PRENTICE HALL

Qt 4:

ПРОГРАММИРОВАНИЕ GUI на C++

ЖАСМИН БЛАНШЕТ МАРК САММЕРФИЛД

ВСТУПИТЕЛЬНОЕ СЛОВО МАТТИАСА ЭТТРИЧА

КУДИЦ-ПРЕСС

Жасмин Бланшет

Марк Саммерфилд

Единственное официальное руководстро по практическому программированию в среде Qt 4.1.

Применяя средства разработки Qt компании «Trolltech», вы сможете создавать на C++ промышленные приложения, которые естественно работают в средах Windows, Linux/UNIX, Linux для встроенных систем без изменения программного кода и Mac Os X. Книга написана сотрудниками компании «Trolltech». Она представляет собой практическое руководство по успешному применению самой мощной из всех созданных до сих пор версий Qt — Qt 4.1.

Из книги «Qt 4: программирование GUI на C++» вы узнаете о наиболее эффективных приемах и методах программирования с применением Qt 4 и овладеете ключевыми технологиями в самых различных областях — от архитектуры Qt модель/представление до мощного графического процессора 2D. Авторы вооружают читателей беспрецедентно глубокими знаниями модели событий и системы компоновки Qt.

На реалистических примерах они описывают высокоэффективные методы во всех областях — от разработки основных элементов графического пользовательского интерфейса до передовых методов интеграции с базой данных и XML. Каждая глава содержит полностью обновленный материал.

Данное издание:

• Включает новые главы по архитектуре Qt 4 модель/представление и поддержке подключаемых модулей Qt, а также краткое введение в программирование встроенных систем на платформе Qtopia.

• Раскрывает все основные принципы программирования в среде Qt — от создания диалоговых и других окон до реализации функциональности приложений.

• Знакомит с передовыми методами управления компоновкой виджетов и обработкой событий.

• Показывает, как можно с наибольшей эффективностью использовать новые программные интерфейсы Qt 4, в частности мощный графический процессор 2D и новые простые в применении классы—контейнеры.

• Представляет передовые методы Qt 4, которых нет ни в одной книге: от создания подключаемых модулей, расширяющих возможности Qt, и приложений, до применения «родных» для конкретной платформы программных интерфейсов.

• Содержит приложение с подробным введением в программирование на C++ в среде Qt для опытных Java—разработчиков.

Жасмин Бланшет (Jasmine Blanchette) — менеджер по документированию и старший разработчик компании «Trolltech» с 2001 года. Он является редактором «Qt Quarterly», информационного бюллетеня компании «Trolltech», и соавтором книги «Qt 3: программирование GUI на C++».

Марк Саммерфилд (Mark Summerfield) — независимый преподаватель и консультант по C++, Qt и Python. Он работал менеджером по документированию в компании «Trolltech» на протяжении трех лет. Марк является соавтором книги «Qt 3: программирование GUI на C++».

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 глянцевых маркетинговых материалах компании «Trolltech». Все это очень хорошо, но здесь не указано самое важное: Qt пользуется успехом, потому что она *нравится* программистам.

Почему программистам нравится одна технология и не нравится другая? Сам я считаю, что разработчики программного обеспечения отдают предпочтение такой технологии, которая «ощущается» как правильная, и не любят все то, что не дает такого ощущения. «Ощущать» технологию как правильную означает многое. В версии этой книги для Qt 3 я упоминал телефонную систему компании «Trolltech» в качестве очень подходящего примера особенно плохой технологии. Эта телефонная система не воспринимается как правильная система, потому что она вынуждает нас совершать случайные действия в столь же случайном контексте. Случайность не создает ощущения правильности. Повторяемость и избыточность тоже воспринимаются как неправильные. Хорошие программисты ленивы. Что нас особенно привлекает в компьютерах (например, в сравнении с садоводством), так это то, что нам не приходится повторять одно и то же раз за разом.

Позвольте мне проиллюстрировать это на практическом примере — на формах компенсации командировочных расходов. Обычно эти формы имеют вид причудливых электронных таблиц; вы их заполняете и получаете реальные деньги. На первый взгляд ничего сложного, и при наличии денежного стимула эта задача становится простой для дипломированного инженера.

Однако в реальной жизни все не так просто. Хотя никто другой в компании не испытывает никаких затруднений при работе с этими формами, у инженеров возникают проблемы. И поговорив с сотрудниками других компаний, убеждаешься в том, что это распространенное явление. Мы откладываем оформление компенсаций до самого последнего момента и иногда вообще можем забыть об этом. Почему так происходит? Заполнение форм на первый взгляд простая, стандартная процедура. Собираешь квитанции, нумеруешь и записываешь эти номера в соответствующие поля с указанием даты, места, описания и суммы. Нумерация квитанций и запись номеров в форму предназначены для облегчения кому-то работы, но, строго говоря, номера избыточны, поскольку дата, место, описание и сумма однозначно идентифицируют квитанцию. Можно подумать, что совсем немного дополнительной работы позволяет вернуть свои деньги.

Однако небольшое раздражение вызывают суточные, которые зависят от места вашей поездки. Имеется некий отдельный документ со списком стандартизованных сумм суточных для всех различных пунктов назначения командировок. Нельзя просто указать «Чикаго»; вместо этого приходится самому находить сумму суточных для Чикаго. Аналогичное раздражение вызывает поле обменного курса. Приходится искать текущий обменный курс где-нибудь в системе помощи Google и затем вводить его в каждое поле. Ну, строго говоря, следует подождать, пока компания, обслуживающая вашу кредитную карту, не пришлет вам счет с указанием фактического используемого ею обменного курса. Хотя сделать это

нетрудно, просмотр различных источников и поиск в них нужных данных с последующим их переносом в различные места формы воспринимается как ничем не оправданное неудобство.

Программирование может очень сильно напоминать заполнение наших форм по компенсации командировочных расходов, только здесь все обстоит еще хуже. И здесь на помощь приходит Qt. Qt не такая. Во-первых, Qt логична. 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 XP, Mac OS X и GNU/Linux, и обеспечивает основу для Qtopia — платформы создания приложений для встроенных систем в Linux.

Задолго до того, как инструментарий Qt стал столь популярным и столь широко используемым, нацеленность разработчиков Qt на поиск правильных решений сделала Qt особой. Верность этому принципу столь же сильна сегодня, и она относится к каждому, кто разрабатывает и сопровождает 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.

Маттиас Эттрич (Matthias Ettrich)

Осло, Норвегия

Июнь, 2006г.

Предисловие

Qt представляет собой комплексную рабочую среду, предназначенную для разработки на C++ 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 95 до XP, Mac OS X, Linux, Solaris, HP-UX и во многих других версиях Unix с X11. Библиотеки и утилиты Qt входят также в состав Qtopia Core — программного продукта, обеспечивающего собственную оконную систему для Embedded 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 компакт-диском, который содержит исходный код программ—примеров. Компакт-диск также содержит версию Qt 4.1.1 с открытым исходным кодом для всех поддерживаемых платформ, а также MinGW — набор свободно доступных средств разработки, которые могут использоваться для создания приложений Qt для Windows. В приложении A рассматривается порядок установки программного обеспечения.

Данная книга разделена на три части. В части I раскрыты все 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 передовой материал. Главы частей II и III можно читать в любой последовательности, но они предполагают знакомство с содержанием части I.

Читатели версии этой книги для Qt 3 обнаружат, что новое издание имеет знакомое содержание и знакомый стиль изложения. Данное издание использует новые возможности Qt 4 (причем некоторые из них были введены в версии Qt 4.1), и представленный здесь программный код демонстрирует принципы хорошего программирования с применением средств разработки Qt 4. Во многих случаях здесь используются примеры, аналогичные примерам в издании для Qt 3. Это никак не отразится на новых читателях, но поможет читателям предыдущего издания самостоятельно привыкнуть к более аккуратному, четкому и более выразительному стилю.

Это издание содержит новые главы, в которых описываются архитектура Qt 4 модель/представление, новый фреймворк для подключаемых модулей и основы программирования встроенных систем с помощью Qtopia, а также новое приложение. И так же как в книге для Qt 3, здесь основное внимание уделяется объяснению принципов Qt—программирования, а не просто изложению другими словами и обобщению 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, которые 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 к книге, содержащего введение в C++, вполне достаточного для того, чтобы можно было использовать эту книгу. В качестве более полного введения в объектно—ориентированное программирование на C++ мы рекомендуем книгу *«C++ How to Program» (Как программировать на C++)*, написанную Харви и Полом Дейтелем (Harvey Deitel and Paul Deitel), и *«C++ Primer» (Язык программирования C++. Вводный курс)*, написанную Стенли Б. Липпманом (Stanley B. Lippman), Жози Лажойе (Josie Lajoie) и Барбарой E. My (Barbara E. Moo).

Qt создала 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; если вы собираетесь создавать программы с открытым исходным кодом, вы можете использовать версию с открытым исходным кодом (с лицензией GPL). Qt является основой, на которой построены K Desktop Environment (KDE) и многие другие приложения с открытым исходным кодом.

Кроме сотен классов Qt существуют дополнения, расширяющие рамки и возможности Qt. Некоторые из этих программных продуктов поставляются компанией «Trolltech» — например, модуль сценариев для приложений Qt (QSA — Qt Script for Applications) и компоненты Qt Solutions, в то время как другие подобные программные продукты поставляются другими компаниями и сообществом по разработке приложений с открытым исходным кодом. Информацию по дополнениям Qt можно найти в сети Интернет по адресу http://www.trolltech.com/products/3rdparty/[0]. Qt также имеет хорошо зарекомендовавшее и преуспевающее сообщество пользователей, которое использует список почтовой рассылки 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 мнение, 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 странице http://doc.trolltech.com/qt-book-errata.html.

Благодарности

Прежде всего, мы хотим выразить свою благодарность Айрику Чеймб-Ингу (Eirik Chambe-Eng), президенту компании «Trolltech». Айрик 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), ведущим разработчиком программного обеспечения в компании «Trolltech» и нашим шефом. Маттиас снисходительно относился к игнорированию нами наших обязанностей, когда мы были полностью вовлечены в процесс написания первого издания 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 деталей 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 давать нам свои ценные советы, и нашими важными рецензентами из компании «Trolltech» были Эндриас Аардал Хансен (Andreas Aardal Hanssen), Хенрик Харц (Henrik Hartz), Виви Глукстад Карлсен (Vivi Gluckstad Karlsen), Трентон Шультц (Trenton Schultz), Энди Шоу (Andy Shaw) и Пал де Вибе (Pel de Vibe).

Кроме упомянутых выше рецензентов мы получали экспертную помощь от Харальда Ферненгела (Harald Fernengel) (базы данных), Волкера Хилшаймера (Volker Hilsheimer) (ActiveX), Бредли Хьюза (Bradley Hughes) (многопоточная обработка), Тронда Кьернесена (Trond Kjernesen) (графика 3D и базы данных), Ларса Кнолла (Lars Knoll) (графика 2D и интернационализация), Сэма Магнусона (Sam Magnuson) (qmake), Мариуса Бугге Монсена (Marius Bugge Monsen) (классы отображения элементов), Димитри Пападопулоса (Dimitri Papadopoulos) (Qt/X11), Пола Олава Твита (Paul Olav Tvete) (пользовательские виджеты и программирование встроенных систем), Рейнера Шмида (Rainer Schmid) (работа с сетью и XML), Амрит Пол Синх (Amrit Pal Singh) (введение в C++) и Гуннара Слетта (Gunnar Sletta) (2D-графика и обработка событий).

Дополнительную благодарность мы выражаем группам подготовки документации и технической поддержки компании «Trolltech» за помощь в решении вопросов, связанных с подготовкой документации, так как книга отнимала у нас столь много времени, и системным администраторам компании «Trolltech» за обеспечение рабочего состояния наших машин и наших сетевых соединений во время работы над проектом.

Что касается производственной части, то Трентон Шультц создал сопроводительный компакт-диск, а Катрин Бор (Cathrine Bore) из «Trolltech» вела для нас контракты и

обеспечивала юридические вопросы. Мы также благодарны Натан Клемент (Nathan Clement) за иллюстрации с троллями. И наконец, мы выражаем нашу благодарность Ларе Уисонг (Lara Wysong) из компании «Pearsons» за очень хорошее управление процессом производства.

Краткая история Qt

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

Хаарвард 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.) Хаарвард 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 летом Хаарвард и Айрик вышли на улицу, чтобы понежиться на солнышке, и когда они присели на скамейку в парке, Хаарвард 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 Хаарвард 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. Хаарвард воспринял эту идею и вручную реализовал ее. К 1993 году Хаарвард и Айрик разработали первое графического ядро Qt и могли создавать свои собственные виджеты. В конце этого года Хаарвард 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 Хаарвард, 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 Хаарвард. 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 Хаарварда, 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 Guldbransen), который в течение своих шести лет работы 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, и 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, которое приобрело десять коммерческих лицензий. Верящие в удачу Айрик и Хаарвард 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 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 была выпущена 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 была выпущена в июне 1999 года. Qt 2 имела новую лицензию для открытого исходного кода — Q Public License (QPL), которая соответствовала Определению открытого исходного кода (Open Source Definition). В августе 1999 года Qt выиграла 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 Qtopia Core (получившую затем название Qt/Embedded) 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 Qtopia Core предлагаются теперь по широко распространенной общедоступной лицензии GNU — General Public License (GPL), а также на условиях коммерческих лицензий. К концу 2000 «Trolltech» учредила компанию «Trolltech Inc.» (США) и выпустила первую версию Qtopia — платформу для разработки приложений для мобильных телефонов и карманных компьютеров. Qtopia Core был удостоен премии журнала «Linux World» в категории «Лучшее решение для системы Embedded Linux» в 2001 и 2002 годах, а Qtopia Phone получила ту же премию в 2004 году.

Qt 3.0 была выпущена в 2001 году. Qt теперь работала в системах Windows, Mac OS X, Unix и Linux (для настольных и встроенных систем). Qt 3 содержала 42 новых класса, и объем ее программного кода превышал 500 000 строк. Qt 3 представляла собой важный шаг вперед по сравнению с Qt 2, которая, в частности, значительно улучшила поддержку локализации и кодировки *Unicode,* ввела совершенно новые виджеты по просмотру и редактированию текста и класс регулярных выражений, аналогичных применяемым языкам Perl. Qt 3 была удостоена премии «Software Development Times» в категории «Высокая продуктивность» в 2002 году.

Летом 2005 года была выпушена Qt 4.0. Имея около 500 классов и более 9000 функций, Qt 4 оказалась больше и богаче любой предыдущей версии; она была разбита 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/представление, 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. Qt 4 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 компонента Qtopia.

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 последнюю декаду Qt превратилась 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

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 мы 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 Б, 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.

```
01 #include <QApplication>
02 #include <QLabel>
03 int main(int argc, char *argv[])
04 {
05 QApplication app(argc, argv);
06 QLabel *label = new QLabel("Hello Qt!");
07 label->show();
08 return app.exec();
09 }
```

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 *виджетом (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 гибка, 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 мерцания экрана.

Строка 8 обеспечивает передачу управления приложением Qt. В этом месте программа переходит в цикл обработки событий, т.е. в своего рода режим «простоя», ожидая со стороны пользователя таких действий, как щелчок мышки 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.

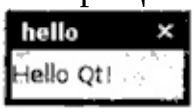


Рис. 1.1. Вывод приветствия программы Hello в системе Linux

Теперь вы можете проверить работу этой программы на своей машине. Сначала

необходимо установить Qt 4.1.1 (или более позднюю версию Qt 4); процесс установки рассмотрен в Приложении А. С этого момента мы будем предполагать, что вы корректно установили библиотеку Qt 4 и ее каталог *bin* занесен в переменную окружения PATH. (В системе Windows это делается автоматически программой установки Qt.) Вам также потребуется поместить файл *hello.cpp* с исходным кодом программы Hello в каталог *hello*. Вы можете набрать файл *hello.cpp* вручную или взять его с компакт-диска, который входит в состав книги; на компакт-диске этот исходный код находится в файле */examples/chap01/hello/hello.cpp*.

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 зависимого от платформы файла *makefile*.

Выполните команду *make* для построения программы [1]. Затем выполняйте программу, задавая команду *hello* в системе Windows или *./hello* в системе Unix и *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 DOS, в котором переменные среды правильно настроены на Qt. Вызвав это окно, вы можете компилировать в нем Qt—приложения, используя описанныевыше команды *qmake* и *make*. Формируемые исполнительные модули помещаются в папку *debug* или *release*, например, *C:\qt-book\hello\release\hello.exe*.

Если вы используете Visual C++ компании Microsoft, 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

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. 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).

Исходный код этого приложения находится на компакт-диске в файле */examples/chap01/quit/quit.cpp*. Ниже приводится содержимое этого файла:

```
01 #include <QAapplication>
02 #include <QPushButton.h>
03 int main(int argc, char *argv[])
04 {
05 QApplication app(argc, argv);
06 QPushButton *button = new QPushButton("Quit");
07 QObject::connect(button, SIGNAL(clicked()),
08 &app, SL0T(quit()));
09 button->show();
10 return app.exec();
11 }
```

Виджеты Qt генерируют *сигналы*[2] 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; более подробно они объясняются в следующей главе.

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, 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 предлагает пользователю указать свой возраст, что можно сделать при помощи либо наборного счетчика (spin box), либо ползунка (slider).

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:

```
01 #include <QApplication>
02 #include <QHBoxLayout>
03 #include <QSlider>
04 #include <QSpinbox>
05 int main(int argc, char *argv[])
06 {
07 QApplication app(argc, argv);
08 QWidget *window = new QWidget;
09 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.

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 родительским 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 по 17, 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 ползунок 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.

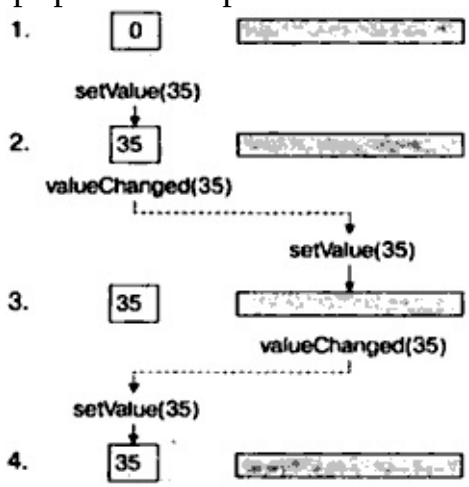


Рис. 1.5. Изменение значения в одном из виджетов приводит к изменению значения в другом виджете.

В строках с 19 по 22 мы размещаем виджеты наборного счетчика и ползунка, используя менеджер компоновки. Менеджер компоновки — это объект, который устанавливает размер и положение виджетов, которые располагаются 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.

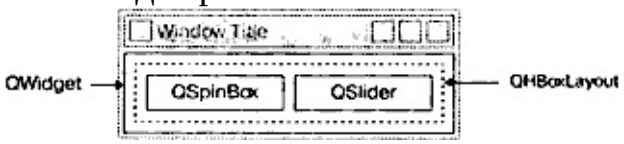


Рис. 1.6. Виджеты приложения Age.

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.

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, а затем соответствующим образом настраиваются их свойства. Программисты добавляют виджеты к компоновщикам графических элементов, которые автоматически устанавливают для них нужные 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 документации 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 и поэтому 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.

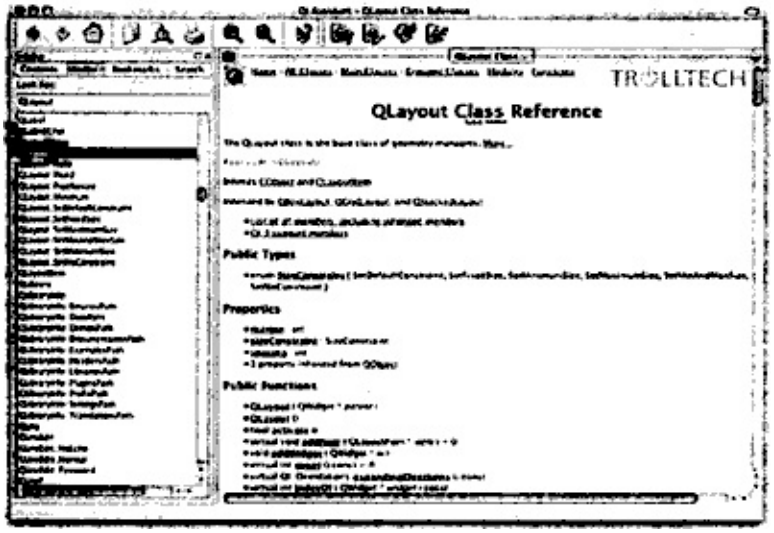


Рис. 1.7. Просмотр документации Qt программой Qt Assistant в системе Mac OS X.

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 родительском классе *QWidget*. На рис. 1.8 показана взаимосвязь классов, которые использовались в этой главе.

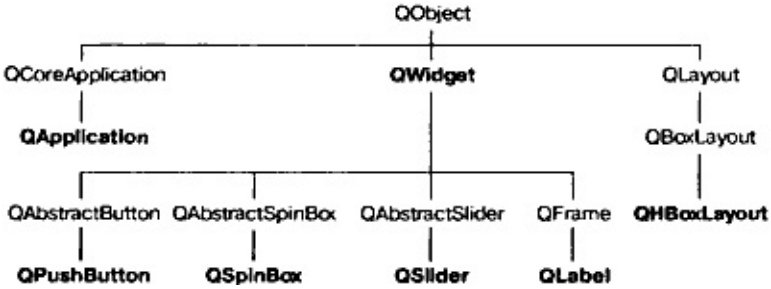


Рис. 1.8. Дерево наследования для классов, используемых в данной главе.

Справочная документация 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 предназначается для программистов Qt и распространяется по всем коммерческим лицензиям.

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.

Стили виджетов

Показанные нами ранее экраны были взяты из системы Linux, но приложения Qt будут выглядеть привычно для любой поддерживаемой платформы. Qt имитирует изобразительные средства используемой платформы, а не делает попытки все представить средствами, принятыми в какой-то одной платформе или каким-то одним инструментарием.

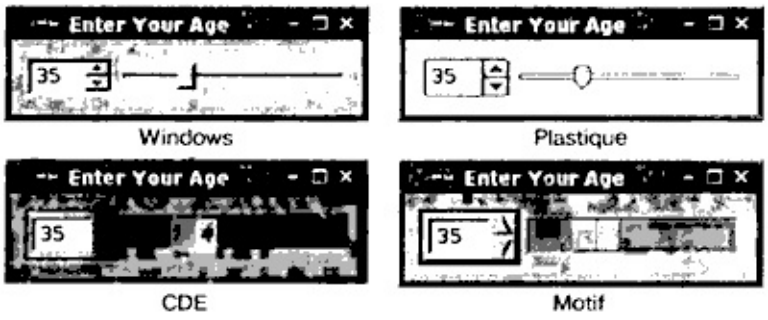


Рис. 1.9. Различные стили вывода графических элементов.

В Qt/X11 и Qtopia Core по умолчанию используется стиль Plastique, который применяет плавные переходы цветов и подавление помех спектрального наложения для обеспечения современного интерфейса пользователя. Пользователи приложений Qt могут переопределять принятый по умолчанию стиль, используя опцию *—style* в команде запуска приложения. Например, для запуска приложения Age со стилем Motif в X11 необходимо просто задать команду

./age -style motif

в командной строке.

Рис. 1.10. Зависимые от платформы стили.

В отличие от других, стили систем Windows XP и Mac доступны только на «родных» платформах, поскольку они реализованы на базе присущих только данной платформе механизмов работы.

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 построении 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) для поиска заданной пользователем последовательности символов, и оно будет полностью написано на C++. Мы реализуем это диалоговое окно в виде его собственного класса. Причем мы сделаем его независимым и самодостаточным компонентом, со своими сигналами и слотами.

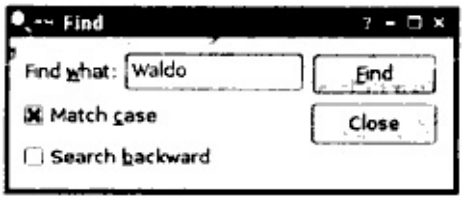


Рис. 2.1. Диалоговое окно поиска.

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

```
01 #ifndef FINDDIALOG_H
02 #define FINDDIALOG_H
03 #include <QDialog.h>
04 class QCheckBox;
05 class QLabel;
06 class QLineEdit;
07 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* наследует свойства класса *QWidget*.

В строках с 4 по 7 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.

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

```
08 class FindDialog : public QDialog
09 {
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, указывающий 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 так далее), но при использовании предварительных объявлений компилятор работает немного быстрее.

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

```
01 #include <QtGui>
02 #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 *filedialog.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.

```
03 FindDialog::FindDialog(QWidget *parent)
04 : QDialog(parent)
05 {
06 label = new QLabel(tr("Find &what:"));
07 lineEdit = new QLineEdit;
08 label->setBuddy(lineEdit);
09 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 17.

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 мы 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)*[3]. Кнопка, для которой задан режим использования по умолчанию, будет срабатывать при нажатии пользователем клавиши Enter (ввод). 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()));
```

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.

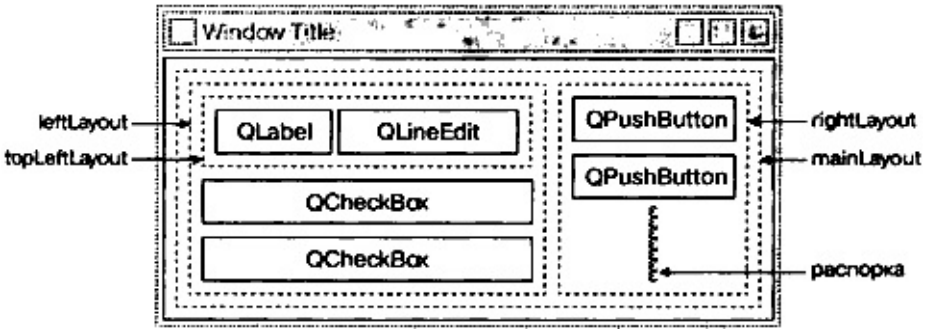


Рис. 2.2. Менеджеры компоновки диалогового окна поиска данных.

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 ответственен 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 наследуют свойства класса *QLayout*, который, в свою очередь, является наследником 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.

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.

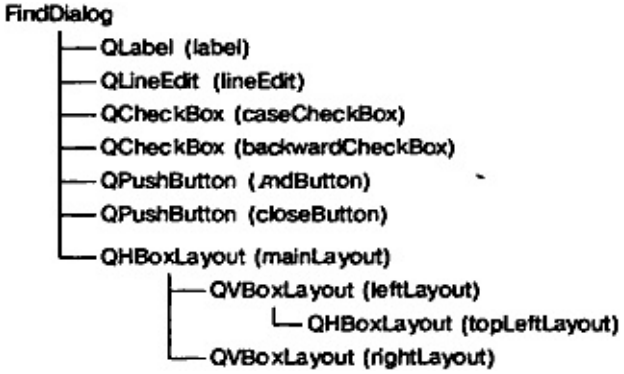


Рис. 2.3. Родословная объектов диалогового окна поиска данных.

```
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:*

```
01 #include <QApplication>
02 #include "finddialog.h"
03 int main(int argc, char *argv[])
04 {
05 QApplication app(argc, argv);
06 FindDialog *dialog = new FindDialog;
07 dialog->show();
08 return app.exec();
09 }
```

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* — мета—объектного компилятора Qt. (Мета—объектная система Qt рассматривается 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 сгенерированные инструкции 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 следующего вида:

```
finddialog.o(.text+0x28): undefined reference to
'FindDialog::QPaintDevice virtual table'
(не определена ссылка на «виртуальную таблицу
FindDialog::QPaintDevice»)
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 LNK2001: 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, они 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 неопределен.

• 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 &)));

• 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 либо 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.

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 (для *Qt Designer)* и перевод текстовых значений (для интернационализации приложений), а также создает основу для системы сценариев в Qt (Qt Script for Applications — QSA).

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++, мета—объектная 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.*

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:*

```
01 class Employee : public QObject
02 {
03 Q_OBJECT
04 public:
05 Employee() { mySalary = 0; }
06 int salary() const { return mySalary; }
07 public slots:
08 void setSalary(int newSalary);
09 signals:
10 void salaryChanged(int newSalary);
11 private:
12 int mySalary;
13 };

14 void Employee::setSalary(int newSalary)
15 {
16 if (newSalary != mySalary) {
17 mySalary = newSalary;
18 emit salaryChanged(mySalary);
19 }
20 }
```

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 (см. рис. 2.4), которое управляет переходом на заданную ячейку таблицы (Go-to-Cell dialog). Создание диалогового окна как при ручном кодирования, так и при использовании *Qt Designer* предусматривает выполнение следующих шагов:

- создание и инициализация дочерних виджетов;
- размещение дочерних виджетов в менеджерах компоновки;
- определение последовательности переходов по клавише табуляции;
- установка соединений «сигнал — слот»;
- реализация пользовательских слотов диалогового окна.

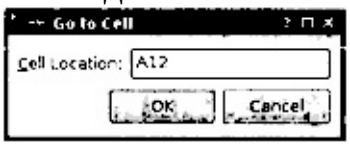


Рис. 2.4. Диалоговое окно для перехода на заданную ячейку таблицы.

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 OK. (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».

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, выберите функцию Edit | User Interface Mode | 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 *QtDesigner*

показан в виде синей пружинки.

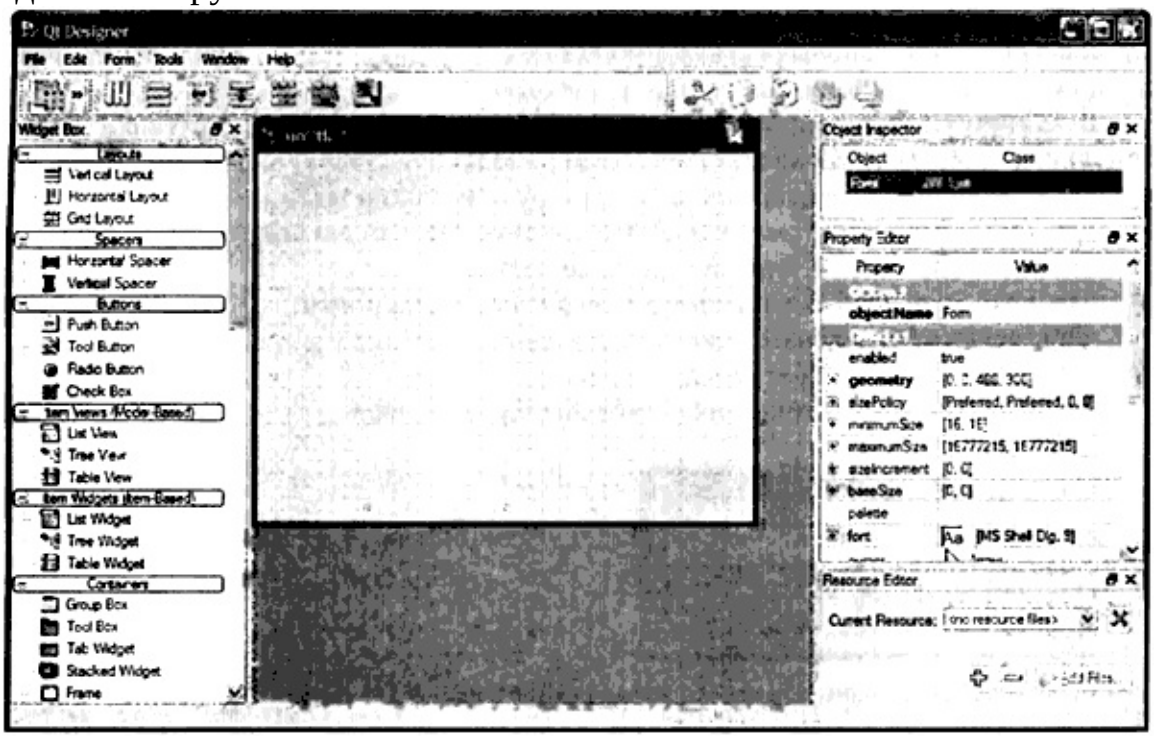


Рис. 2.5. Qt Designer в режиме пристыкованного окна в системе Windows.

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.

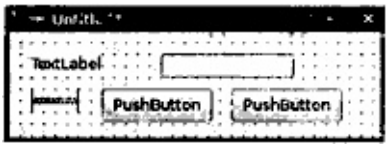


Рис. 2.6. Форма с несколькими виджетами.

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 «OK» (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 и перенесите красную

стрелку на строку редактирования, а затем отпустите кнопку мышки. Теперь эта метка будет выглядеть как Cell Location и иметь строку редактирования в качестве партнера. Выберите Click Edit | Edit Widgets (Правка | Редактировать виджеты) для выхода из режима установки партнеров.

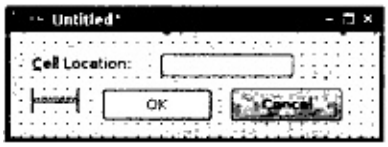


Рис. 2.7. Вид формы после установки свойств виджетов.

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).

2. Щелкните по растяжке, затем, удерживая клавишу Shift, щелкните по клавишам OK и Cancel. Выберите в меню Form | Lay Out Horizontally.

3. Щелкните по свободному месту формы, аннулируя выбор любых виджетов, затем выберите в меню функцию Form | Lay Out Vertically (Форма | Вертикальная компоновка).

4. Выберите в меню функцию Form | Adjust Size для установки предпочитаемого размера формы.

Красными линиями на форме обозначаются созданные менеджеры компоновки. 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. 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.

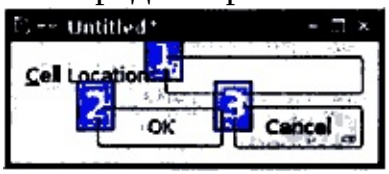


Рис. 2.9. Установка последовательности перевода фокуса по виджетам формы.

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.

```
01 #include <QApplication>
02 #include <QDialog>
03 #include "ui_gotocelldialog.h"
```

```
04 int main(int argc, char *argv[])
05 {
06 QApplication app(argc, argv);
07 Ui::GoToCellDialog ui;
08 QDialog *dialog = new QDialog;
09 ui.setupUi(dialog);
10 dialog->show();
11 return app.exec();
12 }
```

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 наследует никакой Qt—класс. 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 OK 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 наследует *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:

```
01 #ifndef GOTOCELLDIALOG_H
02 #define GOTOCELLDIALOG_H
03 #include <QDialog>
04 #include "ui_gotocelldialog.h"
05 class GoToCellDialog : public QDialog, public Ui::GoToCellDialog
06 {
07 Q_OBJECT
08 public:
09 GoToCellDialog(QWidget *parent = 0);
10 private slots:
11 void on_lineEdit_textChanged();
12 };
13 #endif
```

Реализация методов класса делается в файле *gotocelldialog.cpp:*

```
01 #include <QtGui>
02 #include "gotocelldialog.h"

03 GoToCellDialog::GoToCellDialog(QWidget *parent)
04 : QDialog(parent)
05 {
06 setupUi(this);
07 QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
08 lineEdit->setValidator(new QRegExpValidator(regExp, this));
09 connect(okButton, SIGNAL(clicked()),
10 this, SLOT(accept()));
11 connect(cancelButton, SIGNAL(clicked()),
12 this, SLOT(reject()));
13 }

14 void GoToCellDialog::on_lineEdit_textChanged()
15 {
16 okButton->setEnabled(lineEdit->hasAcceptableInput());
17 }
```

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, функцию 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 OK 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 значение *QDialog::Rejected* (которое равно 0). При использовании этого диалога мы можем использовать значение результата, чтобы узнать, была ли нажата кнопка OK, и действовать соответствующим образом.

Слот *on_lineEdit_textChanged()* устанавливает кнопку OK 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:

```
01 #include <QApplication>
02 #include "gotocelldialog.h"
03 int main(int argc, char *argv[])
04 {
05 QApplication app(argc, argv);
```

```
06 GoToCellDialog *dialog = new GoToCellDialog;
07 dialog->show();
08 return app.exec();
09 }
```

Постройте еще раз приложение (*qmake —project; qmake gotocell.pro*) и выполните его. Наберите в строке редактирования значение «A12» и обратите внимание на то, как кнопка OK становится активной. Попытайтесь ввести какой-нибудь произвольный текст и посмотрите, как сработает функция по проверке допустимости значения. Нажмите кнопку Cancel для закрытия диалогового окна.

Привлекательной особенностью применения *Qt Designer* является возможность для программиста 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 тратиться много времени, поскольку *uic* просто заново генерирует исходный код программы 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 *uic* 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 (area 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 представлениями этого диалогового окна. Расширяемые диалоговые окна обычно применяются в тех приложениях, которые предназначаются как для неопытных, так и для 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. 2.10.

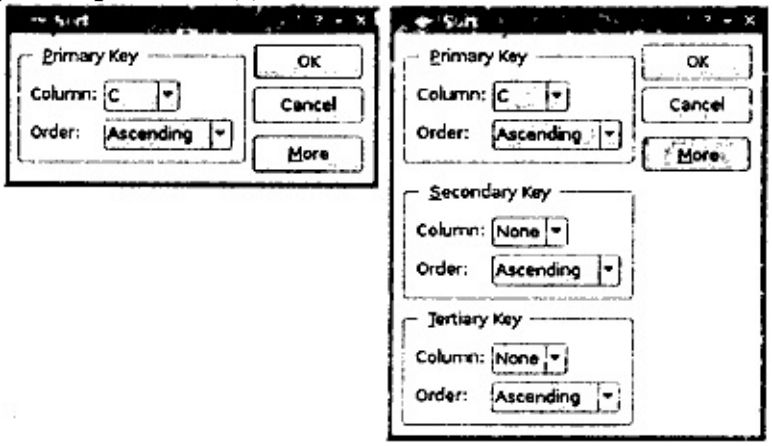


Рис. 2.10. Обычный и расширенный виды окна сортировки данных.

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.

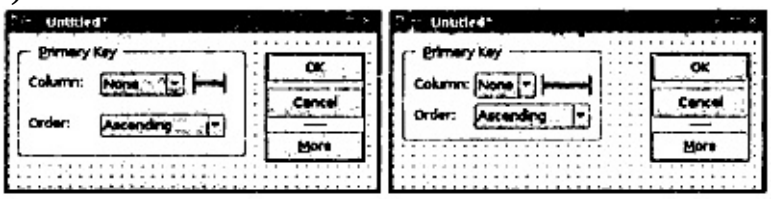
Мы создадим в *Qt Designer* расширенное представление виджета, второй и третий ключи сортировки которого не будут видны при выполнении программы, когда они не нужны. Этот виджет кажется сложным, однако он очень легко строится в *Qt Designer.* Сначала нужно создать ту часть, которая относится к первичному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

1. Выберите функцию меню File | New Form и затем шаблон «Dialog with Buttons Right» (диалог с кнопками, расположенными справа).

2. Создайте кнопку More (больше) и перенесите ее в вертикальный менеджер компоновки ниже вертикальной распорки. Установите свойство *text* кнопки More на значение «&More», а свойство *checkable* — на значение «true». Задайте свойство *default* кнопки OK на значение «true».

3. 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.

4. Передвиньте нижний правый угол элемента группа, увеличивая его. Затем перенесите другие виджеты внутрь элемента группа и расположите их приблизительно так, как показано на рис. 2.11 (а).



(а) Без менеджера компоновки (б) С менеджером компоновки

Рис. 2.11. Размещение дочерних виджетов группового элемента в табличной сетке.

5. Перетащите правый край второго поля с выпадающим списком так, чтобы оно было в два раза шире первого поля.

6. 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).

7. Щелкните правой клавишей мышки 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).

8. Щелкните правой клавишей мышки по второму полю с выпадающим списком и выберите функцию Edit Items. Создайте элементы «Ascending» (по возрастанию) и «Descending» (по убыванию).

9. 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. 2.11 (б).

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 ключей 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) 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. 2.12 (а).

6. Щелкните по форме, чтобы отменить выбор любых виджетов, затем выберите функцию меню Form | Lay Out in a Grid (Форма | Расположить в сетке). Форма должна иметь вид, показанный на рис. 2.12 (б).

7. Свойство *sizeHint* («идеальный» размер) двух вертикальных растяжек установите на значение [20, 0].

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.

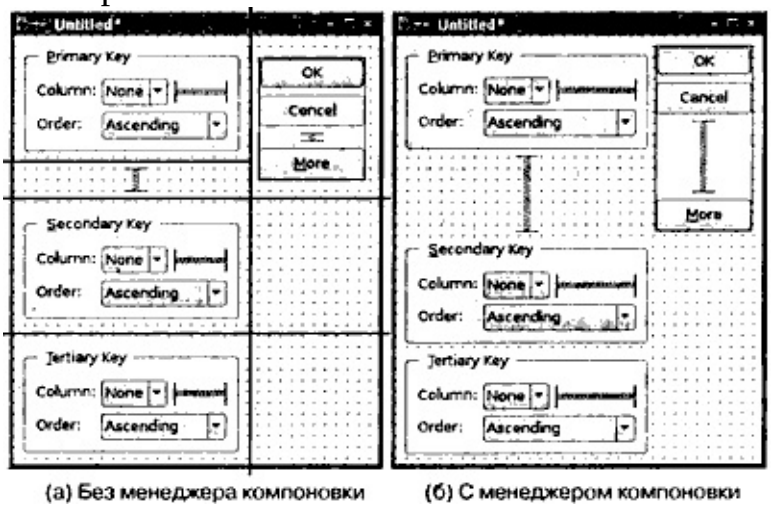


Рис. 2.12. Расположение дочерних элементов формы в сетке.

Переименуйте форму на «SortDialog» (диалоговое окно сортировки) и измените заголовок на «Sort» (сортировка). Задайте имена дочерним виджетам, как показано на рис. 2.13.

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 Моге, 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.

Выберите функцию меню Edit | Edit Signals/Slots (Правка | Редактировать сигналы и слоты) для входа в режим формирования соединений в *Qt Designer.* Соединения представлены синими стрелками между виджетами формы. Поскольку нами выбран шаблон «Dialog with Buttons Right», кнопки OK и CancеI уже подключены к слотам *accept()* и *reject()* виджета *QDialog.* Эти соединения также указаны в окне редактора сигналов и слотов *Qt Designer*.

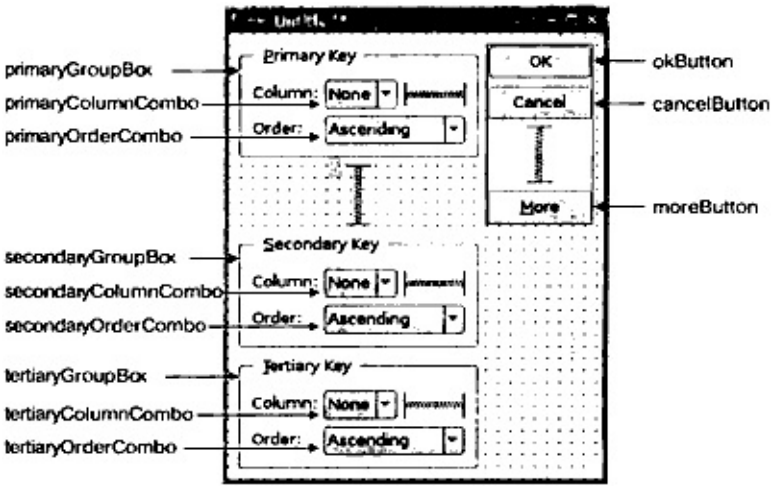


Рис. 2.13. Имена виджетов формы.

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.

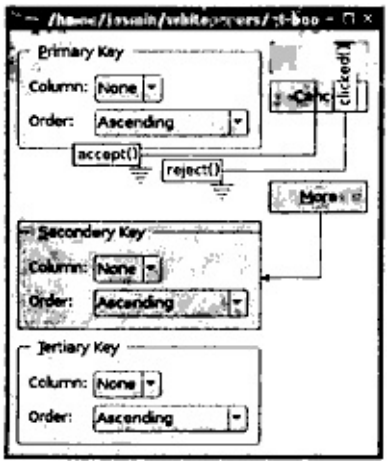


Рис. 2.14. Соединение виджетов формы.

Сначала устанавливается соединение 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).

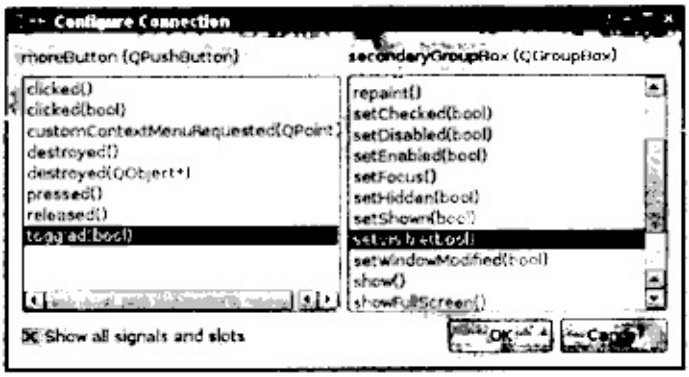


Рис. 2.15. Редактор соединений в *QtDesigner*.

Второе 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:

```
01 #ifndef SORTDIALOG_H
02 #define SORTDIALOG_H
03 #include <QDialog>
04 #include "ui_sortdialog.h"
05 class SortDialog : public QDialog, public Ui::SortDialog
06 {
07 Q_OBJECT
08 public:
09 SortDialog(QWidget *parent = 0);
10 void setColumnRange(QChar first, QChar last);
11 };
12 #endif
```

Затем создаем *sortdialog.cpp:*

```
01 #include <QtGui>
02 #include "sortdialog.h"

03 SortDialog::SortDialog(QWidget *parent)
04 : QDialog(parent)
05 {
06 setupUi(this);
07 secondaryGroupBox->hide();
08 tertiaryGroupBox->hide();
09 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 ключей. Он также устанавливает свойство *sizeConstraint* менеджера компоновки формы на значение *QLayout::SetFixedSize*, не позволяя пользователю изме-

От составителя. Страница №42 в исходном DjVu была пропущена! У кого есть — вставьте.

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 окно с вкладками *QTabWidget*. Здесь сверху окна имеется полоска вкладок, которая находится под управлением стека *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.

Класс стека *QStackedWidget* рассматривается в главе 6 («Управление компоновкой»).

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 точно 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 будет рассматриваться 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. В представленных ниже экранах виджеты используют стиль Plastique.

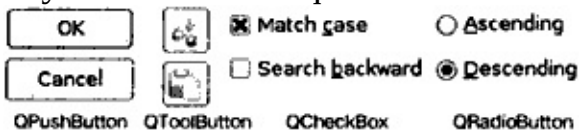


Рис. 2.16. Виджеты кнопок Qt.

Qt содержит четыре вида кнопок: *QPushButton*, *QToolButton*, *QCheckBox* и *QRadioButton*. 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.

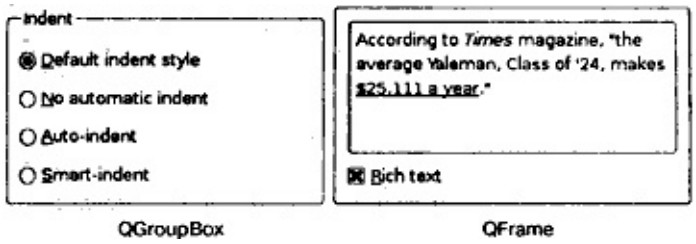


Рис. 2.17. Виджеты одностраничных контейнеров Qt.

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*.

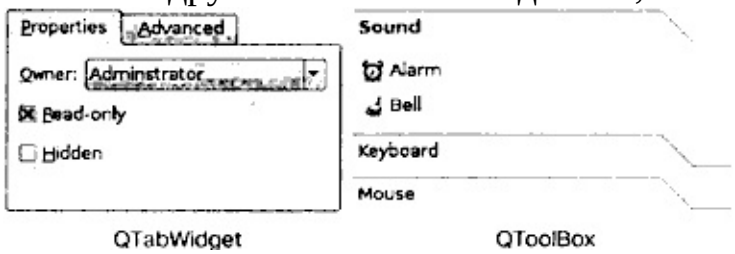


Рис. 2.18. Виджеты многостраничных контейнеров Qt.

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.

Qt имеет несколько виджетов, которые предназначены для простого отображения

информации. Наиболее важным из них является текстовая метка *QLabel*, и она может также использоваться для форматированного отображения текста (используя простой синтаксис, подобный HTML) и вывода изображений.

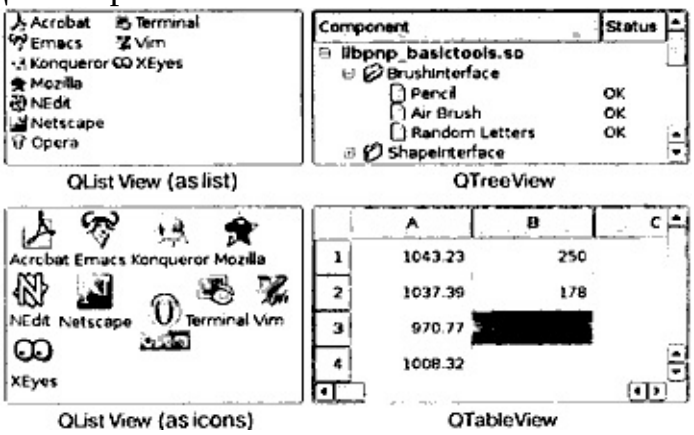


Рис. 2.19. Виджеты для просмотра списков объектов.

Текстовый браузер *QTextBrowser* представляет собой подкласс поля редактирования, работающий только в режиме чтения и обеспечивающий основные возможности формата HTML, включая списки, таблицы, изображения и гипертекстовые ссылки. *Qt Assistant* использует браузер *QTextBrowser* для представления документации пользователю.

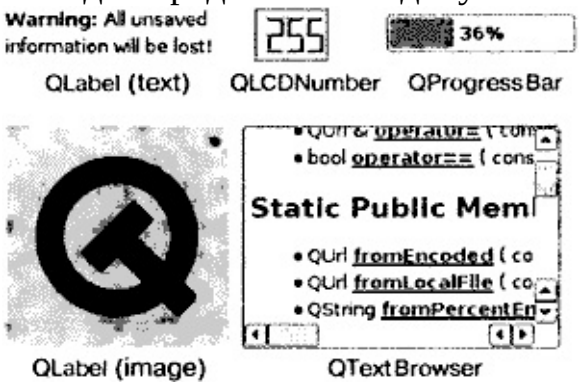


Рис. 2.20. Виджеты отображения данных в Qt.

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

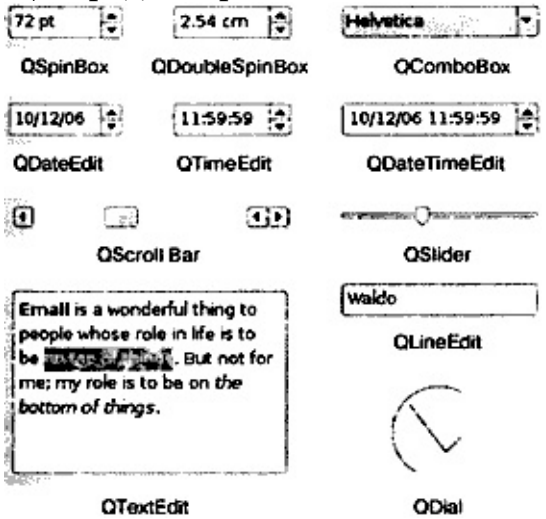


Рис. 2.21. Виджеты ввода данных в Qt.

Qt содержит стандартный набор диалоговых окон для выбора пользователем цвета,

шрифта, файла или для печати документа.

Рис. 2.22. Диалоговое окно выбора цвета и диалоговое окно выбора шрифта в Qt.

В системах Windows и Mac Os X по мере возможности используются «родные» диалоговые окна, а не их общие аналоги.

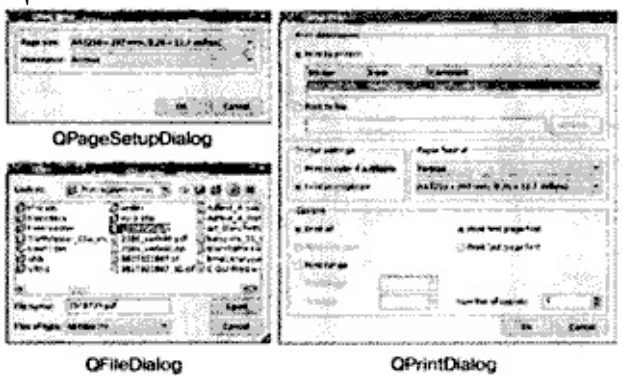


Рис. 2.23. Диалоговое окно для выбора файла и диалоговое окно печати документов в Qt.

Qt содержит разнообразные диалоговые окна для передачи сообщений об ошибках и других сообщений, причем они обеспечивают обратную связь с пользователем. При выполнении продолжительных операций могут использоваться диалоговый индикатор состояния процесса *QProgressDialog* и показанный ранее индикатор состояния процесса без обратной связи *QProgressBar*. Очень удобно пользоваться диалоговым окном *QInputDialog*, когда пользователю требуется ввести одну строку или одно число.

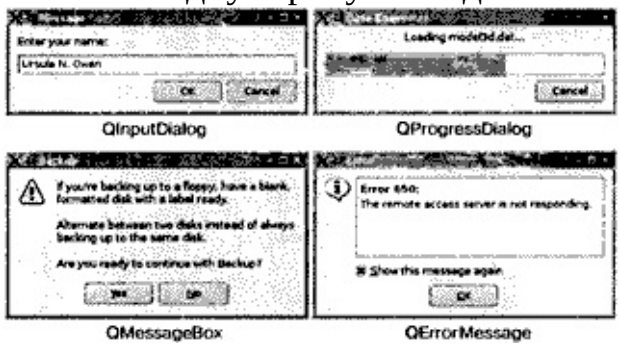
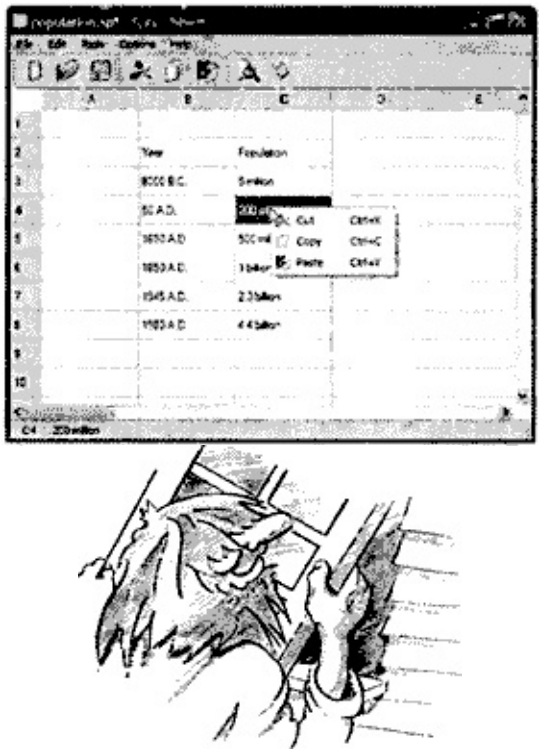


Рис. 2.24. Диалоговые окна для установки обратной связи с пользователем.

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 реализации пользовательских определений слотов.

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.

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 наследниками *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 вы сможете найти в главе «Creating a Main Window Application» (Создание приложения на основе класса главного окна) в онлайновом 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:

```
01 #ifndef MAINWINDOW_H
02 #define MAINWINDOW_H
03 #include <QMainWindow>

04 class QAction;
05 class QLabel;
06 class FindDialog;
07 class Spreadsheet;

08 class MainWindow : public QMainWindow
09 {
10 Q_OBJECT
11 public:
12 MainWindow();
13 protected:
14 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.

```
15 private slots:
16 void newFile();
17 void open();
18 bool save();
19 bool saveAs();
20 void find();
21 void goToCell();
22 void sort();
23 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 может воспользоваться возвращаемым значением, как это мы можем делать при вызове любой обычной функции C++.

```
24 void openRecentFile();
25 void updateStatusBar();
26 void spreadsheetModified();
27 private:
28 void createActions();
29 void createMenus();
30 void createContextMenu();
31 void createToolBars();
32 void createStatusBar();
33 void readSettings();
34 void writeSettings();
35 bool okToContinue();
36 bool loadFile(const QString &fileName);
37 bool saveFile(const QString &fileName);
38 void setCurrentFile(const QString &fileName);
39 void updateRecentFileActions();
40 QString strippedName(const QString &fullFileName);
```

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

```
41 Spreadsheet *spreadsheet;
42 FindDialog *findDialog;
43 QLabel *locationLabel;
44 QLabel *formulaLabel;
45 QStringList recentFiles;
46 QString curFile;

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

50 QMenu *fileMenu;
51 QMenu *editMenu;
…
52 QToolBar *fileToolBar;
53 QToolBar *editToolBar;
54 QAction *newAction;
55 QAction *openAction;
…
56 QAction *aboutQtAction;
```

```
57 };
58 #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 мы кратко рассмотрим реализацию этого подкласса:

```
01 #include <QtGui>
02 #include "finddialog.h"
03 #include "gotocelldialog.h"
04 #include "mainwindow.h"
05 #include "sortdialog.h"
06 #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*.

```
07 MainWindow::MainWindow()
08 {
09 spreadsheet = new Spreadsheet;
10 setCentralWidget(spreadsheet);

11 createActions();
12 createMenus();
13 createContextMenu();
14 createToolBars();
15 createStatusBar();

16 readSettings();
17 findDialog = 0;
18 setWindowIcon(QIcon(":/images/icon.png"));
19 setCurrentFile("");
20 }
```

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.

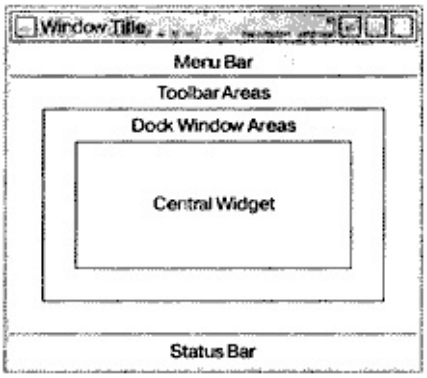


Рис. 3.2. Области главного окна QMainWindow.

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()* мы 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 [4], JPEG, PNG, PNM, 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.

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

• 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 графических 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:

```
<!DOCTYPE RCC><RCC version="1.0">
<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 используемпрефикс 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. Они более подробно рассматриваются в гл. 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.

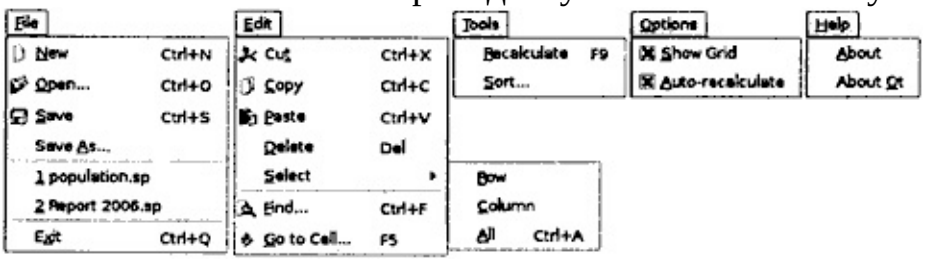


Рис. 3.3. Меню приложения Электронная таблица.

Использование понятия «действия» 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():*

```
01 void MainWindow::createActions()
02 {
03 newAction = new QAction(tr("&New"), this);
04 newAction->setIcon(QIcon(":/images/new.png"));
05 newAction->setShortcut(tr("Ctrl+N"));
06 newAction->setStatusTip(tr("Create a new spreadsheet file"));
07 connect(newAction, SIGNAL(triggered()),
08 this, SLOT(newFile()));
```

Действие New (создать) имеет клавишу быстрого выбора пункта меню (New)[5], родительское окно (главное окно), пиктограмму (*new.png*), клавишу быстрого вызова команды (Ctrl+N) и сообщение в строке состояния. Мы подсоединяем к сигналу этого действия *triggered()* закрытый слот главного окна *newFile();* этот слот мы 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:

```
09 for (int i = 0; i < MaxRecentFiles; ++i)
10 {
11 recentFileActions[i] = new QAction(this);
12 recentFileActions[i]->setVisible(false);
13 connect(recentFileActions[i], SIGNAL(triggered()),
```

```
14 this, SLOT(openRecentFile()));
15 }
```

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.

Теперь перейдем к действию Select All (выделить все):

```
16 selectAllAction = new QAction(tr("&All"), this);
17 selectAllAction->setShortcut(tr("Ctrl+A"));
18 selectAllAction->setStatusTip(tr("Select all the cells in the spreadsheet"));
19 connect(selectAllAction, SIGNAL(triggered()),
20 spreadsheet, SLOT(selectAll()));
```

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

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

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

Действие Show Grid является включаемым. Оно имеет маркер флажка в меню и реализуется как кнопка—переключатель 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 *QTableWidget*. 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*.

```
27 aboutQtAction = new QAction(tr("About &Qt"), this);
28 aboutQtAction->setStatusTip(tr("Show the Qt library's About box"));
29 connect(aboutQtAction, SIGNAL(triggered()),
30 qApp, SLOT(aboutQt()));
31 }
```

Для действия About Qt (справка по средствам разработки Qt) мы используем слот *aboutQt()* объекта *QApplication*, который доступен через глобальную переменную *qApp*.

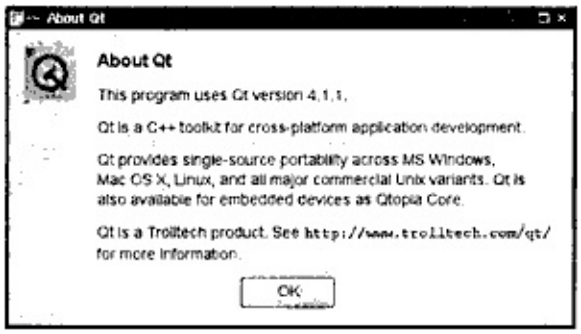


Рис. 3.4. Диалоговое окно About Qt.

