–442
изменение формы курсора, 84, 130
изображения
альфа-компонент, 110, 197
как устройства рисования, 182
печать, 221
пиктограммы, 48–49, 57, 69, 233, 435
распространение с приложением, 48, 582–584
файловая система Qtopia, 580
форматы файлов, 48, 502
цветовая глубина изображения, 110
имитация изобразительных средств (look and feel), 9, 461
индексы моделей, 251
индикаторы занятости, 379
индикаторы состояния процесса, 41, 379
инкрементный синтаксический анализ, 400
исключения
в C++, 99, 636, 671
в ECMAScript, 520
интегрированные среды разработки, 602, 603–604, 618–623
интерактивная система помощи, 413–422
интерактивная справочная документация, 10–11, 673
интернационализация, 425–445
интерпретатор
ECMAScript, 519, 530
Java, 618
интерфейсы
COM, 559, 563
Java и C#, 641
модулей, подключаемых к приложению, 509
информационный бюллетень. *См.* Qt Quarterly
итераторы
const, 281, 282, 284
в стиле Java, 279–281, 288
в стиле STL, 278, 282, 288–289, 671
для чтения и записи, 279–281, 284, 288
только для чтения и для чтения/записи, 279, 281, 283–284
только для чтения, 281, 283–284

К

кадры, буфер (Linux), 576
канва. *См.* классы графических представлений
каркас (фреймворк) (Mac OS X), 604
каталоги, 311–312, 369
кеши, 287, 354
кисти, 113, 184
клавиатурная фокусировка. *См.* фокус клавиши
Alt, 15, 171
Backtab, 171
Ctrl, 116, 171

- Enter, 15, 57, 106, 273
- Esc, 57, 273
- F1, 414, 416
- F2, 246, 252, 340
- Home, 171
- Shift, 116, 171
- Tab, 125, 171
- быстрого выбора пункта меню (QShortcut), 15, 25, 50, 172, 415
- многофункциональные, 587, 589
- пробел, 176
- классы, документация, 10–11, 673
- классы моделей, 249, 257, 320
- классы-контейнеры
 - Java-привязки, 618, 628, 629
 - Qt и STL, 276, 282–284, 672
 - алгоритмы, 289–291
 - ассоциативные массивы, 278
 - ассоциативные массивы, 286
 - в качестве возвращаемых объектов, 282–283
 - векторы, 277, 652
 - итераторы в стиле Java, 279–281, 288
 - массивы байтов, 294
 - массивы битов, 297, 506
 - массивы переменной длины, 297
 - наборы, 288
 - неявное совместное использование, 283, 285, 362, 575, 652
 - оператор цикла foreach, 284–286
 - очереди, 278
 - пары, 297
 - списки, 277–278
 - стеки, 278
 - строки, 291–294
 - хэш-таблицы, 287–288
- классы-шаблоны, 636, 652
- См. также классы-контейнеры
- клиентские процессы (QWS), 576
- кнопки
 - tool, 39
 - включаемые, 31, 51
 - нажимаемые, 5, 15, 24, 29, 39, 456–357, 463
 - панели инструментов, 39
 - переключатели, 31, 39, 51
 - по умолчанию, 15, 24, 57
 - флажки, 39, 450, 467
- кодировки текста, 239, 305–306, 310, 411–412, 426–430
- команда сброса, 571
- команда рестарта, 571
- командная строка, опции
 - configure, 319
 - qmake, 604
 - Qt Jambi-приложений, 617
 - Qt/Embedded Linux-приложений, 576, 578–579
 - Qt-приложений, 3, 9, 161, 634
 - серверов ActiveX, 567–568
 - командная строка, приглашение, 5, 598
 - комментарии в строке состояния, 50, 54, 414
 - компилятор пользовательского интерфейса (uic), 26–31, 38, 119, 378, 442
 - компиляция, 632–634, 664
 - приложений Qt, 4, 601–611, 618, 634
 - компоновка в сетке, 8, 32–33, 145–148
 - компоновщик объектных модулей, 632, 634
 - консольные приложения, 310–311, 318, 363, 603
 - конструкторы
 - C++, 638
 - в ECMAScript, 523, 547
 - копирования, 278–279, 287, 296, 662–663
 - по умолчанию, 277, 287, 296, 639, 641
 - подклассов QObject, 13
 - контекст экрана (OpenGL), 480
 - контекстные меню, 53, 159
 - контролирование событий, 176–179
 - контроллеры (MVC), 241
 - конфликты, выявление, 207
 - координаты рисовальщика, 187–190
 - координаты элемента, 200
 - копирование при записи. См. неявное совместное использование данных
 - коэффициенты растяжения, 55, 148–149, 154
 - кросс-компиляция, 576
 - курсор, эффекты наведения, 357, 451, 458, 466

