Разработка графического интерфейса с помощью библиотеки Qt3

Авторы: Jasmin Blanchette, Mark Summerfield
Перевод: Андрей Киселёв (kis_an [at] mail.ru)

Оригинальная версия была опубликована издательством "Prentice Hall PTR". Вы сможете найти ее по адресу: http://www.phptr.com/conteimages/0131240722/downloads/blanchette_book.pdf .

Данная книга распространяется на условиях Open Publication License, v1.0 или более поздней. Полный текст лицензии вы найдете по адресу: http://www.opencontent.org/openpub/.

"Trolltech" и "Qt" -- зарегистрированные торговые марки компании Trolltech.

OpenGL -- торговая марка Silicon Graphics, Inc.

Все остальные имена компаний и названия программных продуктов, упомянутые здесь, являются торговыми марками их соответствующих владельцев.

Авторы, держатели авторских прав и издатель приложили немало усилий при подготовке этой книги, но не предоставляют никаких явных или подразумеваемых гарантий любого вида и не принимают на себя ответственность за возможные ошибки или упущения. Сведения, приводимые в этой книге, носят исключительно информационный характер и могут быть изменены без дополнительного уведомления. Ни авторы, ни издатель, ни держатели авторских прав не несут никакой ответственности за непредвиденные убытки, прямо или косвенно связанные с использованием информации или программ, содержащихся здесь.

Содержание

Предисловие ... 5
Благодарности ... 7
Часть 1 - Основы Qt ... 10
1 Начало ... 10
1.1 Hello, Qt! ... 10
1.2 Обработка сигналов ... 11
1.3 Работа со справочной системой ... 14
1.3.1 Стили виджетов ... 14
2 Создание диалогов ... 16
2.1 Создание дочернего класса от QDialog ... 16
2.2 Сигналы и слоты ... 21
2.2.1 Метаобъектная Система в библиотеке Qt ... 22
2.3 Быстрая разработка диалогов ... 23
2.4 Диалоги с изменяющимся внешним видом. ... 28
2.5 Динамические диалоги ... 32
2.6 Встроенные виджеты и классы диалогов ... 33
3 Создание главного окна приложения ... 37
3.1 Создание класса-наследника от QMainWindow ... 37
3.2 Создание меню и панелей инструментов ... 40
3.3 Реализация меню "File" ... 44
3.4 Настройка строки состояния ... 49
3.5 Использование диалогов ... 51
3.6 Сохранение пользовательских настроек приложения ... 55
3.7 Работа с несколькими документами одновременно ... 56
3.8 Экран-заставка ... 58
4 Реализация функциональности приложения ... 60
4.1 Центральный виджет ... 60
4.2 Создание класса-потомка от QTable ... 60
4.2.1 Хранение данных в виде отдельных объектов ... 63
4.3 Загрузка и сохранение ... 66
4.4 Реализация меню Edit ... 68
4.5 Реализация других меню ... 72
4.6 Создание дочернего класса от QTableItem ... 74
5 Создание собственных виджетов ... 81
5.1 Переделка существующих виджетов Qt ... 81
5.2 Создание класса-потомка от QWidget ... 82
5.3 Интеграция виджета в Qt Designer ... 89
5.4 Двойная буферизация ... 92
5.4.1 Использование логической операции NOT (НЕ), при рисовании рамки выделенной области ... 104
Часть 2 - Углубленные сведения ... 109
6 Управление размещением виджетов ... 109
6.1 Основы компоновки виджетов ... 109
6.2 Разделители ... 113
6.3 Многостраничные виджеты ... 115
6.4 Области просмотра с прокруткой ... 117
6.5 Стыкуемые окна ... 121
6.6 Многодокументный интерфейс ... 122
7 Обработка событий ... 130
7.1 Обработчики событий ... 130
7.2 Установка фильтров событий ... 134
7.3 Сокращение времени отклика при длительной обработке данных ... 136
8 Двух- и трехмерная графика ... 139
8.1 Рисование средствами QPainter ... 139
8.2 Рисование средствами QCanvas ... 147
8.3 Вывод на печать ... 157

8.4 Графика OpenGL 165
9 Drag and Drop 170
9.1 Реализация механизма 'drag and drop' в приложениях 170
9.2 Поддержка нестандартных типов данных при перетаскивании 174
9.3 Расширенные возможности буфера обмена 177
10 Ввод/вывод 179
10.1 Чтение и запись двоичных данных 179
10.2 Чтение и запись текста 184
10.3 Работа с файлами и каталогами 187
10.4 Взаимодействия между процессами 188
11 Контейнерные классы 191
11.1 Векторы 191
11.2 Списки 194
11.3 Словари (map) 195
11.4 Контейнеры указателей 197
11.5 Классы QString и QVariant 199
11.5.1 Принцип Действия Неявного Совместного Использования Данных 203
12 Базы данных 205
12.1 Установление соединения и выполнение запроса 205
12.2 Представление данных в табличной форме 209
12.3 Разработка форм, ориентированных на работу с базами данных 216
13 Работа с сетью 223
13.1 Класс QFtp 223
13.2 Класс QHttp 227
13.3 Класс QSocket 229
13.4 Протокол UDP и класс QSocketDevice 237
14 XML 241
14.1 Чтение XML-документов с помощью SAX 241
14.2 Чтение XML-документов с помощью DOM 244
14.3 Запись в XML-документы 247
15 Интернационализация 250
15.1 Unicode 250
15.2 Разработка приложений, подготовленных к переводу 253
15.3 Динамическое переключение языков 257
15.4 Перевод существующих приложений 261
16 Разработка справочной системы приложения 264
16.1 Всплывающие подсказки и справка "What's This?" 264
16.2 Использование QTextBrowser для отображения текста справки 266
16.3 Использование Qt Assistant для отображения текста справки 270
17 Многопоточность 271
17.1 Потоки 271
17.2 Взаимодействие с главным потоком приложения 278
17.3 Работа с классами Qt вне главного потока 282
18 Платформо-зависимые особенности 285
18.1 Взаимодействие с API операционной системы 285
18.2 ActiveX 287
18.3 Управление сеансами 297
Об авторах 303
Примечания 304

ВСТУПИТЕЛЬНОЕ СЛОВО

Почему Qt? Почему многие программисты выбирают ее? Ответы на эти вопросы вполне очевидны: Qt -- это единая, сохраняющая совместимость на уровне исходного кода, библиотека. Ее особенность -- богатство возможностей. Ее производительность обеспечивается языком программирования C++. Она доступна в исходных текстах. Она сопровождается хорошо проработанной документацией. Разработчики предоставляют высококачественную техническую поддержку. И многое, многое другое, что вы сможете прочесть в глянцевых проспектах от Trolltech. Это все хорошо, но тут упущен один важный момент: Qt пользуется успехом потому, что она НРАВИТСЯ программистам.

Как так получается, что программистам нравятся одни технологии и не нравятся другие? Лично я полагаю, что инженеру - программисту нравится тот продукт, который несет в себе ощущение правильности, законченности, некоей внутренней красоты и не нравится тот, в котором чувствуются изъяны. Как еще можно объяснить тот факт, что некоторые из ярчайших программистов не в состоянии без посторонней помощи запрограммировать видеомагнитофон? Или что многие инженеры испытывают затруднения, когда сталкиваются лицом к лицу с телефонной системой компании? Я без особых проблем запоминаю длинные последовательности случайных чисел и команд, но когда дело доходит до управления автоответчиком -- я предпочитаю держаться в стороне. Телефонная система нашей компании требует, чтобы клавиша <*> удерживалась не менее 2-х секунд, прежде чем можно будет набирать добавочный код. Если вы забудете это обстоятельство и продолжите набор добавочного кода без выполнения задержки, то вам придется набирать номер с самого начала. Но почему <*>? Почему не <#>, не <1> или <5>? Почему не любая другая из 12-ти клавиш? Почему именно 2 секунды? Почему не 1, не 3 или не 1.5? Почему именно так, а не иначе? Телефон настолько меня раздражает, что я предпочитаю пользоваться им только в самых крайних случаях, когда не позвонить просто невозможно.

Программирование во многом похоже на нашу телефонную систему, только во много раз хуже. И тут к нам на помощь приходит Qt. Она разнолика. С одной стороны она наполнена глубочайшим смыслом. С другой -- полна забавных моментов. Qt позволяет полностью сконцентрироваться на решении задач. Когда разработчики библиотеки сталкивались с какой-либо проблемой, они не просто находили хорошее, быстрое или простое решение -- они находили в первую очередь правильное решение и затем подробно его документировали. Найденные архитектурные решения прошли длительное испытание временем. Тем не менее в библиотеке все еще существуют "узкие" места, но они могут быть и будут исправлены.

Задолго до того как Qt достигла пика своей популярности, разработчики, посвятившие себя этой библиотеке, сделали из нее нечто особенное. Эта преданность не угасла до сих пор и живет во всех, кто так или иначе связан с этой библиотекой. Для нас, работа над Qt -- это привилегия и большая ответственность. Мы гордимся возможностью сделать вашу профессиональную жизнь проще и приятнее.

Библиотека Qt поставляется с замечательным комплектом документации. Но она в основном концентрируется на описании отдельных классов и лишь вскользь касается темы разработки сложных приложений. Данная книга заполняет этот пробел. Здесь рассказывается о том, что может предложить вам Qt, как с ней работать и как получить от нее максимум отдачи. Книга сопровождается большим количеством примеров, советов и подробных описаний.

В настоящее время, на базе Qt, разработано гигантское количество приложений, как коммерческих, так и свободно-распространяемых. Одни из них ориентированы на узкоспециализированный рынок, другие предназначены для массового использования. Такая популярность наполняет нас гордостью и вдохновляет на поиски новых решений, которые сделают нашу библиотеку еще лучше. А с помощью этой книги появится еще больше высококачественных приложений, написанных с использованием библиотеки Qt.

Маттиас Эттрич, Осло, Норвегия, Ноябрь 2003

ПРЕДИСЛОВИЕ

Qt -- это библиотека классов C++ и набор инструментального программного обеспечения, предназначенных для построения многоплатформенных приложений с графическим интерфейсом и исповедующих принцип "написав однажды -- компилируй в любом месте". Qt представляет собой единую платформу для приложений, которые могут работать под управлением Windows 95/98/Me/2000/XP, Mac OS X, Linux, Solaris, HP-UX и других версий Unix.

Цель данной книги -- научить вас писать программы, с графическим интерфейсом, основываясь на Qt 3. Обучение начинается с простенькой программы "Hello, Qt!" и быстро переходит к описанию расширенных возможностей библиотеки, таких как -- создание собственных визуальных компонентов (widgets) и использование технологии "перетащил и бросил" (drag and drop).

Книга концентрируется на описании стиля и техники программирования в Qt 3, вместо того, чтобы просто изложить другими словами документацию, поставляемую разработчиками. Кроме того, поскольку мы принимаем участие в разработке Qt 4, мы постарались преподнести материал таким образом, чтобы полученные вами сведения не потеряли свою актуальность и после выхода в свет Qt 4.

В своем повествовании мы предполагаем, что вы уже знакомы с языком программирования C++. Примеры программ, которые вы здесь встретите, написаны на C++. Причем мы использовали ограниченный круг особенностей этого языка -- только то, что действительно необходимо для того, чтобы начать работать с библиотекой. В тех местах, где использование более сложных конструкций языка C++ неизбежно, мы будем давать довольно подробное описание.

Qt завоевала репутацию мультиплатформенного набора инструментальных средств, однако, не смотря на это, чаще всего она используется для разработки приложений на какой-либо одной платформе. В качестве примера приложения, написанного с помощью Qt и получившего массовое распространение, можно привести Adobe Photoshop Album. На базе Qt построено огромное количество узкоспециализированног программного обеспечения. Сюда можно отнести программы, разработанные для создания 3D-анимации, цифровой обработки видеоизображений, автоматизации разработки электронных компонентов (микросхем), для геологических исследований, для работы в области медицины и многие многие другие. Если вы зарабатываете себе на жизнь разработкой программ для платформы Windows, то вы с легкостью сможете расширить круг своих потребителей, за счет Mac OS X и Linux, просто пересобрав свои приложения под эти платформы.

Qt распространяется на основе нескольких лицензий. Если вы предполагаете создавать программы на коммерческой основе, то вы должны приобрести коммерческую лицензию и коммерческую версию Qt. Если вы разрабатываете программы с открытым исходным кодом, то вы должны использовать некоммерческую версию библиотеки. Qt является основой, на которой построен KDE (K Desktop Environment) и множество других программных продуктов с открытыми исходными текстами.

Кроме библиотеки из сотен классов мы предоставляем дополнительные компоненты, которые расширяют возможности библиотеки. Некоторые из них, такие как: модуль интеграции Qt/Motif и Qt Script for Applications (QSA), предлагаются компанией Trolltech, другие -- третьими фирмами и сообществом open source. За информацией, по дополнениям и расширениям, обращайтесь по адресу: http://www.trolltech.com/products/3rdparty/. Кроме того, Qt имеет свое собственное сообщество пользователей, которые обмениваются между собой информацией через списки рассылки. Перечень списков рассылки вы найдете здесь: http://lists.trolltech.com/.

Книга поделена на две больших части. Часть I охватывает теоретические и практические сведения, необходимые для разработки приложений с графическим интерфейсом в среде Qt 3. Этих сведений будет вполне достаточно для написания несложных программ. Часть II дает более углубленный материал. Главы в этой части могут читаться в любом порядке, но требуют предварительного ознакомления с первой частью книги.

Если вы встретите ошибки в тексте или у вас появятся предложения, по-поводу будущих редакций этой книги, то мы будем рады прочитать ваши сообщения. Направляйте свои письма по адресам: <jasmin.blanchette@trolltech.com> и <mark.summerfield@trolltech.com>. Список обнаруженных опечаток будет размещен по адресу:
http://vig.prenhall.com/catalog/academic/product/0,4096,0131240722,00.html.

БЛАГОДАРНОСТИ

Прежде всего мы хотели бы поблагодарить президента компании Trolltech -- Эрика Чамбенга (Eirik Chambe-Eng). Эрик не только с большим энтузиазмом отнесся к нашему желанию написать книгу, но и позволил уделить этому огромное количество времени. Эрик и главный управляющий Trolltech -- Хаавард Норд (Haavard Nord) взяли на себя труд прочитать рукопись и дали весьма ценные замечания. Их великодушие и дальновидность в немалой степени зависели от усилий Матиаса Эттрича (Matthias Ettrich) -- ведущего разработчика Trolltech и нашего босса. Матиас сквозь пальцы смотрел на наше пренебрежение трудовым распорядком, поскольку нас целиком захватила работа над книгой, и дал немало советов по стилю программирования в Qt.

В качестве независимых рецензентов мы привлекли Пауля Кёртиса (Paul Curtis) и Клауса Шмидинджера (Klaus Schmidinger) -- оба являются прекрасными экспертами в Qt. Они с удивительным вниманием к техническим подробностям прочитали нашу книгу, подметили и исправили ряд трудно уловимых ошибок. А так же внесли свои предложения по улучшению содержимого.

Самым неподкупным нашим рецензентом был Реджинальд Стадльбайер (Reginald Stadlbauer). [1] Его технические познания стали для нас неоценимым источником информации. Он показал нам ряд возможностей Qt, о существовании которых мы даже не подозревали.

Также, в рецензировании книги принимали участие Трентон Шульц (Trenton Schulz), Энди Шоу (Andy Shaw) и Андреас Аардал Ханссен (Andreas Aardal Hanssen). Трентон и Энди дали массу советов по всем аспектам книги и особенно по вопросам, касавшимся Qt/Mac и Qt/Windows. Андреас оказал неоценимую услугу при подготовке первой части книги.

Немалую помощь мы получили от Уорвика Аллисона (Warwick Allison) (2D графика), Эрика Чамбенга (история развития Qt), Матиаса Эттрича (обработка событий и разработка собственных визуальных компонентов), Харальда Ферненгеля (Harald Fernengel) (базы данных), Волкера Хильшеймера (Volker Hilsheimer) (ActiveX), Брэдли Хьюса (Bradley Hughes) (многопоточность), Тронда Кьярнесена (Trond Kjernesen) (3D графика и базы данных), Ларса Кнолла (Lars Knoll) (2D графика), Сэма Магнусона (Sam Magnuson) (qmake), Димитрия Пападопулоса (Dimitri Papadopoulos) (Qt/X11), Пауля Олава Твета (Paul Olav Tvete) (разработка собственных визуальных компонентов и Qt/Embedded), Райнера Шмида (Rainer Schmid) (сети и XML) и Гуннара Слетта (Gunnar Sletta) (обработка событий).

Выражаем особую благодарность команде технической поддержки и системным администраторам Trolltech, которые обеспечивали бесперебойную работу наших компьютеров и сетей.
Краткая история развития Qt.

Первый выход в свет библиотеки Qt состоялся в мае 1995 года. Первоначально она разрабатывалась Хаавардом Нордом (главный управляющий Trolltech) и Эриком Чамбенгом (президент компании Trolltech). Хаавард и Эрик встретились в стенах Норвежского Технологического Института, в городе Тронхейме, где они получали высшее образование.

Хаавард начал интересоваться проблемами создания графического интерфейса на C++ с 1988 года. Тогда он получил от Шведской компании заказ на разработку библиотеки, средствами которой можно было бы реализовать графический интерфейс приложений. Спустя пару лет, летом 1990 года, Хаавард и Эрик начали совместную работу над приложением баз данных, которое обрабатывало снимки, получаемые с аппарата ультразвукового обследования. Система должна была иметь возможность работы через графический интерфейс с пользователем, под управлением операционных систем Unix, Macintosh и Windows. Однажды, Хаавард и Эрик вышли на улицу, чтобы подышать свежим воздухом и насладиться летним солнцем. Они присели на скамейку в парке и Хаавард сказал: "Нам нужна объектно-ориентированная система отображения информации". В результате обсуждения была заложена основа, для создания объектно-ориентированной, мультиплатформенной библиотеки, к разработке которой они должны были вскоре приступить.

В 1991 году Хаавард написал первые несколько классов, из которых потом и появилась Qt. Эрик занялся разработкой общего дизайна библиотеки. На следующий год Эрику пришла в голову идея реализации "сигналов и слотов" -- простой но очень мощной парадигмы программирования графического интерфейса. Хаавард подхватил идею и воплотил ее в код. К 1993 году они завершили разработку первого графического ядра и приступили к созданию визуальных компонентов (widgets). В конце года Хаавард предложил Эрику открыть совместное дело и выпустить "лучшую в мире библиотеку реализации графического интерфейса на C++".

1994 год, для двух молодых программистов, начался неудачно. У них не было ни заказов, ни готового продукта, ни денег. К счастью, их жены имели работу и готовы были поддержать своих супругов в течение двух лет, которые требовались на доведение библиотеки до того состояния, в котором она могла бы приносить доход.

В качестве префикса, в именах классов, был выбран символ "Q", поскольку Хааварду очень нравилось как он выглядел в Emacs. Символ "t" был выбран потому, что с него начиналось слово "toolkit", по аналогии с "Xt" -- "X toolkit". Компания была зарегистрирована 4 марта 1994 года под названием "Quasar Technologies", которое затем было преобразовано в "Troll Tech", а затем и в "Trolltech".

В апреле 1995 года, благодаря содействию профессора, у которого обучался Хаавард, Норвежская компания Metis заключила с ними контракт на разработку программного обеспечения на базе Qt. Примерно в то же время Trolltech нанял Арнта Гульбрандсена (Arnt Gulbrandsen) [2] , который изобрел и воплотил в жизнь справочную систему Qt. Кроме того он внес существенный вклад в разработку библиотеки.

20 мая 1995 года, Qt 0.90 была выложена на sunsite.unc.edu. Шесть дней спустя, выход библиотеки был анонсирован на comp.os.linux.announce. Это был первый публичный выпуск. Qt тогда могла работать как под управлением Windows, так и под управлением Unix, предоставляя разработчикам единый API (Прикладной Интерфейс). Библиотека была выпущена под двумя лицензиями: коммерческой -- для разработки коммерческого программного обеспечения, и свободной -- для разработки программ с открытым исходным кодом. Контракт с фирмой Metis помог сохранить Trolltech на плаву, поскольку на протяжении долгих 10 месяцев не была продано ни одной коммерческой версии библиотеки.

В марте 1996 года, Европейское Космическое Агентство закупило сразу 10 коммерческих лицензий. С несгибаемой верой в успех, Эрик и Хаавард наняли еще одного разработчика. В мае вышла Qt 0.97, а 24 сентября 1996 года увидела свет Qt 1.0. К концу года появилась Qt 1.1, а число покупателей достигло восьми, которые закупили 18 лицензий. В этом же году был основан проект KDE, во главе с Матиасом Эттричем.

В апреле 1997 года вышла Qt 1.2. Решение Матиаса, об использовании Qt в качестве основы для KDE, де-факто сделало библиотеку стандартом, для разработки графического интерфейса в Linux. В сентябре 1997 года вышла Qt 1.3.

Матиас присоединился к Trolltech в 1998 году, а в сентябре этого же года состоялся выход очередной версии Qt -- 1.40. В июне 1999 года вышла Qt 2, в которую было внесено большое количество архитектурных изменений. К тому же она стала более зрелой, чем ее предшественницы. Для поддержки Unicode в нее было добавлено 40 новых классов. Qt 2 была выпущена на основе новой открытой лицензии -- Q Public License (QPL), которая соответствовала Open Source Definition. В августе 1999, Qt победила на LinuxWorld, в номинации "Лучшая библиотека". Примерно в то же время была образована Trolltech Pty Ltd (Австралия).

Первый выпуск Qt/Embedded состоялся в 2000 году. Она была разработана для работы в устройствах под управлением Embedded Linux и предоставляла свою оконную подсистему -- легковесную замену X11. И Qt/Embedded, и Qt/X11 предлагались под широко используемой лицензией GNU General Public License (GPL), так же как и под коммерческими лицензиями. В конце 2000 года, Trolltech образовала Trolltech Inc. (США) и выпустила первую версию Qtopia -- графическая среда для

карманных устройств. В 2001 и в 2002 годах, Qt/Embedded стала победительницей на LinuxWorld, в номинации "Лучшее решение для Embedded Linux".

В 2001 году вышла Qt 3. Теперь эта библиотека может работать под управлением Windows, Unix, Linux, Embedded Linux и Mac OS X. В ее состав вошли 42 новых класса, а общий объем кода перевалил за 500 000 строк. Qt 3 стала победительницей Software Development Times "Jolt Productivity Award" 2002 года.

С момента рождения, из года в год, компания удваивала объем продаж. Этот успех обеспечивался высоким качеством библиотеки и простотой ее использования. На протяжении практически всего периода существования компании, за маркетинговую политику и объем продаж отвечали всего несколько человек. Менее чем за десятилетие, Qt превратилась из малоизвестной библиотеки в программный продукт, известный тысячам и тысячам разработчиков во всем мире.

ЧАСТЬ 1 - ОСНОВЫ QT

1 НАЧАЛО

В этой главе мы расскажем - как использовать функциональные возможности библиотеки Qt, для создания графического интерфейса с пользователем, в программах на языке C++. Напишем несколько небольших программ. А так же дадим краткое введение в два ключевых понятия Qt: **"сигналы и слоты"** и **"разметка"** (**layout**). В главе 2 будут даны более обширные сведения, а в главе 3 рассмотрим пример создания настоящего приложения.

1.1 Hello, Qt!

Ниже приводится текст простейшей Qt программы:

```
 1 #include <qapplication.h>
 2 #include <qlabel.h>
 3 int main(int argc, char *argv[])
 4 {
 5     QApplication app(argc, argv);
 6     QLabel *label = new QLabel("Hello, Qt!", 0);
 7     app.setMainWidget(label);
 8     label->show();
 9     return app.exec();
10 }
```

Здесь, в строках 1 и 2, подключаются определения классов **QApplication** и **QLabel**.

В строке 5 создается экземпляр класса **QApplication**, который управляет ресурсами приложения. Конструктору **QApplication** передаются аргументы **argc** и **argv**, поскольку Qt имеет возможность обрабатывать аргументы командной строки.

В строке 6 создается визуальный компонент **QLabel**, который отображает надпись "Hello, Qt!". В терминологии Qt, все визуальные компоненты, из которых строится графический интерфейс, называются виджетами (**widgets**). Кнопки, меню, полосы прокрутки и разнообразные рамки -- все это виджеты. Одни виджеты могут содержать в себе другие виджеты, например, главное окно приложения -- это самый обычный виджет, который может содержать **QMenuBar**, **QToolBar**, **QStatusBar** и др. Аргумент 0, передаваемый конструктору **QLabel** (в строке 6) -- это **"пустой"** (**null**) указатель, который сообщает о том, что этот виджет не имеет "хозяина", т.е. не включается в другой виджет.

В строке 7 назначается "главный" виджет приложения. Когда пользователь закрывает "главный" виджет приложения (например, нажатием на кнопку **"X"** в заголовке окна), то программа завершает свою работу. Если в программе не назначить главный виджет, то она продолжит исполнение в фоновом режиме даже после того, как пользователь закроет окно.

В строке 8, метка делается видимой. Виджеты всегда создаются невидимыми, чтобы у программиста оставалась возможность настроить параметры отображения до того, как они станут видимы.

В строке 9 выполняется передача управления библиотеке Qt. С этого момента программа переходит в режим ожидания, когда она ничего не делает, а просто ждет действий пользователя, например, нажатие на клавишу или кнопку мыши.
Любое действие пользователя порождает событие (другими словами -- "сообщение"), в ответ на которое программа может вызвать одну или более функций. В этом смысле, приложения с графическим интерфейсом кардинально отличаются от обычных программ, с пакетной обработкой данных, которые приняв ввод от пользователя, они самостоятельно обрабатывают его, выдают результаты и завершают свою работу без дальнейшего участия человека.

Рисунок 1.1. Окно приложения в Windows XP

Теперь самое время проверить работу нашего приложения. Но прежде всего -- необходимо, чтобы у вас была установлена Qt 3.2 (или более поздняя версия), а переменная окружения **PATH** содержала корректный путь к каталогу **bin**. (В Windows настройка переменной **PATH** выполняется автоматически, в процессе установки библиотеки Qt)
Скопируйте текст программы в файл, с именем **hello.cpp**, в каталог **hello**.
Перейдите в этот каталог и дайте команду:

```
qmake -project
```

она создаст платформо-независимый файл проекта (**hello.pro**), а затем дайте следующую команду:

```
qmake hello.pro
```

Эта команда создаст **Makefile**, на основе файла проекта. Дайте команду **make**, чтобы скомпилировать программу и затем запустите ее, набрав в командной строке **hello** (в Windows) или **./hello** (в Unix) или **open hello.app** (в Mac OS X). Если вы работаете в Windows и используете Microsoft Visual C++, то вместо команды **make** вы должны дать команду **nmake**. Как альтернативный вариант -- вы можете создать проект Visual Studio из файла **hello.pro**, запустив команду:

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

и затем скомпилировать программу в Visual Studio.

Рисунок 1.2. Метка с форматированным текстом.

А теперь немного развлечемся. Изменим внешний вид метки, добавив форматирование текста в стиле HTML. Для этого, замените строку

```
QLabel *label = new QLabel("Hello, Qt!", 0);
```

на

```
QLabel *label = new QLabel("<h2 ><i>Hello</i> "
                           "<font color=red>Qt!</font></h2>", 0);
```

и пересоберите приложение.

1.2 Обработка сигналов

Следующий пример показывает -- как организовать реакцию приложения на действия пользователя. Это приложение содержит кнопку, при нажатии на которую программа закрывается. Исходный текст очень похож на предыдущий пример, за исключением того, что теперь, в качестве главного виджета, вместо **QLabel** используется **QPushButton**, и добавлен код, который обслуживает факт ее нажатия.

Рисунок 1.3. Приложение Quit.

```
 1 #include <qapplication.h>
 2 #include <qpushbutton.h>
 3 int main(int argc, char *argv[])
 4 {
 5   QApplication app(argc, argv);
 6   QPushButton *button = new QPushButton("Quit", 0);
 7   QObject::connect(button, SIGNAL(clicked()),
 8                    &app, SLOT(quit()));
 9   app.setMainWidget(button);
10   button->show();
11   return app.exec();
12 }
```

Виджеты Qt имеют возможность посылать приложению сигналы, извещая его о том, что пользователь произвел какое-либо действие или о том, что виджет изменил свое состояние [3] . Например, экземпляры класса **QPushButton** посылают приложению сигнал **clicked()**, когда пользователь нажимает на кнопку. Сигнал может быть "подключен" к функции-обработчику (такие функции-обработчики в Qt называются слотами). Таким образом, когда виджет посылает сигнал, автоматически вызывается слот. В нашем примере мы подключили сигнал **clicked()**, от кнопки, к слоту **quit()**, экземпляра класса **QApplication**. Вызовы **SIGNAL()** и **SLOT()** -- это макроопределения, более подробно мы остановимся на них в следующей главе.

Теперь соберем приложение. Надеемся, что вы уже создали каталог **quit** и разместили в нем файл **quit.cpp**. Дайте команду **qmake**, для создания файла проекта, а затем второй раз -- для создания **Makefile**:

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

Теперь соберите приложение командой **make** и запустите его. Если вы щелкнете по кнопке **"Quit"** или нажмете на клавиатуре клавишу **"Пробел"**, то приложение завершит свою работу.
В следующем примере мы покажем как можно использовать сигналы и слоты для синхронизации двух виджетов. Эта программа предлагает пользователю ввести свой возраст. Сделать это можно либо с помощью кнопок управления счетчиком, либо с помощью ползунка.

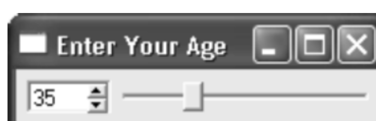


Рисунок 1.4. Приложение Age.

Приложение содержит три виджета: **QSpinBox**, **QSlider** и **QHBox** (область горизонтальной разметки). Главным виджетом приложения назначен **QHBox**. Компоненты **QSpinBox** и **QSlider** помещены внутрь **QHBox** и являются подчиненными, по отношению к нему.

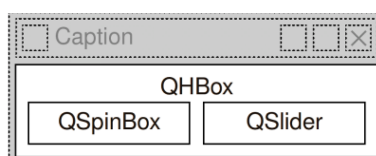


Рисунок 1.5. Виджеты приложения Age.

```
 1     #include <qapplication.h>
 2      #include <qhbox.h>
 3      #include <qslider.h>
 4      #include <qspinbox.h>
 5      int main(int argc, char *argv[])
 6      {
 7        QApplication app(argc, argv);
 8        QHBox *hbox = new QHBox(0);
 9        hbox->setCaption("Enter Your Age");
10        hbox->setMargin(6);
11        hbox->setSpacing(6);
```

```
12        QSpinBox *spinBox = new QSpinBox(hbox);
13        QSlider *slider = new QSlider(Qt::Horizontal, hbox);
14        spinBox->setRange(0, 130);
15        slider->setRange(0, 130);
16        QObject::connect(spinBox, SIGNAL(valueChanged(int)),
17                         slider, SLOT(setValue(int)));
18        QObject::connect(slider, SIGNAL(valueChanged(int)),
19                         spinBox, SLOT(setValue(int)));
20        spinBox->setValue(35);
21        app.setMainWidget(hbox);
22        hbox->show();
23        return app.exec();
24      }
```

В строках с 8 по 11 создается и настраивается **QHBox**. [4] Чтобы вывести текст в заголовке окна, вызывается **setCaption()**. А затем устанавливается размер пустого пространства (6 пикселей) вокруг и между подчиненными виджетами.

В строках 12 и 13 создаются **QSpinBox** и **QSlider**, которым, в качестве владельца, назначается **QHBox**.

Не смотря на то, что мы явно не задали ни положение, ни размеры виджетов **QSpinBox** и **QSlider**, тем менее они очень аккуратно расположились внутри **QHBox**. Собственно для этого и предназначен **QHBox**. Он выполняет автоматическое размещение подчиненных виджетов, назначая им координаты размещения и размеры, в зависимости от их требований и собственных настроек. В Qt имеется много классов, подобных **QHBox**, которые избавляют нас от рутинной работы по ручной подгонке положения и размеров визуальных компонентов.

В строках 14 и 15 устанавливаются допустимые пределы изменения счетчика и ползунка. (Мы можем смело предположить, что возраст нашего пользователя едва ли превысит 130 лет.) Два вызова **connect()**, в строках с 16 по 19 синхронизируют ползунок и счетчик, благодаря чему они всегда будут отображать одно и то же значение. Всякий раз, когда значение одного из виджетов изменяется, он посылает сигнал **valueChanged(int)**, который поступает в слот **setValue(int)** другого виджета.

В строке 20 устанавливается первоначальное значение (35) счетчика. Когда это происходит, счетчик посылает сигнал **valueChanged(int)**, со значением входного аргумента, равным 35. Это число передается в слот **setValue(int)** виджета **QSlider**, который устанавливает значение этого виджета равным 35. После этого уже **QSlider** посылает сигнал **valueChanged(int)**, поскольку его значение только что изменилось, вызывая таким образом слот **setValue(int)** виджета **QSpinBox**. Но на этот раз счетчик не посылает сигнал, поскольку его значение и так было равно 35. Таким образом, предотвращается бесконечная рекурсия. Рисунок 1.6 иллюстрирует эту ситуацию.

В строке 22 **QHBox** делается видимым (вместе со всеми подчиненными виджетами).
Подход к формированию интерфейса в Qt очень прост для понимания и чрезвычайно гибок. В общем случае, программист выбирает необходимые ему виджеты, размещает их внутри областей выравнивания (**layouts**), которые в свою очередь принимают на себя обязанности по размещению виджетов, и настраивает свойства виджетов. На заключительном этапе устанавливаются взаимосвязи виджетов, через механизм сигналов и слотов, которые обусловливают поведение пользовательского интерфейса.

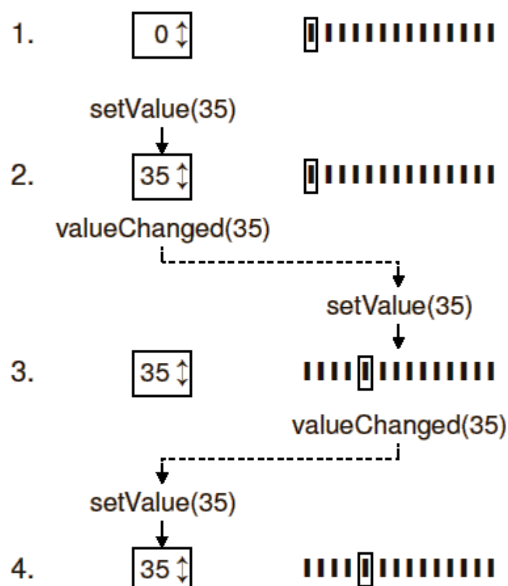


Рисунок 1.6. Изменение одного значения вызывает изменение другого.

1.3 Работа со справочной системой

Справочная система в Qt -- это пожалуй самый основной инструмент любого разработчика. Она описывает все классы и функции в этой библиотеке. (Документация к Qt 3.2 включает в себя описание более 400 классов и 6000 функций.) В этой книге вы встретитесь с большим количеством классов и функций Qt, но далеко не со всеми. Поэтому совершенно необходимо, чтобы вы самостоятельно ознакомились со справочной системой Qt.

1.3.1 Стили виджетов

Скриншоты, которые мы до сих пор видели, были получены в Windows XP. Однако внешний вид виджетов изменяется, в зависимости от платформы, на которой запускается приложение. С другой стороны, Qt в состоянии эмулировать внешний вид любой из поддерживаемых платформ.

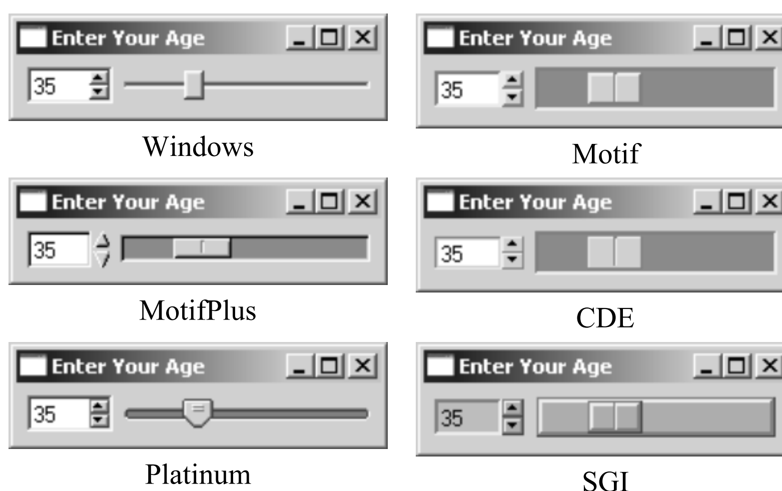


Рисунок 1.7. Стили, поддерживаемые Qt на любой платформе.

Пользователь может задать стиль отображения через параметр командной строки **-style**. Например, чтобы запустить приложение **Age** со стилем отображения Platinum в ОС Unix, нужно дать команду:

```
./age -style=Platinum
```

Windows XP Mac

Рисунок 1.8. Платформо-зависимые стили

В отличие от других, платформо-зависимые стили (Windows XP и Mac) доступны только на этих платформах, т.к. в этом случае отрисовка виджетов производится графическим ядром операционной системы.

Документация хранится в каталоге **doc\html** в виде html-файлов. Для ее просмотра может использоваться любой web-браузер. Но Qt имеет свою утилиту просмотра документации -- Qt Assistant, которая предоставляет очень удобный способ навигации по справочнику, гораздо удобнее, чем этого можно добиться в web-браузере. Чтобы запустить утилиту -- выберите пункт **Qt 3.2.x|Qt Assistant** в меню **"Пуск"** операционной системы Windows или дайте команду **assistant** в Unix.