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

```
01 void MainWindow::createMenus()
02 {
03 fileMenu = menuBar()->addMenu(tr("&File"));
04 fileMenu->addAction(newAction);
05 fileMenu->addAction(openAction);
06 fileMenu->addAction(saveAction);
07 fileMenu->addAction(saveAsAction);
08 separatorAction = fileMenu->addSeparator();
09 for (int i = 0; i < MaxRecentFiles; ++i)
10 fileMenu->addAction(recentFileActions[i]);
11 fileMenu->addSeparator();
12 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 (файл) и затем добавляем к нему действия New, Open, Save и Save As (создать, открыть, сохранить и сохранить как). Мы вставляем разделитель для визуального выделения группы взаимосвязанных пунктов меню. Мы 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.

```
13 editMenu = menuBar()->addMenu(tr("&Edit"));
14 editMenu->addAction(cutAction);
15 editMenu->addAction(copyAction);
16 editMenu->addAction(pasteAction);
17 editMenu->addAction(deleteAction);

18 selectSubMenu = editMenu->addMenu(tr("&Select"));
19 selectSubMenu->addAction(selectRowAction);
20 selectSubMenu->addAction(selectColumnAction);
21 selectSubMenu->addAction(selectAllAction);
```

```
22 editMenu->addSeparator();
23 editMenu->addAction(findAction);
24 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 мы 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.*

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

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

31 menuBar()->addSeparator();
32 helpMenu = menuBar()->addMenu(tr("&Help"));
33 helpMenu->addAction(aboutAction);
34 helpMenu->addAction(aboutQtAction);
35 }
```

Подобным же образом мы создаем меню Tools, Options и Help (инструменты, опции и помощь). Мы вставляем разделитель между меню Options и Help. В системах Motif и CDE этот разделитель сдвигает меню Help вправо; в других случаях этот разделитель игнорируется.

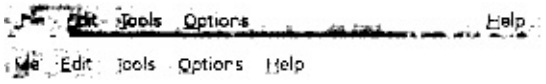


Рис. 3.5. Полоса главного меню в стилях систем Motif и Windows.

```
01 void MainWindow::createContextMenu()
02 {
03 spreadsheet->addAction(copyAction);
04 spreadsheet->addAction(pasteAction);
05 spreadsheet->addAction(cutAction);
06 spreadsheet->setContextMenuPolicy(Qt::ActionsContextMenu);
07 }
```

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 отображение контекстного меню с этими действиями. Контекстные меню вызываются при щелчке правой клавишей мышки по виджету или при нажатии специальной клавиши клавиатуры, зависящей от платформы.

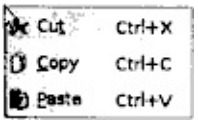


Рис. 3.6. Контекстное меню приложения Электронная таблица.

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()*.

```
01 void Mainwindow::createToolBars()
02 {
03 fileToolBar = addToolBar(tr("&File"));
04 fileToolBar->addAction(newAction);
05 fileToolBar->addAction(openAction);
06 fileToolBar->addAction(saveAction);
07 editToolBar = addToolBar(tr("&Edif));
08 editToolBar->addAction(cutAction);
09 editToolBar->addAction(copyAction);
10 editToolBar->addAction(pasteAction);
11 editToolBar->addSeparator();
12 editToolBar->addAction(findAction);
13 editToolBar->addAction(goToCellAction);
14 }
```

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.

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.

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

```
01 void MainWindow::createStatusBar()
02 {
03 locationLabel = new QLabel(" W999 ");
04 locationLabel->setAlignment(Qt::AlignHCenter);
05 locationLabel->setMinimumSize(locationLabel->sizeHint());
06 formulaLabel = new QLabel;
07 formulaLabel->setIndent(3);
08 statusBar()->addWidget(locationLabel);
09 statusBar()->addWidget(formulaLabel, 1);
10 connect(spreadsheet, SIGNAL(currentCellChanged(int, int, int, int)),
11 this, SLOT(updateStatusBar()));
12 connect(spreadsheet, SIGNAL(modified()),
13 this, SLOT(spreadsheetModified()));
14 updateStatusBar();
15 }
```

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.

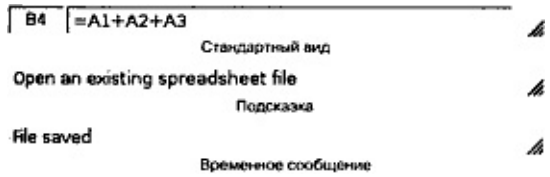


Рис. 3.8. Строка состояния приложения Электронная таблица.

Когда *QStatusBar* располагает виджеты индикаторов, он постарается обеспечить

«идеальный» размер виджетов, заданный функцией *QWidget::sizeHint()*, и затем растянет виджеты, которые допускают растяжение, заполняя дополнительное 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()*.

```
01 void MainWindow::updateStatusBar()
02 {
03 locationLabel->setText(spreadsheet->currentLocation());
04 formulaLabel->setText(spreadsheet->currentFormula());
05 }
```

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()*.

```
06 void MainWindow::spreadsheetModified()
07 {
08 setWindowModified(true);
09 updateStatusBar();
10 }
```

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.

```
01 void MainWindow::newFile()
02 {
03 if (okToContinue ())
04 {
05 spreadsheet->clear();
06 setCurrentFile("");
07 }
08 }
```

Слот *newFile()* вызывается при выборе пользователем пункта меню File | New или при нажатии кнопки New на панели инструментов. Закрытая функция *okToContinue()* задает пользователю вопрос относительно необходимости сохранения изменений («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.

```
01 bool MainWindow::okToContinue()
02 {
03 if (isWindowModified()) {
04 int r = QMessageBox::warning(this,
05 tr("Spreadsheet"), tr("The document has been modified.\n"
06 "Do you want to save your changes?"),
07 QMessageBox::Yes | QMessageBox::Default,
08 QMessageBox::No,
09 QMessageBox::Cancel | QMessageBox::Escape);
10 if (r == QMessageBox::Yes) {
11 return save();
12 } else if (r == QMessageBox::Cancel) {
13 return false;
14 }
15 }
16 return true;
17 }
```

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. Модификатор *QMessageBox::Default* делает Yes кнопкой, которая выбирается по умолчанию. Модификатор *QMessageBox::Escape* задает клавишу Esc в качестве синонима кнопки Cancel.

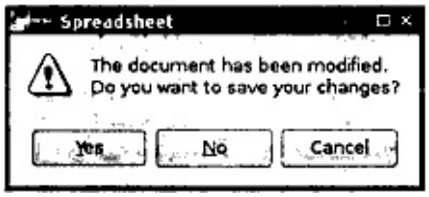


Рис. 3.9. «Сохранить изменения?»

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

QMessageBox::warning(*родительский объект, заголовок, сообщение, кнопка0, кнопка1, …);*

QMessageBox содержит функции *information(), question()* и *critical(),* каждая из которых имеет собственную пиктограмму.

Рис. 3.10. Пиктограммы окна сообщения.

```
01 void MainWindow::open()
02 {
03 if (okToContinue()) {
04 QString fileName = QFileDialog::getOpenFileName(".", fileFilters, this);
05 if (!fileName.isEmpty())
06 loadFile(fileName);
07 }
08 }
```

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)")
```

Закрытая 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:

```
01 bool MainWindow::loadFile(const QString &fileName)
02 {
03 if (!spreadsheet->readFile(fileName)) {
04 statusBar()->showMessage(tr("Loading canceled"), 2000);
05 return false;
06 }
07 setCurrentFile(fileName);
08 statusBar()->showMessage(tr("File loaded"), 2000);
09 return true;
10 }
```

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 2 секунд (2000 миллисекунд) для того, чтобы пользователь знал о выполняемых приложением действиях.

```
01 bool MainWindow::save()
02 {
03 if (curFile.isEmpty()) {
04 return saveAs();
05 } else {
06 return saveFile(curFile);
07 }
08 }

09 bool MainWindow::saveFile(const QString &fileName)
10 {
11 if (!spreadsheet->writeFile(fileName)) {
12 statusBar()->showMessage(tr("Saving canceled"), 2000);
13 return false;
14 }
15 setCurrentFile(fileName);
16 statusBar()->showMessage(tr("File saved"), 2000);
17 return true;
18 }
```

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()*.

```
01 bool MainWindow::saveAs()
02 {
```

```
03 QString fileName = QFileDialog::getSaveFileName(this,
04 tr("SaveSpreadsheet"),
05 tr("Spreadsheet files (*.sp)"));
06 if (fileName.isEmpty())
07 return false;
08 return saveFile(fileName);
09 }
```

С л о т *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*.

```
01 void MainWindow::closeEvent(QCloseEvent *event)
02 {
03 if (okToContinue()) {
04 writeSettings();
05 event->accept();
06 } else {
07 event->ignore();
08 }
09 }
```

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» (закрытие). 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()*.

```
01 void MainWindow::setCurrentFile(const QString &fileName)
02 {
03 curFile = fileName;
04 setWindowModified(false);
05 QString shownName = "Untitled";
06 if (!curFile.isEmpty()) {
07 shownName = strippedName(curFile);
08 recentFiles.removeAll(curFile);
09 recentFiles.prepend(curFile);
```

```
10 updateRecentFileActions();
11 }
12 setWindowTitle(tr("%1[*] - %2").arg(shownName)
13 .arg(tr("Spreadsheet")));
14 }

15 QString MainWindow::strippedName(const QString &fullFileName)
16 {
17 return QFileInfo(fullFileName).fileName();
18 }
```

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 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 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.

```
01 void MainWindow::updateRecentFileActions()
02 {
03 QMutableStringListIterator i(recentFiles);
04 while (i.hasNext()) {
05 if (!QFile::exists(i.next()))
06 i.remove();
07 }
08 for (int j = 0; j < MaxRecentFiles; ++j) {
```

```
09 if (j < recentFiles.count()) {
10 QString text = tr("&%1 %2")
11 .arg(j + 1)
12 .arg(strippedName(recent Files[j]));
13 recentFileActions[j]->setText(text);
14 recentFileActions[j]->setData(recentFiles[j]);
15 recentFileActions[j]->setVisible(true);
16 } else {
17 recentFileActions[j]->setVisible(false);
18 }
19 }
20 separatorAction->setVisible(!recentFiles.isEmpty());
21 }
```

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».

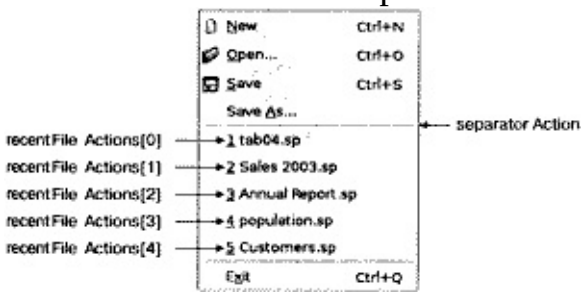


Рис. 3.11. Меню File со списком файлов, которые открывались недавно.

С каждым пунктом меню *recentFileActions* может быть связан элемент данных «data» типа *QVariant*. Тип *QVariant* может хранить многие типы C++ и Qt; он рассматривается в гл. 11. Здесь в элементе меню «data» мы храним полное имя файла, чтобы позже можно было легко его 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.

```
01 void MainWindow::openRecentFile()
02 {
03 if (okToContinue()) {
04 QAction *action = qobject_cast<QAction *>(sender());
05 if (action)
06 loadFile(action->data(). toString());
```

07 }
08 }

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 мета—информации, сгенерированной *moc* — компилятором мета—объектов 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 удачно (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 Б, 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 они 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).

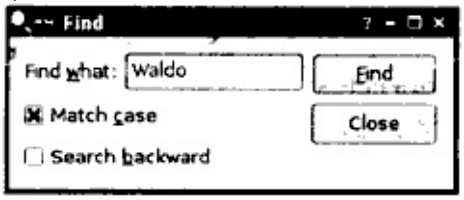


Рис. 3.12. Диалоговое окно Find приложения Электронная таблица.

Мы начнем с диалогового окна Find. 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.

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

```
01 void MainWindow::find()
02 {
03 if (!findDialog) {
04 findDialog = new FindDialog(this);
05 connect(findDialog, SIGNAL (findNext(const QString &,
06 Qt::CaseSensitivity)),
07 spreadsheet, SLOT (findNext(const QString &,
08 Qt::CaseSensitivity)));
09 connect(findDialog, SIGNAL(findPrevious(const QString &,
10 Qt::CaseSensitivity)),
11 spreadsheet, SLOT(findPrevious(const QString &,
12 Qt::CaseSensitivity)));
13 }
14 findDialog->show();
15 findDialog->activateWindow();
16 }
```

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 сэкономит время и память.

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

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

что 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). 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 модальным. Окно называется *модальным*, 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 модальными.

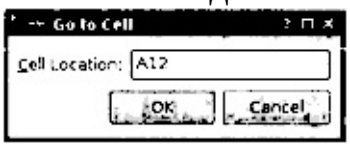


Рис. 3.13. Диалоговое окно Go-to-Cell приложения Электронная таблица.

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

```
01 void MainWindow::goToCell()
02 {
03 GoToCellDialog dialog(this);
04 if (dialog.exec()) {
05 QString str = dialog.lineEdit->text().toUpper();
06 spreadsheet->setCurrentCell(str.mid(1).toInt() - 1,
07 str[0].unicode() - 'A');
08 }
09 }
```

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 знаем, что строка будет иметь правильный формат, потому что осуществляемый нами контроль диалога с помощью *QRegExpValidator* делает кнопку OK активной только в том случае, если за буквой располагается не более трех цифр.

Функция *goToCell()* отличается от приводимого до сих пор программного кода тем, что она создает виджет *(GoToCellDialog)* в виде переменной стека. Мы столь же легко могли бы воспользоваться операторами *new* и *delete*, что увеличило бы программный код только на одну строку:

```
01 void MainWindow::goToCell()
02 {
03 GoToCellDialog *dialog = new GoToCellDialog(this);
04 if (dialog->exec()) {
05 QString str = dialog->lineEdit->text().toUpper();
06 spreadsheet->setCurrentCell(str.mid(1).toInt() - 1,
07 str[0].unicode() - 'A');
08 }
09 delete dialog;
10 }
```

Создание модальных диалоговых окон (и контекстных меню при переопределении *QWidget::contextMenuEvent()*) является обычной практикой программирования, поскольку такое окно (или меню) будет не нужно после его использования, и оно будет автоматически уничтожено при выходе из области видимости.

Теперь мы перейдем к созданию диалогового окна Sort. Это диалоговое окно является модальным и позволяет пользователю упорядочить текущую выбранную область, задавая в качестве ключей сортировки определенные столбцы. На рис. 3.14 показан пример сортировки, когда в качестве главного ключа сортировки используется столбец B, а в качестве вторичного ключа сортировки используется столбец A (в обоих случаях сортировка выполняется по возрастанию значений).

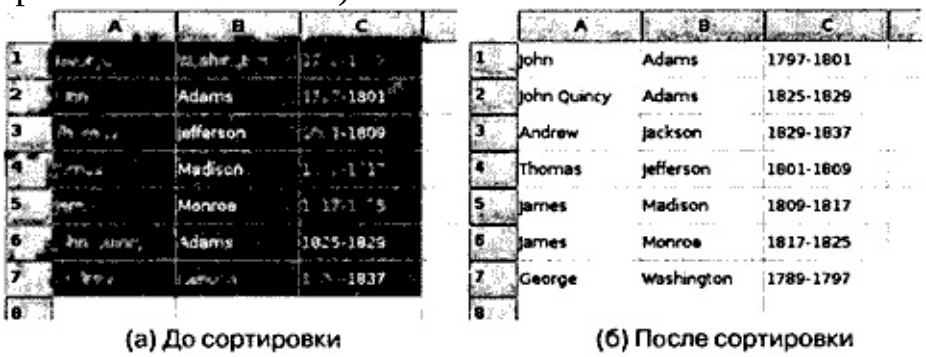


Рис. 3.14. Сортировка выделенной области электронной таблицы.

```
01 void MainWindow::sort()
```

```
02 {
03 SortDialog dialog(this);
04 QTableWidgetSelectionRange range = spreadsheet->selectedRange();
05 dialog.setColumnRange('A' + range.leftColumn(),
06 'A' + range.rightColumn());
07 if (dialog.exec()) {
08 SpreadsheetCompare compare;
09 compare.keys[0] =
10 dialog.primaryColumnCombo->currentIndex();
11 compare.keys[1] =
12 dialog.secondaryColumnCombo->currentIndex() - 1;
13 compare.keys[2] =
14 dialog.tertiaryColumnCombo->currentIndex() - 1;
15 compare.ascending[0] =
16 (dialog.primaryOrderCombo->currentIndex() == 0);
17 compare.ascending[1] =
18 (dialog.secondaryOrderCombo->currentIndex() == 0);
19 compare.ascending[2] =
20 (dialog.tertiaryOrderCombo->currentIndex() == 0);
21 spreadsheet->sort(compare);
22 }
23 }
```

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 OK, 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 значения «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():*

```
01 void MainWindow::sort()
02 {
03 SortDialog dialog(this);
04 QTableWidgetSelectionRange range = spreadsheet->selectedRange();
05 dialog.setColumnRange('A' + range.leftColumn(),
06 'A' + range.rightColumn());
07 if (dialog.exec())
08 spreadsheet->performSort(dialog.comparisonObject());
09 }
```

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:

```
01 void MainWindow::sort()
02 {
03 SortDialog dialog(this);
04 dialog.setSpreadsheet(spreadsheet);
05 dialog.exec();
06 }
```

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 ненадежен 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 Find или Go-to-Cell, но поскольку диалоговые окна About сильно стилизованы, в средствах разработки Qt предусмотрено простое решение:

```
01 void MainWindow::about()
```

```
02 {
03 QMessageBox::about(this, tr("About Spreadsheet"),
04 tr("<h2>Spreadsheet 1.1</h2>"
05 "<p>Copyright &copy; 2006 Software Inc."
06 "<p>Spreadsheet is a small application that "
07 "demonstrates QAction, QMainWindow, QMenuBar, "
08 "QStatusBar, QTableWidget, QToolBar, and many other "
09 "Qt classes."));
10 }
```

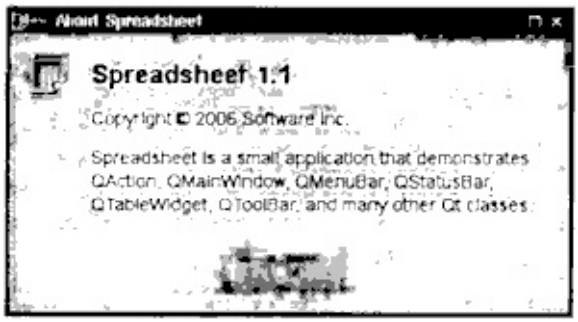


Рис. 3.15. Справка о приложении Электронная таблица.

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.

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.

```
01 void MainWindow::writeSettings()
02 {
03 QSettings settings("Software Inc.", "Spreadsheet");
04 settings.setValue("geometry", geometry());
05 settings.setValue("recentFiles", recentFiles);
06 settings.setValue("showGrid", showGridAction->isChecked());
07 settings.setValue("autoRecalc", autoRecalcAction->isChecked());
08 }
```

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 можно задавать, 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 типы *int, bool, double, QString, QStringList* или любой другой, поддерживаемый *QVariant*, включая зарегистрированные пользовательские типы.

```
01 void MainWindow::readSettings()
02 {
03 QSettings settings("Software Inc.", "Spreadsheet");
04 QRect rect = settings.value("geometry",
05 QRect(200, 200, 400, 400)).toRect();
06 move(rect.topLeft());
07 resize(rect.size());
08 recentFiles = settings.value("recentFiles").toStringList();
09 updateRecentFileActions();
10 bool showGrid = settings.value("showGrid", true).toBool();
```

```
11 showGridAction->setChecked(showGrid);
12 bool autoRecalc = settings.value("autoRecalc", true).toBool();
13 autoRecalcAction->setChecked(autoRecalc);
14 }
```

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 списка недавно используемых файлов, этот список будет пустым при первом запуске приложения.

Qt содержит функцию *QWidget::setGeometry()*, которая дополняет функцию *QWidget::geometry()*, однако они не всегда работают должным образом в системе X11 из-за ограничений многих оконных менеджеров. По этой причине мы используем вместо них функции *move()* и *resize()*. (Подробную информацию по тому вопросу можно найти по адресу http://doc.trolltech.com/4.1/geometry.html.)

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 со многими документами

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

```
01 #include <QApplication>
02 #include "mainwindow.h"
03 int main(int argc, char *argv[])
04 {
05 QApplication app(argc, argv);
06 MainWindow mainWin;
07 mainWin.show();
08 return app.exec();
09 }
```

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 данной 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 окно с пустым документом вместо повторного использования существующего главного окна;

• пункт меню File | Close закрывает текущее главное окно;

• пункт меню File | Exit закрывает все окна.

Рис. 3.16. Новое меню File.

В первоначальной версии меню File не было пункта Close (закрыть), поскольку он выполнял бы ту же функцию, что и пункт меню Exit. Новая функция *main()* примет следующий вид:

```
01 int main(int argc, char *argv[])
02 {
03 QApplication app(argc, argv);
04 MainWindow *mainWin = new MainWindow;
05 mainWin->show();
06 return app.exec();
```

```
07 }
```

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.

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

```
01 void MainWindow::newFile()
02 {
03 MainWindow *mainWin = new MainWindow;
04 mainWin->show();
05 }
```

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:

```
01 void MainWindow::createActions()
02 {
03 closeAction = new QAction(tr("&Close"), this);
04 closeAction->setShortcut(tr("Ctrl+W"));
05 closeAction->setStatusTip(tr("Close this window"));
06 connect(closeAction, SIGNAL(triggered()), this, SLOT(close()));
07 exitAction = new QAction(tr("E&xit"), this);
08 exitAction->setShortcut(tr("Ctrl+Q"));
09 exitAction->setStatusTip(tr("Exit the application"));
10 connect(exitAction, SIGNAL(triggered()),
11 qApp, SLOT(closeAllWindows()));
12 }
```

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 незамеченной. 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 исчезает с экрана, но по-прежнему остается в памяти. При создании многих окон может возникнуть проблема.

Решение состоит в установке признака *Qt::WA_DeleteOnClose* в конструкторе:

```
01 MainWindow::MainWindow()
02 {
03 setAttribute(Qt::WA_DeleteOnClose);
04 }
```

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 загружены два файла с одинаковым именем.

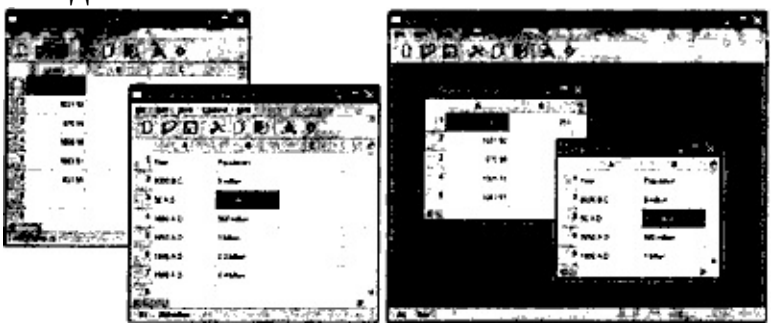


Рис. 3.17. Однодокументный и многодокументный интерфейсы.

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 рассматривается в главе 6 («Управление компоновкой»).

Экранные заставки

Многие приложения при запуске выводят на экран заставки. 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.*

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.

```
01 int main(int argc, char *argv[])
02 {
03 QApplication app(argc, argv);
04 QSplashScreen *splash = new QSplashScreen;
05 splash->setPixmap(QPixmap(":/images/splash.png"));
06 splash->show();
07 Qt::Alignment topRight = Qt::AlignRight | Qt::AlignTop;
08 splash->showMessage(QObject::tr("Setting up the main window…"),
09 topRight, Qt::white);
10 MainWindow mainWin;
11 splash->showMessage(QObject::tr("Loading modules…"),
12 topRight, Qt::white);
13 loadModules();
14 splash->showMessage(QObject::tr("Establishing connections…"),
15 topRight, Qt::white);
16 establishConnections();
17 mainWin.show();
18 splash->finish(&mainWin);
19 delete splash;
20 return app.exec();
21 }
```

Рис. 3.18. Экранная заставка.

Теперь мы завершили пользовательский интерфейс приложения Электронная таблица. В

следующей главе мы реализуем базовые функции электронной таблицы и на этом завершим построение этого приложения.

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. Рабочая область (workspace) интерфейса MDI

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

Менеджеры компоновки, разделители и рабочие 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 «=Al+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* наследует *QTableWidget*. Виджет *QTableWidget* фактически является сеткой, представляющей собой двумерный разряженный 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.

Давайте начнем с реализации виджета и сначала приведем заголовочный файл:

```
01 #ifndef SPREADSHEET_H
02 #define SPREADSHEET_H
03 #include <QTableWidget>
04 class Cell;
05 class SpreadsheetCompare;
```

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

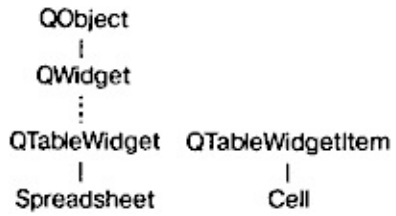


Рис. 4.1. Деревья наследования для классов Spreadsheet и Cell.

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* наследует *QTableWidgetItem*, и мы рассмотрим этот класс в последнем разделе данной главы, где представим его реализацию.

```
06 class Spreadsheet : public QTableWidget
07 {
08 Q_OBJECT
09 public:
10 Spreadsheet(QWidget *parent = 0);
11 bool autoRecalculate() const { return autoRecalc; }
12 QString currentLocation() const;
13 QString currentFormula() const;
14 QTableWidgetSelectionRange selectedRange() const;
15 void clear();
16 bool readFile(const QString &fileName);
17 bool writeFile(const QString &fileName);
18 void sort(const SpreadsheetCompare &compare);
```

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

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 *QTableWidget*, а именно *setCurrentCell()* и *setShowGrid()*.

```
19 public slots:
20 void cut();
```

```
21 void copy();
22 void paste();
23 void del();
24 void selectCurrentRow();
25 void selectCurrentColumn();
26 void recalculate();
27 void setAutoRecalculate(bool recalc);
28 void findNext(const QString &str, Qt::CaseSensitivity cs);
29 void findPrevious(const QString &str, Qt::CaseSensitivity cs);
30 signals:
31 void modified();
```

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

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

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

```
34 private:
35 enum { MagicNumber = 0x7F51C883, RowCount = 999, ColumnCount = 26 };
36 Cell *cell(int row, int column) const;
37 QString text(int row, int column) const;
38 QString formula(int row, int column) const;
39 void setFormula(int row, int column, const QString &formula);
40 bool autoRecalc;
41 };
```

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

```
42 class SpreadsheetCompare
43 {
44 public:
45 bool operator()(const QStringList &row1, const QStringList &row2) const;
46 enum { KeyCount = 3 };
47 int keys[KeyCount];
48 bool ascending[KeyCount];
49 };
50 #endif
```

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

Теперь мы рассмотрим реализацию:

```
01 #include <QtGui>
02 #include "cell.h"
03 #include "spreadsheet.h"
04 Spreadsheet::Spreadsheet(QWidget *parent)
05 : QTableWidget(parent)
06 {
```

```
07 autoRecalc = true;
08 setItemPrototype(new Cell);
09 setSelectionMode(ContiguousSelection);
10 connect(this, SIGNAL(itemChanged(QTableWidgetItem *)),
11 this, SLOT(somethingChanged()));
12 clear();
13 }
```

Обычно при вводе пользователем некоторого текста в пустую ячейку *QTableWidget* 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, *QTableWidget* 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.

```
14 void Spreadsheet::clear()
15 {
16 setRowCount(0);
17 setColumnCount(0);
18 setRowCount(RowCount);
19 setColumnCount(ColumnCount);
20 for (int i = 0; i < ColumnCount; ++i) {
21 QTableWidgetItem *item = new QTableWidgetItem;
22 item->setText(QString(QChar('A' + i)));
23 setHorizontalHeaderItem(i, item);
24 }
25 setCurrentCell(0, 0);
26 }
```

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

Мы могли бы использовать *QTableWidget::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.

Рис. 4.2. Виджеты, составляющие QTableWidget.

QTableWidget 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 *QTableWidget* 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* рассматривается в главе 6.

Хранение данных в объектах типа «элемент»

В приложении Электронная таблица каждая непустая ячейка хранится в памяти в виде одного объекта *QTableWidgetltem* (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, *QTableWidgetltem* 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 *QVariant*, 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.

```
01 Cell *Spreadsheet::cell(int row, int column) const
02 {
03 return static_cast<Cell *>(item(row, column));
04 }
```

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 *QTableWidgetltem.*

```
01 QString Spreadsheet::text(int row, int column) const
02 {
03 Cell *c = cell(row, column);
04 if (c) {
05 return c->text();
06 } else {
07 return "";
08 }
09 }
```

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

```
01 QString Spreadsheet::formula(int row, int column) const
02 {
03 Cell *c = cell(row, column);
04 if (c) {
05 return c->formula();
06 } else {
07 return "";
08 }
09 }
```

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. 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.

```
01 void Spreadsheet::setFormula(int row, int column, const QString &formula)
```

```
02 {
03 Cell *c = cell(row, column);
04 if (!c) {
05 c = new Cell;
06 setItem(row, column, c);
07 }
08 c->setFormula(formula);
09 }
```

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.

```
01 QString Spreadsheet::currentLocation() const
02 {
03 return QChar('A' + currentColumn())
04 + QString::number(currentRow() + 1);
05 }
```

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.

```
01 QString Spreadsheet::currentFormula() const
02 {
03 return formula(currentRow(), currentColumn());
04 }
```

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

```
01 void Spreadsheet::somethingChanged()
02 {
03 if (autoRecalc)
04 recalculate();
05 emit modified();
06 }
```

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:

```
01 bool Spreadsheet::writeFile(const QString &fileName)
02 {
03 QFile file(fileName);
04 if (!file.open(QIODevice::WriteOnly)) {
05 QMessageBox::warning(this, tr("Spreadsheet"),
06 tr("Cannot write file %1:\n%2.")
07 .arg(file.fileName())
08 .arg(file.errorString()));
09 return false;
10 }

11 QDataStream out(&file);
12 out.setVersion(QDataStream::Qt_4_1);
13 out << quint32(MagicNumber);
14 QApplication::setOverrideCursor(Qt::WaitCursor);

15 for (int row = 0; row < RowCount; ++row) {
16 for (int column = 0; column < ColumnCount; ++column) {
17 QString str = formula(row, column);
18 if (!str.isEmpty())
19 out << quint16(row) << quint16(column) << str;
20 }
21 }

22 QApplication::restoreOverrideCursor();
23 return true;
24 }
```

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 базовые типы C++, такие как *char*, *short*, *int*, *long* и *long long*, 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, каждый из которых содержит строку, столбец и формулу одной ячейки. Для экономии места мы не заполняем пустые ячейки.

Рис. 4.3. Формат файла данных для приложения Электронная таблица.

Точное 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 7 в Qt 4.1), 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 7 вне зависимости от версии Qt, в которой оно компилируется. (Для удобства используется константа *QDataStream::Qt_4_1*, равная 7.)

Класс *QDataStream* достаточно универсален. Он может использоваться для объекта *QFile*, но также и для *QBuffer*, *QProcess*, *QTcpSocket* или *QUdpSocket*. 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*.

```
01 bool Spreadsheet::readFile(const QString &fileName)
02 {
03 QFile file(fileName);
04 if (!file.open(QIODevice::ReadOnly)) {
05 QMessageBox::warning(this, tr("Spreadsheet"),
06 tr("Cannot read file %1:\n%2.")
07 .arg(file.fileName())
08 .arg(file.errorString()));
09 return false;
10 }

11 QDataStream in(&file);
12 in.setVersion(QDataStream::Qt_4_1);
13 quint32 magic;
```

```
14 in >> magic;
15 if (magic != MagicNumber) {
16 QMessageBox::warning(this, tr("Spreadsheet"),
17 tr("The file is not a Spreadsheet file."));
18 return false;
19 }

20 clear();
21 quint16 row;
22 quint16 column;
23 QString str;
24 QApplication::setOverrideCursor(Qt::WaitCursor);
25 while (!in.atEnd()) {
26 in >> row >> column >> str;
27 setFormula(row, column, str);
28 }
29 QApplication::restoreOverrideCursor();
30 return true;
31 }
```

Функция *readFile()* очень напоминает *writeFile()*. Для чтения файла мы пользуемся объектом *QFile*, но теперь мы используем флажок *QIODevice::ReadOnly*, а не *QIODevice::WriteOnly*. Затем мы устанавливаем версию *QDataStream* на значение 7. 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.

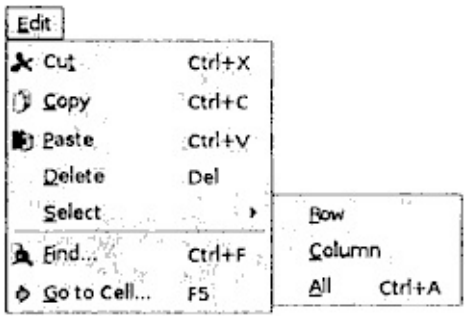


Рис. 4.4. Меню Edit приложения Электронная таблица.

```
01 void Spreadsheet::cut()
02 {
03 copy();
04 del();
05 }
```

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.

```
01 void Spreadsheet::copy()
02 {
03 QTableWidgetSelectionRange range = selectedRange();
04 QString str;
05 for (int i = 0; i < range.rowCount(); ++i) {
06 if (i > 0)
07 str += "\n";
08 for (int j = 0; j < range.columnCount(); ++j) {
09 if (j > 0)
10 str += "\t";
11 str += formula(range.topRow() + i, range.leftColumn() + j);
12 }
13 }
14 QApplication::clipboard()->setText(str);
15 }
```

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.

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 многим приложениям, включая Excel от компании Microsoft.

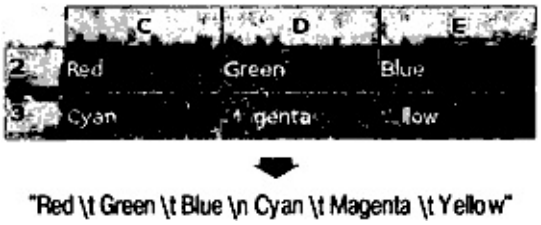


Рис. 4.5. Копирование выделенных ячеек в буфер обмена.

Функция *QTableWidget::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:

```
01 QTableWidgetSelectionRange Spreadsheet::selectedRange() const
02 {
03 QList<QTableWidgetSelectionRange> ranges = selectedRanges();
04 if (ranges.isEmpty())
05 return QTableWidgetSelectionRange();
06 return ranges.first();
07 }
```

Если выделение вообще имеет место, мы возвращаем первую (и единственную) выделенную область. Мы никогда не встретимся с ситуацией, когда не выбрано никакой области, 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.

```
01 void Spreadsheet::paste()
02 {
03 QTableWidgetSelectionRange range = selectedRange();
04 QString str = QApplication::clipboard()->text();
05 QStringList rows = str.split('\n');
06 int numRows = rows.count();
07 int numColumns = rows.first().count('\t') + 1;

08 if (range.rowCount() * range.columnCount() != 1
09 && (range.rowCount() != numRows
10 || range.columnCount() !=numColumns)) {
11 QMessageBox::information(this, tr("Spreadsheet"),
12 tr("The information cannot be pasted because the copy "
13 "and paste areas aren't the same size."));
14 return;
15 }

16 for (int i = 0; i < numRows; ++i) {
17 QStringList columns = rows[i].split('\t');
18 for (int j = 0; j < numColumns; ++j) {
19 int row = range.topRow() + i;
20 int column = range.leftColumn() + j;
```

```
21 if (row < RowCount && column < ColumnCount)
22 setFormula(row, column, columns[j]);
23 }
24 }
25 somethingChanged();
26 }
```

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.

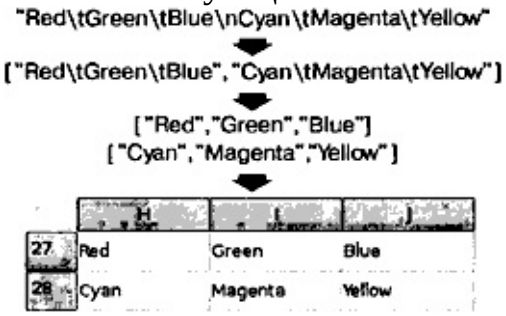


Рис. 4.6. Вставка текста из буфера обмена в электронную таблицу.

```
01 void Spreadsheet::del()
02 {
03 foreach (QTableWidgetItem *item, selectedItems())
04 delete item;
05 }
```

Слот *del()* соответствует пункту меню Edit | Delete (Правка | Удалить). Для очистки ячеек достаточно использовать оператор *delete* для каждого объекта *Cell.* Объект *QTableWidget* 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.

```
01 void Spreadsheet::selectCurrentRow()
02 {
03 selectRow(currentRow());
04 }
05 void Spreadsheet::selectCurrentColumn()
06 {
07 selectColumn(currentColumn());
08 }
```

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()*.

```
01 void Spreadsheet::findNext(const QString &str, Qt::CaseSensitivity cs)
02 {
03 int row = currentRow();
04 int column = currentColumn() + 1;
05 while (row < RowCount) {
06 while (column < ColumnCount) {
07 if (text(row, column).contains(str, cs)) {
08 clearSelection();
09 setCurrentCell(row, column);
10 activateWindow();
11 return;
12 }
13 ++column;
14 }
15 column = 0;
16 ++row;
17 }
18 QApplication::beep();
19 }
```

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 no A25, B25, C25, … Z25 и так далее, 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.

```
01 void Spreadsheet::findPrevious(const QString &str, Qt::CaseSensitivity cs)
02 {
03 int row = currentRow();
04 int column = currentColumn() - 1;
05 while (row>= 0) {
06 while (column >= 0) {
07 if (text(row, column).contains(str, cs)) {
08 clearSelection();
09 setCurrentCell(row, column);
10 activateWindow();
11 return;
```

```
12 }
13 --column;
14 }
15 column = ColumnCount - 1;
16 --row;
17 }
18 QApplication::beep();
19 }
```

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.

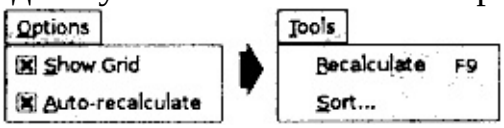


Рис. 4.7. Меню Tools и Options приложения Электронная таблица.

```
01 void Spreadsheet::recalculate()
02 {
03 for (int row = 0; row < RowCount; ++row) {
04 for (int column = 0; column < ColumnCount; ++column) {
05 if (cell(row, column))
06 cell(row, column)->setDirty();
07 }
08 }
09 viewport()->update();
10 }
```

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. В 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*.

```
01 void Spreadsheet::setAutoRecalculate(bool recalc)
02 {
03 autoRecalc = recalc;
04 if (autoRecalc)
05 recalculate();
06 }
```

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 базового класса *QTableView*. Остается только реализовать функцию *Spreadsheet::sort()*, которая вызывается из *MainWindow::sort()*:

```
01 void Spreadsheet::sort(const SpreadsheetCompare &compare)
02 {
03 QList<QStringList> rows;
04 QTableWidgetSelectionRange range = selectedRange();
05 int i;
06 for (i = 0; i < range.rowCount(); ++i) {
07 QStringList row;
08 for (int j = 0; j < range.columnCount(); ++j)
09 row.append(formula(range.topRow() + i,
10 range.leftColumn() + j));
11 rows.append(row);
12 }

13 qStableSort(rows.begin(), rows.end(), compare);
14 for (i = 0; i < range.rowCount(); ++i) {
15 for (int j = 0; j < range.columnCount(); ++j)
16 setFormula(range.topRow() + i, range.leftColumn() + j, rows[i][j]);
17 }

18 clearSelection();
19 somethingChanged();
20 }
```

Сортировка работает на текущей выделенной области и переупорядочивает строки в соответствии со значениями ключей порядка сортировки, 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. Стандартные алгоритмы и структуры данных Qt рассматриваются в главе 11 («Классы—контейнеры»).

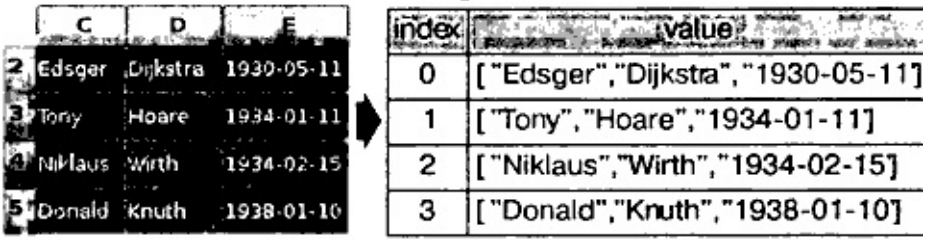


Рис. 4.8. Хранение выделенной области в виде списка строк.

В качестве аргументов функции *qStableSort()* используются 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.

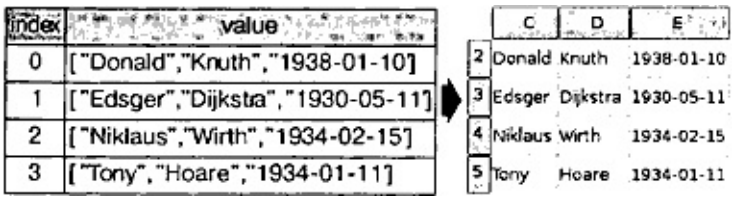


Рис. 4.9. Помещение данных в таблицу после сортировки.

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:

```
01 class SpreadsheetCompare
02 {
03 public:
04 bool operator()(const QStringList &row1,
05 const QStringList &row2) const;
06 enum { KeyCount = 3 };
07 int keys[KeyCount];
08 bool ascending[KeyCount];
09 };
```

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:

```
01 class Square
02 {
03 public:
04 int operator()(int x) const { return x * x; }
05 }
```

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);
```

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 неизящно 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:

```
01 bool SpreadsheetCompare::operator()(const QStringList &row1,
02 const QStringList &row2) const
03 {
04 for (int i = 0; i < KeyCount; ++i) {
05 int column = keys[i];
06 if (column != -1) {
07 if (row1[column] != row2[column]) {
08 if (ascending[i]) {
09 return row1[column] < row2[column];
10 } else {
11 return row1[column] > row2[column];
12 }
13 }
14 }
15 }
16 return false;
17 }
```

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 *QTableWidget.* Ниже приводится заголовочный файл:

```
01 #ifndef CELL_H
02 #define CELL_H
03 #include <QTableWidgetItem>

04 class Cell : public QTableWidgetItem
05 {
06 public:
07 Cell();
08 QTableWidgetItem *clone() const;
09 void setData(int role, const QVariant &value);
10 QVariant data(int role) const;
11 void setFormula(const QString &formula);
12 QString formula() const;
13 void setDirty();

14 private:
15 QVariant value() const;
16 QVariant evalExpression(const QString &str, int &pos) const;
17 QVariant evalTerm(const QString &str, int &pos) const;
18 QVariant evalFactor(const QString &str, int &pos) const;
19 mutable QVariant cachedValue;
20 mutable bool cacheIsDirty;
21 };
22 #endif
```

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

• переменная *cachedValue* кэширует значение ячейки в виде значения типа *QVariant;*

• переменная *cacheIsDirty* принимает значение *true,* если кэшируемое 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 является наследником *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:*

```
01 #include <QtGui>
02 #include "cell.h"
03 Cell::Cell()
04 {
05 setDirty();
06 }
```

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

Каждый элемент *QTableWidgetltem* 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).

```
02 QTableWidgetltem *Cell::clone() const
03 {
04 return new Cell(*this);
05 }
```

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

```
06 void Cell::setFormula(const QString &formula)
07 {
08 setData(Qt::EditRole, formula);
09 }
```

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

```
10 QString Cell::formula() const
11 {
12 return data(Qt::EditRole).toString();
13 }
```

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

```
14 void Cell::setData(int role, const QVariant &value)
15 {
16 QTableWidgetltem::setData(role, value);
```

```
17 if (role == Qt::EditRole)
18 setDirty();
19 }
```

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()*.

```
20 void Cell::setDirty()
21 {
22 cacheIsDirty = true;
23 }
```

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.

```
24 QVariant Cell::data(int role) const
25 {
26 if (role == Qt::DisplayRole) {
27 if (value().isValid()) {
28 return value().toString();
29 } else {
30 return "####";
31 }
32 } else if (role == Qt::TextAlignmentRole) {
33 if (value().type() == QVariant::String) {
34 return int(Qt::AlignLeft | Qt::AlignVCenter);
35 } else {
36 return int(Qt::AlignRight | Qt::AlignVCenter);
37 }
38 } else {
39 return QTableWidgetItem::data(role);
40 }
41 }
```

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).

```
42 const QVariant Invalid;

43 QVariant Cell::value() const
44 {
45 if (cacheIsDirty) {
46 cacheIsDirty = false;

47 QString formulaStr = formula();
48 if (formulaStr.startsWith('\'')) {
49 cachedValue = formulaStr.mid(1);
50 } else if (formulaStr.startsWith('=')) {
51 cachedValue = Invalid;
52 QString expr = formulaStr.mid(1);
53 expr.replace(" ", "");
54 expr.append(QChar::Null);

55 int pos = 0;
56 cachedValue = evalExpression(expr, pos);
57 if (expr[pos] != QChar::Null)
58 cachedValue = Invalid;
59 } else {
60 bool ok;
61 double d = formulaStr.toDouble(&ok);
62 if (ok) {
63 cachedValue = d;
64 } else {
65 cachedValue = formulaStr;
66 }
67 }
68 }
69 return cachedValue;
70 }
```

Закрытая функция *value()* возвращает значение ячейки. Если флажок *cacheIsDirty* 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 удается выполнить, мы устанавливаем *cachedValue* на полученное значение; в противном случае мы 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 указать 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 рассматриваются 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 могут быть число («2»), обозначение ячейки («C5») или выражение в скобках, перед которым может стоять знак минуса.

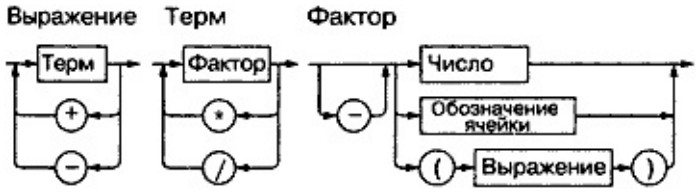


Рис. 4.10. Блок—схема синтаксического анализа выражений электронной таблицы.

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).

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

```
01 QVariant Cell::evalExpression(const QString &str, int &pos) const
02 {
03 QVariant result = evalTerm(str, pos);
04 while (str[pos] != QChar::Null) {
05 QChar op = str[pos];
06 if (op != '+' && op != '-') return result;
07 ++pos;

08 QVariant term = evalTerm(str, pos);
09 if (result.type() == QVariant::Double
10 && term.type() == QVariant::Double) {
11 if (op == '+') {
12 result = result.toDouble() + term.toDouble();
13 } else {
14 result= result.toDouble() - term.toDouble();
15 }
16 } else {
17 result = Invalid;
18 }
19 }
20 return result;
21 }
```

Во-первых, мы вызываем функцию *evalTerm()* для получения значения первого терма. Если за ним идет символ «+» или «—», мы вызываем второй раз *evalTerm();* в противном случае выражение состоит из единственного терма, 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)».

```
01 QVariant Cell::evalTerm(const QString &str, int &pos) const
02 {
03 QVariant result = evalFactor(str, pos);
```

```
04 while (str[pos] != QChar::Null) {
05 QChar op = str[pos];
06 if (op != '*' && op != '/')
07 return result;
08 ++pos;

09 QVariant factor = evalFactor(str, pos);
10 if (result.type() == QVariant::Double &&
11 factor.type() == QVariant::Double) {
12 if (op == '*') {
13 result = result.toDouble() * factor.toDouble();
14 } else {
15 if (factor.toDouble() == 0.0) {
16 result = Invalid;
17 } else {
18 result = result.toDouble() / factor.toDouble();
19 }
20 }
21 } else {
22 result = Invalid;
23 }
24 }
25 return result;
26 }
```

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 деления 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.

```
01 QVariant Cell::evalFactor(const QString &str, int &pos) const
02 {
03 QVariant result;
04 bool negative = false;
05 if (str[pos] == '-') {
06 negative = true;
07 ++pos;
08 }
09 if (str[pos] == '(') {
10 ++pos;
11 result = evalExpression(str, pos);
12 if (str[pos] != ')')
13 result = Invalid;
14 ++pos;
15 } else {
```

```
16 QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
17 QString token;
18 while (str[pos].isLetterOrNumber() || str[pos] == '.') {
19 token += str[pos];
20 ++pos;
21 }

22 if (regExp.exactMatch(token)) {
23 int column = token[0].toUpper().unicode() - 'A';
24 int row = token.mid(1).toInt() - 1;
25 Cell *c = static_cast<Cell *>(tableWidget()->item(row, column));
26 if (c) {
27 result = c->value();
28 } else {
29 result = 0.0;
30 }
31 } else {
32 bool ok;
33 result = token.toDouble(&ok);
34 if (!ok)
35 result = Invalid;
36 }
37 }

38 if (negative) {
39 if (result.type() == QVariant::Double) {
40 result = -result.toDouble();
41 } else {
42 result = Invalid;
43 }
44 }
45 return result;
46 }
```

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» («грязный») 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.

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 мы 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.

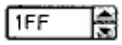


Рис. 5.1. Виджет HexSpinBox.

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.

```
01 #ifndef HEXSPINBOX_H
02 #define HEXSPINBOX_H
03 #include <QSpinBox>
04 class QRegExpValidator;

05 class HexSpinBox : public QSpinBox
06 {
07 Q_OBJECT
08 public:
09 HexSpinBox(QWidget *parent = 0);
10 protected:
11 QValidator::State validate(QString &text, int &pos) const;
12 int valueFromText(const QString &text) const;
13 QString textFromValue(int value) const;
14 private:
15 QRegExpValidator *validator;
16 };
17 #endif
```

Шестнадцатеричный наборный счетчик *HexSpinBox* наследует большую часть функциональности от *QSpinBox*. Он содержит обычный конструктор и переопределяет три виртуальные функции класса *QSpinBox*.

```
01 #include <QtGui>
02 #include "hexspinbox.h"
03 HexSpinBox::HexSpinBox(QWidget *parent)
04 : QSpinBox(parent)
05 {
06 setRange(0, 255);
07 validator = new QRegExpValidator(QRegExp("[0-9A-Fa-f]{1,8}"), this);
08 }
```

Мы устанавливаем по умолчанию диапазон от 0 до 255 (от 0x00 до 0xFF), 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 вводя 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 со значениями каждого символа в диапазонах от «0» до «9», от «A» до «F» или от «a» до «f».

```
09 QValidator::State HexSpinBox::validate(QString &text, int &pos) const
10 {
11 return validator->validate(text, pos);
12 }
```

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.

```
13 QString HexSpinBox::textFromValue(int value) const
14 {
15 return QString::number(value, 16).toUpper();
16 }
```

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.

```
17 int HexSpinBox::valueFromText(const QString &text) const
18 {
19 bool ok;
20 return text.toInt(&ok, 16);
21 }
```

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.

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 наследует как класс *QWidget*, так и класс, сгенерированный компилятором *uic*.

Естественно, комбинация существующих виджетов может быть также полностью запрограммирована вручную. При любом подходе полученный класс наследует непосредственно *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 мышки. 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 бы 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.

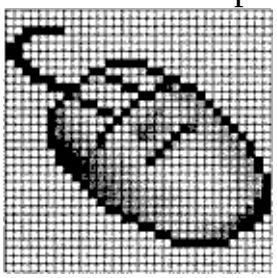


Рис. 5.2. Виджет IconEditor.

Сначала рассмотрим заголовочный файл.

```
01 #ifndef ICONEDITOR_H
02 #define ICONEDITOR_H
03 #include <QColor>
```

```
04 #include <QImage>
05 #include <QWidget>

06 class IconEditor : public QWidget
07 {
08 Q_OBJECT
09 Q_PROPERTY(QColor penColor READ penColor WRITE setPenColor)
10 Q_PROPERTY(QImage iconImage READ iconImage WRITE setIconImage)
11 Q_PROPERTY(int zoomFactor READ zoomFactor WRITE setZoomFactor)

12 public:
13 IconEditor(QWidget *parent = 0);
14 void setPenColor(const QColor &newColor);
15 QColor penColor() const { return curColor; }
16 void setIconImage(const QImage &newImage);
17 QImage iconImage() const { return image; }
18 QSize sizeHint() const;
19 void setZoomFactor(int newZoom);
20 int zoomFactor() const { return zoom; }
```

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.

```
21 protected:
22 void mousePressEvent(QMouseEvent *event);
23 void mouseMoveEvent(QMouseEvent *event);
24 void paintEvent(QPaintEvent *event);

25 private:
26 void setImagePixel(const QPoint &pos, bool opaque);
27 QRect pixelRect(int i, int j) const;
28 QColor curColor;
29 QImage image;
30 int zoom;
31 };
32 #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:*

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

```
02 #include "iconeditor.h"
03 IconEditor::IconEditor(QWidget *parent)
04 : QWidget(parent)
05 {
06 setAttribute(Qt::WA_StaticContents);
07 setSizePolicy(QSizePolicy::Minimum, QSizePolicy::Minimum);
08 curColor = Qt::black;
09 zoom = 8;
10 image = QImage(16, 16, QImage::Format_ARGB32);
11 image.fill(qRgba(0, 0, 0, 0));
12 }
```

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. Переменная *image* имеет тип *QImage*. Мы инициализируем ее областью в 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 «чистого» 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. 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*.

```
13 QSize IconEditor::sizeHint() const
14 {
15 QSize size = zoom * image.size();
16 if (zoom >= 3)
17 size += QSize(1, 1);
18 return size;
19 }
```

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*. Смысл различной политики размеров объясняется в главе 6 («Управление компоновкой»).

```
20 void IconEditor::setPenColor(const QColor &newColor)
21 {
22 curColor = newColor;
23 }
```

Функция *setPenColor()* устанавливает текущий цвет пера. Этот цвет будет использоваться при выводе на экран новых пикселей.

```
24 void IconEditor::setIconImage(const QImage &newImage)
25 {
26 if (newImage != image) {
27 image = newImage.convertToFormat(QImage::Format_ARGB32);
28 update();
29 updateGeometry();
30 }
31 }
```

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 принудительной 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.

```
32 void IconEditor::setZoomFactor(int newZoom)
33 {
34 if (newZoom < 1)
35 newZoom = 1;
36 if (newZoom != zoom) {
37 zoom = newZoom;
38 update();
39 updateGeometry();
40 }
41 }
```

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():*

• 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 невидим 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:

```
42 void IconEditor::paintEvent(QPaintEvent *event)
43 {
44 QPainter painter(this);
45 if (zoom >= 3) {
46 painter.setPen(palette().foreground().color());
47 for (int i = 0; i <= image.width(); ++i)
48 painter.drawLine(zoom * i, 0,
49 zoom * i, zoom * image.height());
50 for (int j = 0; j <= image.height(); ++j)
51 painter.drawLine(0, zoom * j,
52 zoom * image.width(), zoom * j);
53 }

54 for (int i = 0; i < image.width(); ++i) {
55 for (int j = 0; j < image.height(); ++j) {
56 QRect rect = pixelRect(i, j);
57 if (!event->region().intersect(rect).isEmpty()) {
58 QColor color = QColor::fromRgba(image.pixel(i, j));
59 painter.fillRect(rect, color);
60 }
61 }
62 }
63 }
```

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()* имеетследующий 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 *QPoint* 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. Мы можем изменить систему координат в *QPainter*, трансформируя ее такими способами, как смещение, масштабирование, вращение и отсечение. Эти вопросы рассматриваются в главе 8 («Графика 2D и 3D»).

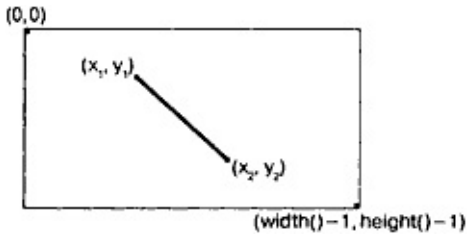


Рис. 5.3. Вычерчивание линии при помощи QPainter.

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: активной, неактивной и нерабочей. 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::QColorGroup*. 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. Мы не выдаем пиксели, которые попадают за пределы данной области, обеспечивая простую оптимизацию.

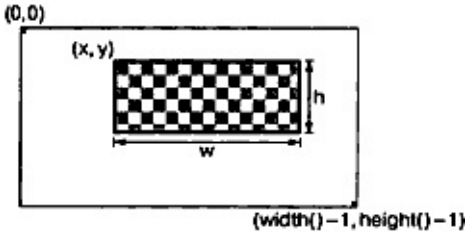


Рис. 5.4. Вычерчивание прямоугольника при помощи QPainter.

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.

```
64 QRect IconEditor::pixelRect(int i, int j) const
65 {
66 if (zoom >= 3) {
67 return QRect(zoom * i + 1, zoom * j + 1, zoom - 1, zoom - 1);
68 } else {
69 return QRect(zoom * i, zoom * j, zoom, zoom);
70 }
71 }
```

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.

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 загораживать линии сетки.

```
72 void IconEditor::mousePressEvent(QMouseEvent *event)
73 {
74 if (event->button() == Qt::LeftButton) {
75 setImagePixel(event->pos(), true);
76 } else if (event->button() == Qt::RightButton) {
77 setImagePixel(event->pos(), false);
78 }
79 }
```

Когда пользователь нажимает кнопку мышки, система генерирует событие «клавиша мышки нажата» (mouse press). Путем переопределения функции *QWidget::mousePressEvent()* мы можем обработать это событие и установить или стереть пиксель изображения, находящийся под курсором мышки.

Если пользователь нажал левую кнопку мышки, 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 мышки, мы также вызываем функцию *setImagePixel(),* но передаем *false* для стирания пикселя.

```
80 void IconEditor::mouseMoveEvent(QMouseEvent *event)
81 {
82 if (event->buttons() & Qt::LeftButton) {
83 setImagePixel(event->pos(), true);
84 } else if (event->buttons() & Qt::RightButton) {
85 setImagePixel(event->pos(), false);
86 }
87 }
```

Функция *mouseMoveEvent()* обрабатывает события «перемещение мышки». По умолчанию эти события генерируются только при нажатой пользователем кнопки мышки. 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 мышки 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().*

```
88 void IconEditor::setImagePixel(const QPoint &pos, bool opaque)
89 {
90 int i = pos.x() / zoom;
```

```
91 int j = pos.y() / zoom;
92 if (image.rect().contains(i, j)) {
93 if (opaque) {
94 image.setPixel(i, j, penColor().rgba());
95 } else {
96 image.setPixel(i, j, qRgba(0, 0, 0, 0));
97 }
98 update(pixelRect(i, j));
99 }
100 }
```

Функция *setImagePixel()* вызывается из *mousePressEvent()* и *mouseMoveEvent()* для установки или стирания пикселя. Параметр *pos* определяет положение мышки на виджете.

На первом этапе надо преобразовать положение мышки из системы координат виджета в систему координат изображения. Это достигается путем деления координат положения мышки *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.

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.

Рис. 5.5. Изменение размеров виджета Qt::WA_StaticContents.

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 *QScrollArea*. 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 в *Qt Designer* некоторыми данными о пользовательском виджете. Впоследствии этот виджет может использоваться в формах, разработанных с помощью *Qt Designer*, но при редактировании или просмотре он отображается просто в виде выбранного встроенного виджета Qt.

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 мышки 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).

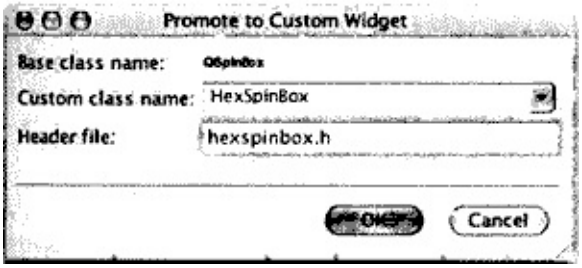


Рис. 5.6. Диалоговое окно для создания пользовательских виджетов Qt Designer.