Л

 - лицензирование 539, 597–598
 - логические координаты, 187–190
 - локализация. См. интернационализация
 - локальная память потока, 353
 - локальные переменные, 645, 665

М

 - макросы, 668–669
 - манипуляторы, 307, 659, 671
 - маркеры флажков, 51, 477
 - массивы байтов (QByteArray), 294
 - массивы битов, 297, 506
 - массивы, 277, 278, 525, 649–654
 - массивы переменной длины, 297
 - мастера, 42–43
 - менеджеры компоновки
 - в Qt Designer, 25, 32–33, 148
 - в Qt Jambi, 615
 - в простом виджете, 76
 - в сравнении с ручной компоновкой, 130, 143–145
 - в стеке, 149–151
 - вложенные, 16, 146–148
 - горизонтальной или вертикальной, 8, 25, 145
 - дочерние, 17, 146–148

идеальные размеры, 17, 37, 55, 110, 125, 145, 148
коэффициенты растяжения, 148–149
кромки и промежутки, 147
политики размера, 110, 124, 148
распорки, 17, 24, 148
сетчатой, 8, 32–33, 145–148
справа налево, 8, 178, 434–435, 472

меню
динамические, 61–63, 164
контекстные, 53, 159
отключение пунктов, 164
помечаемые пункты, 51
создание, 52–53

меню, полосы, 47, 52

«мертвые» блокировки, 358

мерцание, 4, 112

мета объектная система, 21, 546

мета объектный компилятор (moc), 18, 21, 63, 117, 514, 561, 613

метки, 3, 40, 453

метод продвижения (Qt Designer), 117

методы ввода текста, 425, 427, 578

механизмы работы платформы, 9

миграция в MFC, 551

минимальный размер, 37, 56, 145, 149

многозначные ассоциативные массивы, 286

многозначные хэш-таблицы, 287

многоколонные списки, 39, 243, 251, 324, 454

многократная детализация отображаемых данных (drill-down), 330, 342

многопоточная обработка, 343–362

многостраничные панели инструментов, 39

многострочные редакторы. *См.* QTextEdit

многоугольники, 183, 194

многофункциональные клавиши, 587, 589

множественные соединения с базами данных, 324

множественные соединения с базами данных, 74, 77, 159–169

мобильные устройства, 575

модели деревьев, 255

модели прокси, 253

модели списков, 255

модель фильтра, 253

модель/представление, архитектура, 241–243, 255, 275

модель «представление-контроллер» (MVC), 241

модифицированные документы, 56, 61, 165, 167

мутабельные (допускающие запись) итераторы, 279–281, 284, 288

мьютексы, 347, 349, 353, 358

мышка
двойной щелчок, 208, 482
драйверы, 577–578
кнопки, 115–116, 232, 240, 482
колесико, поддержка, 134
курсор, 84, 130, 132
перемещения, 115, 130, 232, 482

событие нажатия кнопки, 115, 130, 170, 193, 232, 482
событие отпускания кнопки, 131, 240

Н

наборы результата, 321, 324

наложения, 121, 491

наследование, 642
См. также создание подкласса
множественное наследование, 27–28, 203, 512, 560, 629, 643