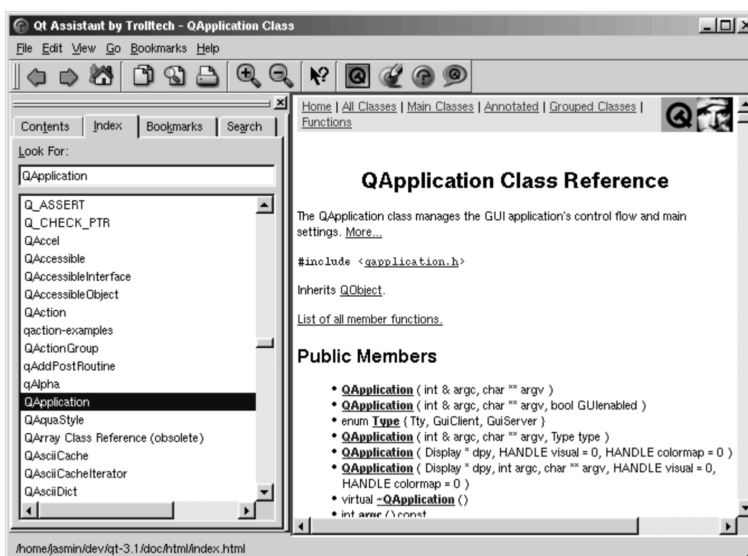


Рисунок 1.9. Внешний вид программы Qt Assistant.

Ссылки в разделе **"API Reference"** дают возможность навигации по классам различными способами. Так, например, перейдя по ссылке **"All Classes"** вы получите список всех классов библиотеки. По ссылке **"Main Classes"** -- только самые основные классы. В качестве упражнения попробуйте найти описания классов и функций, использовавшихся в нашем приложении. Обратите внимание: наследуемые методы описываются в базовых классах, например, описание класса **QPushButton** не содержит метода **show()**, поскольку он наследуется от класса **QWidget**. На рисунке ниже приводится диаграмма наследования для классов, использованных в нашем приложении:

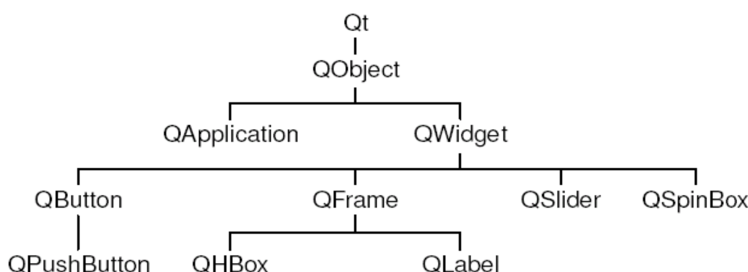


Рисунок 1.10. Дерево наследования интересующих нас классов.

Справочная документация для текущей версии Qt (и некоторых, более ранних версий) выкладывается в он-лайн по адресу: *http://doc.trolltech.com/*. Здесь же вы найдете отдельные статьи из ежеквартальника Qt Quarterly, который рассылается всем коммерческим пользователям.

2 Создание диалогов

В этой главе мы расскажем -- как в Qt создаются диалоговые окна. Диалоговыми они называются потому, что обеспечивают способ общения (диалога) между пользователем и приложением.

Диалоги предоставляют пользователям возможность что-то изменить, что-то задать, в соответствии с их предпочтениями. В большинстве своем, программы с графическим интерфейсом имеют главное окно, с полосой меню и панелью инструментов, и множество диалоговых окон, каждое из которых предназначено для обмена информацией с пользователем или вывода ее в определенном формате. Зачастую приложение может быть оформлено как одно диалоговое окно, которое напрямую взаимодействует с пользователем, получая от него команды и выполняя соответствующие им действия. Такие приложения называются диалоговыми приложениями. Примером диалогового приложения может служить программа-калькулятор.

Для начала мы создадим диалоговое приложение "вручную", чтобы продемонстрировать основные принципы разработки диалогов. Затем покажем, как то же самое можно сделать значительно быстрее, с применением визуального построителя Qt Designer. С помощью этого инструмента разработка диалоговых окон проходит намного быстрее, к тому же с его помощью гораздо проще вносить изменения в визуальный дизайн во время доработки приложения.

2.1 Создание дочернего класса от QDialog

Наш первый пример -- это диалог поиска. Диалог будет реализован в виде класса, со своей собственной функциональностью. Это будет независимый компонент со своими сигналами и слотами.

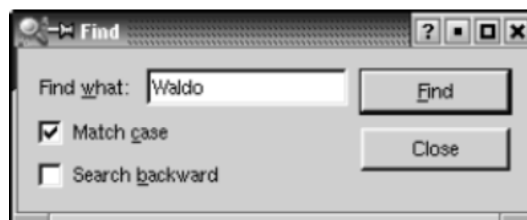


Рисунок 2.1. Диалог поиска в Linux (KDE).

Исходные тексты приложения будут размещаться в двух файлах: **finddialog.h** и **finddialog.cpp**. Начнем с файла **finddialog.h**.

```
 1 #ifndef FINDDIALOG_H
 2 #define FINDDIALOG_H
 3 #include <qdialog.h>
 4 class QCheckBox;
 5 class QLabel;
 6 class QLineEdit;
 7 class QPushButton;
```

Строки 1 и 2 (и 27) предотвращают множественное подключение файла-заголовка.

В строке 3 подключается определение класса **QDialog** -- базового для всех диалогов в Qt. **QDialog** порожден от класса **QWidget**.

Строки с 4 по 7 -- это опережающие описания классов Qt, которые используются в нашем диалоге. Опережающее описание сообщает компилятору C++ о том, что этот класс существует, но подробности описания класса (обычно расположенного в отдельном заголовочном файле) здесь использоваться не будут. Ниже мы еще вернемся к этой теме.

Теперь определим класс **FindDialog**, указав в качестве родительского класса **QDialog**:

```
 8 class FindDialog : public QDialog
 9 {
```

```
10     Q_OBJECT
11 public:
12     FindDialog(QWidget *parent = 0, const char *name = 0);
```

Определение класса начинается с вызова макроса **Q_OBJECT**. Это обязательное требование для всех классов, которые определяют свои собственные сигналы и слоты.

Далее следует типичный, для всех виджетов в Qt, конструктор -- **FindDialog**. Параметр **parent** задает "владельца", т.е. виджет, на котором будет размещен данный компонент, а параметр **name** -- имя (название) виджета. Параметр **name** не является обязательным, в основном он используется для нужд отладки и тестирования.

```
13 signals:
14     void findNext(const QString &str, bool caseSensitive);
15     void findPrev(const QString &str, bool caseSensitive);
```

В секции **signals** описаны два сигнала, которые может посылать наше диалоговое окно, при нажатии на кнопку **Find**. Если включен флажок **"Search backward"** (поиск в обратном направлении), то посылается сигнал **findPrev()**, иначе -- **findNext()**.
Ключевое слово **signals**, фактически является макроопределением. Препроцессор C++ преобразует его в стандартное представление C++ и только потом оно будет передано компилятору.

```
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 приватной секции класса мы объявили два слота. Они необходимы для обеспечения взаимодействия с подчиненными виджетами, указатели на которые описаны чуть ниже. Ключевое слово **slots**, так же как и **signals**, является макроопределением.
Поскольку все поля-переменные -- это указатели, нам нет нужды подключать заголовочные файлы, содержащие полные определения этих классов. Благодаря наличию опережающего описания, компилятор удовольствуется тем, что есть. Вместо опережающего описания классов мы могли бы подключить соответствующие заголовочные файлы (**<qcheckbox.h>**, **<qlabel.h>** и т.д.), но это отрицательно скажется на скорости компиляции. Для маленьких приложений это не так заметно, но для больших проектов опережающее описание может дать существенный выигрыш.

Перейдем к файлу **finddialog.cpp**, который содержит реализацию класса **FindDialog**:

```
 1 #include <qcheckbox.h>
 2 #include <qlabel.h>
 3 #include <qlayout.h>
 4 #include <qlineedit.h>
 5 #include <qpushbutton.h>
 6 #include "finddialog.h"
```

Здесь подключаются заголовочные файлы с описаниями используемых классов Qt. В строке 6 подключается определение нашего класса. Для большинства классов Qt, их определения находятся в заголовочных файлах, имена которых повторяют имя класса (все символы в именах файлов переводятся в нижний регистр) и дополняются символами **.h**.

```
 7 FindDialog::FindDialog(QWidget *parent, const char *name)
 8     : QDialog(parent, name)
 9 {
10     setCaption(tr("Find"));
11     label = new QLabel(tr("Find &what:"), this);
```

```
12      lineEdit = new QLineEdit(this);
13      label->setBuddy(lineEdit);
14      caseCheckBox = new QCheckBox(tr("Match &case"), this);
15      backwardCheckBox = new QCheckBox(tr("Search &backward"), this);
16      findButton = new QPushButton(tr("&Find"), this);
17      findButton->setDefault(true);
18      findButton->setEnabled(false);
19      closeButton = new QPushButton(tr("Close"), this);
```

В строке 8, конструктору базового класса передаются параметры **parent** и **name**.

В строке 10 задается надпись, которая будет выводиться в заголовке окна -- **"Find"**. Функция **tr()** определена в классе **QObject** и любом другом подклассе, описание которого содержит вызов макроса **Q_OBJECT**. Она выполняет трансляцию текста, передаваемого ей, на другие языки человеческого общения. Считается хорошим тоном, все строки, которые будут выводиться на экран, передавать через эту функцию, даже в том случае, если вы не планируете интернационализацию своего приложения. Проблемы интернационализации будут подробно освещены в Главе 15.

Затем, начиная со строки 11, создаются подчиненные виджеты. Здесь символ амперсанда (**'&'**) используется для обозначения клавиши быстрого доступа (акселератор). Например, в строке 16 создается кнопка **Find**, активировать которую можно нажатием на комбинацию клавиш **Alt+F**. Амперсанды так же могут использоваться для передачи фокуса ввода: в строке 11 создается метка, с акселератором (**Alt+W**), а в строке 13 метке назначается "дружественный" (buddy) компонент -- однострочное поле ввода. "Дружественный" виджет -- это такой компонент, который будет получать фокус ввода при нажатии ускоряющей комбинации клавиш метки. Таким образом, когда пользователь нажмет комбинацию клавиш **Alt+W** (акселератор метки), то фокус ввода будет передан "дружественному" виджету, т.е. -- полю ввода.

В строке 17, вызовом метода **setDefault(true)** [5] , назначается кнопка по умолчанию. Кнопка по умолчанию -- это такая кнопка, которая будет активироваться при нажатии на клавишу **Enter**. В строке 18 накладывается запрет на кнопку **Find**. Когда виджет запрещен, он обычно отображается в серых тонах и не реагирует на действия пользователя.

```
20 connect(lineEdit, SIGNAL(textChanged(const QString &)),
21     this, SLOT(enableFindButton(const QString &)));
22 connect(findButton, SIGNAL(clicked()),
23     this, SLOT(findClicked()));
24 connect(closeButton, SIGNAL(clicked()),
25     this, SLOT(close()));
```

Приватный слот **enableFindButton(const QString &)** вызывается при изменении содержимого поля ввода. Приватный слот **findClicked()** вызывается, когда пользователь щелкает по кнопке **Find**. Работа приложения завершается после щелчка по кнопке **Close**. Слот **close()** наследуется от класса **QWidget** и его поведение по умолчанию -- сокрытие виджета. Реализация слотов **findClicked()** и **enableFindButton(const QString &)** будет приведена ниже.

Поскольку **QObject** является одним из предков класса **FindDialog**, то мы можем опустить префикс **QObject::** перед именем метода **connect()**.

```
26      QHBoxLayout *topLeftLayout = new QHBoxLayout;
27      topLeftLayout->addWidget(label);
28      topLeftLayout->addWidget(lineEdit);
29      QVBoxLayout *leftLayout = new QVBoxLayout;
30      leftLayout->addLayout(topLeftLayout);
31      leftLayout->addWidget(caseCheckBox);
32      leftLayout->addWidget(backwardCheckBox);
33      QVBoxLayout *rightLayout = new QVBoxLayout;
34      rightLayout->addWidget(findButton);
35      rightLayout->addWidget(closeButton);
36      rightLayout->addStretch(1);
37      QHBoxLayout *mainLayout = new QHBoxLayout(this);
38      mainLayout->setMargin(11);
39      mainLayout->setSpacing(6);
40      mainLayout->addLayout(leftLayout);
```

```
41     mainLayout->addLayout(rightLayout);
42 }
```

На заключительном этапе выполняется выравнивание виджетов с помощью менеджеров размещения. Менеджер размещения (**layout manager**) -- это объект, который управляет размерами и положением виджетов. Qt предоставляет в наше распоряжение три менеджера размещения: **QHBoxLayout** выравнивает виджеты по горизонтали, **QVBoxLayout** -- по вертикали и **QGridLayouts** -- по сетке. Менеджеры размещения (или, если хотите, менеджеры компоновки) могут содержать как отдельные виджеты, так и другие менеджеры размещения. Вкладывая друг в друга **QHBoxLayout**, **QVBoxLayout** и **QGridLayouts**, в различных комбинациях, можно выстроить весьма замысловатый интерфейс диалога.

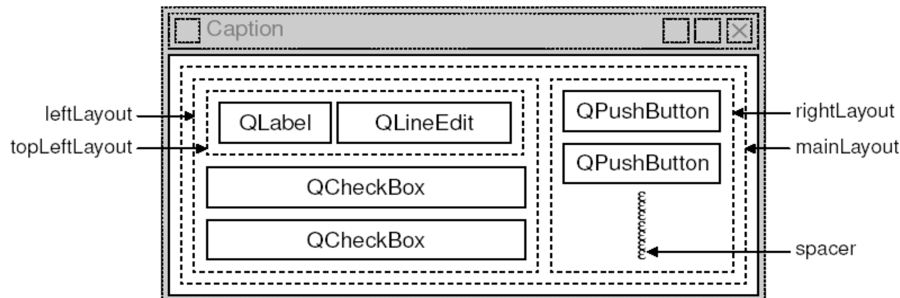


Рисунок 2.2. Компоновка диалога Find.

В нашем приложении мы использовали два **QHBoxLayout** и два **QVBoxLayout**, как это показано на Рисунке 2.2. Внешний менеджер компоновки (**mainLayout**) является главным, поскольку при создании ему был назначен, в качестве владельца, экземпляр класса **FindDialog** -- (**this**). Он отвечает за размещение всех визуальных компонентов в области окна приложения. Оставшиеся 3 менеджера размещения являются подчиненными. Маленькая "пружинка", которая видна в правом нижнем углу рисунка -- это распорка (**spacer**). Она заполняет пустое пространство под кнопками **Find** и **Close**, заставляя их держаться в верхней части области выравнивания.
Здесь есть один важный момент. Менеджеры компоновки не являются виджетами. Они порождены от класса **QLayout**, который в свою очередь порожден от класса **QObject**. На рисунке, контуры виджетов отрисованы сплошными линиями, а областей компоновки -- пунктирными, чтобы подчеркнуть различия, имеющиеся между ними. Во время работы приложения, менеджеры размещения (области выравнивания) не видны.

Хотя менеджеры компоновки и не являются виджетами (визуальными компонентами), тем не менее они могут иметь как владельца, так и подчиненные компоненты. Понятия термина "владелец", у менеджера размещения и виджета, различаются. Если менеджер размещения встраивается в виджет (который передается менеджеру в качестве владельца), как это происходит в случае с **mainLayout**, то он автоматически встраивается в этот виджет. Если менеджер создается без владельца (в данном случае это: **topLeftLayout**, **leftLayout** и **rightLayout**), то он должен быть включен в состав другого менеджера, вызовом **addLayout()**.
Механизм "владелец-подчиненный" реализован в классе **QObject**, который является предком как для **QWidget**, так и для **QLayout**. Когда создается некий объект (виджет, менеджер компоновки или что-то еще), для которого явно указывается владелец, то он добавляется владельцем в свой список подчиненных компонентов. Когда владелец уничтожается, он проходит по списку подчиненных компонентов и уничтожает их одного за другим. Подчиненные компоненты, в свою очередь просматривают свои списки и уничтожают компоненты, подчиненные им и так до тех пор, пока дело не дойдет до компонентов, которые не имеют подчиненных объектов.

Этот механизм упрощает управление памятью в приложении, снижая риск "утечки". Единственные объекты, которые необходимо уничтожать явно -- это те, которые были созданы оператором **new**, и не имеют владельца. Если первым удаляется подчиненный компонент, то Qt автоматически исключит его из списка владельца.

Владелец имеет особое значение для виджетов. Подчиненные виджеты отображаются на экране внутри области, принадлежащей его владельцу. Когда удаляется какой-либо владелец, он не только удалит подчиненные объекты из памяти, но и сотрет их на экране.

Когда один менеджер размещения вставляется в другой, с помощью функции **addLayout()**, то вложенный менеджер автоматически делается подчиненным объемлющему. В противоположность этому, когда в менеджер размещения вставляется виджет, вызовом **addWidget()**, то последний не меняет своего владельца.

На рисунке 2.3 показано дерево "владелец-подчиненный" для приложения **Find**. Порядок взаимоотношений в этом дереве легко выводится из текста конструктора **FindDialog**, достаточно выделить строки, содержащие **new** и **addLayout()**. Важное замечание: Запомните, менеджеры размещения НЕ ЯВЛЯЮТСЯ владельцами виджетов, размещением которых они управляют.

В дополнение к менеджерам размещения, Qt предоставляет несколько виджетов размещения: **QHBox** (с которым мы уже встречались в Главе 1), **QVBox** и **QGrid**. Для своих подчиненных компонентов, эти классы выступают как в качестве владельцев так и в качестве менеджеров размещения. Для небольших приложений виджеты размещения более удобны, но они менее гибкие и требуют больший объем ресурсов, чем менеджеры.

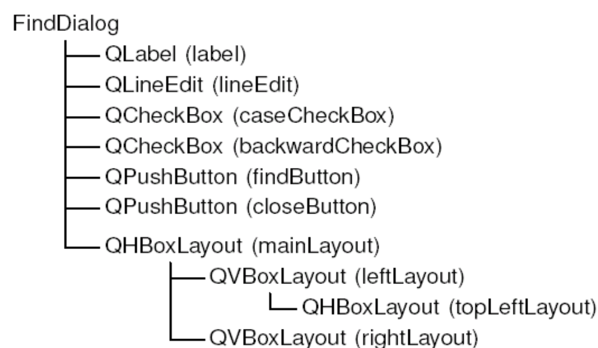


Рисунок 2.3. Дерево "владелец-подчиненный" диалога Find.

На этом мы заканчиваем рассмотрение реализации конструктора **FindDialog**. Поскольку подчиненные виджеты и менеджеры размещения создавались нами с помощью оператора **new**, то, казалось бы, возникает необходимость в написании деструктора, который удалил бы из памяти все, созданные нами компоненты, во время завершения приложения. Однако в этом нет необходимости, поскольку Qt автоматически сделает это во время уничтожения владельца, т.е. во время уничтожения класса **FindDialog**.
Перейдем к рассмотрению слотов:

```
43 void FindDialog::findClicked()
44 {
45     QString text = lineEdit->text();
46     bool caseSensitive = caseCheckBox->isOn();

47     if (backwardCheckBox->isOn())
48         emit findPrev(text, caseSensitive);
49     else
50         emit findNext(text, caseSensitive);
51 }

52 void FindDialog::enableFindButton(const QString &text)
53 {
54     findButton->setEnabled(!text.isEmpty());
55 }
```

Слот **findClicked()** вызывается всякий раз, когда пользователь щелкает мышкой по кнопке **Find**. Кнопка выдает сигнал **findPrev()** или **findNext()**, в зависимости от состояния флажка **Search backward**. Ключевое слово **emit** является макросом, определенным в библиотеке Qt.
Слот **enableFindButton()** вызывается, когда пользователь изменяет содержимое поля ввода. Если оно содержит какие-либо символы, то разрешается кнопка **Find**, в противном случае она запрещается.

Реализацией этих двух слотов завершается разработка нашего диалога. Теперь создадим файл **main.cpp**, в котором разместим текст основной программы для тестирования нашего виджета:

```
 1 #include <qapplication.h>
 2 #include "finddialog.h"

 3 int main(int argc, char *argv[])
 4 {
 5     QApplication app(argc, argv);
 6     FindDialog *dialog = new FindDialog;
 7     app.setMainWidget(dialog);
 8     dialog->show();
 9     return app.exec();
10 }
```

дадим команду **qmake**, как обычно. На этот раз, поскольку наш класс **FindDialog** содержит вызов макроопределения **Q_OBJECT**, утилита **qmake** включит в **Makefile** правила, вызывающие утилиту **moc** -- компилятор метаобъектов.

Для корректной работы утилиты **moc** необходимо, чтобы описание класса размещалось в заголовочном файле, отдельно от файла с реализацией. Код, сгенерированный утилитой **moc** подключает этот заголовочный файл.

Классы, в определении которых встречается макрос **Q_OBJECT**, должны обрабатываться компилятором метаобъектов в обязательном порядке. На самом деле, это не такая большая проблема, поскольку **qmake** автоматически добавит все необходимые правила в **Makefile**. Но если вы забудете перегенерировать **Makefile**, то линковщик будет "жаловаться" на отсутствие некоторых функций. Эти сообщения об ошибках могут порой вводить в заблуждение. Например, GCC выдает примерно такое предупреждение:

```
finddialog.o(.text+0x28): undefined reference to
FindDialog::QPaintDevice virtual table
```

Visual C++ такое:

```
finddialog.obj : error LNK2001: unresolved external symbol
"public:~virtual bool __thiscall FindDialog::qt_property(int,
int,class QVariant *)"
```

Если это произошло, то перезапустите **qmake** еще раз, чтобы обновить **Makefile**, а затем пересоберите приложение.
Теперь запустите программу. Проверьте работу акселераторов **Alt+W**, **Alt+C**, **Alt+B** и **Alt+F**. Попробуйте "пройтись" по виджетам с помощью клавиши **Tab**. По-умолчанию, порядок навигации с помощью клавиши **Tab**, соответствует порядку, в котором создавались компоненты. Но он может быть изменен вызовом метода **QWidget::setTabOrder()**. Установку акселераторов и настройку порядка навигации по компонентам, с помощью клавиши **Tab**, можно считать дружественным жестом в сторону пользователей, которые не могут или не хотят пользоваться мышью. Удобное управление с клавиатуры высоко оценят опытные пользователи.
В Главе 3 мы будем использовать наш диалог в реально работающем приложении. Там сигналы **findPrev()** и **findNext()** будут подключаться к соответствующим слотам.

2.2 Сигналы и слоты

Сигналы и слоты являются одним из фундаментальных механизмов в Qt. Он позволяет наладить обмен информацией между объектами, которые ничего не знают друг о друге. Мы уже пробовали присоединять сигналы к слотам, объявляли свои собственные сигналы и слоты, выполняли реализацию своих слотов и посылали свои сигналы. Теперь рассмотрим этот механизм поближе.

По своей природе, слоты очень близки к обычным функциям-членам в языке C++. Они могут быть виртуальными, они могут подвергаться перегрузке, они могут быть публичными, защищенными или приватными и они могут вызываться напрямую, как и обычные функции-члены. Отличие состоит в

том, что слот может быть подключен к сигналу. В этом случае, функция-слот вызывается автоматически всякий раз, когда посылается сигнал.

Объявление **connect()** выглядит следующим образом:

```
connect(sender, SIGNAL(signal), receiver, SLOT(slot));
```

где **sender** и **receiver** -- это указатели на экземпляры класса **QObject** (или его потомки), а **signal** и **slot** -- это сигнатуры функций. Макросы **SIGNAL()** и **SLOT()** по сути преобразуют свои аргументы в строки. В наших примерах мы до сих пор подключали к каждому из сигналов только один слот. Однако это не единственный способ.

Один сигнал может быть подключен к нескольким слотам:

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

Когда подается сигнал, то функции-слоты вызываются одна за другой, в порядке подключения.

К одному слоту может быть подключено несколько сигналов:

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

Когда посылается какой-либо из сигналов -- вызывается функция-слот.

Сигнал может быть подключен к другому сигналу:

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

Когда посылается первый сигнал, то вслед за ним подается и второй. С точки зрения программы, соединение типа сигнал-сигнал ничем не отличается от соединения типа сигнал-слот.

Связь между сигналом и слотом может быть разорвана:

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

Необходимость в этом возникает довольно редко, поскольку Qt сама автоматически разрывает соединение, если один из объектов уничтожается.

Соединяемые сигналы и слоты должны иметь идентичные сигнатуры (т.е. количество и типы входных аргументов):

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

Исключение составляет случай, когда сигнал имеет большее число аргументов, чем слот. В этом случае "лишние" аргументы просто не передаются в слот.
Если типы входных аргументов не совместимы, или сигнал или слот не определены, Qt выдаст предупреждение во время исполнения. Точно так же Qt выдаст предупреждение, если в сигнатуры сигналов или слотов включены имена аргументов (в методе **connect()**).

2.2.1 Метаобъектная Система в библиотеке Qt

Одно из самых значительных достижений Qt -- это расширение возможностей языка C++ механизмом создания независимых компонентов, которые могут взаимодействовать между собой, не имея

информации друг о друге. Этот механизм получил название Meta Object System и предоставляет два ключевых сервиса: сигналы-слоты и интроспекцию. Интроспекция позволяет получать метаинформацию о потомках класса **QObject** во время исполнения, включая список поддерживаемых сигналов, слотов и имя класса объекта. Этот механизм также реализует поддержку свойств объектов (используются в Qt Designer) и перевод текста (для нужд интернационализации). Стандарт C++ не обеспечивает возможность получения динамической метаинформации, которая необходима метаобъектной системе Qt. Поэтому данная проблема была решена созданием дополнительного инструмента **moc** (метаобъектный компилятор). Он собирает всю необходимую информацию из классов **Q_OBJECT** и делает ее доступной через вызовы обычных функций языка C++, что позволяет метаобъектной системе работать с любым компилятором C++. Механизм работает следующим образом:·Макрос **Q_OBJECT** объявляет ряд функций, который должны присутствовать в каждом потомке **QObject**: **metaObject()**, **className()**, **tr()** и ряд других.·

Утилита **moc** генерирует реализацию сигналов и функций, объявленных макросом **Q_OBJECT**.

Эти функции используются методами **connect()** и **disconnect()**.Все действия выполняются автоматически, утилитами **qmake** и **moc**, так что вы довольно редко будете вспоминать об этом. Но если вас одолевает любопытство -- загляните в исходные файлы, созданные **moc**, и посмотрите -- что да как.

До сих пор мы использовали сигналы и слоты исключительно с виджетами. Однако, этот механизм реализован непосредственно в классе **QObject** и область его применения не ограничивается графическим интерфейсом. Он может использоваться любым классом, наследником **QObject**:

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

Обратите внимание на реализацию слота **setSalary()**. Сигнал **salaryChanged()** посылается только в том случае, когда **newSalary != mySalary**. Такой способ предотвращает попадание в бесконечный цикл при наличии обратной связи с другим объектом.

2.3 Быстрая разработка диалогов

Qt разрабатывалась так, чтобы можно было писать код программы вручную без особого напряжения. Тем не менее, Qt Designer еще больше расширяет возможности программиста, предоставляя ему возможность визуального дизайна.
В этом разделе мы, с помощью Qt Designer, напишем диалог **Go-to-Cell** ("Перейти к ячейке"), показанный на рисунке 2.4. Совершенно неважно, как разрабатывается диалог -- вручную ли, или с помощью Qt Designer, всегда выполняется одна и та же последовательность действий:

- Создаются и инициализируются подчиненные виджеты.
- Подчиненные виджеты вставляются в менеджеры размещения.
- Настраивается порядок навигации по виджетам клавишей Tab.
- Устанавливаются соединения сигнал-слот

- Реализуются дополнительные слоты диалога, если это необходимо.

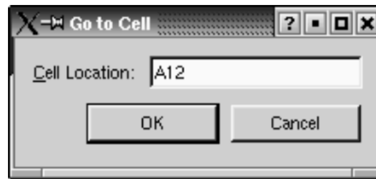


Рисунок 2.4. Диалог перехода к ячейке.

Чтобы запустить Qt Designer, выберите пункт **Qt 3.2.x|Qt Designer** в меню **"Пуск"** -- в ОС Windows или дайте команду **designer** -- в Unix. После того как программа запустится, она предложит на выбор список шаблонов. Щелкните по шаблону **"Dialog"** и нажмите кнопку **OK**. После этого перед вами должна появиться заготовка будущего окна диалога с именем "**Form1**".

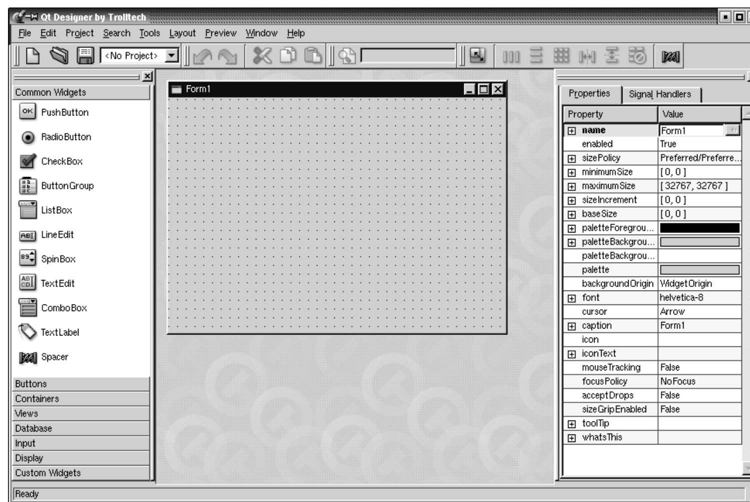


Рисунок 2.5. Qt Designer с заготовкой окна диалога.

Для начала разместим виджеты на форме. На инструментальной панели, слева, щелкните по компоненту **TextLabel**, затем щелкните по форме -- в результате на форме появится компонент "метка". Аналогичным образом разместите на форме одно поле ввода (**LineEdit**), одну горизонтальную распорку (**Spacer**) и две кнопки (**PushButton**). Разместите их так, чтобы у вас получилось нечто похожее на рисунок 2.6. Не тратьте слишком много времени на позиционирование виджетов. Мы все равно будем использовать менеджеры размещения, которые выполнят эту работу за нас.

Распорка (**spacer**) отображается на заготовке в виде синей пружинки. Во время работы уже готовой программы распорки не будут отображаться.

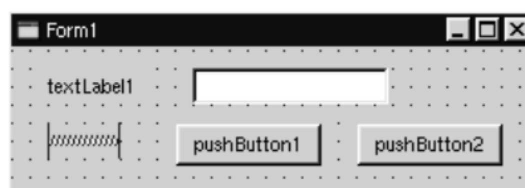


Рисунок 2.6. Внешний вид заготовки формы с виджетами.

Установите свойства для каждого из виджетов, используя Редактор свойств, расположенный в правой части главного окна Qt Designer.

- Щелкните по компоненту **TextLabel** и запишите в его свойство **name** строку "**label**", а в свойство **text** -- "**&Cell Location:**".
- Щелкните по компоненту **LineEdit** и запишите в свойство **name** строку "**lineEdit**".
- Для распорки запишите в свойство **orientation** "**Horizontal**".
- Для первой кнопки запишите в свойство **name** строку "**okButton**", в свойство **enabled** -- "**False**", в свойство **default** -- "**True**" и в свойство **text** -- "**OK**".

- Для второй кнопки. Запишите в свойство **name** строку "**cancelButton**", а в свойство **text** -- в "**OK**".
- Щелкните в любом свободном месте формы и запишите в свойство **name** строку "**GoToCellDialog**", а в свойство **caption** -- "**Go to Cell**".

Но это еще не все, нам нужно назначить дружественный компонент для метки, который будет реагировать на акселератор **Alt+C**. На данный момент метка отображается как "**&Cell Location:**". Выберите пункт меню **Tools|Set Buddy** (курсор мыши приобретет вид крестика). Затем поместите указатель мыши на метку, нажмите левую кнопку и, удерживая ее в нажатом положении, переместите указатель мыши на компонент **LineEdit**. Затем отпустите кнопку мыши. Изображение метки изменится -- символ амперсанда исчезнет, а первый символ метки приобретет знак подчеркивания. В принципе, то же самое можно сделать внутри редактора свойств, установкой свойства **buddy** метки.

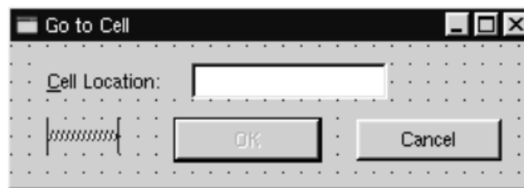


Рисунок 2.7. Внешний вид заготовки формы после установки свойств виджетов.

Следующий шаг -- размещение виджетов на форме с помощью менеджеров компоновки:
- Щелкните мышью по метке. Нажмите клавишу **Shift** и удерживая ее -- щелкните по полю ввода. Оба компонента окажутся выделенными. Теперь выберите пункт меню **Layout|Lay Out Horizontally**.
- Щелкните мышью по распорке. Нажмите клавишу **Shift** и удерживая ее -- щелкните сначала по кнопке "**OK**", а затем по кнопке "**Cancel**". Теперь выберите пункт меню **Layout|Lay Out Horizontally**.
- Щелкните по свободному пространству на форме и выберите пункт меню **Layout|Lay Out Vertically**.
- Выберите пункт меню **Layout|Adjust Size**.

Красные контуры областей выравнивания, которые сейчас видны на заготовке, во время работы программы отображаться не будут.

Рисунок 2.8. Внешний вид заготовки формы после настройки размещения компонентов.

Теперь выберите пункт меню **Tools|Tab Order**. На каждом из виджетов, которые могут принимать фокус, появятся цифры в синих кружочках. Щелчками мыши по компонентам установите желаемый порядок навигации клавишей **Tab** и нажмите **Esc**.

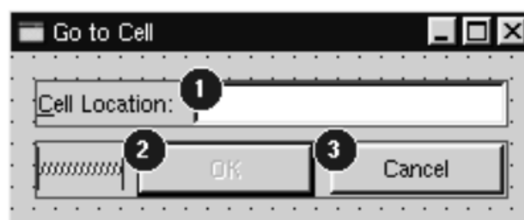


Рисунок 2.9. Установка порядка навигации клавишей Tab.

На этом дизайн внешнего вида формы можно считать завершенным. Теперь перейдем к настройке функциональной части -- свяжем сигналы и слоты и создадим свой слот. Выберите пункт меню **Edit|Connections**, перед вами откроется окно редактора связей:

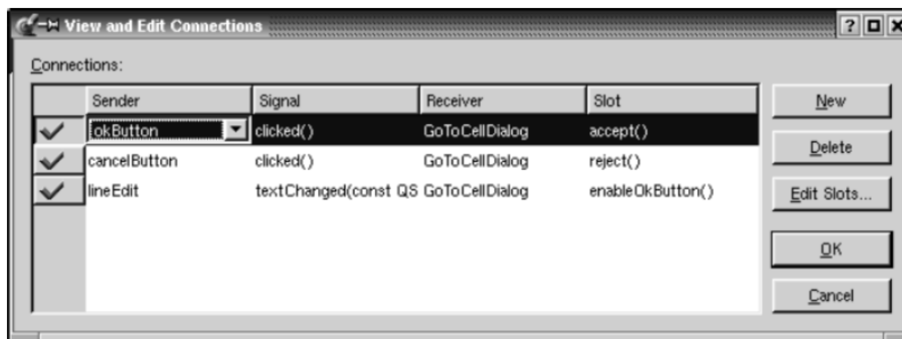


Рисунок 2.10. Окно редактора связей после установки всех соединений.

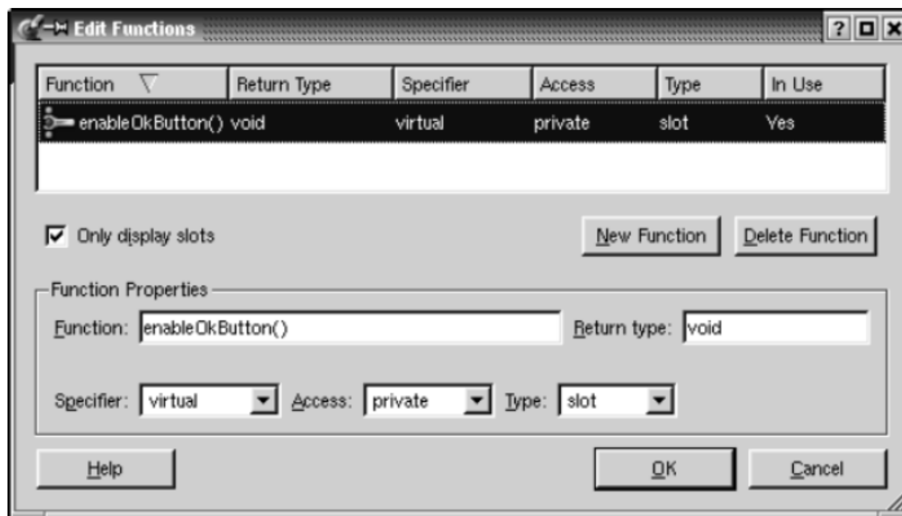


Рисунок 2.11. Окно редактора слотов.

Для начала создадим новый слот: щелкните по кнопке **Edit Slots**.... Перед вами откроется окно редактора слотов (см. рис. 2.11). Создайте приватный слот с именем **enableOkButton()**.
Затем необходимо настроить три соединения. Чтобы создать соединение -- щелкните по кнопке "**New**" и установите поля **Sender**, **Signal**, **Receiver** и **Slot**, выбирая требуемые значения из выпадающих списков в каждом из них. У вас должно получиться следующее:

```
okButton      clicked()                    GoToCellDialog   accept()
cancelButton  clicked()                    GoToCellDialog   reject()
lineEdit      textChanged(const QString &) GoToCellDialog   enableOkButton()
```

Чтобы посмотреть, как будет выглядеть окно диалога во время работы программы -- выберите пункт меню **Preview|Preview Form**. Проверьте порядок навигации клавишей **Tab**. Проверьте работу акселератора **Alt+C** (поле ввода должно получить фокус ввода). Нажмите кнопку **Cancel**, чтобы закрыть окно.

Сохраните результаты работы в файле **gotocelldialog.ui** в каталоге **gotocell** и создайте файл **main.cpp**, в том же каталоге, с помощью обычного текстового редактора:

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

Создайте файл проекта и **Makefile** утилитой **qmake** (**qmake -project**; **qmake gotocell.pro**). Утилита **qmake** сама обнаружит файл **gotocelldialog.ui** и добавит в **Makefile** все необходимые правила по созданию

gotocelldialog.h и **gotocelldialog.cpp**. Все **.ui** файлы преобразуются в код C++ с помощью утилиты **uic** (**User Interface Compiler** -- Компилятор Пользовательских Интерфейсов).
Вся прелесть Qt Designer-а состоит в том, что вы можете свободно изменять дизайн формы без необходимости вторгаться в исходный код на C++. Когда разработка дизайна ведется в тексте программы (вручную) то это может отнять довольно значительное время. Qt Designer сохранит ваши силы и время.

Если теперь запустить программу, то вы заметите:

- Кнопка "OK" всегда остается запрещенной.
- Поле ввода принимает не только те символы, из которых может состоять номер искомой ячейки, но и любые другие.

Мы должны решить эти проблемы.
Щелкните дважды по свободному пространству на форме, чтобы вызвать редактор исходного кода. В окне редактора добавьте следующие строки:

```
#include <qvalidator.h>
void GoToCellDialog::init()
{
    QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
    lineEdit->setValidator(new QRegExpValidator(regExp, this));
}
void GoToCellDialog::enableOkButton()
{
    okButton->setEnabled(lineEdit->hasAcceptableInput());
}
```

Функция **init()** автоматически вызывается конструктором формы (конструктор генерируется утилитой **uic**). Она настраивает проверку корректности ввода для **LineEdit**. Qt предоставляет три класса, выполняющих проверку на корректность: **QIntValidator**, **QDoubleValidator** и **QRegExpValidator**. Для своих нужд мы будем использовать последний, который будет выполнять проверку на основе регулярного выражения: "**[A-Za-z][1-9][0-9]{0,2}**". Это выражение означает: "Позволить ввод одного алфавитного символа в верхнем или нижнем регистре, за которым должна следовать одна цифра, в диапазоне от 1 до 9, за которой может следовать до двух цифр, в диапазоне от 0 до 9".
Передавая аргумент **this** (в вызов конструктора **QRegExpValidator()**), мы делаем объект класса **QRegExpValidator** подчиненным, по отношению к **GoToCellDialog**. Таким образом мы снимаем с себя ответственность за удаление этого объекта из памяти по завершении работы приложения.

Слот **enableOkButton()** разрешает или запрещает кнопку "**OK**", в зависимости от того, насколько правильный номер ячейки содержится в поле ввода. Для проверки правильности используется функция **QLineEdit::hasAcceptableInput()**, которая обращается к объекту класса **QRegExpValidator**, созданному в функции **init()**.

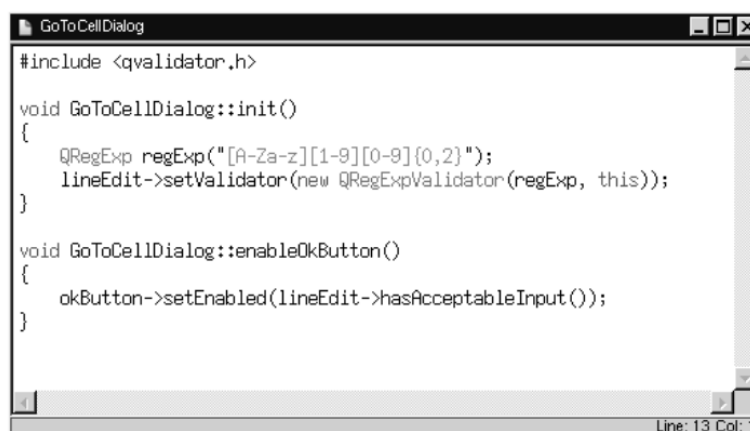


Рисунок 2.12. Окно редактора исходного кода.

После этого опять сохраните диалог. Qt Designer сохранит оба файла -- и **gotocelldialog.ui**, и **gotocelldialog.ui.h**. Пересоберите приложение и запустите его. Введите в поле ввода строку "**A12**" --

кнопка "**OK**" перейдет в разрешенное состояние. Попробуйте набрать произвольный текст и понаблюдайте за тем, как работает проверка корректности ввода. Нажмите кнопку "**Cancel**", чтобы завершить работу программы.

В этом примере мы создали диалог с помощью Qt Designer и добавили некоторый код, с помощью редактора исходного кода Qt Designer-а. Интерфейсная часть диалога была сохранена в файле **gotocelldialog.ui** (по сути файл формата XML), а исходный текст -- в файле **gotocelldialog.ui.h**. Это очень удобно, поскольку **gotocelldialog.ui.h** можно править в любом текстовом редакторе.

Альтернативный подход заключается в разработке формы с помощью Qt Designer (как обычно), а затем создается дополнительный класс, порожденный от класса формы, в котором реализуется вся необходимая функциональность. Например, для нашего диалога **Go-to-Cell** можно было бы создать класс **GoToCellDialogImpl**, как наследник класса **GoToCellDialog** и реализовать в нем все необходимые функции. В результате такого подхода, новый заголовочный файл должен получиться таким:

```
#ifndef GOTOCELLDIALOGIMPL_H
#define GOTOCELLDIALOGIMPL_H
#include "gotocelldialog.h"
class GoToCellDialogImpl : public GoToCellDialog
{
    Q_OBJECT
public:
    GoToCellDialogImpl(QWidget *parent = 0, const char *name = 0);
private slots:
    void enableOkButton();
};
#endif
```

А файл с реализацией:

```
#include <qlineedit.h>
#include <qpushbutton.h>
#include <qvalidator.h>
#include "gotocelldialogimpl.h"
GoToCellDialogImpl::GoToCellDialogImpl(QWidget *parent,
                                       const char *name)
    : GoToCellDialog(parent, name)
{
    QRegExp regExp("[A-Za-z][1-9][0-9]{0,2}");
    lineEdit->setValidator(new QRegExpValidator(regExp, this));
}
void GoToCellDialogImpl::enableOkButton()
{
    okButton->setEnabled(lineEdit->hasAcceptableInput());
}
```

Некоторые разработчики, исповедующие такой подход, наверняка назвали бы базовый класс как: **GoToCellDialogBase**, а класс реализации: **GoToCellDialog**.
Создание классов-наследников может быть упрощено с помощью утилиты **uic** и набора дополнительных аргументов командной строки. Так, например, утилита **uic**, с ключом **-subdecl**, создаст скелетон заголовочного файла, а с ключом **-subimpl** -- соответствующий файл реализации.

В данной книге мы будем работать только с файлами **.ui.h**, поскольку это наиболее общеупотребимая практика, а создание дочерних классов, с помощью **uic**, довольно простая задача. Чтобы поглубже разобраться в различиях этих двух подходов, рекомендуем прочитать главу "Designer Approach" в справочном руководстве, поставляемом вместе с Qt Designer. Кроме того, прочитайте главу "**Creating Dialogs**", где показывается, как можно использовать вкладку "**Members**" для создания полей (переменных-членов) в классе формы.

2.4 Диалоги с изменяющимся внешним видом.

Мы рассмотрели примеры создания диалогов, которые никогда не меняют свой внешний вид. В некоторых случаях желательно иметь диалоги, которые могут динамически изменять свое

представление. Наиболее часто на практике встречаются расширяемые диалоги и многостраничные диалоги. Оба вида диалогов могут быть созданы как в Qt Designer, так и в результате ручного кодирования.
Расширяемые диалоги, обычно выводятся на экран в сокращенном варианте, но дают пользователю возможность выбирать между сокращенным и расширенным режимом представления информации. Расширяемые диалоги как правило используются в тех случаях, когда необходимо сокрыть дополнительные сведения, которые не являются обязательными и пользователь явно не выразил свое желание видеть их. В этом разделе мы разберем процесс создания расширяемого диалога, показанного на рисунке 2.13, с помощью Qt Designer.

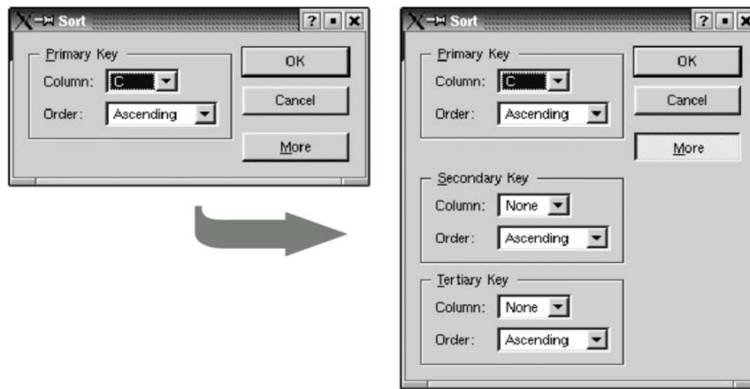


Рисунок 2.13. Диалог сортировки в простом и расширенном режимах.

Это диалог сортировки, используемый в электронных таблицах. Он появляется, когда пользователь пытается отсортировать данные по одному или нескольким столбцам. В сокращенном представлении диалог позволяет выбрать столбец и порядок сортировки, в расширенном варианте добавляется возможность указать еще два столбца и порядок сортировки по каждому из них. Кнопка **"More"** позволяет переходить из сокращенного режима -- в расширенный и обратно.
Мы создадим диалог в его расширенном виде, а потом, во время исполнения, будем скрывать дополнительные виджеты, чтобы обеспечить краткую форму диалога.

- Положите на заготовку формы **GroupBox**, два **TextLabel**, два **ComboBox** и одну горизонтальную распорку.
- "Растяните" **GroupBox** побольше, ухватив мышкой за правый нижний его угол.
- Разместите виджеты внутри **GroupBox**-а примерно так, как показано на рисунке 2.14(а)
- Ухватив мышкой за правый край второго **ComboBox**-а, сделайте его примерно в два раз больше первого.
- Запишите в свойство **title**, **GroupBox**-а, строку "**&Primary Key**". В свойство **text** первой метки -- "**Column:**", второй метки -- "**Order:**".
- Щелкните мышкой дважды по первому **ComboBox**. Перед вами появится окно редактора, в котором добавьте один элемент с текстом "**None**".
- Щелкните мышкой дважды по второму **ComboBox** и добавьте элементы "**Ascending**" и "**Descending**".
- Теперь скомпонуем виджеты внутри **GroupBox**, для этого, щелкните по **GroupBox** и выберите пункт меню **Layout|Lay Out in a Grid**. В результате вы должны получить нечто похожее на рисунок 2.14(б).

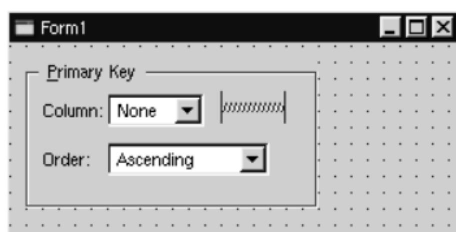


Рисунок 2.14(а). До выполнения компоновки

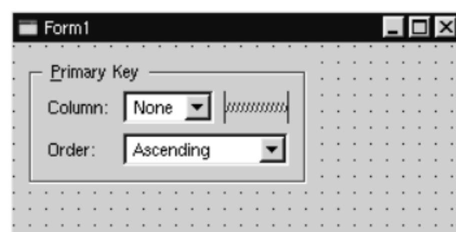


Рисунок 2.14(б). После выполнения компоновки

Если компоновка выполнилась не так как надо или вы допустили какую-нибудь ошибку, вы всегда можете выбрать пункт меню **Edit|Undo** и отменить произведенное действие. После чего можете повторить попытку.
Теперь добавим группы виджетов для расширенного представления:

- Растяните форму диалога, чтобы хватило места для дополнительных виджетов. Выберите **GroupBox** и скопируйте его в буфер обмена, выбрав пункт меню **Edit|Copy**. Вставьте новые группы виджетов, дважды выбрав пункт меню **Edit|Paste**. Переместите новые **GroupBox**-ы на свои места. Измените у них свойство **title**.
- Создайте кнопки "**OK**", "**Cancel**" и "**More**".
- Для кнопки "**OK**" установите свойство **default** в **TRUE**.
- Добавьте две вертикальные распорки.
- Расположите кнопки "**OK**", "**Cancel**" и "**More**" по вертикали. Переместите одну из распорок так, чтобы она встала между кнопками "**Cancel**" и "**More**". Выделите все четыре элемента и выберите пункт меню **Layout|Lay Out Vertically**.
- Вторую распорку переместите так, чтобы она встала между первым и вторым **GroupBox**.
- Установите свойство **sizeHint** у вертикальных распорок в значение (20, 10).
- Разместите виджеты так, как это показано нарисунке 2.15(а).
- Выберите пункт меню **Layout|Lay Out in a Grid**. У вас должна получиться заготовка, показанная на рисунке 2.15(б)

В результате такого размещения мы получили "сетку" из двух колонок и четырех строк -- всего восемь ячеек. Первый **GroupBox**, левая вертикальная распорка, второй и третий **GroupBox** занимают по одной ячейке. Кнопки "**OK**", "**Cancel**", "**More**" и правая вертикальная распорка занимают две ячейки. И в правом нижнем углу диалога у нас остаются две пустых ячейки. Если у вас не получилась такая компоновка виджетов -- отмените ее и повторите попытку.
Проверьте свойство формы **resizeMode**. Оно должно быть установлено как "**Fixed**", благодаря чему пользователь не сможет растягивать окно диалога. Тогда, всю ответственность за размер окна диалога возьмут на себя менеджеры размещения, изменяя его в случае, когда подчиненные виджеты показываются или скрываются. Это гарантирует показ окна диалога с оптимальными размерами.
Дайте форме имя "**SortDialog**" и установите свойство **caption** в "**Sort**". Дайте имена виджетам, в соответствии с рисунком 2.16.

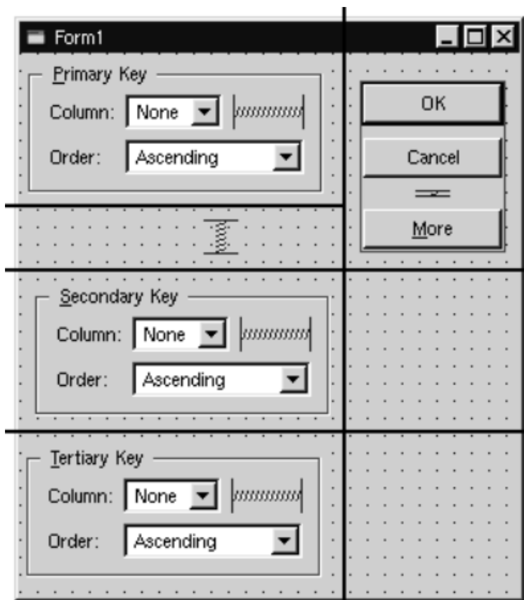


Рисунок 2.15(а). До выполнения компоновки

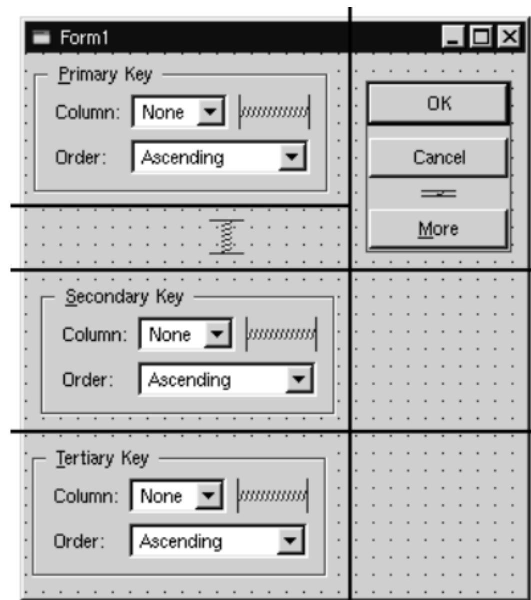


Рисунок 2.15(б). После выполнения компоновки

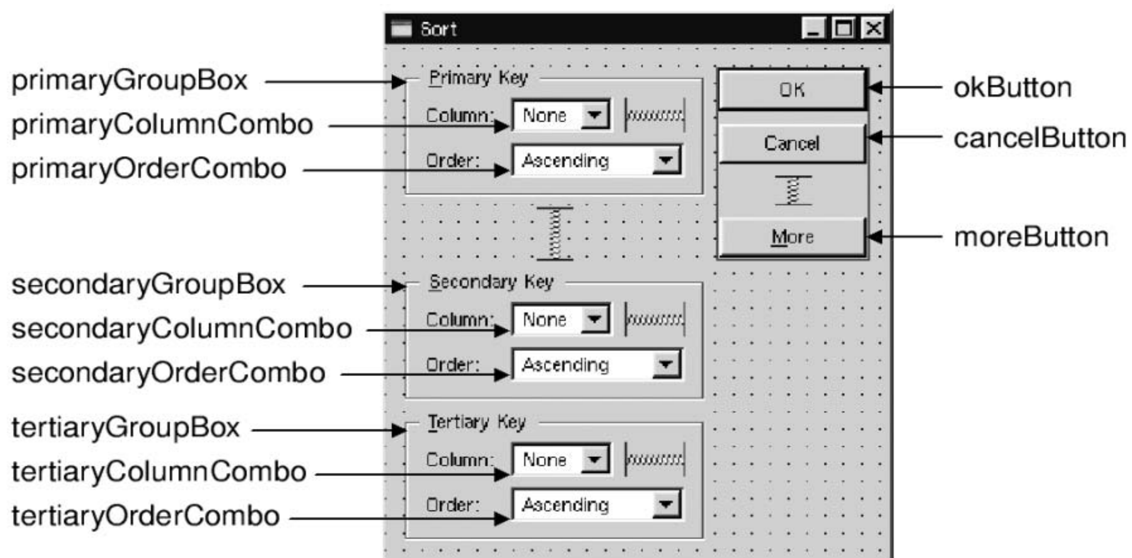


Рисунок 2.16. Имена виджетов на форме.

В заключение соединим сигналы и слоты:

- Соедините **okButton clicked()** с **SortDialog accept()**
- Соедините **cancelButton clicked()** с **SortDialog reject()**
- Соедините **moreButton toggled(bool)** с **secondaryGroupBox setShown(bool)**
- Соедините **moreButton toggled(bool)** с **tertiaryGroupBox setShown(bool)**

Щелкните мышкой по форме дважды, чтобы запустить редактор и добавьте следующий текст:

```
 1 void SortDialog::init()
 2 {
 3     secondaryGroupBox->hide();
 4     tertiaryGroupBox->hide();
 5     setColumnRange('A', 'Z');
 6 }

 7 void SortDialog::setColumnRange(QChar first, QChar last)
 8 {
 9     primaryColumnCombo->clear();
10     secondaryColumnCombo->clear();
11     tertiaryColumnCombo->clear();

12     secondaryColumnCombo->insertItem(tr("None"));
13     tertiaryColumnCombo->insertItem(tr("None"));

14     primaryColumnCombo->setMinimumSize(
15             secondaryColumnCombo->sizeHint());
16     QChar ch = first;
17     while (ch <= last) {
18         primaryColumnCombo->insertItem(ch);
19         secondaryColumnCombo->insertItem(ch);
20         tertiaryColumnCombo->insertItem(ch);
21         ch = ch.unicode() + 1;
22     }
23   }
```

Функция **init** делает второй и третий **GroupBox** невидимыми.
Слот **setColumnRange()** инициализирует содержимое выпадающих списков, в соответствии с именами выделенных колонок в электронной таблице. Мы вставили элемент "**None**", во второй и третий выпадающий списки, на тот случай, если пользователь пожелает выполнить сортировку только по одному столбцу. Не смотря на то, что мы не создавали это слот в Qt Designer, тем не менее он его обнаружит самостоятельно, а **uic** создаст соответствующее объявление в определении класса **SortDialog**.

В строках 14 и 15 можно наблюдать один хитрый трюк, связанный с размещением компонента. Функция **QWidget::sizeHint()** возвращает "идеальный" размер виджета, который пробует соблюсти система размещения. Дело в том, что виджетам с различным содержимым могут быть заданы различные размеры. Для выпадающих списков это означает, что второй и третий списки, содержащие слово "**None**", могут иметь больший размер, чем первый, в котором указано односимвольное имя столбца. Чтобы избежать такой несогласованности, мы задаем минимальный размер, для первого выпадающего списка, равный "идеальному" размеру второго.
Ниже приводится текст функции **main()**, которая устанавливает диапазон выделенных столбцов от "**C**" до "**F**" и затем вызывает диалог:

```
#include <qapplication.h>
#include "sortdialog.h"
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    SortDialog *dialog = new SortDialog;
    app.setMainWidget(dialog);
    dialog->setColumnRange('C', 'F');
    dialog->show();
    return app.exec();
}
```

На этом мы завершаем рассмотрение расширяемого диалога. Из примера видно, что разработка расширяемых диалогов ненамного сложнее, чем обычных. Все что нам потребовалось добавить -- это кнопка перехода из режима в режим, несколько дополнительных сигналов и слотов, и фиксированный размер формы.

Другой тип диалогов, изменяющих свое представление -- это многостраничные диалоги. Создание многостраничных диалогов проходит еще проще. Такого рода диалоги можно строить несколькими способами:

- В качестве основы можно использовать **QTabWidget**. Сверху он имеет набор вкладок, которые управляются встроенным **QWidgetStack**.
- Можно использовать связку **QListBox** и **QWidgetStack**, в которой текущий элемент **QListBox**-а определяет страницу в **QWidgetStack**.
- Или связку из классов **QListView** или **QIconView** и **QWidgetStack**, объединяемые так же как и в случае с **QListBox**.

Класс **QWidgetStack** обсуждается в Главе 6.

2.5 Динамические диалоги

Динамическими называются такие диалоги, которые создаются на основе **.ui** файлов во время работы программы (то бишь "на лету"). В случае с динамическими диалогами, файлы **.ui** не конвертируются утилитой **uic** в код C++, а загружаются программой во время исполнения, с помощью класса **QWidgetFactory**, примерно таким образом:

```
QDialog *sortDialog = (QDialog *)
                      QWidgetFactory::create("sortdialog.ui");
```

Доступ к виджетам на форме такого диалога осуществляется вызовом функции **QObject::child()**:

```
QComboBox *primaryColumnCombo = (QComboBox *)
        sortDialog->child("primaryColumnCombo", "QComboBox");
```

Эта функция возвращает "пустой" указатель, если на форме диалога не был найден виджет с заданным именем и типом.

Класс **QWidgetFactory** размещен в отдельной библиотеке. Чтобы иметь возможность работы с классом **QWidgetFactory** из Qt приложений, необходимо добавить такую строчку в **.pro** файл:

```
LIBS += -lqui
```

Этот синтаксис одинаков для любой платформы, даже при том, что он определенно имеет отношение к Unix.

Динамические диалоги позволяют изменять компоновку формы без необходимости пересборки приложения. Исчерпывающий пример работы с динамическими диалогами вы найдете в главе "Subclassing and Dynamic Dialogs" справочного руководства к Qt Designer.

2.6 Встроенные виджеты и классы диалогов

Qt предоставляет полный набор встроенных виджетов и диалогов общего назначения, которые подойдут в большинстве ситуаций. В этом разделе мы представим изображения большинства из них. Некоторые из специализированных визуальных компонентов будут рассматриваться ниже, в Главе 3 (**QMenuBar**, **QPopupMenu** и **QToolBar**) и в Главе 12 (компоненты для работы с базами данных, такие как **QDataView** и **QDataTable**). Большинство из встроенных виджетов и диалогов будут использоваться в примерах программ в данной книге. Ниже представлены скриншоты виджетов в классическом Windows-стиле.

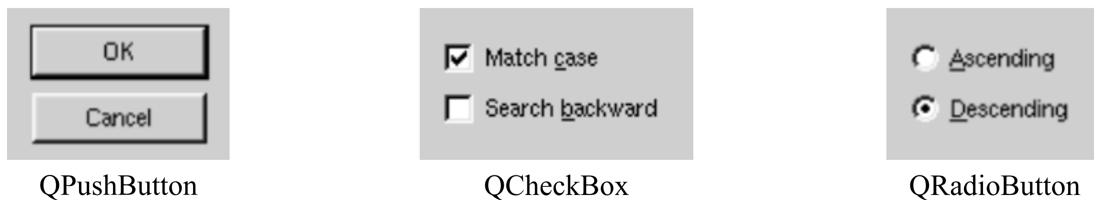


Рисунок 2.17 Кнопки

В Qt имеется три вида "кнопок": **QPushButton**, **QCheckBox** и **QRadioButton**. Кнопка типа **QPushButton** обычно используется для того, чтобы инициировать какое-либо действие. Может работать как кнопка с фиксацией (первый щелчок, чтобы нажать, второй -- чтобы отпустить). Кнопки типа **QRadioButton** обычно используются совместно с **QButtonGroup** и работают как группа кнопок с взаимозависимой фиксацией, т.е. в каждой группе кнопок, в нажатом состоянии может находиться только одна из них. Кнопки типа **QCheckBox**, в отличие от **QRadioButton**, работают как кнопки с взаимонезависимой фиксацией.

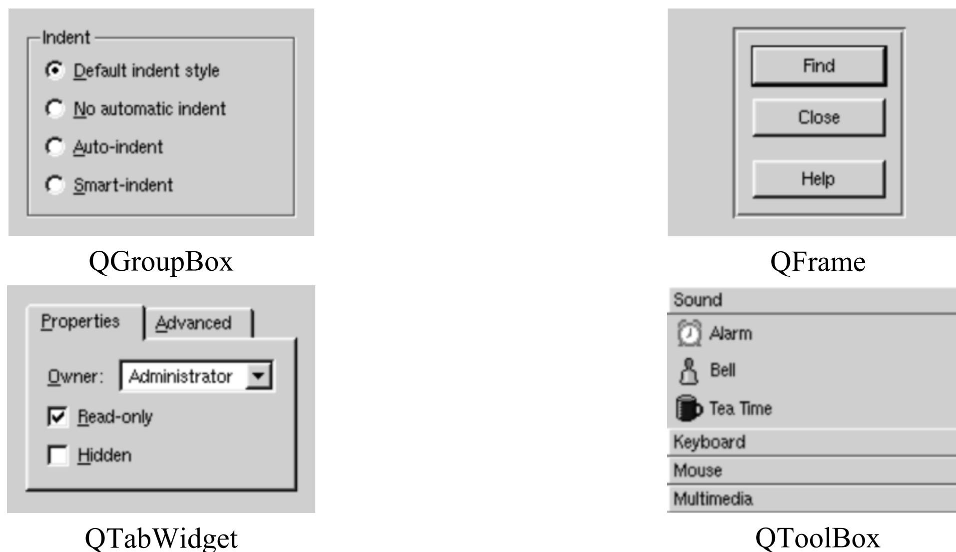
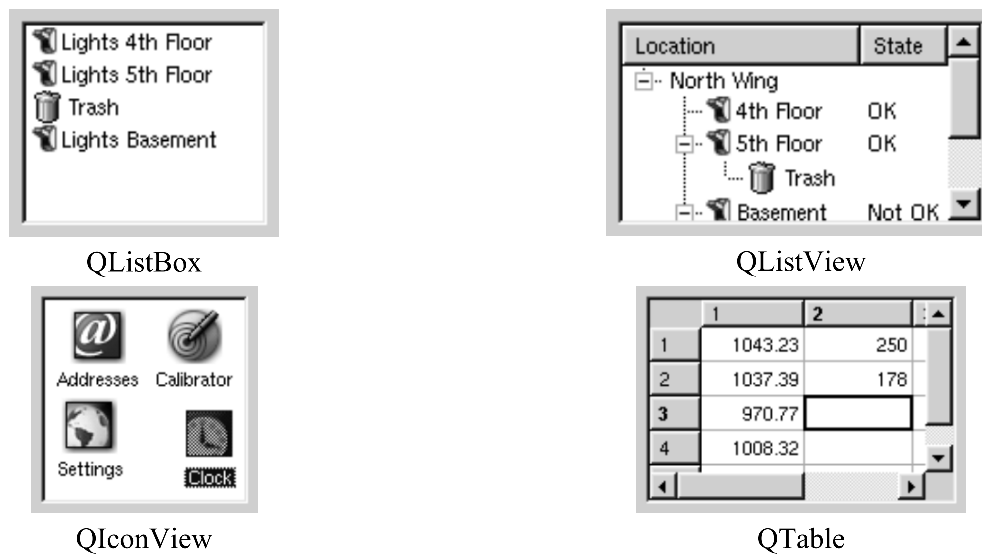


Рисунок 2.18 Контейнеры

Контейнеры в Qt -- это виджеты, которые могут содержать в себе другие виджеты. Кроме того, контейнер **QFrame** может использоваться как самостоятельный виджет, для рисования линий. Здесь не показан контейнер **QButtonGroup**, визуально он идентичен контейнеру **QGroupBox**. Контейнеры **QTabWidget** и **QToolBox** -- это многостраничные виджеты. Каждая страница -- это подчиненный виджет. Нумерация страниц начинается с 0.

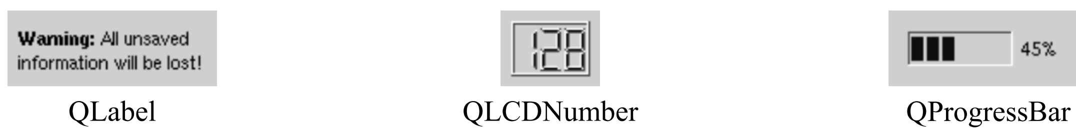


QListBox QListView

QIconView QTable

Рисунок 2.19 Списки элементов.

Списки элементов оптимизированы для работы с большими объемами данных и часто снабжаются полосами прокрутки. Полосы прокрутки реализованы в классе **QScrollView**, являющимся базовым для списков элементов и другого типа виджетов.

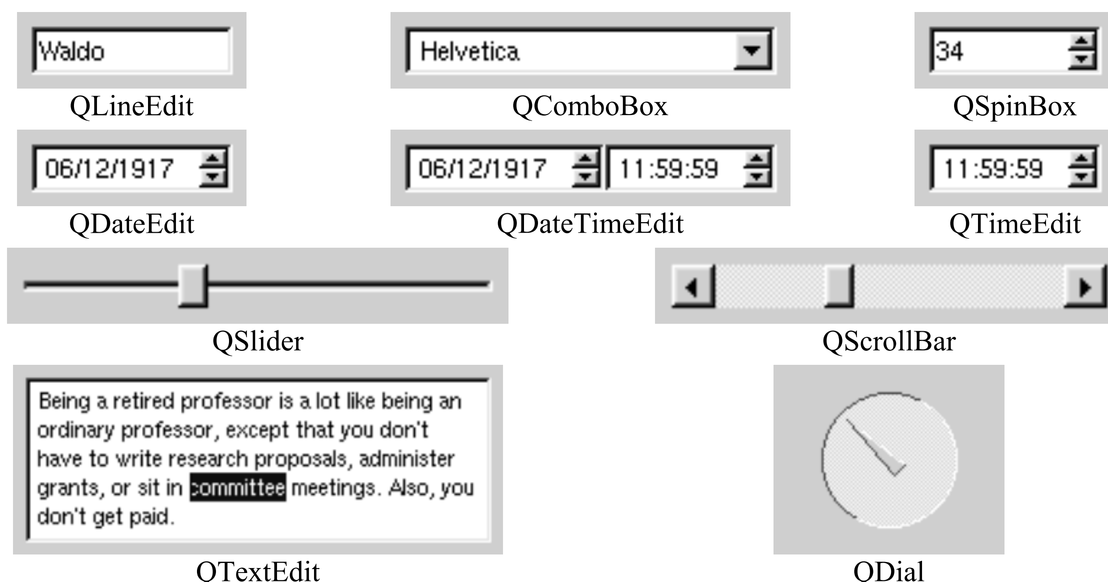


QLabel QLCDNumber QProgressBar

Рисунок 2.20 Виджеты отображения информации.

Виджет **QLabel** используется для вывода надписей в форматированном виде (с использованием простейших тегов HTML)

Виджет **QTextBrowser** (здесь не приводится), порожден от **QTextEdit** и предназначен для вывода текста в режиме **read-only** (только для чтения). Обладает поддержкой синтаксиса HTML, которая включает в себя поддержку списков, таблиц, изображений и гиперссылок. Qt Assistant использует этот виджет для вывода текста документации.



QLineEdit QComboBox QSpinBox

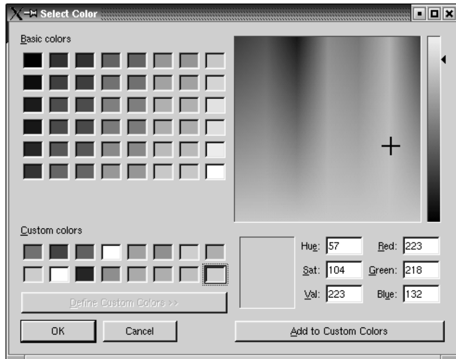
QDateEdit QDateTimeEdit QTimeEdit

QSlider QScrollBar

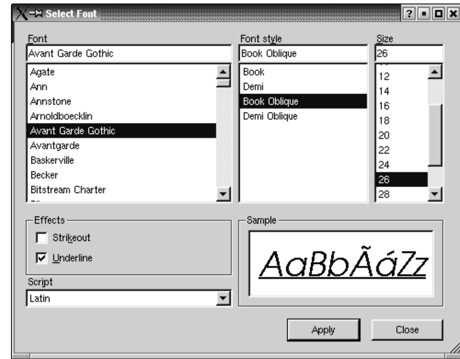
QTextEdit QDial

Рисунок 2.21 Виджеты ввода информации.

Виджет **QLineEdit** обладает возможностью наложения ограничений на вводимые символы с помощью маски ввода или проверки корректности ввода (**validator**). Виджет **QTextEdit**, наследник класса **QScrollView**, обладает возможностью редактирования текста большого объема.



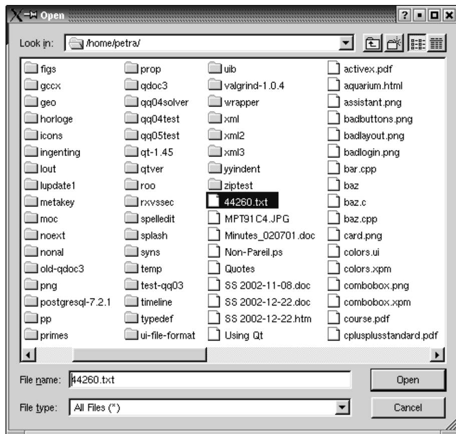
QColorDialog



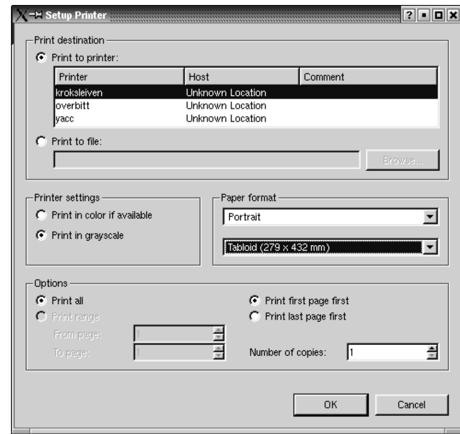
QFontDialog

Рисунок 2.22 Диалоги выбора цвета и шрифта.

В Qt имеется целый набор стандартных, наиболее употребимых диалогов, которые предоставляют пользователю возможность выбрать цвет, шрифт, файл или принтер.



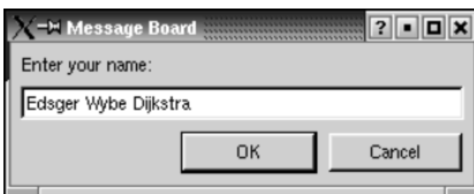
QFileDialog



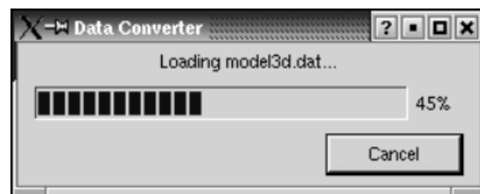
QPrintDialog

Рисунок 2.23 Диалоги выбора файла и принтера.

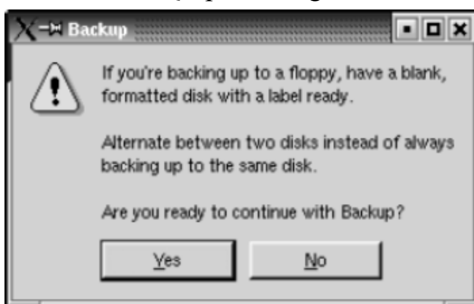
В операционных системах Windows и Mac OS X, по мере возможности, используются "родные" диалоги.



QInputDialog



QProgressDialog



QMessageBox

QErrorMessage

Рисунок 2.24 Диалоги обратной связи с пользователем

В Qt так же имеется целый ряд диалогов обратной связи с пользователем. Диалоги **QMessageBox** и **QErrorMessage** используются для вывода сообщений, причем последний запоминает -- выводилось ли это сообщение ранее. Операции, протяженные по времени, могут быть оформлены в виде **QProgressDialog**, показывающего ход выполнения работы. Для того, чтобы запросить у пользователя ввод одной строки или числа, очень удобно использовать **QInputDialog**.
И последний диалог, который мы покажем -- это **QWizard**, своего рода каркас для создания разного рода "мастеров".

Рисунок 2.25. QWizard.

Библиотека Qt предоставляет широкий выбор виджетов и диалогов общего назначения. Очень часто специфические требования к диалогам могут быть удовлетворены за счет соединения сигналов и слотов и реализации своих собственных слотов в программе.
Иногда программист сталкивается с необходимостью писать свои собственные виджеты. В Qt это делается легко и просто и вашим виджетам будет доступна вся функциональность, которая доступна и обычным встроенным виджетам. Они даже могут быть интегрированы в Qt Designer, таким образом у вас есть возможность пользоваться ими так же как и встроенными визуальными компонентами. Более подробно, проблема разработки своих собственных визуальных компонентов будет рассмотрена в Главе 5.

3 Создание главного окна приложения

В этой главе мы рассмотрим процесс создания главного окна приложения. По прочтении ее, вы сможете построить интерфейс приложения, который будет содержать меню, панели инструментов, строку состояния и набор дополнительных диалогов.

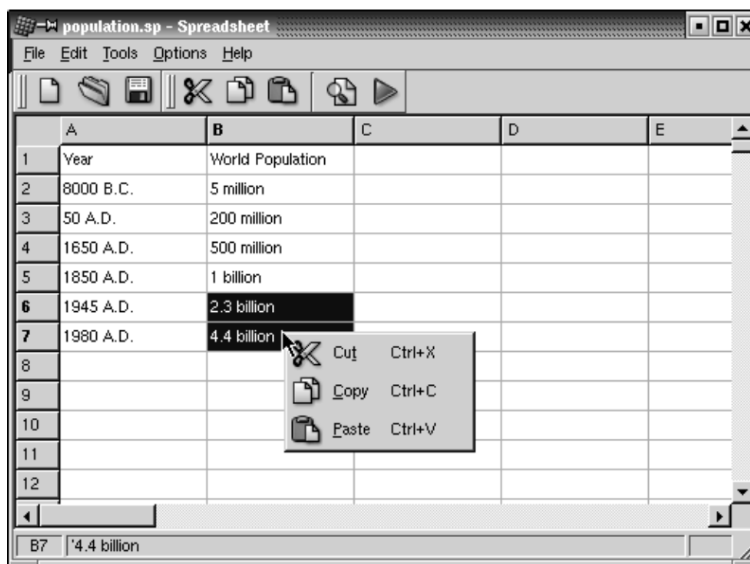


Рисунок 3.1. Приложение -- электронная таблица.

Главное окно -- это своего рода каркас, на который "натягивается" весь пользовательский интерфейс приложения. Здесь мы рассмотрим пример создания главного окна электронной таблицы. Внешний вид нашего будущего приложения приводится на рисунке 3.1. В этом приложении мы будем использовать диалоги "**Find**", "**Go-to-Cell**" и "**Sort**", которые были созданы нами в Главе 2.
Внутри большинства приложений с графическим интерфейсом "прячется" код, который обеспечивает основные функциональные возможности программы, например, работа с файлами или обработка данных, представляемых пользовательским интерфейсом. В Главе 4 мы покажем -- как реализуется такого рода функциональность, на примере нашей электронной таблицы.

3.1 Создание класса-наследника от QMainWindow

Главное окно любого приложения -- это класс-наследник **QMainWindow**. Большинство приемов, используемых при создания диалогов и о которых мы говорили в Главе 2, вполне применимы и при создании главного окна приложения.
Главное окно может быть создано в Qt Designer, но мы все будем делать "вручную", чтобы продемонстрировать процесс создания главного окна во всех деталях. Если вы предпочитаете визуальное проектирование -- прочитайте главу "Creating a Main Window Application" в справочном руководстве к Qt Designer.

Исходные тексты главного окна будут располагаться в двух файлах: **mainwindow.cpp** и **mainwindow.cpp**. Начнем с файла заголовка:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <qmainwindow.h>
#include <qstringlist.h>
class QAction;
class QLabel;
class FindDialog;
class Spreadsheet;

class MainWindow : public QMainWindow
{
  Q_OBJECT
public:
  MainWindow(QWidget *parent = 0, const char *name = 0);
```

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

Это определение класса **MainWindow** -- наследника **QMainWindow**. Оно содержит макрос **Q_OBJECT**, поскольку реализует свои собственные сигналы и слоты.