Недостатками метода продвижения являются недоступность в *Qt Designer* свойств, характерных для пользовательского виджета, и то, что пользовательский виджет представляется в *QtDesigner* 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 мета—объектной 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. Мы 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:

```
01 #include <QDesignerCustomWidgetInterface>
02 class IconEditorPlugin : public QObject,
03 public QDesignerCustomWidgetInterface
04 {
05 Q_OBJECT
06 Q_INTERFACES(QDesignerCustomWidgetInterface)
07 public:
08 IconEditorPlugin(QObject *parent = 0);
09 QString name() const;
10 QString includeFile() const;
11 QString group() const;
12 QIcon icon() const;
13 QString toolTip() const;
14 QString whatsThis() const;
15 bool isContainer() const;
16 QWidget *createWidget(QWidget *parent);
17 };
```

Подкласс *IconEditorPlugin* является фабрикой класса (factory class), который инкапсулирует виджет *IconEditor*. Он является наследником классов *QObject* и *QDesignerCustomWidgetIterface* и использует макрос *Q_INTERFACES()*, указывая компилятору *moc* на то, что второй базовый класс представляет собой подключаемый интерфейс. Его функции применяются *Qt Designer* для создания экземпляров класса и получения информации о нем.

```
01 IconEditorPlugin::IconEditorPlugin(QObject *parent)
02 : QObject(parent)
03 {
04 }
```

IconEditorPlugin имеет тривиальный конструктор.

```
05 QString IconEditorPlugin::name() const
06 {
07 return "IconEditor";
08 }
```

Функция *name()* возвращает имя подключаемого виджета.

```
09 QString IconEditorPlugin::includeFile() const
10 {
11 return "iconeditor.h";
12 }
```

Функция *includeFile()* возвращает имя заголовочного файла для заданного виджета, который инкапсулирован в подключаемом модуле. Заголовочный файл включается в программный код, сгенерированный компилятором *uic*.

```
13 QString IconEditorPlugin::group() const
14 {
15 return tr("Image Manipulation Widgets");
16 }
```

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.

```
17 QIcon IconEditorPlugin::icon() const
18 {
19 return QIcon(":/images/iconeditor.png");
20 }
```

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.

```
21 QString IconEditorPlugin::toolTip() const
22 {
23 return tr("An icon editor widget");
24 }
```

Функция *toolTip()* возвращает всплывающую подсказку, которая появляется, когда мышка находится на пользовательском виджете в панели виджетов *Qt Designer*.

```
25 QString IconEditorPlugin::whatsThis() const
26 {
27 return tr("This widget is presented in Chapter 5 of <i>C++ GUI "
28 "Programming with Qt 4</i> as an example of a custom Qt "
29 "widget.");
30 }
```

Функция *whatsThis()* возвращает текст «What's This?» (что это?) для отображения в *Qt Designer*.

```
31 bool IconEditorPlugin::isContainer() const
32 {
33 return false;
34 }
```

Функция *isContainer()* возвращает *true,* если данный виджет может содержать другие виджеты; в противном случае он 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*.

```
35 QWidget *IconEditorPlugin::createWidget(QWidget *parent)
36 {
37 return new IconEditor(parent);
38 }
```

Функция *createWidget()* вызывается *Qt Designer* для создания экземпляра класса виджета для указанного родительского виджета.

```
39 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 = $(QTDIR)/plugins/designer
```

Файл *.pro* предполагает, что переменная окружения *QTDIR* установлена на каталог, где располагается Qt. Когда вы вводите команду *make* или *nmake* для построения подключаемого модуля, он автоматически устанавливается в каталог *plugins Qt Designer*. Поле 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 все в один подключаемый модуль, реализуя 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). Этот виджет использует двойную буферизацию и также демонстрирует некоторые другие аспекты Qt—программирования, в том числе обработку событий клавиатуры, ручную компоновку виджетов и координатные системы.

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 мышку с нажатой левой кнопкой в другую позицию и освобождая клавишу мышки.

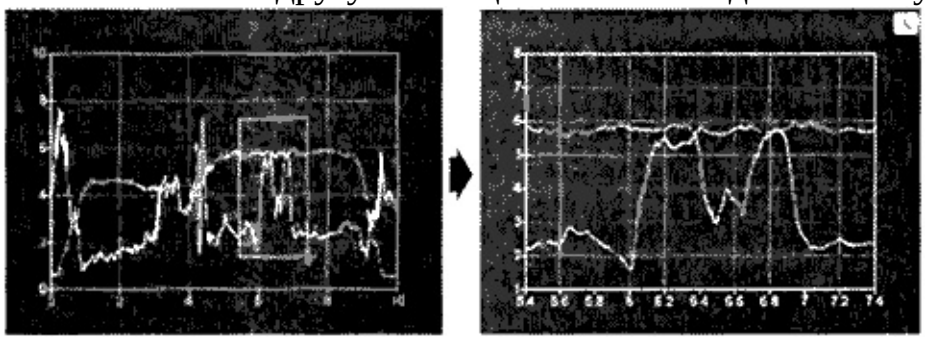


Рис. 5.7. Увеличение изображения виджета Plotter.

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:*

```
01 #ifndef PLOTTER_H
02 #define PLOTTER_H
```

```
03 #include <QMap>
04 #include <QPixmap>
05 #include <QVector>
06 #include <QWidget>
07 class QToolButton;
08 class PlotSettings;

09 class Plotter : public QWidget
10 {
11 Q_OBJECT
12 public:
13 Plotter(QWidget *parent = 0);
14 void setPlotSettings(const PlotSettings &settings);
15 void setCurveData(int id, const QVector<QPointF> &data);
16 void clearCurve(int id);
17 QSize minimumSizeHint() const;
18 QSize sizeHint() const;
19 public slots:
20 void zoomIn();
21 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.

```
22 protected:
23 void paintEvent(QPaintEvent *event);
24 void resizeEvent(QResizeEvent *event);
25 void mousePressEvent(QMouseEvent *event);
26 void mouseMoveEvent(QMouseEvent *event);
27 void mouseReleaseEvent(QMouseEvent *event);
28 void keyPressEvent(QKeyEvent *event);
29 void wheelEvent(QWheelEvent *event);
```

В защищенной секции класса мы объявляем все обработчики событий *QWidget*, которые хотим переопределить.

```
30 private:
31 void updateRubberBandRegion();
32 void refreshPixmap();
33 void drawGrid(QPainter *painter);
34 void drawCurves(QPainter *painter);
35 enum { Margin = 50 };
36 QToolButton *zoomInButton;
```

```
37 QToolButton *zoomOutButton;

38 QMap<int, QVector<QPointF> > curveMap;
39 QVector<PlotSettings> zoomStack;
40 int curZoom;
41 bool rubberBandIsShown;
42 QRect rubberBandRect;
43 QPixmap pixmap;
44 };
```

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.

```
45 class PlotSettings
46 {
47 public:
48 PlotSettings();

49 void scroll(int dx, int dy);
50 void adjust();
51 double spanX() const { return maxX - minX; }
52 double spanY() const { return maxY - minY; }

53 double minX;
54 double maxX;
55 int numXTicks;
56 double minY;
57 double maxY;
58 int numYTicks;

59 private:
60 static void adjustAxis(double &min, double &max, int &numTicks);
61 };
62 #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 6 отметок по оси *x*. Это упростит расчеты в будущем.

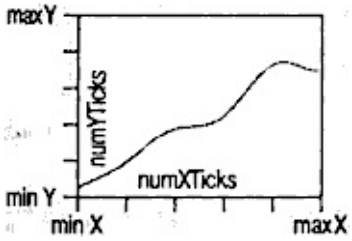


Рис. 5.8. Переменные—члены настроек графика PlotSettings.

Теперь давайте рассмотрим файл реализации:

```
001 #include <QtGui>
002 #include <cmath>
003 #include "plotter.h"
```

Мы включаем необходимые заголовочные файлы и импортируем все символы пространства имен *std* в глобальное пространство имен. Это позволяет нам получать доступ к функциям, объявленным в *<cmath>*, без указания префикса *std::* (например, *floor()* вместо *std::floor()*).

```
004 Plotter::Plotter(QWidget *parent)
005 : QWidget(parent)
006 {
007 setBackgroundRole(QPalette::Dark);
008 setAutoFillBackground(true);
009 setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
010 setFocusPolicy(Qt::StrongFocus);
011 rubberBandIsShown = false;

012 zoomInButton = new QToolButton(this);
013 zoomInButton->setIcon(QIcon(":/images/zoomin.png"));
014 zoomInButton->adjustSize();
015 connect(zoomInButton, SIGNAL(clicked()), this, SLOT(zoomIn()));

016 zoomOutButton = new QToolButton(this);
017 zoomOutButton->setIcon(QIcon(":/images/zoomout.png"));
018 zoomOutButton->adjustSize();
019 connect(zoomOutButton, SIGNAL(clicked()), this, SLOT(zoomOut()));

020 setPlotSettings(PlotSettings());
021 }
```

Вызов *setBackgroundRole()* указывает *QWidget* на необходимость использования для цвета стирания виджета «темного» компонента палитры вместо компонента «window» (окно). 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 ответственен 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 нажития клавиш. Виджет *Plotter* понимает несколько клавиш: «+» для увеличения изображения, «—» для уменьшения изображения и клавиш стрелок для прокрутки вверх, вниз, влево и вправо.

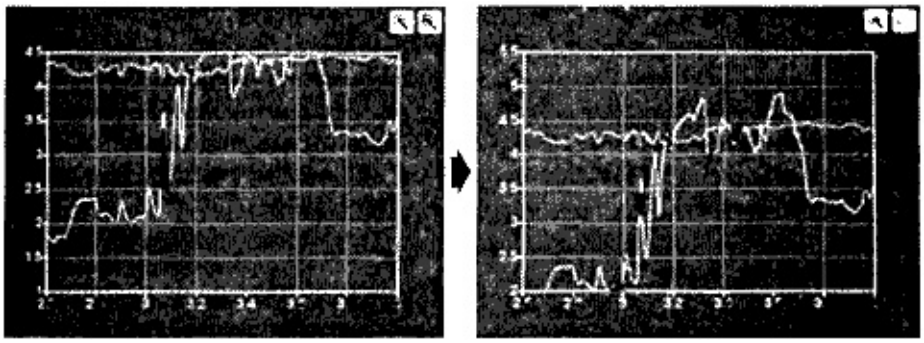


Рис. 5.9. Скроллинг виджета Plotter.

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:

```
<!DOCTYPE RCC><RCC version="1.0">
<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, необходимо явно задавать родительский виджет кнопок, передавая *this* конструктору *QPushButton.*

Вызов в конце функции *setPlotSettings()* завершает инициализацию.

```
022 void Plotter::setPlotSettings(const PlotSettings &settings)
023 {
024 zoomStack.clear();
025 zoomStack.append(settings);
026 curZoom = 0;
027 zoomInButton->hide();
028 zoomOutButton->hide();
029 refreshPixmap();
```

```
030 }
```

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 мы 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.

```
031 void Plotter::zoomOut()
032 {
033 if (curZoom > 0) {
034 --curZoom;
035 zoomOutButton->setEnabled(curZoom > 0);
036 zoomInButton->setEnabled(true);
037 zoomInButton->show();
038 refreshPixmap();
039 }
040 }
```

Слот *zoomOut()* уменьшает масштаб диаграммы, если она отображена крупным планом. Он уменьшает на единицу текущий масштаб изображения и включает 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()*.

```
041 void Plotter::zoomIn()
042 {
043 zoomInButton->setEnabled(curZoom< zoomStack.count() - 1);
044 if (curZoom < zoomStack.count() - 1) {
045 ++curZoom;
046 zoomOutButton->setEnabled(true);
047 zoomOutButton->show();
048 refreshPixmap();
049 }
```

```
050 }
```

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 вглубь стека масштабов изображения, включает или выключает кнопку Zoom In взависимости 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.

```
051 void Plotter::setCurveData(int id, const QVector<QPointF> &data)
052 {
053 curveMap[id] = data;
054 refreshPixmap();
055 }
```

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> >*.

```
056 void Plotter::clearCurve(int id)
057 {
058 curveMap.remove(id);
059 refreshPixmap();
060 }
```

Функция *clearCurve()* удаляет заданную кривую из *curveMap.*

```
061 QSize Plotter::minimumSizeHint() const
062 {
063 return QSize(6 * Margin, 4 * Margin);
064 }
```

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.

```
065 QSize Plotter::sizeHint() const
066 {
067 return QSize(12 * Margin, 8 * Margin);
068 }
```

В функции *sizeHint()* мы возвращаем «идеальный» размер относительно константы *Margin*, причем горизонтальный и вертикальный компоненты 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.

```
069 void Plotter::paintEvent(QPaintEvent * /* event */)
070 {
071 QStylePainter painter(this);
072 painter.drawPixmap(0, 0, pixmap);
073 if (rubberBandIsShown) {
074 painter.setPen(palette().light().color());
075 painter.drawRect(rubberBandRect.normalized()
076 .adjusted(0, 0, -1, -1));
077 }

078 if (hasFocus()) {
079 QStyleOptionFocusRect option;
080 option.initFrom(this);
081 option.backgroundColor = palette().dark().color();
082 painter.drawPrimitive(QStyle::PE_FrameFocusRect, option);
083 }
084 }
```

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*, *QMotifStyle*, *QCDEStyle*, *QMacStyle* и *OPlastiqueStyle*. Все эти 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. 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.

```
085 void Plotter::resizeEvent(QResizeEvent * /* event */ )
086 {
087 int x= width() - (zoomInButton->width()
088 + zoomOutButton->width() + 10);
089 zoomInButton->move(x, 5);
090 zoomOutButton->move(x + zoomInButton->width() + 5, 5);
091 refreshPixmap();
092 }
```

При всяком изменении размера виджета *Plotter* Qt генерирует событие «изменение размера». Здесь мы переопределяем функцию *resizeEvent()* для размещения кнопок Zoom In и Zoom Out 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.

```
093 void Plotter::mousePressEvent(QMouseEvent *event)
094 {
095 QRect rect(Margin, Margin,
096 width() - 2 * Margin, height() - 2 * Margin);
097 if (event->button() == Qt::LeftButton) {
098 if (rect.contains(event->pos())) {
099 rubberBandIsShown = true;
100 rubberBandRect.setTopLeft(event->pos());
101 rubberBandRect.setBottomRight(event->pos());
102 updateRubberBandRegion();
103 setCursor(Qt::CrossCursor);
104 }
105 }
106 }
```

Когда пользователь нажимает левую кнопку мышки, 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 мышки, поставить в очередь событие рисования для вычерчивания резиновой ленты и изменить изображение курсора мышки 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 углов. Здесь мы используем формат с парой точек. То место, где пользователь первый раз щелкнул мышкой, 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 мышки:

• *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.

```
107 void Plotter::mouseMoveEvent(QMouseEvent *event)
108 {
109 if (rubberBandIsShown) {
110 updateRubberBandRegion();
111 rubberBandRect.setBottomRight(event->pos());
112 updateRubberBandRegion();
113 }
114 }
```

Когда пользователь перемещает курсор мышки 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 мышку 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.

```
115 void Plotter::mouseReleaseEvent(QMouseEvent *event)
116 {
117 if ((event->button() == Qt::LeftButton) &&
118 rubberBandIsShown) {
119 rubberBandIsShown = false;
120 updateRubberBandRegion();
121 unsetCursor();

122 QRect rect = rubberBandRect.normalized();
123 if (rect.width() < 4 || rect.height() < 4)
124 return;
125 rect.translate(-Margin, -Margin);

126 PlotSettings prevSettings = zoomStack[curZoom];
127 PlotSettings settings;
128 double dx = prevSettings.spanX() / (width() - 2 * Margin);
130 double dy = prevSettings.spanY() / (height() - 2 * Margin);

131 settings.minX = prevSettings.minX + dx * rect.left();
132 settings.maxX = prevSettings.minX + dx * rect.right();
133 settings.minY = prevSettings.maxY - dy * rect.bottom();
134 settings.maxY = prevSettings.maxY - dy * rect.top();
135 settings.adjust();

136 zoomStack.resize(curZoom + 1);
137 zoomStack.append(settings);
138 zoomIn();
139 }
140 }
```

Когда пользователь отпускает левую кнопку мышки, 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 мышкой 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 обеим осям. Эта ситуация отражена на рис. 5.10 и 5.11.

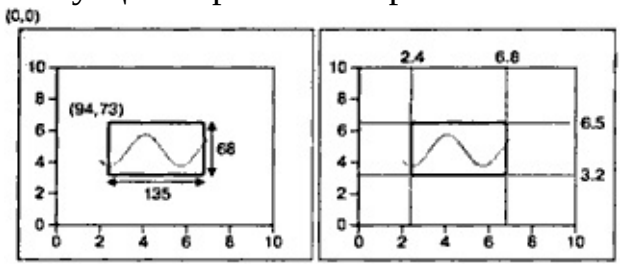


Рис. 5.10. Преобразование прямоугольника резиновой ленты из системы координат виджета в систему координат построителя графиков.

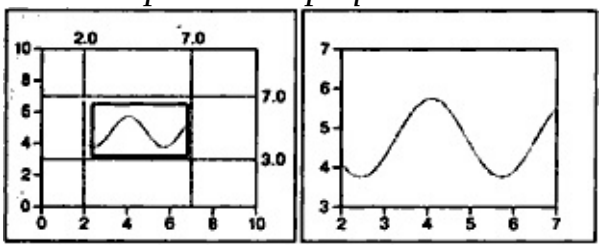


Рис. 5.11. Настройка прямоугольника резиновой ленты в системе координат построителя графиков и увеличение изображения.

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.

```
141 void Plotter::keyPressEvent(QKeyEvent *event)
142 {
143 switch (event->key()) {
144 case Qt::Key_Plus:
145 zoomIn();
146 break;
147 case Qt::Key_Minus:
148 zoomOut();
149 break;
150 case Qt::Key_Left:
151 zoomStack[curZoom].scroll(-1, 0);
152 refreshPixmap();
153 break;
154 case Qt::Key_Right:
155 zoomStack[curZoom].scrol1(+1, 0);
156 refreshPixmap();
157 break;
158 case Qt::Key_Down:
159 zoomStack[curZoom].scroll(0, -1);
```

```
160 refreshPixmap();
161 break;
162 case Qt::Key_Up:
163 zoomStack[curZoom].scroll(0, +1);
164 refreshPixmap();
165 break;
166 default:
167 QWidget::keyPressEvent(event);
168 }
169 }
```

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()*.

```
170 void Plotter::wheelEvent(QWheelEvent *event)
171 {
172 int numDegrees= event->delta() / 8;
173 int numTicks = numDegrees / 15;
174 if (event->orientation() == Qt::Horizontal) {
175 zoomStack[curZoom].scroll(numTicks, 0);
176 } else {
177 zoomStack[curZoom].scroll(0, numTicks);
178 }
179 refreshPixmap();
180 }
```

События колесика мышки возникают при повороте колесика мышки. В большинстве мышек предусматривается колесико для перемещения по вертикали, но некоторые мышки имеют также колесико для перемещения по горизонтали. Qt поддерживает оба вида колесиков. События колесика мышки передаются виджету, на котором находится фокус. Функция *delta()* возвращает перемещение колесика, выраженное в восьмых долях градуса. Обычно шаг работы колесика мышки 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 мышки получило для продвижения по полосе прокрутки. При использовании нами *QScrollArea* (рассматривается в главе 6) с полосами прокрутки *QScrollArea* автоматически управляет событиями колесика мышки и нам не приходится самим переопределять функцию *wheelEvent()*.

Этим завершается реализация обработчиков событий. Теперь давайте рассмотрим закрытые функции.

```
181 void Plotter::updateRubberBandRegion()
182 {
183 QRect rect = rubberBandRect.normalized();
184 update(rect.left(), rect.top(), rect.width(), 1);
```

```
185 update(rect.left(), rect.top(), 1, rect.height());
186 update(rect.left(), rect.bottom(), rect.width(), 1);
187 update(rect.right(), rect.top(), 1, rect.height());
188 }
```

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). Для рисования резиновой ленты в Qt предусмотрен класс *QRubberBand*, однако в нашем случае ручное кодирование обеспечило более тонкое управление.

```
189 void Plotter::refreshPixmap()
190 {
191 pixmap = QPixmap(size());
192 pixmap.fill(this, 0, 0);
193 QPainter painter(&pixmap);
194 painter.initFrom(this);
195 drawGrid(&painter);
196 drawCurves(&painter);
197 update();
198 }
```

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 выравнить 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()*.

```
199 void Plotter::drawGrid(QPainter *painter)
200 {
201 QRect rect(Margin, Margin,
202 width() - 2 * Margin, height() - 2 * Margin);
203 if (!rect.isValid())
204 return;
205 PlotSettings settings = zoomStack[curZoom];
206 QPen quiteDark = palette().dark().color().light();
207 QPen light = palette().light().color();

208 for (int i = 0; i <= settings.numXTicks; ++i) {
```

```
209 int x = rect.left() + (i * (rect.width() - 1)
210 / settings.numXTicks);
211 double label = settings.minX + (i * settings.spanX()
212 / settings.numXTicks);
213 painter->setPen(quiteDark);
214 painter->drawLine(x, rect.top(), x, rect.bottom());
215 painter->setPen(light);
216 painter->drawLine(x, rect.bottom(), x, rect.bottom() + 5);
217 painter->drawText(x - 50, rect.bottom() + 5, 100, 15,
218 Qt::AlignHCenter | Qt::AlignTop,
219 QString::number(label));
220 }

221 for (int j = 0; j <= settings.numVTicks; ++j) {
222 int y = rect.bottom() - (j * (rect.height() - 1)
223 / settings.numYTicks);
224 double label = settings.minY + (j * settings.spanY()
225 / settings.numYTicks);
226 painter->setPen(quiteDark);
227 painter->drawLine(rect.left(), y, rect.right(), y);
228 painter->setPen(light);
229 painter->drawLine(rect.left() - 5, y, rect.left(), y);
230 painter->drawText(rect.left() - Margin, y - 10, Margin - 5, 20,
231 Qt::AlignRight | Qt::AlignVCenter,
232 QString::number(label));
233 }
234 painter->drawRect(rect.adjusted(0, 0, -1, -1));
235 }
```

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* проводит 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 обеиз 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.

```
236 void Plotter::drawCurves(QPainter *painter)
237 {
238 static const QColor colorForIds[6] = {
239 Qt::red, Qt::green, Qt::blue, Qt::cyan, Qt::magenta, Qt::yellow };
240 PlotSettings settings = zoomStack[curZoom];
241 QRect rect(Margin, Margin,
```

```
242 width() - 2 * Margin, height() - 2 * Margin);
243 if (!rect.isValid())
244 return;

245 painter->setClipRect(rect.adjusted(+1, +1, -1, -1));
246 QMapIterator<int, QVector<QPointF> > i(curveMap);
247 while (i.hasNext()) {
248 i.next();
249 int id = i.key();
250 const QVector<QPointF> &data = i.value();
251 QPolygonF polyline(data.count());
252 for (int j = 0; j < data.count(); ++j) {
253 double dx = data[j].x() - settings.minX;
254 double dy = data[j].y() - settings.minY;
255 double x = rect.left() + (dx * (rect.width() - 1)
256 / settings.spanX());
257 double y = rect.bottom() - (dy * (rect.height() - 1)
258 / settings.spanY());
259 polyline[j] = QPointF(x, y);
260 }
261 painter->setPen(colorForIds[uint(id) % 6]);
262 painter->drawPolyline(polyline);
263 }
264 }
```

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 мы выполняем цикл по ее точкам *QPointF*. Функция *key()* позволяет получить идентификатор кривой, а функция *value()* — данные соответствующей кривой в виде вектора *QVector<QPointF>*. Внутри цикла *for* производятся 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*.

```
265 PlotSettings::PlotSettings()
266 {
267 minX = 0.0;
268 maxX = 10.0;
269 numXTicks = 5;
```

```
270 minY = 0.0;
271 maxY = 10.0;
272 numYTicks = 5;
273 }
```

Конструктор *PlotSettings* инициализирует обе оси координат диапазоном от 0 до 10 с пятью отметками.

```
274 void PlotSettings::scroll(int dx, int dy)
275 {
276 double stepX = spanX() / numXTicks;
277 minX += dx * stepX;
278 maxX += dx * stepX;
279 double stepY = spanY() / numYTicks;
280 minY += dy * stepY;
281 maxY += dy *stepY;
282 }
```

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()*.

```
283 void PlotSettings::adjust()
284 {
285 adjustAxis(minX, maxX, numXTicks);
286 adjustAxis(minY, maxY, numYTicks);
287 }
```

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.

```
288 void PlotSettings::adjustAxis(double &min, double &max, int &numTicks)
289 {
290 const int MinTicks = 4;
291 double grossStep = (max - min) / MinTicks;
292 double step = pow(10.0, floor(log10(grossStep)));
293 if (5 * step < grossStep) {
294 step *= 5;
295 } else if (2* step < grossStep) {
296 step *= 2;
297 }
298 numTicks = int (ceil(max / step) - floor(min / step));
299 if (numTicks < MinTicks)
300 numTicks = MinTicks;
301 min = floor(min / step) * step;
302 max = ceil(max / step) * step;
303 }
```

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 так далее), 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 должнытщательно 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 число вида 10^n, меньшее 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…; затем мы округляем это значение до 2 и получаем $10^2 = 100$ в качестве кандидата на значениешага 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 сверху 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 дает 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* (ISBN 0—12—286166—3).

Данная глава является последней в части I. 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 можно построить виджет на основе существующих виджетов, 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

Глава 6. Управление компоновкой

Каждому размещаемому в форме виджету необходимо задать соответствующие размер и позицию. Qt содержит несколько классов, обеспечивающих компоновку виджетов на форме: *QHBoxLayout*, *QVBoxLayout*, *QGridLayout* и *QStackLayout*. 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 *QWorkspace*. Общая черта этих классов — обеспечение гибкой компоновки виджетов, которой может управлять пользователь. Например, *QSplitter* обеспечивает наличие разделительной линии, которую пользователь может передвигать для изменения размеров виджетов, а *QWorkspace* 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, они также рассматриваются 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 (найти файл), показанное на рис. 6.1.

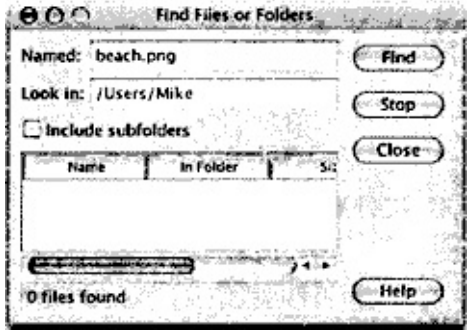


Рис. 6.1. Окно диалога Find File.

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:

```
01 FindFileDialog::FindFileDialog(QWidget *parent)
02 : QDialog(parent)
03 {
04 namedLabel->setGeometry(9, 9, 50, 25);
05 namedLineEdit->setGeometry(65, 9, 200, 25);
06 lookInLabel->setGeometry(9, 40, 50, 25);
07 lookInLineEdit->setGeometry(65, 40, 200, 25);
08 subfoldersCheckBox->setGeometry(9, 71, 256, 23);
09 tableWidget->setGeometry(9, 100, 256, 100);
10 messageLabel->setGeometry(9, 206, 256, 25);
11 findButton->setGeometry(271, 9, 85, 32);
12 stopButton->setGeometry(271, 47, 85, 32);
13 closeButton->setGeometry(271, 84, 85, 32);
14 helpButton->setGeometry(271, 199, 85, 32);
15 setWindowTitle(tr("Find Files or Folders"));
16 setFixedSize(365, 240);
17 }
```

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 пропорциональными размеру окна, а не жестко кодируются в программе. Это может достигаться путем переопределения функции формы *resizeEvent()* для установки геометрических размеров своих дочерних виджетов:

```
01 FindFileDialog::FindFileDialog(QWidget *parent)
02 : QDialog(parent)
03 {
04 SetMinimumSize(265, 190);
05 resize(365, 240);
06 }

07 void FindFileDialog::resizeEvent(QResizeEvent * /* event */)
08 {
09 int extraWidth = width() - minimumWidth();
10 int extraHeight = height() - minimumHeight();
11 namedLabel->setGeometry(9, 9, 50, 25);
12 namedLineEdit->setGeometry(65, 9, 100 + extraWidth, 25);
13 lookInLabel->setGeometry(9, 40, 50, 25);
14 lookInLineEdit->setGeometry(65, 40, 100 + extraWidth, 25);
15 subfoldersCheckBox->setGeometry(9, 71, 156 + extraWidth, 23);
16 tableWidget->setGeometry(9, 100, 156 + extraWidth, 50 + extraHeight);
17 messageLabel->setGeometry(9, 156 + extraHeight, 156 + extraWidth, 25);
18 findButton->setGeometry(171 + extraWidth, 9, 85, 32);
19 stopButton->setGeometry(171 + extraWidth, 47, 85, 32);
20 closeButton->setGeometry(171 + extraWidth, 84, 85, 32);
21 helpButton->setGeometry(171 + extraWidth, 149 + extraHeight, 85, 32);
22 }
```

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.

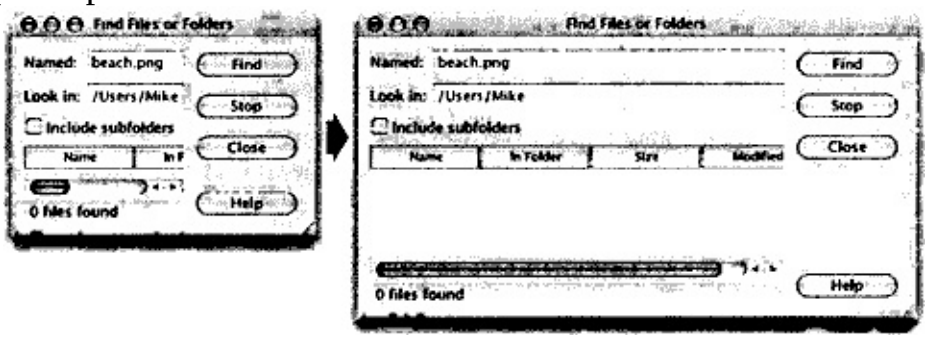


Рис. 6.2. Изменение размеров диалогового окна, допускающего изменение своих размеров.

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.

Существует три наиболее важных менеджера компоновки: *QHBoxLayout*, *QVBoxLayout* и *QGridLayout*. Эти классы наследуют *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:

```
01 FindFileDialog::FindFileDialog(QWidget *parent)
02 : QDialog(parent)
03 {
04 QGridLayout *leftLayout = new QGridLayout;
05 leftLayout->addWidget(namedLabel, 0, 0);
06 leftLayout->addWidget(namedLineEdit, 0, 1);
07 leftLayout->addWidget(lookInLabel, 1, 0);
08 leftLayout->addWidget(lookInLineEdit, 1, 1);
09 leftLayout->addWidget(subfoldersCheckBox, 2, 0, 1, 2);
10 leftLayout->addWidget(tableWidget, 3, 0, 1, 2);
11 leftLayout->addWidget(messageLabel, 4, 0, 1, 2);

12 QVBoxLayout *rightLayout = new QVBoxLayout;
13 rightLayout->addWidget(findButton);
14 rightLayout->addWidget(stopButtpn);
15 rightLayout->addWidget(closeButton);
16 rightLayout->addStretch();
17 rightLayout->addWidget(helpButton);

18 QHBoxLayout *mainLayout = new QHBoxLayout;
19 mainLayout->addLayout(leftLayout);
20 mainLayout->addLayout(rightLayout);
21 setLayout(mainLayout);
22 setWindowTitle(tr("Find Files or Folders"));
23 }
```

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 промежуток между дочерними

виджетами устанавливаются в значения по умолчанию, которые зависят от текущего стиля виджета; они могут быть изменены, если использовать функции *QLayout::setMargin()* и *QLayout::setSpacing()*.

Такое же диалоговое окно можно было бы создать с помощью визуальных средства 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.

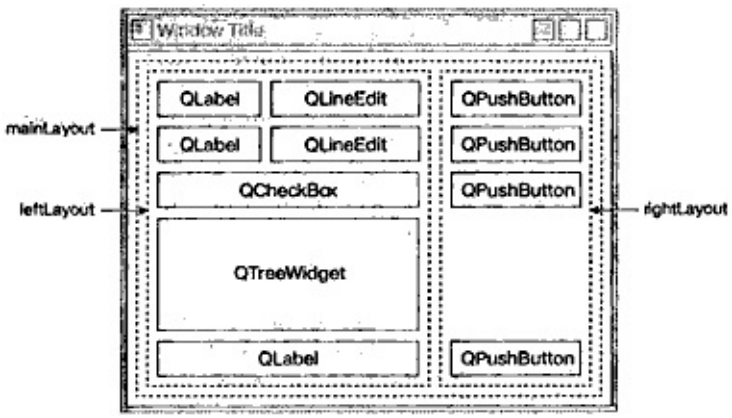


Рис. 6.3. Компоновка диалогового окна Find File.

Применение *QHBoxLayout* и *QVBoxLayout* достаточно очевидное, однако с *QGridLayout* дело обстоит несколько сложнее. Менеджер *QGridLayout* работает с двухмерной 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 *addWidget()* имеют следующий формат:

layout->addWidget(виджет, строка, столбец, колСтрок, колСтолбцов);

Здесь *виджет* является дочерним виджетом, который вставляется в менеджер компоновки, *(строка, столбец)* — коррдинаты верхней левой ячейки, занимаемой виджетом, *колСтрок* — количество строк, занимаемое виджетом, и *колСтолбцов* — количество столбцов, занимаемое виджетом. Если параметры *колСтрок* и *колСтолбцов* не заданы, они принимают значение по умолчанию, равное 1.

Вызов *addStretch()* говорит менеджеру компоновки о необходимости выделения свободного пространства в данной точке. Добавив элемент распорки, мы заставляем менеджер 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 размервиджета является его предпочитаемым размером, но виджет может при необходимости сжиматься или растягиваться;

• *Expanding* (расширяемый) означает, что виджет может сжиматься или растягиваться, но впервую 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.

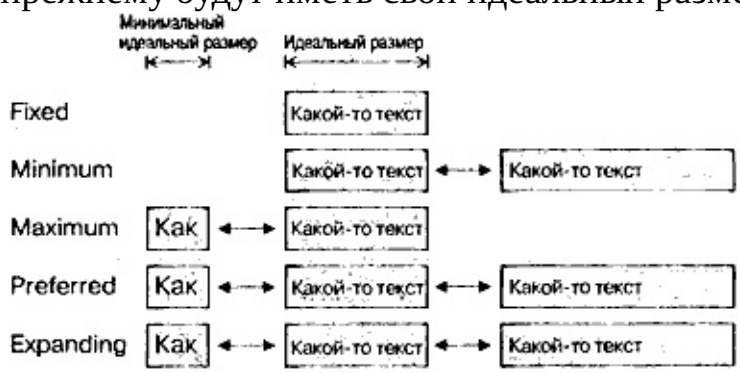


Рис. 6.4. Смысл различных политик размеров.

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.

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* со встроенным *QStackedLayout*.

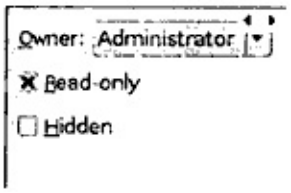


Рис. 6.5. QStackedLayout.

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()*.

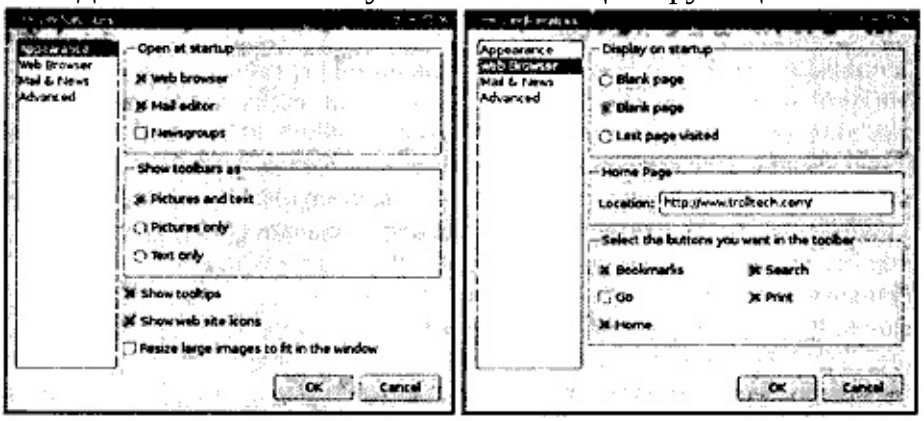


Рис. 6.6. Две страницы диалогового окна Preferences.

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:

```
01 PreferenceDialog::PreferenceDialog(QWidget *parent)
02 : QDialog(parent)
03 {
04 listWidget = new QListWidget;
05 listWidget->addItem(tr("Web Browser"));
06 listWidget->addItem(tr("Mail & News"));
07 listWidget->addItem(tr("Advanced"));
08 listWidget->addItem(tr("Appearance"));

09 stackedLayout = new QStackedLayout;
10 stacked Layout->addWidget(appearancePage);
```

```
11 stackedLayout->addWidget(webBrowserPage);
12 stackedLayout->addWidget(mailAndNewsPage);
13 stackedLayout->addWidget(advancedPage);

14 connect(listWidget, SIGNAL(currentRowChanged(int)).
15 stackedLayout, SLOT(setCurrentIndex(int)));
16 listWidget->setCurrentRow(0);
17 }
```

Мы создаем *QListWidget* и заполняем его названиями страниц. Затем мы 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 шаблона «Dialog» или «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 мышки 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*.

Разделители

Разделитель *QSplitter* представляет собой виджет, который содержит другие виджеты. Виджеты в разделителе 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.

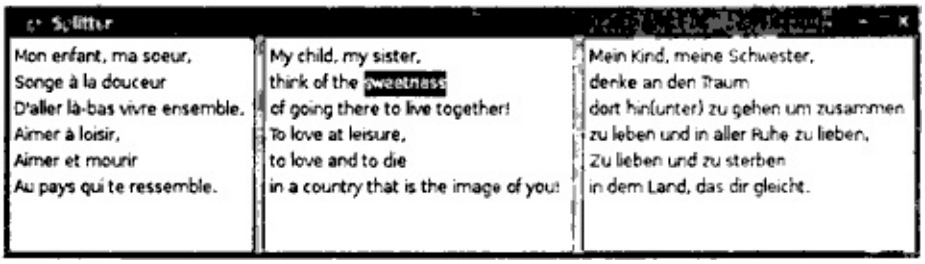


Рис. 6.7. Приложение Splitter.

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:

```
01 int main(int argc, char *argv[])
02 {
03 QApplication app(argc, argv);
04 QTextEdit *editor1 = new QTextEdit;
05 QTextEdit *editor2 = new QTextEdit;
06 QTextEdit *editor3 = new QTextEdit;
07 QSplitter splitter(Qt::Horizontal);
08 splitter.addWidget(editor1);
09 splitter.addWidget(editor2);
10 splitter.addWidget(editor3);
11 splitter.show();
12 return app.exec();
13 }
```

Этот пример состоит из трех полей редактирования *QTextEdit*, расположенных горизонтально в виджете *QSplitter*. В отличие от менеджеров компоновки, которые просто размещают в форме дочерние виджеты, а сами не имеют визуального представления, *QSplitter* наследует *QWidget* и может использоваться как любой другой виджет.

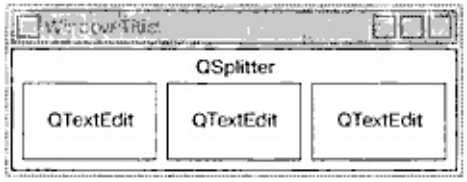


Рис. 6.8. Виджеты приложения Splitter.

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 горизонтального *QSplitter*, который содержит справа от себя вертикальный *QSplitter*.

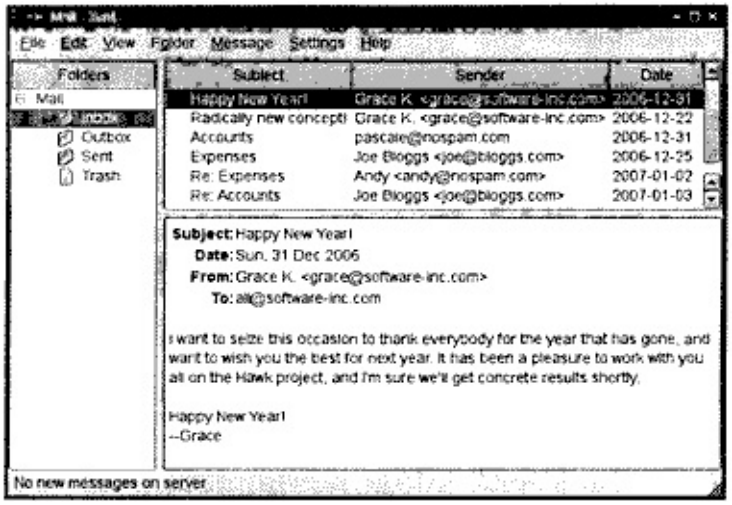


Рис. 6.9. Приложение Mail Client в системе Mac OS X.

Ниже приводится программный код конструктора подкласса *QMainWindow* приложения Mail Client:

```
01 MailClient::MailClient()
02 {
03 …
04 rightSplitter = new QSplitter(Qt::Vertical);
05 rightSplitter->addWidget(messagesTreeWidget);
06 rightSplitter->addWidget(textEdit);
07 rightSplitter->setStretchFactor(1, 1);
08 mainSplitter = new QSplitter(Qt::Horizontal);
09 mainSplitter->addWidget(foldersTreeWidget);
10 mainSplitter->addWidget(rigntSplitter);
11 mainSplitter->setStretchFactor(1, 1);
12 setCentralWidget(mainSplitter);
13 setWindowTitle(tr("Mail Client"));
14 readSettings();
15 }
```

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.

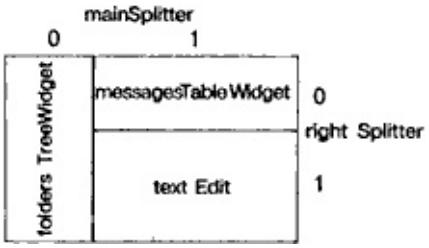


Рис.6.10. Индексация разделителя в приложении Mail Client.

Первый вызов *setStretchFactor()* делаем для *rightSplitter,* устанавливая виджет в позицию 1 *(textEdit)* и коэффициент растяжения на 1. Второй вызов *setStretcnFactor()* делаем 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:

```
01 void MailClient::writeSettings()
02 {
03 QSettings settings("Software Inc.", "Mail Client");
04 settings.beginGroup("mainWindow");
05 settings.setValue("size", size());
06 settings.setValue("mainSplitter", mainSplitter->saveState());
07 settings.setValue("rightSplitter", rightSplitter->saveState());
08 settings.endGroup();
09 }
```

Ниже приводится соответствующая функция по чтению настроек *readSettings():*

```
01 void MailClient::readSettings()
02 {
03 QSettings settings("Software Inc.", "Mail Client");
04 settings.beginGroup("mainWindow");
05 resize(settings.value("size", QSize(480, 360)).toSize());
06 mainSplitter->restoreState(
07 settings.value("mainSplitter").toByteArray());
08 rightSplitter->restoreState(
09 settings.value("rightSplitter").toByteArray());
10 settings.endGroup();
11 }
```

Разделитель *QSplitter* полностью поддерживается *Qt Designer.* 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.

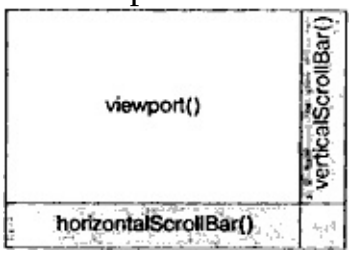


Рис. 6.11. Виджеты, составляющие область с прокруткой QScrollArea.

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, мы можем написать такую программу:

```
01 int main(int argc, char *argv[])
02 {
03 QApplication app(argc, argv);
04 IconEditor *iconEditor = new IconEditor;
05 iconEditor->setIconImage(QImage(":/images/mouse.png"));
06 QScrollArea scrollArea;
07 scrollArea.setWidget(iconEditor);
08 scrollArea.viewport()->setBackgroundRole(QPalette::Dark);
09 scrollArea.viewport()->setAutoFillBackground(true);
10 scrollArea.setWindowTitle(QObject::tr("Icon Editor"));
11 scrollArea.show();
12 return app.exec();
13 }
```

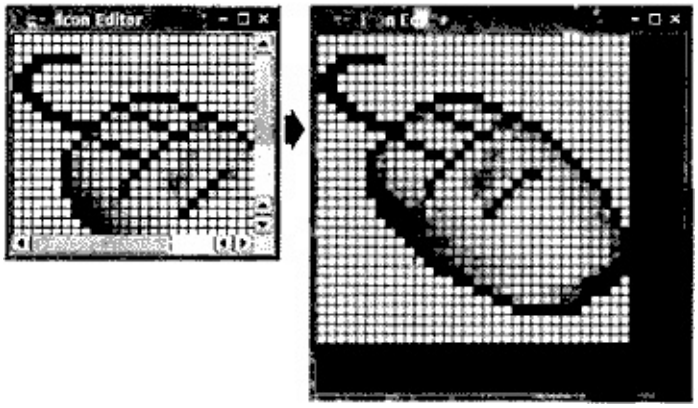


Рис. 6.12. Изменение размеров области с прокруткой QScrollArea.

QScrollArea 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.

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 виджеты и панели инструментов

Прикрепляемыми являются виджеты, которые могут крепиться к определенным областям главного окна приложения *QMainWindow* или быть независимыми «плавающими» окнами. *QMainWindow* имеет четыре области крепления таких виджетов: 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 виджеты представляют собой экземпляры класса *QDockWidget.*

Каждый прикрепляемый виджет 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 виджета. Любые комбинации этих возможностей можно отключать с помощью вызова *QDockWidget::setFeatures().*

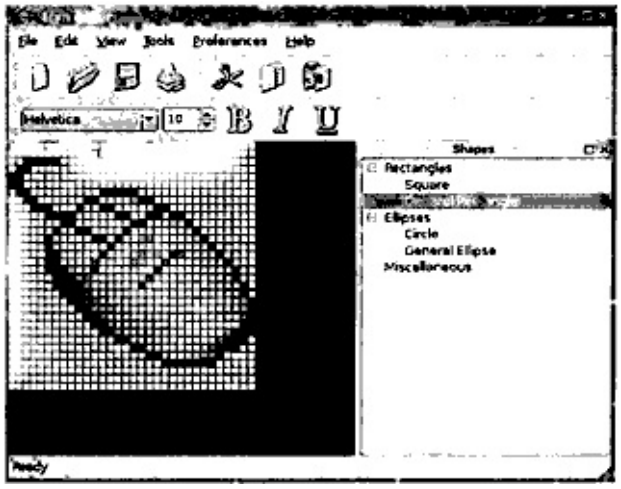


Рис. 6.13. QMainWindow с прикрепленным виджетом.

В ранних версиях Qt панели инструментов рассматривались как прикрепляемые виджеты, 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 *QDockWindow.*

Рис. 6.14. Области крепления виджетов и области панелей инструментов QMainWindow.

Углы, обозначенные пунктирными линиями, могут принадлежать обеим 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->setWidget(treeWidget);
shapesDockWidget->setAllowedAreas(Qt::LeftDockWidgetArea
| Qt::RightDockWidgetArea);
addDockWidget(Qt::RightDockWidgetArea, shapesDockWidget);
```

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.

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->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 запуске приложения, мы можем написать почти такой же программный код, как для сохранения состояния разделителя *QSplitter*, используя функции класса *QMainWindow saveState()* и *restoreState():*

```
01 void MainWindow::writeSettings()
02 {
03 QSettings settings("Software Inc.", "Icon Editor");
04 settings.beginGroup("mainWindow");
05 settings.setValue("size", size());
06 settings.setValue("state", saveState());
07 settings.endGroup();
08 }

09 void MainWindow::readSettings()
10 {
11 QSettings settings("Software Inc.", "Icon Editor");
12 settings.beginGroup("mainWindow");
13 resize(settings.value("size").toSize());
14 restoreState(settings.value("state").toByteArray());
15 settings.endGroup();
16 }
```

Наконец, *QMainWindow* обеспечивает контекстное меню, в котором представлены все прикрепляемые окна и панели инструментов. 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 *QWorkspace* и путем представления каждого документа в виде дочернего окна *QWorkspace*.

Обычно MDI—приложения содержат пункт главного меню Windows (окна) с командами по управлению окнами 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 (редактор), показанное на рис. 6.16.

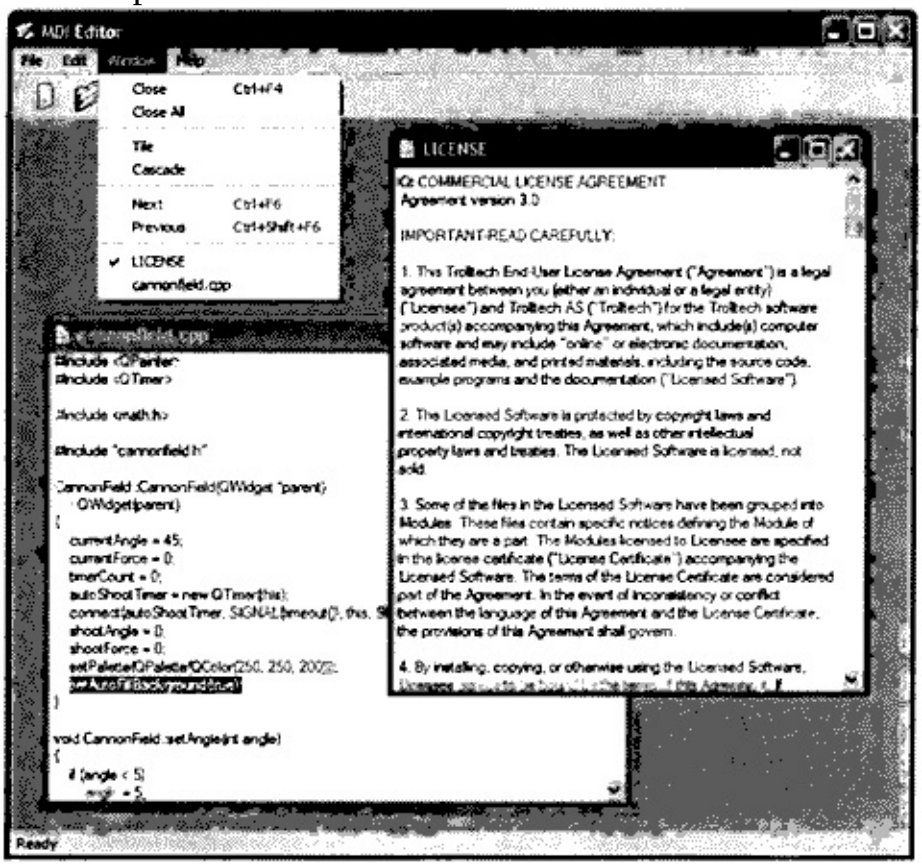


Рис. 6.16. MDI—приложение Editor.

Это приложение состоит из двух классов: *MainWindow* и *Editor*. Его программный код находится на компакт-диске, и поскольку большая часть его либо совпадает, либо очень похожа на программный код приложения Электронная таблица из [части I](), здесь мы представим только новый программный код.

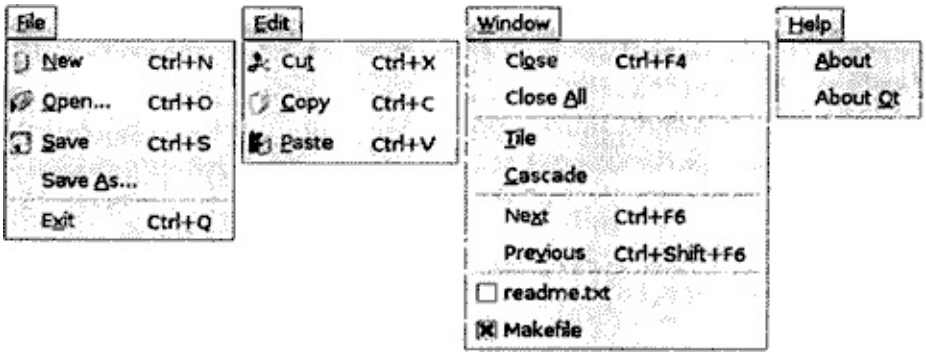


Рис. 6.17. Меню MDI—приложения Editor.

Давайте начнем с класса *MainWindow.*

```
01 MainWindow::MainWindow()
02 {
03 workspace = new QWorkspace;
04 setCentralWidget(workspace);
05 connect(workspace, SIGNAL(windowActivated(QWidget *)),
06 this, SLOT(updateMenus()));
07 createActions();
08 createMenus();
09 createToolBars();
10 createStatusBar();
11 setWindowTitle(tr("MDI Editor"));
12 setWindowIcon(QPixmap(":/images/icon.png"));
13 }
```

В конструкторе *MainWindow* мы создаем виджет *QWorkspace* и делаем его центральным виджетом. Мы связываем сигнал *windowActivated()* класса *QWorkspace* со слотом, который мы будем использовать для обеспечения актуального состояния меню Window.

```
01 void MainWindow::newFile()
02 {
03 Editor *editor = createEditor();
04 editor->newFile();
05 editor->show();
06 }
```

Слот *newFile()* соответствует пункту меню File | New. Он зависит от закрытой функции *createEditor(),* создающей дочерний виджет *Editor.*

```
01 Editor *MainWindow::createEditor()
02 {
03 Editor *editor = new Editor;
04 connect(editor, SIGNAL(copyAvailable(bool)),
05 cutAction, SLOT(setEnabled(bool)));
06 connect(editor, SIGNAL(copyAvailable(bool)),
07 copyAction, SLOT(setEnabled(bool)));
08 workspace->addWindow(editor);
09 windowMenu->addAction(editor->windowMenuAction());
10 windowActionGroup->addAction(editor->windowMenuAction());
11 return editor;
```

```
12 }
```

Функция *createEditor()* создает виджет *Editor* и устанавливает два соединения «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 проблему.

После настройки *Editor* 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.

```
01 void MainWindow::open()
02 {
03 Editor *editor = createEditor();
04 if (editor->open()) {
05 editor->show();
06 } else {
07 editor->close();
08 }
09 }
```

Функция *open()* соответствует пункту меню File | Open. Этот пункт меню создает *Editor* для нового документа и вызывает функцию *open()* для *Editor*. Имеет смысл выполнять файловые операции в классе *Editor*, а не в классе *MainWindow*, поскольку каждый *Editor* требует поддержки своего собственного состояния.

Если функция *open()* завершится неудачей, мы просто закроем редактор, поскольку пользователь уже будет уведомлен об ошибке. Мы не обязаны сами явно удалять объект *Editor;* это происходит автоматически при условии установки атрибута виджета *Qt::WA_DeleteOnClose*, что и делается в конструкторе *Editor*.

```
01 void MainWindow::save()
02 {
03 if (activeEditor()) {
04 activeEditor()->save();
05 }
06 }
```

Слот *save()* вызывает функцию *Editor::save()* для активного редактора, если таковой имеется. И снова программный код по выполнению реальной работы находится в классе *Editor*.

```
01 Editor *MainWindow::activeEditor()
02 {
03 return qobject_cast<Editor *>(workspace->activeWindow());
04 }
```

Закрытая функция *activeEditor()* возвращает активное дочернее окно в виде указателя типа *Editor* или нулевой указатель при отсутствии такого окна.

```
01 void MainWindow::cut()
02 {
03 if (activeEditor())
04 activeEditor()->cut();
05 }
```

Слот *cut()* вызывает функцию *Editor::cut()* для активного редактора. Мы не приводим слоты *copy(), paste()* и *del(),* потому что они имеют такой же вид.

```
01 void MainWindow::updateMenus()
02 {
03 bool hasEditor = (activeEditor() != 0);
04 bool hasSelection = activeEditor()
05 && activeEditor()->textCursor().hasSelection();
06 saveAction->setEnabled(hasEditor);
07 saveAsAction->setEnabled(hasEditor);
08 pasteAction->setEnabled(hasEditor);
09 cutAction->setEnabled(hasSelection);
10 copyAction->setEnabled(hasSelection);
11 closeAction->setEnabled(hasEditor);
12 closeAllAction->setEnabled(hasEditor);
13 tileAction->setEnabled(hasEditor);
14 cascadeAction->setEnabled(hasEditor);
15 nextAction->setEnabled(hasEditor);
16 previousAction->setEnabled(hasEditor);
17 separatorAction->setVisible (hasEditor);
18 if (activeEditor())
19 activeEditor()->windowMenuAction()->setChecked(true);
20 }
```

Слот *updateMenus()* вызывается всякий раз, когда окно становится активным (и когда закрывается последнее окно) для обновления системы меню благодаря помещенному нами в конструктор *MainWindow* соединению «сигнал—слот».

Большинство пунктов меню имеет смысл 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 предьщущего активного окна.

```
01 void MainWindow::createMenus()
02 {
03 windowMenu = menuBar()->addMenu(tr("&Window"));
04 windowMenu->addAction(closeAction);
05 windowMenu->addAction(closeAllAction);
06 windowMenu->addSeparator();
07 windowMenu->addAction(tileAction);
08 windowMenu->addAction(cascadeAction);
09 windowMenu->addSeparator();
10 windowMenu->addAction(nextAction);
11 windowMenu->addAction(previousAction);
```

```
12 windowMenu->addAction(separatorAction);
13 }
```

Закрытая функция *createMenus()* заполняет меню Window командами. Здесь используются типичные для такого рода меню команды, и они легко реализуются с применением слотов *closeActiveWindow(), closeAllWindows(), tile()* и *cascade()* класса *QWorkspace*. Всякий раз, когда пользователь открывает новое окно, в меню Window добавляется список действий. (Это делается в функции *createEditor(),* которую мы видели.) При закрытии пользователем окна редактора соответствующий ему пункт в меню Window удаляется (поскольку его владельцем является это окно редактора), т.е. пункт меню удаляется из меню Window автоматически.

```
01 void MainWindow::closeEvent(QCloseEvent *event)
02 {
03 workspace->closeAllWindows();
04 if (activeEditor()) {
05 event->ignore();
06 } else {
07 event->accept();
08 }
09 }
```

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 окно. 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,* и поэтому мы можем перейти к реализации класса *Editor.* Класс *Editor* представляет одно дочернее окно. Он наследует *QTextEdit*, который обеспечивает функциональность текстового редактора. Точно так же, как любой виджет, который может использоваться в качестве автономного окна, он может использоваться и в качестве дочернего окна в рабочем пространстве интерфейса MDI.

Ниже приводится определение класса:

```
01 class Editor : public QTextEdit
02 {
03 Q_OBJECT
04 public:
05 Editor(QWidget *parent = 0);
06 bool openFile(const QString &fileName);
07 bool save();
08 bool saveAs();
09 void newFile();
10 bool open();

11 protected:
```

```
12 QSize sizeHint() const;
13 QAction *windowMenuAction() const { return action; }
14 void closeEvent(QCloseEvent *event);
15 private slots:
16 void documentWasModified();

17 private:
18 bool okToContinue();
19 bool saveFile(const QString &fileName);
20 void setCurrentFile(const QString &fileName);
21 bool readFile(const QString &fileName);
22 bool writeFile(const QString &fileName);
23 QString strippedName(const QString &fullFileName);
24 QString curFile;
25 bool isUntitled;
26 QString fileFilters;
27 QAction *action;
28 }
```

Присутствующие в классе *MainWindow* приложения Электронная таблица четыре закрытые функции имеются также в классе *Editor: okToContinue(), saveFile(), setCurrentFile()* и *strippedName()*.

```
01 Editor::Editor(QWidget *parent)
02 : QTextEdit(parent)
03 {
04 action = new QAction(this);
05 action->setCheckable(true);
06 connect(action, SIGNAL(triggered()), this, SLOT(show()));
07 connect(action, SIGNAL(triggered()), this, SLOT(setFocus()));
08 isUntitled = true;
09 fileFilters = tr("Text files (*.txt)\nAll files (*)");
10 connect(document(), SIGNAL(contentsChanged()),
11 this, SLOT(documentWasModified()));
12 setWindowIcon(QPixmap(":/images/document.png"));
13 setAttribute(Qt::WA_DeleteOnClose);
14 }
```

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*.

После выпрлнения конструктора мы ожидаем вызова либо функции *newFile()*, либо функции *open()*.

```
01 void Editor::newFile()
02 {
03 static int documentNumber = 1;
04 curFile = tr("document%1.txt").arg(documentNumber);
05 setWindowTitle(curFile + "[*]");
06 action->setText(curFile);
07 isUntitled = true;
08 ++documentNumber;
09 }
```

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. Мы рассматривали этот маркер в главе 3.

```
01 bool Editor::open()
02 {
03 QString fileName = QFileDialog::getOpenFileName(
04 this, tr("Open"), fileFilters);
05 if(fileName.isEmpty())
06 return false;
07 return openFile(fileName);
08 }
```

Функция *open()* пытается открыть существующий файл при помощи функции *openFile()*.

```
01 bool Editor::save()
02 {
03 if (isUntitled) {
04 return saveAs();
05 } else {
06 return saveFile(curFile);
07 }
```

Функция *save()* используёт переменную *isUntitled* для определения вида вызываемой функции *saveFile()* или *saveAs()*.

```
01 void Editor::closeEvent(QCloseEvent *event)
02 {
03 if (okToContinue()) {
04 event->accept();
05 } else {
06 event->ignore();
```

```
07 }
08 }
```

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?» (Сохранить изменения?). Если функция *okToContinue()* возвращает *true*, мы обрабатывам событие закрытия; в противном случае мы «игнорируем» его и окно оставляем прежним.

```
01 void Editor::setCurrentFile(const QString &fileName)
02 {
03 curFile = fileName;
04 isUntitled = false;
05 action->setText(strippedName(curFile));
06 document()->setModified(false);
07 setWindowTitle(strippedName(curFile) + "[*]");
08 setWindowModified(false);
09 }
```

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*.

```
01 QSize Editor::sizeHint() const
02 {
03 return QSize(72 * fontMetrics().width('x'),
04 25 * fontMetrics().lineSpacing());
05 }
```

Функция *sizeHint()* возвращает размер, рассчитанный на основе ширины буквы «x» и высоты строки текста. *QWorkspace* использует идеальный размер в качестве начального размера окна.