настройки приложения, 69–70, 154, 159, 248

начальный поток, 354

недействительные значения типа QVariant, 98

недействительные индексы моделей, 253, 255, 266

неконстантные итераторы, 279–281, 284, 288

немутабельные (не допускающие запись) итераторы, 281, 283–284

непоименованное соединение с базой данных, 324

непоименованные документы, 166
QWidget, 92, 111–112, 116, 126, 133, 174–175, 192, 489
update(QGraphicsItem, 205, 219

непрозрачность, 110, 197, 448

неразрешенные символы, 19, 636, 640, 644

неупорядоченные ассоциативные контейнеры. *См.* хэш-таблицы

неявное совместное использование данных, 283, 285, 362, 575, 652

нулевые таймеры, 161, 181

нулевые указатели, 645

О

области «прикрепления» окон, 47–48, 156–158

область отображения
рисовальщика, 187–190, 193–194
с прокруткой, 81, 155, 200

область отображения, координаты, 200

обобщенные алгоритмы, 93, 276, 289–291, 297, 672

обработка в фоновом режиме, 161, 181

обратный порядок байтов, 300, 428, 638, 657

общая ошибка защиты, 645

объектные файлы, 632
ObjectSafetyImpl, 563–564
объектные типы, 519, 625

объекты
анализ внутреннего состояния, 21, 613, 670
в ECMAScript, 523–527
в многопоточных приложениях, 361
взаимодействие родительских и дочерних, 28, 616
восстановление родительских связей, 8, 17, 55, 155
динамические свойства, 21, 450, 535

- динамическое приведение типов, 63, 657
 - имена, 158, 449, 570
 - механизм сигналов и слотов, 19–21, 613–615
 - обработка событий, 170–181
 - свойства, 21, 24, 108, 449–450, 535, 557–558, 561–562, 621
 - умные указатели, 646
 - объекты, проверяющие правильность значений, 28, 105
 - объекты-партнеры (buddies), 15, 24
 - объявление и определение, 664
 - обязательные поля, 450
 - ограничение области отображения, 137, 187
 - однодокументный интерфейс (SDI), 74, 617
 - однозначные ассоциативные массивы, 287
 - однократные таймеры, 161, 176, 192
 - окна, 4
 - активные, 64, 113, 161, 163
 - главные, 45–48, 71–74, 76, 156–169
 - дочерние, MDI-интерфейса, 159
 - зависимые от платформы идентификаторы, 552
 - закрытие, 5, 15
 - пиктограмма, 48
 - рисовальщиков, 187–190, 193–194. *См. также* виджеты
 - сообщений, 40–43, 56–57
 - стили оформления, 578
 - строка заголовка, 7, 57, 157
 - оконные менеджеры, 568
 - операторы приведения типов нового стиля, 63, 657–658
 - операторы присваивания, 278–279, 287, 662–663
 - операции с буфером обмена, 87–89, 163, 213–215, 240
 - определение и объявление, 664
 - определение классов (C++), 638–644
 - организация встроенных функций, 639–640
 - основание 16 (шестнадцатеричные), 106, 307
 - основание 2 (двоичные), 307
 - основание 8 (восьмеричные), 307
 - особенности воспроизведения, 515
 - отключенные действия, 164
 - открытая секция, 639
 - отладочный режим, 5, 22, 501, 603
 - отображение виджетов серым цветом, 15
 - отображение на основе элементов, 199–221
 - ошибка сегментации, 645
 - ошибка шины, 645
 - ошибки при линковании, 19, 636, 640, 644
- П**
- пакеты (Mac OS X), 310, 417, 604
 - палитры, 113, 124, 447–448
 - память, выделенная. *См.* new, оператор
 - панели инструментов, 49, 53–54, 156–159, 589
 - панель задач, 58
 - перевод приложений Qt на другие языки, 15, 425, 430–436, 442–445
 - перегрузка операторов, 94, 629, 659–660
 - передний план, 200, 448, 490
 - переключатели редактирования, 244, 246, 250, 329
 - переменные среды
 - CLASSPATH, 616, 618, 625–626, 629
 - DYLD_LIBRARY_PATH, 629
 - LD_LIBRARY_PATH, 629
 - PATH, 4, 422, 571, 600, 629
 - QT_PLUGIN_PATH, 501, 629
 - QTDIR, 609
 - QWS_DEPTH, 578
 - QWS_KEYBOARD, 577, 579
 - QWS_MOUSE_PROTO, 577, 579
 - QWS_SIZE, 578
 - в файлах .pro, 605
 - парсеры XML, не проверяющие достоверность документа, 400, 405
 - парсеры с рекурсивным спуском (recursive-descent parsers), 100, 306, 395, 402, 406
 - перегруженные операторы, 94, 659–660
 - передача событий, 172, 178, 230
 - переключатели, 39
 - перерисовка, 111–112, 116
 - печать, 221–228
 - пиксельные карты, 123, 182
 - пиктограммы, 48, 49, 57, 69, 233, 435
 - платформозависимые программные интерфейсы, 552–574
 - побитовые отображения, 187
 - поверхностное копирование (shallow copy). *См.* неявное совместное использование данных
 - подключаемые модули
 - для Eclipse, 618–623
 - для Qt Designer, 117–119, 622
 - для Qt, 499–510
 - для приложений Qt, 510–515
 - создание при помощи qmake, 604
 - подклассы
 - COM-интерфейсов, 562
 - QAbstractItemModel, 265
 - QAbstractTableModel, 257, 260
 - QApplication, 570
 - QAxAggregated, 562
 - QAxBindable, 559
 - QAxObject, 559
 - QAxWidget, 559
 - QDesignerCustomWidgetInterface, 118
 - QDialog, 13, 26, 36, 243, 613
 - QGLWidget, 479, 484, 492
 - QGraphicsItem, 203, 216–217
 - QGraphicsLineItem, 202
 - QGraphicsView, 220
 - QImageIOHandler, 503
 - QImageIOPlugin, 502