Функция **closeEvent()**, в классе **QWidget**, объявлена как виртуальная. Она автоматически вызывается перед завершением приложения. Мы перекрываем ее в **MainWindow** для того, чтобы иметь возможность спросить у пользователя -- желает ли он сохранить произведенные изменения, а также для того, чтобы сохранить на диск пользовательские настройки.
Аналогично, функция **contextMenuEvent()** вызывается, когда пользователь щелкает правой кнопкой мыши по виджету. Мы перекрываем ее в **MainWindow** для того, чтобы вывести контекстное меню.

```
private slots:
  void newFile();
  void open();
  bool save();
  bool saveAs();
  void find();
  void goToCell();
  void sort();
  void about();
```

Реализация действий некоторых пунктов меню, таких как **File|New** и **Help|About**, выполнена в виде приватных слотов **MainWindow**. Большинство слотов имеют тип **void**, но слоты **save()** и **saveAs** возвращают результат типа **bool**. Значение, возвращаемое слотом, игнорируется в случае вызова по сигналу, но когда слот вызывается как обычная функция, то мы получаем от него возвращаемое значение, которое можем использовать для своих нужд.

```
  void updateCellIndicators();
  void spreadsheetModified();
  void openRecentFile(int param);

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

Дополнительные приватные функции, необходимые для обслуживания пользовательского интерфейса.

```
  Spreadsheet *spreadsheet;
  FindDialog *findDialog;
  QLabel *locationLabel;
  QLabel *formulaLabel;
  QLabel *modLabel;
  QStringList recentFiles;
  QString curFile;
  QString fileFilters;
  bool modified;

  enum { MaxRecentFiles = 5 };
  int recentFileIds[MaxRecentFiles];

  QPopupMenu *fileMenu;
  QPopupMenu *editMenu;
```

```
  QPopupMenu *selectSubMenu;
  QPopupMenu *toolsMenu;
  QPopupMenu *optionsMenu;
  QPopupMenu *helpMenu;
  QToolBar *fileToolBar;
  QToolBar *editToolBar;
  QAction *newAct;
  QAction *openAct;
  QAction *saveAct;
  ...
  QAction *aboutAct;
  QAction *aboutQtAct;
};
#endif
```

Кроме функций, класс главного окна имеет ряд скрытых переменных. Все они будут описаны по мере необходимости.
Теперь перейдем к реализации:

```
#include <qaction.h>
#include <qapplication.h>
#include <qcombobox.h>
#include <qfiledialog.h>
#include <qlabel.h>
#include <qlineedit.h>
#include <qmenubar.h>
#include <qmessagebox.h>
#include <qpopupmenu.h>
#include <qsettings.h>
#include <qstatusbar.h>

#include "cell.h"
#include "finddialog.h"
#include "gotocelldialog.h"
#include "mainwindow.h"
#include "sortdialog.h"
#include "spreadsheet.h"
```

Здесь подключаются заголовки всех классов Qt, которые используются в приложении, а также заголовок класса главного окна и ряд других заголовочных файлов, таких как **finddialog.h**, **gotocelldialog.h** и **sortdialog.h**, которые мы создали в предыдущей главе.

```
  MainWindow::MainWindow(QWidget *parent, const char *name)
    : QMainWindow(parent, name)
  {
    spreadsheet = new Spreadsheet(this);
    setCentralWidget(spreadsheet);
    createActions();
    createMenus();
    createToolBars();
    createStatusBar();
    readSettings();
    setCaption(tr("Spreadsheet"));
    setIcon(QPixmap::fromMimeSource("icon.png"));
    findDialog = 0;
    fileFilters = tr("Spreadsheet files (*.sp)");
    modified = false;
  }
```

Конструктор начинается с создания виджета **Spreadsheet**, который будет центральным виджетом главного окна. Центральный виджет занимает все пространство, находящееся между панелью инструментов (**toolbar**) и строкой состояния (**statusbar**). Класс **Spreadsheet** является потомком класса **QTable** и добавляет некоторые свойства, характерные для электронных таблиц. Среди них можно назвать поддержку формул, которая будет реализована в Главе 4.

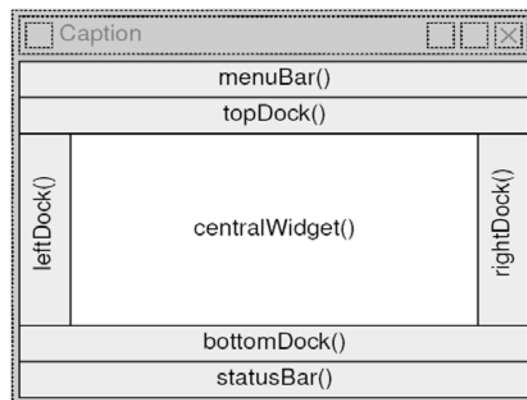


Рисунок 3.2. Раскладка виджетов в главном окне.

Далее вызываются приватные функции **createActions()**, **createMenus()**, **createToolBars()** и **createStatusBar()**, которые создают остальную часть главного окна. Для восстановления пользовательских настроек вызывается функция **readSettings()**.

В качестве иконки приложения устанавливается **icon.png**. Qt поддерживает различные форматы графических файлов, включая **BMP**, **GIF** [6] , **JPEG**, **MNG**, **PNG**, **PNM**, **XBM** и **XPM**. Вызов **QWidget::setIcon()** выводит иконку в верхний левый угол окна. К сожалению, отсутствует платформо-независимый способ помещения иконки на рабочий стол.

Программы с графическим интерфейсом обычно используют достаточно большое число изображений. Qt, в свою очередь, предоставляет значительное число методов работы с изображениями в приложениях. Наиболее употребимые из них:

- Сохранение изображений в файлы и загрузка из файлов, в процессе работы приложения.
- Включение изображений формата XPM в исходный код.
- Механизм "коллекции изображений" ("image collection").

В данном примере мы будем использовать механизм "коллекции изображений", поскольку работать с ним намного проще, чем загружать файлы в процессе исполнения. К тому же он может взаимодействовать со всеми, поддерживаемыми библиотекой, графическими форматами. Все изображения мы будем хранить в каталоге **images**. Чтобы создать файл на языке C++ (он будет создан утилитой **uic**), который будет хранить наши изображения, добавим следующие строки в файл **.pro**:

```
IMAGES = images/icon.png \
         images/new.png \
         images/open.png \
         ...
         images/find.png \
         images/gotocell.png
```

Изображения будут помещены в исполняемый файл приложения и могут быть получены вызовом **QPixmap::fromMimeSource()**. Преимущество такого способа организации хранения изображений заключается в том, что они никогда не потеряются, поскольку находятся внутри исполняемого файла. Если главное окно создается в Qt Designer, то вы можете использовать визуальные средства, предоставляемые построителем, для вставки изображений в коллекцию.

3.2 Создание меню и панелей инструментов

Большинство современных приложений имеют как меню, так и панели инструментов, которые содержат более или менее идентичный набор команд. Меню дают пользователю возможность заняться "исследованием" приложения, изучать доступные команды, а панели инструментов служат для быстрого доступа к командам, используемым наиболее часто.

Qt значительно упрощает разработку меню и панелей инструментов за счет использования концепции "действия" (**action**). **Action** -- это элемент, который может быть добавлен в меню и/или на панель инструментов. Для создания меню и панели инструментов необходимо выполнить следующие шаги:

- Создать "действие" (action).
- Добавить его в меню.
- Добавить его на панель инструментов.

В нашем приложении все необходимое выполняет функция **createActions()**:

```
void MainWindow::createActions()
{
  newAct = new QAction(tr("&New"), tr("Ctrl+N"), this);
  newAct->setIconSet(QPixmap::fromMimeSource("new.png"));
  newAct->setStatusTip(tr("Create a new spreadsheet file"));
  connect(newAct, SIGNAL(activated()), this, SLOT(newFile()));
```

В данном случае создается новое "действие" с названием "**New**", горячей комбинацией клавиш **Ctrl+N** и с владельцем -- главным окном приложения. Затем к "действию" прицепляются иконка (**new.png**) и текст подсказки, который будет выводиться в строке состояния. В заключение -- сигнал **activated()** подключается к слоту главного окна **newFile()**, который будет описан в следующем разделе. Без этого соединения, при выборе пункта меню "**File|New**" или при нажатии на кнопку "**New**" (в панели инструментов), ничего происходить не будет.
Аналогичным образом создаются все остальные "действия" (**action**).

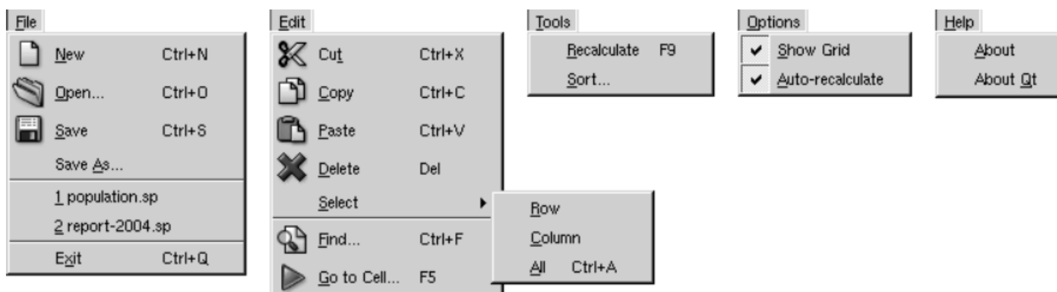


Рисунок 3.3. Меню приложения Spreadsheet.

Команда (action) "**Show Grid**", в меню "**Options**" реализуется несколько иначе:

```
  showGridAct = new QAction(tr("&Show Grid"), 0, this);
  showGridAct->setToggleAction(true);
  showGridAct->setOn(spreadsheet->showGrid());
  showGridAct->setStatusTip(tr("Show or hide the spreadsheet s " "grid"));
  connect(showGridAct, SIGNAL(toggled(bool)), spreadsheet, SLOT(setShowGrid(bool)));
```

Эта команда имеет два фиксированных состояния -- включено-выключено. В меню рядом с ней отображается галочка (когда включено), а на панели инструментов она выглядит как кнопка с фиксацией. Когда "действие" включено, то компонент **Spreadsheet** отображается в окне приложения. Мы инициализируем "действие" значением по-умолчанию -- включено. Таким образом выполняется начальная синхронизация "действия" с фактическим режимом отображения компонента **Spreadsheet**. Затем мы подключаем сигнал **toggled(bool)** к слоту **setShowGrid(bool)**, компонента **Spreadsheet**. После этого "действие" (**action**) добавляется в меню или на панель инструментов. Теперь пользователь может "включать" и "выключать" таблицу.

Команды "**Show Grid**" и "**Auto-recalculate**" -- являются "действиями" с независимой фиксацией. Однако, **QAction** имеет наследника -- **QActionGroup**, с помощью которого можно создавать группы "действий" с зависимой фиксацией.

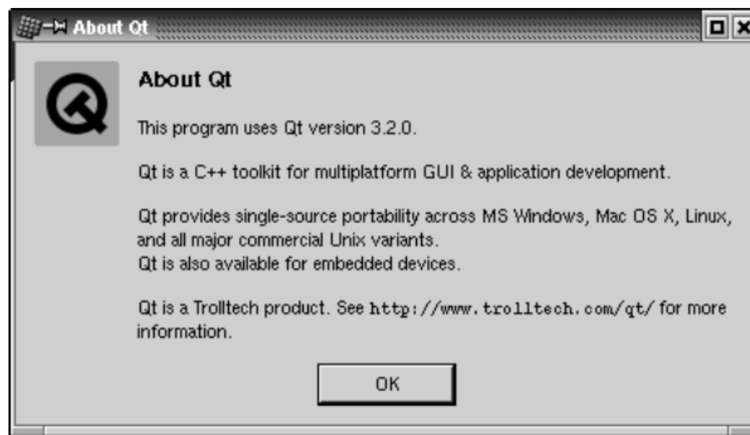


Рисунок 3.4. About Qt.

```
  aboutQtAct = new QAction(tr("About &Qt"), 0, this);
  aboutQtAct->setStatusTip(tr("Show the Qt library's About box"));
  connect(aboutQtAct, SIGNAL(activated()), qApp, SLOT(aboutQt()));
}
```

Для вызова диалога "**AboutQt**" используется слот **aboutQt()** глобальной переменной **qApp** -- экземпляр класса **QApplication**.
После создания всех "действий" (**action**) мы можем разместить их в меню:

```
void MainWindow::createMenus()
{
  fileMenu = new QPopupMenu(this);
  newAct->addTo(fileMenu);
  openAct->addTo(fileMenu);
  saveAct->addTo(fileMenu);
  saveAsAct->addTo(fileMenu);
  fileMenu->insertSeparator();
  exitAct->addTo(fileMenu);

  for (int i = 0; i < MaxRecentFiles; ++i)
    recentFileIds[i] = -1;
```

В Qt все меню являются экземплярами **QPopupMenu**. Мы создали меню "**File**" и затем добавили в него пункты "**New**", "**Open**", "**Save**", "**Save As**" и "**Exit**". Перед пунктом "**Exit**" добавлен разделитель, чтобы визуально отделить его от остальных. Цикл **for** инициализирует **recentFileIds** -- массив файлов, открывавшихся недавно. Мы еще вернемся к этому массиву, когда приступим к рассмотрению реализации слотов меню "**File**" в следующем разделе.

```
  editMenu = new QPopupMenu(this);
  cutAct->addTo(editMenu);
  copyAct->addTo(editMenu);
  pasteAct->addTo(editMenu);
  deleteAct->addTo(editMenu);

  selectSubMenu = new QPopupMenu(this);
  selectRowAct->addTo(selectSubMenu);
  selectColumnAct->addTo(selectSubMenu);
  selectAllAct->addTo(selectSubMenu);
  editMenu->insertItem(tr("&Select"), selectSubMenu);

  editMenu->insertSeparator();
  findAct->addTo(editMenu);
  goToCellAct->addTo(editMenu);
```

Меню "**Edit**" включает в себя подменю, которое так же является экземпляром класса **QPopupMenu**. Мы просто создаем подменю и вставляем его в то место меню "**Edit**", где оно должно находиться.

```
  toolsMenu = new QPopupMenu(this);
  recalculateAct->addTo(toolsMenu);
  sortAct->addTo(toolsMenu);
```

```
  optionsMenu = new QPopupMenu(this);
  showGridAct->addTo(optionsMenu);
  autoRecalcAct->addTo(optionsMenu);

  helpMenu = new QPopupMenu(this);
  aboutAct->addTo(helpMenu);
  aboutQtAct->addTo(helpMenu);

  menuBar()->insertItem(tr("&File"), fileMenu);
  menuBar()->insertItem(tr("&Edit"), editMenu);
  menuBar()->insertItem(tr("&Tools"), toolsMenu);
  menuBar()->insertItem(tr("&Options"), optionsMenu);
  menuBar()->insertSeparator();
  menuBar()->insertItem(tr("&Help"), helpMenu);
}
```

Меню "**Tools**", "**Options**" и "**Help**" создаются аналогичным образом и в конце, все созданные меню вставляются в полосу меню, в верхней части главного окна приложения. Функция **QMainWindow::menuBar()** возвращает указатель на экземпляр класса **QMenuBar**, который создается автоматически, при первом вызове **menuBar()**. Мы добавили разделитель между меню "**Options**" и "**Help**". В случае отображения в стиле **Motif** и ему подобных, меню "**Help**" смещается в крайнее правое положение, в других стилях отображения разделитель игнорируется.

Рисунок 3.5. Полоса меню в стиле Motif и Windows.

Создание панелей инструментов происходит очень похожим образом:

```
void MainWindow::createToolBars()
{
  fileToolBar = new QToolBar(tr("File"), this);
  newAct->addTo(fileToolBar);
  openAct->addTo(fileToolBar);
  saveAct->addTo(fileToolBar);

  editToolBar = new QToolBar(tr("Edit"), this);
  cutAct->addTo(editToolBar);
  copyAct->addTo(editToolBar);
  pasteAct->addTo(editToolBar);
  editToolBar->addSeparator();
  findAct->addTo(editToolBar);
  goToCellAct->addTo(editToolBar);
}
```

Мы создали две панели инструментов -- "**File**" и "**Edit**". Как и меню, панели инструментов могут включать в себя разделители.

Рисунок 3.6. Панели инструментов приложения Spreadsheet

Теперь, когда мы закончили создание главного меню и панелей инструментов, добавим контекстное меню.

```
void MainWindow::contextMenuEvent(QContextMenuEvent *event)
{
  QPopupMenu contextMenu(this);
  cutAct->addTo(&contextMenu);
  copyAct->addTo(&contextMenu);
  pasteAct->addTo(&contextMenu);
  contextMenu.exec(event->globalPos());
}
```

Когда пользователь щелкает правой кнопкой мыши, то виджету посылается событие (**event**) "контекстное меню". Перекрывая метод **QWidget::contextMenuEvent()**, мы можем перехватить это событие и показать контекстное меню в позиции курсора мыши.

Рисунок 3.7. Контекстное меню приложения Spreadsheet.

События, точно так же как сигналы и слоты, являются одним из фундаментальных аспектов программирования в Qt. События рождаются в недрах библиотеки, чтобы сообщить о щелчках мышью, о нажатиях клавиш, о попытках изменения размеров и т.п. Они могут быть перехвачены и обработаны за счет перекрытия виртуальных функций, как это делается в нашем случае.
Отлавливать событие мы будем в **MainWindow** потому, что здесь реализуются все наши "действия" (**actions**). Однако, это событие можно поймать и в виджете **Spreadsheet**. Когда пользователь щелкнет правой кнопкой мыши по виджету, то этот виджет первым получит событие. Если виджет перекроет реализацию функции **contextMenuEvent()** и обработает событие, то дальше передаваться оно не будет. В противном случае оно будет передано дальше -- владельцу виджета (**MainWindow**). Более подробно события будут рассматриваться в Главе 7.
Функция-обработчик события вызова контекстного меню отличается от всего, что мы до сих пор видели, поскольку она создает экземпляр **QPopupMenu**, размещая его на стеке. Хотя, в принципе, можно было бы создать/удалить этот виджет и с помощью операторов **new/delete**:

```
  QPopupMenu *contextMenu = new QPopupMenu(this);
  cutAct->addTo(contextMenu);
  copyAct->addTo(contextMenu);
  pasteAct->addTo(contextMenu);
  contextMenu->exec(event->globalPos());
  delete contextMenu;
```

Еще один примечательный аспект -- это функция **exec()**. Она выводит меню на экран, в заданную позицию, и ожидает, пока пользователь не сделает свой выбор, после чего управление возвращается в точку вызова. С этого момента экземпляр **QPopupMenu** нам больше не нужен, поэтому он удаляется. В случае размещения на стеке -- он будет уничтожен автоматически, по завершении работы функции.
Часть интерфейса, касающуюся меню и панелей инструментов, можно считать завершенной. В следующем разделе мы рассмотрим реализацию слотов меню "**File**".

3.3 Реализация меню "File".

В этом разделе мы рассмотрим реализацию всех слотов меню "File".

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

Слот **newFile()** вызывается, когда пользователь выбирает пункт меню "**File|New**" или щелкает по кнопке "**New**" на панели инструментов. Функция **maybeSave()** спрашивает пользователя: "Do you want to save your changes?" ("Желаете ли сохранить изменения?"), если файл был изменен. Она возвращает **true**, если пользователь ответил "**Yes**" или "**No**" (в случае ответа "**Yes**" -- файл сохраняется), и **false** -- если пользователь нажал на кнопку "**Cancel**" ("Отмена"). Приватная функция **setCurrentFile()** обновляет заголовок окна программы, показывая, что редактируется неозаглавленный документ.

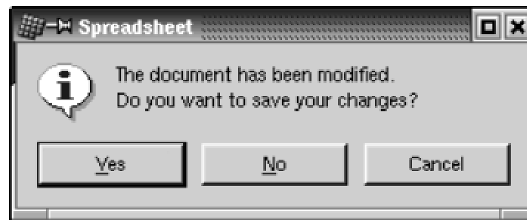


Рисунок 3.8. Запрос: "Do you want to save your changes?"

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

Функция **maybeSave()** выводит перед пользователем диалоговое окно с запросом (см. рис. 3.8). Диалог имеет три кнопки -- три варианта ответа: "**Yes**", "**No**" и "**Cancel**". Модификатор **QMessageBox::Default** назначает кнопку "**Yes**" -- кнопкой по-умолчанию. Модификатор **QMessageBox::Escape** связывает кнопку "**No**" с клавишей **Esc**.

Вызов **QMessageBox::warning()** может показаться на первый взгляд немного не понятным. Синтаксис этого метода:

```
QMessageBox::warning(parent, caption, messageText,
                     button0, button1, ...);
```

Класс **QMessageBox** имеет еще ряд аналогичных методов: **information()**, **question()** и **critical().** Все они отображают диалоговое окно с различными иконками.

Рисунок 3.9. Иконки диалога запроса

```
void MainWindow::open()
{
  if (maybeSave()) {
    QString fileName =
            QFileDialog::getOpenFileName(".", fileFilters, this);
    if (!fileName.isEmpty())
      loadFile(fileName);
  }
}
```

Слот **open()** соответствует пункту меню "**File|Open**". Аналогично слоту **newFile()** -- сначала вызывается функция **maybeSave()**, чтобы сохранить имеющиеся изменения. Затем, с помощью функции **QFileDialog::getOpenFileName()**, у пользователя запрашивается имя открываемого файла. Она выводит перед пользователем диалоговое окно, которое предлагает выбрать требуемый файл и возвращает программе его имя или пустую строку, если пользователь отменил операцию открытия файла. Функции **getOpenFileName()** передаются три аргумента. Первый аргумент -- это каталог, где может находиться файл, в нашем случае -- это текущий каталог. Второй аргумент -- **fileFilters**, задет фильтр имен файлов. Фильтр состоит из двух частей -- текста описания и шаблона. В конструкторе **MainWindow** фильтр был инициализирован так:

```
  fileFilters = tr("Spreadsheet files (*.sp)");
```

Если бы наша программа дополнительно поддерживала файлы форматов **CSV** и **Lotus 1-2-3**, то фильтр имен файлов мог бы быть инициализирован следующим образом:

```
  fileFilters = tr("Spreadsheet files (*.sp)\n"
                   "Comma-separated values files (*.csv)\n"
                   "Lotus 1-2-3 files (*.wk?)");
```

И наконец третий аргумент указывает, что окно диалога является подчиненным, по отношению к главному окну приложения.
Для диалоговых окон, отношение "владелец-подчиненный", носит иной смысл, чем для виджетов. Диалоговое окно всегда отображает поверх других окон, но если оно имеет владельца, то центрируется относительно него. Этот же диалог вызывается по нажатию на кнопку "**Open**", на панели инструментов.

```
void MainWindow::loadFile(const QString &fileName)
{
  if (spreadsheet->readFile(fileName)) {
    setCurrentFile(fileName);
    statusBar()->message(tr("File loaded"), 2000);
  } else {
    statusBar()->message(tr("Loading canceled"), 2000);
  }
}
```

Функция **loadFile()** вызывается из **open()** для загрузки файла. Мы вынесли операцию загрузки файла в отдельную функцию, потому что она потребуется нам при реализации слота, открывающего недавно использовавшиеся файлы.

Непосредственное чтение файла с диска выполняется в функции **Spreadsheet::readFile()**. Если чтение прошло без ошибок, то вызывается **setCurrentFile()**, чтобы обновить заголовок окна. В противном случае **readFile()** выведет окно с сообщением об ошибке. Обычно, считается хорошей практикой давать возможность низкоуровневым компонентам выводить свои сообщения, поскольку в этом случае диагностика ошибок может быть выполнена более точно.
В обоих случаях, в строку состояния выводится сообщение, которое демонстрируется 2000 миллисекунд (2 секунды).

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

void MainWindow::saveFile(const QString &fileName)
{
  if (spreadsheet->writeFile(fileName)) {
    setCurrentFile(fileName);
    statusBar()->message(tr("File saved"), 2000);
  } else {
    statusBar()->message(tr("Saving canceled"), 2000);
  }
}
```

Слот **save()** соответствует пункту меню "**File|Save**". Если файлу ранее уже было назначено имя, то он сохраняется вызовом **saveFile()**, иначе вызывается **saveAs()**.

```
bool MainWindow::saveAs()
{
  QString fileName =
```

```
          QFileDialog::getSaveFileName(".", fileFilters, this);
  if (fileName.isEmpty())
    return false;

  if (QFile::exists(fileName)) {
    int ret = QMessageBox::warning(this, tr("Spreadsheet"),
                tr("File %1 already exists.\n"
                   "Do you want to overwrite it?")
                .arg(QDir::convertSeparators(fileName)),
                QMessageBox::Yes | QMessageBox::Default,
                QMessageBox::No | QMessageBox::Escape);
    if (ret == QMessageBox::No)
      return true;
  }
  if (!fileName.isEmpty())
    saveFile(fileName);
  return true;
}
```

Слот **saveAs()** соответствует пункту меню "**File|Save As**". Он запрашивает у пользователя имя сохраняемого файла, вызовом **QFileDialog::getSaveFileName()**. Если пользователь нажмет кнопку "**Cancel**", то возвращается значение **false**, которое затем передается выше, функцией **maybeSave()**. Иначе возвращается имя файла, которое может быть как новым именем, так и именем существующего файла. В последнем случае перед пользователем демонстрируется предупреждение:

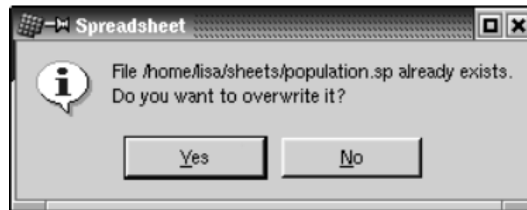


Рисунок 3.10. Запрос: "Do you want to overwrite it?"

Диалогу передается текст:

```
tr("File %1 already exists\n"
   "Do you want to override it?")
.arg(QDir::convertSeparators(fileName))
```

где функция **QString::arg()** выполняет подстановку спецификатора "**%1**" своим аргументом. Например, если предположить, что имя файла **A:\tab04.sp**, то вышеприведенный код будет полностью эквивалентен следующему:

```
"File A:\\tab04.sp already exists.\n"
"Do you want to override it?"
```

есстественно, если исходить из предположения, что приложение не было переведено на какой либо другой язык. Функция **QDir::convertSeparators()** выполняет преобразование платформо-зависимых разделителей элементов пути в файловой системе ("**/**" -- для Unix и Mac OS X, "****" -- для Windows) в символ прямого слэша.

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

Когда пользователь выбирает пункт меню "**File|Exit**" или закрывает приложение нажатием на кнопку "**X**" в заголовке окна, то вызывается слот **QWidget::close()**. Он передает приложению событие "**close**".

Перекрыв функцию **QWidget::closeEvent()**, мы можем предотвратить закрытие окна и решить -- что делать дальше.

Если в приложении имеются несохраненные данные и пользователь отменил операцию сохранения файла, то мы просто "игнорируем" событие и продолжаем работу. В противном случае -- мы подтверждаем закрытие приложения и приложение завершает свою работу.

```
void MainWindow::setCurrentFile(const QString &fileName)
{
  curFile = fileName;
  modLabel->clear();
  modified = false;
  if (curFile.isEmpty()) {
    setCaption(tr("Spreadsheet"));
  } else {
    setCaption(tr("%1 - %2").arg(strippedName(curFile))
                            .arg(tr("Spreadsheet")));
    recentFiles.remove(curFile);
    recentFiles.push_front(curFile);
    updateRecentFileItems();
  }
}

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

В функции **setCurrentFile()** мы записываем имя файла в приватную переменную-член **curFile**, сбрасываем признак "изменен" и обновляем заголовок окна. Обратите внимание: теперь мы использовали два спецификатора, вида "**%n**". Подстановкой первого ("**%1**") занимается первый вызов **arg()**, второго ("**%2**") -- второй вызов. Такую форму записи можно несколько упростить:

```
setCaption(strippedName(curFile) + tr(" - Spreadsheet"));
```

но использование **arg()** дает большую гибкость переводчикам. Чтобы не загромождать заголовок окна длинной строкой, мы удалили из нее путь к файлу с помощью функции **strippedName()**.
Затем обновляется список файлов **recentFiles**, использовавшихся недавно. Для начала вызывается **remove()**, которая удаляет имя файла из списка, а затем **push_front()** добавляет имя файла в начало. Вызов **remove()** необходим для предотвращения появления дублирующихся записей. После обновления списка вызывается **updateRecentFileItems()**, которая выполняет обновление меню "**File**". Переменная **recentFiles** имеет тип **QStringList** (список строк **QString**). В Главе 11 мы подробнее остановимся на классах-контейнерах, таких как **QStringList**.

На этом реализация меню "**File**" практически завершена. Но остается еще один момент. Необходимо выполнить реализацию слота открывающего файлы из списка недавно использовавшихся файлов.

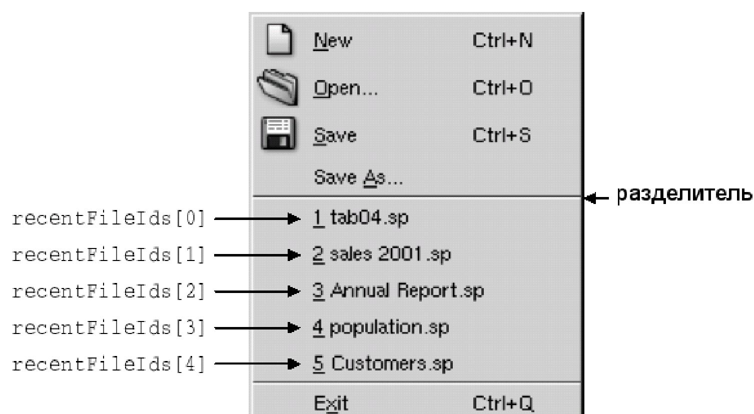


Рисунок 3.11. Меню "File" со списком недавно использовавшихся файлов.

Функция **updateRecentFileItems()** вызывается для обновления элементов меню, соответствующих недавно открывавшимся файлам. Для начала удаляются все "лишние" элементы, начиная с конца списка (длина списка не может превышать числа **MaxRecentFiles**, которое определено в **mainwindow.h** и равно числу 5)
Затем в меню добавляется новый элемент или используется существующий. В самый первый раз в меню добавляется разделитель, отделяющий список файлов от остальных пунктов. Чуть ниже мы объясним назначение функции **setItemParameter()**.

На первый взгляд кажется странным, что в функции **updateRecentFileItems()** мы создаем элементы меню, но никогда не удаляем их! Однако тут нет ничего странного. Мы можем смело утверждать, что в течение сессии список файлов уменьшаться не будет.

Функция **QPopupMenu::insertItem()** имеет следующий синтаксис:

```
fileMenu->insertItem(text, receiver, slot, accelerator, id, index);
```

где **text** -- это текст, который будет отображаться, в данном случае мы используем имя файла без пути к нему. Можно было бы использовать полное имя файла, но это сделает панель меню слишком широкой. Если у вас возникнет необходимость сохранять в меню полный путь к файлу вместе с его именем, то рекомендуем оформлять список файлов в виде подменю.
Аргументы **receiver** и **slot** определяют функцию-обработчик, которая будет вызываться при выборе этого пункта меню. В нашем примере мы указали слот **openRecentFile(int)** главного окна.
В аргументах **accelerator** и **id** мы передаем значения по-умолчанию. Это означает, что данный пункт меню не имеет комбинации "горячих" клавиш, а идентификатор (**id**) генерируется автоматически. Мы сохраняем полученный **id** в массиве **recentFileIds**, что позднее позволит нам обращаться к пункту меню по его идентификатору.

Аргумент **index** -- это порядковый номер записи в меню. Значение **fileMenu->count()-2**, означает, что пункт меню вставляется выше разделителя, отделяющего пункт "**Exit**".

```
void MainWindow::openRecentFile(int param)
{
  if (maybeSave())
    loadFile(recentFiles[param]);
}
```

Слот **openRecentFile()** открывает файл, соответствующий выбранному пункту меню. В качестве аргумента **param** передается число, записанное нами вызовом **setItemParameter()**. Мы выбрали числа такими, что теперь можем использовать их как индексы в списке **recentFiles**.

ID	text	param
-32	1 tab04.sp	0
-33	2 sales 2001.sp	1
-34	3 Annual Report.sp	2
-35	4 population.sp	3
-36	5 Customers.sp	4

index	value
0	A:\tab04.sp
1	C:\sales 2001.sp
2	D:\Annual Report.sp
3	C:\population.sp
4	C:\Customers.sp

Рисунок 3.12. Соответствие между пунктами меню и полными именами файлов.

Таким образом, мы решаем проблему сопоставления пунктов меню полным именам файлов. Менее элегантный способ заключается в создании пяти "действий" (action) и соединении их с пятью различными слотами.

3.4 Настройка строки состояния

На этом мы заканчиваем описание настройки меню и панели инструментов, и переходим к строке состояния. В обычном режиме строка состояния содержит три индикатора: номер текущей ячейки,

формула в текущей ячейке и MOD -- признак того, что файл был изменен. Но помимо этого строка состояния может использоваться для отображения подсказок и других сообщений.

Обычный режим

Подсказка

Сообщение

Рисунок 3.13. Строка состояния

Создается и настраивается строка состояния в функции **createStatusBar()**, которая вызывается из конструктора **MainWindow**:

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

  formulaLabel = new QLabel(this);

  modLabel = new QLabel(tr(" MOD "), this);
  modLabel->setAlignment(AlignHCenter);
  modLabel->setMinimumSize(modLabel->sizeHint());
  modLabel->clear();

  statusBar()->addWidget(locationLabel);
  statusBar()->addWidget(formulaLabel, 1);
  statusBar()->addWidget(modLabel);

  connect(spreadsheet, SIGNAL(currentChanged(int, int)),
          this, SLOT(updateCellIndicators()));
  connect(spreadsheet, SIGNAL(modified()),
          this, SLOT(spreadsheetModified()));
  updateCellIndicators();
}
```

Она возвращает указатель на созданный ею компонент - строку состояния. (Компонент строки состояния создается автоматически, при первом вызове функции **statusBar()**.) Индикаторы -- это простые метки (**QLabel**), текст в которых изменяется по мере необходимости. Во время создания, меткам передается указатель на "владельца" (**this**), хотя в этом нет особой необходимости, поскольку **QStatusBar::addWidget()** "переподчиняет" их автоматически, назначая в качестве "владельца" сам компонент строки состояния.

Из рисунка 3.13 видно, что метки имеют различные размеры. Адрес ячейки и индикатор MOD -- самые короткие, а метка, отображающая действующую в ячейке формулу, самая длинная. Кроме того, при изменении размеров окна, все дополнительное пространство должно выделяться именно ей. Такое поведение достигается за счет указания фактора "**stretch**" (значение 1 в вызове **QStatusBar::addWidget()**). Для других двух меток этот фактор по-умолчанию принимается равным нулю, что означает фиксированный размер меток.

Когда **QStatusBar** размещает виджеты, он сначала выделяет место для "нерастягиваемых" компонентов (выделяя им "идеальный" размер, определяемый вызовом **QWidget::sizeHint()**), а затем все остальное пространство отдается "растягиваемым" виджетам. "Идеальный" размер, в свою очередь, зависит от содержимого виджетов и меняется при изменении содержимого. Задавая начальные значения меток ("**W999**" и "**MOD**"), мы тем самым определяем их минимально возможный размер.