Ниже приводится файл *main.cpp* MDI—приложения *Editor:*

```
01 #include <QApplication>
02 #include "mainwindow.h"
03 int main(int argc, char *argv[])
04 {
05 QApplication app(argc, argv);
06 QStringList args = app.arguments();
07 MainWindow mainWin;
08 if (args.count() > 1) {
09 for (int i = 1; i < args.count(); ++i)
10 mainWin.openFile(args[i]);
11 } else {
12 mainWin.newFile();
13 }
14 mainWin.show();
```

```
15 return app.exec();
16 }
```

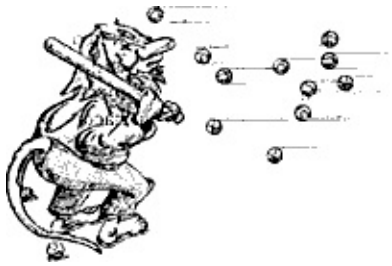
Если пользователь задает в командной строке какие-нибудь файлы, мы пытаемся их загрузить, в противном случае мы начинаем работу с пустым документом. Такие характерные для Qt опции командной строки, как *—style* и *—font* (стиль и шрифт), автоматически убираются из списка аргументов конструктором *QApplication*. Поэтому, если мы напишем в командной строке

```
mdieditor -style motif readme.txt
```

QApplication::arguments() возвратит *QStringList* с двумя элементами («mdieditor» и «readme.txt»), а MDI—приложение Editor запустится с документом *readme.txt*.

Интерфейс MDI представляет собой один из способов работы одновременно со многими документами. В системе MacOS X более предпочтителен подход, связанный с применением нескольких окон верхнего уровня. Этот подход рассматривается в разделе «Работа со многими документами» главы 3.

Глава 7. Обработка событий



События генерируются оконной системой или Qt в ответ на различные действия. Когда пользователь нажимает или отпускает клавишу или кнопку мышки, генерируется событие клавиши клавиатуры или кнопки мышки; когда окно впервые выводится на экран, генерируется событие рисования, указывая появившемуся окну на необходимость его прорисовки. Большинство событий генерируются 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 соственных 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 мышки или клавиатуры, сгенерировавших этот сигнал. Но если мы реализуем такой класс, как *QPushButton*, нам необходимо написать программный код для обработки событий мышки и клавиатуры и при необходимости сгенерировать сигнал *clicked()*.

Переопределение обработчиков событий

В Qt событие (event) — это объект, производный от *QEvent*. Qt обрабатывает более сотни типов событий, каждое из которых идентифицируется определенным значением перечисления. Например, *QEvent::type()* возвращает *QEvent::MouseButtonPress* для событий нажатия кнопки мышки.

Для событий многих типов недостаточно тех данных, которые могут храниться в простом объекте *QEvent:* например, для событий нажатия кнопки мышки необходимо иметь информацию о том, какая кнопка мышки привела к возникновению данного события, а также о том, где находился курсор мышкй 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 ознакомились в предыдущих главах со многими обработчиками событий при реализации *MainWindow*, *IconEditor* и *Plotter*. Существует много других типов событий, приводимых 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:

```
01 void CodeEditor::keyPressEvent(QKeyEvent *event)
02 {
03 switch (event->key()) {
04 case Qt::Key_Home:
05 if (event->modifiers() & Qt::ControlModifier) {
06 goToBeginningOfDocument();
07 } else {
08 goToBeginningOfLine();
09 }
10 break;
11 case Qt::Key_End:
12 …
13 default:
14 QWidget::keyPressEvent(event);
```

```
15 }
16 }
```

Клавиши Tab и Backtab (Shift+Tab) представляют собой особый случай. Они обрабатываются функцией *QWidget::event()* 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:

```
01 bool CodeEditor::event(QEvent *event)
02 {
03 if (event->type() == QEvent::KeyFress) {
04 QKeyEvent *keyEvent = static_cast<QKeyEvent *>event;
05 if (keyEvent->key() == Qt::Key_Tab) {
06 insertAtCurrentPosition('\t');
07 return true;
08 }
09 }
10 return QWidget::event(event);
11 }
```

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 передала 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:

```
01 MainWindow::MainWindow()
02 {
03 editor = new CodeEditor;
04 setCentralWidget(editor);
05 goToBeginningOfLineAction =
06 new QAction(tr("Go to Beginning of Line"), this);
07 goToBeginningOfLineAction->setShortcut(tr("Home"));
08 connect(goToBeginningOfLineAction, SIGNAL(activated()),
09 editor, SLOT(goToBeginningOfLine()));
10 goToBeginningOfDocumentAction =
11 new QAction(tr("Go to Beginning of Document"), this);
12 goToBeginningOfDocumentAction->setShortcut(tr("Ctrl+Home"));
13 connect(goToBeginningOfDocumentAction, SlGNAL(activated()),
14 editor, SLOT(goToBeginningOfDocument());
15 …
16 }
```

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 используется в *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 события является событие таймера. Если большинство других событий возникают 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.* 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.

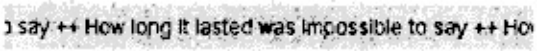


Рис. 7.1. Виджет Ticker.

Ниже приводится заголовочный файл:

```
01 #ifndef TICKER_H
02 #define TICKER_H
03 #include <QWidget>

04 class Ticker : public QWidget
05 {
06 Q_OBJECT
07 Q_PROPERTY(QString text READ text WRITE setText)

08 public:
09 Ticker(QWidget *parent = 0);
10 void setText(const QString &newText);
11 QString text() const { return myText; }
12 QSize sizeHint() const;

13 protected:
14 void paintEvent(QPaintEvent *event);
15 void timerEvent(QTimerEvent *event);
16 void showEvent(QShowEvent *event);
17 void hideEvent(QHideEvent *event);

18 private:
19 QString myText;
20 int offset;
21 int myTimerId;
22 };
```

```
23 #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:

```
01 #include <QtGui>
02 #include "ticker.h"
03 Ticker::Ticker(QWidget *parent)
04 : QWidget(parent)
05 {
06 offset = 0;
07 myTimerId = 0;
08 }
```

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.

```
09 void Ticker::setText(const QString &newText)
10 {
11 myText = newText;
12 update();
13 updateGeometry();
14 }
```

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.

```
15 QSizeTicker::sizeHint() const
16 {
17 return fontMetrics().size(0, text());
18 }
```

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.)

```
19 void Ticker::paintEvent(QPaintEvent * /* event */)
20 {
21 QPainter painter(this);
22 int textWidth = fontMetrics().width(text());
23 if (textWidth < 1)
24 return;
25 int x= -offset;
26 while (x < width()) {
27 painter.drawText(x, 0, textWidth, height(),
28 Qt::AlignLeft | Qt::AlignVCenter, text());
29 x += textWidth;
```

```
30 }
31 }
```

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*.

```
32 void Ticker::showEvent(QShowEvent * /* event */)
33 {
34 myTimerId = startTimer(30);
35 }
```

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.

```
36 void Ticker::timerEvent(QTimerEvent *event)
37 {
38 if (event->timerId() == myTimerId) {
39 ++offset;
40 if (offset >= fontMetrics().width(text()))
41 offset= 0;
42 scroll(-1, 0);
43 } else {
44 QWidget::timerEvent(event);
45 }
46 }
```

Функция *timerEvent()* вызывается системой 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 наш базовый класс.

```
47 void Ticker::hideEvent(QHideEvent * /* event */)
48 {
49 killTimer(myTimerId);
50 }
```

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).

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:

```
01 void MyLineEdit::keyPressEvent(QKeyEvent *event)
02 {
03 if (event->key()== Qt::Key_Space) {
04 focusNextChild();
05 } else {
06 QLineEdit::keyPressEvent(event);
07 }
08 }
```

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 сострит 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:*

```
01 CustomerInfoDialog::CustomerInfoDialog(QWidget *parent)
02 : QDialog(parent)
03 {
04 firstNameEdit->installEventFilter(this);
05 lastNameEdit->installEventFilter(this);
06 cityEdit->installEventFilter(this);
07 phoneNumberEdit->installEvehtFilter(this);
08 }
```

После регистрации фильтра события 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:

```
01 bool CustomerInfoDialog::eventFilter(QObject *target, QEvent *event)
02 {
03 if (target == firstNameEdit || target == lastNameEdit
04 || target == cityEdit || target == phoneNumberEdit) {
05 if (event->type() == QEvent::KeyPress) {
06 QKeyEvent *keyEvent = static_cast<QKeyEvent *>(event);
07 if (keyEvent->key() == Qt::Key_Space) {
08 focusNextChild();
09 return true;
10 }
11 }
12 }
13 return QDialog::eventFilter(target, event);
14 }
```

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 функрию *focusNextChild()* для перехода фокуса на следующий виджет в фокусной цепочке и возвращаем *true* для уведомления Qt о завершении нами обработки события. Если бы мы вернули *false*, Qt отослала 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 4.1 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 мышки, посылаемых для отключенных 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 мышки 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.

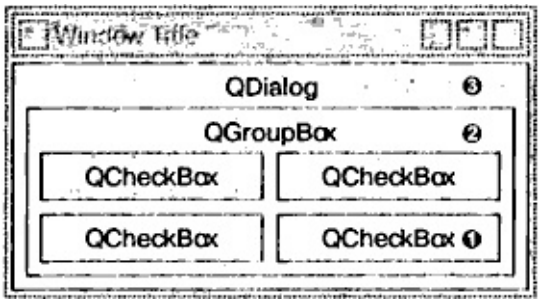


Рис. 7.2. Передача события в диалоговом окне.

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.*

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 начинает выполняться цикл 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 невосприимчивым 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 18.

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:*

```
01 bool Spreadsheet::writeFile(const QString &fileName)
02 {
03 QFile file(fileName);
04 …
05 for (int row = 0; row < RowCount; ++row) {
06 for (int column = 0; column < ColumnCount; ++column) {
07 QString str = formula(row, column);
08 if (!str.isEmpty())
09 out << quint16(row) << quint16(column) << str;
10 }
11 qApp->processEvents();
12 }
13 return true;
14 }
```

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 событий мышки и клавиатуры.

Часто нам хочется показывать индикатор состояния процесса *QProgressDialog* в ходе выполнения продолжительной операции. *QProgressDialog* имеет полоску 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:

```
01 bool Spreadsheet::writeFile(const QString &fileName)
02 {
03 QFile file(fileName);
04 …
05 QProgressDialog progress(this);
07 progress.setLabelText(tr("Saving %1").arg(fileName));
08 progress.setRange(0, RowCount);
09 progress.setModal(true);
10 for (int row = 0; row < RowCount; ++row) {
11 progress.setValue(row);
12 qApp->processEvents();
13 if (progress.wasCanceled()) {
14 file.remove();
15 return false;
16 }
17 for (int column = 0; column < ColumnCount; ++column) {
18 QString str = formula(row, column);
19 if (!str.isEmpty())
20 out << quint16(row) << quint16(column) << str;
21 }
22 }
23 return true;
24 }
```

Мы создаем *QProgressDialog*, в котором *RowCount* 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 либо нажатия пользователем кнопки мышки 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.

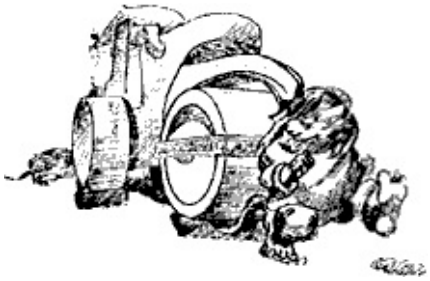
Кроме многопоточности и применения *QProgressDialog* существует совершенно другой способ работы с продолжительными операциями. Вместо выполнения заданной обработки сразу по поступлении запроса пользователя мы можем отложить эту обработку до момента перехода приложения в состояние ожидания. Этим способом 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 ожидании:

```
01 void Spreadsheet::timerEvent(QTimerEvent *event)
02 {
03 if(event->timerId() == myTimerId) {
04 while (step < MaxStep &&
05 !qApp->hasPendingEvents()) {
06 performStep(step);
07 ++step;
08 }
09 } else {
10 QTableWidget::timerEvent(event);
11 }
12 }
```

Если функция *hasPendingEvents()* возвращает *true*, мы останавливаем процесс и передаем управление обратно Qt. Этот процесс будет возобновлен после обработки Qt всех своих ожидающих событий.

Глава 8. Графика 2D и 3D



Основу используемых в Qt средств графики 2D составляет класс *QPainter* (рисовальщик Qt). Этот класс может использоваться для рисования геометрически 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 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 преобразование координат, что делает графику 2D независимой от разрешающей способности.

QPainter может использоваться для вычерчивания на таких «устройствах рисования», как *QWidget, QPixmap* или *QImage*. *QPainter* удобно применять, когда мы программируем пользовательские виджеты или классы пользовательских графических элементов с особым внешним видом и режимом работы. Класс *QPainter* можно также использовать совместно с *QPrinter* для вывода графики на печатающее устройство и для генерации файлов PDF. Это значит, что во многих случаях мы можем использовать тот же самый программный код при отображении данных на экран и при получении напечатанных отчетов.

В качестве альтернативы классам *QPainter* можно использовать OpenGL. OpenGL является стандартной библиотекой графических средств 2D и 3D. Модуль *QtOpenGL* 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.

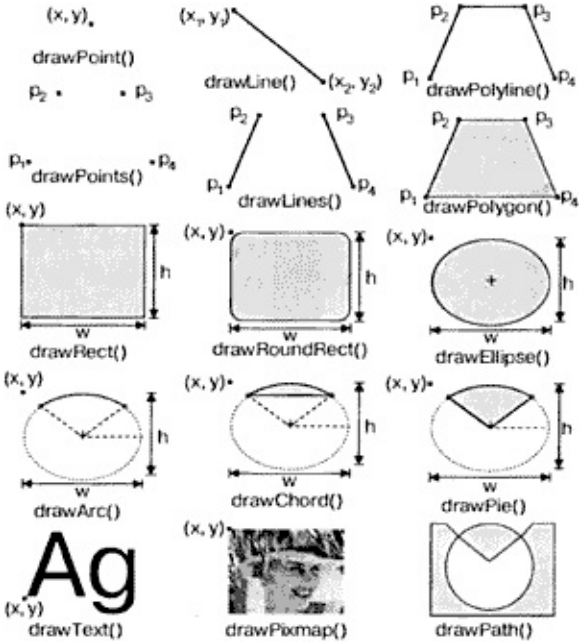


Рис. 8.1. Часто используемые функции draw…() рисовальщика QPainter.

Параметры настройки *QPainter* влияют на режим рисования. 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.

• **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.

• **Шрифт** используется для отображения текста. Шрифт имеет много атрибутов, в том числе название и размер.

Эти настройки можно в любое время модифицировать при помощи функций *setPen()*, *setBrush()* и *setFont()*, вызываемых для объектов *QPen*, *QBrush* или *QFont*.

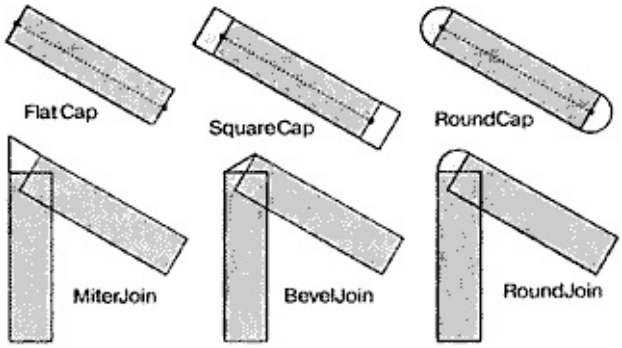


Рис. 8.2. Стили окончания линий и стили соединения линий.

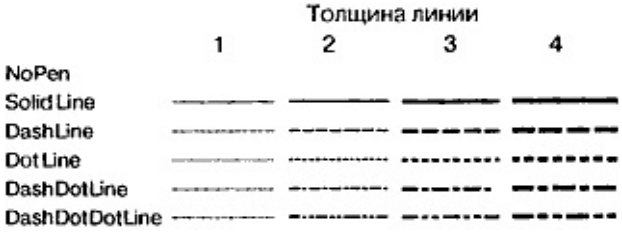


Рис. 8.3. Стили пера.

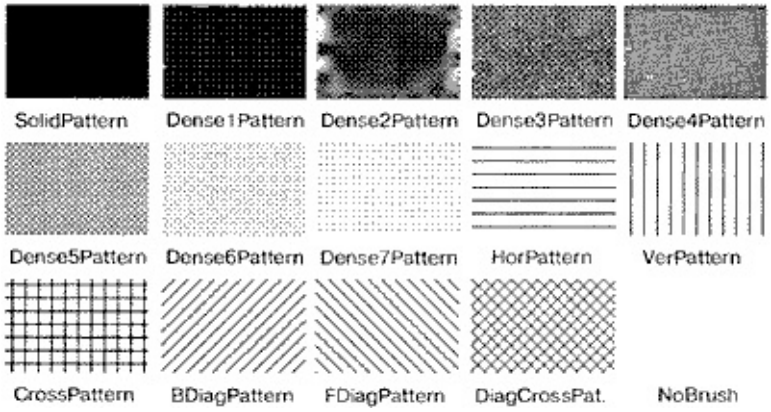


Рис. 8.4. Определенные в Qt стили кисти.

Рис. 8.5. Примеры геометрических фигур.

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.

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 второго и третьего порядка 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.

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 выше 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 трехмерности изображения, например стиль Plastique использует 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.

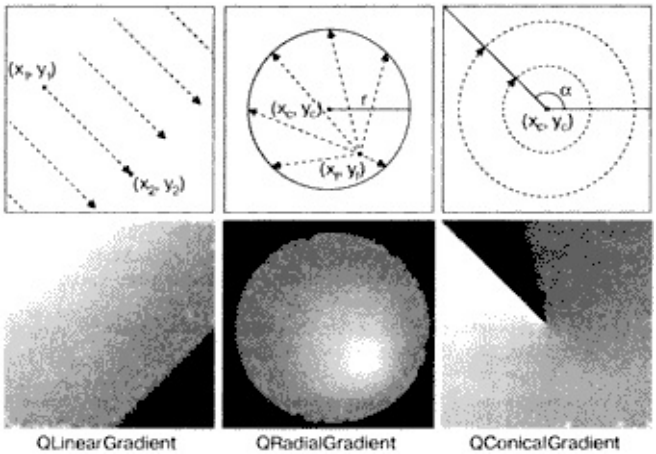


Рис. 8.6. Кисти QPainter с цветовыми градиентами.

• **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.

• **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.

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 (background brush)** 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 (brush origin)** задает точку начала отображения шаблона кисти, в качестве которой обычно используется точка верхнего левого угла виджета.

• **Границы области рисования (clip region)** определяют область рисования устройства. Операции рисования, которые выходят за пределы этой области, игнорируются.

• **Область отображения, окно и универсальная матрица преобразования (viewport, window и world matfix)** определяют способ перевода логических координат *QPainter* в физические координаты устройства рисования. По умолчанию системы логических и физических координат совпадают. Системы координат рассматриваются в следующем разделе.

• **Режим композиции (composition mode)** определяет способ взаимодействия новых выводимых пикселей с пикселями, уже присутствующими на устройстве рисования. По умолчанию используется режим «source over», при котором новые пиксели рисуются 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 функцию *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 рисовальщика

В используемой по умолчанию координатной системе рисовальщика *QPainter* точка (0, 0) находится в левом верхнем углу устройства рисования; значение координат *x* увеличивается при перемещении вправо, а значение координат *y* увеличивается при перемещении вниз. Каждый пиксель занимает область 1 × 1 в координатной системе, применяемой по умолчанию.

Необходимо помнить об одной важной особенности: центр пикселя имеет «полупиксельные» координаты. Например, пиксель в верхнем левом углу 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 представляет 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 создалось впечатление расположения пикселя точно в точке соприкосновения всех этих четырех пикселей. Если этот эффект нежелателен, его можно избежать, указывая полупиксельные координаты, например (100.5, 100.5).

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.

Рис. 8.7. Вычерчивание прямоугольника 6 × 5 при отсутствии сглаживания линий.

Теперь, когда мы ознакомились 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 (—50, —50) теперь соответствует точке с физическими координатами (0, 0), а точка с логическими координатами (+50, +50) соответствует точке с физическими координатами (320, 200). В этом примере мы не изменяли область отображения.

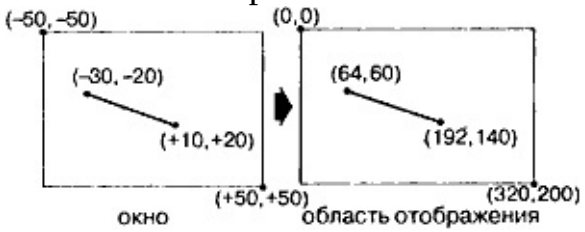


Рис. 8.8. Преобразование логических координат в физические координаты.

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 45°, мы бы использовали такой программный код:

```
QMatrix matrix;
matrix.rotate(45.0);
painter.setMatrix(matrix);
painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
```

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 (10, 20) в качестве точки поворота, мы можем перенести начало координат окна, выполнить поворот и затем сделать обратный перенос начала координат окна, устанавливая его в прежнее положение.

```
QMatrix matrix;
matrix.translate(-10.0, -20.0);
matrix.rotate(45.0);
matrix.translate(+10.0, +20.0);
painter.setMatrix(matrix);
painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
```

Более удобно осуществлять преобразования путем применения соответствующих функций класса *QPainter — translate(), scale(), rotate()* и *shear():*

```
painter.translate(-10.0, -20.0);
painter.rotate(45.0);
painter.translate(+10.0, +20.0);
painter.drawText(rect, Qt::AlignCenter, tr("Revenue"));
```

Но если мы хотим регулярно делать одни и те же преобразования, то они будут выполняться более эффективно при их хранении в объекте *QMatrix* и затем применении универсальной матрицы преобразования для рисовальщика всякий раз, когда требуется выполнить преобразование.

Рис. 8.9. Виджет OvenTimer.

Для иллюстрации преобразования рисовальщика мы рассмотрим программный код виджета *OvenTimer* (таймер духовки), показанного на рис. 8.9. Виджет *OvenTimer* имитирует кухонные таймеры, которые широко использовались до того, как в духовках стали применяться встроенные часы. Пользователь может повернуть ручку и установить требуемую длительность. Диск переключателя автоматически поворачивается против часовой стрелки, пока не достигнет отметки 0, в результате чего *OvenTimer* генерирует сигнал *timeout()*.

```
01 class OvenTimer : public QWidget
02 {
03 Q_OBJECT
04 public:
05 OvenTimer(QWidget *parent = 0);
06 void setDuration(int secs);
07 int duration() const;
08 void draw(QPainter *painter);
09 signals:
10 void timeout();
11 protected:
12 void paintEvent(QPaintEvent *event);
13 void mousePressEvent(QMouseEvent *event);
14 private:
15 QDateTime finishTime;
16 QTimer *updateTimer;
17 QTimer *finishTimer;
18 }
```

Класс *OvenTimer* наследует *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 = 1;
```

Мы начнем с нескольких констант, управляющих внешним видом и режимом работы таймера духовки.

```
01 OvenTimer::OvenTimer(QWidget *parent)
02 : QWidget(parent)
03 {
04 finishTime = QDateTime::currentDateTime();
05 updateTimer = new QTimer(this);
06 connect(updateTimer, SIGNAL(timeouf()),
07 this, SLOT(update()));
08 finishTimer = new QTimer(this);
09 finishTimer->setSingleShot(true);
10 connect(finishTimer, SIGNAL(timeout()),
11 this, SIGNAL(timeout()));
12 connect(finishTimer, SIGNAL(timeout()),
13 updateTimer, SLOT(stop()));
14 }
```

В конструкторе мы создаем два объекта *QTimer: updateTimer* используется для обновления внешнего вида виджета через каждую секунду, а *finishTimer* генерирует сигнал виджета *timeout()* при достижении отметки 0. Объект *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 каждую секунду, когда таймер неактивен.

```
01 void OvenTimer::setDuration(int secs)
02 {
03 if (secs > MaxSeconds) {
04 secs = MaxSeconds;
05 } else if (secs <= 0) {
06 secs = 0;
07 }
08 finishTime = QDateTime::currentDateTime().addSecs(secs);
09 if (secs > 0) {
10 updateTimer->start(UpdateInterval * 1000);
11 finishTimer->start(secs * 1000);
12 } else {
13 updateTimer->stop();
14 finishTimer->stop();
15 }
16 update();
17 }
```

Функция *setDuration()* выставляет таймер духовки, задавая требуемое количество секунд. 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.

```
01 int OvenTimer::duration() const
02 {
03 int secs = QDateTime::currentDateTime().
04 secsTo(finishTime);
05 if (secs < 0)
06 secs = 0;
07 return secs;
08 }
```

Функция *duration()* возвращает количество секунд, оставшееся до завершения работы таймера. Если таймер неактивен, мы возвращаем 0.

```
01 void OvenTimer::mousePressEvent(QMouseEvent *event)
02 {
03 QPointF point = event->pos() - rect().center();
04 double theta = atan2(-point.x(), -point.y()) * 180 / 3.14159265359;
05 setDuration(duration() + int(theta / DegreesPerSecond));
06 update();
07 }
```

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.

```
01 void OvenTimer::paintEvent(QPaintEvent * /* event */)
02 {
03 QPainter painter(this);
04 painter.setRenderHint(QPainter::Antialiasing, true);
05 int side = qMin(width(), height());
06 painter.setViewport((width() - side) / 2, (height() - side) / 2,
07 side, side);
08 painter.setWindow(-50, -50, 100, 100);
09 draw(&painter);
10 }
```

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.

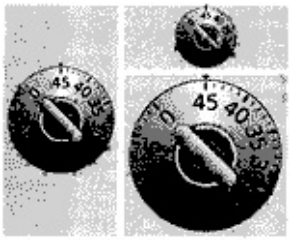


Рис. 8.10. Вид виджета OvenTimer при трех различных размерах.

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.

Теперь давайте рассмотрим программный код рисования:

```
01 void OvenTimer::draw(QPainter *painter)
02 {
03 static const int triangle[3][2] = {
04 { -2, -49 }, { +2, -49 }, { 0, -47 }
05 };
10 QPen thickPen(palette().foreground(), 1.5);
11 QPen thinPen(palette().foreground(), 0.5);
12 QColor niceBlue(150, 150, 200);
13 painter->setPen(thinPen);
14 painter->setBrush(palette().foreground());
15 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.

```
16 QConicalGradient coneGradient(0, 0, -90.0);
17 coneGradient.setColorAt(0.0, Qt::darkGray);
18 coneGradient.setColorAt(0.2, niceBlue);
19 coneGradient.setColorAt(0.5, Qt::white);
20 coneGradient.setColorAt(1.0, Qt::darkGray);
21 painter->setBrush(coneGradient);
22 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°.

```
23 QRadialGradient haloGradient(0, 0, 20, 0, 0);
24 haloGradient.setColorAt(0.0, Qt::lightGray);
25 haloGradient.setColorAt(0.8, Qt::darkGray);
26 haloGradient.setColorAt(0.9, Qt::white);
27 haloGradient.setColorAt(1.0, Qt::black);
28 painter->setPen(Qt::NoPen);
29 painter->setBrush(haloGradient);
30 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.

```
31 QLinearGradient knobGradient(-7, -25, 7, -25);
32 knobGradient.setColorAt(0.0, Qt::black);
33 knobGradient.setColorAt(0.2, niceBlue);
34 knobGradient.setColorAt(0.3, Qt::lightGray);
35 knobGradient.setColorAt(0.8, Qt::white);
36 knobGradient.setColorAt(1.0, Qt::black);

37 painter->rotate(duration() * DegreesPerSecond);
38 painter->setBrush(knobGradient);
39 painter->setPen(thinPen);
40 painter->drawRoundRect(-7, -25, 14, 50, 150, 50);

41 for (int i = 0; i <= MaxMinutes; ++i) {
42 if (i % 5 == 0) {
43 painter->setPen(thickPen);
44 painter->drawLine(0, -41, 0, -44);
45 painter->drawText(-15, -41, 30, 25,
46 Qt::AlignHCenter | Qt::AlignTop,
47 QString::number(i));
48 } else {
49 painter->setPen(thinPen);
50 painter->drawLine(0, -42, 0, -44);
51 }
52 painter->rotate(-DegreesPerMinute);
53 }
54 }
```

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 5 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 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 первоначальной матрицы преобразования на каждом шаге итерации:

```
41 for (int i = 0; i <= MaxMinutes; ++i) {
42 painter->save();
43 painter->rotate(-i * DegreesPerMinute);
44 if (i % 5 == 0) {
45 painter->setPen(thickPen);
46 painter->drawLine(0, -41, 0, -44);
47 painter->drawText(-15, -41, 30, 25,
48 Qt::AlignHCenter | Qt::AlignTop,
49 QString::number(i));
50 } else {
51 painter->setPen(thinPen);
52 painter->drawLine(0, -42, 0, -44);
53 }
54 painter->restore();
55 }
```

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. Основным недостатком этого подхода является то, что возможности Qt ограничиваются 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, которые отличаются от алгоритмов в X11 и Windows, что приводит к получению немного других результатов.

Когда точность важнее эффективности, мы можем рисовать по *QImage* и копировать 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 которому мы рисуем, должен создаваться с аргументом *QImage::Format_RGB32* или *QImage::Format_ARGB32_Premultiplied.*

Второй формат почти идентичен обычному формату ARGB32 *(0xaarrggbb);* отличие в том, что красный, зеленый и синий компоненты «предварительно умножаются» 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 значением *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 рисовании 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 выглядеть следующим образом:

```
01 void MyWidget::paintEvent(QPaintEvent *event)
02 {
03 QPainter painter(this);
04 painter.setRenderHint(QPainter::Antialiasing, true);
05 draw(&painter);
06 }
```

Ниже показано, как можно переписать виджетную функцию *paintEvent()* для применения независимого от платформы графического процессора Qt:

```
01 void MyWidget::paintEvent(QPaintEvent *event)
02 {
```

```
03 QImage image(size(), QImage::Format_ARGB32_Premultiplied);
04 QPainter imagePainter(&image);
05 imagePainter.initFrom(this);
06 imagePainter.setRenderHint(QPainter::Antialiasing, true);
07 imagePainter.eraseRect(rect());
08 draw(&imagePainter);
09 imagePainter.end();
10 QPainter widgetPainter(this);
11 widgetPainter.drawImage(0,0, image);
12 }
```

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 высококачественный результат 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 относящимся к перу, кисти, градиенту и изображению.

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. 8.11 показан результат рисования полупрозрачной бабочки поверх тестового шаблона при использовании разных режимов.

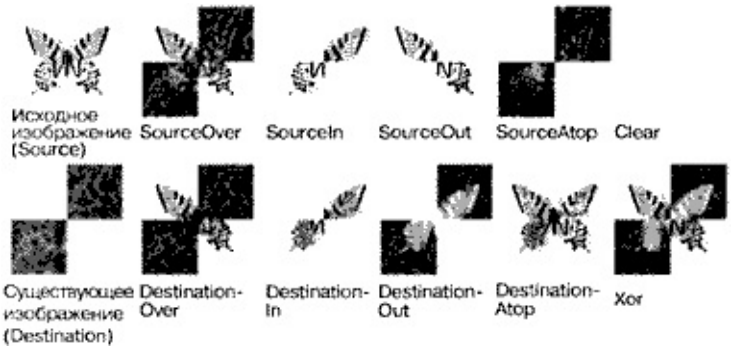


Рис. 8.11. Режимы композиции QPainter.

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* 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)*.

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 распечатывает объект *QImage:*

```
01 void PrintWindow::printImage(const Qlmage &image)
02 {
03 QPrintDialog printDialog(&printer, this);
04 if (printDialog.exec()) {
05 QPainter painter(&printer);
06 QRect rect = painter.viewport();
07 QSize size = image.size();
08 size.scale(rect.size(), Qt::KeepAspectRatio);
09 painter.setViewport(rect.x(). rect.y(), size.width(), size.height());
10 painter.setWindow (image.rect());
11 painter.drawImage(0, 0, image);
12 }
13 }
```

Рис. 8.12. Вывод на печатающее устройство объекта QImage.

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 OK; 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.

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:

```
01 void PrintWindow::printFlowerGuide(const QStringList &entries)
02 {
03 QString html;
04 foreach(QString entry, entries) {
05 QStringList fields = entry.split(": ");
06 QString title = Qt::escape(fields[0]);
07 QString body = Qt::escape(fields[1]);
08 html = "<table width=\"100%\" border=1 cellspacing=0>\n"
09 "<tr><td bgcolor=\"lightgray\"><font size=\"+1\">"
10 "<b><i>" + title + "</i></b></font>\n<tr><td>" + body"
11 + "\n</table>\n<br>\n";
12 }
13 printHtml(html);
14 }
```

На первом этапе *QStringList* преобразуется в формат HTML. Каждый цветок представляется таблицей HTML с двумя ячейками. Мы используем функцию *Qt::escape()*

для замены специальных символов «&», «<», «>» на соответствующие элементы формата HTML(«&», «<», «>»). Затем мы вызываем функцию *printHtml()* для печати текста.

```
01 void PrintWindow::printHtml(const QString &html)
02 {
03 QPrintDialog printDialog(&printer, this);
04 if (printDialog.exec()) {
05 QPainter painter(&printer);
06 QTextDocument textDocument;
07 textDocument.setHtml(html);
08 textDocument.print(&printer);
09 }
10 }
```

Функция *printHtml()* выводит диалоговое окно *QPrintDialog* и выполняет печать документа HTML. Она может без изменений повторно использоваться в любом приложении Qt для распечатки страниц произвольного текста в формате HTML.

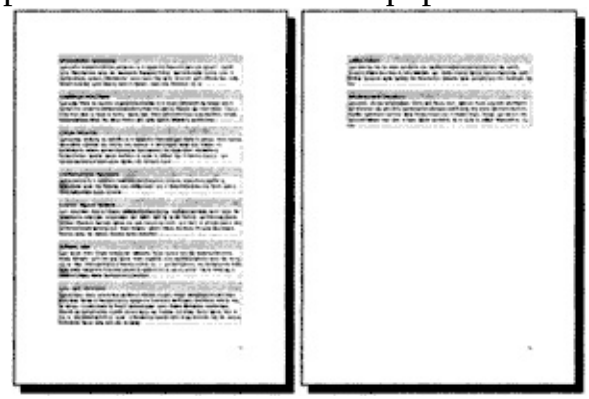


Рис. 8.13. Вывод на печать цветочного справочника с применением QTextDocument.

Преобразование документа в формат HTML и использование *QTextDocument* для его распечатки являются самым удобным способом печати отчетов и других сложных документов. В тех случаях, когда требуется обеспечить больший контроль, мы можем вручную выполнить компоновку страниц и их рисование. Давайте теперь посмотрим, как можно напечатать цветочный справочник при помощи класса *QPainter*. Ниже приводится новая версия функции *printFlowerGuide():*

```
01 void PrintWindow::printFlowerGuide(const QStringList &entries)
02 {
03 QPrintDialog printDialog(&printer, this);
04 if (printDialog.exec()) {
05 QPainter painter(&printer);
06 QList<QStringList> pages;
07 paginate(&painter, &pages, entries);
08 printPages(&painter, pages);
09 }
10 }
```

После настройки принтера и построения объекта рисовальщика мы вызываем вспомогательную функцию *paginate()* для определения содержимого каждой страницы. В результате получается вектор списков *QStringList*, причем каждый список *QStringList* содержит элементы одной страницы. Результат мы передаем функции *printPages()*.

Например, предположим, что цветочный справочник содержит всего 6 элементов, которые мы обозначим буквами *А, Б, В, Г, Д* и *Е* . Теперь предположим, что имеется достаточно места для элементов *А* и *Б* на первой странице, *В, Г* и *Д* на второй странице и *Е* на третьей странице. Тогда список *pages* содержал бы список *[А, Б]* в элементе с индексом 0, список *[В, Г, Д]* в элементе с индексом 1 и список *[Е]* в элементе с индексом 2.

```
01 void PrintWindow::paginate(QPainter *painter, QList<QStringList> *pages,
02 const QStringList &entries)
03 {
04 QStringList currentPage;
05 int pageHeight = painter->window().height() - 2 * LargeGap;
06 int y = 0;
07 foreach (QString entry, entries) {
08 int height = entryHeight(painter, entry);
09 if (y + height > pageHeight && !currentPage.empty()) {
10 pages->append(currentPage);
11 currentPage.clear();
12 y = 0;
13 }
14 currentPage.append(entry);
15 y += height + MediumGap;
16 }
17 if (!currentPage.empty())
18 pages->append(currentPage);
19 }
```

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.

```
01 int PrintWindow::entryHeight(QPainter *painter, const QString &entry)
02 {
03 int textWidth = painter->window().width() - 2 * SmallGap;
04 QString title = fields[0];
05 QString body = fields[1];
06 QStringList fields = entry.split(": ");
07 int maxHeight = painter->window().height();
08 painter->setFont(titleFont);
09 QRect titleRect = painter->boundingRect(0, 0, textWidth, maxHeight,
10 Qt::TextWordWrap, title);
11 painter->setFont(bodyFont);
12 QRect bodyRect = painter->boundingRect(0, 0, textWidth, maxHeight,
13 Qt::TextWordWrap, body);
14 return titleRect.height() + bodyRect.height() + 4 * SmallGap;
15 }
```

Функция *entryHeight()* использует *QPainter::boundingRect()* для вычисления размера области, занимаемой одним элементом по вертикали. На рис. 8.14 показана компоновка элементов одного цветка на странице и проиллюстрирован смысл констант *SmallGap* и *MediumGap*.

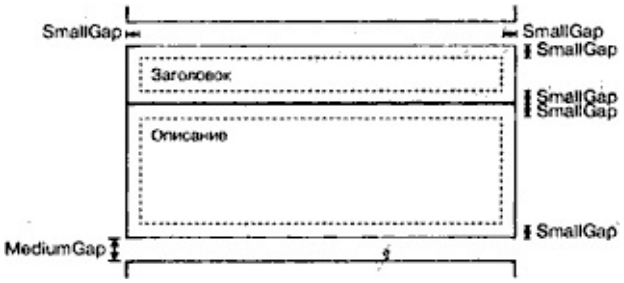


Рис. 8.14. Компоновка элементов справочника цветов на странице.

```
01 void PrintWindow::printPages(QPainter *painter,
02 const QList<QStringList> &pages)
03 {
04 int firstPage = printer.fromPage() - 1;
05 if (firstPage >= pages.size())
06 return;
07 if (firstPage == -1)
08 firstPage = 0;
09 int lastPage = printer.toPage() - 1;
10 if (lastPage == -1 || lastPage >= pages.size())
11 lastPage = pages.size() - 1;
12 int numPages = lastPage - firstPage + 1;
13 for (int i = 0; i < printer.numCopies(); ++i) {
14 for (int j = 0; j < numPages; ++j) {
15 if (i != 0 || j != 0)
16 printer.newPage();
17 int index;
18 if (printer.pageOrder() == QPrinter::FirstPageFirst) {
19 index = firstPage + j;
20 } else {
21 index = lastPage - j;
22 }
23 printPage(painter, pages[index], index + 1);
24 }
25 }
26 }
```

Функция *printPages()* предназначена для печати каждой страницы функцией *printPage()* с обеспечением правильного числа и правильной последовательности вызовов последней. 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. 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().)*

Рис. 8.15 аналогичен 8.13.

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.

```
01 void PrintWindow::printPage(QPainter *painter,
02 const QStringList &entries, int pageNumber)
03 {
04 painter->save();
05 painter->translate(0, LargeGap);
06 foreach (QString entry, entries) {
07 QStringList fields = entry.split(": ");
08 QString title = fields[0];
09 QString body = fields[1];
10 printBox(painter, title, titleFont, Qt::lightGray);
11 printBox(painter, body, bodyFont, Qt::white);
12 painter->translate(0, MediumGap);
13 }
14 painter->restore();
15 painter->setFont(footerFont);
16 painter->drawText(painter->window(),
17 Qt::AlignHCenter | Qt::AlignBottom,
18 QString::number(pageNumber));
19 }
```

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.

```
01 void PrintWindow::printBox(QPainter *painter, const QString &str,
02 const QFont &font, const QBrush &brush)
03 {
04 painter->setFont(font);
05 int boxWidth = painter->window().width();
06 int textWidth = boxWidth - 2 * SmallGap;
07 int maxHeight = painter->window().height();
08 QRect textRect = painter->boundingRect(SmallGap, SmallGap,
```

```
09 textWidth, maxHeight, Qt::TextWordWrap, str);
10 int boxHeight = textRect.height() + 2 * SmallGap;
11 painter->setPen(QPen(Qt::black, 2, Qt::SolidLine));
12 painter->setBrush(brush);
13 painter->drawRect(0, 0, boxWidth, boxHeight);
14 painter->drawText(textRect, Qt::TextWordWrap, str);
15 painter->translate(0, boxHeight);
16 }
```

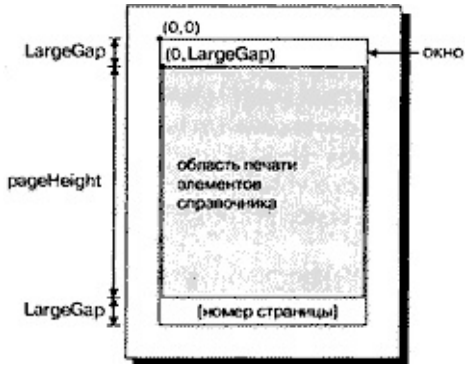


Рис. 8.16. Компоновка страницы справочника по цветам.

Функция printBox() вычерчивает контур блока, затем отображает текст внутри него.

Графические средства OpenGL

OpenGL является стандартным программным интерфейсом, предназначенным для воспроизведения графики 2D и 3D. Приложения Qt могут отображать графику 3D, используя модуль *QtOpenGL*, который рассчитан на применение системной библиотеки OpenGL. При изложении данного раздела предполагается, что вы знакомы с OpenGL. Если вы не знакомы с OpenGL, хорошо начинать его изучение с посещения сайта http://www.opengl.org/.

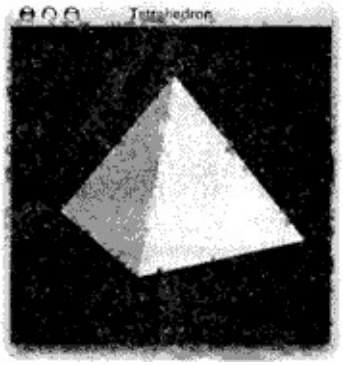


Рис. 8.17. Приложение Тетраэдр.

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* наследует *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.

Для демонстрации этого подхода мы рассмотрим программный код приложения Тетраэдр, показанного на рис. 8.17. Это приложение отображает в пространстве тетраэдр или четырехгранник, грани которого имеют различные цвета. Пользователь может поворачивать тетраэдр, нажимая кнопку мышки и перемещая ее. Пользователь может задавать цвет поверхности грани путем двойного щелчка с последующим выбором цвета в диалоговом окне *QColorDialog*, которое выдается на экран.

```
01 class Tetrahedron : public QGLWidget
02 {
03 Q_OBJECT
04 public:
05 Tetrahedron(QWidget *parent = 0);

06 protected:
07 void initializeGL();
08 void resizeGL(int width, int height);
09 void paintGL();
10 void mousePressEvent(QMouseEvent *event);
11 void mouseMoveEvent(QMouseEvent *event);
12 void mouseDoubleClickEvent(QMouseEvent *event);

13 private:
14 void draw();
```

```
15 int faceAtPosition(const QPoint &pos);
16 GLfloat rotationX;
17 GLfloat rotationY;
18 GLfloat rotationZ;
19 QColor faceColors[4];
20 QPoint lastPos;
21 }
```

Класс *Tetrahedron* наследует *QGLWidget*. Функции класса *QGLWidget initializeGL(), resizeGL()* и *paintGL()* переопределяются. Обработчики событий мышки класса *QWidget* переопределяются обычным образом.

```
01 Tetrahedron::Tetrahedron(QWidget *parent)
02 : QGLWidget(parent)
03 {
04 setFormat(QGLFormat(QGL::DoubleBuffer | QGL::DepthBuffer)
05 rotationX = -21.0;
06 rotationY = -57.0;
07 rotationZ = 0.0;
08 faceColors[0] = Qt::red;
09 faceColors[1] = Qt::green;
10 faceColors[2] = Qt::blue;
11 faceColors[3] = Qt::yellow;
12 }
```

В конструкторе мы вызываем функцию *QGLWidget::setFormat()* для установки контекста экрана OpenGL и инициализируем закрытые переменные этого класса.

```
01 void Tetrahedron::initializeGL()
02 {
03 qglClearColor(Qt::black);
04 glShadeModel(GL_FLAT);
05 glEnable(GL_DEPTH_TEST);
06 glEnable(GL_CULL_FACE);
07 }
```

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 *glClearIndex()* при использовании режима индексированных цветов.

```
01 void Tetrahedron::resizeGL(int width, int height)
02 {
03 glViewport(0, 0, width, height);
04 glMatrixMode(GL_PROJECTION);
05 glLoadIdentity();
06 GLfloat x = GLfloat(width) / height;
```

```
07 glFrustum(-x, x, -1.0, 1.0, 4.0, 15.0);
08 glMatrixMode(GL_MODELVIEW);
09 }
```

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.

```
01 void Tetrahedron::paintGL()
02 {
03 glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
04 draw();
05 }
```

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()*.

```
01 void Tetrahedron::draw()
02 {
04 static const GLfloat P1[3]= { 0.0, -1.0, +2.0 };
05 static const GLfloat P2[3] = { +1.73205081, -1.0, -1.0 };
06 static const GLfloat P3[3] = { -1.73205081, -1.0, -1.0 };
07 static const GLfloat P4[3] = { 0.0, +2.0, 0.0 };
08 static const GLfloat * const coords[4][3] = {
09 { P1, P2, P3 }, { P1, P3, P4 }, { P1, P4, P2 }, { P2, P4, P3 }
10 };

11 glMatrixMode(GL_MODELVIEW);
12 glLoadIdentity();
13 glTranslatef(0.0, 0.0, -10.0);
14 glRotatef(rotationX, 1.0, 0.0, 0.0);
15 glRotatef(rotationY, 0.0, 1.0, 0.0);
16 glRotatef(rotationZ, 0.0, 0.0, 1.0);

17 for (int i = 0; i < 4; ++i) {
18 glLoadName(i);
19 glBegin(GL_TRIANGLES);
20 qglColor(faceColors[i]);
21 for (int j = 0; j < 3; ++j) {
22 glVertex3f(coords[i][j][0],
23 coords[i][j][1], coords[i][j][2]);
24 }
25 glEnd();
26 }
27 }
```

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.

```
01 void Tetrahedron::mousePressEvent(QMouseEvent *event)
02 {
03 lastPos = event->pos();
04 }

05 void Tetrahedron::mouseMoveEvent(QMouseEvent *event)
06 {
07 GLfloat dx = GLfloat(event->x() - lastPos.x()) / width();
08 GLfloat dy = GLfloat(event->y() - lastPos.y()) / height();
09 if (event->buttons() & Qt::LeftButton) {
10 rotationX += 180 * dy;
11 rotationY += 180 * dx;
12 updateGL();
13 } else if (event->buttons() & Qt::RightButton) {
14 rotationX += 180 * dy;
15 rotationZ += 180 * dx;
16 updateGL();
17 }
18 lastPos = event->pos();
19 }
```

Функции класса *QWidget mousePressEvent()* и *mouseMoveEvent()* переопределяются, чтобы разрешить пользователю поворачивать изображение щелчком мышки и ее перемещением. Левая кнопка мышки позволяет пользователю поворачивать вокруг осей *x* и *y*, а правая кнопка мышки — вокруг осей *x* и *z*.

После модификации переменных *rotationX* и *rotationY* и л и *rotationZ* мы вызываем функцию *updateGL()* для перерисовки сцены.

```
01 void Tetrahedron::mouseDoubleClickEvent(QMouseEvent *event)
02 {
03 int face = faceAtPosition(event->pos());
04 if (face != -1) {
05 QColor color = QColorDialog::getColor(faceColors[face], this);
06 if (color.isValid()) {
07 faceColors[face] = color;
08 updateGL();
09 }
10 }
11 }
```

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). 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 мы вызываем функцию *updateGL()* для перерисовки экрана.

```
01 int Tetrahedron::faceAtPosition(const QPoint &pos)
02 {
03 const int MaxSize = 512;
04 GLuint buffer[MaxSize];
05 GLint viewport[4];

06 glGetIntegerv(GL_VIEWPORT, viewport);
07 glSelectBuffer(MaxSize, buffer);
08 glRenderMode(GL_SELECT);
09 glInitNames();
10 glPushName(0);

11 glMatrixMode(GL_PROJECTION);
12 glPushMatrix();
13 glLoadIdentity();
14 gluPickMatrix(GLdouble(pos.x()),
15 GLdouble(viewport[3] - pos.y()),
16 5.0, 5.0, viewport);
17 GLfloat x = GLfloat(width()) / height();
18 glFrustum(-x, x, -1.0, 1.0, 4.0, 15.0);
19 draw();
20 glMatrixMode(GL_PROJECTION);
21 glPopMatrix();

22 if (!glRenderMode(GL_RENDER))
23 return -1;
24 return buffer[3];
25 }
```

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 (ее «имя») из записи нажатия OpenGL.

Ниже приводится файл *main.cpp:*

```
01 #include <QApplication>
02 #include <iostream>
03 #include "tetrahedron.h"
04 using namespace std;

05 int main(int argc, char *argv[])
06 {
07 QApplication app(argc, argv);
08 if (!QGLFormat::hasOpenGL()) {
09 cerr << "This system has no OpenGL support" << endl;
```

```
10 return 1;
11 }
12 Tetrahedron tetrahedron;
13 tetrahedron.setWindowTitle(QObject::tr("Tetrahedron"));
14 tetrahedron.resize(300, 300);
15 tetrahedron.show();
16 return app.exec();
17 }
```

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*.

Глава 9. Технология «drag-and-drop»

Технология «drag-and-drop» («перетащить и отпустить») 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:*

```
01 class MainWindow : public QMainWindow
02 {
03 Q_OBJECT
04 public:
05 MainWindow();
06 protected:
07 void dragEnterEvent(QDragEnterEvent *event);
08 void dropEvent(QDropEvent *event);
09 private:
10 bool readFile(const QString &fileName);
11 QTextEdit *textEdit;
12 }
```

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.

```
01 MainWindow::MainWindow()
02 {
03 textEdit = new QTextEdit;
04 setCentralWidget(textEdit);
05 textEdit->setAcceptDrops(false);
06 setAcceptDrops(true);
07 setWindowTitle(tr("Text Editor"));
08 }
```

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 объектв области отображения *QTextEdit* и включая ее для главного окна, мы получаем события отпускания объектов в *MainWindow* для всего главного окна.

```
01 voidMainWindow::dragEnterEvent(QDragEnterEvent *event)
```

```
02 {
03 if (event->mimeData()->hasFormat("text/uri-list"))
04 event->acceptProposedAction();
05 }
```

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 универсальных идентификаторов ресурсов (URI — universal 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. Типы MIME используются буфером обмена и механизмом «drag-and-drop» для идентификации различных типов данных. Официальный список MIME—типов доступен по адресу http://www.iana.org/assignments/media-types/.

```
01 void MainWindow::dropEvent(QDropEvent *event)
02 {
03 QList<QUrl> urls = event->mimeData()->urls();
04 if (urls.isEmpty())
05 return;
06 QString fileName = urls.first().toLocalFile();
07 if (fileName.isEmpty())
08 return;
09 if (readFile(fileName))
10 setWindowTitle(tr("%1 -%2").arg(fileName)
11 .arg(tr("Drag File")));
12 }
```

Функция *dropEvent()* вызывается, когда пользователь отпускает объект на виджете. Мы вызываем функцию *QMimeData::urls()* для получения списка адресов *QUrl*. Обычно пользователи переносят одновременно только один файл, но возможен также перенос сразу нескольких выделенных файлов. Если имеется несколько URL или полученный URL оказывается нелокальным, 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 входить в приложение Project Chooser (составитель проектов), показанное на рис. 9.1.

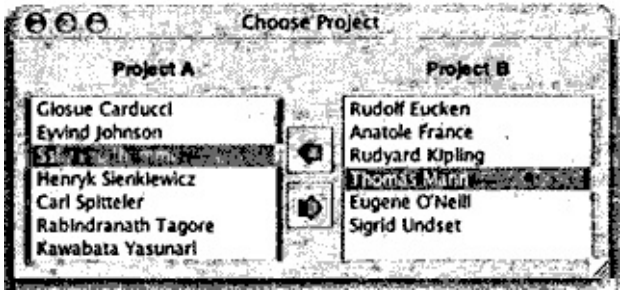


Рис. 9.1. Приложение Project Chooser.

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:

```
01 class ProjectListWidget : public QListWidget
02 {
03 Q_OBJECT
04 public:
05 ProjectListWidget(QWidget *parent= 0);
06 protected:
07 void mousePressEvent(QMouseEvent *event);
08 void mouseMoveEvent(QMouseEvent *event);
09 void dragEnterEvent(QDragEnterEvent *event);
10 void dragMoveEvent(QDragMoveEvent *event);
11 void dropEvent(QDropEvent *event);
12 private:
13 void startDrag();
14 QPoint startPos;
15 };
```

ProjectListWidget переопределяет пять обработчиков событий, которые объявлены в *QWidget*.

```
01 ProjectListWidget::ProjectListWidget(QWidget *parent)
02 : QListWidget(parent)
03 {
04 setAcceptDrops(true);
05 }
```

В конструкторе мы обеспечиваем возможность приема переносимого объекта в виджете со списком.

```
01 void ProjectListWidget::mousePressEvent(QMouseEvent *event)
02 {
03 if (event->button() == Qt::LeftButton)
04 startPos = event->pos();
05 QListWidget::mousePressEvent(event);
06 }
```

Когда пользователь нажимает левую кнопку мышки, мы сохраняем позицию мышки в закрытой переменной *startPos*. Мы вызываем определенную в классе *QListWidget* функцию *mousePressEvent()* для обеспечения обработки в *QListWidget* обычным образом события

нажатия кнопкй мышки.

```
01 void ProjectListWidget::mouseMoveEvent(QMouseEvent *event)
02 {
03 if (event->buttons() & Qt::LeftButton) {
04 int distance = (event->pos() - startPos).manhattanLength();
05 if (distance >= QApplication::startDragDistance())
06 startDrag();
07 }
08 QListWidget::mouseMoveEvent(event);
09 }
```

Действие, при котором пользователь перемещает курсор мышки и одновременно держит нажатой левую кнопку, мы рассматриваем как начало перетаскивания объекта. Мы вычисляем расстояние между текущей позицией мышки и позицией нажатия левой кнопки мышки. Если это расстояние превышает рекомендованное в *QApplication* расстояние для регистрации начала перетаскивания (обычно 4 пикселя), мы вызываем закрытую функцию *startDrag()* для запуска процесса перетаскивания объекта. Это предотвращает инициирование процесса перетаскивания из-за дрожания руки пользователя.

```
01 void ProjectListWidget::startDrag()
02 {
03 QListWidgetItem *item = currentItem();
04 if (item) {
05 QMimeData *mimeData = new QMimeData;
06 mimeData->setText(item->text());
07 QDrag *drag = new QDrag(this);
08 drag->setMimeData(mimeData);
09 drag->setPixmap(QPixmap(":/images/person.png"));
10 if (drag->start(Qt::MoveAction) == Qt::MoveAction)
11 delete item;
12 }
13 }
```

В функции *startDrag()* 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 типрв 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::start()* 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.

```
01 void ProjectListWidget::dragEnterEvent(QDragEnterEvent *event)
02 {
03 ProjectListWidget *source =
04 qobject_cast<ProjectListWidget *>(event->source());
05 if (source && source != this) {
06 event->setDropAction(Qt::MoveAction);
07 event->accept();
08 }
09 }
```

Виджет *ProjectListWidget* не только инициирует перенос объектов, он также является местом приема таких объектов, если они приходят от другого виджета *ProjectListWidget* того же самого приложения. *QDragEnterEvent::source()* возвращает указатель на виджет, который инициирует перенос, если этот виджет принадлежит тому же самому приложению; в противном случае он возвращает нулевой указатель. Мы используем *qobject_cast<T>()*, чтобы убедиться в инициировании переноса виджетом *ProjectListWidget*. Если все верно, мы указываем Qt на нашу готовность восприятия данного действия как переноса.

```
01 void ProjectListWidget::dragMoveEvent(QDragMoveEvent *event)
02 {
03 ProjectListWidget *source =
04 qobject_cast<ProjectListWidget *>(event->source());
05 if (source && source != this) {
07 event->setDropAction(Qt::MoveAction);
08 event->accept();
09 }
10 }
```

Программный код функции *dragMoveEvent()* идентичен 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)*.

```
01 void ProjectListWidget::dropEvent(QDropEvent *event)
02 {
03 ProjectListWidget *source =
04 qobject_cast<ProjectListWidget *>(event->source());
05 if (source && source != this) {
06 addItem(event->mimeData()->text());
07 event->setDropAction(Qt::MoveAction);
08 event->accept();
09 }
10 }
```

В *DropEvent()* мы используем функцию *QMimeData::text()* для получения перенесенного текста и создаем элемент с этим текстом. Нам также необходимо воспринять данное событие как «операцию 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 преобразбвать наши структуры данных в тип *QByteArray*, даже если переносимый объект не принимается, а 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 излишне замедлять 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:

```
01 void MyTableWidget::mouseMoveEvent(QMouseEvent *event)
02 {
03 if (event->buttons() & Qt::LeftButton) {
04 int distance = (event->pos() - startPos).manhattanLength();
05 if(distance >= QApplication::startDragDistance())
06 startDrag();
07 }
08 QTableWidget::mouseMoveEvent(event);
09 }

10 void MyTableWidget::startDrag()
11 {
12 QString plainText= selectionAsPlainText();
13 if (plainText.isEmpty())
```

```
14 return;

15 QMimeData *mimeData = new QMimeData;
16 mimeData->setText(plainText);
17 mimeData->setHtml(toHtml(plainText));
18 mimeData->setData("text/csv", toCsv(plainText).toUtf8());

19 QDrag *drag = new QDrag(this);
20 drag->setMimeData(mimeData);
21 if (drag->start(Qt::CopyAction | Qt::MoveAction) == Qt::MoveAction)
22 deleteSelection();
23 }
```

Закрытая функция *startDrag()* 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.

```
01 QString MyTableWidget::toCsv(const QString &plainText)
02 {
03 QString result = plainText;
04 result.replace("\\", "\\\\");
05 result.replace("\"", "\\\"");
06 result.replace("\t", "\", \"")
07 result.replace("\n", "\"\n\"");
08 result.prepend("\"");
09 result.append("\"");
10 return result;
11 }

12 QString MyTableWidget::toHtml(const QString &plainText)
13 {
14 QString result = Qt::escape(plainText);
15 result.replace("\t", "<td>");
16 result.replace("\n", "\n<tr><td>");
17 result.prepend("<table>\n<tr><td>");
18 result.append("\n</table>");
19 return result;
20 }
```

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()*.

```
01 void MyTableWidget::dropEvent(QDropEvent *event)
02 {
03 if (event->mimeData()->hasFormat("text/csv")) {
04 QByteArray csvData = event->mimeData()->data("text/csv");
05 QString csvText = QString::fromUtf8(csvData);
06 …
07 event->acceptProposedAction();
08 } else if (event->mimeData()->hasFormat("text/plain")) {
09 QString plainText = event->mimeData()->text();
10 …
11 event->acceptProposedAction();
12 }
13 }
```

Хотя мы предоставляем данные в трех разных форматах, мы принимаем в *dropEvent()* только два из них. Если подьзователь 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 *QMimeData*, чтобы отложить или избежать (потенциально затратных) преобразований между элементами *QTableWidgetltem* и массивом *QByteArray*. Ниже приводится определение нашего подкласса:

```
01 class TableMimeData : public QMimeData
02 {
03 Q_OBJECT
04 public:
05 TableMimeData(const QTableWidget *tableWidget,
06 const QTableWidgetSelectionRange &range);
07 const QTableWidget *tableWidget() const
08 { return myTableWidget; }
09 QTableWidgetSelectionRange range() const { return myRange; }
10 QStringList formats() const;
```

```
11 protected:
12 QVariant retrieveData(const QString &format,
13 QVariant::Type preferredType) const;

14 private:
15 static QString toHtml(const QString &plainText);
16 static QString toCsv(const QString &plainText);
17 QString text(int row, int column) const;
18 QString rangeAsPlainText() const;
19 const QTableWidget *myTableWidget;
20 QTableWidgetSelectionRange myRange;
21 QStringList myFormats;
22 };
```

Вместо реальных данных мы храним объект *QTableWidgetSelectionRange,* который определяет область переносимых ячеек и сохраняет указатель на *QTableWidget*. Функции *formats()* и *retrieveData()* класса *QMimeData* переопределяются.

```
01 TableMimeData::TableMimeData(const QTableWidget *tableWidget,
02 const QTableWidgetSelectionRange &range)
03 {
04 myTableWidget = tableWidget;
05 myRange = range;
06 myFormats << "text/csv" << "text/html" << "text/plain";
07 }
```

В конструкторе мы инициализируем закрытые переменные.

```
01 QStringList TableMimeData::formats() const
02 {
03 return myFormats;
04 }
```

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.

```
01 QVariant TableMimeData::retrieveData(const QString &format,
02 QVariant::Type preferredType) const
03 {
04 if (format == "text/plain") {
05 return rangeAsPlainText();
06 } else if (format =="text/csv") {
07 return toCsv(rangeAsPlainText()); }
08 else if (format == "text/html") {
09 return toHtml(rangeAsPlainText());
10 } else {
11 return QMimeData::retrieveData(format, preferredType);
12 }
13 }
```

Функция *retrieveData()* возвращает данные для заданного MIME—типа в виде объекта *QVariant*. Параметр *format* обычно содержит одну из строк, возвращенных 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.