- QItemDelegate, 281
 - QListWidget, 232
 - QMainWindow, 45, 209, 565
 - QMimeData, 237
 - QObject, 20–21, 567
 - QScriptable, 548–549
 - QSpinBox, 104–106
 - QStyle, 129, 461–477
 - QStylePlugin, 499–500
 - QTableWidget, 77–79
 - QTableWidgetItem, 95
 - QTcpServer, 383
 - QTcpSocket, 383
 - QTextEdit, 165
 - QThread, 344, 357
 - QTreeWidgetItem, 567
 - QWidget, 107–109, 122, 173, 191, 621
 - QWindowsStyle, 464
 - QXmlDefaultHandler, 406
 - в C++, 641–643
 - в ECMAScript, 526
 - встроенных виджетов, 104–107
 - интерфейсов подключаемых модулей, 513
 - классов Ui::, 26, 36, 313, 377
 - подменю, 53
 - ползунки, 7, 41
 - полиморфизм
 - в C++, 641–643
 - в ECMAScript, 526
 - политика получения фокуса, 125
 - политики размера, 111, 124, 148
 - полосы прокрутки, 39, 41, 81, 134, 155
 - полупиксельные координаты, 188
 - получение родословной, 8, 17, 55, 154
 - пользовательские делегаты, 270–275
 - пользовательские модели, 255–270
 - пользовательские представления, 275
 - пользовательские свойства, 108, 621
 - пользовательские элементы управления ActiveX, 559
 - полупрозрачность, 110, 197, 448, 468
 - поля в шаблоне запроса (SQL), 322
 - поля с выпадающими списками, 42, 254, 333, 357–461
 - помощники, 42–43
 - порядок байтов, 85, 300, 428, 506, 638, 657
 - порядок перехода по клавише tab, 19, 25, 171
 - последовательные драйверы, 305
 - последовательные контейнеры, 277–286
 - построение Qt приложений, 4, 601–611, 618, 634
 - построитель графического пользовательского интерфейса. *См.* Qt Designer
 - построитель пользовательских интерфейсов. *См.* Qt Designer
 - поток графического интерфейса. *См.* начальный поток
 - поток синхронизация, 347–354
 - поток манипуляторы, 307, 659, 671
 - потокозащищенность, 360
 - предварительно скомпилированные заголовочные файлы, 608
 - предварительные объявления, 13–14
 - предварительный просмотр в Qt Designer, 25, 151
 - предварительный просмотр печати, 221
 - представления типа «master-detail» (главный-подчиненный), 320, 335–337, 342
 - предупреждения (компилятор), 21, 603
 - преобразование кодировок, 306, 411, 427–429, 443
 - препроцессор, 635, 667–670
 - приведение типов, 63, 463, 549, 656–659
 - привязки, генерация, 623–630
 - приложения «клиент-сервер», 376–387
 - присваивание значений полям в шаблоне запроса (SQL), 322
 - «притяжение» виджетов. *См.* WA_StaticContents
 - пробельный символ, 293
 - пробелы (в строках), 293, 308
 - продолжительная обработка, 179, 343
 - продолжительные процессы, 179, 343
 - прозрачность, 110, 197, 448, 468
 - промежутки (в менеджерах компоновки), 147
 - пропорциональное изменение размеров, 144–145
 - просмотр каталогов, 311–312
 - пространства имен
 - C++, 634, 665–667
 - XML, 402, 405, 408
 - протоколы для работы с Интернетом
 - DNS. *См.* QHostInfo
 - FTP, 363–373
 - HTTP, 373–376
 - SSL, 375
 - TCP, 376–387
 - TLS, 375
 - UDP, 387–391
 - прототипы
 - в ECMAScript, 524–527, 546
 - функций C++, 634, 639
 - элементов таблицы, 80
 - прототипы функций, 634, 639
 - прототипы элементов, 80
 - процессы, 313, 577
 - пружины. *См.* разделители
 - прямой порядок байтов, 85, 300, 428, 638, 657
 - прямоугольник интервалов (CSS), 453–455
 - прямоугольник полей (CSS), 453
 - пути. *См.* QPainterPath
- Р**
- работа с сетью, 363–391
 - работа со многими документами, 71–74
 - рабочие пространства. *См.* MDI
 - разделители