В конце функции мы выполняем соединения между сигналами **Spreadsheet** к двум слотам **MainWindow**: **updateCellIndicators()** и **spreadsheetModified()**.

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

Слот **updateCellIndicator()** обновляет метки, которые отображают адрес текущей ячейки и действующую в ней формулу. Он вызывается всякий раз, когда пользователь переходит из одной ячейки в другую. Кроме того, он вызывается как обычная функция, в конце **createStatusBar()**, для инициализации меток. Это совершенно необходимо, так как **Spreadsheet** не выдает сигнал **currentChanged()** во время инициализации.

```
void MainWindow::spreadsheetModified()
{
  modLabel->setText(tr("MOD"));
  modified = true;
  updateCellIndicators();
}
```

Слот **spreadsheetModified()** обновляет все три индикатора и устанавливает признак **modified** в **true**. (Эта переменная используется для определения несохраненных изменений).

3.5 Использование диалогов

Здесь мы расскажем о принципах работы с диалогами в Qt -- о том как они создаются, инициализируются, запускаются и как от них получить выбор, сделанный пользователем. Здесь мы будем использовать диалоги "**Find**", "**Go-to-Cell**" и "**Sort**", созданные нами в Главе 2. Кроме того, мы создадим простенький диалог "**About**" ("**О программе**").

Начнем с диалога "**Find**". Так как мы хотим, чтобы пользователь имел возможность переключаться между главным окном приложения и окном диалога, необходимо, чтобы диалоговое окно было НЕМОДАЛЬНЫМ. Немодальным называется такое окно, которое работает независимо от остальных окон приложения.
Как правило, при создании немодальных диалогов, их сигналы подключаются к слотам, которые обеспечивают реакцию на действия пользователя.

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

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

Диалог "**Find**" предназначен для выполнения поиска некоторого значения в таблице. Слот **find()** вызывается, когда пользователь выбирает пункт меню "**Edit|Find**", и предназначен для вывода диалогового окна на экран. С этого момента возможны три сценария дальнейшего развития событий:
- Пользователь вызвал диалог впервые.
- Диалог вызывался ранее, но после этого пользователь закрыл его.
- Диалог вызывался ранее и не закрывался (окно диалога видно на экране).

Если диалог не был создан ранее, то он создается и устанавливаются соединения между сигналами **findNext()** и **findPrev()** диалога, и соответствующим слотами **Spreadsheet**. Мы могли бы создать диалог и в конструкторе **MainWindow**, но не делаем этого по соображениям уменьшения времени, необходимого на запуск приложения.

Далее вызываются **show()**, **raise()** и **setActiveWindow()**, которые выводят окно диалога на экран, поверх других окон приложения, и активизируют его. Метод **show()** делает окно диалога видимым, но оно может уже присутствовать на экране -- в этом случае функция **show()** ничего не делает. Так как нам необходимо вывести диалог поверх других окон и активизировать его, мы должны вызвать **raise()** и **setActiveWindow()**. В качестве альтернативы можно предложить следующий код:

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

но он более медлительный.

Перейдем к диалогу "**Go-to-Cell**". В этом случае нет необходимости переключаться между окном приложения и окном диалога. Отсюда следует, что окно диалога "**Go-to-Cell**" должно быть МОДАЛЬНЫМ. Модальным называется такое окно, которое блокирует возможность взаимодействия пользователя с другими окнами приложения до тех пор, пока не будет закрыто модальное окно. Все диалоги нашего приложения, за исключением "**Find**", будут модальными.
Немодальные диалоги вызываются при помощи функции **show()** (если перед этим не вызывалась функция **setModal()**, которая делает окно модальным). Модальные диалоги вызываются функцией **exec()**. Как правило, для модальных диалогов не требуется устанавливать соединения между сигналами и слотами.

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

Функция **QDialog::exec()** возвращает **true**, если результат диалога принимается пользователем, и **false** -- в противном случае. (Помните? В главе 2 мы соединяли сигнал кнопки **OK** со слотом **accept()**, а сигнал от кнопки **Cancel** со слотом **reject()**.) Если пользователь нажмет кнопку **OK**, то мы выполним переход к заданной ячейке, если **Cancel** -- **exec()** вернет **false** и мы не будем ничего предпринимать.

Функция **QTable::setCurrentCell()** принимает два аргумента: номер строки и номер колонки. В нашем приложении, адрес A1, например, соответствует ячейке (0, 0) в таблице, а адрес B27 -- ячейке (26, 1). Чтобы получить номер строки, из **QString**, возвращаемой **QLabel::text()**, извлекается ее часть, с помощью **QString::mid()** и затем преобразуется в целое число с помощью **QString::toInt()**. После этого, из полученного числа вычитается единица (поскольку нумерация строк в **QTable** начинается с 0). Чтобы получить номер колонки, из кода символа колонки мы просто вычитаем код символа "**A**".
В отличие от диалога "**Find**", экземпляр диалога "**Go-to-Cell**" создается на стеке. Это общепринятая практика для модальных диалогов, вызываемых из разного рода меню, поскольку они становятся не нужны после их использования.

А теперь перейдем к диалогу сортировки. Этот диалог так же является модальным и позволяет отсортировать выделенный дипазон ячеек по заданным колонкам. На рисунке 3.14 показан пример сортировки по колонкам B (первичный ключ) и A (вторичный ключ) в порядке возрастания.

	A	B	C	D
1	George	Washington	1789-1797	
2	John	Adams	1797-1801	
3	Thomas	Jefferson	1801-1809	
4	James	Madison	1809-1817	
5	James	Monroe	1817-1825	
6	John Quincy	Adams	1825-1829	
7	Andrew	Jackson	1829-1837	
8				

(а) До сортировки

	A	B	C	D
1	John	Adams	1797-1801	
2	John Quincy	Adams	1825-1829	
3	Andrew	Jackson	1829-1837	
4	Thomas	Jefferson	1801-1809	
5	James	Madison	1809-1817	
6	James	Monroe	1817-1825	
7	George	Washington	1789-1797	
8				

(б) После сортировки

Рисунок 3.14. Сортировка выбранного диапазона ячеек

```
void MainWindow::sort()
{
  SortDialog dialog(this);
  QTableSelection sel = spreadsheet->selection();
  dialog.setColumnRange( A  + sel.leftCol(),  A  + sel.rightCol());

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

Алгоритм функции **sort()**:

- Диалог создается на стеке и инициализируется.
- Диалог запускается вызовом **exec()**
- Если пользователь нажал кнопку **OK**, то из виджетов диалога извлекается необходимая информация и выполняется сортировка.

Объект **compare** хранит первичный, вторичный и третичный ключи сортировки, а так же порядок сортировки по каждому из ключей. (Определение класса **SpreadsheetCompare** мы опишем в следующей главе.) Этот объект используется функцией **Spreadsheet::sort()** для сравнения двух строк. Массив **keys** хранит номера колонок-ключей. Например, если выбран диапазон ячеек с C2 по E5, то колонка C имеет номер 0. Массив **ascending** хранит порядок сортировки для каждого из ключей. Функция **QComboBox::currentItem()** возвращает индекс текущего выбранного элемента списка, начиная с 0. Для вторичного и третичного ключей, из индекса вычитается 1, чтобы учесть элемент "**None**". Реализация **sort()** очень чувствительна к дизайну диалога "**Sort**", точнее -- эта "чувствительность" связана с выпадающими списками и элементами списков "**None**". Если вы измените диалог, то скорее всего вам придется изменить и код функции. Пока этот диалог вызывается из одного места в программе -- обслуживание его не так трудоемко. Но как только вы попытаетесь вызывать диалог из разных точек в программе, то обслуживание всех изменений, вносимых в него, может превратиться в кошмарный сон.

Чтобы избежать подобных трудностей, можно порекомендовать сделать диалог более "интеллектуальным", который сам будет создавать экземпляр класса **SpreadsheetCompare** и передавать его в вызывающую функцию. В этом случае функция **sort()** могла бы выглядеть так:

```
void MainWindow::sort()
{
  SortDialog dialog(this);
```

```
  QTableSelection sel = spreadsheet->selection();
  dialog.setColumnRange( 'A' + sel.leftCol(), 'A' + sel.rightCol());
  if (dialog.exec())
    spreadsheet->performSort(dialog.comparisonObject());
}
```

Такой подход применяется к слабосвязанным компонентам и практически всегда оправдан в тех случаях, когда один и тот же диалог вызывается более чем из одного места в программе.
Более радикальный подход -- передать диалогу указатель на **Spreadsheet** и позволить ему напрямую работать с таблицей. Это несколько снижает универсальность диалога, так как он теперь будет "привязан" к определенному типу виджета, но значительно упрощает код за счет отказа от функции **SortDialog::setColumnRange()**. В этом случае, код функции **MainWindow::sort()** приобретает такой вид:

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

  dialog.exec();
}
```

Этот подход является полной противоположностью. Теперь уже не программа должна "знать" архитектуру и алгоритм работы диалога, а диалог должен "знать" об архитектуре вызывающей программы. Такой подход может оказаться оправданным, когда диалогу необходимо предоставить возможность оперативного изменения данных. Но и в этом случае код программы крайне чувствителен к реализации диалога.
Некоторые разработчики выбирают один из описанных вариантов и применяют только его. В этом есть свои преимущества, поскольку все диалоги имеют одинаковое строение, хотя при этом могут оказаться не использованными преимущества других подходов. В любом случае, решение о том, какой из подходов должен использоваться, необходимо принимать в каждом конкретном приложении, на основе требований, предъявляемых к диалогам.

В завершение этого раздела мы создадим простенький диалог -- окно, содержащее сведения о программе и разработчике. Такой диалог можно создать самому, аналогично рассмотренным ранее диалогам "**Find**" или "**Sort**", но Qt предоставляет более простое решение.

```
void MainWindow::about()
{
  QMessageBox::about(this, tr("About Spreadsheet"),
          tr("<h2>Spreadsheet 1.0</h2>"
          "<p>Copyright &copy; 2003 Software Inc."
          "<p>Spreadsheet is a small application that "
          "demonstrates <b>QAction</b>, <b>QMainWindow</b>, "
          "<b>QMenuBar</b>, <b>QStatusBar</b>, "
          "<b>QToolBar</b>, and many other Qt classes."));
}
```

Вызывается диалог функцией **QMessageBox::about()**. Очень похоже на функцию **QMessageBox::warning()**, за одним маленьким исключением: вместо стандартной иконки "**warning**", используется иконка приложения.

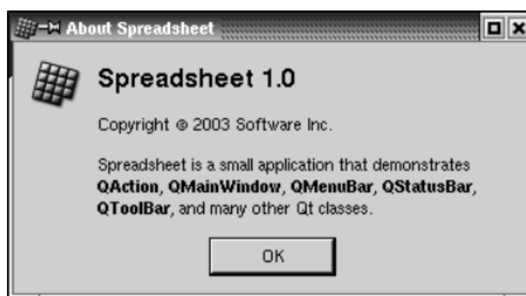


Рисунок 3.15. Диалог с информацией о программе.

До сих пор мы использовали ряд, очень удобных в обращении, статических функций-членов из классов **QMessageBox** и **QFileDialog**. Эти функции "на лету" создают диалоги, инициализируют их и вызывают функцию **exec()**. Но можно, хотя это и менее удобно, самому создать **QMessageBox** или **QFileDialog**, подобно любому другому виджету, и явно вызвать **exec()** или даже **show()**.

3.6 Сохранение пользовательских настроек приложения

В конструкторе **MainWindow**, для загрузки пользовательских настроек, мы вызывали функцию **readSettings()**. Аналогично, для их сохранения, в обработчике **closeEvent()**, вызывалась функция **writeSettings()**. Пришло время рассмотреть реализацию обоих функций, которые являются методами класса **MainWindow**.
Для своего приложения, в качестве хранилища настроек, мы выбрали класс **QSettings**. Экземпляр этого класса может быть создан и использован в любой момент, по мере необходимости.

```
void MainWindow::writeSettings()
{
  QSettings settings;
  settings.setPath("software-inc.com", "Spreadsheet");
  settings.beginGroup("/Spreadsheet");
  settings.writeEntry("/geometry/x", x());
  settings.writeEntry("/geometry/y", y());
  settings.writeEntry("/geometry/width", width());
  settings.writeEntry("/geometry/height", height());
  settings.writeEntry("/recentFiles", recentFiles);
  settings.writeEntry("/showGrid", showGridAct->isOn());
  settings.writeEntry("/autoRecalc", showGridAct->isOn());
  settings.endGroup();
}
```

Функция **writeSettings()** сохраняет геометрию главного окна (положение на экране и размеры), список недавно использовавшихся файлов и состояние флагов **Show Grid** и **Auto-recalculate**.

Место, куда **QSettings** сохраняет настройки, зависит от используемой платформы. В Windows сохранение производится в системный реестр, в Unix -- в текстовый файл, в Mac OS X используется Carbon API. Методу **setPath()** передаются названия организации и программного продукта. Эти сведения используются для преобразование в платформо-зависимое представление места сохранения настроек.

Настройки хранятся в виде пары: ключ-значение. Ключ очень похож на строку пути в файловой системе и всегда должен начинаться с названия приложения. Например, **/Spreadsheet/geometry/x** или **/Spreadsheet/showGrid**. (Функция **beginGroup()** "запоминает" префикс ключа - название приложения, которое будет автоматически подставляться в начало ключа, что позволяет нам сэкономить на своих усилиях.) Значение, той или иной настройки, может быть одного из пяти типов: **int**, **bool**, **double**, **QString** или **QStringList**.

```
void MainWindow::readSettings()
{
  QSettings settings;
  settings.setPath("software-inc.com", "Spreadsheet");
  settings.beginGroup("/Spreadsheet");
  int x = settings.readNumEntry("/geometry/x", 200);
  int y = settings.readNumEntry("/geometry/y", 200);
  int w = settings.readNumEntry("/geometry/width", 400);
  int h = settings.readNumEntry("/geometry/height", 400);
  move(x, y);
  resize(w, h);

  recentFiles = settings.readListEntry("/recentFiles");
  updateRecentFileItems();

  showGridAct->setOn(
          settings.readBoolEntry("/showGrid", true));
  autoRecalcAct->setOn(
          settings.readBoolEntry("/autoRecalc", true));
```

```
  settings.endGroup();
}
```

Функция **readSettings()** загружает настройки, предварительно сохраненные вызовом **writeSettings()**. Второй аргумент в "**read**"-функциях -- это значение по-умолчанию, возвращаемое в том случае, если запрошенный ключ отсутствует. Значения по-умолчанию используются на самом первом запуске приложения, когда настройки еще не были сохранены.
На этом мы заканчиваем рассмотрение реализации главного окна приложения. В следующих разделах мы покажем, как изменить наше приложение, чтобы оно могло работать одновременно с несколькими документами, и как добавить к приложению показ заставки во время загрузки.

3.7 Работа с несколькими документами одновременно

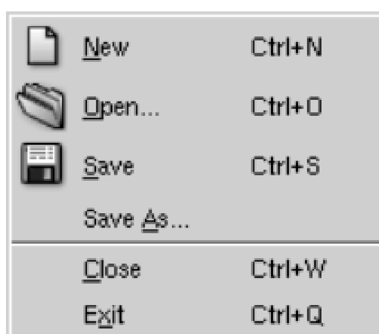
Мы готовы приступить к созданию функции **main()**:

```
#include <qapplication.h>
#include "mainwindow.h"

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

Эта функция немного отличается от того, что мы видели до сих пор: экземпляр **MainWindow** был создан на стеке, без использования оператора new. Благодаря этому, объект класса **MainWindow** будет уничтожен автоматически, по завершении работы функции.

В данном случае, программа **Spreadsheet** представляет из себя единственное главное окно, и может работать только с одним документом в каждый конкретный момент времени. Если нам потребуется работать с несколькими документами одновременно, мы должны будем запустить несколько экземпляров программы.

Но для пользователя было бы гораздо удобнее, если бы программа могла запускать несколько главных окон с различными документами, как, например, это делают некоторые web-браузеры. Попробуем внести дополнительные изменения в нашу программу, чтобы она могла одновременно работать с несколькими документами. Для этого, прежде всего, необходимо немного изменить меню **File**:



Пункт **File|New** создает новое главное окно с пустым документом, вместо того, чтобы создавать новый документ в этом же окне.·

Пункт **File|Close** закрывает текущее главное окно.

Пункт **File|Exit** закрывает все окна приложения.

Рисунок 3.16. Новое меню File

В своем первоначальном варианте, меню **File** не имело пункта **Close**, поскольку смысл операции закрытия окна был равносилен завершению приложения (пункт **Exit**).
Так выглядит новый вариант функции **main()**:

```
#include <qapplication.h>
#include "mainwindow.h"
```

```
int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  MainWindow *mainWin = new MainWindow;
  mainWin->show();
  QObject::connect(&app, SIGNAL(lastWindowClosed()),
                   &app, SLOT(quit()));
  return app.exec();
}
```

Здесь мы связали сигнал **lastWindowClosed()** со слотом **quit()**, который завершает приложение.
В данном варианте, теперь имеет смысл создавать экземпляр **MainWindow** оператором new, поскольку затем, при закрытии окна, он будет удаляться оператором **delete**. Эта необходимость не возникает в случае приложения, которое работает с единственным документом.
Ниже приводится измененный вариант слота **MainWindow::newFile()**:

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

Здесь просто создается новый экземпляр **MainWindow**. Может показаться странным, что мы нигде не запоминаем указатель на вновь созданный объект, но здесь нет никакой ошибки -- Qt хранит указатели на все окна сама, без нашего участия.
Ниже приводится код, создающий "действия" (actions) **Close** и **Exit**:

```
  closeAct = new QAction(tr("&Close"), tr("Ctrl+W"), this);
  connect(closeAct, SIGNAL(activated()), this, SLOT(close()));

  exitAct = new QAction(tr("E&xit"), tr("Ctrl+Q"), this);
  connect(exitAct, SIGNAL(activated()), qApp, SLOT(closeAllWindows()));
```

Слот c**loseAllWindows()** закрывает все окна приложения, кроме тех, которые отвергнут событие close. Это в точности соответствует нашим требованиям. Нам нет нужды беспокоиться о несохраненных изменениях, поскольку сохранение выполняется в обработчике **MainWindow::closeEvent()**, при закрытии окна.
Теперь наше приложение в состоянии работать с несколькими окнами. К сожалению, на данный момент у нас в программе кроется трудноуловимая ошибка. Если пользователь будет создавать и закрывать окна приложения, то может наступить момент, когда вся доступная память в машине будет исчерпана! Это происходит потому, что мы создаем новые окна, выбирая пункт меню **File|New**, но нигде не удаляем их из памяти. Когда пользователь закрывает очередное окно, то объект класса **MainWindow** не удаляется из памяти, а просто делается невидимым.
Решение этой проблемы заключается в добавлении флага **WDestructiveClose** в конструктор:

```
MainWindow::MainWindow(QWidget *parent, const char *name)
    : QMainWindow(parent, name, WDestructiveClose)
{
  ...
}
```

Он вынуждает Qt удалять объект окна при его закрытии. Этот флаг один из множества, которые могут быть переданы в конструктор наследника от **QWidget**, но другие флаги используются довольно редко.
Однако, утечка памяти -- не единственная проблема, с которой мы можем столкнуться. Весь наш первоначальный дизайн предполагал работу с единственным главным окном. Теперь, каждое из окон приложения может иметь свой список недавно использовавшихся файлов и свои дополнительные настройки. Совершенно очевидно, что список недавно использовавшихся файлов должен быть глобальным для всего приложения. Сделать это можно довольно легко, достаточно просто объявить переменную **recentFiles** статической. Но, теперь везде, где необходимо вызвать **updateRecentFileItems()** для обновления меню **File**, мы должны вызвать эту функцию для всех главных окон. Ниже приводится код, который делает это:

```
  QWidgetList *list = QApplication::topLevelWidgets();
  QWidgetListIt it(*list);
  QWidget *widget;
  while ((widget = it.current())) {
    if (widget->inherits("MainWindow"))
      ((MainWindow *)widget)->updateRecentFileItems();
      ++it;
  }
  delete list;
```

Здесь выполняется перебор всех виджетов верхнего уровня и вызывается функция **updateRecentFileItems()** во всех экземплярах **MainWindow**. Аналогичный подход может быть использован для синхронизации флагов **Show Grid** и **Auto-recalculate**, а так же для предотвращения загрузки одного и того же документа дважды. Тип **QWidgetList** определен как **QPtrList<QWidget>**, который будет обсуждаться в Главе 11 (Классы-контейнеры).

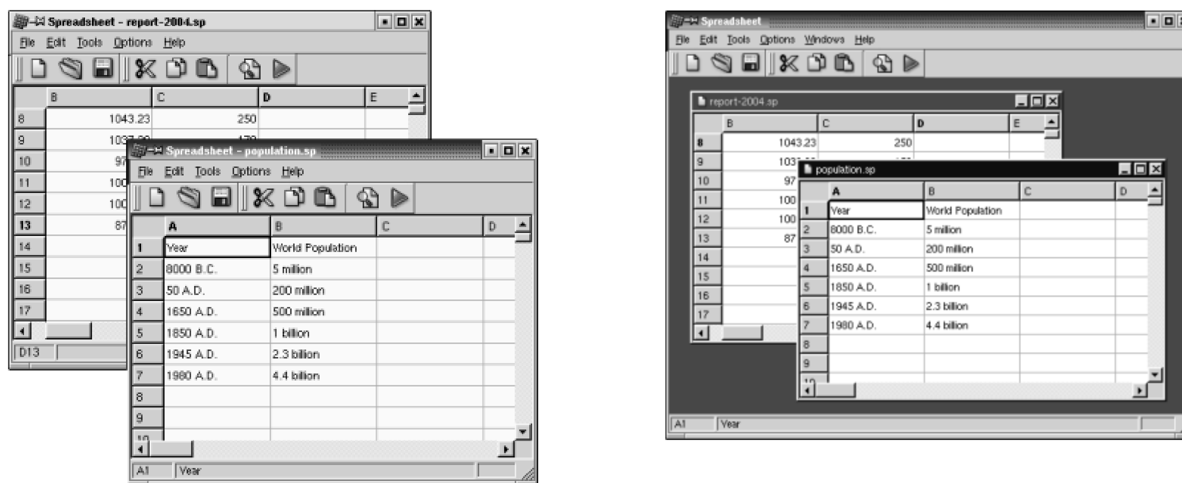


Рисунок 3.17. SDI и MDI

Когда приложение открывает каждый следующий документ в новом окне, то говорят, что приложение относится к классу SDI-приложений (от англ. single document interface -- однодокументный интерфейс). Популярная альтернатива SDI -- MDI (от англ. multiple document interface -- многодокументный интерфейс), в этом случае приложение имеет одно главное окно, которое может включать в себя несколько дочерних окон с открытыми документами и разделяющими между собой пространство главного окна. С помощью Qt можно создавать как SDI, так и MDI приложения. На рисунке 3.17 показаны оба варианта оформления приложения Spreadsheet. Более подробно MDI будет описан в Главе 6.

3.8 Экран-заставка

Многие приложения показывают заставку во время запуска. Одни разработчики "показывают" заставку, чтобы снизить психологическое давление на пользователя, ожидающего пока приложение загрузится, другие -- в рекламных целях. Добавление экрана-заставки к приложению -- довольно простая задача.

В Qt реализация экрана-заставки выполнена в виде класса **QSplashScreen**, который выводит на экран окно с изображением до того, как приложение будет полностью загружено. Имеется возможность отображать ход загрузки на заставке. Код, который выводит заставку на экран, как правило, размещается в функции **main()**, перед вызовом **QApplication::exec()**.

Ниже показан пример функции **main()**, использующей **QSplashScreen** для вывода заставки на время загрузки дополнительных модулей и установки соединения по сети.

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

```
  splash->message(QObject::tr("Setting up the main window..."),
                  Qt::AlignRight | Qt::AlignTop, Qt::white);
  MainWindow mainWin;
  app.setMainWidget(&mainWin);

  splash->message(QObject::tr("Loading modules..."),
                  Qt::AlignRight | Qt::AlignTop, Qt::white);
  loadModules();

  splash->message(QObject::tr("Establishing connections..."),
                  Qt::AlignRight | Qt::AlignTop, Qt::white);
  establishConnections();

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

  return app.exec();
}
```

Рисунок 3.18. Виджет QSplashScreen.

На этом мы завершаем разработку пользовательского интерфейса приложения **Spreadsheet** и в следующей главе перейдем к реализации основной его функциональности.

4 Реализация функциональности приложения

В предыдущих двух главах мы описывали процесс создания пользовательского интерфейса приложения **Spreadsheet**. В этой главе мы наполним свое приложение необходимой функциональностью. Кроме всего прочего, мы рассмотрим -- как загружать и сохранять документы, как хранить данные в памяти, как реализовать операции с буфером обмена и как добавить поддержку формул электронной таблицы в компонент **QTable**.

4.1 Центральный виджет

Центральная область окна приложения может быть занята виджетом любого типа. Рассмотрим возможные варианты:

1 Стандартный виджет Qt.
В качестве центрального виджета может быть использован один из стандартного набора, предлагаемого библиотекой Qt, например, **QTable** или **QTextEdit**. В этом случае функциональность приложения должна быть реализована отдельно (например в потомке класса **QMainWindow**).

2 Виджет, созданный разработчиком
Для специализированных применений, разработчики достаточно часто создают свои собственные виджеты. Например, программа редактирования файлов с иконками может иметь, в качестве центрального, виджет **IconEditor**. В Главе 5 мы рассмотрим проблему создания своих виджетов.

3 Обычный QWidget с менеджером размещения
Иногда область окна приложения заполняют несколько виджетов. В этом случае, в качестве центрального, создается виджет **QWidget**, который выступает в роли владельца для всех других виджетов и использует менеджер размещения для их компоновки.

4 Splitter (разделитель)
Другой способ размещения нескольких виджетов в окне -- использовать разделитель (**QSplitter**). **QSplitter** размещает подчиненные виджеты по горизонтали, подобно **QHBox**, или по вертикали, подобно **QVBox**, с возможностью управления размерами виджетов по одной из осей. Разделители могут содержать в себе виджеты любого другого типа, в том числе и другие разделители.

5 Рабочее пространство MDI
Если приложение реализует многодокументный интерфейс (MDI), то в этом случае всю область окна занимает виджет **QWorkspace**, а каждое дочернее окно является подчиненным, по отношению к нему.

Области размещения (**layouts**), разделители (**splitters**) и рабочие пространства MDI могут комбинироваться как со стандартными виджетами Qt, так и с виджетами, разработанными вами. В Главе 6 мы рассмотрим эти классы поближе.

В нашем примере, в качестве центрального, используется виджет, порожденный от **QTable**. Класс **QTable** уже имеет многое из того, что нам необходимо, но он не поддерживает формулы, например: "=A1+A2+A3", а так же не поддерживает операции с буфером обмена. Поэтому мы займемся добавлением необходимой функциональности в виде класса **Spreadsheet**, потомка класса **QTable**.

4.2 Создание класса-потомка от QTable

Начнем создание нашего класса **Spreadsheet** с оформления файла заголовка:

```
#ifndef SPREADSHEET_H
#define SPREADSHEET_H

#include <qstringlist.h>
#include <qtable.h>

class Cell;
class SpreadsheetCompare;
```

Заголовочный файл начинается с опережающего описания классов **Cell** и **SpreadsheetCompare**.

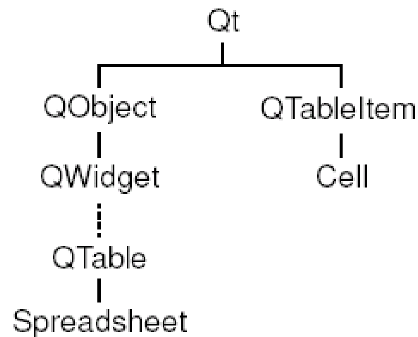


Рисунок 4.1. Дерево наследования классов Spreadsheet и Cell.

Атрибуты ячейки в **QTable**, такие как текст и выравнивание, хранятся в элементе **QTableItem**. В отличие от **QTable**, класс **QTableItem** не является виджетом и предназначен исключительно для хранения данных. Класс **Cell** порожден от **QTableItem**. В дополнение к атрибутам родительского класса, он имеет возможность хранить формулу вычисления содержимого ячейки.
Мы подробно обсудим реализацию класса **Cell** в последнем разделе этой главы.

```
class Spreadsheet : public QTable
{
  Q_OBJECT
public:
  Spreadsheet(QWidget *parent = 0, const char *name = 0);

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

Класс **Spreadsheet** является потомком класса **QTable**.

В Главе 3, при разработке **MainWindow**, мы уже использовали некоторые из публичных методов **Spreadsheet**. Например, мы вызывали **clear()** из **MainWindow::newFile()**. Кроме того, были использованы некоторые функции, унаследованные от **QTable**, например **setCurrentCell()** и **setShowGrid()**.

```
public slots:
  void cut();
  void copy();
  void paste();
  void del();
  void selectRow();
  void selectColumn();
  void selectAll();
  void recalculate();
  void setAutoRecalculate(bool on);
  void findNext(const QString &str, bool caseSensitive);
  void findPrev(const QString &str, bool caseSensitive);

signals:
  void modified();
```

Spreadsheet предоставляет несколько слотов, которые реализуют функциональность пунктов меню **Edit**, **Tools** и **Options**.

```
protected:
  QWidget *createEditor(int row, int col, bool initFromCell) const;
```

```
  void endEdit(int row, int col, bool accepted, bool wasReplacing);
```

Дополнительно он перекрывает реализацию ряда виртуальных функций **QTable**, которые вызываются, когда пользователь изменяет значение в ячейке. Это необходимо для поддержки формул в ячейках.

```
private:
  enum { MagicNumber = 0x7F51C882, NumRows = 999, NumCols = 26 };

  Cell *cell(int row, int col) const;
  void setFormula(int row, int col, const QString &formula);
  QString formula(int row, int col) const;
  void somethingChanged();

  bool autoRecalc;
};
```

В приватной секции мы определили три константы, четыре функции и одну переменную.

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

  enum { NumKeys = 3 };
  int keys[NumKeys];
  bool ascending[NumKeys];
};
#endif
```

Заголовочный файл завершается определением класса **SpreadsheetCompare**. Мы опишем его, когда коснемся реализации метода **Spreadsheet::sort()**.
Теперь перейдем к рассмотрению реализации каждой из функций:

```
#include <qapplication.h>
#include <qclipboard.h>
#include <qdatastream.h>
#include <qfile.h>
#include <qlineedit.h>
#include <qmessagebox.h>
#include <qregexp.h>
#include <qvariant.h>

#include <algorithm>
#include <vector>
using namespace std;

#include "cell.h"
#include "spreadsheet.h"
```

Мы подключили заголовочные файлы классов, использующихся в приложении, а так же стандартные заголовки C++: <**algorithm**> и <**vector**>. Директива **using namespace** импортирует все имена из пространства **std** в глобальное пространство имен, что позволяет использовать сокращенную форму записи: **stable_sort()** и **vector<?>** вместо полной формы: **std::stable_sort()** и **std::vector<?>**.

```
Spreadsheet::Spreadsheet(QWidget *parent, const char *name)
    : QTable(parent, name)
{
  autoRecalc = true;
  setSelectionMode(Single);
  clear();
}
```

В конструкторе устанавливается режим выборки строк в **QTable** -- **Single**. Это означает, что в таблице может существовать только одна выделенная область ячеек, в каждый конкретный момент времени.

```
void Spreadsheet::clear()
```

```
{
  setNumRows(0);
  setNumCols(0);
  setNumRows(NumRows);
  setNumCols(NumCols);
  for (int i = 0; i < NumCols; i++)
    horizontalHeader()->setLabel(i, QChar('A' + i));
  setCurrentCell(0, 0);
}
```

Функция **clear()** вызывается для инициализации таблицы, в конструкторе и в **MainWindow::newFile()**. Собственно очистка производится за счет изменения размера таблицы до (0 X 0), после чего восстановливается ее первоначальный размер (26 X 999). Затем выполняется заполнение меток столбцов: "A", "B", ..., "Z" (номера столбцов 1, 2, ..., 26, соответственно) и перемещение курсора в ячейку A1.

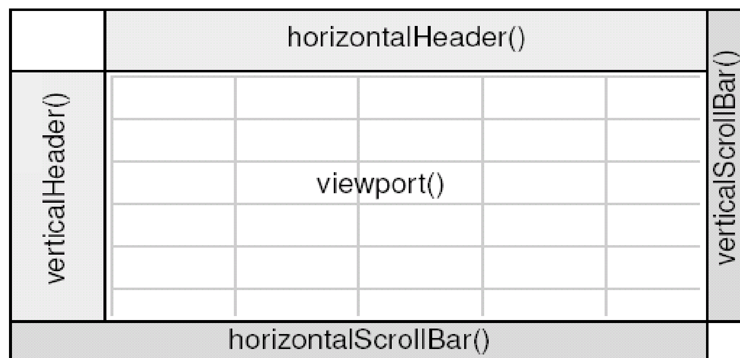


Рисунок 4.2. Виджеты, составляющие QTable.

QTable состоит из нескольких подчиненных виджетов. В их число входят: горизонтальный заголовок -- **QHeader**, находящийся в верхней части; вертикальный заголовок -- **QHeader**, находящийся слева; полосы прокрутки -- **QScrollBar** справа и снизу. Центральную область занимает специальный виджет, который называется viewport, в котором **QTable** рисует сетку с ячейками. Доступ к подчиненным виджетам реализуется через функции **QTable** и его базового класса **QScrollView**. Например, в функции **clear()** мы обращались к горизонтальному заголовку через вызов **QTable::horizontalHeader()**.

4.2.1 Хранение данных в виде отдельных объектов

В приложении **Spreadsheet**, все не пустые ячейки хранятся в памяти, в виде отдельных объектов **QTableItem**. Этот способ присущ не только **QTable**, такие классы, как **QIconView**, **QListBox** и **QListView** тоже хранят свои данные в виде отдельных элементов (**QIconViewItem**, **QListBoxItem** и **QListViewItem**, соответственно). Классы элементов используются в Qt для хранения данных. Например, **QTableItem** может хранить такие данные, как строка, рисунок и указатель на таблицу **QTable**. Создавая класс, наследник от класса элемента, мы можем расширить его возможности и хранить в нем дополнительные данные, а за счет перекрытия виртуальных функций -- управлять этими данными. Некоторые библиотеки, при реализации своих элементов хранения данных, предусматривают в них дополнительный указатель типа void, который может использоваться для присоединения к элементу дополнительных данных. Qt не следует этому примеру и не обременяет свои классы лишними указателями, которые могут не использоваться. Вместо этого, она дает программистам свободу создания дочерних классов и возможность подстраивать их под свои нужды. Если вам так необходим дополнительный указатель на внешнюю структуру данных, то вы с легкостью сможете добавить его в дочернем классе. В **QTable** достаточно просто можно "обойти" механизм работы с элементами, путем повторной реализации низкоуровневых функций, таких как **paintCell()** и **clearCell()**. Если данные, отображаемые в **QTable**, уже находятся в памяти или во внешних структурах, то такой способ поможет избежать ненужного дублирования информации. За подробностями обращайтесь к статье "A Model/View Table for Large Datasets".Ожидается, что Qt 4, в этом отношении, будет проявлять большую гибкость, чем Qt 3. Кроме того, Qt 4 вероятно предложит единственный унифицированный класс элемента данных, используемый всеми классами отображения, допуская показ одного и того же элементами различными способами.

QScrollView -- это единственный базовый класс для виджетов, которые могут отображать значительные объемы данных. Он предоставляет область просмотра и две полосы прокрутки, которые могут быть включены и выключены. Подробнее мы остановимся на этом в Главе 6.

```
Cell *Spreadsheet::cell(int row, int col) const
{
  return (Cell *)item(row, col);
}
```

Приватная функция **cell()** возвращает указатель на объект, находящийся на пересечении заданных строки и столбца. Это практически то же самое, что и **QTable::item()**, за исключением того, что она возвращает указатель на экземпляр класса **Cell**, а не **QTableItem**.

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

Функция **formula()** возвращает формулу для заданной ячейки. Если **cell()** вернет пустой указатель (ячейка отсутствует, т.е. пустая), то в качестве формулы возвращается пустая строка.

```
void Spreadsheet::setFormula(int row, int col,
                             const QString &formula)
{
  Cell *c = cell(row, col);
  if (c) {
    c->setFormula(formula);
    updateCell(row, col);
  } else {
    setItem(row, col, new Cell(this, formula));
  }
}
```

Функция **setFormula()** устанавливает формулу вычисления для заданной ячейки. Если объект хранения данных для ячейки уже существует, то формула записывается в этот объект и затем вызывается **updateCell()**, чтобы сообщить **QTable** о необходимости перерисовать ячейку (если она видна на экране). В противном случае создается новый объект **Cell** и вызывается **QTable::setItem()**, для вставки объекта в таблицу и перерисовки ячейки. Нам нет нужды беспокоиться об уничтожении **Cell**, поскольку **QTable** берет владение объектом на себя и сама удалит его, когда придет время.

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

Функция **currentLocation()** возвращает "адрес" ячейки, в обычном, для электронной таблицы, формате, где за символом, обозначающим столбец, следует номер строки. **MainWindow::updateCellIndicators()** использует эту функцию для отображения адреса текущей ячейки в строке состояния.

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

Функция **currentFormula()** возвращает формулу для текущей ячейки. Она также вызывается из **MainWindow::updateCellIndicators()**.