```
01 void MyTableWidget::dropEvent(QDropEvent *event)
02 {
03 const TableMimeData *tableData =
04 qobject_cast<const TableMimeData *>(event->mimeData());
05 if (tableData) {
06 const QTableWidget *otherTable = tableData->tableWidget();
07 QTableWidgetSelectionRange otherRange = tableData->range();
08 …
09 event->acceptProposedAction();
10 } else if (event->mimeData()->hasFormat("text/csv")) {
11 QByteArray csvData = event->mimeData()->data("text/csv");
12 QString csvText = QString::fromUtf8(csVData);
13 …
14 event->acceptProposedAction();
15 } else if (event->mimeData()->hasFormat("text/plain")) {
16 QString plainText = event->mimeData()->text();
17 …
18 event->acceptProposedAction();
19 }
20 QTableWidget::mouseMoveEvent(event);
21 }
```

Функция *dropEvent()* аналогична функции с тем же названием, которую мы рассматривали ранее в данном разделе, но на этот раз мы ее оптимизируем, делая вначале проверку возможности приведения типа *QMimeData* в тип *TableMimeData*. Если *qobject_cast<T>()* срабатывает, это значит, что перенос был инициирован виджетом *MyTableWidget*, расположенным 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 недостаточно 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 *mimeData()* для буфера обмена.

В системе X11, как правило, можно вставлять выделенные объекты нажатием средней кнопки мышки, 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 *mouseReleaseEvent()* текстового редактора, поддерживающего вставку по нажатии средней кнопки мышки.

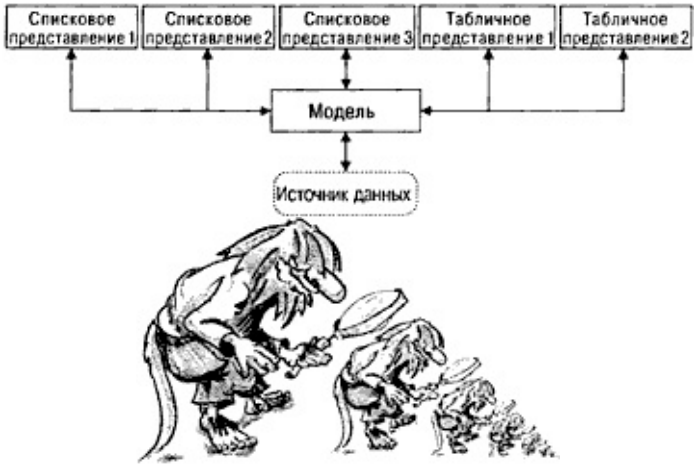
```
01 void MyTextEditor::mouseReleaseEvent(QMouseEvent *event)
02 {
03 QClipboard *clipboard = QApplication::clipboard();
04 if (event->button() == Qt::MidButton
05 && clipboard->supportsSelection()) {
06 QString text = clipboard->text(QClipboard::Selection);
07 pasteText(text);
08 }
09 }
```

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.

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 очень 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 за обеспечение отображаемых данных 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.

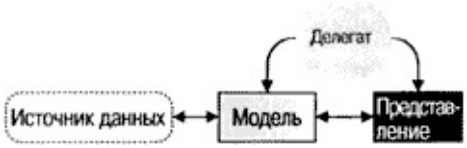


Рис. 10.1. Архитектура модель/представление в Qt.

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 (delegate)*. 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.

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. 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.

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 *QTableWidgetltem)*. При реализации этих удобных классов используются пользовательские модели, обеспечивающие появление требуемых элементов в представлениях.

Рис. 10.2. Одна модель может обслуживать несколько представлений.

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*, во втором примере — редактируемый *QTableWidget* и в третьем примере — используемый только для чтения *QTreeWidget*.

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.

Сначала покажем фрагмент заголовочного файла диалогового окна:

```
01 class FlowChartSymbolPicker : public QDialog {
02 Q_OBJECT
03 public:
04 FlowChartSymbolPicker(const QMap<int, QString> &symbolMap,
05 QWidget *parent = 0);
06 int selectedId() const { return id; }
07 void done(int result);
08 …
09 }
```

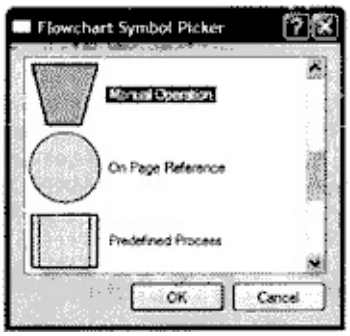


Рис. 10.3. Приложение Выбор символа блок—схемы (Flowchart Symbol Picker).

При создании диалогового окна мы должны передать его конструктору ассоциативный массив *QMap<int, QString>*, и после выполнения конструктора мы можем получить идентификатор выбранного элемента (или —1, если пользователь ничего не выбрал), вызывая *selectedId()*.

```
01 FlowChartSymbolPicker::FlowChartSymbolPicker(
02 const QMap<int, QString> &symbolMap, QWidget *parent)
03 : QDialog(parent)
04 {
05 id = -1;
06 listWidget = new QListWidget;
```

```
07 listWidget->setIconSize(QSize(60, 60));
08 QMapIterator<int, QString> i(symbolMap);
09 while (i.hasNext()) {
10 i.next();
11 QListWidgetItem *item = new QListWidgetItem(i.value(),
12 listWidget);
13 item->setIcon(iconForSymbol(i.value()));
14 item->setData(Qt::UserRole, i.key());
15 …
16 }
17 }
```

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.

```
01 void FlowChartSymbolPicker::done(int result)
02 {
03 id = -1;
04 if (result == QDialog::Accepted) {
05 QListWidgetItem *item = listWidget->currentItem();
06 if (item)
07 id = item->data(Qt::UserRole).toInt();
08 }
09 QDialog::done(result);
10 }
```

Функция *done()* класса *QDialog* переопределяется. Она вызывается при нажатии пользователем кнопок 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 связать их со слотами, 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.

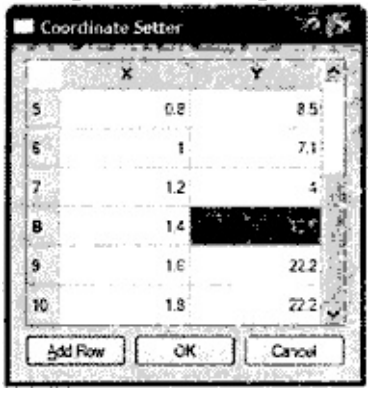


Рис. 10.4. Приложение Редактор координат.

Как и в предыдущем примере, мы основное внимание уделим программному коду, относящемуся к представлению элементов, и начнем с конструктора.

```
01 CoordinateSetter::CoordinateSetter(QList<QPointF> *coords,
02 QWidget *parent)
03 : QDialog(parent)
04 {
05 coordinates = coords;
06 tableWidget = new QTableWidget(0, 2);
07 tableWidget->setHorizontalHeaderLabels(
08 QStringList() << tr("X") << tr("Y"));
09 for (int row = 0; row < coordinates->count(); ++row) {
10 QPointF point = coordinates->at(row);
11 addRow();
12 tableWidget->item(row, 0)->setText(
13 QString::number(point.x()));
14 tableWidget->item(row, 1)->setText(
15 QString::number(point.y()));
16 }
17 …
18 }
```

Конструктор *QTableWidget* принимает начальное число строк и столбцов таблицы, выводимой на экран. Каждый элемент в *QTableWidget* представлен объектом *QTableWidgetltem,* включая элементы заголовков строк и столбцов. Функция *setHorizontalHeaderLabels()* задает заголовки всем столбцам, используя соответствующий

текст из переданного списка строк. По умолчанию *QTableWidget* обеспечивает заголовки строк числовыми метками, начиная с 1; именно это нам и нужно, поэтому нам не приходится задавать вручную заголовки строк.

После создания и центровки заголовков столбцов мы в цикле просматриваем все переданные нам данные с координатами. Для каждой пары (x, y) мы создаем два элемента *QTableWidgetItem*, соответствующие координатам x и y. Эти элементы добавляются в таблицу, используя функцию *QTableWidget::setItem()*, в аргументах которой кроме самого элемента задаются его строка и столбец.

По умолчанию виджет *QTableWidget* 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 сохранятся в элементах *QTableWidgetltem.* Запретить редактирование мы можем с помощью вызова *setEditTriggers(QAbstractItemView::NoEditTriggers).*

```
01 void CoordinateSetter::addRow()
02 {
03 int row = tableWidget->rowCount();
04 tableWidget->insertRow(row);
05 QTableWidgetltem *item0 = new QTableWidgetltem;
06 item0->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);
07 tableWidget->setItem(row, 0, item0);
08 QTableWidgetltem *item1 = new QTableWidgetltem;
09 item1->setTextAlignment(Qt::AlignRight | Qt::AlignVCenter);
10 tableWidget->setItem(row, 1, item1);
11 tableWidget->setCurrentItem(item0);
12 }
```

Слот *addRow()* вызывается, когда пользователь щелкает по кнопке Add Row (добавить строку). Мы добавляем в конец таблицы новую строку, используя *insertRow()*. Если пользователь попытается отредактировать какую-нибудь ячейку новой строки, *QTableWidget* автоматически создаст новый объект *QTableWidgetItem.*

```
01 void CoordinateSetter::done(int result)
02 {
03 if (result == QDialog::Accepted) {
04 coordinates->clear();
05 for (int row = 0; row < tableWidget->rowCount(); ++row) {
06 double x = tableWidget->item(row, 0)->text().toDouble();
07 double y = tableWidget->item(row, 1)->text().toDouble();
08 coordinates->append(QPointF(x, y));
09 }
10 }
11 QDialog::done(result);
12 }
```

Наконец, когда пользователь нажимает кнопку OK, 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.

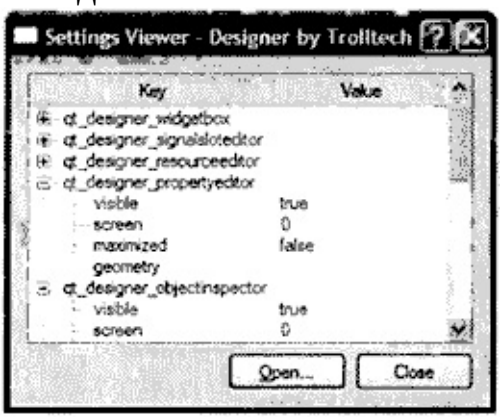


Рис. 10.5. Приложение Просмотр параметров настройки (Settings Viewer).

Ниже приводится фрагмент конструктора:

```
01 SettingsViewer::SettingsViewer(QWidget *parent)
02 : QDialog(parent)
03 {
04 organization = "Trolltech";
05 application = "Designer";
06 treeWidget = new QTreeWidget;
08 treeWidget->setColumnCount(2);
09 treeWidget->setHeaderLabels(
10 QStringList() << tr("Key") << tr("Value"));
11 treeWidget->header()->setResizeMode(0, QHeaderView::Stretch);
12 treeWidget->header()->setResizeMode(1, QHeaderView::Stretch);
13 …
14 setWindowTitle(tr("Settings Viewer"));
15 readSettings();
16 }
```

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*. В конце мы вызываем функцию *readSettings()*.

```
01 void SettingsViewer::readSettings()
02 {
03 QSettings settings(organization, application);
04 treeWidget->clear();
05 addChildSettings(settings, 0, "");
06 treeWidget->sortByColurnn(0);
07 treeWidget->setFocus();
08 setWindowTitle(tr("Settings Viewer - %1 by %2")
09 .arg(application).arg(organization));
10 }
```

Параметры настройки приложения хранятся в виде набора ключей и значений, имеющих иерархическую структуру. Закрытая 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 0, задавая корень в качестве родительского объекта.

```
01 void SettingsViewer::addChildSettings(QSettings &settings,
02 QTreeWidgetItem *parent, const QString &group)
03 {
04 QTreeWidgetItem *item;
05 settings.beginGroup(group);

06 foreach (QString key, settings.childKeys()) {
07 if (parent) {
08 item = new QTreeWidgetItem(parent);
09 } else {
10 item = new QTreeWidgetItem(treeWidget);
11 }
12 item->setText(0, key);
13 item->setText(1, settings.value(key).toString());
14 }

15 foreach (QString group, settings.childGroups()) {
16 if (parent) {
17 item = new QTreeWidgetItem(parent);
18 } else {
19 item = new QTreeWidgetItem(treeWidget);
20 }
21 item->setText(0, group);
21 addChildSettings(settings, item, group);
22 }
23 settings.endGroup();
24 }
```

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 *QTableWidgetItem.* Если в качестве родительского элемента задан 0, мы создаем дочерний элемент собственно виджета *QTreeWidget* (т.е. создается элемент верхнего уровня); в противном случае мы создаем дочерний элемент для объекта *parent*. 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 *QTreeWidget* 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:

• QStringListModel — хранит список строк;
• QStandardltemModel — хранит данные произвольной иерархической структуры;
• QDirModel — формирует структуру локальной файловой системы;
• QSqlQueryModel — формирует набор результата SQL—запроса;
• QSqlTableModel — формирует SQL—таблицу;
• QSqlRelationalTableModel — формирует SQL—таблицу с внешними ключами (foreign keys);
• QSortFilterProxyModel — сортирует и/или пропускает через фильтр другую модель.

В данном разделе мы рассмотрим способы применения моделей *QStringListModel*, *QDirModel* и *QSortFilterProxyModel*. SQL—модели рассматриваются 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.

Ниже приводится соответствующий фрагмент конструктора:

```
01 TeamLeadersDialog::TeamLeadersDialog(const QStringList &leaders,
02 QWidget *parent)
03 : QDialog(parent)
04 {
05 model = new QStringListModel(this);
06 model->setStringList(leaders);
07 listView = new QListView;
08 listView->setModel(model);
09 listView->setEditTriggers(QAbstractItemView::AnyKeyPressed
10 | QAbstractItemView::DoubleClicked);
11 …
12 }
```

Рис. 10.6. Приложение Лидеры команд (Team Leaders).

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.

```
01 void TeamLeadersDialog::insert()
02 {
03 int row = listView->currentIndex().row();
04 model->insertRows(row, 1);
05 QModelIndex index = model->index(row);
07 listView->setCurrentIndex(index);
08 listView->edit(index);
09 }
```

Когда пользователь нажимает на кнопку Insert (вставка), 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.

```
01 void TeamLeadersDialog::del()
02 {
03 model->removeRows(listView->currentIndex().row(), 1);
04 }
```

В конструкторе сигнал *clicked()* кнопки Delete (удалить) 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.

```
01 QStringList TeamLeadersDialog::leaders() const
02 {
03 return model->stringList();
04 }
```

Наконец, функция *leaders()* позволяет считывать отредактированные строки, когда диалоговое окно закрыто.

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 отображает список файлов или каталогов. В следующем примере 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.

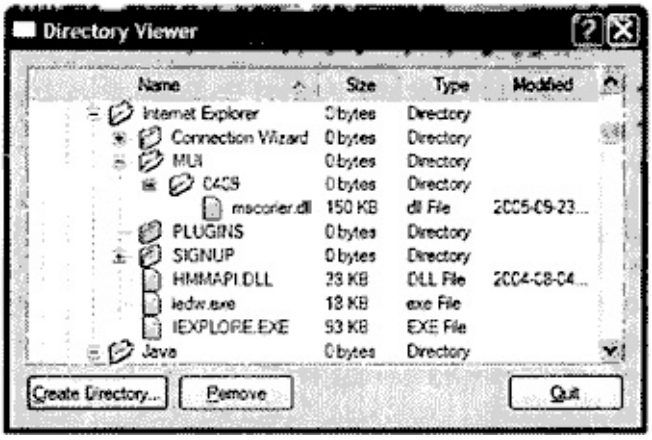


Рис. 10.7. Приложение Просмотр каталога.

Мы начнем с создания и настройки модели и представления в конструкторе диалогового окна Просмотр каталога (Directory Viewer).

```
01 DirectoryViewer::DirectoryViewer(QWidget *parent)
02 : QDialog(parent)
03 {
04 model = new QDirModel;
05 model->setReadOnly(false);
06 model->setSorting(QDir::DirsFirst | QDir::IgnoreCase | QDir::Name);
07 treeView = new QTreeView;
08 treeView->setModel(model);
09 treeView->header()->setStretchLastSection(true);
10 treeView->header()->setSortIndicator(0, Qt::AscendingOrder);
11 treeView->header()->setSortIndicatorShown(true);
12 treeView->header()->setClickable(true);
13 QModelIndex index = model->index(QDir::currentPath());
14 treeView->expand(index);
15 treeView->scrollTo(index);
16 treeView->resizeColumnToContents(0);
17 …
18 }
```

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 мышки, пользователь может сортирбвать 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 (создать каталог) и Remove (удалить) со слотами, выполняющими соответствующие действия. Нам не нужно иметь кнопку Rename (переименовать), поскольку пользователи могут переименовывать элементы каталога по месту, нажимая клавишу F2 и осуществляя ввод символов с клавиатуры.

```
01 voidDirectoryViewer::createDirectory()
02 {
03 QModelIndex index;
04 if (!index.isValid())
05 return;
06 QString dirName = QInputDialog::getText(this,
07 tr("Create Directory"), tr("Directory name"));
08 if (!dirName.isEmpty()) {
09 if (!model->mkdir(index,dirName).isValid())
10 QMessageBox::information(this,
11 tr("Create Directory"),
12 tr("Failed to create the directory"));
13 }
14 }
```

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.

Последний пример в этом разделе показывает, 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 (ColorNames).

Ниже приводится фрагмент конструктора ColorNamesDialog:

```
01 ColorNamesDialog::ColorNamesDialog(QWidget *parent)
02 : QDialog(parent)
```

```
03 {
04 sourceModel = new QStringListModel(this);
05 sourceModel->setStringList(QColor::colorNames());
06 proxyModel = new QSortFilterProxyModel(this);
07 proxyModel->setSourceModel(sourceModel);
08 proxyModel->setFilterKeyColumn(0);

09 listView = new QListView;
10 listView->setModel(proxyModel);

11 syntaxComboBox = new QComboBox;
12 syntaxComboBox->addItem(tr("Regular expression"), QRegExp::RegExp);
13 syntaxComboBox->addItem(tr("Wildcard"), QRegExp::Wildcard);
14 syntaxComboBox->addItem(tr("Fixed string"), QRegExp::FixedString);
15 …
16 }
```

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 модеди. 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.

```
01 void ColorNamesDialog::reapplyFilter()
02 {
03 QRegExp::PatternSyntax syntax =
04 QRegExp::PatternSyntax(syntaxComboBox->itemData(
05 syntaxComboBox->currentIndex()).toInt());
06 QRegExp regExp(filterLineEdit->text(), Qt::CaseInsensitive, syntax);
07 proxyModel->setFilterRegExp(regExp);
08 }
```

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 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).*

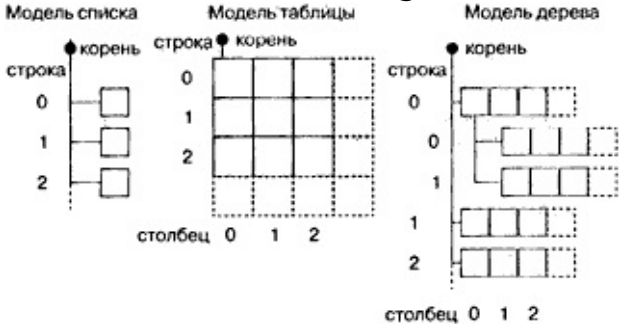


Рис. 10.9. Схематическое представление моделей Qt.

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 функции *QModelIndex::row()* и *QModelIndex::column()*. В моделях списка и таблицы родительский элемент всех остальных элементов является корневым элементом, который представдяется 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.

В первом примере этого раздела представлена модель таблицы, используемой только для чтения; она показывает курсы различных валют относительно друг друга.

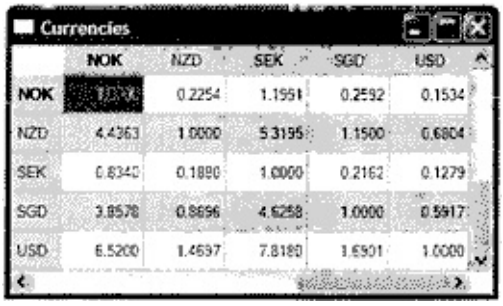


Рис. 10.10. Приложение Курсы валют (Currencies).

Это приложение можно было бы реализовать при помощи простой таблицы, но мы хотим использовать 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 × 162 = 26 244 значения; в представленной ниже пользовательской модели 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:

```
01 class CurrencyModel : public QAbstractTableModel
02 {
03 public:
04 CurrencyModel(QObject *parent = 0);
05 void setCurrencyMap(const QMap<QString, double> &map);
06 int rowCount(const QModelIndex &parent) const;
07 int columnCount(const QModelIndex &parent) const;
08 QVariant data(const QModelIndex &index, int role) const;
09 QVariant headerData(int section, Qt::Orientation orientation,
10 int role) const;
11 private:
12 QString currencyAt(int offset) const;
13 QMap<QString, double> currencyMap;
```

```
14 };
```

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*. 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.

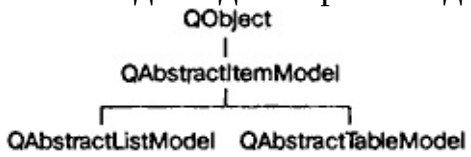


Рис. 10.11. Дерево наследования для абстрактных классов моделей.

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())*.

```
01 CurrencyModel::CurrencyModel(QObject*parent)
02 : QAbstractTableModel(parent)
03 {
04 }
```

В конструкторе нам ничего не надо делать, кроме передачи базовому классу *parent* в качестве параметра.

```
01 int CurrencyModel::rowCount(const QModelIndex &
02 /* родительский элемент */) const
03 {
04 return currencyMap.count();
05 }

06 int CurrencyModel::columnCount(const QModelIndex &
07 /* родительский элемент */) const
08 {
09 return currencyMap.count();
10 }
```

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.

```
01 QVariant CurrencyModel::data(const QModelIndex &index, int role) const
02 {
03 if (!index.IsValid())
04 return QVariant();
05 if (role == Qt::TextAlignmentRole) {
06 return int(Qt::AlignRight | Qt::AlignVCenter);
07 } else if (role == Qt::DisplayRole) {
08 QString rowCurrency = currencyAt(index.row());
09 QString columnCurrency = currencyAt(index.column());
10 if (currencyMap.value(rowCurrency) == 0.0)
```

```
11 return "####";
12 double amount = currencyMap.value(columnCurrency)
13 / currencyMap.value(rowCurrency);
14 return QString("%1").arg(amount, 0, 'f', 4);
15 }
16 return QVariant();
17 }
```

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 пользовательский делегат). Вместо этого мы возвращаем значение в виде строки, отформатированной нужным нам образом.

```
01 QVariant CurrencyModel::headerData(int section,
02 Qt::Orientation /* ориентация */, int role) const
03 {
04 if (role != Qt::DisplayRole)
05 return QVariant();
06 return currencyAt(section);
07 }
```

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*.

```
01 void CurrencyModel::setCurrencyMap(const QMap<QString, double> &map)
02 {
03 currencyMap = map;
04 reset();
05 }
```

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.

```
01 QString CurrencyModel::currencyAt(int offset) const
02 {
03 return (currencyMap.begin() + offset).key();
04 }
```

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) также используется табличная модель, но на этот раз все данные вводятся пользователем.

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)*.

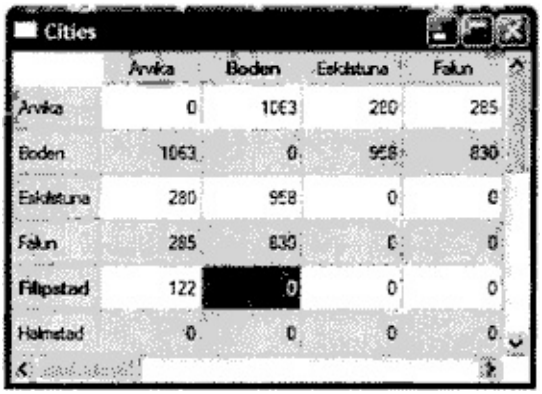


Рис. 10.12. Приложение Города.

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:

```
01 class CityModel : public QAbstractTableModel
02 {
03 Q_OBJECT
04 public:
```

```
05 CityModel(QObject *parent = 0);
06 void setCities(const QStringList &cityNames);
07 int rowCount(const QModelIndex &parent) const;
08 int columnCount(const QModelIndex &parent) const;
09 QVariant data(const QModelIndex &index, int role) const;
10 bool setData(const QModelIndex &index, const QVariant &value,
11 int role);
12 QVariant headerData(int section, Qt::Orientation orientation,
13 int role) const;
14 Qt::ItemFlags flags(const QModelIndex &index) const;
15 private:
16 int offsetOf(int row, int column) const;
17 QStringList cities;
18 QVector<int> distances;
19 };
```

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.

```
01 CityModel::CityModel(QObject *parent)
02 : QAbstractTableModel(parent)
03 {
04 }
```

Конструктор передает параметр parent базовому классу и больше ничего не делает.

```
01 int CityModel::rowCount(const QModelIndex &
02 /* родительский элемент */) const
03 {
04 return cities.count();
05 }
06 int CityModel::columnCount(const QModelIndex &
07 /* родительский элемент */) const
08 {
09 return cities.count();
10 }
```

Поскольку мы имеем квадратную матрицу городов, количество строк и столбцов равно количеству городов в нашем списке.

```
01 QVariant CityModel::data(const QModelIndex &index, int role) const
02 {
03 if (!index.isValid())
04 return QVariant();
05 if (role == Qt::TextAlignmentRole) {
06 return int(Qt::AlignRight | Qt::AlignVCenter);
07 } else if (role == Qt::DisplayRole) {
08 if (index.row() == index.column())
09 return 0;
10 int offset = offsetOf(index.row(), index.column());
```

```
11 return distances[offset];
12 }
13 return QVariant();
14 }
```

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.

```
01 QVariant CityModel::headerData(int section,
02 Qt::Orientation /* ориентация */,
03 int role) const
04 {
05 if (role == Qt::DisplayRole)
06 return cities[section];
07 return QVariant();
08 }
```

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*.

```
01 bool CityModel::setData(const QModelIndex &index,
02 const QVariant &value, int role)
03 {
04 if (index.isValid() && index.row() != index.column()
05 && role == Qt::EditRole) {
06 int offset = offsetOf(index.row(), index.column());
07 distances[offset] = value.toInt();
08 QModelIndex transposedIndex = createIndex(
09 index.column(), index.row());
10 emit dataChanged(index, index);
11 emit dataChanged(transposedIndex, transposedIndex);
12 return true;
13 }
14 return false;
15 }
```

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 и затем столбец; здесь мы передаем параметры в обратном порядке, чтобы получить индекс модели элемента, расположенного по другую строну 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 передаются 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 неуспешность редактирования.

```
01 Qt::ItemFiags CityModel::flags(const QModelIndex &index) const
02 {
03 Qt::ItemFlags flags = QAbstractItemModel::flags(index);
04 if (index.row() != index.column())
05 flags |= Qt::ItemIsEditable;
06 return flags;
07 }
```

Функция *flags()* используется моделью 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).

```
01 void CityModel::setCities(const QStringList &cityNames)
02 {
03 cities = cityNames;
04 distances.resize(cities.count() * (cities.count() - 1) / 2);
05 distances.fill(0);
06 reset();
07 }
```

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.

```
01 int CityModel::offsetOf(int row, int column) const
02 {
03 if (row < column)
04 qSwap(row, column);
05 return (row * (row - 1) / 2) + column;
06 }
```

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 × (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.

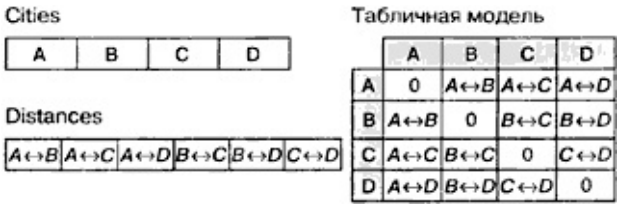


Рис. 10.13. Структуры данных cities и distances и табличная модель.

Последний пример в данном разделе представляет собой модель, которая показывает дерево грамматического разбора заданного регулярного выражения. Регулярное выражение состоит из одного или нескольких термов, разделяемых символами '|'. Так, регулярное выражение «alpha|bravo|charlie» содержит три терма. Каждый терм представляет собой последовательность из одного или нескольких факторов: например, терм «bravo» состоит из пяти факторов (каждая буква является фактором). Факторы могут состоять из атома и необязательного квантификатора (quantifier), например '*', '+' и '?'. Поскольку регулярные выражения могут иметь подвыражения, заключенные в скобки, они могут быть представлены рекурсивными деревьями грамматического разбора.

Регулярное выражение, показанное на рис. 10.14, «ab|(cd)?e» означает, что за 'a' следует 'b' или допускается два варианта: за 'c' идет 'd' и затем 'e' или просто имеется 'e'. Поэтому подойдут строки «ab» и «cde», но не подойдут строки «bc» или «cd».

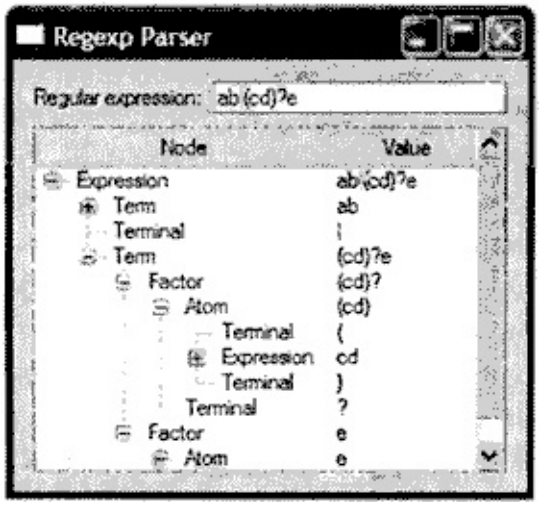


Рис. 10.14. Приложение Парсер регулярных выражений.

Приложение Парсер регулярных выражений (Regexp Parser) состоит из четырех классов:

• *RegExpWindow* — окно, которое позволяет пользователю вводить регулярное выражение и показывает соответствующее дерево грамматического разбора;

• *RegExpParser* формирует дерево грамматического разбора для заданного регулярного выражения;

• *RegExpModel* — модель дерева, используемая деревом грамматического разбора;

• *Node* (вершина) представляет один элемент в дереве грамматического разбора.

Давайте начнем с класса *Node:*

```
01 class Node {
02 public:
03 enum Type { RegExp, Expression, Term, Factor, Atom, Terminal };
04 Node(Type type, const QString &str = "");
05 ~Node();
06 Type type;
07 QString str;
```

```
08 Node *parent;
09 QList<Node *> children;
10 };
```

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).

```
01 Node::Node(Type type, const QString &str)
02 {
03 this->type = type;
04 this->str = str;
05 parent = 0;
06 }
```

Конструктор просто инициализирует тип и строку вершины. Поскольку все данные открыты, в программном коде, использующим *Node*, можно непосредственно манипулировать типом, строкой, родительским элементом и дочерними элементами.

```
01 Node::~Node()
02 {
03 qDeleteAll(children);
04 }
```

Функция *qDeleteAll()* проходит по всем указателям контейнера и вызывает оператор *delete* для каждого из них. Она не устанавливает указатели в 0, 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:

```
01 class RegExpModel : public QAbstractItemModel
02 {
03 public:
04 RegExpModel(QObject *parent = 0);
05 ~RegExpModel();
06 void setRootNode(Node *node);
07 QModelIndex index(int row, int column,
08 const QModelIndex &parent) const;
09 QModelIndex parent(const QModelIndex &child) const;
10 int rowCount(const QModelIndex &parent) const;
11 int columnCount(const QModelIndex &parent) const;
12 QVariant data(const QModelIndex &index, int role) const;
13 QVariant headerData(int section,
14 Qt::Orientation Orientation, int role) const;
15 private:
16 Node *nodeFromIndex(const QModelIndex &index) const;
17 Node *rootNode;
18 };
```

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 функции 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.

```
01 RegExpModel::RegExpModel(QObject *parent)
02 : QAbstractItemModel(parent)
03 {
04 rootNode = 0;
05 }
```

В конструкторе модели нам надо просто задать корневой вершине безопасное нулевое значение и передать указатель *parent* базовому классу.

```
01 RegExpModel::~RegExpModel()
02 {
03 delete rootNode;
04 }
```

В деструкторе мы удаляем корневую вершину. Если корневая вершина имеет дочерние вершины, то каждая из них удаляется и эта процедура повторяется рекурсивно деструктором *Node*.

```
01 void RegExpModel::setRootNode(Node *node)
02 {
03 delete rootNode;
04 rootNode = node;
05 reset();
06 }
```

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.

```
01 QModelIndex RegExpModel::index(int row, int column,
02 const QModelIndex &parent) const
03 {
04 if (!rootNode)
05 return QModelIndex();
06 Node *parentNode = nodeFromIndex(parent);
07 return createIndex(row, column, parentNode->children[row]);
08 }
```

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 достаточным 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*, используя закрытую функцию *nodeFromIndex():*

```
01 Node *RegExpModel::nodeFromIndex(
02 const QModelIndex &index) const
03 {
04 if (index.isValid()) {
05 return static_cast<Node *>(index.internalPointer());
06 } else {
07 return rootNode;
07 }
```

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.

```
01 int RegExpModel::rowCount(const QModelIndex
02 &parent) const
03 {
04 Node *parentNode = nodeFromlndex(parent);
05 if (!parentNode)
06 return 0;
07 return parentNode->children.count();
08 }
```

Число строк для заданного элемента определяется просто количеством дочерних элементов.

```
01 int RegExpModel::columnCount(const QModelIndex &
02 /* родительский элемент */) const
03 {
04 return 2;
05 }
```

Число столбцов фиксировано и равно 2. Первый столбец содержит типы вершин; второй столбец содержит значения вершин.

```
01 QModelIndex RegExpModel::parent(const QModelIndex
02 &child) const
03 {
04 Node*node = nodeFromIndex(child);
05 if (!node)
06 return QModelIndex();
07 Node *parentNode = node->parent;
08 if (!parentNode)
09 return QModelIndex();
```

```
10 Node *grandparentNode = parentNode->parent;
11 if (!grandparentNode)
12 return QModelIndex();
13 int row = grandparentNode->children.indexOf(parentNode);
14 return createIndex(row, child.column(), parentNode);
15 }
```

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).

```
01 QVariant RegExpModel::data(const QModelIndex
02 &index, int role) const
03 {
04 if (role != Qt::DisplayRole)
05 return QVariant();
06 Node *node = nodeFromIndex(index);
07 if (!node)
08 return QVariant();
09 if (index.column() == 0) {
10 switch (node->type) {
11 case Node::RegExp:
12 return tr("RegExp");
13 case Node::Expression:
14 return tr("Expression");
15 case Node::Term:
16 return tr("Term");
17 case Node::Factor:
18 return tr("Factor");
19 case Node::Atom:
20 return tr("Atom");
21 case Node::Terminal:
22 return tr("Terminal");
23 default:
24 return tr("Unknown");
25 }
26 } else if (index.column() == 1) {
27 return node->str;
28 }
29 return QVariant();
30 }
```

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 удается 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, вбзвращаем значение вершины (ее строку).

```
01 QVariant RegExpModel::headerData(int section,
02 Qt::Orientation orientation, int role) const
03 {
04 if (orientation == Qt::Horizontal && role == Qt::DisplayRole) {
05 if (section == 0) {
06 return tr("Node");
07 else if (section == 1) {
08 return tr("Value");
09 }
10 }
11 return QVariant();
12 }
```

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 *RegExpModel*, давайте посмотрим, как создается корневая вершина, когда пользователь изменяет текст в строке редактирования.

```
01 void RegExpWindow::regExpChanged(const QString&regExp)
02 {
03 RegExpParser parser;
04 Node *rootNode = parser.parse(regExp);
05 regExpModel->setRootNode(rootNode);
06 }
```

При изменении пользователем текста в строке редактирования вызывается слот главного окна *regExpChanged()*. В этом слоте выполняется синтаксический анализ введенного пользователем текста, и парсер возвращает указатель на корневую вершину дерева грамматического разбора.

Мы не показываем класс *RegExpParser*, потому что он не имеет отношения к графическому интерфейсу или программированию модели/представления. Полный исходный код для этого примера находится на компакт-диске.

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 выполняются 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) используется пользовательский делегат. В этом окне отображаются названия музыкальных фонограмм и их длительность. 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*.

Рис. 10.15. Приложение Редактор фонограмм.

Диалоговое окно Редактор фонограмм использует *QTableWidget* — удобный подкласс отображения элементов, который работает с объектами *QTableWidgetItem*. Данные представлены в виде списка фонограмм *Track:*

```
01 class Track
02 {
03 public:
04 Track(const QString &title = "", int duration = 0);
05 QString title;
06 int duration;
07 };
```

Ниже приводится фрагмент конструктора, показывающий, как создается и пополняется табличный виджет:

```
01 TrackEditor::TrackEditor(QList<Track> *tracks, QWidget *parent)
02 : QDialog(parent)
03 {
04 this->tracks = tracks;
```

```
05 tableWidget = new QTableWidget(tracks->count(), 2);
06 tableWidget->setItemDelegate(new TrackDelegate(1));
07 tableWidget->setHorizontalHeaderLabels(
08 QStringList() << tr("Track") << tr("Duration"));
09 for (int row = 0; row < tracks->count(); ++row) {
10 Track track = tracks->at(row);
11 QTableWidgetItem *item0 = new QTableWidgetItem(track.titie);
12 tableWidget->setItem(row, 0, item0);
13 QTableWidgetItem *item1 = new QTableWidgetItem(
14 QString::number(track.duration));
15 item1->setTextAlignment(Qt::AlignRight);
16 tableWidget->setItem(row, 1, item1);
17 }
18 …
19 }
```

Конструктор создает табличный виджет и, вместо того чтобы просто использовать делегата по умолчанию, связывает 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 диалогового окна *TrackEditor* нет ничего необычного, поэтому теперь рассмотрим класс *trackDelegate*, который обеспечивает воспроизведение и редактирование данных фонограммы.

```
01 class TrackDelegate : public QItemDelegate
02 {
03 Q_OBJECT
04 public:
05 TrackDelegate(int durationColumn, QObject *parent = 0);
06 void paint(QPainter *painter, const
07 QStyleOptionViewItem &option,
08 const QModelIndex &index) const;
09 QWidget *createEditor(QWidget *parent,
10 const QStyleOptionViewItem &option,
11 const QModelIndex &index) const;
12 void setEditorData(QWidget *editor,
13 const QModelIndex &index) const;
14 void setModelData(QWidget *editor,
15 QAbstractItemModel *model,
16 const QModelIndex &index) const;
17 private slots:
18 void commitAndCloseEditor();
19 private:
20 int durationColumn;
21 };
```

Мы используем *QItemDelegate* в качестве нашего базового класса, чтобы можно было

воспользоваться возможностями делегата по умолчанию. Так же мы могли бы использовать *QAbstractItemDelegate*, если бы хотели начать с чистого листа. 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.

```
01 TrackDelegate::TrackDelegate(int durationColumn, QObject *parent)
02 : QItemDelegate(parent)
03 {
04 this->durationColumn = durationColumn;
05 }
```

Параметр конструктора *durationColumn* указывает делегату, какой номер столбца содержит длительность фонограммы.

```
01 void TrackDelegate::paint(QPainter *painter,
02 const QStyleOptionViewItem &option,
03 const QModelIndex &index) const
04 {
05 if (index.column() == durationColumn) {
06 int secs = index.model()->data(index, Qt::DisplayRole).toInt();
07 QString text= QString("%1:%2")
08 .arg(secs/60, 2, 10, QChar('0'))
09 .arg(secs % 60, 2, 10, QChar('0'));
10 QStyleOptionViewItem myOption = option;
11 myOption.displayAlignment = Qt::AlignRight | Qt::AlignVCenter;
12 drawDisplay(painter, myOption, myOption.rect, text);
13 drawFocus(painter, myOption, myOption.rect);
14 } else {
15 QItemDelegate::paint(painter, option, index);
16 }
17 }
```

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 делая в противном случае. Функцией *drawDisplay()* очень удобно пользоваться, особенно совместно с нашими собственными опциями стиля. Мы могли бы также рисовать, используя рисовальщик непосредственно.

```
01 QWidget *TrackDelegate::createEditor(QWidget *parent,
02 const QStyleOptionViewItem &option,
03 const QModelIndex &index) const
04 {
05 if (index.column() == durationColumn) {
```

```
06 QTimeEdit *timeEdit = new QTimeEdit(parent);
07 timeEdit->setDisplayFormat("mm:ss");
08 connect(timeEdit, SIGNAL(editingFinished()),
09 this, SLOT(commitAndCloseEditor()));
10 return timeEdit;
11 } else {
12 return QItemDelegate::createEditor(parent, option, index);
13 }
14 }
```

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.

```
01 void TrackDelegate::commitAndCloseEditor()
02 {
03 QTimeEdit *editor = qobject_cast<QTimeEdit *>(sender());
04 emit commitData(editor);
05 emit closeEditor(editor);
06 }
```

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.

```
01 void TrackDeIegate::setEditorData(QWidget *editor,
02 const QModelindex &index) const
03 {
04 if (index.column() == durationColumn) {
05 int secs = index.model()->data(index, Qt::DisplayRole).toInt();
06 QTimeEdit *timeEdit = qobject_cast<QTimeEdit *>(editor);
07 timeEdit->setTime(QTime(0, secs / 60, secs % 60));
08 } else {
09 QItemDelegate::setEditorData(editor, index);
10 }
11 }
```

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.

```
01 void TrackDelegate::setModelData(QWidget *editor,
02 QAbstractItemModel *model, const QModelIndex &index) const
03 {
04 if (index.column() == durationColumn) {
05 QTimeEdit *timeEdit = qobject_cast<QTimeEdit *>(editor);
06 QTime time = timeEdit->time();
07 int secs = (time.minute() * 60) + time.second();
08 model->setData(index, secs);
09 } else {
10 QItemDelegate::setModelData(editor, model, index);
11 }
12 }
```

Если пользователь прекращает редактирование (например, щелкнув левой кнопкой мышки 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.

Кррме 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 онлайновой документации Qt всесторонне рассматриваются вопросы программирования классов по отображению элементов. См. http://doc.trolltech.com/4.1/model-view.html, где приводится список всех таких классов, и http://doc.trolltech.com/4.1/model-view-programming.html, где даются дополнительная информация и ссылки на соответствующие примеры, включенные в Qt.

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 Qtopia Core (версия Qt для мобильных устройств), 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*, которым 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. 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 неэффективны.

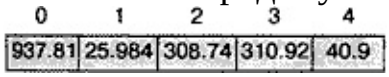


Рис. 11.1. Вектор чисел двойной точности.

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. 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 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 наследуют *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 сбит 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:

```
01 class Movie
02 {
03 public:
04 Movie(const QString &title = "", int duration = 0);
05 void setTitle(const QString &title) { myTitle = title; }
06 QString title() const { return myTitle; }
07 void setDuration(int duration) { myDuration = duration; }
08 QString duration() const { return myDuration; }
```

```
09 private:
10 QString myTitle;
11 int myDuration;
12 };
```

Этот класс имеет конструктор, для которого необязательно 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 могут использоваться два типа итераторов в стиле Java: итератор, используемый только для чтения, и итератор, используемый как для чтения, так и для записи. Классами итераторов первого типа являются *QVectorIterator<T>*, *QLinkedListIterator<T>* и *QListIterator<T>*. Соответствующие итераторы чтения—записи имеют слово *Mutable* (изменчивый) 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.

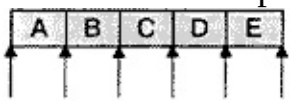


Рис. 11.3. Допустимые позиции итераторов в стиле Java.

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.

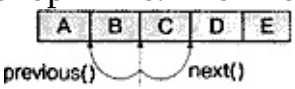


Рис. 11.4. Влияние функций previous() и next() на итераторы в стиле Java.

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 так же 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). 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().*

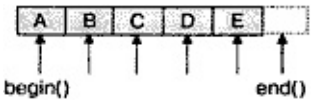


Рис. 11.5. Допустимые позиции итераторов в стиле STL.

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 даннъис (implicit sharing)*. Это означает, что операция копирования Qt—контейнера выполняется почти так же 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:

```
01 QVector<double> sineTable()
02 {
03 QVector<double> vect(360);
04 for (int i = 0; i <360; ++i)
05 vect[i] = sin(i / (2 * M_PI));
06 return vect;
07 }
```

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:

```
01 using namespace std;
02 void sineTable(vector<double> &vect)
03 {
04 vect.resize(360);
05 for (int i = 0; i < 360; ++i)
06 vect[i] = sin(i / (2 * M_PI));
07 }
```

В результате вызов будет не столь простым и менее понятным:

```
vector<double> table;
sineTable(table);
```

В Qt применяется неявное совместное использование данных во всех ее 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") {
cout << "Found Citizen Kane" << endl;
break;
}
}
```

Псевдоключевое 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, необязательно 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") {
cout << "Found Citizen Kane" << 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 *QString* ссылается 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 значения «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 счетчиками. 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. Такая организация данных обеспечивает хорошую производительность операций поиска и вставки, а также при проходе данных в порядке их сортировки. Внутренне *QMap<K, T>* реализуется как слоеный список (skip—list).

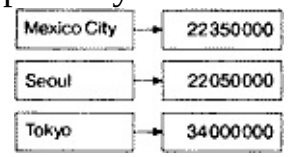


Рис. 11.6. Ассоциативный массив, связывающий QString с int.

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 — 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 нуль. 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 контейнер *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 доступен 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» итераторы 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" << "Mariette";
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());
```

Алгоритм *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 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, и для их удаления 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 значений.

Строки, массивы байтов и объекты произвольного типа

QString, *QByteArray* и *QVariant* — три класса, которые имеют много общего с контейнерами и 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 применяются в любой программе 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 17.

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 на «1950» и «%4» заменяется на «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 «%1».

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 так далее. 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 код выводит на консоль слово «pays»[6]:

QString str = "polluter pays principle";

qDebug() << str.mid(9, 4);

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);

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"))

Это проще и быстрее, чем:

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 не зависимым от регистра, мы можем использовать функции *toUpper()* или *toLower()*. Например:

if (fileName.toLower() == "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), затем мы вставляем слово «sunny» в позицию 2.

Существуют перегруженные версии функции *replace(),* которые заменяют все подстроки, совпадающие со значением первого аргумента, вторым аргументом. Например, ниже показано, как можно заменить все символы «&» в строке на «&»:

str.replace("&", "&");

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

_ _ _BOB_\t_THE_ _\nDOG_\n

Строка, возвращаемая функцией *trimmed(),* имеет вид

BOB_\t_THE_ _\nDOG

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

BOB_THE_DOG

Строку можно разбить на подстроки типа *QStringList* при помощи функции *QList::split():*

QString str = "polluter pays principle";

QStringList words = str.split(" ");

В приведенном выше примере мы разбиваем строку «polluter pays principle» на три подстроки: «polluter», «pays» и «principle». Функция *split()* имеет необязательный третий 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 **, используя *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>*.

Широкое распространение получило применение этого типа в классах отображения элементов, в модуле баз данных и в классе *QSettings*, 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 «Peag», 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, как правило, 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

неграфического интерфейса и типом *QVariant*, однако на практике обычно используются функции преобразования вида *to…()* (например, *toString()*) для преобразования типов неграфического 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>()*.

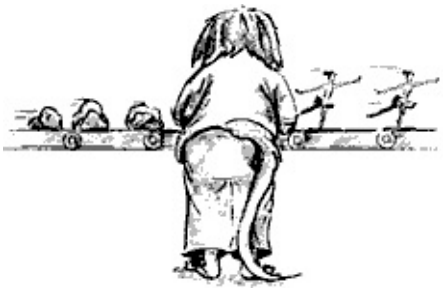
Если в пользовательском типе данных предусмотрены операторы << и >> для записи и чтения из потока данных *QDataStream*, их можно зарегистрировать, используя функцию *qRegisterMetaTypeStreamOperators<T>()*. Это позволяет, среди прочего, хранить параметры настройки пользовательских типов данных, используя *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 *QBitArray*, который мы будем использовать в первом разделе главы 19. 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.

Алгоритмы Qt, включая несколько не рассмотренных здесь, например *qCopyBackward()* и *qEqual()*, описаны в документации Qt, которую можно найти по адресу http://doc.trolltech.com/4.1/algorithms.html. Более подробное описание контейнеров Qt, в том числе информацию об их временных и объемных характеристиках, можно найти на странице http://doc.trolltech.com/4.1/containers.html.

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:*

• *QFile* — получает доступ к файлам, находящимся в локальной файловой системе или внедренным в исполняемый модуль,

• *QTemporaryFile* — создает временные файлы в локальной файловой системе и получает доступ к ним,

• *QBuffer* — считывает или записывает данные в *QByteArray*,

• *QProcess* — запускает внешние программы и обеспечивает связь между процессами,

• *QTcpSocket* — передает поток данных по сети, используя протокол TCP,

• *QUdpSocket* — передает и принимает из сети дейтаграммы UDP.

QProcess, QTcpSocket и *QUdpSocket* 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 будут рассмотрены отдельно в главах 14 и 15, специально им посвященным.

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)) {
cerr << "Cannot open file for writing: "
<< qPrintable(file.errorString()) << endl;
return;
}
QDataStream out(&file);
out.setVersion(QDataStream::Qt_4_1);
out << quint32(0x12345678) << image << map;
```

Если не удается 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::toStdString()*, 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 4.1 большинство сложных форматов имеют версию 7. Мы можем либо жестко закодировать в программе константу 7, либо использовать символическое имя *QDataStream::Qt_4_1*.

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)) {
cerr << "Cannot open file for reading: "
<< qPrintable(file.errorString()) << endl;
return;
}
QDataStream in(&file);
in.setVersion(QDataStream::Qt_4_1);
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 4.1 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/4.1/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:*

```
01 class Painting
02 {
03 public:
04 Painting() { myYear = 0; }
05 Painting(const QString &title, const QString &artist, int year) {
06 myTitle = title;
07 myArtist = artist;
08 myYear = year;
09 }
10 void setTitle(const QString &title) { myTitle = title; }
11 QString title() const { return myTitle; }
```

```
12 …
13 private:
14 QString myTitle;
15 QString myArtist;
16 int myYear;
17 };
18 QDataStream &operator << (QDataStream &out, const Painting &painting);
19 QDataStream &operator >> (QDataStream &in, Painting &painting);
```

Ниже показана возможная реализация оператора *<<:*

```
01 QDataStream &operator << (QDataStream &out, const Painting &painting)
02 {
03 out << painting.title() << painting.artist()
04 << quint32(painting.year());
05 return out;
06 }
```

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 <<.

```
01 QDataStream &operator >> (QDataStream &in, Painting &painting)
02 {
03 QString title;
04 QString artist;
05 quint32 year;
06 in >> title >> artist >> year;
07 painting = Painting(title, artist, year);
08 return in;
09 }
```

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 бы то 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_1*. 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_1*, 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 ни был). При считывании файла мы считываем номер версии потока:

```
01 quint32 magic;
02 quint16 streamVersion;
03 QDataStream in(&file);
04 in >> magic >> streamVersion;
05 if (magic != MagicNumber) {
06 cerr << "File is not recognized by this application" << endl;
07 return false;
08 } else if (streamVersion > in.version()) {
09 cerr << "File is from a more recent version of the application"
10 << endl;
11 return false;
12 }
13 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. 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_1);
out << quint32(MagicNumber) << quint16(0x0103);
```

При считывании данных мы определяем версию *QDataStream* на основе номера версии

приложения:

```
01 QDataStream in(&file);
02 in >> magic >> appVersion;
03 if (magic != MagicNumber) {
04 cerr << "File is not recognized by this application" << endl;
05 return false;
06 } else if (appVersion > 0x0103) {
07 cerr << "File is from a more recent version of the application"
08 << endl;
09 return false;
10 }
11 if (appVersion < 0x0103) {
12 in.setVersion(QDataStream::Qt_3_0);
13 } else {
14 in.setVersion(QDataStream::Qt_4_1);
15 }
```

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 7 потока данных (Qt_4_1).

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 новой версией приложения, записанных в старой версии, 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*, а вместо этого мы должны вызывать функции *write()* и *readAll()* класса *QIODevice*. Например:

```
01 bool copyFile(const QString &source, const QString &dest)
02 {
03 QFile sourceFile(source);
04 if (!sourceFile.open(QIODevice::ReadOnly))
05 return false;
06 QFile destFile(dest);
07 if (!destFile.open(QIODevice::WriteOnly))
08 return false;
09 destFile.write(sourceFile.readAll());
10 return sourceFile.error() == QFile::NoError
11 && destFile.error() == QFile::NoError;
12 }
```

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 большему 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.

Существуют другие сценарии, когда 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 сокеты). 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 19, 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 файлы исходных текстов программ. Работа с XML—файлами рассматривается отдельно в главе 15.

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)) {
cerr << "Cannot open file for writing: "
<< qPrintable(file.errorString()) << 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 << "Norway" << "Sweden";
```

Если out является объектом типа *QTextStream,* то данные в действительности записываются в виде строки «NorwaySweden». Мы не можем рассчитывать на то, что приведенная ниже строка правильно считает данные:

```
in >> str1 >> str2;
```

Фактически произойдет то, что строка *str1* получит все слово «NorwaySweden», 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 17 («Интернационализация»).

QTextStream имеет различные опции, аналогичные опциям <*iostream*>. Установить опции можно путем передачи в поток специальных объектов — манипуляторов потока. В следующем примере устанавливаются опции *showbase*, *uppercasedigits* и *hex* перед выводом целого числа 12345678, и в результате получается текст «0xBC614E»:

```
out << showbase << uppercasedigits << hex << 12345678;
```

Ниже перечислены функции, устанавливающие опции для *QTextStream* (рис. 12.1):

• 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).

Опции можно также устанавливать с помощью функций—членов:

```
out.setNumberFlags(QTextStream::ShowBase
| QTextStream::UppercaseDigits);
out.setIntegerBase(16);
```

```
out << 12345678;
```

Класс *QTextStream*, как и *QDataStream*, работает с каким-нибудь подклассом *QIODevice: QFile, QTemporaryFile, QBuffer, QProcess, QTcpSocket* или *QUdpSocket*. 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 непустую 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. Всю эту работу делает функция *tidyFile():*

```
01 void tidyFile(QIODevice *inDevice, QIODevice *outDevice)
02 {
03 QTextStream in(inDevice);
04 QTextStream out(outDevice);
05 const int TabSize = 8;
06 int endlCount = 0;
07 int spaceCount = 0;
08 int column = 0;
09 QChar ch;
10 while (!in.atEnd()) {
11 in >> ch;
12 if (ch == '\n') {
13 ++endlCount;
14 spaceCount = 0;
15 column = 0;
16 } else if (ch == '\t') {
17 int size = TabSize - (column % TabSize);
18 spaceCount += size;
19 column += size;
20 } else if (ch == ' ') {
21 ++spaceCount;
22 ++column;
23 } else {
24 while (endlCount > 0) {
25 out << endl;
26 --endlCount;
27 column = 0;
28 }
29 while (spaceCount > 0) {
30 out << ' ';
31 --spaceCount;
32 ++column;
33 }
34 out << ch;
35 ++column;
36 }
37 }
```

```
38 out << endl;
39 }
```

Мы создаем для ввода и вывода данных объекты *QTextStream*, полученные на базе устройств *QIODevice*, переданных конструктору. Мы поддерживаем три переменные для 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.

```
01 int main()
02 {
03 QFile inFile;
04 QFile outFile;
05 inFile.open(stdin, QFile::ReadOnly);
06 outFile.open(stdout, QFile::WriteOnly);
07 tidyFile(&inFile, &outFile);
08 return 0;
09 }
```

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/4.1/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 приложений 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:

```
01 qlonglong imageSpace(const QString &path)
02 {
03 qlonglong size = 0;
04 QDir dir(path);
05 QStringList filters;
06 foreach (QByteArray format, QImageReader::supportedImageFormats())
07 filters += "*." + format;
08 foreach (QString file, dir.entryList(filters, QDir::Files))
09 size += QFileInfo(dir, file).size();
10 foreach (QString subDir, dir.entryList(QDir::Dirs
11 | QDir::NoDotAndDotDot))
12 size += imageSpace(path + QDir::separator() + subDir);
13 return size;
14 }
```

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*. Второй аргумент 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 времена 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 подсоединяем имя подкаталога, 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::convertSeparators()* для преобразования слешей в соответствующий разделитель конкретной платформы.

Давайте добавим функцию *main()* в нашу небольшую программу:

```
01 int main(int argc, char *argv[])
```

```
02 {
03 QCoreApplication app(argc, argv);
04 QStringList args = app.arguments();
05 QString path = QDir::currentPath();
06 if (args.count() > 1)
07 path = args[1];
08 cout << "Space used by images in " << qPrintable(path)
09 << " and its subdirectories is "
10 << (imageSpace(path) / 1024) << " KB" << endl;
11 return 0;
12 }
```

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()*.

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:

```
<!DOCTYPE RCC><RCC version="1.0">
<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, используя *QFile*.

Преимуществом внедрения данных в исполняемый модуль является невозможность их потери и возможность создания действительно автономных исполняемых модулей (если использовалась статическая компоновка). Двумя недостатками являются 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/4.1/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.

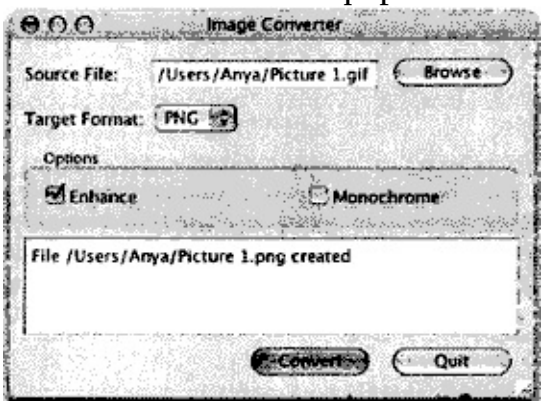


Рис. 12.2. Приложение Image Converter.

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:

```
01 #ifndef CONVERTDIALOG_H
02 #define CONVERTDIALOG_H
03 #include <QDialog>
04 #include <QProcess>
05 #include "ui_convertdialog.h"
06 class ConvertDialog : public QDialog,
07 public Ui::ConvertDialog
08 {
09 Q_OBJECT
10 public:
11 ConvertDialog(QWidget *parent = 0);
12 private slots:
13 void on_browseButton_clicked();
14 void on_convertButton_clicked();
15 void updateOutputTextEdit();
16 void processFinished(int exitCode, QProcess::ExitStatus exitStatus);
17 void processError(QProcess::ProcessError error);
18 private:
19 QProcess process;
```

```
20 QString targetFile;
21 };
22 #endif
```

Этот заголовочный файл создается по тому знакомому образцу, который используется в подклассах форм *Qt Designer*. Благодаря механизму автоматического связывания *QtDesigner* слоты *on_browseButton_clicked()* и *on_convertButton_clicked()* автоматически связываются с сигналом *clicked()* кнопок Browse (просмотреть) и Convert (преобразовать).

```
01 ConvertDialog::ConvertDialog(QWidget *parent)
02 : QDialog(parent)
03 {
04 setupUi(this);
05 connect(&process, SIGNAL(readyReadStandardError()),
06 this, SLOT(updateOutputTextEdit()));
07 connect(&process, SIGNAL(finished(int, QProcess::ExitStatus)),
08 this, SLOT(processFinished(int, QProcess::ExitStatus)));
09 connect(&process, SIGNAL(error(QProcess::ProcessError)),
10 this, SLOT(processError(QProcess::ProcessError)));
11 }
```

Вызов *setupUi()* создает и компонует все виджеты форм, устанавливает соединения сигнал—слот для слотов *on_objectName_signalName()* и связывает кнопку Quit (выйти) с функцией *QDialog::accept()*. 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()*.

```
01 void ConvertDialog::on_browseButton_clicked()
02 {
03 QString initialName = sourceFileEdit->text();
04 if (initialName.isEmpty())
05 initialName = QDir::homePath();
06 QString fileName = QFileDialog::getOpenFileName(this,
07 tr("Choose File"), initialName);
08 fileName = QDir::convertSeparators(fileName);
09 if (!fileName.isEmpty()) {
10 sourceFileEdit->setText(fileName);
11 convertButton->setEnabled(true);
12 }
13 }
```

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.

```
01 void ConvertDialog::on_convertButton_clicked()
02 {
03 QString sourceFile = sourceFileEdit->text();
04 targetFile = QFileInfo(sourceFile).path()
05 + QDir::separator() + QFileInfo(sourceFile).baseName()
```

```
06 + "." + targetFormatComboBox->currentText().toLower();
07 convertButton->setEnabled(false);
08 outputTextEdit->clear();
09 QStringList args;
10 if (enhanceCheckBox->isChecked())
11 args << "-enhance";
12 if (monochromeCheckBox->isChecked())
13 args << "-monochrome";
14 args << sourceFile << targetFile;
15 process.start("convert", args);
16 }
```

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.

```
01 void ConvertDialog::updateOutputTextEdit()
02 {
03 QByteArray newData = process.readAllStandardError();
04 QString text = outputTextEdit->toPlainText()
05 + QString::fromLocal8Bit(newData);
06 outputTextEdit->setPlainText(text);
07 }
```

При всякой записи внешним процессом в поток *cerr* вызывается слот *updateOutputTextEdit()*. Мы считываем текст сообщения об ошибке и добавляем его в существующий текст *QTextEdit.*

```
01 void ConvertDialog::processFinished(int exitCode,
02 QProcess::ExitStatus exitStatus)
03 {
04 if (exitStatus == QProcess::CrashExit) {
05 outputTextEdit->append(tr("Conversion program crashed"));
06 } else if (exitCode != 0) {
07 outputTextEdit->append(tr("Conversion failed"));
08 } else {
09 outputTextEdit->append(tr("File %1 created").arg(targetFile));
10 }
11 convertButton->setEnabled(true);
```

```
12 }
```

По окончании процесса мы уведомляем пользователя о результате и включаем кнопку Convert.

```
01 void ConvertDialog::processError(QProcess::ProcessError error)
02 {
03 if (error == QProcess::FailedToStart) {
04 outputTextEdit->append(tr("Conversion program not found"));
05 convertButton->setEnabled(true);
06 }
07 }
```

Если процесс не удается 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()*.

```
01 void ExternalEditor::edit()
02 {
03 QTemporaryFile outFile;
04 if (!outFile.open())
05 return;
06 QString fileName = outFile.fileName();
07 QTextStream out(&outFile);
08 out << textEdit->toPlainText();
09 outFile.close();
10 QProcess::execute(editor, QStringList() << options << fileName);
11 QFile inFile(fileName);
12 if (!inFile.open(QIODevice::ReadOnly))
13 return;
14 QTextStream in(&inFile);
15 textEdit->setPlainText(in.readAll());
16 }
```

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).