- в именах файлов, 312, 316
 - в панелях инструментов, 54
 - в полосах меню, 53
 - между пунктами меню, 53
 - разделители строк, 305, 311
 - разрешающая способность (устройства рисования), 189, 223
 - раскрытия списка, кнопка, 458–460
 - распорки, 16, 24, 33, 148
 - растяжки. *См.* распорки
 - расширяемые диалоговые окна, 31–38
 - реальное копирование, 284–285, 661
 - регулярные выражения, 28, 103, 105, 255, 525
 - редактор форм. *См.* Qt Designer
 - редакторы даты и времени, 41, 435
 - реентерабельность, 361
 - режим выпуска, 5, 501, 603
 - режим передачи (FTP), 366
 - режимы композиций, 187, 198–199, 468
 - резиновые ленты, 121, 130–132, 134, 492, 496
 - реляционные делегаты, 328, 333, 338
 - рисовальщики. *См.* QPainter
 - родительский объект
 - виджета, 7, 29
 - графического элемента, 200
 - диалогового окна, 57
 - объекта, 28, 616
 - объекта, проверяющего правильность значений, 28
 - элемента дерева, 248
 - элемента модели, 255, 268
 - элемента, 82, 403
 - ручная компоновка, 130, 144–145
- С**
- сборка мусора
 - в ECMAScript, 519
 - в Java и C#, 646
 - для приложений Qt Jambi, 615–617
 - связанный буфер, 350
 - свойства объектов
 - COM, 557, 561–563
 - ECMAScript, 523
 - Qt Jambi, 621
 - Qt, 21, 24, 108, 449–450, 535
 - связанные списки, 277–278
 - связь между процессами, 313–318
 - См. также* ActiveX
 - сглаживание линий, 185, 188, 197, 472, 486, 488, 515
 - семафоры, 349–352
 - серверные приложения, 376, 383–387
 - серверный процесс (QWS), 576
 - сжатие данных
 - двоичных, 305
 - событий, 112
 - сигналы и слоты
 - Q_ENUMS(), макрос, 556, 561
 - SIGNAL() и SLOT(), макросы, 6, 20
 - signals и slots, псевдоключевые слова, 13, 20
 - автоматические соединения, 28, 314, 621
 - в Qt Jambi, 613–616
 - в многопоточных приложениях, 354–358
 - в подклассах ActiveX, 559
 - генерация сигналов, 18
 - динамический вызов слотов, 362
 - значения, возвращаемые слотами, 47
 - объявление, 13, 20
 - разъединение, 20
 - реализация слотов, 18, 21
 - соединение, 6, 8, 19–20, 34–36, 614
 - сравнение с событиями, 170
 - типы параметров, 20, 613–614
 - установка соединений в Qt Designer, 34–36
 - символические связи, 370
 - символьные строки, 291–294, 652–654
 - символы шаблона поиска, 58, 253, 311
 - синхронизация потоков, 347–354
 - система координат
 - для графического представления, 200
 - рисовальщика, 113, 187–197
 - системный реестр, 70
 - системы записи, 426
 - скрипты, создание, 517–550
 - словари. *См.* хэш-таблицы
 - слоеные списки (skip-lists), 286
 - слоты
 - Q_ENUMS(), макрос, 556, 561
 - SLOT(), макрос, 6, 20
 - slots псевдоключевое слово, 14, 20
 - автоматические соединения, 28, 314, 621
 - в Qt Jambi, 614
 - в многопоточных приложениях, 354–358
 - в подкласса ActiveX, 559
 - возвращаемые значения, 47
 - динамический вызов, 362
 - объявление, 13, 20
 - разъединение, 20
 - реализация, 18, 21
 - соединение с сигналом, 6, 8, 19–20, 34–36, 614
 - типы параметров, 20, 614
 - установка соединений в Qt Designer, 34–36
 - службы, 387, 551
 - события отсроченного удаления, 361
 - события, 4, 170–181
 - в сравнении с сигналами, 170
 - входа в режим перетаскивания, 230, 233
 - вывода объекта на экран, 174
 - выхода из режима перетаскивания, 231
 - зависимые от платформы, 555
 - закрытия, 45, 60, 165, 347, 572, 589
 - изменения локализации, 440–442
 - изменения направления компоновки, 178
 - изменения размеров, 129, 145
 - изменения языка, 441–442