```
QWidget *Spreadsheet::createEditor(int row, int col,
                                   bool initFromCell) const
{
```

```
  QLineEdit *lineEdit = new QLineEdit(viewport());
  lineEdit->setFrame(false);
  if (initFromCell)
    lineEdit->setText(formula(row, col));
  return lineEdit;
}
```

Функция **createEditor()** перекрывает реализацию в **QTable**. Она вызывается, когда пользователь начинает редактирование содержимого ячейки -- либо после щелчка мышью по ячейке, либо по нажатии на клавишу F2, либо когда пользователь просто начинает набирать текст. Назначение этой функции заключается в создании виджета-редактора, который будет отображаться поверх ячейки. Если функция вызывается по щелчку мыши или по нажатию на клавишу F2, то **initFromCell** получает значение **true**, в результате производится редактирование существующего содержимого ячейки, иначе -- прежние данные игнорируются.

createEditor() создает объект класса **QLineEdit** и записывает в него содержимое ячейки, если **initFromCell** содержит значение **true**. Мы выполнили повторную реализацию этой функции для того, чтобы показывать формулу ячейки вместо ее содержимого.

Объект **QLineEdit** создается как подчиненный области просмотра **QTable**. В свою очередь **QTable** берет на себя обязательство по установке размеров и положения **QLineEdit** на экране такими, чтобы они полностью совпадали с размерами и положением ячейки. Она так же берется автоматически уничтожить объект **QLineEdit**, когда необходимость в нем отпадет.

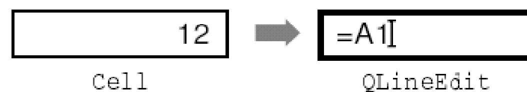


Рисунок 4.3. Редактирование ячейки за счет подстановки QLineEdit.

В большинстве случаев, формула и содержимое ячейки -- суть одно и то же. Например, формула "Hello" превращается в строку "Hello". Таким образом, если пользователь напечатает в ячейке слово "Hello" и нажмет Enter, то ячейка будет отображать слово "Hello". Однако тут есть ряд исключений:

- Если формула -- это число, то она будет интерпретирована как число. Например, формула "1.50" интерпретируется как число 1.5, типа double, и отображается в таблице с выравниванием по правому краю.
- Если формула начинается с одиночной кавычки, то она интерпретируется как текст. Например, формула "'12345" будет интерпретирована как строка символов "12345".
- Если формула начинается с символа "=", то она будет интерпретирована как арифметическое выражение. Например, если ячейка A1 содержит "12", а A2 -- "6", то формула "=A1+A2" вернет сумму "18".

Действия по преобразованию формулы в значение выполняются классом **Cell**. Важное примечание: имейте ввиду, что текст, отображаемый в ячейке, это результат преобразования формулы, а не сама формула.

```
void Spreadsheet::endEdit(int row, int col, bool accepted,
                          bool wasReplacing)
{
  QLineEdit *lineEdit = (QLineEdit *)cellWidget(row, col);
  if (!lineEdit)
    return;
  QString oldFormula = formula(row, col);
  QString newFormula = lineEdit->text();

  QTable::endEdit(row, col, false, wasReplacing);

  if (accepted && newFormula != oldFormula) {
    setFormula(row, col, newFormula);
    somethingChanged();
  }
}
```

Функция **endEdit()** перекрывает аналогичную в **QTable**. Она вызывается, когда пользователь завершает редактирование ячейки -- либо щелчком мыши по любой другой ячейке (что подтверждает внесенные изменения), либо нажатием на клавишу **Enter** (что так же подтверждает внесенные изменения), либо нажатием на клавишу **Esc** (что отвергает внесенные изменения). Основное назначение функции -- переместить содержимое компонента редактора в объект **Cell**, если внесенные изменения подтверждены.

Доступ к редактору выполняется через обращение к **QTable::cellWidget()**. Мы можем без опаски выполнить приведение типа к **QLineEdit**, поскольку создаваемый нами компонент редактора -- всегда **QLineEdit**.

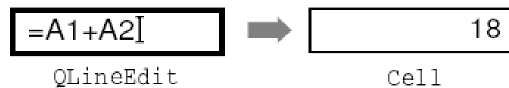


Рисунок 4.4. Передача содержимого QLineEdit обратно в ячейку.

В теле функции вызывается, унаследованная от **QTable**, функция **endEdit()**, поскольку мы должны известить **QTable** об окончании редактирования. В качестве третьего аргумента ей передается **false**, для предотвращения модификации элемента таблицы, поскольку мы сами выполняем все необходимые действия по созданию и модификации элементов. Если "новая" формула отличается от "старой", то вызывается **setFormula()**, для записи формулы в объект класса **Cell**. Вслед за этим вызывается функция **somethingChanged()**.

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

Она выполняет пересчет всего содержимого таблицы, если установлен флаг **Auto-recalculate**, а затем выдает сигнал **modified()**.

4.3 Загрузка и сохранение

Теперь перейдем к реализации загрузки и сохранения файлов (в двоичном формате), создаваемых нашей программой. Делать мы это будем с помощью **QFile** и **QDataStream**, которые предоставляют платформо-независимый интерфейс для операций ввода/вывода двоичных данных.
Начнем с функции записи файла на диск:

```
bool Spreadsheet::writeFile(const QString &fileName)
{
  QFile file(fileName);
  if (!file.open(IO_WriteOnly)) {
    QMessageBox::warning(this, tr("Spreadsheet"),
                         tr("Cannot write file %1:\n%2.")
                         .arg(file.name())
                         .arg(file.errorString()));
    return false;
  }
  QDataStream out(&file);
  out.setVersion(5);

  out << (Q_UINT32)MagicNumber;

  QApplication::setOverrideCursor(waitCursor);
  for (int row = 0; row < NumRows; ++row) {
    for (int col = 0; col < NumCols; ++col) {
      QString str = formula(row, col);
      if (!str.isEmpty())
        out << (Q_UINT16)row << (Q_UINT16)col << str;
    }
  }
  QApplication::restoreOverrideCursor();
```

```
  return true;
}
```

Функция **writeFile()** вызывается из **MainWindow::saveFile()**, для записи файла на диск. В случае успеха возвращает **true**, иначе -- **false**.
Функция начинается с создания экземпляра **QFile**, с заданным именем файла, после чего файл открывается на запись. Затем создается объект **QDataStream**, который, используя **QFile**, записывает данные на диск. Непосредственно перед записью данных, мы меняем внешний вид курсора мыши, показывая занятость приложения. А после записи -- восстанавливаем его. В конце функции файл автоматически закрывается деструктором **QFile**.
QDataStream поддерживает основные типы языка C++, а так же ряд типов, определяемых библиотекой Qt. Синтаксис соответствует стандарту классов <**iostream**>. Например,

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

записывает переменные x, y и z в поток, а

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

читает их из потока.

Поскольку базовые типы языка C++ **char**, **short**, **int**, **long** и **long long** могут иметь различный размер на разных платформах, в целях безопасности их следует приводить к одному из следующих: **Q_INT8**, **Q_UINT8**, **Q_INT16**, **Q_UINT16**, **Q_INT32**, **Q_UINT32**, **Q_INT64**, **Q_UINT64**, которые гарантированно имеют декларируемый, в битах, размер.
QDataStream -- довольно универсальный класс, он может совместно работать не только с **QFile**, но так же и с **QBuffer**, и с **QSocket**, и с **QSocketDevice**. Точно так же **QFile** может использоваться совместно с **QTextStream** и как самостоятельное средство работы с файлами. В Главе 10 мы глубже рассмотрим эти классы.

Формат файлов приложения **Spreadsheet** очень прост. Начинается файл с 32-х битного числа, идентифицирующего формат (MagicNumber определена как 0x7F51C882 в **spreadsheet.h**). Далее следует серия блоков, каждый из которых соответствует одной ячейке (номера строки и колонки, и формула). Для экономии мы не записываем в файл пустые ячейки.

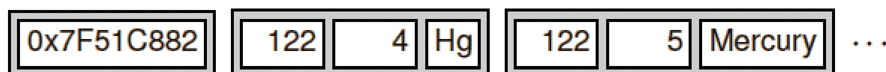


Рисунок 4.5. Формат файла Spreadsheet.

Двоичное представление типов данных определяется классом **QDataStream**. Например, тип **Q_UINT16** представлен двумя байтами, следующими в порядке **big-endian** (т.е. первым идет старший байт, потом -- младший). Тип **QString** записывается как последовательность символов в кодировке **Unicode**.
Двоичное представление типов, определяемых библиотекой Qt, стало осуществляться еще в Qt 1.0 и, вероятно, будет развиваться и дальше, чтобы в процессе развития библиотеки имелась возможность представлять в двоичном виде вновь появляющиеся типы. По-умолчанию, **QDataStream** использует самую современную версию двоичного формата (версия 5 в Qt 3.2), но способен работать и с более ранними версиями. Воизбежание проблем с совместимостью, на тот случай, если наша программа будет скомпилирована с более свежим выпуском Qt, мы укажем **QDataStream** на то, что необходимо использовать 5-ю версию, независимо от того, с какой версией Qt была скомпилирована программа.

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

```
  QDataStream in(&file);
  in.setVersion(5);

  Q_UINT32 magic;
  in >> magic;
  if (magic != MagicNumber) {
    QMessageBox::warning(this, tr("Spreadsheet"),
                         tr("The file is not a " "Spreadsheet file."));
    return false;
  }

  clear();

  Q_UINT16 row;
  Q_UINT16 col;
  QString str;

  QApplication::setOverrideCursor(waitCursor);
  while (!in.atEnd()) {
    in >> row >> col >> str;
    setFormula(row, col, str);
  }
  QApplication::restoreOverrideCursor();
  return true;
}
```

Функция **readFile()** очень похожа на **writeFile()**. Для работы с файлом опять используется **QFile**, только на этот раз при открытии файла устанавливается флаг режима доступа **IO_WriteOnly**. Далее идет установка версии формата. При чтении данных в двоичном представлении всегда должна указываться та же версия, которая использовалась при записи.
Если сигнатура файла (magic number) представлена корректным значением, то вызывается **clear()**, для очистки таблицы, поскольку в файле могут быть представлены не все ячейки.

4.4 Реализация меню Edit

Приступим к созданию слотов, соответствующих пунктам меню **Edit**.

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

Слот **cut()** соответствует пункту меню **Edit|Cut**. Реализация слота чрезвычайно проста, поскольку логика работы соответствует последовательности команд **Copy** и **Delete**.

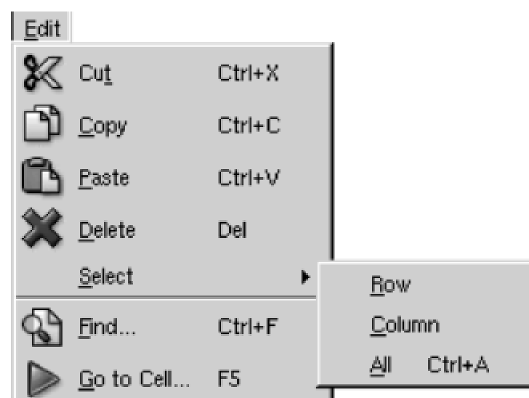


Рисунок 4.6. Меню Edit приложения Spreadsheet.

```
void Spreadsheet::copy()
{
  QTableSelection sel = selection();
  QString str;
```

```
  for (int i = 0; i < sel.numRows(); ++i) {
    if (i > 0)
      str += "\n";
    for (int j = 0; j < sel.numCols(); ++j) {
      if (j > 0)
        str += "\t";
      str += formula(sel.topRow() + i, sel.leftCol() + j);
    }
  }

  QApplication::clipboard()->setText(str);
}
```

Слот **copy()** соответствует пункту меню **Edit|Copy**. Здесь осуществляется обход ячеек в выделенной области. Формула каждой из выбранных ячеек добавляется к **QString**, где ячейки, находящиеся в одной строке, разделяются символом табуляции, а строки отделяются символом перевода строки.

	C	D	E
2	Red	Green	Blue
3	Cyan	Magenta	Yellow

"Red **\t** Green **\t** Blue **\n** Cyan **\t** Magenta **\t** Yellow"

Рисунок 4.7. Копирование выделенной области в буфер обмена.

Доступ к системному буферу обмена в Qt осуществляется через статическую функцию **QApplication::clipboard()**. Вызовом **QClipboard::setText()** мы помещаем содержимое **QString** в буфер обмена. Выбранный нами формат строки, где в качестве разделителя ячеек используется символ табуляции, а в качестве разделителя строк -- символ перевода строки, могут воспринимать и другие приложения, в том числе Microsoft Excel.

```
QTableSelection Spreadsheet::selection()
{
  if (QTable::selection(0).isEmpty())
    return QTableSelection(currentRow(), currentColumn(),
                           currentRow(), currentColumn());

  return QTable::selection(0);
}
```

Функция **selection()** возвращает границы выделенной области. Она обращается к **QTable::selection()**, которая возвращает выделенную область по ее номеру. Поскольку мы ранее установили режим выделения **Single**, то в нашем приложении может существовать только одна область выделения -- это область с номером 0. Но возможен вариант, когда в таблице нет выделенной области. **QTable** не рассматривает текущую ячейку как выделенную область. Это вполне разумно, но в данном случае -- немного неудобно. Поэтому, в случае, когда нет выделенной области, функция **selection()** вернет текущую ячейку.

```
void Spreadsheet::paste()
{
  QTableSelection sel = selection();
  QString str = QApplication::clipboard()->text();
  QStringList rows = QStringList::split("\n", str, true);
  int numRows = rows.size();
  int numCols = rows.first().contains("\t") + 1;

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

```
  }

  for (int i = 0; i < numRows; ++i) {
    QStringList cols = QStringList::split("\t", rows[i], true);
    for (int j = 0; j < numCols; ++j) {
      int row = sel.topRow() + i;
      int col = sel.leftCol() + j;
      if (row < NumRows && col < NumCols)
        setFormula(row, col, cols[j]);
    }
  }
  somethingChanged();
}
```

Слот **paste()** соответствует пункту меню **Edit|Paste**. Сначала принимается текст из буфера обмена. Затем он переносится в **QStringList**, с разбивкой по строкам, вызовом статической функции **QStringList::split()**.

Далее определяется размерность области копирования. Количество строк в таблице соответствует количеству строк в **QStringList**, а количество столбцов -- на один больше, чем количество символов табуляции в первой строке.

Если выбрана только одна ячейка, то она используется как верхний левый угол области вставки, если имеется выделенная область, то вставка осуществляется в нее.

В процессе вставки, каждая строка разбивается на ячейки, вызовом **QStringList::split()**, но на этот раз в качестве разделителя используется символ табуляции. Рисунок 4.8 демонстрирует процесс вставки данных в таблицу из буфера обмена.

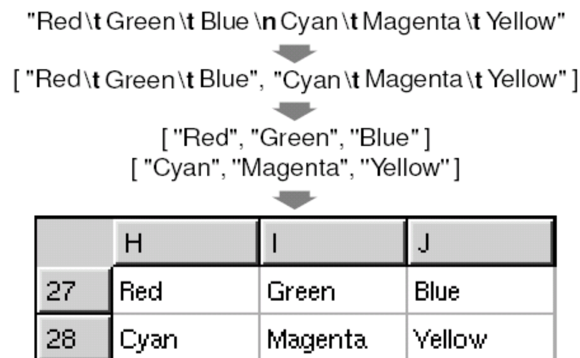


Рисунок 4.8. Вставка текста из буфера обмена в таблицу.

```
void Spreadsheet::del()
{
  QTableSelection sel = selection();
  for (int i = 0; i < sel.numRows(); ++i) {
    for (int j = 0; j < sel.numCols(); ++j)
      delete cell(sel.topRow() + i, sel.leftCol() + j);
    }
    clearSelection();
}
```

Слот **del()** соответствует пункту меню **Edit|Delete**. Для того, чтобы очистить ячейку, достаточно просто удалить объект **Cell**. Когда **QTable** обнаруживает удаление какого либо из **QTableItem**, то она автоматически перерисовывает себя на экране. Если после удаления ячейки вызвать **cell()**, то она вернет пустой указатель.

```
void Spreadsheet::selectRow()
{
  clearSelection();
  QTable::selectRow(currentRow());
}
```

```
void Spreadsheet::selectColumn()
{
  clearSelection();
  QTable::selectColumn(currentColumn());
}

void Spreadsheet::selectAll()
{
  clearSelection();
  selectCells(0, 0, NumRows - 1, NumCols - 1);
}
```

Слоты **selectRow()**, **selectColumn()**, **selectAll()** соответствуют пунктам меню **Edit|Select|Row**, **Edit|Select|Column**, **Edit|Select|All**. функциональность этих слотов основана на функциях **QTable**: **selectRow()**, **selectColumn()**, **selectCells()**.

```
void Spreadsheet::findNext(const QString &str, bool caseSensitive)
{
  int row = currentRow();
  int col = currentColumn() + 1;

  while (row < NumRows) {
    while (col < NumCols) {
      if (text(row, col).contains(str, caseSensitive)) {
        clearSelection();
        setCurrentCell(row, col);
        setActiveWindow();
        return;
      }
      ++col;
    }
    col = 0;
    ++row;
  }
  qApp->beep();
}
```

Слот **findNext()** начинает поиск с ячейки, стоящей справа от текущей и двигается вправо до конца строки. Затем переходит на следующую строку, продолжая поиск с первой ячейки следующей строки и так далее до тех пор, пока не будет найден искомый текст или пока не будет достигнут конец таблицы. Например, если текущая ячейка C27, то поиск начинается с ячейки D27 и далее проверяются ячейки E27, F27, ..., Z27, затем A28, B28, C28, ..., Z28 и так далее, до ячейки Z999. Как только искомый текст будет обнаружен -- сбрасывается выделение, курсор перемещается в ячейку, содержимое которой совпало с искомым текстом, и активизируется окно с таблицей. Если поиск не увенчался успехом, то выдается звуковой сигнал, извещающий о том, что искомый текст не найден.

```
void Spreadsheet::findPrev(const QString &str, bool caseSensitive)
{
  int row = currentRow();
  int col = currentColumn() - 1;

  while (row >= 0) {
    while (col >= 0) {
      if (text(row, col).contains(str, caseSensitive)) {
        clearSelection();
        setCurrentCell(row, col);
        setActiveWindow();
        return;
      }
      --col;
    }
    col = NumCols - 1;
    --row;
  }
  qApp->beep();
}
```

Слот **findPrev()** очень похож на **findNext()**, только поиск ведется в обратном направлении и заканчивается по достижении ячейки A1.

4.5 Реализация других меню

Теперь перейдем к рассмотрению реализации слотов меню **Tools** и **Options**.

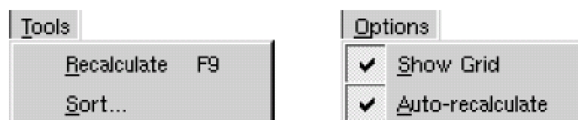


Рисунок 4.9. Меню Tools и Options.

```
void Spreadsheet::recalculate()
{
  int row;

  for (row = 0; row < NumRows; ++row) {
    for (int col = 0; col < NumCols; ++col) {
      if (cell(row, col))
        cell(row, col)->setDirty();
      }
    }
    for (row = 0; row < NumRows; ++row) {
      for (int col = 0; col < NumCols; ++col) {
        if (cell(row, col))
          updateCell(row, col);
      }
    }
}
```

Слот **recalculate()** соответствует пункту меню **Tools|Recalculate**. Он, кроме того, в случае необходимости, вызывается программой автоматически.
В первой группе циклов все ячейки помечаются, вызовом **setDirty()**, как требующие пересчета. В результате, когда в следующий раз **QTable** вызовет метод **text()** ячейки **Cell**, то ее содержимое будет пересчитано.

Вторая группа циклов выполняет **updateCell()** каждой ячейки, чтобы перерисовать ее. В результате, **QTable** вызовет метод **text()** ячейки **Cell**, чтобы получить ее значение, а поскольку все ячейки были помечены вызовом **setDirty()**, то будет выполнен пересчет.

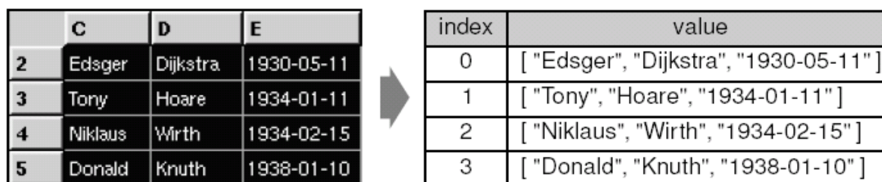
```
void Spreadsheet::setAutoRecalculate(bool on)
{
  autoRecalc = on;
  if (autoRecalc)
    recalculate();
}
```

Слот **setAutoRecalculate()** соответствует пункту меню **Options|Auto-recalculate**. Если этот флаг включен, то как только в таблице появляются какие либо изменения, выполняется автоматический пересчет всех ячеек таблицы. В этом случае, **recalculate()** вызывается из **somethingChanged()**.

```
void Spreadsheet::sort(const SpreadsheetCompare &compare)
{
  vector<QStringList> rows;
  QTableSelection sel = selection();
  int i;

  for (i = 0; i < sel.numRows(); ++i) {
    QStringList row;
    for (int j = 0; j < sel.numCols(); ++j)
      row.push_back(formula(sel.topRow() + i,
                    sel.leftCol() + j));
    rows.push_back(row);
```

```
  }

  stable_sort(rows.begin(), rows.end(), compare);

  for (i = 0; i < sel.numRows(); ++i) {
    for (int j = 0; j < sel.numCols(); ++j)
      setFormula(sel.topRow() + i, sel.leftCol() + j,
                 rows[i][j]);
  }
  clearSelection();
  somethingChanged();
}
```

Сортировка выполняется на выделенной области и переупорядочивает строки в соответствии с заданными ключами и порядком сортировки, хранящимися в объекте compare. Функция сортировки представляет каждую строку таблицы в виде **QStringList**, а выделенную область -- как массив строк. Класс **vector<?>** -- это стандартный класс C++, мы подробно опишем его в Главе 11. Для простоты будем выполнять сортировку по формулам, а не по значениям.

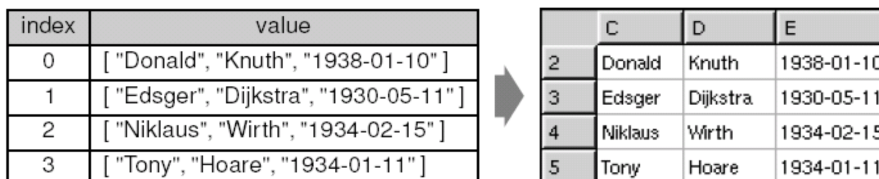


	C	D	E
2	Edsger	Dijkstra	1930-05-11
3	Tony	Hoare	1934-01-11
4	Niklaus	Wirth	1934-02-15
5	Donald	Knuth	1938-01-10

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

Рисунок 4.10. Сохранение выделенной области в виде массива строк.

Собственно сортировка выполняется стандартной, для C++, функцией **stable_sort()**. Она принимает начальный итератор, конечный итератор и функцию сравнения. Функция сравнения -- это такая функция, которая принимает два аргумента (два **QStringList**) и возвращает **true**, если первый аргумент "меньше чем" второй и **false** -- в противном случае. Объект **compare**, который мы передаем в **stable_sort()**, на самом деле не является функцией сравнения, но он может быть использован как таковая, а как -- мы вскоре увидим.



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

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

Рисунок 4.11. Перемещение отсортированных данных обратно в таблицу.

После того, как **stable_sort()** отсортирует массив строк, мы перемещаем данные обратно в таблицу, сбрасываем выделение и вызываем **somethingChanged()**.
В **spreadsheet.h**, класс **SpreadsheetCompare** определен как:

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

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

Это особый класс, поскольку он реализует оператор **()**, что позволяет использовать его так, как будто это обычная функция. Такие классы называют функторами (**functor**). Чтобы до конца понять принцип работы функторов, рассмотрим простой пример:

```
class Square
{
```

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

Класс **Square** реализует единственную функцию -- **operator()(int)**, которая возвращает квадрат входного аргумента. Такое именование функции, а скажем не **compute(int)**, дает нам возможность использовать экземпляр класса **Square** как обычную функцию:

```
  Square square;
  int y = square(5);
```

Теперь вернемся к классу **SpreadsheetCompare**:

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

Отсюда видно, что объект **compare** может использоваться как обычная функция **compare()**. Дополнительно, он имеет доступ к параметрам сортировки, которые хранятся в виде переменных-членов.

Альтернативный подход вынудил бы нас хранить параметры сортировки в глобальных переменных и использовать обычную функцию сравнения. Это очень неэлегантное решение, которое может породить трудноуловимые ошибки. Функторы -- это более мощная идиома взаимодействия с шаблонными функциями, такими как **stable_sort()**. Ниже приводится реализация функции, которая сравнивает две строки таблицы:

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

Она возвращает **true**, если первая строка "меньше" чем вторая, и **false** -- в противном случае. Стандартная функция **stable_sort()** использует результат сравнения для выполнения сортировки. Массивы **keys** и **ascending**, заполняются внутри функции **MainWindow::sort()** (описаной в Главе 2). Каждый ключ сортировки -- это индекс столбца или -1 (в случае "**None**").

Сравнению подвергаются части строк, соответствующие ячейкам, заданным ключами сортировки, с учетом порядка сортировки. В зависимости от того были найдены отличия или нет -- возвращается значение **true** или **false**. Если строки равны, то возвращается **false**.
На этом мы завершаем рассмотрение класса **Spreadsheet**. В следующем разделе мы обсудим класс **Cell**. Он используется для хранения формулы и реализует свой метод **text()**, который используется для получения значения ячейки.

4.6 Создание дочернего класса от QTableItem

Класс **Cell** порожден от класса **QTableItem**. Он предназначен для совместной работы с **Spreadsheet**, но никак не зависит от этого класса и теоретически может работать с любым экземпляром **QTable**.

Заголовочный файл:

```
#ifndef CELL_H
#define CELL_H

#include <qtable.h>
#include <qvariant.h>

class Cell : public QTableItem
{
public:
  Cell(QTable *table, const QString &formula);

  void setFormula(const QString &formula);
  QString formula() const;
  void setDirty();
  QString text() const;
  int alignment() const;

private:
  QVariant value() const;
  QVariant evalExpression(const QString &str, int &pos) const;
  QVariant evalTerm(const QString &str, int &pos) const;
  QVariant evalFactor(const QString &str, int &pos) const;

  QString formulaStr;
  mutable QVariant cachedValue;
  mutable bool cacheIsDirty;
};
#endif
```

Класс **Cell** расширяет функциональные возможности своего предка за счет добавления трех приватных переменных:

- **formulaStr** -- формула ячейки, **QString**.
- **cachedValue** -- кэш ячейки, **QVariant**.
- **cacheIsDirty** -- **true**, если значение в кэше необходимо обновить.

Переменные типа **QVariant** могут хранить значения самых разнообразных типов языка C++ и Qt. Мы используем ее по той простой причине, что ячейки в таблице могут хранить как числа типа **double**, так и строки **QString**.

Переменные **cachedValue** и **cacheIsDirty** объявлены со спецификатором **mutable**. Это позволяет модифицировать их из const-функций. В противном случае нам пришлось бы пересчитывать значение ячейки всякий раз, при вызове функции **text()**, но это было бы неэффективной тратой времени. Примечательно, что в определении класса отсутствует макрос **Q_OBJECT**. Дело в том, что **Cell** -- это обычный класс, который не имеет ни сигналов, ни слотов. Фактически, **QTableItem** не является наследником класса **QObject**, поэтому **Cell** не может иметь своих собственных сигналов и слотов. Вообще, классы элементов в Qt не являются потомками **QObject**, чтобы свести накладные расходы к минимуму. Если вам потребуются сигналы и слоты в классах-элементах, то вы можете реализовать свой виджет, который будет содержать элемент или, в исключительных случаях, воспользоваться возможностью множественного наследования, указав в качестве одного из предков класс **QObject**. Перейдем к файлу **cell.cpp**:

```
#include <qlineedit.h>
#include <qregexp.h>

#include "cell.h"

Cell::Cell(QTable *table, const QString &formula)
    : QTableItem(table, OnTyping)
{
  setFormula(formula);
}
```

Конструктор принимает указатель на **QTable** и формулу. Указатель на таблицу передается в унаследованный конструктор **QTableItem** и позднее может быть получен вызовом **QTableItem::table()**. Второй аргумент, передаваемый конструктору базового класса -- **OnTyping**, указывает, что компонент-редактор должен появляться сразу же, как только пользователь начнет вводить символы в текущую ячейку.

```
void Cell::setFormula(const QString &formula)
{
  formulaStr = formula;
  cacheIsDirty = true;
}
```

Функция **setFormula()** записывает формулу в ячейку. Она так же устанавливает флаг **cacheIsDirty**, который сигнализирует о том, что **cachedValue** должно быть пересчитано. Она вызывается из конструктора **Cell** и из **Spreadsheet::setFormula()**.

```
QString Cell::formula() const
{
  return formulaStr;
}
```

Функция `formula()` вызывается из `Spreadsheet::formula()`.

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

Функция **setDirty()** вызывается в случае, когда необходимо заставить ячейку пересчитать свое значение. Она просто устанавливает флаг **cacheIsDirty**. Пересчет выполняется только тогда, когда это действительно необходимо.

```
QString Cell::text() const
{
  if (value().isValid())
    return value().toString();
  else
    return "####";
}
```

Функция **text()** перекрывает метод **QTableItem**. Она возвращает текст, который должен быть отображен в таблице. Значение ячейки вычисляется функцией **value()**. Если оно не является допустимым (скорее всего из-за ошибки в формуле), то возвращается строка "**####**".

Функция **value()** возвращает значение типа **QVariant**. Этот тип может хранить значения самых разных типов, таких как **double** или **QString** и предоставляет в распоряжение программиста ряд методов преобразования вариантного типа в другие типы. Например, вызов **toString**, для варианта типа **double**, вернет его строковое представление.

```
int Cell::alignment() const
{
  if (value().type() == QVariant::String)
    return AlignLeft | AlignVCenter;
  else
    return AlignRight | AlignVCenter;
}
```

Функция **alignment()** перекрывает метод **QTableItem**. Она возвращает значение, характеризующее выравнивание текста в ячейке. В нашем случае для строк используется выравнивание по левому краю, для чисел -- по правому. Все значения, независимо от своего типа, центрируются по вертикали.

```
const QVariant Invalid;

QVariant Cell::value() const {
  if (cacheIsDirty) {
```

```
    cacheIsDirty = false;

    if (formulaStr.startsWith("'")) {
      cachedValue = formulaStr.mid(1);
    } else if (formulaStr.startsWith("=")) {
      cachedValue = Invalid;
      QString expr = formulaStr.mid(1);
      expr.replace(" ", "");
      int pos = 0;
      cachedValue = evalExpression(expr, pos);
      if (pos < (int)expr.length())
        cachedValue = Invalid;
    } else {
      bool ok;
      double d = formulaStr.toDouble(&ok);
      if (ok)
        cachedValue = d;
      else
        cachedValue = formulaStr;
    }
  }
  return cachedValue;
}
```

Приватная функция **value()** возвращает значение ячейки. Если установлен флаг **cacheIsDirty**, то значение ячейки пересчитывается.
Если формула начинается с одиночной кавычки (например, " '12345"), то в качестве значения возвращается часть строки, начиная с позиции 1 и до конца. (Одиночная кавычка занимает позицию 0.)
Если формула начинается с символа "=", то берется часть строки, начиная с позиции 1 и до конца, Из нее удаляются все пробелы. Затем производится вычисление по формуле, с помощью функции **evalExpression()**. Аргумент pos, передаваемый по ссылке, указывает -- с какого символа в строке необходимо начинать разбор выражения. По окончании работы функции он содержит позицию символа, на котором завершился разбор. Если pos не соответствует позиции последнего символа в строке, то это означает ошибку в выражении и в этом случае **cachedValue** будет содержать значение **Invalid**.
Если формула начинается не с символа "=" и не с одиночной кавычки, то делается попытка преобразовать строку в число с плавающей точкой. Если преобразование завершилось успешно, то в **cachedValue** записывается число типа **double**, в противном случае -- строка с формулой. Например, формула "1.50" будет благополучно преобразована в число 1.5, а формула "World Population" не может быть преобразована в число и в этом случае в **cachedValue** будет записана сама строка "World Population".

Функция **value()** -- это const-функция. Но благодаря тому, что переменные-члены **cachedValue** и **cacheIsValid** объявлены как **mutable**, компилятор позволит нам модифицировать их внутри функции. Вам может показаться, что достаточно убрать спецификатор const функции **value()** и можно будет отказаться от спецификатора mutable, для переменных **cachedValue** и **cacheIsValid**, но такой вариант все равно породит ошибку времени компиляции, поскольку **value**() вызывается из const-функции **text()**. Вообще, в мире C++, кэширование и **mutable** идут рядом, рука об руку.
Мы практически закончили рассмотрение приложения **Spreadsheet**. Осталось только разобраться с синтаксическим анализом формул. Далее, до конца этого раздела, мы сконцентрируемся на **evalExpression()** и двух вспомогательных функциях **evalTerm()** и **evalFactor()**. Реализация функций достаточно сложна, но они совершенно необходимы для нашего приложения. С другой стороны, поскольку эти функции напрямую не связаны с разработкой графического интерфейса, вы смело можете пропустить оставшуюся часть раздела и сразу перейти к Главе 5.

Функция **evalExpression()** возвращает результат вычисления выражения. Выражение -- это один или более термов (**term**), отделяемых друг от друга операторами '+' или '-', например, "2*C5+D6" -- это выражение, состоящее из термов "2*C5" и "D6". Термы, в свою очередь, могут состоять из одного или более факторов (**factor**), отделяемых друг от друга операторами '*' или '/', например, терм "2*C5" состоит из двух факторов -- "2" и "C5". И наконец, фактор может быть числом ("2"), адресом ячейки ("C5") или выражением в скобках с необязательным предшествующим знаком '-' (признак

отрицательного числа). Разложив выражение на термы, а термы на факторы, мы получим правильную обработку приоритетов операций.

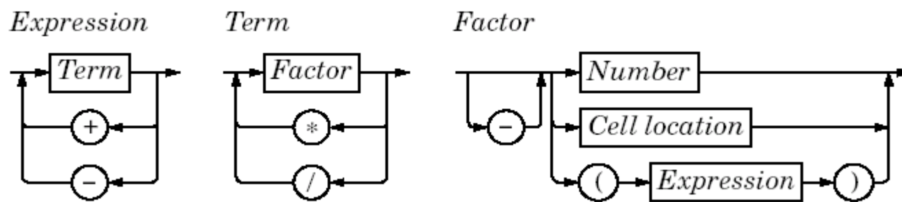


Рисунок 4.12. Синтаксическая диаграмма выражений в электронной таблице.

Синтаксическая диаграмма выражений приведена на рисунке 4.12. Каждому из элементов грамматики (**Expression**, **Term** и **Factor**) сопоставлена своя функция-член класса **Cell**, которая выполняет синтаксический анализ этих элементов и чья структура очень близко следует грамматике. Синтаксические анализаторы подобного типа называются анализаторами рекурсивного спуска. Начнем с функции **evalExpression()**, которая отвечает за разбор элемента **Expression**:

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

    QVariant term = evalTerm(str, pos);
    if (result.type() == QVariant::Double
       && term.type() == QVariant::Double) {
      if (op == '+')
        result = result.toDouble() + term.toDouble();
      else
        result = result.toDouble() - term.toDouble();
    } else {
      result = Invalid;
    }
  }
  return result;
}
```

В первой строке, вызовом **evalTerm()**, предпринимается попытка получить значение первого терма. Если за ним стоит оператор '+' или '-', то **evalTerm()** вызывается второй раз, в противном случае, выражение состоит из единственного терма и мы возвращаем его значение как результат выражения. После того, как будут получены значения обоих термов -- вычисляется результат операции. Если оба терма имеют тип **double**, вычисляется результат этого же типа, в противном случае, возвращается результат **Invalid**.
Так продолжается до тех пор, пока не будут исчерпаны все термы. В данной ситуации все работает корректно, благодаря тому, что операции сложения и вычитания лево-ассоциативны, т.е. выражению "1-2-3" соответствует "(1-2)-3", а не "1-(2-3)".

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

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

```
            if (factor.toDouble() == 0.0)
              result = Invalid;
            else
              result = result.toDouble() / factor.toDouble();
          }
    } else {
      result = Invalid;
    }
  }
  return result;
}
```

evalTerm() очень похожа на **evalExpression()**, за исключением того, что она обслуживает операции умножения и деления. Единственный тонкий момент -- необходимо избежать выполнения деления на ноль. Вообще нецелесообразно проверять на равенство значения с плавающей точкой, поскольку могут возникнуть ошибки, связанные с погрешностью округления, хотя в данном случае, выполнять такую проверку вполне допустимо.

```
QVariant Cell::evalFactor(const QString &фьзжstr, int &фьзжpos) const
{
  QVariant result;
  bool negative = false;

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

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

    while (str[pos].isLetterOrNumber() || str[pos] == '.') {
      token += str[pos];
      ++pos;
    }

    if (regExp.exactMatch(token)) {
      int col = token[0].upper().unicode() - 'A';
      int row = token.mid(1).toInt() - 1;

      Cell *c = (Cell *)table()->item(row, col);
      if (c)
        result = c->value();
      else
        result = 0.0;
    } else {
      bool ok;
      result = token.toDouble(&ok);
      if (!ok)
        result = Invalid;
    }
  }

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

Функция **evalFactor()** гораздо сложнее, чем **evalExpression()** и **evalTerm()**. Начинается она с проверки -- не инвертирован ли фактор (наличие унарного минуса). Затем проверяется -- не начинается ли он с открывающей скобки. Если да, то содержимое скобок вычисляется как выражение, вызовом **evalExpression()**. Это то самое место, где возникает рекурсия -- **evalExpression()** вызывает **evalTerm()**, которая вызывает **evalFactor()**, которая опять вызывает **evalExpression()**.

Если фактор не является выражением в скобках, то извлекается лексема, которая может оказаться адресом ячейки или числом. Если лексема соответствует регулярному выражению **QRegExp**, то она воспринимается как адрес ячейки и вызывается **value()** для данной ячейки. Ячейка может находиться в любом месте электронной таблицы, а ее значение может так же вычисляться на основе других ячеек. Подобные зависимости не являются проблемой для нас, просто это может потребовать некоторого дополнительного времени для рассчета значений тех ячеек, у которых установлен флаг **cacheIsDirty**. Если лексема не является адресом ячейки, то она считается числом.

Что произойдет, если значение ячейки **A1** вычисляется по формуле "**=A1**"? Или если ячейка **A1** вычисляется по формуле "**=A2**", а ячейка **A2** -- по формуле "**=A1**"? Хотя мы и не предусмотрели проверки циклических зависимостей, тем не менее, наш анализатор довольно изящно решает эту проблему, возвращая ошибочный **QVariant**. Это происходит потому, что в функции **value()** сбрасывается флаг **cacheIsDirty**, а в **cachedValue** записывается **Invalid** до того, как будет вызвана функция **evalExpression()**. Если **evalExpression()** рекурсивно вызывает **value()** своей собственной ячейки, то ей сразу же возвращается значение **Invalid**, которое становится результатом всего выражения.