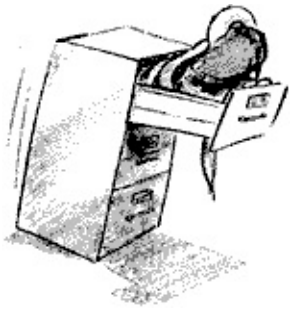
После закрытия пользователем текстового редактора процесс завершает свою работу и функция *execute()* возвращает управление. Затем мы открываем временный файл и считываем его содержимое в *QTextEdit*. *QTemporaryFile* автоматически удаляет временный файл, когда объект выходит из области видимости.

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, например веб—браузера или почтового клиента. 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.

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:

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 6.x и 7.x,
QSQLITE — SQLite версии 3 и выше,
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 данных.

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 «главная—подчиненные» (master—detail), и возможность многократной детализации выводимых на экран данных (drill-down), что продемонстрируют некоторые 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:

```
01 bool createConnection()
02 {
03 QSqlDatabase *db = QSqlDatabase::addDatabase("QOCI8");
04 db->setHostName("mozart.konkordia.edu");
05 db->setDatabaseName("musicdb");
06 db->setUserName("gbatstone");
07 db->setPassword("T17aV44");
08 if (!db->open()) {
09 db->lastError().showMessage();
10 return false;
11 }
12 return true;
13 }
```

Во-первых, мы вызываем функцию *QSqlDatabase::addDatabase()* для создания объекта *QSqlDatabase*. Первый аргумент функции *addDatabase()* задает драйвер базы данных, который Qt должна использовать для доступа к базе данных. В данном случае мы используем MySQL *(??? — в коде QOCI, Oracle Call Interface)*.

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():*

```
01 int main(int argc, char *argv[])
02 {
03 QApplication app(argc, argv);
04 if (!createConnection())
05 return 1;
06 return app.exec();
07 }
```

После установки соединения мы можем применять *QSqlQuery* для выполнения любой инструкции SQL, поддерживаемой используемой базой данных. Ниже приводится пример выполнения команды *SELECT:*

```
QSqlQuery query;
query.exec("SELECT title, year FROM cd WHERE year >= 1998");
```

После вызова функции *exec()* мы можем просмотреть результат запроса:

```
while (query.next()) {
QString title = query.value(0).toString();
int year = query.value(1).toInt();
cerr << qPrintable(title) << ": " << year << 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 просмотра результирующего набора.

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, 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).tolnt();
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.

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();
cerr << qPrintable(title) << ": " << year << 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();
cerr << qPrintable(title) << ": " << year << endl;
}
```

Для вставки записи в таблицу базы данных мы действуем так же, как если бы делали вставку в двумерную модель: сначала вызываем функцию *insertRow()* для создания новой пустой строки (записи) и затем 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()* указываются 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.

Представление данных в табличной форме

Во многих случаях табличное представление является самым простым представлением набора данных для пользователей. В этом и последующих разделах мы рассмотрим простое приложение CD Collection (Коллекция компакт-дисков), в котором модель *QSqlTableModel* и ее подкласс *QSqlRelationalTableModel* используются для просмотра и взаимодействия пользователей с данными, хранимыми в базе данных.

Главная форма показывает представление «master—detail» для компакт-дисков и дорожек текущего компакт-диска (рис. 13.1).

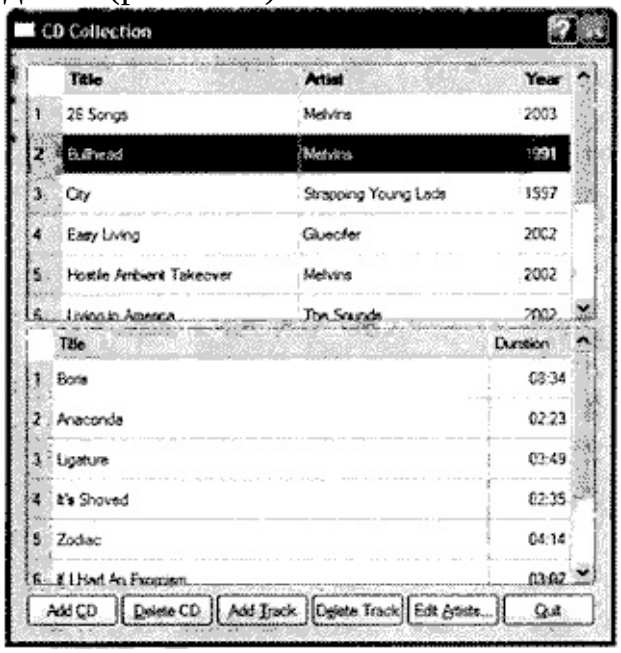


Рис. 13.1. Приложение CD Collection.

В приложении используются три таблицы, определенные следующим образом:

```
CREATE TABLE artist (
id INTEGER PRIMARY KEY,
name VARCHAR(40) NOT NULL,
country VARCHAR(40));
CREATE TABLE cd (
id INTEGER PRIMARY KEY,
title VARCHAR(40) NOT NULL,
artistid INTEGER NOT NULL,
year INTEGER N0T NULL,
FOREIGN KEY (artistid) REFERENCES artist);
CREATE TABLE track (
id INTEGER PRIMARY KEY,
title VARCHAR(40) NOT NULL,
duration INTEGER NOT NULL,
cdid INTEGER NOT NULL,
FOREIGN KEY (cdid) REFERENCES cd);
```

Некоторые базы данных не поддерживают внешние ключи. В этом случае мы должны убрать фразы *FOREIGN KEY.* Пример будет все-таки работать, но база данных не будет поддерживать целостность на уровне ссылок.

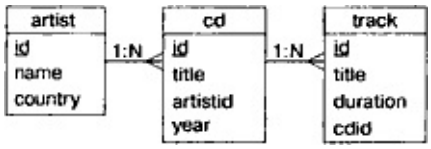


Рис. 13.2. Таблицы приложения CD Collection.

В этом разделе мы создадим диалоговое окно, позволяющее пользователю редактировать список артистов, используя простую форму с таблицей. Пользователь может вставлять, обновлять или удалять артистов при помощи кнопок формы. Обновления можно делать напрямую, просто редактируя текст ячеек. Изменения вносятся в базу данных при нажатии пользователем кнопки Enter или при переходе на другую запись.

Рис. 13.3. Диалоговое окно ArtistForm.

Ниже приводится определение класса для диалогового окна ArtistForm:

```
01 class ArtistForm : public QDialog
02 {
03 Q_OBJECT
04 public:
05 ArtistForm(const QString &name, QWidget *parent = 0);
06 private slots:
07 void addArtist();
08 void deleteArtist();
09 void beforeInsertArtist(QSqlRecord &record);
10 private:
11 enum {
12 Artist_Id = 0,
13 Artist_Name = 1,
14 Artist_Country = 2
15 };
16 QSqlTableModel *model;
17 QTableView *tableView;
18 QPushButton *addButton;
19 QPushButton *deleteButton;
20 QPushButton *closeButton;
21 };
```

Конструктор этого класса очень похож на конструктор, который использовался бы для создания формы, построенной для модели, отличной от SQL—модели:

```
01 ArtistForm::ArtistForm(const QString &name, QWidget *parent)
02 : QDialog(parent)
03 {
```

```
04 model = new QSqlTableModel(this);
05 model->setTable("artist");
06 model->setSort(Artist_Name, Qt::AscendingOrder);
07 model->setHeaderData(Artist_Name, Qt::Horizontal, tr("Name"));
08 model->setHeaderData(Artist_Country, Qt::Horizontal, tr("Country"));
09 model->select();
10 connect(model, SIGNAL(beforeInsert(QSqlRecord &)),
11 this, SLOT(beforeInsertArtist(QSqlRecord &)));
12 tableView = new QTableView;
13 tableView->setModel(model);
14 tableView->setColumnHidden(Artist_Id, true);
15 tableView->setSelectionBehavior(QAbstractItemView::SelectRows);
16 tableView->resizeColumnsToContents();
17 for (int row = 0; row < model->rowCount(); ++row) {
18 QSqlRecord record = model->record(row);
19 if (record.value(Artist_Name).toString() == name) {
20 tableView->selectRow(row);
21 break;
22 }
23 }
24 …
25 }
```

Конструктор начинается с создания объекта *QSqlTableModel*. Мы передаем *this* в качестве родителя, чтобы владельцем модели стала форма. Нами выбрана сортировка по столбцу 1 (задается константой *Artist_Name)*, который соответствует полю имени. Если бы мы не задали заголовки столбцов, то использовались бы имена полей. Мы предпочитаем их указать, чтобы обеспечить правильный регистр и локализацию.

Затем создается *QTableView* для визуального отображения модели. Мы не показываем поле *id* и устанавливаем такую ширину столбцов, которая будет достаточна для размещения в них текста без необходимости вывода многоточия.

Конструктор *ArtistForm* принимает имя артиста, который будет выбран при выводе на экран диалогового окна. Мы проходим по записям таблицы *artist* и выбираем этого артиста. Остальная часть программного кода конструктора используется для создания кнопок и подключения к ним слотов, а также для компоновки дочерних виджетов в диалоговом окне.

```
01 void ArtistForm::addArtist()
02 {
03 int row = model->rowCount();
04 model->insertRow(row);
05 QModelIndex index = model->index(row, Artist_Name);
06 tableView->setCurrentIndex(index);
07 tableView->edit(index);
08 }
```

Для добавления нового артиста мы вставляем одну пустую строку в конец табличного представления *QTableView*. Теперь пользователь может вводить имя нового артиста и его страну. Если пользователь подтверждает вставку, нажимая кнопку Enter, генерируется

сигнал *beforeInsert()*, и после этого новая запись вставляется в базу данных.

```
01 void ArtistForm::beforeInsertArtist(QSqlRecord &record)
02 {
03 record.setValue("id", generateId("artist"));
04 }
```

В конструкторе мы связываем сигнал модели *beforeInsert()* с этим слотом. Мы передаем неконстантную ссылку на запись непосредственно перед ее вставкой в базу данных. Здесь мы устанавливаем значение поля *id*.

Поскольку нам потребуется вызывать функцию *generateId()* несколько раз, мы определяем ее как inline—функцию в заголовочном файле и включаем ее каждый раз по мере необходимости. Ниже дается простой (и неэффективный) способ ее реализации:

```
01 inline int generateId(const QString &table)
02 {
03 QSqlQuery query;
04 query.exec("SELECT MAX(id) FROM " + table);
05 int id = 0;
06 if (query.next())
07 id = query.value(0).tolnt() + 1;
08 return id;
09 }
```

Функция *generateId()* может гарантированно работать правильно, если она выполняется в рамках контекста одной транзакции соответствующей команды *INSERT*. Некоторые базы данных поддерживают средство автоматической генерации полей, и обычно значительно лучше использовать предусмотренные в базе данных специальные средства поддержки этой операции.

Удаление — это последняя операция, которую позволяет сделать диалоговое окно *ArtistForm*. Вместо каскадного удаления (вскоре будет рассмотрено) мы разрешаем удалять артистов только в том случае, если в коллекции нет их компакт-дисков.

```
01 void ArtistForm::deleteArtist()
02 {
03 tableView->setFocus();
04 QModelIndex index = tableView->currentIndex();
05 if (!index.isValid())
06 return;
07 QSqlRecord record = model->record(index.row());
08 QSqlTableModel cdModel;
09 cdModel.setTable("cd");
10 cdModel.setFilter("artistid = " + record.value("id").toString());
11 cdModel.select();
12 if (cdModel.rowCount() == 0) {
13 model->removeRow(tableView->currentIndex().row());
14 } else {
15 QMessageBox::information(this, tr("Delete Artist"),
16 tr("Cannot delete %1 because there are CDs associated "
17 "with this artist in the collection.")
```

```
18 .arg(record.value("name").toString()));
19 }
20 }
```

Если выделена какая-то запись, мы проверяем наличие компакт-дисков у данного артиста, и если они отсутствуют, мы сразу же удаляем эту запись артиста. В противном случае мы выводим на экран окно с сообщением о причине невыполнения удаления. Строго говоря, здесь следовало бы использовать транзакцию, потому что из программного кода видно, что между вызовами функций *cdModel.select()* и *model->removeRow()* у артиста может появиться свой компакт-диск. Транзакция будет рассмотрена в следующем разделе.

Создание форм по технологии «master—detail»

Теперь мы рассмотрим главную форму, которая реализует подход «master—detail». Главный вид представляет собой список компакт-дисков. Вид описания деталей представляет собой список дорожек текущего компакт-диска. Это диалоговое окно является главным окном приложения CD Collection (Коллекция компакт-дисков); оно показано на рис. 13.1.

```
01 class MainForm : public QWidget
02 {
03 Q_OBJECT
04 public:
05 MainForm();
06 private slots:
07 void addCd();
08 void deleteCd();
09 void addTrack();
10 void deleteTrack();
11 void editArtists();
12 void currentCdChanged(const QModelIndex &index);
13 void beforeInsertCd(QSqlRecord &record);
14 void beforeInsertTrack(QSqlRecord &record);
15 void refreshTrackViewHeader();
16 private:
17 enum {
18 Cd_Id = 0,
19 Cd_Title = 1,
20 Cd_ArtistId = 2,
21 Cd_Year = 3
22 };
23 enum {
24 Track_Id = 0,
25 Track_Title = 1,
26 Track_Duration = 2,
27 Track_CdId = 3
28 };
29 QSqlRelationalTableModel *cdModel;
30 QSqlTableModel *trackModel;
31 QTableView *cdTableView;
32 QTableView *trackTableView;
33 QPushButton *addCdButton;
34 QPushButton *deleteCdButton;
35 QPushButton *addTrackButton;
36 QPushButton *deleteTrackButton;
37 QPushButton *editArtistsButton;
```

```
38 QPushButton *quitButton;
39 };
```

Мы используем для таблицы компакт-дисков *cd* модель *QSqlRelationalTableModel*, а не простую модель *QSqlTableModel*, потому что нам придется работать с внешними ключами. Мы рассмотрим по очереди все функции, начиная с конструктора, который мы разобьем на несколько секций из-за его большого размера.

```
01 MainForm::MainForm()
02 {
03 cdModel = new QSqlRelationalTableModel(this);
04 cdModel->setTable("cd");
05 cdModel->setRelation(Cd_ArtistId,
06 QSqlRelation("artist", "id", "name"));
07 cdModel->setSort(Cd_Title, Qt::AscendingOrder);
08 cdModel->setHeaderData(Cd_Title, Qt::Horizontal, tr("Title"));
09 cdModel->setHeaderData(Cd_ArtistId, Qt::Horizontal, tr("Artist"));
10 cdModel->setHeaderData(Cd_Year, Qt::Horizontal, tr("Year"));
11 cdModel->select();
```

Конструктор начинается с настройки модели *QSqlRelationalTableModel*, которая управляет таблицей *cd*. Вызов *setRelation()* указывает модели на то, что ее поле *artistid* (индекс которого находится в переменной *Cd_ArtistId)* содержит идентификатор *id* внешнего ключа из таблицы артистов *artist* и что вместо идентификаторов необходимо выводить на экран содержимое соответствующего поля *name*. Если пользователь переходит в режим редактирования этого поля (например, нажимая клавишу F2), модель автоматически выведет на экран поле с выпадающим списком имен всех артистов, и если пользователь выбирает другого артиста, таблица *cd* будет обновлена.

```
12 cdTableView = new QTableView;
13 cdTableView->setModel(cdModel);
14 cdTableView->setItemDelegate(new QSqlRelationalDelegate(this));
15 cdTableView->setSelectionMode(QAbstractItemView::SingleSelection);
16 cdTableView->setSelectionBehavior(QAbstractItemView::SelectRows);
17 cdTableView->setColumnHidden(Cd_Id, true);
18 cdTableView->resizeColumnsToContents();
```

Настройка представления таблицы *cd* выполняется аналогично тому, что мы уже делали. Единственным существенным отличием является применение *QSqlRelationalDelegate* вместо делегата по умолчанию. Именно этот делегат обеспечивает работу с внешними ключами.

```
19 trackModel = new QSqlTableModel(this);
20 trackModel->setTable("track");
21 trackModel->setHeaderData(Track_Title, Qt::Horizontal, tr("Title"));
22 trackModel->setHeaderData(Track_Duration, Qt::Horizontal,
23 tr("Duration"));
24 trackTableView = new QTableView;
25 trackTableView->setModel(trackModel);
26 trackTableView->setItemDelegate(
27 new TrackDelegate(Track_Duration, this));
```

```
28 trackTableView->setSelectionMode(QAbstractItemView::SingleSelection);
29 trackTableView->setSelectionBehavior(QAbstractItemView::SelectRows);
```

Для дорожек мы собираемся выводить на экран только названия песен и их длительности, поэтому достаточно использовать модель *QSqlTableModel*. (Поля *id* и *cdid*, используемые в рассмотренном ниже слоте *currentCdChanged()*, не выводятся на экран.) Единственно, на что следует обратить внимание в этой части программного кода, — это использование разработанного в главе 10 класса *TrackDelegate*, показывающего времена дорожек в виде «минуты:секунды» и позволяющего их редактировать с помощью удобного класса *QTimeEdit*.

Создание представлений и кнопок, их компоновка и соединения сигнал—слот не содержат ничего особенного, поэтому из оставшейся части конструктора мы покажем только несколько не совсем очевидных соединений.

```
30 …
31 connect(cdTableView->selectionModel(),
32 SIGNAL(currentRowChanged(const QModelIndex &,
33 const QModelIndex &)),
34 this, SLOT(currentCdChanged(const QModelIndex &)));
35 connect(cdModel, SIGNAL(beforeInsert(QSqlRecord &)),
36 this, SLOT(beforeInsertCd(QSqlRecord &)));
37 connect(trackModel, SIGNAL(beforeInsert(QSqlRecord &)),
38 this, SLOT(beforeInsertTrack(QSqlRecord &)));
39 connect(trackModel, SIGNAL(rowsInserted(
40 const QModelIndex &, int, int)),
41 this, SLOT(refreshTrackViewHeader()));
42 …
43 }
```

Первое соединение необычно, поскольку вместо связывания виджета мы связываем модель выборки. Класс *QItemSelectionModel* используется для отслеживания выборок в представлениях. Связанный с моделью выборки представления таблицы, наш слот *currentCdChanged()* будет вызываться при всяком перемещении пользователя от одной записи к другой.

```
01 void MainForm::currentCdChanged(const QModelIndex &index)
02 {
03 if (index.isValid()) {
04 QSqlRecord record = cdModel->record(index.row());
05 int id = record.value("id").toInt();
06 trackModel->setFilter(QString("cdid = %1").arg(id));
07 } else {
08 trackModel->setFilter("cdid = -1");
09 }
10 trackModel->select();
11 refreshTrackViewHeader();
12 }
```

Этот слот вызывается при каждой смене текущего компакт-диска. Это происходит при переходе пользователя к другому компакт-диску (щелкая мышкой по соответствующей

строке или используя клавиши Up и Down). Если компакт-диск недействителен (например, если вообще нет компакт-дисков или был вставлен новый компакт-диск, или текущий компакт-диск был только что удален), мы устанавливаем идентификатор *cdid* таблицы дорожек *track* в значение —1 (недействительный идентификатор, которому не соответствует никакая запись).

Затем, установив фильтр, мы выбираем ему соответствующие записи дорожек. Функция *refreshTrackViewHeader()* будет рассмотрена вскоре.

```
01 void MainForm::addCd()
02 {
03 int row = 0;
04 if (cdTableView->currentIndex().isValid())
05 row = cdTableView->currentIndex().row();
06 cdModel->insertRow(row);
07 cdModel->setData(cdModel->index(row, Cd_Year),
08 QDate::currentDate().year());
09 QModelIndex index = cdModel->index(row, Cd_Title);
10 cdTableView->setCurrentIndex(index);
11 cdTableView->edit(index);
12 }
```

Когда пользователь нажимает клавишу Add CD (добавить компакт-диск), в таблицу *cdTableView* вставляется новая пустая строка и мы переходим в режим редактирования. Мы также устанавливаем значение по умолчанию для поля *year*. В этот момент пользователь может редактировать запись, заполняя пустые поля и выбирая артиста из выпадающего списка, который автоматически выдается моделью *QSqlRelationalTableModel* благодаря вызову *setRelation()*, а также изменяя год, если не подходит значение по умолчанию. Если пользователь подтверждает вставку нажатием клавиши Enter, запись вставляется. Пользователь может отменить вставку, нажав клавишу Esc.

```
01 void MainForm::beforeInsertCd(QSqlRecord &record)
02 {
03 record.setValue("id", generateId("cd"));
04 }
```

Этот слот вызывается, когда *cdModel* генерирует свой сигнал *beforeInsert()*. Мы используем его для заполнения поля *id*, как это делалось при вставке нового артиста, и здесь применимо то же самое предостережение: данная операция должна выполняться в рамках транзакции, а в идеальном случае должно использоваться зависимое от базы данных средство создания идентификаторов (например, автоматическая генерация идентификаторов).

```
01 void MainForm::deleteCd()
02 {
03 QModelIndex index = cdTableView->currentIndex();
04 if (!index.isValid())
05 return;
06 QSqlDatabase db = QSqlDatabase::database();
07 db.transaction();
08 QSqlRecord record = cdModel->record(index.row());
```

```
09 int id = record.value(Cd_Id).toInt();
10 int tracks = 0;
11 QSqlQuery query;
12 query.exec(QString("SELECT COUNT(*) FROM track WHERE cdid = %1")
13 .arg(id));
14 if (query.next())
15 tracks = query.value(0).tolnt();
16 if (tracks > 0) {
17 int r = QMessageBox::question(this, tr("Delete CD"),
18 tr("Delete \"%1\" and all its tracks?")
19 .arg(record.value(Cd_ArtistId).toString()),
20 QMessageBox::Yes | QMessageBox::Default,
21 QMessageBox::No | QMessageBox::Escape);
22 if (r == QMessageBox::No) {
23 db.rollback();
24 return;
25 }
26 query.exec(QString("DELETE FROM track WHERE cdid = %1")
27 .arg(id));
28 }
29 cdModel->removeRow(index.row());
30 cdModel->submitAll();
31 db.commit();
32 currentCdChanged(QModelIndex());
33 }
```

Когда пользователь нажимает клавишу Delete CD (удалить компакт-диск), вызывается этот слот. Если имеется текущий компакт-диск, мы определяем, сколько у него дорожек. Если нет ни одной дорожки, мы просто удаляем запись компакт-диска. Если имеется по крайней мере одна дорожка, мы просим пользователя подтвердить удаление, и, если он нажимает кнопку Yes, мы удаляем все дорожки и затем запись самого компакт-диска. Все это делается в рамках транзакции, поэтому каскадное удаление либо совсем не будет выполнено, либо выполнится полностью при условии, что ваша база данных поддерживает транзакции.

Обработка данных дорожки очень похожа на обработку данных компакт-диска. Для обновления данных пользователь может просто редактировать ячейки. Что касается длительностей дорожек, то класс *TrackDelegate* гарантирует удобный формат отображения времен и они легко могут редактироваться с использованием *QTimeEdit*.

```
01 void MainForm::addTrack()
02 {
03 if (!cdTableView->currentIndex().isValid())
04 return;
05 int row = 0;
06 if (trackTableView->currentIndex().isValid())
07 row = trackTableView->currentIndex().row();
08 trackModel->insertRow(row);
```

```
09 QModelIndex index = trackModel->index(row, Track_Title);
10 trackTableView->setCurrentIndex(index);
11 trackTableView->edit(index);
12 }
```

Эта функция работает так же, как *addCd()*, со вставкой в представление новой пустой строки.

```
01 void MainForm::beforeInsertTrack(QSqlRecord &record)
02 {
03 QSqlRecord cdRecord = cdModel->record(cdTableView->currentIndex().row());
04 record.setValue("id", generateId("track"));
05 record.setValue("cdid", cdRecord.value(Cd_Id).toInt());
06 }
```

Если пользователь подтверждает вставку, инициированную функцией *addTrack()*, указанная выше функция вызывается для заполнения полей *id* и *cdid*. Упомянутые ранее предостережения применимы, конечно, и в этом случае.

```
01 void MainForm::deleteTrack()
02 {
03 trackModel->removeRow(trackTableView->currentIndex().row());
04 if (trackModel->rowCount() == 0)
05 trackTableView->horizontalHeader()->setVisible(false);
06 }
```

Если пользователь нажимает кнопку Delete Track (удалить дорожку), мы сразу же удаляем дорожку. Если предпочтительнее подтверждать удаление, мы могли бы легко выдать окно с сообщением и кнопками Yes и No.

```
01 void MainForm::refreshTrackViewHeader()
02 {
03 trackTableView->horizontalHeader()->setVisible(
04 trackModel->rowCount() > 0);
05 trackTableView->setColumnHidden(Track_Id, true);
06 trackTableView->setColumnHidden(Track_CdId, true);
07 trackTableView-> resizeColumnsToContents();
08 }
```

Слот *refreshTrackViewHeader()* вызывается из различных мест; он гарантирует вывод на экран горизонтального заголовка в представлении дорожек только в случае наличия дорожек. Он не показывает поля идентификаторов *id* и *cdid* и изменяет видимые размеры столбцов таблицы в зависимости от текущего содержимого таблицы.

```
01 void MainForm::editArtists()
02 {
03 QSqlRecord record = cdModel->record(cdTableView->currentIndex().row());
04 ArtistForm artistForm(record.value(Cd_ArtistId).toString(), this);
05 artistForm.exec();
06 cdModel->select();
07 }
```

Этот слот, вызывается при нажатии пользователем кнопки Edit Artists (правка артистов). Он обеспечивает вывод на экран данных о компакт-дисках текущего артиста, вызывая форму

ArtistForm, рассмотренную в предыдущем разделе, и делая выборку по соответствующему артисту. Если нет текущей записи, функция *record()* возвратит безвредную пустую запись, которая не будет соответствовать (и поэтому не будет выбрана) никакому артисту в форме артистов. В действительности при вызове *record.value(Cd_ArtistId)*, используемого из-за применения модели *QSqlRelationalTableModel*, которая идентификаторам артистов ставит в соответствие их имена, возвращается имя артиста (а оно будет пустой строкой, если запись пустая). В конце мы снова выбираем данные модели *cdModel*, что заставляет *cdTableView* обновить свои видимые ячейки. Это делается для того, чтобы гарантировать правильный вывод на экран имен артистов, поскольку некоторые из них пользователь мог изменить в диалоговом окне *ArtistForm.*

Для проектов, использующих SQL—классы, необходимо добавить строку

QT += sql

в файлы *.pro;* это обеспечит сборку приложения с библиотекой *QtSql*.

Данная глава показывает, что Qt—классы архитектуры модель/представление позволяют достаточно просто просматривать и редактировать данные, размещенные в базах данных SQL. В тех случаях, когда внешние ключи ссылаются на таблицы с большим количеством записей (например, тысячи записей и больше), по-видимому, лучше всего создать свой собственный делегат и использовать его для представления формы со «списком значений» и с возможностями поиска, а не полагаться на выпадающие списки модели *QSqlRelationalTableModel*. Кроме того, в ситуациях, когда требуется отображать записи в виджете формы, мы должны обеспечить это сами в своем программном коде — использовать *QSqlQuery* или *QSqlTableModel* для взаимодействия с базой данных и связать содержимое виджетов пользовательского интерфейса (который мы хотим использовать для представления и редактирования данных) с соответствующей базой данных.

Глава 14. Работа с сетью

Qt обеспечивает 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, в то время как UDP — ненадежный, не ориентированный на соединения протокол, 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.

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():*

```
01 int main(int argc, char *argv[])
02 {
03 QCoreApplication app(argc, argv);
04 QStringList args = app.arguments();
05 if (args.count() != 2) {
06 cerr << "Usage: ftpget url" << endl << "Example:" << endl
07 << " ftpget ftp://ftp.trolltech.com/mirrors" << endl;
08 return 1;
09 }
10 FtpGet getter;
11 if (!getter.getFile(QUrl(args[1])))
12 return 1;
13 QObject::connect(&getter, SIGNAL(done()), &app, SLOT(quit()));
14 return app.exec();
15 }
```

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 скачан.

Всю работу делает подкласс *FtpGet*, который определяется следующим образом:

```
01 class FtpGet : public QObject
02 {
03 Q_OBJECT
04 public:
05 FtpGet(QObject *parent = 0);
06 bool getFile(const QUrl &url);
07 signals:
08 void done();
09 private slots:
```

```
10 void ftpDone(bool error);
11 private:
12 QFtp ftp;
13 QFile file;
14 …
15 };
```

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 закрытые переменные. Переменная *ftp* имеет тип *QFtp* и инкапсулирует соединение с сервером FTP; переменная *file* используется для записи скачанного из сети файла на диск.

```
01 FtpGet::FtpGet(QObject *parent)
02 : QObject(parent)
03 {
04 connect(&ftp, SIGNAL(done(bool)), this, SLOT(ftpDone(bool)));
05 }
```

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.

```
01 bool FtpGet::getFile(const QUrl &url)
02 {
03 if (!url.isValid()) {
04 cerr << "Error: Invalid URL" << endl;
05 return false;
06 }
07 if (url.scheme() != "ftp") {
08 cerr << "Error: URL must start with 'ftp:'" << endl;
09 return false;
10 }
11 if (url.path().isEmpty()) {
12 cerr << "Error: URL has no path" << endl;
13 return false;
14 }
15 QString localFileName = QFileInfo(url.path()).fileName();
16 if (localFileName.isEmpty())
17 localFileName = "ftpget.out";
18 file.setFileName(localFileName);
19 if (!file.open(QIODevice::WriteOnly)) {
20 cerr << "Error: Cannot open "
21 << qPrintable(file.fileName()) << " for writing: "
22 << qPrintable(file.errorString()) << endl;
23 return false;
```

```
24 }
25 ftp.connectToHost(url.host(), url.port(21));
26 ftp.login();
27 ftp.get(url.path(), &file);
28 ftp.close();
29 return true;
30 }
```

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 удается 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 передаются ни имя пользователи, ни пароль, делается попытка анонимного входа в систему. Второй аргумент 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.

```
01 void FtpGet::ftpDone(bool error)
02 {
03 if (error) {
04 cerr << "Error: " << qPrintable(ftp.errorString()) << endl;
05 } else {
06 cerr << "File downloaded as " << qPrintable(file.fileName()) << endl;
07 }
08 file.close();
09 emit done();
10 }
```

После выполнения всех 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()*. 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:

```
01 bool FtpGet::getFile(const QUrl &url)
02 {
03 …
04 connectId = ftp.connectToHost(url.host(), url.port(21));
05 loginId = ftp.login();
06 getId = ftp.get(url.path(), &file);
07 closeId = ftp.close();
08 return true;
09 }

10 void FtpGet::ftpCommandStarted(int id)
11 {
12 if (id == connectId) {
13 cerr << "Connecting..." << endl;
14 } else if (id == loginId) {
15 cerr << "Logging in..." << endl;
16 …
17 }
```

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:*

```
01 class Spider : public QObject
```

```
02 {
03 Q_OBJECT
04 public:
05 Spider(QObject *parent = 0);
06 bool getDirectory(const QUrl &url);
07 signals:
08 void done();
09 private slots:
10 void ftpDone(bool error);
11 void ftpListInfo(const QUrlInfo &urlInfo);
12 private:
13 void processNextDirectory();
14 QFtp ftp;
15 QList<QFile *> openedFiles;
16 QString currentDir;
17 QString currentLocalDir;
18 QStringList pendingDirs;
19 };
```

Начальный каталог определяется как объект типа *QUrl* и устанавливается при помощи функции *getDirectory()*.

```
01 Spider::Spider(QObject *parent)
02 : QObject(parent)
03 {
04 connect(&ftp, SIGNAL(done(bool)), this, SLOT(ftpDone(bool)));
05 connect(&ftp, SIGNAL(listInfo(const QUrlInfo &)),
06 this, SLOT(ftpListInfo(const QUrlInfo &)));
07 }
```

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.

```
01 bool Spider::getDirectory(const QUrl &url)
02 {
03 if (!url.isValid()) {
04 cerr << "Error: Invalid URL" << endl;
05 return false;
06 }
07 if (url.scheme() != "ftp") {
08 cerr << "Error: URL must start with 'ftp:'" << endl;
09 return false;
10 }
11 ftp.connectToHost(url.host(), url.port(21));
12 ftp.login();
13 QString path = url.path();
14 if (path.isEmpty())
```

```
15 path = "/";
16 pendingDirs.append(path);
17 processNextDirectory();
18 return true;
19 }
```

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.

```
01 void Spider::processNextDirectory()
02 {
03 if (!pendingDirs.isEmpty()) {
04 currentDir = pendingDirs.takeFirst();
05 currentLocalDir = "downloads/" + currentDir;
06 QDir(".").mkpath(currentLocalDir);
07 ftp.cd(currentDir);
08 ftp.list();
09 } else {
10 emit done();
11 }
12 }
```

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.

```
01 void Spider::ftpListInfo(const QUrlInfo &urlInfo)
02 {
03 if (urlInfo.isFile()) {
04 if (urlInfo.isReadable()) {
05 QFile *file = new QFile(currentLocalDir + "/"
06 + urlInfo.name());
07 if (!file->open(QIODevice::WriteOnly)) {
08 cerr << "Warning: Cannot open file << qPrintable(
09 QDir::convertSeparators(file->fileName()))
10 << endl;
11 return;
12 }
13 ftp.get(urlInfo.name(), file);
14 openedFiles.append(file);
15 }
16 } else if (urlInfo.isDir() && !urlInfo.isSymLink()) {
```

```
17 pendingDirs.append(currentDir + "/" + urlInfo.name());
18 }
19 }
```

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.

```
01 void Spider::ftpDone(bool error)
02 {
03 if (error) {
04 cerr << "Error: " << qPrintable(ftp.errorString()) << endl;
05 } else {
06 cout << "Downloaded " << qPrintable(currentDir) << " to "
07 << qPrintable(QDir::convertSeparators(
08 QDir(currentLocalDir).canonicalPath()));
09 }
10 qDeleteAll(openedFiles);
11 openedFiles.clear();
12 processNextDirectory();
13 }
```

Слот *ftpDone()* вызывается после завершения всех команд FTP или при возникновении ошибки. Мы удаляем объекты *QFile* для предотвращения утечек памяти, а 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 такой:

```
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 скачаны все файлы из всех каталогов и список *pendingDirs* в результате не станет пустым.

Если возникнет сетевая ошибка при загрузке пятого файла, скажем, из двадцати файлов в каталоге, остальные файлы не будут скачаны. 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 так:

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.

```
01 int main(int argc, char *argv[])
02 {
03 QCoreApplication app(argc, argv);
04 QStringList args = app.arguments();
05 if (args.count() != 2) {
06 cerr << "Usage: spider url" << endl << "Example:" << endl
07 << " spider ftp://ftp.trolltech.com/freebies/leafnode" << endl;
08 return 1;
09 }
10 Spider spider;
11 if (!spider.getDirectory(QUrl(args[1])))
12 return 1;
13 QObject::connect(&spider, SIGNAL(done()), &app, SLOT(quit()));
14 return app.exec();
15 }
```

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.

```
01 HttpGet::HttpGet(QObject *parent)
02 : QObject(parent)
03 {
04 …
05 connect(&http, SIGNAL(done(bool)), this, SLOT(httpDone(bool)));
06 }
```

В конструкторе мы подсоединяем сигнал *done(bool)* объекта *QHttp* к закрытому слоту *httpDone(bool)*.

```
01 bool HttpGet::getFile(const QUrl &url)
02 {
03 if (!url.isValid()) {
04 cerr << "Error: Invalid URL" << endl;
05 return false;
06 }
07 if (url.scheme() != "http") {
08 cerr << "Error: URL must start with 'http:'" << endl;
09 return false;
10 }
11 if (url.path().isEmpty()) {
12 cerr << "Error: URL has no path" << endl;
13 return false;
14 }
15 QString localFileName = QFileInfo(url.path()).fileName();
16 if (localFileName.isEmpty())
17 localFileName = "httpget.out";
18 file.setFileName(localFileName);
19 if (!file.open(QIODevice::WriteOnly)) {
20 cerr << "Error: Cannot open "
21 << qPrintable(file.fileName()) << " for writing: "
```

```
22 << qPrintable(file.errorString()) << endl;
23 return false;
24 }
25 http.setHost(url.host(), url.port(80));
26 http.get(url.path(), &file);
27 http.close();
28 return true;
29 }
```

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.

```
01 void HttpGet::httpDone(bool error)
02 {
03 if (error) {
04 cerr << "Error: " << qPrintable(http.errorString()) << endl;
05 } else {
06 cerr << "File downloaded as " << qPrintable(file.fileName()) << endl;
07 }
08 file.close();
09 emit done();
10 }
```

После выполнения запросов HTTP мы файл закрываем, уведомляя пользователя о возникновении ошибки.

Функция *main()* очень похожа на такую же функцию в примере *ftpget:*

```
01 int main(int argc, char *argv[])
02 {
03 QCoreApplication app(argc, argv);
04 QStringList args = app.arguments();
05 if (args.count() != 2) {
06 cerr << "Usage: httpget url" << endl << "Example:" << endl
07 << " httpget http://doc.trolltech.com/qq/index.html" << endl;
08 return 1;
09 }
10 HttpGet getter;
11 if (!getter.getFile(QUrl(args[1])))
12 return 1;
13 QObject::connect(&getter, SIGNAL(done()), &app, SLOT(quit()));
14 return app.exec();
15 }
```

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* (который предоставляется компонентом Qt Solution компании «Trolltech») для работы с HTTP через SSL.

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 передавать открытое устройство *QIODevice*, например *QFile*. Для обеспечения большего контроля мы можем использовать функцию *request()*, которая принимает произвольные заголовок и данные HTTP. Например:

```
QHttpRequestHeader 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* наследует *QIODevice* через класс *QAbstractSocket*, и поэтому чтение с него или запись на него могут производиться с применением средств класса *QDataStream* и л и *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 называется Trip Planner (планировщик путешествий) и позволяет пользователям составлять план путешествия на поезде. Сервер называется Trip Server (сервер путешествий) и 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 расположены текстовая метка *QLabel*, показывающая состояние последней операции, и индикатор состояния процесса *QProgressBar*.

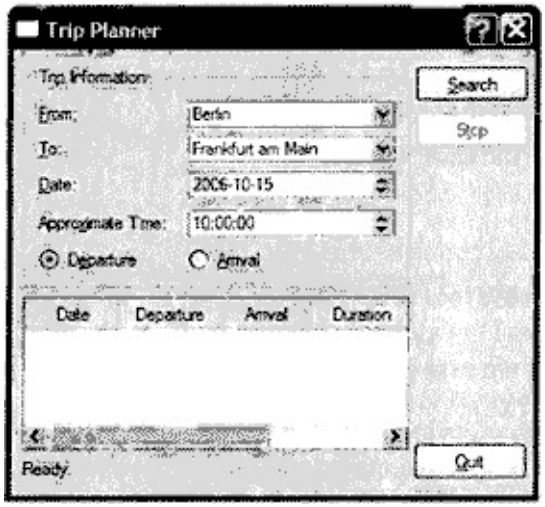


Рис. 14.1. ПриложениеTnp Planner.

Пользовательский интерфейс приложения Trip Planner был создан при помощи *QtDesigner* в файле *tripplanner.ui*. Ниже мы основное внимание уделим исходному коду подкласса *QDialog*, который реализует функциональность приложения:

```
#include "ui_tripplanner.h"
01 class TripPlanner : public QDialog, public Ui::TripPlanner
02 {
03 Q_OBJECT
04 public:
05 TripPlanner(QWidget *parent = 0);
06 private slots:
07 void connectToServer();
08 void sendRequest();
09 void updateTableWidget();
10 void stopSearch();
11 void connectionClosedByServer();
12 void error();
13 private:
14 void closeConnection();
15 QTcpSocket tcpSocket;
16 quint16 nextBlockSize;
17 };
```

Класс *TripPlanner* наследует не только *QDialog,* но и *Ui::TripPlanner* (который генерируется компилятором *uic*, используя файл *tripplanner.ui)*. Переменная—член *tcpSocket* инкапсулирует соединение TCP. Переменная *nextBlockSize* используется при синтаксическом анализе блоков, поступивших с сервера.

```
01 TripPlanner::TripPlanner(QWidget *parent)
02 : QDialog(parent)
03 {
04 setupUi(this);
05 QDateTime dateTime = QDateTime::currentDateTime();
06 dateEdit->setDate(dateTime.date());
07 timeEdit->setTime(QTime(dateTime.time().hour(), 0));
08 progressBar->hide();
```

```
09 progressBar->setSizePolicy(QSizePolicy::Preferred,
10 QSizePolicy::Ignored);
11 tableWidget->verticalHeader()->hide();
12 tableWidget->setEditTriggers(QAbstractItemView::NoEditTriggers);
13 connect(searchButton, SIGNAL(clicked()),
14 this, SLOT(connectToServer()));
15 connect(stopButton, SIGNAL(clicked()), this, SLOT(stopSearch()));
16 connect(&tcpSocket, SIGNAL(connected()),
17 this, SLOT(sendRequest()));
18 connect(&tcpSocket, SIGNAL(disconnected()),
19 this, SLOT(connectionClosedByServer()));
20 connect(&tcpSocket, SIGNAL(readyRead()),
21 this, SLOT(updateTableWidget()));
22 connect(&tcpSocket, SIGNAL(error(QAbstractSocket::SocketError)),
23 this, SLOT(error()));
24 }
```

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.

```
01 void TripPlanner::connectToServer()
02 {
03 tcpSocket.connectToHost("tripserver.zugbahn.de", 6178);
04 tableWidget->setRowCount(0);
05 searchButton->setEnabled(false);
06 stopButton->setEnabled(true);
07 statusLabel->setText(tr("Connecting to server..."));
08 progressBar->show();
09 nextBlockSize = 0;
10 }
```

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.

```
01 void TripPlanner::sendRequest()
02 {
03 QByteArray block;
04 QDataStream out(&block, QIODevice::WriteOnly);
05 out.setVersion(QDataStream::Qt_4_1);
06 out << quint16(0) << quint8('S') << fromComboBox->currentText()
07 << toComboBox->currentText() << dateEdit->date()
08 << timeEdit->time();
09 if (departureRadioButton->isChecked()) {
10 out << quint8('D');
11 } else {
12 out << quint8('A');
13 }
14 out.device()->seek(0);
15 out << quint16(block.size() - sizeof(quint16));
16 tcpSocket.write(block);
17 statusLabel->setText(tr("Sending request..."));
18 }
```

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 писать данные непосредственно в *QTcpSocket*, поскольку мы не знаем 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)* (то есть 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.

```
01 void TripPlanner::updateTableWidget()
02 {
03 QDataStream in(&tcpSocket);
```

```
04 in.setVersion(QDataStream::Qt_4_1);
05 forever {
06 int row = tableWidget->rowCount();

07 if (nextBlockSize == 0) {
08 if (tcpSocket.bytesAvailable() < sizeof(quint16))
09 break;
10 in >> nextBlockSize;
11 }
12 if (nextBlockSize == 0xFFFF) {
13 closeConnection();
14 statusLabel->setText(tr("Found %1 trip(s)").arg(row));
15 break;
16 }
17 if (tcpSocket.bytesAvailable() < nextBlockSize)
18 break;

19 QDate date;
20 QTime departureTime;
21 QTime arrivalTime;
22 quint16 duration;
23 quint8 changes;
24 QString trainType;

25 in >> date >> departureTime >> duration >> changes >> trainType;
26 arrivalTime = departureTime.addSecs(duration * 60);
27 tableWidget->setRowCount(row + 1);
28 QStringList fields;
29 fields << date.toString(Qt::LocalDate)
30 << departureTime.toString(tr("hh:mm"))
31 << arrivalTime.toString(tr("hh:mm"))
32 << tr("%1 hr %2 min").arg(duration / 60).arg(duration % 60)
33 << QString::number(changes) << trainType;

34 for (int i = 0; i < fields.count(); ++i)
35 tableWidget->setItem(row, i, new QTableWidgetItem(fields[i]));
36 nextBlockSize = 0;
37 }
38 }
```

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. Цикл *forever* необходим, потому что мы не обязательно получаем от сервера блоки по одному[7]. Мы можем получить целый блок или только его часть или полтора блока либо даже все блоки сразу.

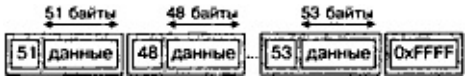


Рис. 14.2. Блоки приложения Trip Server.

Итак, как действует цикл *forever?* Если переменная *nextBlockSize* равна 0, это означает, что мы не прочитали размер следующего блока. Мы пытаемся прочитать 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 неизвестен и его необходимо считать.

```
01 void TripPlanner::closeConnection()
02 {
03 tcpSocket.close();
04 searchButton->setEnabled(true);
05 stopButton->setEnabled(false);
06 progressBar->hide();
07 }
```

Закрытая функция *closeConnection()* закрывает соединение сервера TCP и обновляет интерфейс пользователя. Она вызывается из функции *updateTableWidget()*, когда считывается значение 0xFFFF, и из нескольких других слотов, которые мы вскоре рассмотрим.

```
01 void TripPlanner::stopSearch()
02 {
03 statusLabel->setText(tr("Search stopped"));
04 closeConnection();
05 }
```

Слот *stopSearch()* подсоединяется к сигналу *clicked()* кнопки Stop. По существу, он просто вызывает функцию *closeConnection()*.

```
01 void TripPlanner::connectionClosedByServer()
02 {
03 if (nextBlockSize != 0xFFFF)
04 statusLabel->setText(tr("Error: Connection closed by server" ));
```

```
05 closeConnection();
06 }
```

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.

```
01 void TripPlanner::error()
02 {
03 statusLabel->setText(tcpSocket.errorString());
04 closeConnection();
05 }
```

Слот *error()* подсоединяется к сигналу *error(QAbstractSocket::SocketError)* объекта *QTcpSocket*. Мы игнорируем код ошибки и используем функцию *QTcpSocket::errorString()*, которая возвращает понятное человеку сообщение о последней возникшей ошибке.

На этом завершается рассмотрение класса *TripPlanner*. Функция *main()* приложения Trip Planner выглядит обычным образом:

```
01 int main(int argc, char *argv[])
02 {
03 QApplication app(argc, argv);
04 TripPlanner tripPlanner;
05 tripPlanner.show();
06 return app.exec();
07 }
```

Теперь давайте реализуем сервер. Сервер состоит из двух классов: *TripServer* и *ClientSocket*. Класс *TripServer* наследует *QTcpServer* — класс, который позволяет нам принимать входящие соединения TCP. Класс *ClientSocket* переопределяет *QTcpSocket* и обслуживает одно соединение. В каждый момент времени в памяти имеется ровно столько объектов типа *ClientSocket*, сколько обслуживается клиентов.

```
01 class TripServer : public QTcpServer
02 {
03 Q_OBJECT
04 public:
05 TripServer(QObject *parent = 0);
06 private:
07 void incomingConnection(int socketId);
08 };
```

Класс *TripServer* переопределяет функцию *incomingConnection()* из класса *QTcpServer*. Данная функция вызывается всякий раз, когда клиент пытается подсоединиться к порту, который прослушивает сервер.

```
01 TripServer::TripServer(QObject *parent)
02 : QTcpServer (parent)
03 {
04 }
```

Конструктор *TripServer* тривиален.

```
01 void TripServer::incomingConnection(int socketId)
```

```
02 {
03 ClientSocket *socket = new ClientSocket(this);
04 socket->setSocketDescriptor(socketId);
05 }
```

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 а*втоматически удалит сам себя при прекращении соединения.

```
01 class ClientSocket : public QTcpSocket
02 {
03 Q_OBJECT
04 public:
05 ClientSocket(QObject *parent = 0);
06 private slots:
07 void readClient();
08 private:
09 void generateRandomTrip(const QString &from, const QString &to,
10 const QDate &date, const QTime &time);
11 quint16 nextBlockSize;
12 };
```

Класс *ClientSocket* наследует *QTcpSocket* и инкапсулирует состояние одного клиента.

```
01 ClientSocket::ClientSocket(QObject *parent)
02 : QTcpSocket(parent)
03 {
04 connect(this, SIGNAL(readyRead()), this, SLOT(readClient()));
05 connect(this, SIGNAL(disconnected()), this, SLOT(deleteLater()));
06 nextBlockSize = 0;
07 }
```

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.

```
01 void ClientSocket::readClient()
02 {
03 QDataStream in(this);
04 in.setVersion(QDataStream::Qt_4_1);
05 if (nextBlockSize == 0) {
06 if (bytesAvailable() < sizeof(quint16))
07 return;
08 in >> nextBlockSize;
09 }
10 if (bytesAvailable() < nextBlockSize)
11 return;
```

```
12 quint8 requestType;
13 QString from;
14 QString to;
15 QDate date;
16 QTime time;
17 quint8 flag;
18 in >> requestType;
19 if (requestType == 'S') {
20 in >> from >> to >> date >> time >> flag;
21 srand(from.length() * 3600 + to.length() * 60 + time.hour());
22 int numTrips = rand() % 8;
23 for (int i = 0; i < numTrips; ++i)
24 generateRandomTrip(from, to, date, time);
25 QDataStream out(this);
26 out << quint16(0xFFFF);
27 }
28 close();
29 }
```

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 осуществляли бы поиск информации в базе данных расписания железнодорожных рейсов и попытались бы 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.

```
01 void ClientSocket::generateRandomTrip(const QString & /* откуда */,
02 const QString & /* куда */, const QDate &date, const QTime &time)
03 {
04 QByteArray block;
05 QDataStream out(&block, QIODevice::WriteOnly);
06 out.setVersion(QDataStream::Qt_4_1);
07 quint16 duration = rand() % 200;
08 out << quint16(0) << date << time << duration << quint8(1)
09 << QString("InterCity");
10 out.device()->seek(0);
11 out << quint16(block.size() - sizeof(quint16));
12 write(block);
13 }
```

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()*.

```
01 int main(int argc, char *argv[])
02 {
03 QApplication app(argc, argv);
04 TripServer server;
05 if (!server.listen(QHostAddress::Any, 6178)) {
06 cerr << "Failed to bind to port" << endl;
07 return 1;
08 }
09 QPushButton quitButton(QObject::tr("&Quit"));
10 quitButton.setWindowTitle(QObject::tr("Trip Server"));
11 QObject::connect(&quitButton, SIGNAL(clicked()),
12 &app, SLOT(quit()));
13 quitButton.show();
14 return app.exec();
15 }
```

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.

Этим завершается наш пример системы клиент—сервер. В данном случае нами использовался блокоориентированный 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.

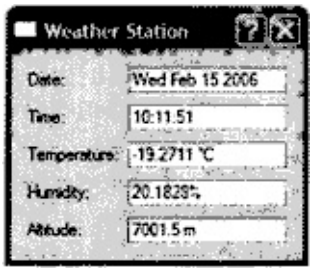


Рис. 14.3. Приложение Weather Station.

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 2 секунды дейтаграммы UDP с параметрами текущего атмосферного состояния. Приложение Weather Station получает эти дейтаграммы и выводит их на экран. Мы начнем с рассмотрения программного кода приложения Weather Balloon.

```
01 class WeatherBalloon : public QPushButton
02 {
03 Q_OBJECT
04 public:
05 WeatherBalloon(QWidget *parent = 0);
06 double temperature() const;
07 double humidity() const;
08 double altitude() const;
09 private slots:
10 void sendDatagram();
11 private:
12 QUdpSocket udpSocket;
13 QTimer timer;
14 };
```

Класс *WeatherBalloon* наследует *QPushButton*. Он использует свою закрытую переменную типа *QUdpSocket* для обеспечения связи с приложением Weather Station.

```
01 WeatherBalloon::WeatherBalloon(QWidget *parent)
02 : QPushButton(tr("Quit"), parent)
03 {
03 connect(this, SIGNAL(clicked()), this, SLOT(close()));
04 connect(&timer, SIGNAL(timeout()), this, SLOT(sendDatagram()));
05 timer.start(2 * 1000);
```

```
06 setWindowTitle(tr("Weather Balloon"));
07 }
```

В конструкторе мы запускаем *QTimer* для вызова *sendDatagram()* через каждые 2 секунды.

```
01 void WeatherBalloon::sendDatagram()
02 {
03 QByteArray datagram;
04 QDataStream out(&datagram, QIODevice::WriteOnly);
05 out.setVersion(QDataStream::Qt_4_1);
06 out << QDateTime::currentDateTime() << temperature()
07 << humidity() << altitude();
08 udpSocket.writeDatagram(datagram, QHostAddress::LocalHost, 5824);
09 }
```

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() (в коде "writeDatagram". wtf?).* 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 *QAbstractSocket*, класс *QUdpSocket* 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.

```
01 int main(int argc, char *argv[])
02 {
03 QApplication app(argc, argv);
04 WeatherBalloon balloon;
05 balloon.show();
06 return app.exec();
07 }
```

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.

```
01 class WeatherStation : public QDialog
```

```
02 {
03 Q_OBJECT
04 public:
05 WeatherStation(QWidget *parent = 0);
06 private slots:
07 void processPendingDatagrams();
08 private:
09 QUdpSocket udpSocket;
10 QLabel *dateLabel;
11 QLabel *timeLabel;
12 QLineEdit *altitudeLineEdit;
13 };
```

Класс *WeatherStation* наследует *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.

```
01 WeatherStation::WeatherStation(QWidget *parent)
02 : QDialog(parent)
03 {
04 udpSocket.bind(5824);
05 connect(&udpSocket, SIGNAL(readyRead()),
06 this, SLOT(processPendingDatagrams()));
07 }
```

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.

```
01 void WeatherStation::processPendingDatagrams()
02 {
03 QByteArray datagram;
04 do {
05 datagram.resize(udpSocket.pendingDatagramSize());
06 udpSocket.readDatagram(datagram.data(), datagram.size());
07 } while (udpSocket.hasPendingDatagrams());
08 QDateTime dateTime;
09 double temperature;
10 double humidity;
11 double altitude;
12 QDataStream in(&datagram, QIODevice::ReadOnly);
13 in.setVersion(QDataStream::Qt_4_1);
14 in >> dateTime >> temperature >> humidity >> altitude;
15 dateLineEdit->setText(dateTime.date().toString());
```

```
16 timeLineEdit->setText(dateTime.time().toString());
17 temperatureLineEdit->setText(tr("%1° C").arg(temperature));
18 humidityLineEdit->setText(tr("%1%").arg(humidity));
19 altitudeLineEdit->setText(tr("%1 m").arg(altitude));
20 }
```

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.

```
01 int main(int argc, char *argv[])
02 {
03 QApplication app(argc, argv);
04 WeatherStation station;
05 station.show();
06 return app.exec();
07 }
```

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 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.

Глава 15. XML

XML (Extensible Markup Language — расширяемый язык разметки) — это универсальный формат текстовых файлов, который получил широкое распространение при обмене и хранении данных. Qt обеспечивает два различных программных интерфейса для чтения документов XML; эти интерфейсы входят в состав модуля *QtXml:*

• SAX (Simple API for XML — простой программный интерфейс для документов XML) позволяет обрабатывать «события синтаксического анализа» непосредственно в приложении в соответствующих виртуальных функциях.

• DOM (Document Object Model — объектная модель документа) преобразует документ XML в структуру в виде дерева, которая затем может обрабатываться в приложении.

Существует много факторов, которые необходимо учитывать в каждом конкретном случае при выборе между DOM и SAX. SAX является интерфейсом более низкого уровня и обычно работает быстрее, что делает его особенно пригодным как для решения простых задач (например, для поиска в документе XML всех повторений заданного тега), так и для чтения очень больших файлов, которые не помещаются в оперативной памяти. Но для большинства приложений удобство применения DOM перевешивает потенциальные преимущества более высокого быстродействия и более эффективного использования памяти в SAX.

Создавать файлы XML можно двумя способами: мы можем сгенерировать XML вручную или представить данные в виде дерева DOM, размещенного в памяти, и «попросить» это дерево записать себя в файл.

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. 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—обработчиках, уведомляющих о возникновении соответствующих событий в ходе синтаксического анализа документа. (Эти события никак не связаны с такими событиями Qt, как события клавиатуры и события мышки.) Например, пусть парсер выполняет анализ следующего документа XML:

<doc>
<quote> Ars longa vita brevis</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 наследует все классы—обработчики 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.

Теперь мы рассмотрим пример, который показывает способы применения *QXmlSimpleReader* и *QXmlDefaultHandler* для синтаксического анализа файла XML заранее известного формата и для отображения его содержимого в виджете *QTreeWidget*. Подкласс *QXmlDefaultHandler* имеет название *SaxHandler*, и он используется для обработки предметного указателя книги, который содержит элементы и подэлементы.

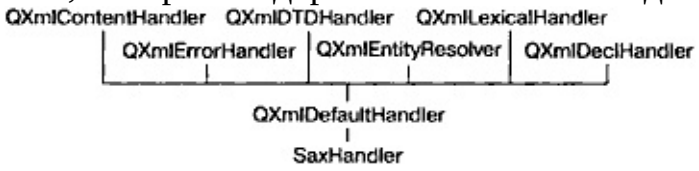


Рис. 15.1. Дерево наследования для SaxHandler.

Ниже приводится файл предметного указателя книги, который отображается в виджете *QTreeWidget* и показан на рис. 15.2:

```
<?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>
```

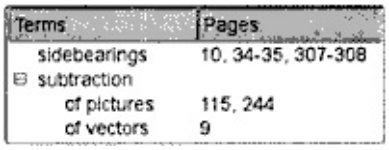


Рис. 15.2. Файл предметного указателя книги, загруженный в виджет QTreeWidget.

Первый этап в реализации парсера заключается в создании подкласса *QXmlDefaultHandler:*

```
01 class SaxHandler : public QXmlDefaultHandler
02 {
03 public:
04 SaxHandler(QTreeWidget *tree);
05 bool startElement(const QString &namespaceURI,
06 const QString &localName,
07 const QString &qName,
08 const QXmlAttributes &attributes);
09 bool endElement(const QString &namespaceURI,
10 const QString &localName,
```

```
11 const QString &qName);
12 bool characters(const QString &str);
13 bool fatalError(const QXmlParseException &exception);
14 private:
15 QTreeWidget *treeWidget;
16 QTreeWidgetItem *currentItem;
17 QString currentText;
18 };
```

Класс *SaxHandler* наследует *QXmlDefaultHandler* и переопределяет четыре функции: *startElement(), endElement(), characters()* и *fatalError()*. Первые четыре функции объявлены в *QXmlContentHandler;* последняя функция объявлена в *QXmlErrorHandler*.

```
01 SaxHandler::SaxHandler(QTreeWidget *tree)
02 {
03 treeWidget = tree;
04 currentItem = 0;
05 }
```

Конструктор *SaxHandler* принимает объект типа *QTreeWidget,* который мы собираемся заполнять информацией, содержащейся в файле XML.

```
01 bool SaxHandler::startElement(const QString & /* namespaceURI */,
02 const QString & /* localName */,
03 const QString &qName,
04 const QXmlAttributes &attributes)
05 {
06 if (qName == "entry") {
07 if (currentItem) {
08 currentItem = new QTreeWidgetItem(currentItem);
09 } else {
10 currentItem = new QTreeWidgetItem(treeWidget);
11 }
12 currentItem->setText(0, attributes.value("term"));
13 } else if (qName == "page") {
14 currentText.clear();
15 }
16 return true;
17 }
```

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, подробно описанный в справочной документации.

Если обнаружен тег *<entry>,* мы создаем новый элемент списка *QTreeWidget*. 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.

```
01 bool SaxHandler::characters(const QString &str)
02 {
03 currentText += str;
04 return true;
05 }
```

Функция *characters()* используется для извлечения символьных данных из документа XML. Мы просто добавляем символы в конец переменной *currentText*.

```
01 bool SaxHandler::endElement(const QString & /* namespaceURI */,
02 const QString & /* localName */, const QString &qName)
03 {
04 if (qName == "entry") {
05 currentItem = currentItem->parent();
06 } else if (qName == "page") {
07 if (currentItem) {
08 QString allPages = currentItem->text(1);
09 if (!allPages.isEmpty())
10 allPages += ", ";
11 allPages += currentText;
12 currentItem->setText(1, allPages);
13 }
14 }
15 return true;
16 }
```

Функция *endElement()* вызывается при обнаружении закрывающего тега. Так же 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*. 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.

```
01 bool SaxHandler::fatalError(const QXmlParseException &exception)
02 {
03 QMessageBox::warning(0, QObject::tr("SAX Handler"),
04 QObject::tr("Parse error at line %1, column %2:\n%3.")
```

```
05 .arg(exception.lineNumber())
06 .arg(exception.columnNumber())
07 .arg(exception.message()));
08 return false;
09 }
```

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*. Теперь давайте посмотрим, как можно использовать этот класс:

```
01 bool parseFile(const QString &fileName)
02 {
03 QStringList labels;
04 labels << QObject::tr("Terms") << QObject::tr("Pages");
05 QTreeWidget *treeWidget = new QTreeWidget;
06 treeWidget->setHeaderLabels(labels);
07 treeWidget->setWindowTitle(QObject::tr("SAX Handler"));
08 treeWidget->show();

09 QFile file(fileName);
10 QXmlInputSource inputSource(&file);
11 QXmlSimpleReader reader;
12 SaxHandler handler(treeWidget);
13 reader.setContentHandler(&handler);
14 reader.setErrorHandler(&handler);
15 return reader.parse(inputSource);
16 }
```

Мы задаем два столбца в виджете *QTreeWidget*. Затем мы создаем объект типа *QFile* для считываемого файла и объект типа *QXmlSimpleReader* для синтаксического анализа файла. Нам не требуется самим открывать *QFile; QXmlInputSource* делает это автоматически.