нажатия клавиши, 133, 171, 175
обработка, 112, 171–175, 177, 555
ожидающие обработки, 181
отпускания клавиши, 171
отпускания объекта, 230, 234
отсроченное удаление, 361
передача, 172, 179, 230
перемещения объекта, 231
рисования, 112, 128, 174, 183, 193, 197, 487, 553
сжатие, 112
синтаксического анализа, 405
скрытия объекта, 175
таймера, 172–176, 181, 558
фильтрация, 176–179, 466, 555
совместно используемые классы, 283, 285, 362, 575, 652
содержимое, прямоугольник (CSS), 453
соединение с базой данных, 320, 322–323
соединение с базой данных, используемое по умолчанию, 323
сокеты. *См.* QTcpSocket
сообщения. *См.* события
соотношение геометрических размеров, 193
сортировка, 66, 93, 252–253, 276, 290, 328
ссылки (C++), 644, 648–649
списки, 278
справочная документация, 10–11, 673
стандартная библиотека C++, 276, 634, 636, 670–673
статическая связь, 664
статические члены, 643–644
стеки, 278
стеки виджетов. *См.* виджеты в стеках
стековая память, 71, 645, 646
стековые компоновки, 149–151
стили, 9, 53, 129, 446–477
 оформления, 578
 пользовательские, 129, 461–477
строки, 291–294, 652–654
строки редактирования, 40, 452, 454
строки состояний, 47, 55–56, 360, 414
строкоориентированные протоколы, 376, 387
структуры данных. *См.* классы-контейнеры
суррогатные пары, 426
сцена, координаты, 200
счетчик ссылок, 285, 362
счетчики (наборные), 7, 41, 104–106, 470–471