На этом мы завершаем обсуждение синтаксического анализатора формул. Он может быть расширен за счет введения обработки предопределенных функций электронной таблицы, таких как "**sum()**" и "**avg()**", в синтаксическом элементе фактор. Довольно просто в него можно добавить операцию конкатенации ("**+**") строк.

5 Создание собственных виджетов

В этой главе мы расскажем -- как создаются визуальные компоненты (виджеты) в Qt. Визуальные компоненты могут создаваться путем наследования существующих виджетов Qt или напрямую -- от **QWidget**. Мы продемонстрируем оба варианта, а так же рассмотрим -- как можно интегрировать свои компоненты в Qt Designer. И в завершение главы представим компонент, который использует прием двойной буферизации для устранения эффекта мерцания во время перерисовки.

5.1 Переделка существующих виджетов Qt

Иногда возникает необходимость в расширении функциональных возможностей стандартных виджетов. Самое простое решение -- это создать класс потомок от соответствующего виджета Qt и наделить его необходимыми свойствами.

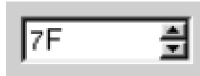


Рисунок 5.1. Виджет HexSpinBox.

В этом разделе мы продемонстрируем виджет шестнадцатиричного счетчика. Стандартный виджет **QSpinBox** поддерживает только десятичный формат представления чисел, но, за счет создания дочернего класса, его можно "заставить" принимать и обрабатывать шестнадцатиричный формат.

```
#ifndef HEXSPINBOX_H
#define HEXSPINBOX_H

#include <qspinbox.h>

class HexSpinBox : public QSpinBox {
public:
  HexSpinBox(QWidget *parent, const char *name = 0);

protected:
  QString mapValueToText(int value);
  int mapTextToValue(bool *ok);
};
#endif
```

Большую часть своих функциональных возможнойстей, виджет **HexSpinBox** наследует от **QSpinBox**. Он имеет типичный конструктор и перекрывает две виртуальные функции своего предка. Поскольку класс **HexSpinBox** не определяет своих собственных сигналов и слотов, то он не нуждается в макроопределении **Q_OBJECT**.

```
#include <qvalidator.h>

#include "hexspinbox.h"

HexSpinBox::HexSpinBox(QWidget *parent, const char *name)
    : QSpinBox(parent, name)
{
  QRegExp regExp("[0-9A-Fa-f]+");
  setValidator(new QRegExpValidator(regExp, this));
  setRange(0, 255);
}
```

Пользователь может изменять значение счетчика либо щелкая по кнопкам со стрелками, либо вводя числа в окошко редактора. В последнем случае мы должны ограничить набор допустимых символов шестнадцатиричными цифрами. Для этого используется **QRegExpValidator**, который пропускает только символы из диапазонов (0..9), (A..F) и (a..f). Дополнительно задается диапазон изменения чисел -- от 0 по 255 (от 0x00 по 0xFF), который больше подходит для шестнадцатиричных чисел, чем диапазон (0..99), устанавливаемый **QSpinBox** по-умолчанию .

```
QString HexSpinBox::mapValueToText(int value)
{
```

```
  return QString::number(value, 16).upper();
}
```

Функция **mapValueToText()** преобразует число в строку. Она используется для обновления окошка редактора, когда пользователь изменяет число нажатием на кнопки "вверх" и "вниз". Собственно преобразование выполняется функцией **QString::number()**, которой вторым аргументом передается число 16 -- основание системы счисления. Она возвращает шестнадцатиричное представление числа с символами в нижнем регистре, а вызов **QString::upper()** переводит их в верхний регистр.

```
int HexSpinBox::mapTextToValue(bool *ok)
{
  return text().toInt(ok, 16);
}
```

Функция **mapTextToValue()** выполняет обратное преобразование -- из строки в число. Она вызывается, когда пользователь вводит число с клавиатуры и завершает его нажатием на клавишу **Enter**.
Собственно преобразование выполняется функцией **QString::toInt()**, которая принимает строку (возвращаемую вызовом **QString::toInt()**) и число 16 -- основание системы счисления.
Если преобразование было выполнено успешно, то **QString::toInt()** запишет в аргумент *ok значение **true** и **false** -- в противном случае. Это полностью соответствует тому, чего ожидает **QSpinBox**.

Это собственно все, что мы хотели рассказать о **HexSpinBox**. Расширение возможностей других виджетов Qt выполняется аналогичным образом: выбирается необходимый виджет, создается класс-потомок и перекрываются некоторые виртуальные функции, изменяющие поведение класса-предка. Это общепринятая в Qt техника программирования. Фактически мы с ней уже сталкивались в Главе 4, когда создавали класс-потомок от **QTable** и перекрывали методы **createEditor()** и **endEdit()**.

5.2 Создание класса-потомка от QWidget

В большинстве своем, нестандартные виджеты образуются за счет комбинирования существующих компонентов, как встроенных виджетов Qt, так и других нестандартных виджетов, таких как **HexSpinBox**. Нестандартные визуальные компоненты, которые состоят из существующих виджетов, как правило могут разрабатываться в среде Qt Designer. Для этого:

- Создается новая форма, по шаблону "**Widget**".
- На форму добавляются необходимые виджеты.
- Устанавливаются соединения между сигналами и слотами и добавляется необходимый код (либо в файл **.ui.h**, либо в класс реализации).

Есстественно, все это может быть сделано и вручную. Но какой бы подход вы ни выбрали, в конечном итоге новый класс является наследником **QWidget**.

Если виджет не имеет собственных сигналов и слотов, и не перекрывает методов родителя, то возможна простая сборка виджета путем аггрегирования существующих виджетов, без создания класса-потомка. Такой подход использовался нами в Главе 1, при создании приложения "Age", когда мы просто "собрали" его из трех компонентов: **QHBox**, **QSpinBox** и **QSlider**. Но даже в этом случае можно было бы породить дочерний класс от **QHBox** и в его конструкторе создать виджеты **QSpinBox** и **QSlider**.

Если среди виджетов Qt нет ни одного, подходящего под имеющуюся задачу, и при этом нет таких виджетов, с помощью которых можно было бы собрать свой компонент, то у нас остается единственная возможность -- создать класс-потомок от **QWidget** и реализовать в нем необходимые обработчики событий и функции отрисовки. Этот подход дает нам абсолютную свободу в определении внешнего вида и поведения нового компонента. Многие виджеты Qt, например: **QLabel**, **QPushButton** и **QTable** реализованы именно таким способом.
С целью демонстрации этого подхода, мы создадим свой виджет **IconEditor**, который может использоваться в программе редактирования иконок.
Как обычно, начнем с файла заголовка:

```
#ifndef ICONEDITOR_H
#define ICONEDITOR_H
```

```
#include <qimage.h>
#include <qwidget.h>

class IconEditor : public QWidget
{
  Q_OBJECT
  Q_PROPERTY(QColor penColor READ penColor WRITE setPenColor)
  Q_PROPERTY(QImage iconImage READ iconImage WRITE setIconImage)
  Q_PROPERTY(int zoomFactor READ zoomFactor WRITE setZoomFactor)

public:
  IconEditor(QWidget *parent = 0, const char *name = 0);
  void setPenColor(const QColor &newColor);
  QColor penColor() const { return curColor; }
  void setZoomFactor(int newZoom);
  int zoomFactor() const { return zoom; }
  void setIconImage(const QImage &newImage); const
  QImage &iconImage() const { return image; }
  QSize sizeHint() const;
```

Класс **IconEditor** использует макрос **Q_PROPERTY**, для объявления свойств **penColor**, **iconImage** и **zoomFactor**. Каждое из свойств имеет свой тип и функции "чтения" и "записи" ("read" и "write"). Например, свойство **penColor** имеет тип **QColor** и функции "чтения"/"записи" **-- penColor()** и **setPenColor()**, соответственно.

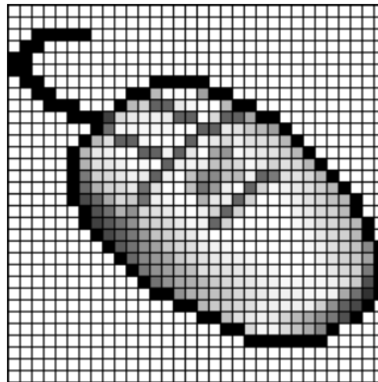


Рисунок 5.2. Виджет IconEditor.

Когда мы будем работать с виджетом в Qt Designer, то эти свойства появятся в инспекторе свойств, сразу же после свойств, унаследованных от **QWidget**. Свойства могут иметь любой тип, который поддерживает **QVariant**. Макроопределение **Q_PROPERTY** необходимо вставлять в классы, которые определяют свойства.

```
protected:
  void mousePressEvent(QMouseEvent *event);
  void mouseMoveEvent(QMouseEvent *event);
  void paintEvent(QPaintEvent *event);

private:
  void drawImagePixel(QPainter *painter, int i, int j);
  void setImagePixel(const QPoint &pos, bool opaque);

  QColor curColor;
  QImage image;
  int zoom;
};
#endif
```

Наш виджет перекрывает три защищенные функции своего предка и добавляет несколько приватных функций и переменных. Эти три приватные переменные хранят значения трех свойств, которые были определены чуть выше.
Файл реализации начинается с директив подключения заголовочных файлов и конструктора класса **IconEditor**:

```
#include <qpainter.h>

#include "iconeditor.h"

IconEditor::IconEditor(QWidget *parent, const char *name)
    : QWidget(parent, name, WStaticContents)
{
  setSizePolicy(QSizePolicy::Minimum, QSizePolicy::Minimum);
  curColor = black;
  zoom = 8;
  image.create(16, 16, 32);
  image.fill(qRgba(0, 0, 0, 0));
  image.setAlphaBuffer(true);
}
```

В конструкторе имеется ряд моментов, такие как -- вызов **setSizePolicy()** и передача флага **WStaticContents** унаследованному конструктору, к которым мы вскоре вернемся.

В переменную **zoom** записывается число 8. Это означает, что каждый пиксель иконки будет отображаться в виде квадрата 8 X 8. Устанавливается черный цвет "чернил", символ black -- это предопределенная константа в Qt. Сама иконка хранится в переменной **image**, доступ к которой осуществлен посредством функций **setIconImage()** и **iconImage()**. Программа-редактор должна вызывать **setIconImage()**, когда пользователь открывает файл с иконкой, и **iconImage()** -- когда пользователь сохраняет иконку в файл.

Переменная **image** имеет тип **QImage**. При инициализации мы задаем ей размер 16 X 16 и глубину цвета -- 32 бита, затем очищаем ее и разрешаем альфа-буфер.
Класс **QImage** хранит изображения в платформо-независимом виде. Глубина цвета может быть выбрана одной из следующих: 1 бит, 8 бит или 32 бита. Изображения с 32-х битной глубиной цвета используют по 8 бит на каждый цветовой канал -- красный, зеленый и синий, для каждого пикселя. Оставшиеся 8 бит определяют значение альфа-составляющей пикселя -- степень прозрачности. Например, пиксель чистого красного цвета должен иметь значения цветовых (красный, зеленый, синий) и альфа каналов -- 255, 0, 0, 255. В Qt этот цвет может быть задан как:

```
QRgb red = qRgba(255, 0, 0, 255);
```

или как:

```
QRgb red = qRgb(255, 0, 0);
```

Тип **QRgb** определен как **unsigned int**, а **QRgb()** и **QRgba()** -- это inline-функции, которые составляют 32-х битное значение цвета из своих аргументов. Допустимо определять цвет таким образом:

```
QRgb red = 0xFFFF0000;
```

где первая пара символов FF соответствует альфа-составляющей, а вторая пара FF -- красной составляющей цвета. В конструкторе **IconEditor** мы заполнили **QImage** прозрачным цветом, т.е. в качестве значения альфа-составляющей указали число 0.
В Qt имеется два типа для хранения значения цвета -- **QRgb** и **QColor**. **QRgb** -- это лишь тип, определенный через **typedef**, который используется **QImage** для хранения значения цвета, а **QColor** -- это полноценный класс, со множеством полезных функций, который широко используется в Qt. В нашем случае, мы будем использовать **QRgb**, когда будем иметь дело с **QImage** и **QColor** во всех остальных случаях, включая свойство **penColor**.

```
QSize IconEditor::sizeHint() const
{
  QSize size = zoom * image.size();
  if (zoom >= 3)
    size += QSize(1, 1);
  return size;
}
```

Функция **sizeHint()** перекрывает метод класса-родителя и возвращает "идеальный" размер виджета. Она умножает размер изображения на масштабный коэффициент (zoom). Если масштабный коэффициент больше 3, то добавляется по одному пикселу, в каждой из координатных осей, чтобы имелась возможность разместить координатную сетку (Сетка не отображается, если коэффициент равен 2 или 1).

Идеальный размер виджета главным образом используется в целях размещения компонента на форме. Менеджеры размещения в Qt всегда пытаются выделить виджету тот объем площади на форме, который наиболее близко соответствует идельному размеру виджета.
В дополнение к идеальному размеру, виджет имеет политику изменения размера, которая сообщает менеджеру размещения -- может ли виджет растягиваться или сжиматься. Вызовом **setSizePolicy()** мы указали политику изменения размеров в обоих направлениях, как **QSizePolicy::Minimum**. Тем самым, виджет сообщает менеджерам размещения о том, что идеальный размер является минимально возможным или, говоря другими словами, виджет может быть растянут, но никогда не должен сжиматься меньше идеальных размеров. Это поведение может быть изменено в Qt Designer, установкой свойства **sizePolicy** виджета. Смысл и назначение различных политик управления размерами будут обсуждаться в Главе 6.

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

Функция **setPenColor()** устанавливает текущий цвет "чернил", который используется для "закрашивания" пикселей.

```
void IconEditor::setIconImage(const QImage &newImage)
{
  if (newImage != image) {
    image = newImage.convertDepth(32);
    image.detach();
    update();
    updateGeometry();
  }
}
```

Функция **setIconImage()** подготавливает новое изображение к редактированию. Вызов **convertDepth()** устанвливает глубину цвета равной 32-м битам, поскольку мы везде исходим из предположения, что изображение имеет 32-х битную глубину цвета.
Затем вызывется **detach()**, для получения полной копии изображения. Это совершенно необходимо, поскольку **QImage** пытается сэкономить память и время, копируя изображение только в том случае, когда его явно попросят об этом. Такая оптимизация называется явное совместное использование. Она будет подробно обсуждаться в разделе Контейнеры указателей, Главы 11 .

После того, как изображение будет скопировано, мы вызываем **QWidget::update()**, чтобы перерисовать виджет. Затем вызывается **QWidget::updateGeometry()**, чтобы сообщить менеджеру размещения о том, что идеальный размер виджета изменился. После чего будет выполнена автоматическая перекомпоновка виджетов, с учетом нового идеального размера.

```
void IconEditor::setZoomFactor(int newZoom)
{
  if (newZoom < 1)
    newZoom = 1;
  if (newZoom != zoom) {
    zoom = newZoom;
    update();
    updateGeometry();
  }
}
```

Функция **setZoomFactor()** устанавливает масштабный коэффициент изображения. Для предотвращения деления на ноль, все значения меньше 1 корректируются. Если масштабный

коэффициент действительно изменился, то вызываются **update()** и **updateGeometry()**, чтобы перерисовать виджет и известить менеджеров размещения об изменении идеального размера. Функции **penColor()**, **iconImage()** и **zoomFactor()** реализованы в виде inline-функций в файле заголовка. Теперь перейдем к функции **paintEvent()**. Это самая важная функция. Она вызывается, когда необходимо перерисовать виджет. Ее реализация в **QWidget** фактически ничего не делает, оставляя на месте виджета пустое пространство.

Аналогично функциям **contextMenuEvent()** и **closeEvent()**, с которыми мы сталкивались в Главе 3, функция **paintEvent()** является обработчиком события. В Qt, для обработки любого вида события, предусматривается своя функция-обработчик. Обработка событий более подробно будет обсуждаться в Главе 7.

Существует несколько ситуаций, когда возникает событие **paint** и вызывается **paintEvent()**:

- Когда виджет выводится на экране самый первый раз, система автоматически генерирует событие **paint**, чтобы заставить виджет нарисовать себя.
- При изменении размера виджета система так же генерирует это событие.
- Если виджет был закрыт другим окном, а затем опять открылся, то генерируется событие **paint** для той области, которая была скрыта (если оконная система не сохранила эту область).

Событие так же порождается в результате вызова **QWidget::update()** или **QWidget::repaint()**. Отличия между ними заключаются в том, что **repaint()** вызывает немедленную перерисовку, а **update()** просто ставит событие **paint** в очередь, которая обрабатывается библиотекой Qt. (Обе функции ничего не делают, если виджет невидим на экране.) Если **update()** вызывается несколько раз, то Qt помещает в очередь только одно событие paint. В виджете **IconEditor** мы всегда будем использовать только функцию **update()**.

```
void IconEditor::paintEvent(QPaintEvent *)
{
  QPainter painter(this);

  if (zoom >= 3) {
    painter.setPen(colorGroup().foreground());
    for (int i = 0; i <= image.width(); ++i)
      painter.drawLine(zoom * i, 0,
                       zoom * i, zoom * image.height());
    for (int j = 0; j <= image.height(); ++j)
      painter.drawLine(0, zoom * j,
                       zoom * image.width(), zoom * j);
  }

  for (int i = 0; i < image.width(); ++i) {
    for (int j = 0; j < image.height(); ++j)
      drawImagePixel(&painter, i, j);
  }
}
```

Обработка события начинается с создания объекта **QPainter**. Если масштабный коэффициент больше 2, то рисуются вертикальная и горизонтальная линии, формирующие сетку, с помощью функции **QPainter::drawLine()**.

Функция **QPainter::drawLine()** имеет следующий синтаксис вызова:

```
painter.drawLine(x1, y1, x2, y2);
```

где (x1, y1) -- это координаты начала, а (x2, y2) -- координаты конца линии. Имеется перегруженная версия этой функции, которая принимает координаты в виде двух **QPoint**.
Верхний левый пиксель виджета, в Qt, имеет координаты (0, 0), правый нижний пиксель -- (width()-1, height-1). То есть, по сути, обычная Декартова система координат, с небольшим отличием -- ось OY направлена вниз, что имеет определенный смысл при программировании графического интерфейса. Система координат в **QPainter** может быть подвергнута таким трансформациям, как трансляция, масштабирование, вращение и сдвиг. Более подробно мы обсудим эту тему в Главе 8.

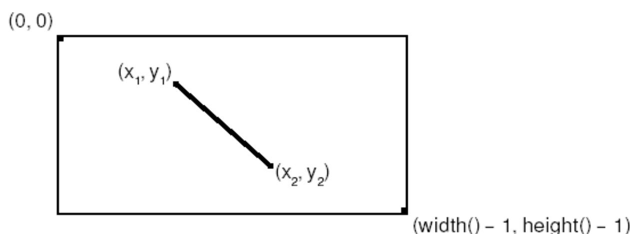


Рисунок 5.3. Пример рисования линии с помощью QPainter.

Прежде чем нарисовать линию, устанавливается цвет "чернил", вызовом **setPen()**. Можно было бы жестко "зашить" цвет в исходном коде, например black или gray, но лучше использовать палитру виджета.

Любой виджет снабжается своей собственной палитрой цветов, которая определяет -- какой цвет для каких целей используется. Например, в палитре есть запись, которая определяет цвет фона (обычно светло-серый), есть запись, которая определяет цвет текста (обычно черный). Как правило, палитра содержит цвета, соответствующие системной цветовой схеме. Используя палитру виджета, можно быть уверенным, что учитываются цветовые предпочтения пользователя.

Палитра содержит в себе три основные группы цветов: активные, неактивные и запрещенные. Решение о том, какую группу цветов использовать, зависит от текущего состояния виджета:

- Группа активных цветов используется, когда окно, на котором размещается виджет, активно.
- Группа неактивных цветов используется, когда окно, на котором размещается виджет, неактивно.
- Группа запрещенных цветов используется, когда доступ к виджету запрещен.

Функция **QWidget::palette()** возвращает палитру виджета в виде экземпляра класса **QPalette**. Доступ к отдельным цветовым группам, имеющим тип **QColorGroup**, осуществляется через функции **active()**, **inactive()** и **disabled()**. Для удобства, в класс **QWidget** была введена функция **colorGroup()**, которая возвращает ту или иную цветовую группу, в зависимости от состояния виджета, благодаря этому, вам довольно редко придется напрямую обращаться к палитре.
Функция **paintEvent()** завершается перерисовкой самого изображения, вызовом **IconEditor::drawImagePixel()**, которая отрисовывает каждый пиксель иконки в виде закрашенного квадрата.

```
void IconEditor::drawImagePixel(QPainter *painter, int i, int j)
{
  QColor color;
  QRgb rgb = image.pixel(i, j);

  if (qAlpha(rgb) == 0)
    color = colorGroup().base();
  else
    color.setRgb(rgb);

  if (zoom >= 3) {
    painter->fillRect(zoom * i + 1, zoom * j + 1,
                      zoom - 1, zoom - 1, color);
  } else {
    painter->fillRect(zoom * i, zoom * j,
                      zoom, zoom, color);
  }
}
```

Функция **drawImagePixel()** рисует пиксели средствами **QPainter**, с учетом масштабного коэффициента. Параметры i и j -- это координаты пикселя в системе координат **QImage**, но не в системе координат виджета (если масштабный коэффициент равен 1, то эти две системы координат полностью совпадают). Если пиксель прозрачен (альфа-составляющая равна 0), то для рисования пикселя используется цвет "base" текущей группы (обычно -- белый). В противном случае -- используется цвет пикселя в **QImage**. Затем вызывается **QPainter::fillRect()**, которая рисует закрашенный квадрат. Если поверх изображения рисуется координатная сетка, то размер квадрата уменьшается на 1 по обеим осям.

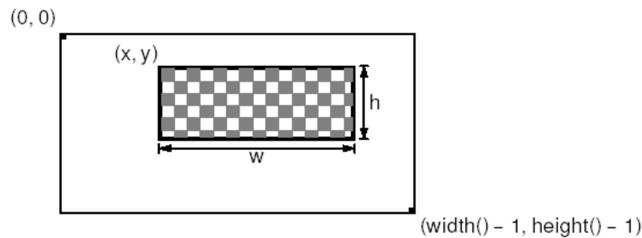


Рисунок 5.4. Пример рисования прямоугольника с помощью QPainter.

Функция **QPainter::fillRect()** имеет следующий синтаксис:

```
painter->fillRect(x, y, w, h, brush);
```

где (x, y) -- координаты левого верхнего угла прямоугольника, w x h -- его размеры, а **brush** задает цвет заполнения и шаблон заполнения. Передавая **QColor**, в качестве аргумента **brush**, мы задаем сплошной режим закрашивания.

```
void IconEditor::mousePressEvent(QMouseEvent *event)
{
  if (event->button() == LeftButton)
    setImagePixel(event->pos(), true);
  else if (event->button() == RightButton)
    setImagePixel(event->pos(), false);
}
```

Когда пользователь нажимает кнопку мыши, система генерирует событие "mouse press". За счет перекрытия метода родителя **QWidget::mousePressEvent()**, мы получаем возможность перехватывать и обрабатывать это событие, закрашивая или очищая пиксель в изображении, находящийся под указателем мыши.
Когда пользователь щелкает левой кнопкой мыши, вызывается приватная функция **setImagePixel()** с аргументом **true**, сообщая о том, что пиксель должен быть закрашен текущим цветом "чернил". Если пользователь щелкает правой кнопкой мыши, то в функцию **setImagePixel()** передается аргумент **false** и пиксель очищается.

```
void IconEditor::mouseMoveEvent(QMouseEvent *event)
{
  if (event->state() & LeftButton)
    setImagePixel(event->pos(), true);
  else if (event->state() & RightButton)
    setImagePixel(event->pos(), false);
}
```

Функция **mouseMoveEvent()** обрабатывает событие "mouse move" (перемещение указателя мыши). По-умолчанию это событие возникает только в том случае, когда пользователь перемещает указатель мыши при нажатой, и удерживаемой в нажатом состоянии, кнопке. Но имеется возможность изменить это поведение, вызовом **QWidget::setMouseTracking()**, однако в данном примере нам этого не требуется. Аналогично предыдущему обработчику, в зависимости от того, какая кнопка мыши нажата, пиксели либо закрашиваются, либо очищаются. Поскольку возможна ситуация, когда пользователь нажал и удерживает сразу две кнопки -- значение, возвращаемое **QMouseEvent::state()**, представляет собой битовую карту, в которой каждой из кнопок мыши соответствует свой бит (в этой карте так же есть биты, определяющие состояние клавиш **Shift** и **Ctrl** на клавиатуре). Проверка факта нажатия на ту или иную клавишу, выполняется с помощью оператора **&**. Если клавиша нажата, то вызывается **setImagePixel()**.

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

  if (image.rect().contains(i, j)) {
    if (opaque)
```

```
        image.setPixel(i, j, penColor().rgb());
      else
        image.setPixel(i, j, qRgba(0, 0, 0, 0));

      QPainter painter(this);
      drawImagePixel(&painter, i, j);
  }
}
```

Функция **setImagePixel()** вызывается из обработчиков **mousePressEvent()** и **mouseMoveEvent()** для закрашивания или очистки пикселя. Параметр pos определяет позицию указателя мыши в системе координат виджета.

На первом этапе выполняется переход от системы координат виджета к системе координат изображения. Переход осуществляется делением координат указателя мыши x и y на коэффициент масштабирования. Затем проверяется -- находятся ли координаты точки в допустимом диапазоне. Проверка выполняется с помощью **QImage::rect()** и **QRect::contains()**, которые проверяют попадание i в диапазон 0..image.width()-1 и попадание j в диапазон 0..image.height()-1.

В зависимости от параметра **opaque**, пиксель в изображении либо окрашивается в заданный цвет, либо очищается. "Очистка" пикселя заключается в том, что он делается прозрачным. В конце вызывается **drawImagePixel()** для перерисовки пикселя.

Теперь мы вернемся к флагу **WStaticContents**, который мы передавали родительскому конструктору. Этот флаг сообщает Qt, что содержимое виджета не изменяется, при увеличении размеров виджета, и всегда находится в верхнем левом углу. Qt использует эту информацию, чтобы избежать напрасной перерисовки областей, которые уже видны, при увеличении размеров виджета.
Когда размеры виджета изменяются, Qt обычно генерирует событие paint для всей видимой области виджета. Но, если виджет был создан с флагом **WStaticContents**, то действие события ограничивается пикселями, которые ранее не были показаны. Если же размеры виджета уменьшаются, то событие paint вообще не возникает.

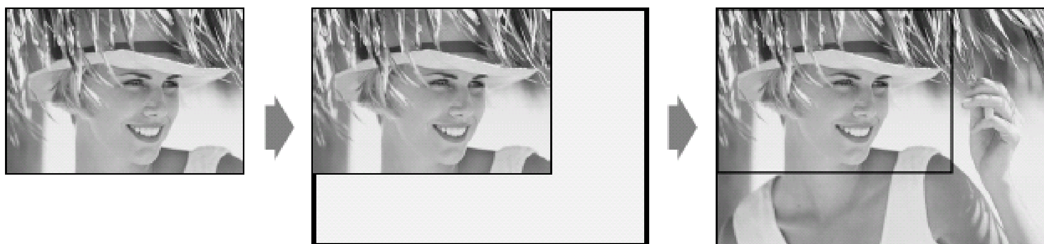


Рисунок 5.5. Изменение размеров виджета, созданного с флагом WStaticContents.

На этом, работу по созданию виджета **IconEditor** можно считать законченной. Используя знания, полученные в первых главах книги, вы без труда напишете код, который будет использовать **IconEditor** в качестве центрального виджета в **QMainWindow**, как подчиненный виджет внутри области компоновки или внутри **QScrollView**. В следующем разделе мы покажем, как интегрировать его в Qt Designer.

5.3 Интеграция виджета в Qt Designer

Прежде, чем мы сможем использовать наш виджет в Qt Designer, мы должны известить его об этом. Существует два подхода: подключение как "простого виджета" и как плагина.

Методика "простого виджета" заключается в заполнении полей диалога Qt Designer. После этого виджет может вставляться в формы, разрабатываемые в среде Qt Designer, но отображаться на форме, во время редактирования и предварительного просмотра, он будет в виде черного прямоугольника. Ниже приводится последовательность действий по интеграции **HexSpinBox** таким способом:

- Выберите пункт меню **Tools|Custom|Edit Custom Widget**. Перед вами появится диалоговое окно "**Edit Custom Widgets**".
- Щелкните по кнопке "**New Widget**".

- Измените имя класса **MyCustomWidget** на **HexSpinBox** и имя заголовочного файла **mycustomwidget.h** на **hexspinbox.h**.
- Измените "**Size Hint**" на (60, 20).
- Измените "**Size Policy**" на (Minimum, Fixed).

После этого виджет появится в секции "**Custom Widgets**" в палитре компонентов Qt Designer.

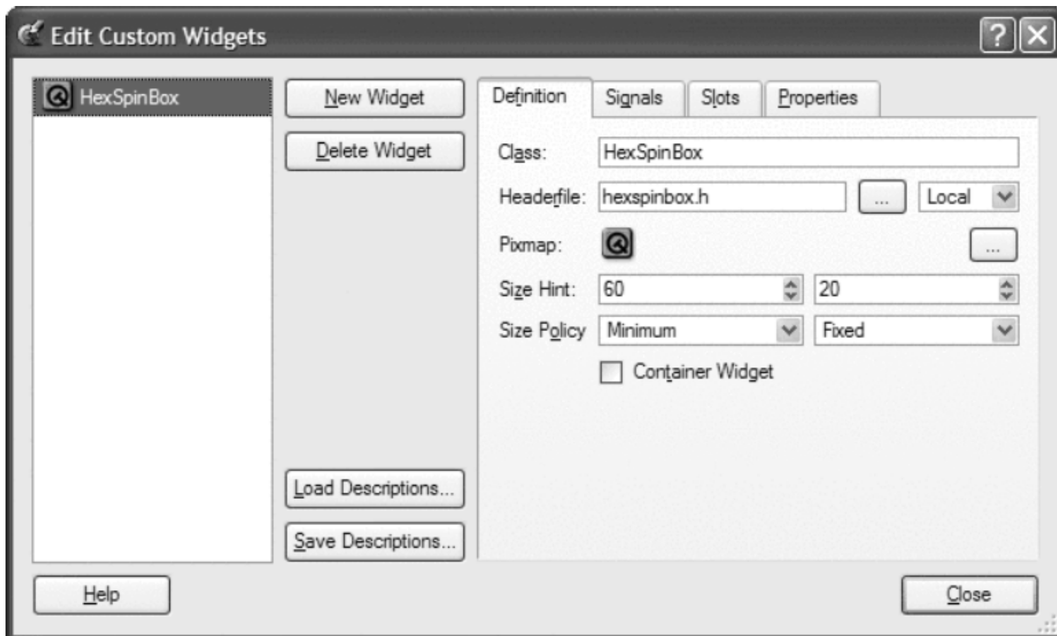


Рисунок 5.6. Диалог "Edit Custom Widgets".

Подключение виджета в виде плагина требует создания отдельной библиотеки, которую Qt Designer мог бы загружать во время своей работы и создавать с ее помощью экземпляры виджета. При таком подходе на форме, во время ее редактирования и предварительного просмотра, будет отображаться настоящий виджет. Продемонстрируем подключение виджета к Qt Designer, в виде плагина, на примере **IconEditor**.

Прежде всего, необходимо создать класс-потомок от **QWidgetPlugin** и перекрыть некоторые виртуальные функции. Весь код можно разместить в тех же самых файлах с исходными текстами, но мы создадим файлы плагина отдельно. Допустим, что файлы, с исходным кодом плагина, находятся в каталоге **iconeditorplugin**, а с исходным кодом самого компонента -- в параллельном каталоге **iconeditor**.

Заголовочный файл плагина:

```
#include <qwidgetplugin.h>
#include "../iconeditor/iconeditor.h"

class IconEditorPlugin : public QWidgetPlugin
{
public:
  QStringList keys() const;
  QWidget *create(const QString &key, QWidget *parent,
                  const char *name);
  QString includeFile(const QString &key) const;
  QString group(const QString &key) const;
  QIconSet iconSet(const QString &key) const;
  QString toolTip(const QString &key) const;
  QString whatsThis(const QString &key) const;
  bool isContainer(const QString &key) const;
};
```

Класс **IconEditorPlugin** является своего рода "фабрикой", которая изготавливает и выпускает экземпляры виджета **IconEditor**. Функции плагина используются средой Qt Designer для создания экземпляров класса и получения необходимой информации.

```
QStringList IconEditorPlugin::keys() const
{
  return QStringList() << "IconEditor";
}
```

Функция **keys()** возвращает список виджетов, "выпускаемых" плагином-фабрикой. Наш плагин "выпускает" только один виджет -- **IconEditor**.

```
QWidget *IconEditorPlugin::create(const QString &, QWidget *parent,
                                  const char *name)
{
  return new IconEditor(parent, name);
}
```

Функцию **create()** вызывает Qt Designer, когда необходимо создать экземпляр виджета. Первый аргумент -- имя класса виджета. В данном примере мы можем игнорировать его, поскольку наш плагин обслуживает только один класс. Все остальные функции так же получают имя класса в первом аргументе.

```
QString IconEditorPlugin::includeFile(const QString &) const
{
  return "iconeditor.h";
}
```

Функция **includeFile()** возвращает имя заголовочного файла виджета, который представляет плагин. Имя файла заголовка подключается к коду, создаваемому утилитой **uic**.

```
bool IconEditorPlugin::isContainer(const QString &) const
{
  return false;
}
```

Функция **isContainer()** возвращает **true**, если виджет может содержать в себе другие виджеты, иначе -- **false**. Например, **QFrame** может содержать в себе другие виджеты. В нашем случае возвращается **false**, поскольку нет смысла делать из **IconEditor** контейнер для других виджетов. Строго говоря, любой виджет может быть площадкой для размещения других виджетов, но Qt Designer отвергает такую возможность, если **isContainer**() возвращает **false**.

```
QString IconEditorPlugin::group(const QString &) const
{
  return "Plugin Widgets";
}
```

Функция **group**() возвращает имя секции палитры компонентов, в которой будет размещен виджет. Если такой секции пока нет, она будет создана автоматически.

```
QIconSet IconEditorPlugin::iconSet(const QString &) const
{
  return QIconSet(QPixmap::fromMimeSource("iconeditor.png"));
}
```

Функция **iconSet()** возвращает иконку для палитры компонентов.

```
QString IconEditorPlugin::toolTip(const QString &) const
{
  return "Icon Editor";
}
```

Функция **toolTip()** возвращает текст подсказки, которая появляется при наведении указателя мыши на иконку виджета в палитре компонентов.

```
QString IconEditorPlugin::whatsThis(const QString &) const
{
  return "Widget for creating and editing icons";
}
```

Функция **whatsThis()** возвращает текст, который появляется по запросу "What's This?" Qt Designer-а.