Наконец, мы создаем объект типа *SaxHandler*, который используется для объекта *reader* одновременно в качестве обработчика содержимого файла и обработчика ошибок, и мы вызываем функцию *parse()* для выполнения синтаксического анализа.

Вместо простого объекта файла мы передаем функции *parse()* объект *QXmlInputSource*. Этот класс открывает заданный файл, читает его (учитывая кодировку символов в объявлении *<?xml?>)* и предоставляет интерфейс для чтения файла парсером.

В классе *SaxHandler* мы всего лишь переопределили функции, унаследованные от классов *QXmlContentHandler* и *QXmlErrorHandler*. Если бы мы стали переопределять функции других классов—обработчиков, нам пришлось бы вызывать соответствующие функции—установщики для объекта reader.

Для сборки приложения с библиотекой *QtXml* в файл *.pro* необходимо добавить следующую строку:

```
QT += xml
```

Чтение документов XML при помощи интерфейса DOM

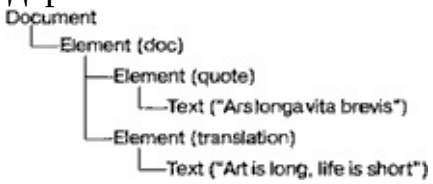
DOM является стандартным программным интерфейсом синтаксического анализа документов XML, который разработан Консорциумом всемирной паутины (W3C). Qt обеспечивает уровень 2 интерфейса DOM для чтения, обработки и записи документов XML без проверки их достоверности.

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>Ars longa vita brevis</quote>
<translation>Art is long, life is short</translation>
</doc>
```

Ему соответствует следующее дерево DOM:



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*. Рис. 15.3 показывает, какие типы дочерних узлов допустимы для соответствующих родительских узлов. Узлы, показанные серым, не могут иметь дочерних узлов.

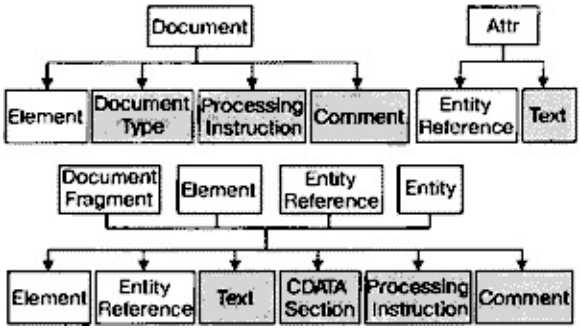


Рис. 15.3. Родственные связи между узлами DOM.

Для иллюстрации применения DOM при чтении файлов XML мы напишем парсер для файла предметного указателя книги, описанного в предыдущем разделе.

```
01 class DomParser
02 {
```

```
03 public:
04 DomParser(QIODevice *device, QTreeWidget *tree);
05 private:
06 void parseEntry(const QDomElement &element,
07 QTreeWidgetItem *parent);
08 QTreeWidget *treeWidget;
09 };
```

Мы определяем класс с названием *DomParser*, который выполняет синтаксический анализ предметного указателя книги, представленного в виде документа XML, и отображает результат в виджете *QTreeWidget*. Этот класс не наследует никакой другой класс.

```
01 DomParser::DomParser(QIODevice *device, QTreeWidget *tree)
02 {
03 treeWidget = tree;
04 QString errorStr;
05 int errorLine;
06 int errorColumn;
07 QDomDocument doc;
08 if (!doc.setContent(device, true, &errorStr,
09 &errorLine, &errorColumn)) {
10 QMessageBox::warning(0, QObject::tr("DOM Parser"),
11 QObject::tr("Parse error at line %1, column %2:\n%3")
12 .arg(errorLine).arg(errorColumn).arg(errorStr));
13 return;
14 }
15 QDomElement root = doc.documentElement();
16 if (root.tagName() != "bookindex")
17 return;
18 QDomNode node = root.firstChild();
19 while (!node.isNull()) {
20 if (node.toElement().tagName() == "entry")
21 parseEntry(node.toElement(), 0);
22 node = node.nextSibling();
23 }
24 }
```

В конструкторе мы создаем объект *QDomDocument* и вызываем для него функцию *setContent()*, чтобы с его помощью прочесть документ XML с устройства *QIODevice*. Функция *setContent()* автоматически открывает устройство, если оно еще не открыто. Затем мы вызываем функцию *documentElement()* для объекта *QDomDocument*, чтобы получить его одиночный дочерний элемент *QDomElement*, после чего мы проверяем, является ли данный элемент <*bookindex*>. Мы выполняем цикл по всем дочерним узлам, и если узлом является элемент <*entry*>, мы вызываем функцию *parseEntry()* для его синтаксического анализа.

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.

```
01 void DomParser::parseEntry(const QDomElement &element,
02 QTreeWidgetItem *parent)
03 {
04 QTreeWidgetItem *item;
05 if (parent) {
06 item = new QTreeWidgetTtem(parent);
07 } else {
08 item = new QTreeWidgetItem(treeWidget);
09 }
10 item->setText(0, element.attribute("term"));
11 QDomNode node = element.firstChild();
12 while (!node.isNull()) {
13 if (node.toElement().tagName() == "entry") {
14 parseEntry(node.toElement(), item);
15 } else if (node.toElement().tagName() == "page") {
16 QDomNode childNode = node.firstChild();
17 while (!childNode.isNull()) {
18 if (childNode.nodeType() == QDomNode::TextNode) {
19 QString page = childNode.toText().data();
20 QString allPages = item->text(1);
21 if (!allPages.isEmpty())
22 allPages += ", ";
23 allPages += page;
24 item->setText(1, allPages);
25 break;
26 }
27 childNode = childNode.nextSibling();
28 }
29 }
30 node = node.nextSibling();
31 }
32 }
```

В функции *parseEntry()* мы создаем элемент объекта *QTreeWidget*. Если тег вложен в другой *<entry>*, новый тег определяет подэлемент предметного указателя, и мы создаем элемент *QTreeWidgetItem* как дочерний для внешнего элемента *QTreeWidgetItem*. В противном случае мы создаем элемент *QTreeWidgetItem* с *treeWidget* в качестве его родительского элемента, делая его элементом верхнего уровня. Мы вызываем функцию *setText()* для установки текста столбца 0 на значение атрибута *term* тега *<entry>*.

После инициализации нами элемента *QTreeWidgetItem* мы выполняем цикл по дочерним узлам элемента *QDomElement*, который соответствует текущему тегу *<entry>*.

Если элементом является *<entry>*, мы вызываем функцию *parseEntry()*, передавая текущий элемент в качестве второго аргумента. Затем будет создан новый элемент

QTreeWidgetItem, в качестве родительского элемента которого выступает внешний элемент *QTreeWidgetItem*.

Если элементом является *<page>*, мы просматриваем список дочерних элементов для поиска узла *Text*. После его обнаружения мы вызываем функцию *toText()* для преобразования его в объект типа *QDomText* и функцию *data()* для получения текста в виде строки типа *QString*. Затем мы добавляем текст в разделяемый запятыми список номеров страниц в столбце 1 элемента *QTreeWidgetItem*.

Давайте теперь посмотрим, как мы можем использовать класс *DomParser* для синтаксического анализа файла:

```
01 void parseFile(const QString &fileName)
02 {
03 QStringList labels;
04 labels << QObject::tr("Terms") << QObject::tr("Pages");
05 QTreeWidget *treeWidget = new QTreeWidget;
06 treeWidget->setHeaderLabels(labels);
07 treeWidget->setWindowTitle(QObject::tr("DOM Parser"));
08 treeWidget->show();
09 QFile file(fileName);
10 DomParser(&file, treeWidget);
11 }
```

Мы начинаем с настройки *QTreeWidget*. Затем мы создаем объекты *QFile* и *DomParser*. При выполнении конструктора *DomParser* осуществляется синтаксический анализ файла и пополняется виджет 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, проход по дереву DOM может быть достаточно непростым делом. Простая операция по извлечению текста между тегами *<page>* и *</page>* требует обработки в цикле элементов списка при помощи функций *firstChild()* и *nextSibling()* класса *QDomNode*. Программисты, которым очень часто приходится использовать интерфейс DOM, создают свои собственные высокоуровневые функции—оболочки для упрощения выполнения таких наиболее распространенных операций, как извлечение текста между открывающими и закрывающими тегами.

Запись документов XML

Существует два основных подхода к формированию файлов XML в приложениях Qt:

• 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: SAX или DOM.

Ниже приводится фрагмент программного кода, который иллюстрирует способ создания дерева DOM и его записи при помощи *QTextStream:*

```
const int Indent = 4;

QDomDocument doc;
QDomElement root = doc.createElement("doc");
QDomElement quote = doc.createElement("quote");
QDomElement translation = doc.createElement("translation");
QDomText latin = doc.createTextNode("Ars longa vita brevis");
QDomText english = doc.createTextNode("Art is long, life is short");

doc.appendChild(root);
root.appendChild(quote);
root.appendChild(translation);
quote.appendChild(latin);
translation.appendChild(english);

QTextStream out(&file);
doc.save(out, Indent);
```

Второй аргумент функции *save()* задает размер отступа. При ненулевом его значении читаемость сформированного файла будет лучше. Ниже приводится полученный на выходе файл XML:

```
<doc>
....<quote>Ars longa vita brevis</quote>
....<translation>Art is long, life is short</translation>
</doc>
```

Порядок действий будет другим, если в приложении дерево DOM используется в качестве главной структуры данных. В таких случаях приложения обычно считывают документы XML, применяя интерфейс DOM, затем модифицируют в памяти дерево DOM и, наконец, вызывают функцию *save()* для обратного преобразования дерева в документ XML.

По умолчанию функция *QDomDocument::save()* использует для генерации файла кодировку *UTF-8*. Мы можем применить другую кодировку, если добавить XML—объявление, например такое, как

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

в начало дерева DOM. Следующий фрагмент программного кода показывает, как это делать:

```
QTextStream out(&file);
QDomNode xmlNode = doc.createProcessingInstruction("xml",
```

```
"version=\"1.0\" encoding=\"ISO-8859-1\"");
doc.insertBefore(xmlNode, doc.firstChild());
doc.save(out, Indent);
```

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";
```

В статье «Generating XML» (Формирование документов XML) в журнале *«Qt Quarterly»*, доступной в сети Интернет по адресу http://doc.trolltech.com/qq/qq05-generating-xml.html, рассматривается очень простой класс, позволяющий легко формировать файлы XML. Этот класс решает вопросы, связанные со специальными символами, отступами и кодировкой, позволяя нам полностью сконцентрироваться на документе XML, который мы собираемся формировать. Он предназначен для работы с Qt 3, но его очень легко перенести на Qt 4.

Глава 16. 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 мышки 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 мышки 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"));

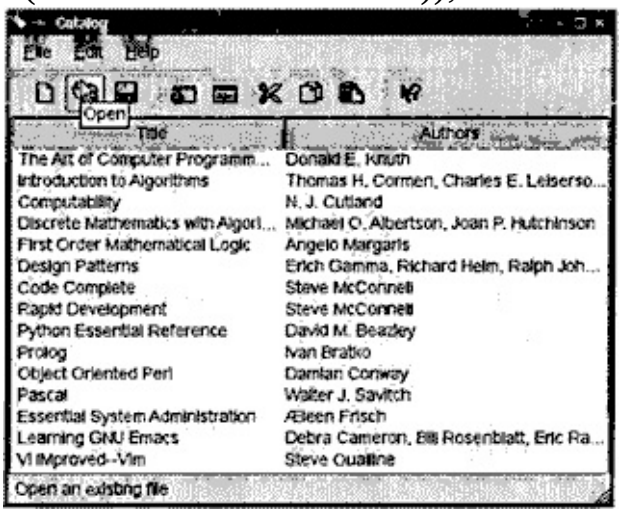


Рис. 16.1. В этом приложении отображаются всплывающая подсказка и комментарий в строке состояния.

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), либо 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\">"
" 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 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/4.1/richtext-html-subset.html.

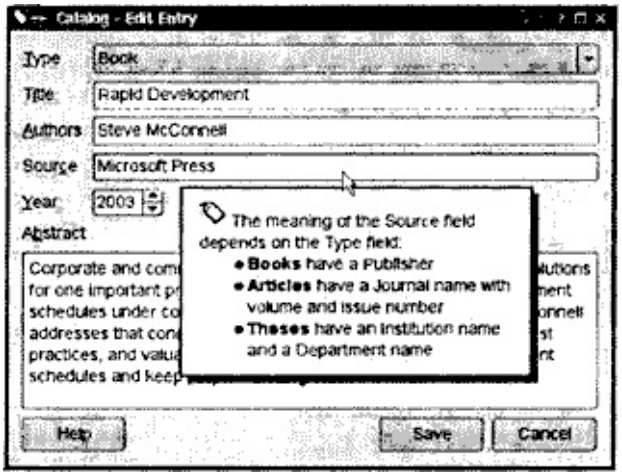


Рис. 16.2. Диалоговое окно с отображением текста справки «что это такое?»

Кроме того, мы можем задавать текст справки «что это такое?» для команды:

```
openAct->setWhatsThis(tr("<img src=open.png> "
"Click this option to open an existing file."));
```

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 *QTextBrowser* в качестве простого браузера системы помощи

Для больших приложений может потребоваться более сложная система помощи в отличие от той, которую обычно обеспечивают всплывающие подсказки, комментарии в строке состояния и справки «что это такое?». Простое решение состоит в применении браузера системы помощи. Приложения, которые включают в себя браузер системы помощи, обычно имеют подпункт меню Help в меню Help главного окна и кнопку Help в каждом диалоговом окне.

В данном разделе мы представим простой браузер системы помощи, показанный на рис. 16.3, 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.

Мы начинаем с заголовочного файла:

```
01 #include <QWidget>
02 class QPushButton;
03 class QTextBrowser;
04 class HelpBrowser : public QWidget
05 {
06 Q_OBJECT
07 public:
08 HelpBrowser(const QString &path,
09 const QString &page, QWidget *parent = 0);
10 static void showPage(const QString &page);
11 private slots:
12 void updateWindowTitle();
13 private:
14 QTextBrowser *textBrowser;
15 QPushButton *homeButton;
16 QPushButton *backButton;
17 QPushButton *closeButton;
18 };
```

Класс *HelpBrowser* содержит статическую функцию, которую можно вызывать в любом месте в приложении. Данная функция создает окно *HelpBrowser* и выводит на экран заданную страницу.

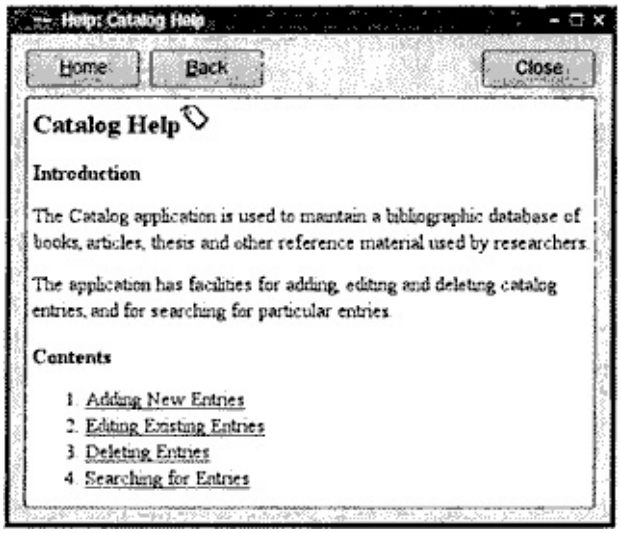


Рис. 16.3. Виджет HelpBrowser.

Ниже приводится начало реализации:

```
01 #include <QtGui>
02 #include "helpbrowser.h"
03 HelpBrowser::HelpBrowser(const QString &path,
04 const QString &page, QWidget *parent)
05 : QWidget(parent)
06 {
07 setAttribute(Qt::WA_DeleteOnClose);
08 setAttribute(Qt::WA_GroupLeader);
09 textBrowser = new QTextBrowser;
10 homeButton = new QPushButton(tr("&Home"));
11 backButton = new QPushButton(tr("&Back"));
12 closeButton = new QPushButton(tr("Close"));
13 closeButton->setShortcut(tr("Esc"));
14 QHBoxLayout *buttonLayout = new QHBoxLayout;
15 buttonLayout->addWidget(homeButton);
16 buttonLayout->addWidget(backButton);
17 buttonLayout->addStretch();
18 buttonLayout->addWidget(closeButton);
19 QVBoxLayout *mainLayout = new QVBoxLayout;
20 mainLayout->addLayout(buttonLayout);
21 mainLayout->addWidget(textBrowser);
22 setLayout(mainLayout);
23 connect(homeButton, SIGNAL(clicked()),
24 textBrowser, SLOT(home()));
25 connect(backButton, SIGNAL(clicked()),
26 textBrowser, SLOT(backward()));
27 connect(closeButton, SIGNAL(clicked()),
28 this, SLOT(close()));
29 connect(textBrowser, SIGNAL(sourceChanged(const QUrl &)),
30 this, SLOT(updateWindowTitle()));
31 textBrowser->setSearchPaths(QStringList() << path << ":/images");
```

```
32 textBrowser->setSource(page);
33 }
```

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 которых начинаются с символов *: /* (двоеточие и слеш). Параметр *page* содержит имя файла документации с возможным указанием метки HTML (anchor).

```
01 void HelpBrowser::updateWindowTitle()
02 {
03 setWindowTitle(tr("Help: %1")
04 .arg(textBrowser->documentTitle()));
05 }
```

При всяком изменении исходной страницы вызывается слот *updateWindowTitle()*. Функция *documentTitle()* возвращает текст, содержащийся в теге *<title>* этой страницы.

```
01 void HelpBrowser::showPage(const QString &page)
02 {
03 QString path = QApplication::applicationDirPath() + "/doc";
04 HelpBrowser *browser = new HelpBrowser(path, page);
05 browser->resize(500, 400);
06 browser->show();
07 }
```

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 подкаталоге *doc* того каталога, где находится исполняемый модуль 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:

```
01 void MainWindow::help()
02 {
03 HelpBrowser::showPage("index.html");
04 }
```

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:

```
01 void EntryDialog::help()
```

```
02 {
03 HelpBrowser::showPage("forms.html#editing");
04 }
```

Здесь мы выводим на экран другой справочный файл, *forms.html*, и позиционируем браузер *QTextBrowser* на метку *editing*.

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 документации.

Связь между приложением Qt и *QtAssistant* 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*.

```
01 #ifndef HELPBROWSER_H
02 #define HELPBROWSER_H
03 class QAssistantClient;
04 class QString;
05 class HelpBrowser
06 {
07 public:
08 static void showPage(const QString &page);
09 private:
10 static QAssistantClient *assistant;
11 };
12 #endif
```

Ниже приводится новый файл *helpbrowser.cpp:*

```
01 #include <QApplication>
02 #include <QAssistantClient>
03 #include "helpbrowser.h"
04 QAssistantClient *HelpBrowser::assistant = 0;
05 void HelpBrowser::showPage(const QString &page)
06 {
07 QString path = QApplication::applicationDirPath() + "/doc/" + page;
08 if (!assistant)
09 assistant = new QAssistantClient("");
10 assistant->showPage(path);
11 }
```

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 *QTextBrowser* или *Qt Assistant* можно спользовать 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*. Для приложений X11 подходящий метод мог бы состоять в создании файлов HTML и запуске веб-браузера, с использованием *QProcess*. В Mac OS X подсистема Apple Help предоставляет аналогичные функциональные возможности для *Qt Assistant*.

На этом мы завершаем часть II. В части III рассматриваются более продвинутые и специализированные средства разработки Qt. Их применение при программировании на C++ 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. Advanced Qt

Глава 17. Интернационализация

Кроме латинского алфавита, используемого для английского и многих европейских языков, Qt 4 обеспечивает широкую поддержку остальных мировых систем 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.

Разрешение ввода текста пользователями на их родном языке часто оказывается недостаточным; 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 предназначается 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[8]. *Unicode* содержит коды *ASCII* и *ISO 8859-1 (Latin-1)* в качестве своего подмножества с прежним их представлением. Например, английская буква «A» имеет значение 0x41 в кодировках *ASCII*, *Latin-1* и *Unicode*, а буква «B» 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 «A»:

```
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(0x3A3);
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 в программе Qt строк символов в кодировке *Unicode*, 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 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* и л и *Latin-1,* мы можем использовать такие стандартные функции (определенные в файле *<cctype>),* к а к *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 регистров, 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 соответствущие 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 это обычно подразумевает кодировку *Latin-1*.

Если мы разработали свой собственный формат файлов и собираемся считывать 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* могут выполняться очень быстро. Однако 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. 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 *Unicode*, если файл начинается с признака, определяющего порядок байтов. Такой режим работы можно отключить с помощью вызова *setAutoDetectUnicode(false)*. Если 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*, состоит в том, что он — супермножество 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 вполовину 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 15.

Другое применение объектов *QTextCodec* заключается в указании кодировки строк в исходном коде. Давайте рассмотрим пример, когда группа японских программистов создает приложение, предназначенное главным образом для применения на 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("♦♦"));
```

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 *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("♦♦♦♦♦");

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*
- *Big5*
- *Big5-HKSCS*
- *EUC-JP*
- *EUC-KR*
- *GB18030-0*
- *IBM 850*
- *IBM 866*
- *IBM 874*
- *ISO 2022-JP*
- *ISO 8859-1*
- *ISO 8859-2*
- *ISO 8859-3*
- *ISO 8859-4*
- *ISO 8859-5*
- *ISO 8859-6*
- *ISO 8859-7*
- *ISO 8859-8*
- *ISO 8859-9*
- *ISO 8859-10*
- *ISO 8859-13*
- *ISO 8859-14*
- *ISO 8859-15*
- *ISO 8859-16*
- *Iscii-Bng*
- *Iscii-Dev*
- *Iscii-Gjr*
- *Iscii-Knd*
- *Iscii-Mlm*
- *Iscii-Ori*
- *Iscii-Pnj*

- *Iscii-Tlg*
- *Iscii-Tml*
- *JIS X 0201*
- *JIS X 0208*
- *KOI8-R*
- *KOI8-U*
- *MuleLao-1*
- *ROMAN8*
- *Shift-JIS*
- *TIS-620*
- *TSCII*
- *UTF-8*
- *UTF-16*
- *UTF-16BE*
- *UTF-16LE*
- *Windows-1250*
- *Windows-1251*
- *Windows-1252*
- *Windows-1253*
- *Windows-1254*
- *Windows-1255*
- *Windows-1256*
- *Windows-1257*
- *Windows-1258*
- *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.

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 приводится несколько примеров:

```
01 RockyWidget::RockyWidget(QWidget *parent)
02 : QWidget(parent)
03 {
04 QString str1 = tr("Letter");
05 QString str2 = RockyWidget::tr("Letter");
06 QString str3 = SnazzyDialog::tr("Letter");
07 QString str4 = SnazzyDialog::tr("Letter", "US paper size");
08 }
```

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:

```
01 int main(int argc, char *argv[])
02 {
03 QApplication app(argc, argv);
04 QPushButton button(QObject::tr("Hello Qt!"));
05 button.show();
06 return app.exec();
07 }
```

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 *QApplication::translate()*, которая принимает три аргумента: контекст, исходный текст и необязательный комментарий. Например, ниже приводится другой способ перевода «Hello Qt!»:

```
QApplication::translate("Global Stuff", "Hello Qt!");
```

На этот раз мы поместили текст в контекст «Global Stuff» (глобальное вещество — *ну нихрена себе перевод :)*).

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 отражается на том, как следует записывать программный код. 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 возможность 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 «Нфte %1 trouvй». Параметр «%1» замещается на содержимое переменной *hostName*.

Хотя в целом не рекомендуется вызывать *tr()* для переменной, это может сработать. 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:

```
01 void OrderForm::init()
```

```
02 {
03 static const char * const flowers[] = {
04 QT_TR_NOOP("Medium Stem Pink Roses"),
05 QT_TR_NOOP("One Dozen Boxed Roses"),
06 QT_TR_NOOP("Calypso Orchid"),
07 QT_TR_NOOP("Dried Red Rose Bouquet"),
08 QT_TR_NOOP("Mixed Peonies Bouquet"),
09 0
10 };
11 for (int i = 0; flowers[i]; ++i)
12 comboBox->addItem(tr(flowers[i]));
13 }
```

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 пропущенные вызовы 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:

```
01 int main(int argc, char *argv[])
02 {
03 QApplication app(argc, argv);
04 QTranslator appTranslator;
05 appTranslator.load("myapp_" + QLocale::system().name(),
06 qApp->applicationDirPath());
07 app.installTranslator(&appTranslator);
08 …
09 return app.exec();
10 }
```

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 «^», «КЬ», «№» и «ЫІ» — *в исходном бумажном издании французский куда-то подевался %)*).

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 том же каталоге, где находится исполняемый модуль.

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(),
qApp->applicationDirPath());
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. *QApplication* 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:

```
<!DOCTYPE RCC><RCC version="1.0">
<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 Back и Forward (назад и вперед) веб-браузера 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 *e s* (Espanol), *:/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 обеспечивают определение предпочитаемого пользователем языка в функции *main()* и загрузка там соответствующих файлов *.qm*. Но в некоторых ситуациях пользователям необходимо иметь возможность динамического переключения языка. 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 нетрудно 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» (центр заказов). Приложение содержит меню Language (язык), чтобы пользователь имел возможность задавать язык во время работы приложения. По умолчанию применяется английский язык.

Рис. 17.1. 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:

```
01 MainWindow::MainWindow()
02 {
03 journalView = new JournalView;
04 setCentralWidget(journalView);
05 qApp->installTranslator(&appTranslator);
06 qApp->installTranslator(&qtTranslator);
07 qmPath = qApp->applicationDirPath() + "/translations";
08 createActions();
09 createMenus();
10 retranslateUi();
```

```
11 }
```

В конструкторе мы устанавливает центральный виджет *JournalView* как подкласс *QTableWidget*. Затем мы настраиваем несколько закрытых переменных—членов, имеющих отношение к переводу:

• переменная *appTranslator* является объектом *QTranslator*, который используется для хранения текущего перевода приложения;

• переменная *qtTranslator* является объектом *QTranslator*, который используется для хранения перевода библиотеки Qt;

• переменная *qmPath* имеет тип *QString* и задает путь к каталогу, который содержит файлы перевода приложения.

В конце мы вызываем закрытые функции *createActions()* и *createMenus()* для создания системы меню и также закрытую функцию *retranslateUi()* для первой установки значений видимых пользователем строк.

```
01 void MainWindow::createActions()
02 {
03 newAction = new QAction(this);
04 connect(newAction, SIGNAL(triggered()), this, SLOT(newFile()));
05 …
06 aboutQtAction = new QAction(this);
07 connect(aboutQtAction, SIGNAL(triggered()), qApp, SLOT(aboutQt()));
08 }
```

Функция *createActions()* создает объекты *QAction* как обычно, но без установки текстов пунктов меню и клавиш быстрого вызова команд. Это будет сделано в функции *retranslateUi()*.

```
01 void MainWindow::createMenus()
02 {
03 fileMenu = new QMenu(this);
04 fileMenu->addAction(newAction);
05 fileMenu->addAction(openAction);
06 fileMenu->addAction(saveAction);
07 fileMenu->addAction(exitAction);
08 …
09 createLanguageMenu();
10 helpMenu = new QMenu(this);
11 helpMenu->addAction(aboutAction);
12 helpMenu->addAction(aboutQtAction);
13 menuBar()->addMenu(fileMenu);
14 menuBar()->addMenu(editMenu);
15 menuBar()->addMenu(reportsMenu);
16 menuBar()->addMenu(languageMenu);
17 menuBar()->addMenu(helpMenu);
18 }
```

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():*

```
01 void MainWindow::retranslateUi()
02 {
03 newAction->setText(tr("&New"));
04 newAction->setShortcut(tr("Ctrl+N"));
05 newAction->setStatusTip(tr("Create a new journal"));
06 …
07 aboutQtAction->setText(tr("About &Qt"));
08 aboutQtAction->setStatusTip(tr("Show the Qt library's About box"));
09 fileMenu->setTitle(tr("&File"));
10 editMenu->setTitle(tr("&Edit"));
11 reportsMenu->setTitle(tr("&Reports"));
12 languageMenu->setTitle(tr("&Language"));
13 helpMenu->setTitle(tr("&Help"));
14 setWindowTitle(tr("Call Center"));
15 }
```

Именно в функции *retranslateUi()* выполняются все вызовы *tr()* для класса *MainWindow*. Она вызывается в конце конструктора *MainWindow* и также при каждом изменении пользователем языка приложения при помощи меню Language.

Мы устанавливаем для каждого пункта меню *QAction* его текст, клавишу быстрого вызова команды и комментарий в строке состояния. Мы также задаем заголовок окну и каждому меню *QMenu.*

Рассмотренная ранее функция *createMenus()* вызывала функцию *createLanguageMenu()* для заполнения меню Language списком языков:

```
01 void MainWindow::createLanguageMenu()
02 {
03 languageMenu = new QMenu(this);
04 languageActionGroup = new QActionGroup(this);
05 connect(languageActionGroup, SIGNAL(triggered(QAction *)),
06 this, SLOT(switchLanguage(QAction *)));
07 QDir dir(qmPath);
08 QStringList fileNames = dir.entryList(QStringList("callcenter_*.qm"));
09 for (int i = 0; i < fileNames.size(); ++i) {
10 QString locale = fileNames[i];
11 locale.remove(0, locale.indexOf('_') + 1);
12 locale.truncate(locale.lastIndexOf('.'));
13 QTranslator translator;
14 translator.load(fileNames[i], qmPath);
15 QString language = translator.translate("MainWindow",
16 "English");
17 QAction *action = new QAction(tr("&%1 %2")
18 .arg(i + 1).arg(language), this);
19 action->setCheckable(true);
20 action->setData(locale);
```

```
21 languageMenu->addAction(action);
22 languageActionGroup->addAction(action);
23 if (language == "English")
24 action->setChecked(true);
25 }
26 }
```

Вместо жесткого кодирования поддерживаемых приложением языков мы создаем один пункт меню для каждого файла *.qm*, расположенного в каталоге приложения *translations*. 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.

Определенную трудность составляет представление 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, «Francais» при переводе на французский язык и «♦♦♦» 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()*.

```
01 void MainWindow::switchLanguage(QAction *action)
02 {
03 QString locale = action->data().toString();
04 appTranslator.load("callcenter_" + locale, qmPath);
05 qtTranslator.load("qt_" + locale, qmPath);
06 retranslateUi();
07 }
```

Слот *switchLanguage()* вызывается, когда пользователь выбирает язык из меню Language. Мы загружаем 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:

```
01 void MainWindow::changeEvent(QEvent *event)
02 {
03 if (event->type() == QEvent::LocaleChange) {
04 appTranslator.load("callcenter_"
05 + QLocale::system().name(), qmPath);
```

```
06 qtTranslator.load("qt_" + QLocale::system().name(), qmPath);
07 retranslateUi();
08 }
09 QMainWindow::changeEvent(event);
10 }
```

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.

```
01 JournalView::JournalView(QWidget *parent)
02 : QTableWidget(parent)
03 {
04 retranslateUi();
05 }
```

Класс *JournalView* является подклассом *QTableWidget*. В конце конструктора мы вызываем закрытую функцию *retranslateUi()* для перевода строк виджета. Это напоминает то, что мы делали для класса *MainWindow*.

```
01 void JournalView::changeEvent(QEvent *event)
02 {
03 if (event->type() == QEvent::LanguageChange)
04 retranslateUi();
05 QTableWidget::changeEvent(event);
06 }
```

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 *QApplication*. 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*.

```
01 void JournalView::retranslateUi()
02 {
```

```
03 QStringList labels;
04 labels << tr("Time") << tr("Priority") << tr("Phone Number")
05 << tr("Subject");
06 setHorizontalHeaderLabels(labels);
07 }
```

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, получения 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*.

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.

С левой стороны главного окна утилиты *Qt Linguist* отображается список всех контекстов переводимого на другие языки приложения. Для приложения Электронная таблица этими контекстами являются «FindDialog», «GoToCellDialog», «MainWindow», «SortDialog» и «Spreadsheet». Справа вверху выводится список всех исходных строк для текущего контекста. Каждая исходная строка сопровождается переводом и флажком Done (готово). Справа по центру находится область, где мы можем вводить перевод текущей исходной строки. 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.

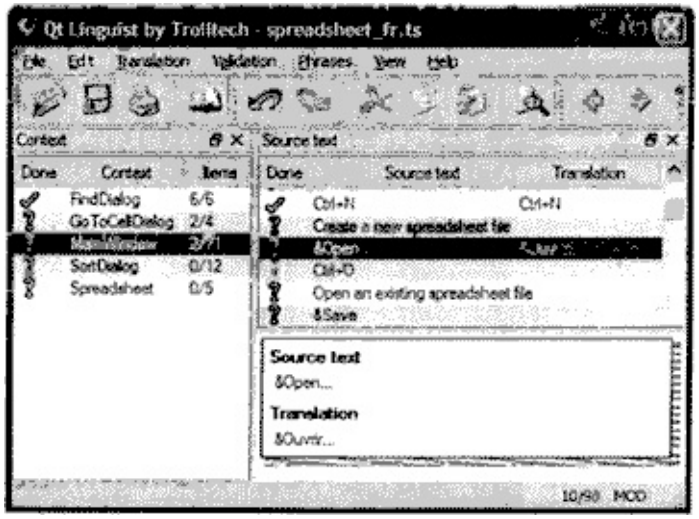


Рис. 17.2. *Qt Linguist в действии.*

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

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

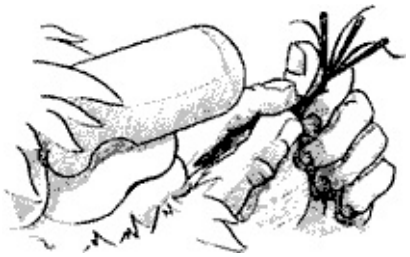
Флажок Done игнорируется утилитой *lrelease;* он может использоваться переводчиками для идентификации законченных переводов и тех, перевод которых необходимо выполнить заново. Непереведенные строки при выполнении приложения выводятся на языке оригинала.

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 «умные». 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/4.1/linguist-manual.html. Это руководство содержит полное описание интерфейса пользователя для *Qt Linguist* и учебное пособие для поэтапного обучения программистов.

Глава 18. 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. В результате такие приложения способны реагировать на действия пользователя даже при продолжительной обработке. Еще одним преимуществом многопоточной обработки является возможность в многопроцессорных системах одновременно выполнять несколько потоков на разных процессорах, увеличивая производительность.

В данной главе мы сначала продемонстрируем способы создания подкласса *QThread* и способы применения классов *QMutex*, *QSemaphore* и *QWaitCondition* для синхронизации потоков. 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.

Создание потоков

Обеспечить многопоточную обработку в приложении Qt достаточно просто: мы только 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:

```
01 class Thread : public QThread
02 {
03 Q_OBJECT
04 public:
05 Thread();
06 void setMessage(const QString &message);
07 void stop();
08 protected:
09 void run();
10 private:
11 QString messageStr;
12 volatile bool stopped;
12 };
```

Класс *Thread* наследует *QThread* и переопределяет функцию *run()*. Он содержит две дополнительные функции: *setMessage()* и *stop()*.

Переменная *stopped* объявляется со спецификатором *volatile* (изменчивый), 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.

```
01 Thread::Thread()
02 {
03 stopped = false;
04 }
```

Мы устанавливаем в конструкторе переменную *stopped* на значение *false*.

```
01 void Thread::run()
02 {
03 while (!stopped)
04 cerr << qPrintable(messageStr);
05 stopped = false;
06 cerr << endl;
07 }
```

Функция *run()* вызывается для запуска потока. Пока переменная *stopped* имеет значение *false*, эта функция будет выводить на консоль заданное сообщение. Работа потока завершается, когда завершается функция *run()*.

```
01 void Thread::stop()
02 {
03 stopped = true;
```

```
04 }
```

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. 18.1. Приложение Threads.

Теперь мы рассмотрим способы применения класса *Thread* в небольшом приложении Qt, которое применяет два потока, A и B, не считая главный поток.

```
01 class ThreadDialog : public QDialog
02 {
03 Q_OBJECT
04 public:
05 ThreadDialog(QWidget *parent = 0);
06 protected:
07 void closeEvent(QCloseEvent *event);
08 private slots:
09 void startOrStopThreadA();
10 void startOrStopThreadB();
11 private:
12 Thread threadA;
13 Thread threadB;
14 QPushButton *threadAButton;
15 QPushButton *threadBButton;
16 QPushButton *quitButton;
17 };
```

В классе *ThreadDialog* объявляются две переменные типа *Thread* и несколько кнопок для обеспечения основных средств интерфейса пользователя.

```
01 ThreadDialog::ThreadDialog(QWidget *parent)
02 : QDialog(parent)
03 {
04 threadA.setMessage("A");
05 threadB.setMessage("B");
06 threadAButton = new QPushButton(tr("Start A"));
```

```
07 threadBButton = new QPushButton(tr("Start B"));
08 quitButton = new QPushButton(tr("Quit"));
09 quitButton->setDefault(true);
10 connect(threadAButton, SIGNAL(clicked()),
11 this, SLOT(startOrStopThreadA()));
12 connect(threadBButton, SIGNAL(clicked()),
13 this, SLOT(startOrStopThreadB()));
14 …
15 }
```

В конструкторе мы вызываем функцию *setMessage()* для периодического вывода на экран первым потоком буквы «A» и вторым потоком буквы «B».

```
01 void ThreadDialog::startOrStopThreadA()
02 {
03 if (threadA.isRunning()) {
04 threadA.stop();
05 threadAButton->setText(tr("Start A"));
06 } else {
07 threadA.start();
08 threadAButton->setText(tr("Stop A"));
09 }
10 }
```

Когда пользователь нажимает кнопку потока A, функция *startOrStopThreadA()* останавливает поток, если он выполняется, и запускает его в противном случае. Она также обновляет текст кнопки.

```
01 void ThreadDialog::startOrStopThreadB()
02 {
03 if (threadB.isRunning()) {
04 threadB.stop();
05 threadBButton->setText(tr("Start B"));
06 } else {
07 threadB.start();
08 threadBButton->setText(tr("Stop B"));
09 }
10 }
```

Программный код функции *startOrStopThreadB()* очень похож.

```
01 void ThreadDialog::closeEvent(QCloseEvent *event)
02 {
03 threadA.stop();
04 threadB.stop();
05 threadA.wait();
06 threadB.wait();
07 event->accept();
08 }
```

Если пользователь выбирает пункт меню Quit или закрывает окно, мы даем команду останова для каждого выполняющегося потока и ожидаем их завершения (используя

функцию *QThread::wait())* прежде, чем сделать 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.

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:

```
01 void Thread::run()
02 {
03 forever {
04 mutex.lock();
05 if (stopped) {
06 stopped = false;
07 mutex.unlock();
08 break;
09 }
10 mutex.unlock();
11 cerr << qPrintable(messageStr.ascii);
12 }
13 cerr << endl;
14 }
```

Функция *stop()* стала бы такой:

```
01 void Thread::stop()
02 {
03 mutex.lock();
04 stopped = true;
05 mutex.unlock();
06 }
```

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:

```
01 void Thread::run()
02 {
03 forever {
04 {
05 QMutexLocker locker(&mutex);
06 if (stopped) {
07 stopped = false;
08 break;
09 }
10 }
11 cerr << qPrintable(messageStr);
12 }
13 cerr << endl;
14 }

15 void Thread::stop()
16 {
17 QMutexLocker locker(&mutex);
18 stopped = true;
18 }
```

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:

```
01 MyData data;
02 QReadWriteLock lock;

03 void ReaderThread::run()
04 {
05 …
06 lock.lockForRead();
07 access_data_without_modifying_it(&data);
08 lock.unlock();
```

```
09 …
10 }

11 void WriterThread::run()
12 {
13 …
14 lock.lockForWrite();
15 modify_data(&data);
16 lock.unlock();
17 …
18 }
```

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:*

- QSemaphore semaphore(1) — QMutex mutex,
- Semaphore.acquire() — mutex.lock(),
- Semaphore.release() — mutex.unlock().

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. 18.2 для небольшого 16-байтового буфера.

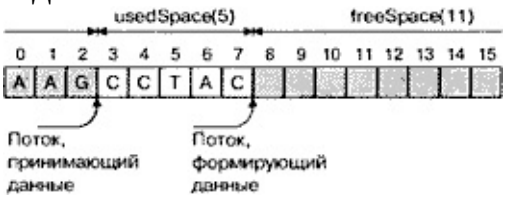


Рис. 18.2. Модель взаимодействия двух потоков: формирующего и принимающего данные.

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 считает буфер целиком и так далее. Однако в многопроцессорных системах это не позволит обоим потокам работать одновременно с разными частями буфера.

Одни 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.

```
01 void Producer::run()
02 {
03 for (int i = 0; i < DataSize; ++i) {
04 freeSpace.acquire();
05 buffer[i % BufferSize] = "ACGT"[uint(rand()) % 4];
06 usedSpace.release();
07 }
08 }
```

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 («A», «C», «G» или «T») и затем освобождаем байт и помечаем его как «использованный», тем самым указывая на возможность его считывания потоком—приемником.

```
01 void Consumer::run()
02 {
03 for (int i = 0; i < DataSize; ++i) {
04 usedSpace.acquire();
05 cerr << buffer[i % BufferSize];
06 freeSpace.release();
07 }
08 cerr << endl;
09 }
```

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, помечая его как «свободный», тем самым позволяя первому потоку вновь присвоить ему некоторое значение.

```
01 int main()
02 {
03 Producer producer;
04 Consumer consumer;
05 producer.start();
06 consumer.start();
07 producer.wait();
08 consumer.wait();
09 return 0;
10 }
```

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 «A», «C», «G» и «T» 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 «P» при генерации каждого байта первым потоком и букву «c» 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: «PcPcPcPcPcPcPcPcPcPc». 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: «PPPPccccPPPPccccPPcc». Существует много других вариантов. Семафоры дают большую свободу действий планировщикам потоков в специфических системах, что позволяет 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 путем применения 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.

```
01 void Producer::run()
02 {
03 for (int i = 0; i < DataSize; ++i) {
04 mutex.lock();
05 while (usedSpace == BufferSize)
06 bufferIsNotFull.wait(&mutex);
07 buffer[i % BufferSize] = "ACGT"[uint(rand()) % 4];
08 ++usedSpace;
09 bufferIsNotEmpty.wakeAll();
10 mutex.unlock();
11 }
12 }
```

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 вновь блокирует его 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».

```
01 void Consumer::run()
02 {
03 for (int i = 0; i < DataSize; ++i) {
04 mutex.lock();
05 while (usedSpace == 0)
06 bufferIsNotEmpty.wait(&mutex);
07 cerr << buffer[i % BufferSize];
08 --usedSpace;
09 bufferIsNotFull.wakeAll();
10 mutex.unlock();
11 }
```

```
12 cerr << endl;
13 }
```

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:

```
01 QThreadStorage<QHash<int, double> *> cache;
02 void insertIntoCache(int id, double value)
03 {
04 if (!cache.hasLocalData())
05 cache.setLocalData(new QHash<int, double>);
06 cache.localData()->insert(id, value);
07 }

08 void removeFromCache(int id)
09 {
10 if (cache.hasLocalData())
11 cache.localData()->remove(id);
12 }
```

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()*.

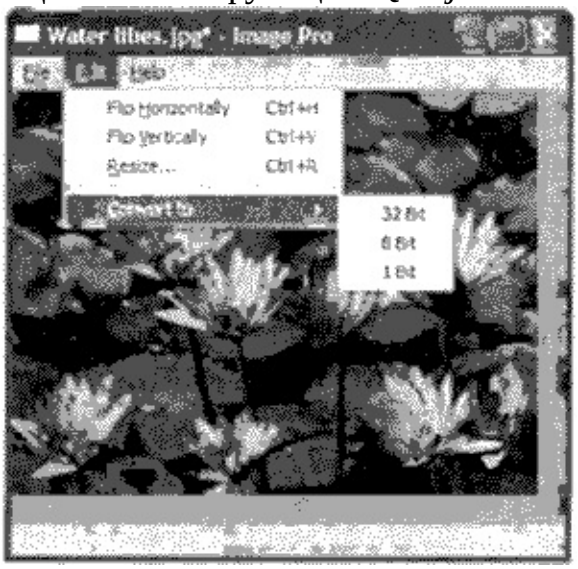


Рис. 18.3. Приложение Image Pro.

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 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 «транзакций», и он генерирует события для главного окна, чтобы сообщать о том, как идет процесс их выполнения.

```
01 ImageWindow::ImageWindow()
02 {
03 imageLabel = new QLabel;
04 imageLabel->setBackgroundRole(QPalette::Dark);
05 imageLabel->setAutoFillBackground(true);
06 imageLabel->setAlignment(Qt::AlignLeft | Qt::AlignTop);
07 setCentralWidget(imageLabel);
08 createActions();
09 createMenus();
10 statusBar()->showMessage(tr("Ready"), 2000);
11 connect(&thread, SIGNAL(transactionStarted(const QString &)),
12 statusBar(), SLOT(showMessage(const QString &)));
13 connect(&thread, SIGNAL(finished()),
14 this, SLOT(allTransactionsDone()));
15 setCurrentFile("");
16 }
```

Интересной частью конструктора *ImageWindow* являются два соединения сигнал—слот. В обоих случаях сигнал генерируется объектом *TransactionThread*, который мы вскоре рассмотрим.

```
01 void ImageWindow::flipHorizontally()
02 {
03 addTransaction(new FlipTransaction(Qt::Horizontal));
04 }
```

Слот *flipHorizontally()* создает транзакцию зеркального отражения и регистрирует ее при помощи закрытой функции *addTransaction()*. Функции *flipVertically()*, *resizeImage()*, *convertTo32Bit()*, *convertTo8Bit()* и *convertTo1Bit()* реализуются аналогично.

```
01 void ImageWindow::addTransaction(Transaction *transact)
02 {
03 thread.addTransaction(transact);
04 openAction->setEnabled(false);
05 saveAction->setEnabled(false);
06 saveAsAction->setEnabled(false);
07 }
```

Функция *addTransaction()* добавляет транзакцию в очередь транзакций вторичного потока и отключает команды Open, Save и Save As на время обработки транзакций.

```
01 void ImageWindow::allTransactionsDone()
02 {
03 openAction->setEnabled(true);
04 saveAction->setEnabled(true);
05 saveAsAction->setEnabled(true);
06 imageLabel->setPixmap(QPixmap::fromImage(thread.image()));
07 setWindowModified(true);
08 statusBar()->showMessage(tr("Ready"), 2000);
09 }
```

Слот *allTransactionsDone()* вызывается, когда очередь транзакций *TransactionThread*

становится пустой.

Теперь давайте рассмотрим класс *TransactionThread:*

```
01 class TransactionThread : public QThread
02 {
03 Q_OBJECT
04 public:
05 void addTransaction(Transaction *transact);
06 void setImage(const QImage &image);
07 QImage image();
08 signals:
09 void transactionStarted(const QString &message);
10 protected:
11 void run();
12 private:
13 QMutex mutex;
14 QImage currentImage;
15 QQueue<Transaction *> transactions;
16 };
```

Класс *TransactionThread* содержит список обрабатываемых транзакций, которые выполняются по очереди в фоновом режиме.

```
01 void TransactionThread::addTransaction(Transaction *transact)
02 {
03 QMutexLocker locker(&mutex);
04 transactions.enqueue(transact);
05 if (!isRunning())
06 start();
07 }
```

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.

```
01 void TransactionThread::setImage(const QImage &image)
02 {
03 QMutexLocker locker(&mutex);
04 currentImage = image;
05 }

06 QImage TransactionThread::image()
07 {
08 QMutexLocker locker(&mutex);
09 return currentImage;
10 }
```

Функции *setImage()* и *image()* позволяют главному потоку установить изображение, для которого будут выполняться транзакции, и получить обработанное изображение после завершения всех транзакций. И вновь мы защищаем доступ к переменной—члену при

помощи мьютекса.

```
01 void TransactionThread::run()
02 {
03 Transaction *transact;
04 forever {
05 mutex.lock();
06 if (transactions.isEmpty()) {
07 mutex.unlock();
08 break;
09 }
10 QImage oldImage = currentImage;
11 transact = transactions.dequeue();
12 mutex.unlock();

13 emit transactionStarted(transact->message());
14 QImage newImage = transact->apply(oldImage);
15 delete transact;

16 mutex.lock();
17 currentImage = newImage;
18 mutex.unlock();
19 }
20 }
```

Функция *run()* просматривает очередь транзакций и по очереди выполняет все транзакции путем вызова для них функции *apply()*.

После старта транзакции мы генерируем сигнал *transactionStarted()* с сообщением, выводимым в строке состояния приложения. Когда обработка всех транзакций завершается, функция *run()* возвращает управление и *QThread* генерирует сигнал *finished()*.

```
01 class Transaction
02 {
03 public:
04 virtual ~Transaction() { }
05 virtual QImage apply(const QImage &image) = 0;
06 virtual QString message() = 0;
07 };
```

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*. (Кроме того, если мы его не предусмотрим, некоторые компиляторы выдадут предупреждение.) *Transaction* имеет три конкретных подкласса: *FlipTransaction, ResizeTransaction* и *ConvertDepthTransaction.* Нами будет рассмотрен только подкласс *FlipTransaction;* другие два подкласса имеют аналогичное определение.

```
01 class FlipTransaction : public Transaction
02 {
```

```
03 public:
04 FlipTransaction(Qt::Orientation orientation);
05 QImage apply(const QImage &image);
06 QString message();
07 private:
08 Qt::Orientation orientation;
09 };
```

Конструктор *FlipTransaction* принимает один параметр, который задает ориентацию зеркального отражения (по горизонтали или по вертикали).

```
01 QImage FlipTransaction::apply(const QImage &image)
02 {
03 return image.mirrored(
04 orientation == Qt::Horizontal, orientation == Qt::Vertical);
05 }
```

Функция *apply()* вызывает *QImage::mirrored()* для объекта *QImage*, полученного в виде параметра, и возвращает сформированный объект *QImage*.

```
01 QString FlipTransaction::message()
02 {
03 if (orientation == Qt::Horizontal) {
04 return QObject::tr("Flipping image horizontally...");
05 } else {
06 return QObject::tr("Flipping image vertically...");
07 }
08 }
```

Функция *messageStr()* возвращает сообщение, отображаемое в строке состояния в ходе выполнения операции. Данная функция вызывается из функции *transactionThread::run()*, когда генерируется сигнал *transactionStarted()*.

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>, QWaitCondition* и часть программного интерфейса *QThread*. Кроме того, несколько функций являются потокозащищенными, в частности *QObject::connect(), QObject::disconnect(), QCoreApplication::postEvent(), QCoreApplication::removePostedEvent()* 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

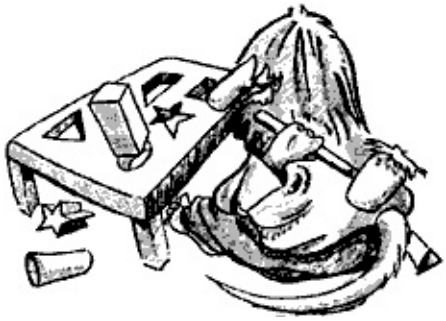
QLabel из вторичного потока, мы можем генерировать сигнал, связанный с *QLabel::setText()*, или вызвать из этого потока функцию *QMetaObject::invokeMethod()*. Например:

```
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 ассемблер 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/4.1/sql-driver.html. Полный список предостережений, относящихся к многопоточной обработке, находится на веб-странице http://doc.trolltech.com/4.1/threads.html.

Глава 19. 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* добавляются 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 разрешении 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 фреймворка 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 создавать 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.

Ниже приведен список Qt—классов подключаемых модулей и обработчиков, исключая Qtopia Core (рис. 19.1):

- QAccessibleBridgePlugin — QAccessibleBridge,
- QAccessiblePlugin — QAccessibleIntertace,
- QIconEnginePlugin — QIconEngine,
- QImageIOPlugin — QImageIOHandler,
- QInputContextPlugin — QInputContext,
- QPictureFormatPlugin — нет обработчика,
- QSqlDriverPlugin — QSqlDriver,
- QStylePlugin — QStyle,
- QTextCodecPlugin — QTextCodec.

В демонстрационных целях мы реализуем подключаемый модуль, способный считывать в Windows монохромные файлы курсоров (файлы *.cur).* Эти файлы могут содержать несколько изображений разного размера для одного курсора. После построения и установки этого подключаемого модуля Qt сможет считывать файлы *.cur* и получать доступ к отдельным курсорам (например, с помощью классов *QImage, QImageReader* или *QMovie);* также можно будет преобразовывать эти курсоры в любой другой формат файлов изображений, воспринимаемый Qt (например, BMP, JPEG и PNG). Кроме того, подключаемый модуль может разворачиваться совместно с Qt—приложениями, поскольку они автоматически проверяют стандартные места расположения подключаемых модулей Qt и загружают все найденные модули.

Новые классы—оболочки подключаемых модулей должны быть подклассом *QImageIOPlugin* и должны обеспечить реализацию нескольких виртуальных функций:

```
01 class CursorPlugin : public QImageIOPlugin
02 {
03 public:
04 QStringList keys() const;
05 Capabilities capabilities(QIODevice *device,
06 const QByteArray &format) const;
07 QImageIOHandler *create(QIODevice *device,
08 const QByteArray &format) const;
09 };
```

Функция *keys()* возвращает список форматов изображений, которые поддерживает подключаемый модуль. Можно считать, что параметр *format* функций *capabilities()* и

create() имеет значение из этого списка.

```
01 QStringList CursorPlugin::keys() const
02 {
03 return QStringList() << "cur";
04 }
```

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.

```
01 QImageIOPlugin::Capabilities
02 CursorPlugin::capabilities(QIODevice *device,
03 const QByteArray &format) const
04 {
05 if (format == "cur")
06 return CanRead;
07 if (format.isEmpty()) {
08 CursorHandler handler;
09 handler.setDevice(device);
10 if (handler.canRead())
11 return CanRead;
12 }
13 return 0;
14 }
```

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 порязрадной 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.

```
01 QImageIOHandler *CursorPlugin::create(QIODevice *device,
02 const QByteArray &format) const
03 {
04 CursorHandler *handler = new CursorHandler;
05 handler->setDevice(device);
06 handler->setFormat(format);
07 return handler;
08 }
```

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:

```
01 class CursorHandler : public QImageIOHandler
02 {
03 public:
04 CursorHandler();
05 bool canRead() const;
06 bool read(QImage *image);
07 bool jumpToNextImage();
08 int currentImageNumber() const;
09 int imageCount() const;
10 private:
11 enum State { BeforeHeader, BeforeImage, AfterLastImage, Error };
12 void readHeaderIfNecessary() const;
13 QBitArray readBitmap(int width, int height, QDataStream &in) const;
14 void enterErrorState() const;
15 mutable State state;
16 mutable int currentImageNo;
17 mutable int numImages;
18 };
```

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.

```
01 CursorHandler::CursorHandler()
02 {
03 state = BeforeHeader;
04 currentImageNo = 0;
05 numImages = 0;
06 }
```

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.

```
01 bool CursorHandler::canRead() const
```

```
02 {
03 if (state == BeforeHeader) {
04 return device()->peek(4) == QByteArray("\0\0\2\0", 4);
05 } else {
06 return state != Error;
07 }
08 }
```

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.

```
01 int CursorHandler::currentImageNumber() const
02 {
03 return currentImageNo;
04 }
```

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.

```
01 int CursorHandler::imageCount() const
02 {
03 readHeaderIfNecessary();
04 return numImages;
05 }
```

Эта функция возвращает количество изображений, содержащихся в файле. Для правильного файла, при чтении которого не возникает ошибок, она возвращает по крайней мере 1.

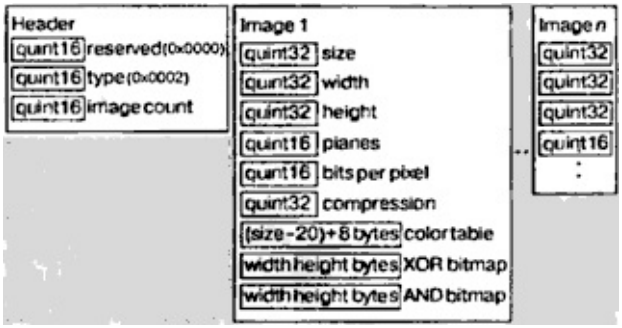


Рис. 19.2. Формат файла .cur.

Следующая функция довольно сложная, поэтому мы рассмотрим ее по частям:

```
01 bool CursorHandler::read(QImage *image)
```

```
02 {
03 readHeaderIfNecessary();
04 if (state != BeforeImage)
05 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.

```
06 quint32 size;
07 quint32 width;
08 quint32 height;
09 quint16 numPlanes;
10 quint16 bitsPerPixel;
11 quint32 compression;
12 QDataStream in(device());
13 in.setByteOrder(QDataStream::LittleEndian);
14 in >> size;
15 if (size != 40) {
16 enterErrorState();
17 return false;
18 }
19 in >> width >> height >> numPlanes >> bitsPerPixel >> compression;
20 height /= 2;
21 if (numPlanes != 1 || bitsPerPixel != 1 || compression != 0) {
22 enterErrorState();
23 return false;
24 }
25 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, уменьшая 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*.

```
26 QBitArray xorBitmap = readBitmap(width, height, in);
27 QBitArray andBitmap = readBitmap(width, height, in);
28 if (in.status() != QDataStream::Ok) {
29 enterErrorState();
30 return false;
31 }
```

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 бит завершается неудачей, попытка чтения второго массива в результате даст пустой массив *QBitArray*.

```
32 *image = QImage(width, height, QImage::Format_ARGB32);
33 for (int i = 0; i < int(height); ++i) {
34 for (int j = 0; j < int(width); ++j) {
35 QRgb color;
36 int bit = (i * width) + j;
37 if (andBitmap.testBit(bit)) {
38 if (xorBitmap.testBit(bit)) {
39 color = 0x7F7F7F7F;
40 } else {
41 color = 0x00FFFFFF;
42 }
43 } else {
44 if (xorBitmap.testBit(bit)) {
45 color = 0xFFFFFFFF;
46 } else {
47 color = 0xFF000000;
48 }
50 }
51 image->setPixel(j, i, color);
52 }
53 }
```

Мы конструируем новый объект *QImage* с правильными размерами и устанавливаем на него указатель изображения. 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 пикселей 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)*.

```
54 ++currentImageNo;
```

```
55 if (currentImageNo == numImages)
56 state = AfterLastImage;
57 return true;
58 }
```

Завершив чтение изображения, мы обновляем текущий номер изображения и обновляем состояние, если прочитано последнее изображение. В конце функции устройство будет указывать на начало следующего изображения или на конец файла.

```
01 bool CursorHandler::jumpToNextImage()
02 {
03 QImage image;
04 return read(&image);
05 }
```

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.

```
01 void CursorHandler::readHeaderIfNecessary() const
02 {
03 if (state != BeforeHeader)
04 return;
05 quint16 reserved;
06 quint16 type;
07 quint16 count;
08 QDataStream in(device());
09 in.setByteOrder(QDataStream::LittleEndian);
10 in >> reserved >> type >> count;
11 in.skipRawData(16 * count);
12 if (in.status() != QDataStream::Ok || reserved != 0
13 || type != 2 || count == 0) {
14 enterErrorState();
15 return;
16 }
17 state = BeforeImage;
18 currentImageNo = 0;
19 numImages = int(count);
20 }
```

Закрытая функция *readHeaderIfNecessary()* вызывается из *imageCount()* и *read()*. Если заголовок файла уже был прочитан, состояние не будет иметь значение *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.

```
01 void CursorHandler::enterErrorState() const
02 {
```

```
03 currentImageNo = 0;
04 numImages = 0;
05 state = Error;
06 }
```

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.

```
01 QBitArray CursorHandler::readBitmap(int width, int height,
02 QDataStream &in) const
03 {
04 QBitArray bitmap(width * height);
05 quint8 byte;
06 quint32 word;
07 for (int i = 0; i < height; ++i) {
08 for (int j = 0; j < width; ++j) {
09 if ((j % 32) == 0) {
10 word = 0;
11 for (int k = 0; k < 4; ++k) {
12 in >> byte;
13 word = (word << 8) | byte;
14 }
15 }
16 bitmap.setBit(((height - i - 1) * width) + j,
17 word & 0x80000000);
18 word <<= 1;
19 }
20 }
21 return bitmap;
22 }
```

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.

Этим завершается реализация класса *CursorHandler* — подключаемого модуля, предназначенного для работы с изображениями курсоров. Подключаемые модули для изображений других форматов могли бы создаваться аналогично, хотя в некоторых случаях может потребоваться реализация дополнительных функций программного интерфейса *QImageIOHandler*, в частности функций, используемых для записи изображений. Подключаемые модули другого вида, например кодировки текста или драйверы баз данных, создаются по тому же самому образцу: реализуются класс—оболочка, обеспечивающий общий программный интерфейс подключаемых модулей, который может использоваться приложением, и обработчик, обеспечивающий базовую функциональность.