Т

таблицы стилей, 39, 446–461
табличные модели, 255
табличные представления, 39, 243, 324, 326
таймеры
 0-миллисекундные, 161, 181
 timerEvent() и QTimer, 175
 однократные, 161, 176, 192
текстовый процессор, 223, 425–426

текстуры, 184, 200, 491, 495
тип передачи (FTP), 366
типы данных, элементарные, 519, 637–637
пользовательские типы данных (в объектах QVariant), 296, 536, 548–549
типы значений, 278–279, 625, 661–663
типы ссылок (Java и C#), 661–663
транзакции (SQL), 322–323, 341
требование к формату окончания строк, 305, 311

У

угловые скобки (<>), 279, 652, 656
углы, 134, 185, 187, 190
удобные подклассы отображения элементов, 243–249
указатели, 644–647, 649–650
универсальная матрица преобразования, 187, 190
универсальные двоичные файлы (Mac OS X), 607
универсальные идентификаторы ресурсов (universal resource identifiers – URI), 231
универсальные указатели информационных ресурсов (URL), 231–232, 364
упорядоченные ассоциативные контейнеры. *См.* ассоциативные массивы
управление памятью, 4, 28, 72–73, 616, 645–646
управление сеансами, 568–574
условия ожидания, 352–354
устройства произвольного доступа, 305
устройства рисования, 182, 221
«утка», правило, 521
уровень детализации, 208, 218
условная компиляция, 668–669

Ф

файлы

XML, чтение и запись, 392–409
атрибуты, 311
временные, 298, 318, 592–593
изображений, форматы, 48
кодировки, 427–429
недавно открывавшиеся, 52, 61–63
просмотр каталогов, 311–312
разделитель в путях доступа, 312, 316
скачивание из сети и загрузка на удаленный компьютер, 363–376
текстовые «ввод-вывод», 305–311, 427–429
фильтры названий, 58, 311

файлы проектов

для qmake, 4, 602–608
для Visual Studio, 5, 604
для Xcode, 599, 604

файлы ресурсов, 48–49, 312–313, 616

для хранения изображений, 48, 125, 415
для хранения транзакций, 434
локализация, 435

сравнение с ресурсами Qtopia, 588
физические координаты устройства, 187–190
фиксация данных, политики, 333
фильтры
 для имен файлов, 58, 311
 для событий, 176–179, 466, 555
 для табличных моделей SQL, 324, 339–340
флажки, 39, 450, 467
фокус, 15, 25, 125, 129, 171, 178, 273
фокусный прямоугольник, 129, 273
фон, 113, 124, 135, 187, 200, 447–448,
 455–456, 489
формат ARGB с «предварительным умноже-
нием», 197
форматированный текст (rich text), 5, 39,
 223, 415
функторы, 93

Х

хосты, адреса, *См.* IP-адреса
хосты, имена, 389
хэш-таблицы, 287–288

Ц

цвет стирания изображения, 124, 135
цветовая глубина изображения, 110
цикл обработки событий, 4, 178, 355, 361,
 363, 366, 373, 385
цикл соединений сигнал-слот, 8, 21
циклический буфер, 349

Ч

чувствительность к регистру, 290, 293, 427

Ш

шаблоны (Qt Designer), 23, 107, 151
 заполнения, 184
 стандартная библиотека, 276, 284, 671
шестнадцатеричные числа, 106, 307
шрифт, 42, 145, 184, 426, 579
 метрика, 136, 169, 174, 219, 514
 раскрывающийся список для выбора, 42
 предварительное отображение, 579

Э

экран, драйверы, 578
экранные заставки, 74–75
экспорт
 подключаемых модулей, 120, 500, 503, 515
 элементов управления ActiveX, 562,
 563–564, 566, 568
элементы (HTML), 224
элементы управления. *См.* виджеты

Я

язык ассемблера, 285, 362
язык описания интерфейсов (IDL), 567
языки, поддерживаемые в Qt, 426
языки с записью справа налево, 8, 434, 435, 472