Файл *.pro* для подключаемых модулей отличается от файлов *.pro*, используемых для приложений, поэтому мы покажем его состав:

```
TEMPLATE = lib
CONFIG += plugin
HEADERS = cursorhandler.h \
cursorplugin.h
SOURCES = cursorhandler.cpp \
cursorplugin.cpp
DESTDIR = $(QTDIR)/plugins/imageformats
```

По умолчанию файлы *.pro* используют шаблон *app*, но здесь мы должны указать шаблон *lib*, потому что подключаемый модуль является библиотекой, а не автономным приложением. Строка с элементом *CONFIG* указывает Qt на то, что у нас не простая библиотека, а библиотека подключаемого модуля. Элемент *DESTDIR* определяет каталог размещения подключаемого модуля. Каждый подключаемый модуль Qt должен находиться в соответствующем подкаталоге каталога *plugins*, и поскольку наш подключаемый модуль обеспечивает новый формат изображений, помещаем его в *plugins/imageformats*. Список имен каталогов и типов подключаемых модулей приводится на веб-странице http://doc.trolltech.com/4.1/plugins-howto.html. В данном случае мы предполагаем, что переменная среды *QTDIR* определяет каталог, в котором находится Qt.

Для Qt в рабочем (release) и отладочном (debug) режимах создаются различные подключаемые модули, поэтому, если установлены обе версии Qt, имеет смысл указать в файле *.pro* ту из них, которая будет использоваться, добавляя строку

```
CONFIG += release
```

Приложения, использующие подключаемые модули Qt, должны разворачиваться совместно со своими подключаемыми модулями. Подключаемые модули Qt должны располагаться в конкретных подкаталогах (например, в *imageformats* для форматов изображений). Приложения Qt ищут подключаемые модули в каталоге *plugins*, который располагается в каталоге размещения исполняемого модуля приложения, поэтому поиск подключаемых модулей изображений будет выполняться в *application_dir/plugins/imageformats*. Если требуется развернуть подключаемые модули Qt в другом каталоге, можно установить дополнительный путь поиска, используя функцию *QCoreApplication::addLibraryPath()*.

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. 19.3. 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*.

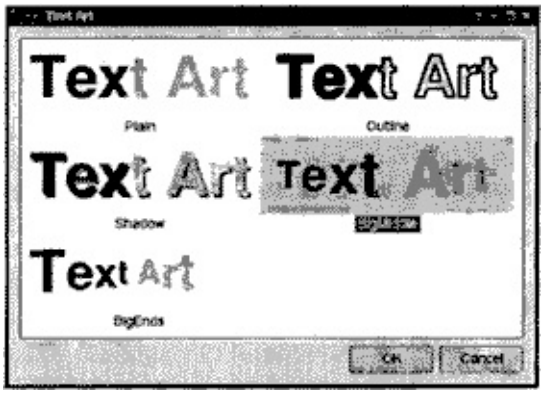


Рис. 19.3. Приложение Text Art.

В приложении Text Art определяется один интерфейс:

```
01 class TextArtInterface
02 {
03 public:
04 virtual ~TextArtInterface() { }
05 virtual QStringList effects() const = 0;
06 virtual QPixmap applyEffect(const QString &effect,
07 const QString &text,
08 const QFont &font,
09 const QSize &size,
10 const QPen &pen,
11 const QBrush &brush) = 0;
12 };
13 Q_DECLARE_INTERFACE(TextArtInterface,
14 "com.software-inc.TextArt.TextArtInterface/1.0")
```

В классе интерфейса обычно объявляются 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.

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:

```
01 TextArtDialog::TextArtDialog(const QString &text, QWidget *parent)
02 : QDialog(parent)
03 {
04 listWidget = new QListWidget;
05 listWidget->setViewMode(QListWidget::IconMode);
06 listWidget->setMovement(QListWidget::Static);
07 listWidget->setIconSize(QSize(260, 80));
08 …
09 loadPlugins();
10 populateListWidget(text);
11 …
12 }
```

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()*.

```
01 void TextArtDialog::loadPlugins()
02 {
03 QDir pluginDir(QApplication::applicationDirPath());
04 #if defined(Q_OS_WIN)
05 if (pluginDir.dirName().toLower() == "debug"
06 || pluginDir.dirName().toLower() == "release")
07 pluginDir.cdUp();
08 #elif defined(Q_OS_MAC)
09 if (pluginDir.dirName() == "MacOS") {
10 pluginDir.cdUp();
11 pluginDir.cdUp();
12 pluginDir.cdUp();
13 }
```

```
14 #endif
15 if (!pluginDir.cd("plugins"))
16 return;
17 foreach (QString fileName, pluginDir.entryList(QDir::Files)) {
18 QPluginLoader loader(pluginDir.absoluteFilePath(fileName));
19 if (TextArtInterface *interface =
20 qobject_cast<TextArtInterface *>(loader.instance()))
21 interfaces.append(interface);
22 }
23 }
```

В функции *loadPlugins()* мы пытаемся загрузить все файлы, находящиеся в каталоге приложения *plugins*. (В Windows исполняемый модуль приложения обычно находится в подкаталоге *debug* или *release*, поэтому поднимаемся на один каталог выше. В Mac OS X учитываем структуру группового каталога *(bundle directory)*.)

Если файл, который мы пытаемся загрузить, является подключаемым модулем Qt и имеет ту же саму 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 получении этих интерфейсов мог бы выглядеть следующим образом:

```
01 QObject *plugin = loader.instance();
02 if (TextArtInterface *i = qobject_cast<TextArtInterface *>(plugin))
03 textArtInterfaces.append(i);
04 if (BorderArtInterface *i = qobject_cast<BorderArtInterface *>(plugin))
05 borderArtInterfaces.append(i);
06 if (TextureInterface *i = qobject_cast<TextureInterface *>(plugin))
07 textureInterfaces.append(i);
```

Тип одного подключаемого модуля может успешно приводиться к нескольким указателям интерфейсов, поскольку подключаемые модули могут обеспечивать несколько интерфейсов, используя множественное наследование.

```
01 void TextArtDialog::populateListWidget(const QString &text)
02 {
03 QSize iconSize = listWidget->iconSize();
04 QPen pen(QColor("darkseagreen"));
05 QLinearGradient gradient(0, 0, iconSize.width() / 2,
06 iconSize.height() / 2);
07 gradient.setColorAt(0.0. QColor("darkolivegreen"));
08 gradient.setColorAt(0.8, QColor("darkgreen"));
09 gradient.setColorAt(1.0, QColor("lightgreen"));
10 QFont font("Helvetica", iconSize.height(), QFont::Bold);
11 foreach (TextArtInterface *interface, interfaces) {
12 foreach (QString effect, interface->effects()) {
```

```
13 QListWidgetItem *item = new QListWidgetItem(
14 effect, listWidget);
15 QPixmap pixmap = interface->applyEffect(effect,
16 text, font, iconSize, pen, gradient);
17 item->setData(Qt::DecorationRole, pixmap);
18 }
19 }
20 listWidget->setCurrentRow(0);
21 }
```

Функция *populateListWidget()* начинается с создания некоторых переменных, передаваемых функции *applyEffect()*, в частности пера, линейного градиента и шрифта. 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*, используя *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. Прилагаемый к этой книге компакт-диск содержит 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 Plugin (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:

```
01 class BasicEffectsPlugin
02 : public QObject, public TextArtInterface
03 {
04 Q_OBJECT
05 Q_INTERFACES(TextArtInterface)
06 public:
07 QStringList effects() const;
08 QPixmap applyEffect(const QString &effect, const QString &text,
09 const QFont &font, const QSize &size,
10 const QPen &pen, const QBrush &brush);
11 };
```

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>()*.

```
01 QStringList BasicEffectsPlugin::effects() const
02 {
03 return QStringList() << "Plain" << "Outline" << "Shadow";
04 }
```

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:

```
01 QPixmap BasicEffectsPlugin::applyEffect(const QString &effect,
02 const QString &text, const QFont &font, const QSize &size,
03 const QPen &pen, const QBrush &brush)
04 {
05 QFont myFont = font;
06 QFontMetrics metrics(myFont);
07 while ((metrics.width(text) > size.width() ||
08 metrics.height() > size.height())
09 && myFont.pointSize() > 9) {
```

```
10 myFont.setPointSize(myFont.pointSize() - 1);
11 metrics = QFontMetrics(myFont);
12 }
```

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.

```
13 QPixmap pixmap(size);
14 QPainter painter(&pixmap);
15 painter.setFont(myFont);
16 painter.setPen(pen);
17 painter.setBrush(brush);
18 painter.setRenderHint(QPainter::Antialiasing, true);
19 painter.setRenderHint(QPainter::TextAntialiasing, true);
20 painter.setRenderHint(QPainter::SmoothPixmapTransform, true);
21 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.

```
22 if (effect == "Plain") {
23 painter.setPen(Qt::NoPen);
24 } else if (effect == "Outline") {
25 QPen pen(Qt::black);
26 pen.setWidthF(2.5);
27 painter.setPen(pen);
28 } else if (effect == "Shadow") {
29 QPainterPath path;
30 painter.setBrush(Qt::darkGray);
31 path.addText(((size.width() - metrics.width(text)) / 2) + 3,
32 (size.height() - metrics.descent()) + 3, myFont, text);
33 painter.drawPath(path);
34 painter.setBrush(brush);
35 }
```

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 нее.

```
36 QPainterPath path;
37 path.addText((size.width() - metrics.width(text)) / 2,
38 size.height() - metrics.descent(), myFont, text);
39 painter.drawPath(path);
40 return pixmap;
41 }
```

Теперь у нас имеются перо и кисти, соответствующим образом установленные для

каждого текстового эффекта, а для эффекта «Shadow» нарисована 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 курсоров Windows:

```
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 (информация о подключаемых модулях), которое содержит списки подключаемых модулей и интерфейсов, доступных в приложении.

Глава 20. 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, позволяющие упростить перевод в Qt приложений Motif/Xt и MFC. Подобное расширение для приложений Tcl/Tk обеспечивается фирмой «Froglogic», а компанией «Klaralvdalens Datakonsult» разработан конвертор ресурсов Windows компании Microsoft. Дополнительную информацию вы найдете на следующих веб-страницах:

- http://www.trolltech.com/products/solutions/catalog/
- http://www.froglogic.com/tq/
- http://www.kdab.net/knut/

Для встроенных приложений компания «Trolltech» обеспечивает Qtopia — рабочую среду для разработки таких приложений. Она рассматривается в главе 21.

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, зависимых от платформы. Например, в следующем программном коде используется функция *winId()* для отображения слева заголовка панели инструментов, используя «родные» функции Mac OS X:

```
#ifdef Q_WS_MAC
ChangeWindowAttributes(HIViewGetWindow(HIViewRef(toolWin.winId())),
kWindowSideTitlebarAttribute, kWindowNoAttributes);
#endif
```

Рис. 20.1. 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:

```
01 void GdiControl::paintEvent(QPaintEvent * /* event */)
02 {
03 RECT rect;
04 GetClientRect(winId(), &rect);
05 HDC hdc = GetDC(winId());
06 FillRect(hdc, &rect, HBRUSH(COLOR_WINDOW + 1));
07 SetTextAlign(hdc, TA_CENTER | TA_BASELINE);
```

```
08 TextOutW(hdc, width() / 2, height() / 2,
09 text.utf16(), text.size());
10 ReleaseDC(winId(), hdc);
11 }
```

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:*

```
01 void MyWidget::paintEvent(QPaintEvent * /* event */)
02 {
03 QPainter painter(this);
04 painter.fillRect(rect().adjusted(20, 20, -20, -20), Qt::red);
05 #ifdef Q_WS_WIN
06 HDC hdc = painter.paintEngine()->getDC();
07 Rectangle(hdc, 40, 40, width() - 40, height() - 40);
08 painter.paintEngine()->releaseDC();
09 #endif
10 }
```

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* и *Q_WS_QWS* (Qtopia). Мы должны обеспечить включение хотя бы одного заголовка Qt перед их использованием в приложениях. Qt также обеспечивает препроцессорные символы для идентификации операционной системы:

- *Q_OS_AIX*
- *Q_OS_BSD4*
- *Q_OS_BSDI*
- *Q_OS_CYGWIN*
- *Q_OS_DGUX*
- *Q_OS_DYNIX*
- *Q_OS_FREEBSD*
- *Q_OS_HPUX*
- *Q_OS_HURD*
- *Q_OS_IRIX*
- *Q_OS_LINUX*
- *Q_OS_LYNX*
- *Q_OS_MAC*
- *Q_OS_NETBSD*
- *Q_OS_OPENBSD*
- *Q_OS_OS2EMX*
- *Q_OS_OSF*
- *Q_OS_QNX6*
- *Q_OS_QNX*

• *Q_OS_RELIANT*
• *Q_OS_SCO*
• *Q_OS_SOLARIS*
• *Q_OS_ULTRIX*
• *Q_OS_UNIXWARE*
• *Q_OS_WIN32*
• *Q_OS_WIN64*

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 установки отличий между различными версиями Windows (2000, ME и так далее) или Mac OS X (10.2, 10.3 и так далее).

Кроме макросов операционной и оконной систем существует также ряд макросов компилятора. Например, *Q_CC_MSVC* определяется в том случае, если компилятором является Visual C++ компании Microsoft. Такие макросы полезны, когда приходится обходить ошибки компилятора.

Несколько классов графического пользовательского интерфейса Qt обеспечивают зависимые от платформы функции, которые возвращают описатели (handle) базового объекта для низкоуровневой обработки. Они перечислены на рис. 20.2:

Mac OS X:
• *ATSFontFormatRef QFont::handle();*
• *CGImageRef QPixmap::macCGHandle();*
• *GWorldPtr QPixmap::macQDAlphaHandle();*
• *GWorldPtr QPixmap::macQDHandle();*
• *RgnHandle QRegion::handle();*
• *HIViewRef 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();*

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 Qtopia), прежде чем они будут преобразованы в события *QEvent*. Мы можем делать это путем создания подкласса *QApplication* и переопределения соответствующего зависимого от платформы фильтра событий — одну из следующих функций: *x11EventFilter(), winEventFilter(), macEventFilter()* и *qwsEventFilter()*. Мы можем поступать по-другому и осуществлять доступ к зависимым от платформы событиям, которые передаются заданному *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 события джойстика.

Более подробную информацию относительно применения зависимых от платформы средств, в том числе как развертывать приложения Qt на различных платформах, можно найти в сети Интернет по адресу http://doc.trolltech.com/4.1/win-system.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.

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 вприложение Qt при помощи модуля *QAxContainer.* Приложение Qt добавляет кнопку Open, кнопку Play/Pause, кнопку Stop и ползунок в элемент управления ActiveX Media Player системы Windows.

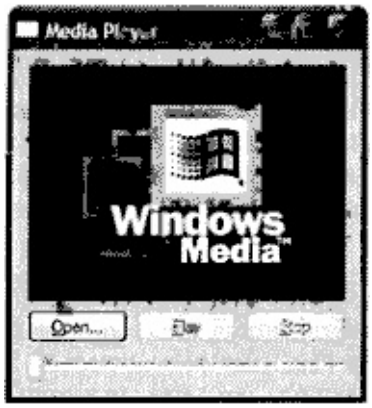


Рис. 20.3. Приложение Media Player.

Главное окно приложения имеет тип *PlayerWindow:*

```
01 class PlayerWindow : public QWidget
02 {
03 Q_OBJECT
04 Q_ENUMS(ReadyStateConstants)
05 public:
06 enum PlayStateConstants {
07 Stopped = 0, Paused = 1, Playing = 2 };
08 enum ReadyStateConstants {
09 Uninitialized = 0, Loading = 1, Interactive = 3, Complete = 4 };
10 PlayerWindow();
11 protected:
12 void timerEvent(QTimerEvent *event);
13 private slots:
14 void onPlayStateChange(int oldState, int newState);
15 void onReadyStateChange(ReadyStateConstants readyState);
```

```
16 void onPositionChange(double oldPos, double newPos);
17 void sliderValueChanged(int newValue);
18 void openFile();
19 private:
20 QAxWidget *wmp;
21 QToolButton *openButton;
22 QToolButton *playPauseButton;
23 QToolButton *stopButton;
24 QSlider *seekSlider;
25 QString fileFilters;
26 int updateTimer;
27 };
```

Класс *PlayerWindow* наследует *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 *.

```
01 PlayerWindow::PlayerWindow()
02 {
03 wmp = new QAxWidget;
04 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*.

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*.

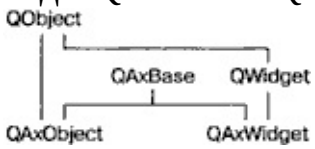


Рис. 20.4. Дерево наследования для модуля QAxContainer.

Типы данных COM автоматически преобразуются в соответствующие типы объектов, как показано на рис. 20.5:

- *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*.

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 так далее), входной параметр становится ссылкой с модификатором *const* (например, *const QString &)*.

Для получения списка всех свойств, сигналов и слотов, доступных в объектах *QAxObject* и *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:*

```
05 wmp->setProperty("ShowControls", false);
06 wmp->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
07 connect(wmp, SIGNAL(PlayStateChange(int, int)),
08 this, SLOT(onPlayStateChange(int, int)));
09 connect(wmp, SIGNAL(ReadyStateChange(ReadyStateConstants)),
10 this, SLOT(onReadyStateChange(ReadyStateConstants)));
11 connect(wmp, SIGNAL(PositionChange(double, double)),
12 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 подсоединяем три события ActiveX компонента COM к трем слотам.

```
13 stopButton = new QToolButton;
14 stopButton->setText(tr("&Stop"));
15 stopButton->setEnabled(false);
16 connect(stopButton, SIGNAL(clicked()), wmp, SLOT(Stop()));
17 …
18 }
```

Остальная часть конструктора *PlayerWindow* следует обычному образцу, за исключением того, что мы подсоединяем некоторые сигналы Qt к слотам объекта COM *(Play(), Pause()* и *Stop())*. Мы показали здесь реализацию только кнопки Stop, поскольку другие кнопки реализуются аналогично.

```
01 void PlayerWindow::timerEvent(QTimerEvent *event)
02 {
03 if (event->timerId() == updateTimer) {
04 double curPos = wmp->property("CurrentPosition").toDouble();
```

```
05 onPositionChange(-1, curPos);
06 } else {
07 QWidget::timerEvent(event);
08 }
09 }
```

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 имеется на компакт-диске.

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(01DefaultFolders)",
"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.

Рис. 20.6. Виджет AxBouncer в Internet Explorer.

Ниже приводится начало определения класса виджета *AxBouncer:*

```
01 class AxBouncer : public QWidget, public QAxBindable
02 {
03 Q_OBJECT
04 Q_ENUMS(SpeedValue)
05 Q_PROPERTY(QColor color READ color WRITE setColor)
06 Q_PROPERTY(SpeedValue speed READ speed WRITE setSpeed)
07 Q_PROPERTY(int radius READ radius WRITE setRadius)
08 Q_PROPERTY(bool running READ isRunning)
```

AxBouncer наследует как *QWidget*, так 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*.

Если при использовании множественного наследования имеются классы, производные от *QObject*, мы должны всегда располагать производные от *QObject* классы первыми для того, чтобы компилятор *moc* 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:

```
09 public:
10 enum SpeedValue { Slow, Normal, Fast };
11 AxBouncer(QWidget *parent = 0);
12 void setSpeed(SpeedValue newSpeed);
13 SpeedValue speed() const { return ballSpeed; }
14 void setRadius(int newRadius);
15 int radius() const { return ballRadius; }
16 void setColor(const QColor &newColor);
17 QColor color() const { return ballColor; }
18 bool isRunning() const { return myTimerId != 0; }
19 QSize sizeHint() const;
```

```
20 QAxAggregated *createAggregate();
21 public slots:
22 void start();
23 void stop();
24 signals:
25 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.

```
26 protected:
27 void paintEvent(QPaintEvent *event);
28 void timerEvent(QTimerEvent *event);
29 private:
30 int intervalInMilliseconds() const;
31 QColor ballColor;
32 SpeedValue ballSpeed;
33 int ballRadius;
34 int myTimerId;
35 int x;
36 int delta;
37 };
```

Защищенная и закрытая секции этого класса имеют тот же вид, как и для стандартного виджета Qt.

```
01 AxBouncer::AxBouncer(QWidget *parent)
02 : QWidget(parent)
03 {
04 ballColor = Qt::blue;
05 ballSpeed = Normal;
06 ballRadius = 15;
07 myTimerId = 0;
08 x = 20;
09 delta = 2;
10 }
```

Конструктор AxBouncer инициализирует закрытые переменные этого класса.

```
01 void AxBouncer::setColor(const QColor &newColor)
02 {
03 if (newColor != ballColor &&
04 requestPropertyChange("color")) {
05 ballColor = newColor;
06 update();
07 propertyChanged("color");
08 }
09 }
```

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 так же 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:*

```
01 class ObjectSafetyImpl : public QAxAggregated, public IObjectSafety
02 {
03 public:
04 long queryInterface(const QUuid &iid, void **iface);
05 QAXAGG_IUNKNOWN
06 HRESULT WINAPI GetInterfaceSafetyOptions(REFIID riid,
07 DWORD *pdwSupportedOptions, DWORD *pdwEnabledOptions);
08 HRESULT WINAPI SetInterfaceSafetyOptions(REFIID riid,
09 DWORD pdwSupportedOptions, DWORD pdwEnabledOptions);
10 };
```

Класс *ObjectSafetyImpl* наследует как *QAxAggregated*, так и *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.

```
01 long ObjectSafetyImpl::queryInterface(const QUuid &iid, void **iface)
02 {
03 *iface = 0;
```

```
04 if (iid == IID_IObjectSafety) {
05 *iface = static_cast<IObjectSafety *>(this);
06 } else {
07 return E_NOINTERFACE;
08 }
09 AddRef();
10 return S_OK;
11 }
```

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.*

```
01 HRESULT WINAPI ObjectSafetyImpl::GetInterfaceSafetyOptions(
02 REFIID /* riid */, DWORD *pdwSupportedOptions,
03 DWORD *pdwEnabledOptions)
04 {
05 *pdwSupportedOptions =
06 INTERFACESAFE_FOR_UNTRUSTED_DATA
07 | INTERFACESAFE_FOR_UNTRUSTED_CALLER;
08 *pdwEnabledOptions = *pdwSupportedOptions;
09 return S_OK;
10 }

11 HRESULT WINAPI ObjectSafetyImpl::SetInterfaceSafetyOptions(
12 REFIID /* riid */, DWORD /* pdwSupportedOptions */,
13 DWORD /* pdwEnabledOptions */)
14 {
15 return S_OK;
16 }
```

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:*

```
01 #include <QAxFactory>
02 #include "axbouncer.h"
03 QAXFACTORY_DEFAULT(AxBouncer,
04 "{5e2461aa-a3e8-4f7a-8b04-307459a4c08c}",
05 "{533af11f-4899-43de-8b7f-2ddf588d1015}",
06 "{772c14a5-a840-4023-b79d-19549ece0cd9}",
07 "{dbce1e56-70dd-4f74-85e0-95c65d86254d}",
08 "{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. расположенный на компакт-диске 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.

```
01 class AddressBook : public QMainWindow
02 {
03 Q_OBJECT
04 Q_PROPERTY(int count READ count)
```

```
05 Q_CLASSINFO("ClassID",
06 "{588141ef-110d-4beb-95ab-ee6a478b576d}")
07 Q_CLASSINFO("InterfaceID",
08 "{718780ec-b30c-4d88-83b3-79b3d9e78502}")
09 Q_CLASSINFO("ToSuperClass", "AddressBook")
10 public:
11 AddressBook(QWidget *parent = 0);
12 ~AddressBook();
13 int count() const;
14 public slots:
15 ABItem *createEntry(const QString &contact);
16 ABItem *findEntry(const QString &contact) const;
17 ABItem *entryAt(int index) const;
18 private slots:
19 void addEntry();
20 void editEntry();
21 void deleteEntry();
22 private:
23 void createActions();
24 void createMenus();
25 QTreeWidget *treeWidget;
26 QMenu *fileMenu;
27 QMenu *editMenu;
28 QAction *exitAction;
29 QAction *addEntryAction;
30 QAction *editEntryAction;
31 QAction *deleteEntryAction;
32 };
```

Виджет *AddressBook* является главным окном приложения. Предоставляемые им свойства и 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 суперклассы вплоть до *QWidget*. Атрибут *ToSuperClass* позволяет определить суперкласс 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, определенные в суперклассах *AddressBook*.

```
01 class ABItem : public QObject, public QListViewItem
02 {
03 Q_OBJECT
04 Q_PROPERTY(QString contact READ contact WRITE setContact)
05 Q_PROPERTY(QString address READ address WRITE setAddress)
06 Q_PROPERTY(QString phoneNumber
07 READ phoneNumber WRITE setPhoneNumber)
08 Q_CLASSINFO("ClassID",
09 "{bc82730e-5f39-4e5c-96be-461c2cd0d282}")
10 Q_CLASSINFO("InterfaceID",
11 "{c8bc1656-870e-48a9-9937-fbe1ceff8b2e}")
12 Q_CLASSINFO("ToSuperClass", "ABItem")
13 public:
14 ABItem(QTreeWidget *treeWidget);
15 void setContact(const QString &contact);
16 QString contact() const { return text(0); }
17 void setAddress(const QString &address);
18 QString address() const { return text(1); }
19 void setPhoneNumber(const QString &number);
20 QString phoneNumber() const { return text(2); }
21 public slots:
22 void remove();
23 };
```

Класс *ABItem* представляет один элемент в адресной книге. Он наследует *QTreeWidgetItem* и поэтому может отображаться в *QTreeWidget*, и он также наследует *QObject* и поэтому может экспортироваться как объект COM.

```
01 int main(int argc, char *argv[])
02 {
03 QApplication app(argc, argv);
04 if (!QAxFactory::isServer()) {
05 AddressBook addressBook;
06 addressBook.show();
07 return app.exec();
08 }
09 return app.exec();
10 }
```

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.

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.

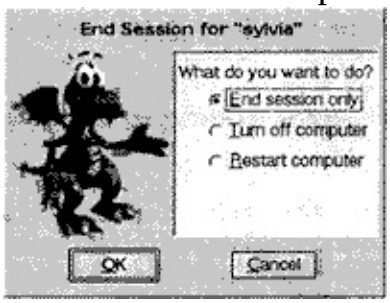


Рис. 20.7. Выход из системы KDE.

Windows 2000 и XP, 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 приложения Tic-Tac-Toe (крестики-нолики), которое работает под управлением менеджера сеансов. Во-первых, давайте рассмотрим функцию *main():*

```
01 int main(int argc, char *argv[])
02 {
03 Application app(argc, argv);
04 TicTacToe toe;
05 toe.setObjectName("toe");
06 app.setTicTacToe(&toe);
07 toe.show();
08 return app.exec();
09 }
```

Мы создаем объект *Application.* Класс *Application* наследует *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.

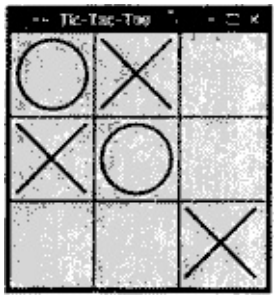


Рис. 20.8. Приложение Tic-Tac-Toe.

Ниже приводится определение класса *Application:*

```
01 class Application : public QApplication
02 {
03 Q_OBJECT
04 public:
05 Application(int &argc, char *argv[]);
06 void setTicTacToe(TicTacToe *tic);
07 void saveState(QSessionManager &sessionManager);
08 void commitData(QSessionManager &sessionManager);
09 private:
10 TicTacToe *ticTacToe;
11 };
```

Класс *Application* сохраняет указатель виджета *TicTacToe* в закрытой переменной.

```
01 void Application::saveState(QSessionManager &sessionManager)
02 {
03 QString fileName = ticTacToe->saveState();
04 QStringList discardCommand;
05 discardCommand << "rm" << fileName;
06 sessionManager.setDiscardCommand(discardCommand);
07 }
```

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_ключ

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 TicTacToe, вы должны установить его в каталог, например, */usr/bin* и вызывать его по команде *tictactoe*.

Для простых приложений, в том числе и для TicTacToe, 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.

```
01 void Application::commitData(QSessionManager &sessionManager)
02 {
03 if (ticTacToe->gameInProgress()
04 && sessionManager.allowsInteraction()) {
05 int r = QMessageBox::warning(ticTacToe, tr("Tic-Tac-Toe"),
06 tr("The game hasn't finished.\n"
07 "Do you really want to quit?"),
08 QMessageBox::Yes | QMessageBox::Default,
09 QMessageBox::No | QMessageBox::Escape);
10 if (r == QMessageBox::Yes) {
11 sessionManager.release();
12 } else {
13 sessionManager.cancel();
14 }
15 }
16 }
```

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.

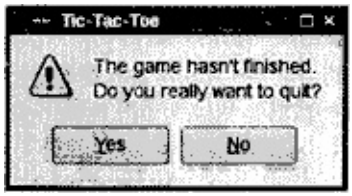


Рис. 20.9. «Вы действительно хотите завершить работу?».

Теперь давайте рассмотрим класс *TicTacToe:*

```
01 class TicTacToe : public QWidget
02 {
03 Q_OBJECT
04 public:
05 TicTacToe(QWidget *parent = 0);
06 bool gameInProgress() const;
07 QString saveState() const;
08 QSize sizeHint() const;

09 protected:
10 void paintEvent(QPaintEvent *event);
11 void mousePressEvent(QMouseEvent *event);

12 private:
13 enum { Empty = '-', Cross = 'X', Nought = '0' };

14 void clearBoard();
15 void restoreState();
16 QString sessionFileName() const;
17 QRect cellRect(int row, int column) const;
18 int cellWidth() const { return width() / 3; }
19 int cellHeight() const { return height() / 3; }
20 bool threeInARow(int row1, int col1, int row3, int col3) const;

21 char board[3][3];
22 int turnNumber;
23 };
```

Класс *TicTacToe* наследует *QWidget* и переопределяет функции *sizeHint(), paintEvent()* и *mousePressEvent()*. Он также обеспечивает функции *gameInProgress()* и *saveState()*, которые мы использовали в нашем классе Application.

```
01 TicTacToe::TicTacToe(QWidget *parent, const char *name)
02 : QWidget(parent, name)
03 {
04 clearBoard();
05 if (qApp->isSessionRestored())
06 restoreState();
07 setWindowTitle(tr("Tic-Tac-Toe"));
08 }
```

В конструкторе мы стираем игровое поле и, если приложение было вызвано с опцией — *session*, вызываем закрытую функцию *restoreState()* для восстановления старого сеанса.

```
01 void TicTacToe::clearBoard()
02 {
03 for (int row= 0; row < 3; ++row) {
04 for (int column = 0; column < 3; ++column) {
05 board[row][column] = Empty;
```

```
06 }
07 }
08 turnNumber = 0;
09 }
```

В функции *clearBoard()* мы стираем все ячейки и устанавливаем *turnNumber* на значение 0.

```
01 QString TicTacToe::saveState() const
02 {
03 QFile file(sessionFileName());
04 if (file.open(QIODevice::WriteOnly)) {
05 QTextStream out(&file);
06 for (int row = 0; row < 3; ++row) {
07 for (int column = 0; column < 3; ++column) {
08 out << board[row][column];
09 }
10 }
11 }
12 return file.fileName();
13 }
```

В функции *saveState()* мы записываем состояние игрового поля на диск. Формат достаточно простой: «X» для крестиков, «0» для ноликов и «—» для пустых ячеек.

```
01 QString TicTacToe::sessionFileName() const
02 {
03 return QDir::homePath() + "/.tictactoe_"
04 + qApp->sessionId() + "_" + qApp->sessionKey();
05 }
```

Закрытая 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.

```
01 void TicTacToe::restoreState()
02 {
03 QFile file(sessionFileName());
04 if (file.open(QIODevice::ReadOnly)) {
05 QTextStream in(&file);
06 for (int row = 0; row < 3; ++row) {
07 for (int column = 0; column < 3; ++column) {
08 in >> board[row][column];
09 if (board[row][column] != Empty)
10 ++turnNumber;
11 }
12 }
13 }
14 update();
15 }
```

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 могут 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 выводятся 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/4.1/session.html.

Глава 21. 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 Qtopia Core (ранее она называлась Qt/Embedded) — это версия Qt, оптимизированная для разработки встроенных систем под Linux. Qtopia Core 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.

Qtopia Core может работать на любом оборудовании, функционирующем под управлением Linux (включая архитектуры Intel x86, MIPS, ARM, StrongARM, Motorola 68000 и PowerPC). Эта система имеет буфер фреймов основной памяти, отображаемой на дисплей, и поддерживает компилятор C++. В отличие от Qt/X11, она не нуждается в системе X Window; вместо этого в ней реализуется собственная оконная система (own window system — QWS), которая приводит к значительной экономии постоянной и основной памяти. Для еще большего уменьшения расхода памяти можно перекомпилировать Qtopia Core 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 Qtopia Core.

Кроме того, Qtopia Core использует преимущества многих функций, присущих также версиям Qt для настольных компьютеров, в частности широко применяется неявное совместное использование данных («копирование при записи») как метод экономии основной памяти, поддерживаются пользовательские стили виджетов с помощью класса *QStyle* и обеспечивается система компоновки виджетов, позволяющая максимально использовать пространство экрана.

Qtopia Core 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 Qtopia

Приложения Qtopia Core могут разрабатываться на любой платформе, позволяющей запускать цепочки многоплатформенных 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/. Поскольку Qtopia Core имеет программный интерфейс Qt, в большинстве разработках, как правило, можно использовать версию Qt для настольных компьютеров, например Qt/X11 или Qt/Windows.

Система конфигурации Qtopia Core поддерживает кросс-компиляторы с помощью опции *—embedded* скрипта *configure*. Например, для построения ARM—архитектуры мы могли бы ввести команду

./configure -embedded arm

Можно создавать пользовательские конфигурации путем добавления новых файлов в каталог Qt *mkspecs/qws*.

Qtopia Core рисует непосредственно в буфере фреймов системы Linux (область основной памяти, связанная с дисплеем). Для обращения к буферу фреймов, возможно, потребуется получить разрешение для записи на устройство */dev/fb0*.

Для выполнения приложений Qtopia Core сначала необходимо запустить один процесс, выполняющий функции сервера. Этот сервер отвечает за распределение между клиентами областей экрана и за генерацию событий мышки и клавиатуры. Любое приложение Qtopia Core 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 Qtopia Core при помощи совместно используемой области в основной памяти. Внутренне операции рисования реализованы так, что клиенты рисуют самих себя в совместно используемой области памяти и отвечают за оформление собственных окон. Это сводит к минимуму объем данных, передаваемых между клиентами и сервером, и в результате интерфейс пользователя работает без задержек. Приложения Qtopia Core обычно используют рисовальщик *QPainter* для рисования самих себя, но они могут также получать непосредственный доступ к видеооборудованию, используя класс *QDirectPainter*.

Клиенты могут связываться друг с другом при помощи протокола QCOP. 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 формат данных в сообщении принимает вид функции C++.

На работу приложений Qtopia Core 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/4.1/emb-envvars.html.

Если в качестве платформы разработки используется Unix, приложение можно тестировать с использованием виртуального буфера фреймов Qtopia *(qvfb)* — приложения X11, которое имитирует пиксель за пикселем реальный буфер фреймов. Это значительно сокращает цикл разработки. Для включения поддержки в Qtopia Core 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

Другой опцией, работающей на большинстве платформ, является VNC (Virtual Network Computing — вычисление в виртуальной сети), которая используется для удаленного выполнения приложения. Для включения поддержки VNC в Qtopia Core передайте опцию *—qt—gfx—vnc* в скрипт *configure*. Затем запустите ваше приложение Qtopia Core 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 Qtopia Core (например, *QWS_SIZE=320x480* и *QWS_DEPTH=32).*

Настройка Qtopia Core

При установке Qtopia Core можно указать функции, которые мы хотим устранить, чтобы снизить расход памяти. В состав Qtopia Core входит сотня 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.*

Qtopia Core содержит пять примеров конфигурации *(minimum, small, medium, large* и *dist)*, которые находятся в файлах *src/corelib/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 — использовать графическую утилиту *qconfig*, расположенную в подкаталоге Qt *tools*.

Qtopia Core предоставляет следующие классы для интерфейса с входными и выходными устройствами и для настройки пользовательского интерфейса оконной системы:

- *QScreen — драйверы экрана,*
- *QScreenDriverPlugin — подключаемые модули драйверов экрана,*
- *QWSMouseHandler — драйверы мышки,*
- *QMouseDriverPlugin — подключаемые модули драйверов мышки,*
- *QWSKeyboardHandler — драйверы клавиатуры,*
- *QKbdDriverPlugin — подключаемые модули драйверов клавиатуры,*
- *QWSInputMethod — методы ввода,*
- *QDecoration — стили оформления окон,*
- *QDecorationPlugin — подключаемые модули 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 Qtopia Core, 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 наследует класс *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 Qtopia Core, могут получать доступ к

уникальному экземпляру *QWSServer* через глобальную переменную *qwsServer*, которую инициализирует конструктор *QApplication*.

Qtopia Core поддерживает следующие форматы шрифтов: TrueType (TTF), PostScript Type 1, Bitmap Distribution Format (BDF) и Qt Prerendered 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* позволяет воспринимать файлы TTF или PostScript Type 1 и сохранять результат в формате QPF. Можно поступить по-другому — запустить наши приложения с опцией командной строки *—savefonts*.

На момент написания книги компания «Trolltech» разрабатывает дополнительный уровень, расположенный над Qtopia Core и позволяющий еще быстрее и удобнее разрабатывать приложения для встроенных систем. Можно надеяться, что следующее издание данной книги будет содержать больше информации по этому вопросу.

Приложение А. Установка Qt

В данном приложении рассматривается порядок установки Qt в вашу систему с компакт-диска, который прилагается к этой книге. Компакт-диск содержит версию Qt 4.1.1 для Windows, Mac OS X и X11 (для Linux и большинства версий Unix). Все они включают SQLite — общедоступную, не нуждающуюся в сервере базу данных и драйвер SQLite. Для вашего удобства на компакт-диске представлены различные версии Qt. Для серьезных разработок программного обеспечения лучше всего загрузить из Интернета последнюю версию Qt по адресу http://www.trolltech.com/download/ или приобрести коммерческую версию.

Компания «Trolltech» также обеспечивает Qtopia Core 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 Qtopia Core с соответствующей веб-страницы сайта компании «Trolltech».

Приложения представленных в книге примеров содержатся на компакт-диске в каталоге *examples*. Кроме того, Qt предоставляет много небольших приложений—примеров, которые располагаются в подкаталоге *examples*.

Замечание о лицензировании

Qt выпускается в двух формах: с открытым исходным кодом и коммерческая. Версия с открытым исходным кодом распространяется бесплатно; за коммерческую версию приходится платить.

Представленное на компакт-диске программное обеспечение пригодно для создания приложений для себя или для образовательных целей.

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.

Компакт-диск содержит GPL—версии Qt для Windows, Mac OS X и X11. Полные, юридически верные тексты лицензий включены в пакеты программ компакт-диска; здесь же имеется информация о том, как получить коммерческие версии.

Установка Qt/Windows

Когда вы вставляете компакт-диск в машину с системой Windows, автоматически запускается программа установки. Если этого не происходит, с помощью Проводника попадите в корневой каталог компакт-диска и дважды щелкните по *install.exe*. (Эта программа в Проводнике может выглядеть как *install*, что зависит от конкретной настройки вашей системы.)

Если у вас уже установлен компилятор MinGW C++, необходимо указать каталог его размещения; в противном случае установите соответствующий переключатель для установки компилятора MinGW. Находящаяся на компакт-диске GPL—версия Qt не будет работать с Visual C++, поэтому необходимо установить компилятор MinGW, если он еще у вас не установлен. Программа установки позволяет также установить примеры, прилагаемые к книге. 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 v4.1.1 (opensource)». Эта папка будет содержать ярлыки для *Qt Assistant* и *Qt Designer*, а также «Qt 4.1.1 Command Prompt», который запускает окно консоли. При запуске этого окна выполняется установка переменных среды для компиляции программ Qt с помощью MinGW. Именно 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 (Tiger) и Xcode Tools 2.x (вместе с компилятором GCC 4.0.x), можно воспользоваться установщиком, как это описано ниже. Если вы имеете более старую версию Mac OS X или более старую версию GCC, необходимо вручную установить пакет с исходными текстами. Этот пакет называется *qt—mac—opensource—4.1.1.tar.gz* и располагается в каталоге *mac* компакт-диска. После установки этого пакета следуйте инструкциям по установке Qt в системе X11, которые приводятся в следующем разделе.

Для использования программы установки вставьте компакт-диск и дважды щелкните по пакету *Qt.mpkg*. Это приведет к запуску программы установки *Installer.app*, и Qt будет установлена со стандартными примерами, документацией и примерами, прилагаемыми к данной книге. Qt будет установлена в каталог */Developer*, а примеры книги в */Developer/Examples/Qt4Book*.

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 каталог */Developer/Examples/Qt4Book/chap01/hello* и введите следующую команду:

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

Установка Qt/X11

Для установки Qt в системе X11 в свой стандартный каталог вам могут потребоваться полномочия *root*. Если у вас нет таких полномочий, используйте аргумент *—prefix* скрипта *configure* для указания каталога, в который вам разрешено записывать данные.

1. Перейдите на временный каталог. Например:

cd /tmp

2. Распакуйте архивный файл, расположенный на компакт-диске:

cp /cdrom/x11/qt-x11-opensource-src-4.1.1.tgz

gunzip qt-x11-opensource-src-4.1.1.tgz

tar xf qt-x11-opensource-src-4.1.1.tar

Это создает каталог */tmp/qt—x11—opensource—src—4.1.1* при условии, что ваш компакт-диск смонтирован, как */cdrom.* 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-4.1.1

./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 результате Qt будет установлена в */usr/local/Trolltech/Qt—4.1.1*. 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 строки в ваш файл *.profile:*

PATH=/usr/local/Trolltech/Qt-4.1.1/bin:$PATH

export PATH

Если вы используете оболочку *csh* или *tcsh,* добавьте следующую строку в ваш файл *.login:*

setenv PATH /usr/local/Trolltech/Qt-4.1.1/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—4.1.1/lib*. Это необязательно 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*.

Приложение Б. 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 написания 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++» (Дизайн и эволюция C++), 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 ознакомит 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 30 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 разработчики C++ должны

использовать различные библиотеки (часто зависимые от платформы).

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 рабочую среду, 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.

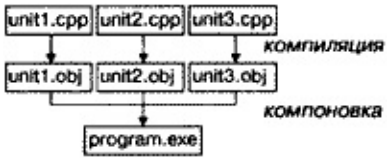


Рис. Б.1. Процесс компиляции программы на C++ (в Windows).

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.

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 этот файл и затем повторно скомпоновать приложение для создания нового исполняемого модуля.

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:*

```
01 double square(double n)
02 {
03 return n * n;
04 }
```

Этот файл содержит лишь глобальную функцию с именем *square(),* которая возвращает квадрат своего параметра.

Ниже показан файл *main.cpp:*

```
01 #include <cstdlib>
02 #include <iostream>
03 using namespace std;
04 double square(double);
05 int main(int argc, char *argv[])
06 {
```

```
07 if (argc != 2) {
08 cerr << "Usage: square <number>" << endl;
09 return 1;
10 }
11 double n = strtod(argv[1], 0);
12 cout << "The square of " << argv[1] << " is " << square(n) << 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++ функции *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.

Директива *using namespace* в строке 3 указывает компилятору на то, что мы хотим импортировать в глобальное пространство имен все идентификаторы, объявленные в пространстве имен *std*. Это позволяет нам пользоваться записью *strtod()*, *cout*, *cerr* и *endl* вместо указания полных имен: *std::strtod()*, *std::cout*, *std::cerr* и *std::endl*. В C++ оператор *::* разделяет компоненты сложного имени.

В строке 4 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
ld 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
```

Эта 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:

```
01 #include <cstdlib>
02 #include <iostream>
03 #include "square.h"
04 using namespace std;
05 int main(int argc, char *argv[])
06 {
07 if (argc != 2) {
08 cerr << "Usage: square <number>" << endl;
09 return 1;
10 }
11 double n = strtod(argv[1], 0);
12 cout << "The square of " << argv[1] << " is " << square(n) << 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, в то время как 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 получили бы 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 они компонуются 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» (Язык программирования C++) , написанной Бьерном Страуструпом, или «C++ for Java Programmers» (C++ для программистов Java), 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. Б.2 приводятся список элементарных типов C++ и их определение на платформах, поддерживаемых Qt 4:

- *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* [9] — 64-битовый целый тип,
- *float* — 32-битовое значение числа с плавающей точкой (IEEE 754),
- *double* — 64 битовое значение числа с плавающей точкой (IEEE 754).

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 *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 как символов *ASCII*, так 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.*

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 —209 486 515. 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 («Ввод—вывод»), 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 отличия на нескольких примерах. Начнем с класса, представляющего пару координат *(x, y):*

```
01 #ifndef POINT2D_H
02 #define POINT2D_H
03 class Point2D
04 {
05 public:
06 Point2D() {
07 xVal = 0;
08 yVal = 0;
09 }
10 Point2D(double x, double y) {
11 xVal = x;
12 yVal = y;
13 }
14 void setX(double x) { xVal = x; }
15 void setY(double y) { yVal = y; }
16 double x() const { return xVal; }
17 double y() const { return yVal; }
18 private:
19 double xVal;
20 double yVal;
21 };
22 #endif
```

Представленное выше определение класса обычно оформляется в виде заголовочного файла, типичным названием которого может быть *point2d.h*. В этом примере проявляются следующие характерные особенности C++:

• 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:

```
01 #ifndef POINT2D_H
02 #define POINT2D_H
03 class Point2D
04 {
05 public:
06 Point2D();
07 Point2D(double x, double y);
08 void setX(double x);
09 void setY(double y);
10 double x() const;
11 double y() const;
12 private:
13 double xVal;
14 double yVal;
15 };
16 #endif
```

Реализация функций выполнялась бы в файле *point2d.cpp:*

```
01 #include "point2d.h"
02 Point2D::Point2D()
03 {
04 xVal = 0.0;
05 yVal = 0.0;
06 }
07 Point2D::Point2D(double x, double y)
08 {
09 xVal = x;
10 yVal = y;
11 }
12 void Point2D::setX(double x)
13 {
14 xVal = x;
15 }
16 void Point2D::setY(double y)
17 {
18 yVal = y;
19 }
20 double Point2D::x() const
21 {
22 return xVal;
23 }
24 double Point2D::y() const
25 {
26 return yVal;
27 }
```

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.

```
01 #include "point2d.h"
02 int main()
03 {
04 Point2D alpha;
05 Point2D beta(0.666, 0.875);
06 alpha.setX(beta.y());
07 beta.setY(alpha.x());
08 return 0;
09 }
```

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:

```
01 #ifndef SHAPE_H
02 #define SHAPE_H
03 #include "point2d.h"
04 class Shape
05 {
06 public:
07 Shape(Point2D center) { myCenter = center; }
08 virtual void draw() = 0;
09 protected:
10 Point2D myCenter;
```

```
11 };
12 #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 обобщенный класс *Object*, который наследуется всеми другими классами. 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:*

```
01 #ifndef CIRCLE_H
02 #define CIRCLE_H
03 #include "shape.h"
04 class Circle : public Shape
05 {
06 public:
07 Circle(Point2D center, double radius = 0.5)
08 : Shape(center) {
09 myRadius = radius;
10 }
11 void draw() {
12 // здесь выполняются какие-то действия
13 }
14 private:
15 double myRadius;
16 };
17 #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:

```
01 class LabeledCircle : public Circle
02 {
03 public:
04 void draw() {
05 Circle::draw();
06 drawLabel();
07 }
08 };
```

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:

```
01 #ifndef TRUCK_H
02 #define TRUCK_H
03 class Truck
04 {
05 public:
06 Truck() { ++counter; }
07 ~Truck() { --counter; }
08 static int instanceCount() { return counter; }
09 private:
10 static int counter;
11 };
12 #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 ~, 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:

```
01 #include <iostream>
02 #include "truck.h"
03 using namespace std;
04 int main()
05 {
06 Truck truck1;
07 Truck truck2;
08 cout << Truck::instanceCount() << " equals 2" << endl;
09 return 0;
10 }
```

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:

```
01 #include "point2d.h"
02 int main()
03 {
04 Point2D alpha;
05 Point2D beta;
06 Point2D *ptr;
07 ptr = &alpha;
08 ptr->setX(1.0);
09 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:*

```
01 #include "point2d.h"
02 int main()
03 {
04 Point2D *point = new Point2D;
05 point->setX(1.0);
06 point->setY(2.5);
07 delete point;
08 return 0;
09 }
```

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 становятся больше ненужными. 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);
```

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 обнаруживать 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 со 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;
cout << *p << " equals 10" << endl;
cout << *q << " equals 20" << endl;
*p = 40;
cout << i << " equals 40" << endl;
p = q;
*p = 100;
cout << i << " equals 40" << endl;
cout << j << " equals 100" << 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. Основными отличиями являются следующие:

• 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. По умолчанию 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>
using namespace std;
double manhattanDistance(Point2D a, Point2D b)
{
return abs(b.x() - a.x()) + abs(b.y() - a.y());
}
```

Эта функция может вызываться следующим образом:

```
Point2D harlem(77.5, 50.0);
Point2D broadway(12.5, 40.0);
double distance = manhattanDistance(broadway, harlem);
```

Опытные С—программисты избегают операций копирования путем объявления параметров в виде указателей вместо значений:

```
double manhattanDistance(const Point2D *ap, const Point2D *bp)
{
return abs(bp->x() - ap->x()) + abs(bp->y() - ap->y());
}
```

После этого при вызове функции должны передаваться адреса вместо значений:

```
Point2D harlem(77.5, 50.0);
Point2D broadway(12.5, 40.0);
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 abs(b.x() - a.x()) + 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* н а *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 что 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 10 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)
cout << fibonacci[i] << endl;
```

Массив можно также проходить с помощью указателя:

```
const int *ptr = &fibonacci[0];
while (ptr != &fibonacci[10]) {
cout << *ptr << 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)* и л и *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:

```
01 #include <iostream>
02 using namespace std;
03 void printIntegerTable(const int *table, int size)
04 {
05 for (int i = 0; i < size; ++i)
06 cout << table[i] << endl;
07 }
08 int main()
09 {
10 const int fibonacci[10] = { 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 };
11 printIntegerTable(fibonacci, 10);
12 return 0;
13 }
```

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 *std::memcpy()*, которая копирует блок памяти. Например:

```
memcpy(temp, fibonacci, sizeof(fibonacci));
```

При объявлении массива C++ его размер должен быть константой[10]. Если необходимо создать массив переменного размера, это можно сделать несколькими способами:

• Выделять память под массив можно динамически:

```
int *fibonacci = new int[n];
```

Оператор *new []* выделяет последовательные участки памяти под определенное количество элементов и возвращает указатель на первый элемент. Благодаря принципу

«эквивалентности указателей и массивов» обращаться к элементам можно с помощью указателей: *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 *std::vector<T>:*

```
#include <vector>
using namespace std;
vector<int> fibonacci(n);
```

Обращаться к элементам можно с помощью оператора *[]*, как это делается для обычного массива C++. При использовании вектора *std::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 *std::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 *std::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:

```
01 void hello1()
02 {
03 const char str[] = {
04 'H', 'e', 'l', 'l', 'o', ' ', 'w', 'o', 'r' 'l', 'd', '\0'
05 };
06 cout << str << endl;
07 }
08 void hello2()
09 {
10 const char str[] = "Hello world!";
11 cout << str << endl;
12 }
13 void hello3()
14 {
15 cout << "Hello world!" << endl;
16 }
17 void hello4()
18 {
19 const char *str = "Hello world!";
20 cout << str << endl;
21 }
```

В первой функции строка объявляется как массив и инициализируется посимвольно. 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:

```
01 #include <cctype>
02 #include <iostream>
```

```
03 using namespace std;
04 void makeUppercase(char *str)
05 {
06 for (int i = 0; str[i] != '\0'; ++i)
07 str[i] = toupper(str[i]);
08 }
09 void writeLine(const char *str)
10 {
11 cout << str << endl;
12 }
13 int main(int argc, char *argv[])
14 {
15 for (int i = 1; i < argc; ++i) {
16 makeUppercase(argv[i]);
17 writeLine(argv[i]);
18 }
19 return 0;
20 }
```

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 («Классы—контейнеры») и в главе 17 («Интернационализация»).

Перечисления

C++ позволяет с помощью перечисления объявлять набор поименованных констант аналогично тому, как это делается в C#. 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 (wilcardSyntaxCheckBox->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 онлайновой 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:

- **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);
cout << x << " equals 314" << 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)
cout << "The byte order is big-endian" << 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)
cout << "The byte order is big-endian" << 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 наследником класса *A*. Например:

```
A *obj = new B;
B *b = static_cast<B *>(obj);
b->someFunctionDeclaredInB();
```

Если объект не является экземпляром *B* (но все же наследует *A)*, 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 («Реализация функциональности приложения»).

• *reinterpret_cast<T>()* преобразует любой тип указателя или ссылки в любой другой их тип. Например:

```
short j = 0x1234;
if (reinterpret_cast<char &>(j) == 0x12)
cout << "The byte order is big-endian" << 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 использованием.

C++ обеспечивает много способов приведения типов, однако в большинстве случаев это даже не приходится делать. При использовании таких классов—контейнеров, как *std::vector<T>* и *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:*

```
01 #ifndef POINT2D_H
02 #define POINT2D_H
03 class Point2D
04 {
05 public:
06 Point2D();
07 Point2D(double x, double y);
08 void setX(double x);
09 void setY(double y);
10 double x() const;
11 double y() const;
12 Point2D &operator+=(const Point2D &other)
13 {
14 xVal += other.xVal;
15 yVal += other.yVal;
16 return *this;
17 }
18 Point2D &operator-=(const Point2D &other)
19 {
20 xVal -= other.xVal;
21 yVal -= other.yVal;
22 return *this;
23 }
24 private:
25 double xVal;
26 double yVal;
```

```
27 };

28 inline Point2D operator+(const Point2D &a, const Point2D &b)
29 {
30 return Point2D(a.x() + b.x(), a.y() + b.y());
31 }
32 inline Point2D operator-(const Point2D &a, const Point2D &b)
33 {
34 return Point2D(a.x() - b.x(), a.y() - b.y());
35 }
36 #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 beta(77.5, 50.0);
Point2D alpha(12.5, 40.0);
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 beta(77.5, 50.0);
Point2D alpha(12.5, 40.0);
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 копирование члена в член. Для класса Point2D это равносильно тому, как если бы мы написали следующий программный код в определении класса:

```
01 class Point2D
02 {
03 public:
04 Point2D(const Point2D &other)
05 : xVal(other.xVal), yVal(other.yVal) { }
06 Point2D &operator=(const Point2D &other)
07 {
08 xVal = other.xVal;
09 yVal = other.yVal;
10 return *this;
11 }
12 …
13 private:
14 double xVal;
15 double yVal;
16 };
```

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 будет 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 никакому классу и к которым можно обращаться из любой другой функции. Мы видели несколько примеров глобальных функций, в частности *main()* — точка 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:*

```
01 int randomNumbers[128];
02 static int seed = 42;
03 static int nextRandomNumber()
04 {
05 seed = 1009 + (seed * 2011);
06 return seed;
07 }
08 void populateRandomArray()
09 {
10 for (int i = 0; i < 128; ++i)
11 randomNumbers[i] = nextRandomNumber();
12 }
```

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:

```
01 #include <iostream>
02 using namespace std;
03 extern int randomNumbers[128];
04 void populateRandomArray();
```

```
05 int main()
06 {
07 populateRandomArray();
08 for (int i = 0; i < 128; ++i)
09 cout << randomNumbers[i] << endl;
10 return 0;
11 }
```

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:

```
01 #ifndef RANDOM_H
02 #define RANDOM_H
03 extern int randomNumbers[128];
04 void populateRandomArray();
05 #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:

```
01 void nextPrime()
02 {
03 static int n = 1;
04 do {
05 ++n;
06 } while (!isPrime(n));
07 return n;
08 }
```

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:

```
01 #ifndef SOFTWAREINC_RANDOM_H
02 #define SOFTWAREINC_RANDOM_H
03 namespace SoftwareInc
04 {
05 extern int randomNumbers[128];
06 void populateRandomArray();
07 }
08 #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:*

```
01 #include "random.h"
02 int SoftwareInc::randomNumbers[128];
03 static int seed = 42;
04 static int nextRandomNumber()
05 {
06 seed = 1009 + (seed * 2011);
07 return seed;
08 }
09 void SoftwareInc::populateRandomArray()
10 {
11 for (int i = 0; i < 128; ++i)
12 randomNumbers[i] = nextRandomNumber();
13 }
```

В отличие от классов, пространства имен можно «повторно открывать» в любое время. Например:

```
01 namespace Alpha
02 {
03 void alpha1();
04 void alpha2();
05 }
06 namespace Beta
```

```
07 {
08 void beta1();
09 }
10 namespace Alpha
11 {
12 void alpha3();
13 }
```

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 — 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, расположенном 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 неопределенным, 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* пропускается.

• 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. Б.3 приводится список базовых заголовочных файлов C++:

• <*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.

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 *QBitArray*, *QLocale*, *QString* и *QMetaObject*.

Стандартный C++ также включает ряд заголовочных файлов, обеспечивающих ввод—вывод (см. рис. Б.4):

• <*fstream*> — шаблонные классы, манипулирующие внешними файлами,

• <*iomanip*> — манипуляторы потоков ввода—вывода, принимающие один аргумент,

• <*ios*> — шаблонный базовый класс потоков ввода—вывода,

• <*iosfwd*> — предварительные объявления нескольких шаблонных классов потоков ввода—вывода,

• <*iostream*> — стандартные потоки ввода—вывода *(cin*, *cout*, *cerr*, *ctog)*,

• <*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.

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 («Ввод—вывод») 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.

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. Б.5 приводится список заголовочных файлов библиотеки STL:

- <*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 недопускающие повторения элементов,
- <*stack*> — шаблонный контейнер стека,
- <*utility*> — основные шаблонные функции,
- <*vector*> — шаблонный контейнер вектора.

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.

На рис. Б.6 приводится список заголовочных файлов библиотеки C:

- <*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,
- <*clocale*> — функции и типы, относящиеся к локализации,
- <*cmath*> — математические функции и константы,
- <*csetjmp*> — функции для выполнения нелокальных переходов,
- <*csignal*> — функции для обработки системных сигналов,
- <*cstdarg*> — макросы для реализации функций с переменным числом аргументов,
- <*cstddef*> — определения, общие для некоторых стандартных заголовочных файлов,
- <*cstdio*> — функции ввода—вывода,
- <*cstdlib*> — общие вспомогательные функции,
- <*cstring*> — функции для манипулирования массивами *char*,
- <*ctime*> — типы и функции для манипулирования временем,
- <*cwchar*> — утилиты для работы с многобайтовыми символами и символами расширенной кодировки,
- <*cwctype*> — функции классификации и отображения символов расширенной кодировки.

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()*.

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.

0

От составителя: поскольку библиотека Qt в 2008м была куплена Nokia, адреса теперь не "www.trolltech.com", а "qt.nokia.com". Всё остальное — такое же :).

note_0

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 новую версию Qt 4.

note_1

2

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

note_2

3

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*. — Прим. автора.
note_3

4

Если вы находитесь в стране, которая признает патенты на программное обеспечение и где компания Unisys обладает патентом на LZW—распаковку, вам, возможно, потребуется лицензия от компании Unisys на право применения ее технологии работы с файлами GIF. По этой причине поддержка файлов GIF по умолчанию отключена в Qt. Мы надеемся, что срок действия ограничений этой лицензии истечет к концу 2004 года. GIF support is disabled in Qt by default because the decompression algorithm used by GIF files was patented in some countries where software patents are recognized. We believe that this patent has now expired worldwide. To enable GIF support in Qt, pass the *—qt—gif* command—line option to the configure script or set the appropriate option in the Qt installer.

note_4

5

Подчеркнутый символ в пункте меню означает, что этот пункт можно выбрать в активном меню, нажав клавиатурную комбинацию «Alt» + «подчеркнутый символ». — *Примеч. науч. ред.*

note_5

6

Используемый здесь удобный синтаксис *qDebug() << arg* требует включения заголовочного файла *<QtDebug>*, в то время как синтаксис *qDebug("...", arg)* доступен в любом файле, который включает по крайней мере один заголовочный файл Qt.

note_6

Ключевое слою *forever* обеспечивается Qt. Оно просто разворачивается в оператор *for (;;)*.

note_7

8

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).
note_8

9

Компания Microsoft вместо *long long* использует тип *__int64*. В программах Qt доступен тип *qlonglong* в качестве альтернативы, работающей на всех платформах Qt.

note_9

10

Некоторые компиляторы позволяют использовать также переменные, однако нельзя на это рассчитывать в переносимых программах.

note_10

FB2 document info

Document ID: 7cd89ad4-62a1-4054-aadb-d74564f19f4f
Document version: 1
Document creation date: 11 Novemver 2009
Created using: Abbyy FineReader, FB Editor v2.0, AlReader2 software

Document authors :

- graphist

Document history:

?.? — DjVu на lib.rus.ec 1.0 — преобразование в fb2, вычитка

About

This book was generated by Lord KiRon's FB2EPUB converter version 1.0.28.0.

Эта книга создана при помощи конвертера FB2EPUB версии 1.0.28.0 написанного Lord KiRon