```
Q_EXPORT_PLUGIN(IconEditorPlugin)
```

Файл с исходным текстом плагина должен завершаться вызовом макроса **Q_EXPORT_PLUGIN()**.
Файл **.pro** для сборки плагина выглядит примерно так:

```
TEMPLATE   = lib
CONFIG    += plugin
HEADERS    = ../iconeditor/iconeditor.h
SOURCES    = iconeditorplugin.cpp \
             ../iconeditor/iconeditor.cpp
IMAGES     = images/iconeditor.png
DESTDIR    = $(QTDIR)/plugins/designer
```

Предполагается, что переменная окружения QTDIR содержит путь к каталогу, куда была установлена библиотека Qt. Когда вы собираете плагин командой **make** или **nmake**, он автоматически устанавливается в каталог **plugins** Qt Designer-а.

После сборки плагина вы можете использовать **IconEditor** в Qt Designer точно так же, как встроенные виджеты Qt.

5.4 Двойная буферизация

Двойная буферизация применяется для создания быстроменяющегося интерфейса и устранения эффекта мерцания. Мерцание возникает, когда те же самые пиксели перерисовываются несколько раз подряд, в различные цвета и в течение очень короткого отрезка времени. Если это происходит с одним пикселем, то эффект мерцания практически не заметен из-за незначительных размеров одного пикселя, но если это происходит с большой группой пикселей, эффект становится заметным и может вызывать у пользователя чувство раздражения.

Когда Qt генерирует событие **paint**, виджет сначала "стирается" -- т.е. все пиксели окрашиваются цветом фона. Затем, в функции **paintEvent()** виджету остается окрасить только те пиксели, цвет которых отличается от цвета фона. Такой двухшаговый алгоритм довольно удобен, поскольку мы перерисовываем только то, что нужно, нимало не беспокоясь о других писелях.
К сожалению, этот алгоритм является основной причиной возникновения мерцания. Например, если пользователь изменяет размеры виджета, область окна под виджетом сначала полностью очищается, а затем выполняется рисование виджета. Особенно ярко эффект мерцания проявляется, когда оконная система отображает содержимое окна при изменении его размеров, поскольку в этом случае виджеты многократно рисуются и стираются.

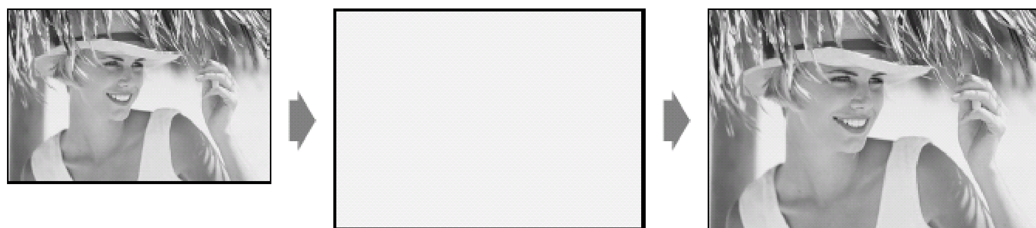


Рисунок 5.7. Порядок перерисовки виджета, при изменении размеров.

Флаг **WStaticContents**, с которым мы создавали виджет **IconEditor**, может рассматриваться как одно из возможных решений проблемы мерцания, но оно применимо только к виджетам, содержимое которых не зависит от их размера. Такие виджеты -- довольно редкое явление. В большинстве своем они стремятся растягивать свое содержимое, чтобы полностью занять отводимое им пространство. Они требуют полной перерисовки, при изменении размеров. В этом случае тоже можно устранить эффект мерцания, но решение проблемы немного сложнее.

Первое правило, на пути к устранению мерцания -- конструировать виджет с флагом **WNoAutoErase**. Этот флаг предотвращает стирание виджета перед передачей событие paint.

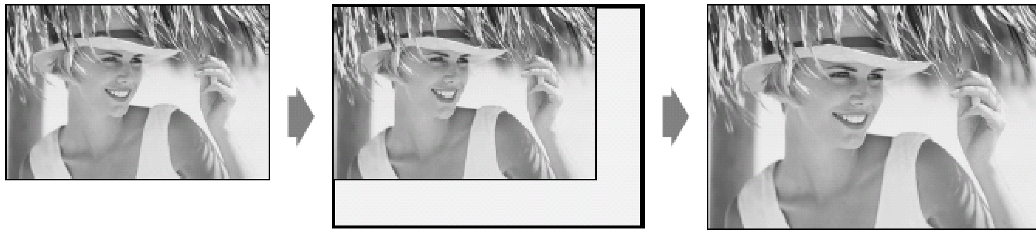


Рисунок 5.8. Порядок перерисовки виджета, созданного с флагом WNoAutoErase.

При использовании флага **WNoAutoErase** очень важно, чтобы обработчик события **paint** явно окрашивал все пиксели виджета. Любой из пикселей, которые явно не были окрашены нужным цветом, сохранит свой прежний цвет, причем этот цвет не обязательно будет цветом фона.

Правило второе -- окрашивать каждый из пикселей только один раз. Самый простой способ выполнить это требование -- рисовать виджет сначала в памяти, а затем копировать полученный рисунок. При таком подходе уже не важно -- сколько раз окрашивался тот или иной пиксель, поскольку рисование проходит не на экране. Этот прием называется двойной буферизацией. Процесс добавления двойной буферизации в виджет не очень сложен. Предположим, что оригинальный обработчик события **paint** выглядит примерно так:

```
void MyWidget::paintEvent(QPaintEvent *)
{
  QPainter painter(this);
  drawMyStuff(&painter);
}
```

Тогда версия обработчика, использующего технику двойной буферизации, могла бы выглядеть как то так:

```
void MyWidget::paintEvent(QPaintEvent *event)
{
  static QPixmap pixmap;
  QRect rect = event->rect();

  QSize newSize = rect.size().expandedTo(pixmap.size());
  pixmap.resize(newSize);
  pixmap.fill(this, rect.topLeft());

  QPainter painter(&pixmap, this);
  painter.translate(-rect.x(), -rect.y());
  drawMyStuff(&painter);
  bitBlt(this, rect.x(), rect.y(), &pixmap, 0, 0,
         rect.width(), rect.height());
}
```

Сначала устанавливаются размеры **QPixmap** такими, чтобы они были не меньше размеров прямоугольника, описывающего область перерисовки. (Чаще всего область перерисовки имеет прямоугольную или Г-образную форму, но может иметь и более сложный вид.) Экземпляр **QPixmap** объявлен статическим, чтобы избежать постоянных операций по его созданию/удалению. По тем же причинам мы никогда не уменьшаем его размер -- вызовы **QSize::expandedTo()** и **QPixmap::resize()** приводят к тому, что в течение всей своей "жизни" **QPixmap** будет только расти. Далее, **QPixmap** заполняется цветом фона виджета. Второй аргумент функции **fill()** указывает -- в какой позиции виджета будет находиться верхний левый угол **QPixmap**. (Это важно в том случае, когда виджет имеет фоновое изображение и процесс "стирания" заключается не в заполнении виджета однородным цветом, а в рисовании фонового изображения.)
Класс **QPixmap** очень напоминает **QImage** и **QWidget**. Подобно **QImage**, он хранит изображение, но глубина цвета и цветовая палитра зависят от настроек дисплея, подобно **QWidget**. Если оконная система работает с 8-ми битным цветом, все **QWidget** и **QPixmap** ограничиваются 256-ю цветами, а Qt автоматически переводит 24-х битный цвет в 8-ми битное представление.

Затем создается **QPainter**. Передавая указатель **this** конструктору, мы заставляем **QPainter** взять некоторые настройки, например шрифт, из виджета. Вызовом **translate()** осуществляется переход к системе координат виджета.
В завершение, изображение копируется в виджет с помощью глобальной функции **bitBlt()** (от англ. "bit-block transfer" -- "перемещение битового блока").

Двойная буферизация бывает полезной не только для устранения эффекта мерцания. Выгода от ее использования особенно заметна в тех случаях, когда формирование изображения -- довольно сложный процесс, и оно должно периодически обновляться на экране. Такие изображения могут сохраняться в памяти, вместе с виджетом, и копироваться в виджет при наступлении события **paint**. Это особенно полезно, когда необходимо внести в рисунок лишь незначительные изменения, например -- нарисовать границы прямоугольника выделения.
В завершение этой главы мы рассмотрим создание виджета **Plotter**. Он использует двойную буферизацию, а так же демонстрирует некоторые аспекты программирования в Qt, включая обработку событий от клавиатуры и системы координат.
Виджет **Plotter** отображает одну или несколько кривых, каждая из которых задается массивом координат. Пользователь может выделить мышью некоторую область на графике, а **Plotter** изменит масштаб отображения по осям так, что выделенная область целиком займет пространство виджета.

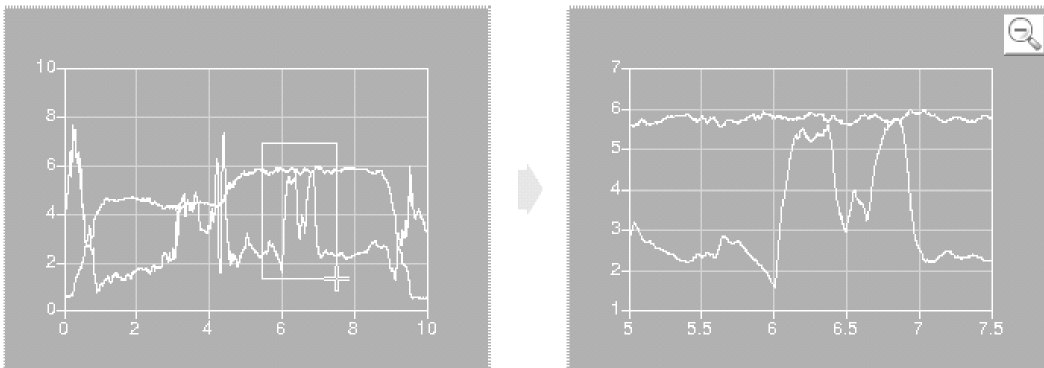


Рисунок 5.9. Изменение масштаба в компоненте Plotter.

Пользователь может неоднократно изменять масштаб таким образом. Откат на шаг назад выполняется нажатием на кнопку "**Zoom Out**", а возврат, после выполнения отката -- кнопкой "**Zoom In**". Эти кнопки видны только тогда, когда пользователь хотя бы раз изменял масштаб отображения.
Компонент может хранить данные любого числа кривых. Он так же имеет стек из экземпляров класса **PlotSettings**, на котором хранится история изменения масштаба пользователем.
Начнем с файла заголовка:

```
#ifndef PLOTTER_H
#define PLOTTER_H

#include <qpixmap.h>
#include <qwidget.h>

#include <map>
#include <vector>

class QToolButton;
class PlotSettings;

typedef std::vector<double> CurveData;
```

Мы подключили стандартные заголовки <**map**> и <**vector**>. Мы не импортировали символы из пространства имен **std** -- для заголовочных файлов это считается дурным тоном.
Мы определили **CurveData**, как синоним **std::vector<double>**. Координаты точек, определяющих кривую на графике, предполагается хранить в виде массива пар координат x и y. Например, кривая задана тремя точками, с координатами (0, 24), (1, 44), (2, 89), что соответствует массиву значений [0, 24, 1, 44, 2, 89].

```
class Plotter : public QWidget
```

```
{
  Q_OBJECT

public:
  Plotter(QWidget *parent = 0, const char *name = 0,
          WFlags flags = 0);
  void setPlotSettings(const PlotSettings &settings);
  void setCurveData(int id, const CurveData &data);
  void clearCurve(int id);
  QSize minimumSizeHint() const;
  QSize sizeHint() const;

public slots:
  void zoomIn();
  void zoomOut();
```

Компонент имеет три публичных метода для его настройки, и два публичных слота изменяющих масштаб отображения. Кроме того, перекрыты методы предка **minimumSizeHint()** и **sizeHint()**.

```
protected:
  void paintEvent(QPaintEvent *event);
  void resizeEvent(QResizeEvent *event);
  void mousePressEvent(QMouseEvent *event);
  void mouseMoveEvent(QMouseEvent *event);
  void mouseReleaseEvent(QMouseEvent *event);
  void keyPressEvent(QKeyEvent *event);
  void wheelEvent(QWheelEvent *event);
```

В защищенной секции класса объявлены функции-обработчики событий, которые мы должны реализовать.

```
private:
  void updateRubberBandRegion();
  void refreshPixmap();
  void drawGrid(QPainter *painter);
  void drawCurves(QPainter *painter);

  enum { Margin = 40 };

  QToolButton *zoomInButton;
  QToolButton *zoomOutButton;
  std::map<int, CurveData> curveMap;
  std::vector<PlotSettings> zoomStack;
  int curZoom;
  bool rubberBandIsShown;
  QRect rubberBandRect;
  QPixmap pixmap;
};
```

В приватной секции объявлены константа, несколько функций, связанных с рисованием, и несколько переменных-членов. Константа **Margin** определяет ширину пустого пространства вокруг графика. Среди переменных присутствует **QPixmap**, которая хранит копию изображения виджета, идентичного тому, что отображается на экране. График с кривыми всегда сначала рисуется в этой переменной, а затем копируется в виджет.

```
class PlotSettings
{
public:
  PlotSettings();

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

  double minX;
  double maxX;
  int numXTicks;
  double minY;
```

```
  double maxY;
  int numYTicks;

private:
  void adjustAxis(double &min, double &max, int &numTicks);
};
#endif
```

Класс **PlotSettings** определяет дипазоны изменения аргументов по осям x и y, а так же количество рисок, отображаемых на каждой из осей. На рисунке 5.10 показано соответствие между объектом **PlotSettings** и масштабом отображения виджета **Plotter**.
Строго говоря, переменные **numXTicks** и **numYTicks** хранят не число рисок, а число интервалов между рисками, т.е. если, например, в переменной **numXTicks** хранится число 5, то фактически, на оси x будет нарисовано 6 рисок. Такой подход упрощает расчеты, которые мы будем рассматривать чуть ниже.

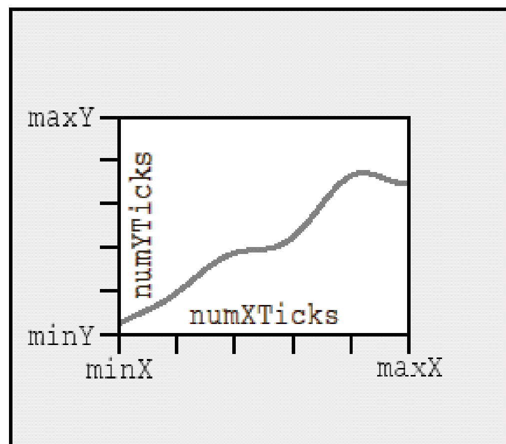


Рисунок 5.10. Переменные-члены класса PlotSettings.

Перейдем к файлу ревлизации:

```
#include <qpainter.h>
#include <qstyle.h>
#include <qtoolbutton.h>

#include <cmath>
using namespace std;

#include "plotter.h"
```

Мы подключили все необходимые заголовочные файлы и импортировали все имена из пространства имен **std**.

```
Plotter::Plotter(QWidget *parent, const char *name, WFlags flags)
    : QWidget(parent, name, flags | WNoAutoErase)
{
  setBackgroundMode(PaletteDark);
  setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);
  setFocusPolicy(StrongFocus);
  rubberBandIsShown = false;

  zoomInButton = new QToolButton(this);
  zoomInButton->setIconSet(QPixmap::fromMimeSource("zoomin.png"));
  zoomInButton->adjustSize();
  connect(zoomInButton, SIGNAL(clicked()), this, SLOT(zoomIn()));

  zoomOutButton = new QToolButton(this);
  zoomOutButton->setIconSet( QPixmap::fromMimeSource("zoomout.png"));
  zoomOutButton->adjustSize();
  connect(zoomOutButton, SIGNAL(clicked()), this, SLOT(zoomOut()));
  setPlotSettings(PlotSettings());
}
```

Третьим аргументом, конструктор **Plotter** принимает набор флагов. Этот аргумент просто передается родительскому конструктору, правда, при этом попутно включается флаг **WNoAutoErase**. Этот параметр имеет особое значение для виджетов, которые могут использоваться как автономные окна, поскольку позволяет пользователю класса сконфигурировать рамку окна и полосу заголовка.

Вызов **setBackgroundMode()** устанавливает в качестве фонового, вместо элемента палитры "background", элемент палитры -- "dark" (темный). Хотя в конструктор базового класса и передается флаг **WNoAutoErase**, тем не менее, по-прежнему необходимо иметь какой нибудь цвет в качестве фонового, которым будут закрашиваться пиксели, появляющиеся при увеличении размеров виджета, до того, как сработает обработчик **paintEvent()**. Поскольку фон виджета **Plotter** будет темным, то определенно имеет смысл окрашивать новые пиксели именно в темный цвет.

Затем, вызовом **setSizePolicy()**, устанавливается политика изменения размеров виджета. В данном случае, виджет может свободно изменять свои размеры по обеим осям. Такая политика изменения размеров характерна для виджетов, которые могут занимать значительную часть площади экрана. По-умолчанию, политика изменения размеров, для обеих осей, имеет значение **QSizePolicy::Preferred**, т.е. -- виджет "предпочитает" иметь размеры, равные "идеальным" значениям, но допускает и сжатие до минимально возможного размера (**minimumSizeHint()**), и растягивание до неопределенного предела. Вызов **setFocusPolicy()** указывает виджету, что он может принимать фокус по щелчку мыши или по клавише **Tab**. Когда **Plotter** владеет фокусом, он может принимать и обрабатывать события от клавиатуры. Он реагирует на нажатия клавиш: "+" -- увеличить изображение, "-" -- уменьшить изображение и клавиши со стрелками -- для перемещения графика вверх, вниз, влево и вправо.

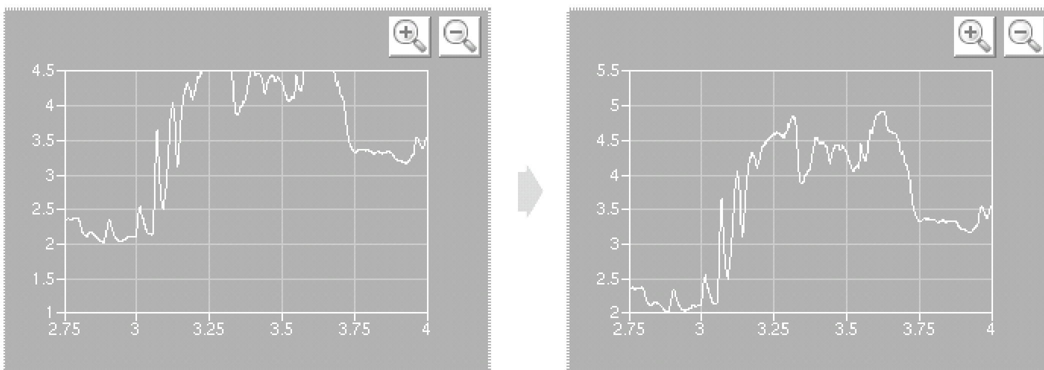


Рисунок 5.11. Перемещение графика клавишами управления курсором.

Остальной код конструктора создает две кнопки **QToolButton** с иконками. С помощью этих кнопок пользователь сможет перемещаться, взад и вперед, по стеку истории изменения масштаба. Иконки для кнопок хранятся в коллекции изображений, поэтому в файл .pro мы добавили следующие строки:

```
IMAGES += images/zoomin.png \
          images/zoomout.png
```

Вызовы методов **adjustSize()** кнопок, устанавливают размеры кнопок равные их "идеальным" размерам.
И, наконец, вызов **setPlotSettings()** завершает инициализацию виджета.

```
void Plotter::setPlotSettings(const PlotSettings &settings)
{
  zoomStack.resize(1);
  zoomStack[0] = settings;
  curZoom = 0;
  zoomInButton->hide();
  zoomOutButton->hide();
  refreshPixmap();
}
```

Функция **setPlotSettings()** используется для того, чтобы указать **PlotSettings**, который должен использоваться для отображения графика. Она вызывается из конструктора и может вызываться

пользователем класса. Каждый раз, когда пользователь изменяет масштаб отображения, создается новый экземпляр **PlotSettings** и помещается на стек истории изменения масштаба.

Стек представляют две переменные:
- **zoomStack** -- хранит экземпляры **PlotSettings** в виде одномерного массива **vector**<**PlotSettings**>.
- **curZoom** -- индекс экземпляра **PlotSettings** (в массиве **zoomStack**), который представляет текущие настройки.

После вызова **setPlotSettings()**, стек содержит только одну запись и обе кнопки, **Zoom In** и **Zoom Out**, скрыты. Они останутся невидимыми до тех пор, пока мы не вызовем их методы **show()** в слотах **zoomIn()** и **zoomOut()**. (Обычно, для того, чтобы сделать подчиненные виджеты видимыми, достаточно вызвать метод **show()** владельца, но в данном случае, мы явно вызывали **hide()** у подчиненных виджетов, поэтому они останутся скрытыми до тех пор, пока мы явно не вызовем методы **show()**.) Вызов **refreshPixmap()** обновляет изображение. Как правило, в таких случаях, мы вызываем **update()**, но в данной ситуации все делается несколько иначе, поскольку необходимо, чтобы **QPixmap** хранил самую последнюю версию картинки, отображаемой на экране. После регенерации картинки, **refreshPixmap()** вызывает **update()**, чтобы скопировать полученное изображение в виджет.

```
void Plotter::zoomOut()
{
  if (curZoom > 0) {
    --curZoom;
    zoomOutButton->setEnabled(curZoom > 0);
    zoomInButton->setEnabled(true);
    zoomInButton->show();
    refreshPixmap();
  }
}
```

Слот **zoomOut()** уменьшает изображение, если оно перед этим было увеличено. Индекс текущего элемента на стеке уменьшается, и разрешается или запрещается кнопка **Zoom Out**, в зависимости от того -- возможно ли дальнейшее перемещение к началу истории. Кнопка **Zoom In** разрешается и делается видимой. В конце, изображение обновляется вызовом **refreshPixmap()**.

```
void Plotter::zoomIn()
{
  if (curZoom < (int)zoomStack.size() - 1) {
    ++curZoom;
    zoomInButton->setEnabled( curZoom < (int)zoomStack.size() - 1);
    zoomOutButton->setEnabled(true);
    zoomOutButton->show();
    refreshPixmap();
  }
}
```

Если пользователь сначала увеличил изображение, а затем опять уменьшил, **PlotSettings** положит предыдущее значение масштаба на стек и мы сможем опять увеличить изображение нажатием на кнопку. (По прежнему остается возможность увеличить размер изображения, выделив мышью требуемый участок графика)
Слот увеличивает значение переменной **curZoom**, для перемещения на очередной уровень в стеке масштабов. Разрешает или запрещает кнопку **Zoom In** в зависимости от того -- достигнуто ли дно стека. И разрешает кнопку **Zoom Out**. Напоследок вызывается **refreshPixmap()**, чтобы обновить изображение на экране.

```
void Plotter::setCurveData(int id, const CurveData &data)
{
  curveMap[id] = data;
  refreshPixmap();
}
```

Функция **setCurveData()** заносит массив координат для заданной кривой. Если кривая с таким ID уже существует, то она заменяется новыми данными, в противном случае в график вставляется новая кривая. Координаты точек кривых хранятся в переменной **curveMap**, имеющей тип **map<int, CurveData>**.

И опять же, для обновления отображения на экране, вместо **update()**, вызывается **refreshPixmap()**.

```
void Plotter::clearCurve(int id)
{
  curveMap.erase(id);
  refreshPixmap();
}
```

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

```
QSize Plotter::minimumSizeHint() const
{
  return QSize(4 * Margin, 4 * Margin);
}
```

Функция **minimumSizeHint()** очень похожа на **sizeHint()**, с тем лишь отличием, что последняя возвращает "идеальные" размеры виджета, а **minimumSizeHint()** -- минимальные "идеальные" размеры. Менеджеры размещения никогда не будут пытаться уменьшить размеры виджета меньше этих пределов.
В данном случае функция возвращает размер 160 X 160, который учитывает размер рамки вокруг графика и некоторое пространство для рисования самого графика. Ниже этого размера график будет получаться слишком маленьким.

```
QSize Plotter::sizeHint() const
{
  return QSize(8 * Margin, 6 * Margin);
}
```

Функция **sizeHint()** возвращет "идеальные" размеры виджета, устанавливая его пропорции как 4:3.
На этом мы завершаем обзор публичных методов и слотов класса **Plotter** и переходим к защищенным обработчикам событий.

```
void Plotter::paintEvent(QPaintEvent *event)
{
  QMemArray<QRect> rects = event->region().rects();
  for (int i = 0; i < (int)rects.size(); ++i)
    bitBlt(this, rects[i].topLeft(), &pixmap, rects[i]);

  QPainter painter(this);

  if (rubberBandIsShown) {
    painter.setPen(colorGroup().light());
    painter.drawRect(rubberBandRect.normalize());
  }
  if (hasFocus()) {
    style().drawPrimitive(QStyle::PE_FocusRect, &painter,
                          rect(), colorGroup(), QStyle::Style_FocusAtBorder,
                          colorGroup().dark());
  }
}
```

Как правило, в **paintEvent()** сосредотачивается весь код, который отвечает за рисование виджета на экране. Но в нашем случае, рисование выполняет функция **refreshPixmap()**, поэтому здесь мы просто переносим буфер с рисунком в виджет.
Вызов **QRegion::rect()** возвращает массив из **QRect**, который задает область перерисовки. Для копирования каждой подобласти, из буфера с изображением в виджет, используется функция **bitBlt()**. Это функция с глобальной областью видимости. Она имеет следующий синтаксис:

```
bitBlt(dest, destPos, source, sourceRect);
```

где **source** -- это виджет-источник (в нашем случае -- буфер с картинкой), **dest** -- виджет-приемник (или **pixmap**) и **destPos** -- координаты верхнего левого угла области в приемнике, в которую будет выполняться копирование.

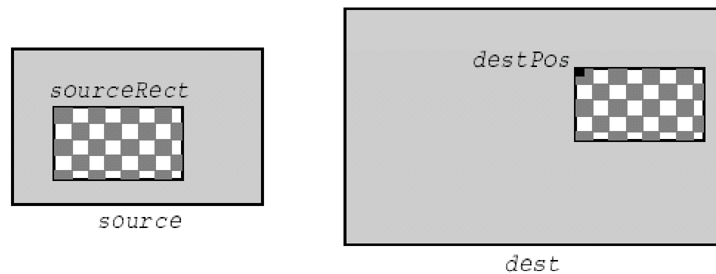


Рисунок 5.12. Копирование некоторой прямоугольной области из буфера в виджет.

В принципе, функция **bitBlt()** могла бы быть вызвана всего один раз, для отрисовки ограниченного прямоугольника. Однако, поскольку у нас **update()** вызывается в цикле из обработчика событий от мыши, для стирания и перерисовки границ области выделения, то мы получаем дополнительно еще четыре области перерисовки, в которых размещается рамка выделения (два вертикальных и два горизонтальных прямоугольника, шириной в 1 пиксель). Поэтому мы вынуждены вызывать **bitBlt()** для переноса каждой из подобластей.

Как только перенос картинки из буфера будет завершен, мы приступаем к рисованию границ области выделения. Рамка рисуется цветом группы "light", чтобы обеспечить приемлемую контрастность рамки и фона. Обратите внимание: рамка рисуется прямо на виджете, оставляя буфер с рисунком в неприкосновенности. Собственно рисование выполняется функцией **drawPrimitive()**.

Функция **QWidget::style()** возвращает стиль рисования виджета. В Qt стиль рисования виджета -- это подкласс **QStyle**. В список встроенных стилей входят **QWindowsStyle**, **QWindowsXPStyle**, **QMotifStyle** и **QMacStyle**. Каждый из них предоставляет свою реализации виртуальных методов. Функция **drawPrimitive()** -- одна из них. Она рисует графические примитивы, такие как панели, кнопки и границы областей выделения, в соответствии с выбранным стилем. Как правило, для всех виджетов приложения устанавливается единый стиль отображения (**QApplication::style()**), но он может быть изменен для каждого из виджетов, вызовом **QWidget::setStyle()**.
Создавая дочерние классы от **QStyle**, вы можете опредлять свои собственные стили отображения. Делается это обычно для того, чтобы подчеркнуть индивидуальность приложения (или группы приложений). Тем не менее, считается хорошим тоном соблюдать единый стиль отображения, выбранный пользователем при настройке рабочего окружения.

Стандартные виджеты Qt, практически всегда отрисовывают себя, основываясь на **QStyle**. Именно по этой причине они похожи на "родные" графические элементы самой операционной системы. Свои виджеты вы можете отрисовывать либо используя **QStyle**, либо собирая их из стандартных виджетов Qt. В случае с **Plotter** мы использовали оба подхода: прямоугольная рамка выделения рисуется с помощью **QStyle**, а кнопки **Zoom In** и **Zoom Out** -- это стандартные виджеты.

```
void Plotter::resizeEvent(QResizeEvent *)
{
  int x = width() - (zoomInButton->width()
          + zoomOutButton->width() + 10);
  zoomInButton->move(x, 5);
  zoomOutButton->move(x + zoomInButton->width() + 5, 5);
  refreshPixmap();
}
```

Когда необходимо изменить размеры **Plotter**, Qt генерирует событие "**resize**". Здесь мы реализуем обработку этого события. Кнопки **Zoom In** и **Zoom Out** размещаются в правом верхнем углу виджета, с небольшим (5 пикселей) промежутком между ними.
Если бы кнопки размещались в левом верхнем углу виджета, то мы могли бы просто установить их на место в конструкторе **Plotter**. Но мы выбрали правый верхний угол, поэтому необходимо постоянно следить за размерами виджета и перемещать кнопки в нужное место всякий раз, когда изменяются его размеры.

Нам не нужно изначально устанавливать кнопки в конструкторе, поскольку перед тем как виджет впервые появится на экране, Qt сгенерирует событие "**resize**".

В качестве альтернативы, можно было бы вставить в наш виджет менеджер размещения (например **QGridLayout**), который управлял бы расположением кнопок. Однако, это усложнило бы реализацию виджета и он стал бы более ресурсоемким. Когда виджет создается "с нуля", как в данном лучае, то правильнее будет отказаться от услуг менеджеров размещения и устанавливать все подчиненные компоненты вручную.
В конце обработчика, для перерисовки графика с новыми размерами, вызывается **refreshPixmap()**.

```
void Plotter::mousePressEvent(QMouseEvent *event)
{
  if (event->button() == LeftButton) {
    rubberBandIsShown = true;
    rubberBandRect.setTopLeft(event->pos());
    rubberBandRect.setBottomRight(event->pos());
    updateRubberBandRegion();
    setCursor(crossCursor);
  }
}
```

Когда пользователь нажимает левую кнопку мыши, мы начинаем показывать рамку выделяемой области. Для этого, в переменную **rubberBandIsShown**, записывается значение **true**, переменная **rubberBandRect** инициализируется текущими координатами указателя мыши, затем планируются события "**paint**", для отрисовки рамки, и наконец изменяется вид указателя мыши -- теперь он представляется в виде крестика.
Qt предоставляет два основных механизма управления внешним видом указателя мыши:

- **QWidget::setCursor()** -- устанавливает внешний вид курсора для конкретного виджета. По-умолчанию используется внешний вид курсора, установленный для владельца виджета или в виде стрелки (если в иерархии владельцев ни у кого не задан вид курсора мыши).
- **QApplication::setOverrideCursor()** -- устанавливает вид курсора для всего приложения в целом. Она отменяет действие **QWidget::setCursor()** всех виджетов, пока не будет вызвана **restoreOverrideCursor()**.

В Главе 4 мы уже пользовались функцией **QApplication::setOverrideCursor()**, с аргументом **waitCursor**, чтобы показать занятость приложения.

```
void Plotter::mouseMoveEvent(QMouseEvent *event)
{
  if (event->state() & LeftButton) {
    updateRubberBandRegion();
    rubberBandRect.setBottomRight(event->pos());
    updateRubberBandRegion();
  }
}
```

Когда пользователь перемещает указатель мыши, удерживая при этом левую кнопку в нажатом состоянии, вызывается **updateRubberBandRegion()**. Она ставит в очередь планировщика событие "**paint**", чтобы перерисовать области, где находилась рамка области выделения, затем записывает новые координаты в **rubberBandRect** и вторично выполняет перерисовку рамки выделения. В результате прежняя рамка стирается и рисуется новая, в соответствии с изменившимися координатами указателя мыши.

Переменная **rubberBandRect** имеет тип **QRect**. Экземпляры этого класса могут поставлять значения в виде (x, y, w, h), где (x, y) --это координаты левого верхнего угла, а w, h -- ширина и высота прямоугольника либо в виде пар координат верхнего левого и правого нижнего углов. В нашем случае мы используем представление в виде пар координат. В качестве координат верхнего левого угла устанавливаются координаты указателя мыши в момент нажатия на кнопку, а текущее положение курсора мыши принимается за правый нижний угол рамки выделения.
Если пользователь переместит указатель влево или вверх, то может получиться ситуация, когда то, что мы считаем правым нижним углом, окажется левее и/или выше левого верхнего угла. В этом случае **QRect** будет представлять высоту и ширину прямоугольника отрицательными числами. Чтобы избежать сложностей с отрицательными числами, в **QRect** предусмотрена функция **normalize()**, которая возвращает нормализованные координаты прямоугольника.

```
void Plotter::mouseReleaseEvent(QMouseEvent *event)
{
  if (event->button() == LeftButton) {
    rubberBandIsShown = false;
    updateRubberBandRegion();
    unsetCursor();

    QRect rect = rubberBandRect.normalize();
    if (rect.width() < 4 || rect.height() < 4)
      return;
    rect.moveBy(-Margin, -Margin);

    PlotSettings prevSettings = zoomStack[curZoom];
    PlotSettings settings;
    double dx = prevSettings.spanX() / (width() - 2 * Margin);
    double dy = prevSettings.spanY() / (height() - 2 * Margin);
    settings.minX = prevSettings.minX + dx * rect.left();
    settings.maxX = prevSettings.minX + dx * rect.right();
    settings.minY = prevSettings.maxY - dy * rect.bottom();
    settings.maxY = prevSettings.maxY - dy * rect.top();
    settings.adjust();

    zoomStack.resize(curZoom + 1);
    zoomStack.push_back(settings);
    zoomIn();
  }
}
```

Когда левая кнопка мыши отпускается, производится стирание рамки области выделения и восстанавливается прежний вид указателя мыши. Если размер выделенной области не менее, чем 4 X 4, выполняется изменение масштаба отображения графика. Если меньше -- скорее всего пользователь щелкнул по виджету по ошибке или хотел передать ему фокус. В этом случае ничего не делается.
Процесс изменения масштаба осложнен тем обстоятельством, что нам приходится работать одновременно с двумя системами координат -- системой координат виджета и системой координат самого графика. Большая часть работы заключается в переходе от одной системы координат к другой.
После выполнения преобразований, вызывается **PlotSettings::adjust()**, которая округляет размеры и находит наиболее разумные значения для рисок, наносимых на оси графика.

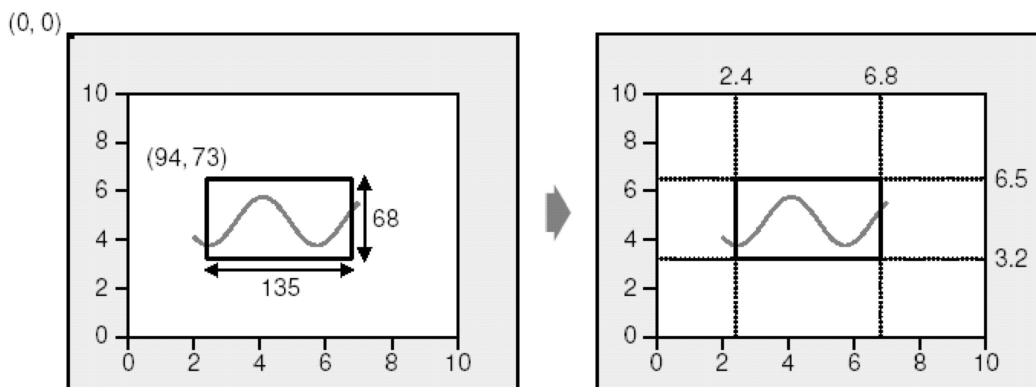


Рисунок 5.13. Преобразование координат рамки выделения из системы координат виджета, в систему координат графика.

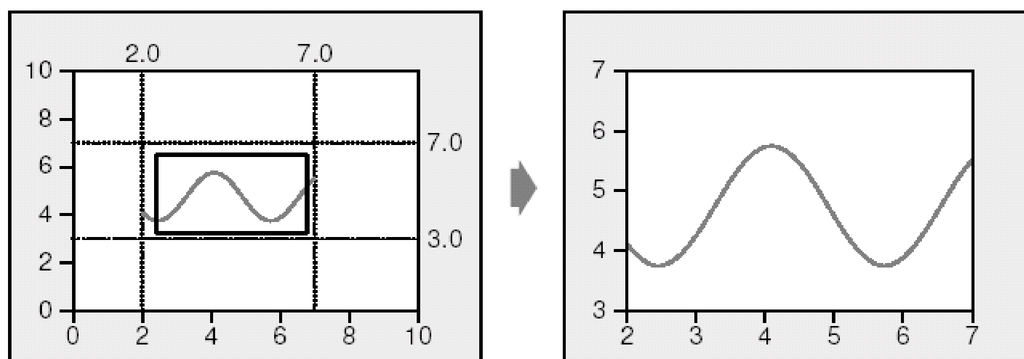


Рисунок 5.14. Округление и переход к новому масштабу отображения.

Затем выполняется масштабирование. Сначала на стек добавляется новый экземпляр **PlotSettings**, а затем вызывается **zoomIn()**.

```
void Plotter::keyPressEvent(QKeyEvent *event)
{
  switch (event->key()) {
    case Key_Plus:
        zoomIn();
        break;
    case Key_Minus:
        zoomOut();
        break;
    case Key_Left:
        zoomStack[curZoom].scroll(-1, 0);
        refreshPixmap();
        break;
    case Key_Right:
        zoomStack[curZoom].scroll(+1, 0);
        refreshPixmap();
        break;
    case Key_Down:
        zoomStack[curZoom].scroll(0, -1);
        refreshPixmap();
        break;
    case Key_Up:
        zoomStack[curZoom].scroll(0, +1);
        refreshPixmap();
        break;
    default:
        QWidget::keyPressEvent(event);
  }
}
```

Когда виджет **Plotter** владеет фокусом ввода, нажатие клавиш на клавиатуре приводит к вызову функции **keyPressEvent()**. Наша реализация обработчика обслуживает шесть клавиш: "+", "-" и клавиши управления курсором ("вверх", "вниз", "влево" и "вправо"). Если нажата клавиша, которую мы не обрабатываем, вызывается обработчик класса-предка. Для простоты мы игнорируем состояние клавиш-модификаторв: **Ctrl**, **Shift** и **Alt**. Состояние этих клавиш может быть получено через **QKeyEvent::state()**.

```
void Plotter::wheelEvent(QWheelEvent *event)
{
  int numDegrees = event->delta() / 8;
  int numTicks = numDegrees / 15;

  if (event->orientation() == Horizontal)
    zoomStack[curZoom].scroll(numTicks, 0);
  else
    zoomStack[curZoom].scroll(0, numTicks);

  refreshPixmap();
}
```

Событие "**wheel**" возникает, когда выполняется вращение колесика мыши. Чаще всего встречаются мыши, имеющие только одно колесико -- колесико вертикальной прокрутки, но есть и такие, которые имеют дополнительное колесико горизонтальной прокрутки. Qt поддерживает оба типа колесиков. Событие "**wheel**" передается виджету, если он владеет фокусом ввода. Функция **delta()** возвращает угол поворота колесика в восьмых долях градуса. В большинстве случаев, один "шаг" колесика мыши равен 15 градусам.

На этом мы завершаем обзор обработчиков событий и переходим к приватным функциям:

```
void Plotter::updateRubberBandRegion()
{
  QRect rect = rubberBandRect.normalize();

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

Функция **updateRubberBand()** вызывается из обработчиков событий **mousePressEvent()**, **mouseMoveEvent()** и **mouseReleaseEvent()**, чтобы стереть и вновь нарисовать рамку области выделения. Она содержит четыре вызова **update()**, которые ставят в очередь события "**paint**" для четырех небольших прямоугольников, в которых отображаются стороны рамки.

5.4.1 Использование логической операции NOT (НЕ), при рисовании рамки выделенной области

Один из самых распространенных способов рисование рамки области выделения состоит в использовании логической операции **NOT** (или **XOR**), которая замещает значение цвета каждого пикселя рамки на обратное. Ниже приводится альтернативная версия **updateRubberBandRegion()**, которая использует такую методику рисования:

```
void Plotter::updateRubberBandRegion()
{
   QPainter painter(this);
   painter.setRasterOp(NotROP);
   painter.drawRect(rubberBandRect.normalize());
}
```

Вызовом **setRasterOp()** задается операция наложения **NotROP**. В оригинальной версии используется значение по-умолчанию -- **CopyROP**, которая означает простое копирование нового изображения поверх имеющегося. Кода функция **updateRubberBandRegion()** вызывается вторично, для тех же самых координат, то восстанавливается начальное значение цвета пикселей, поскольку вторая логическая операция **NOT** отменяет действие первой. Преимущество такого подхода заключается в отсутствии необходимости сохранять копию закрашиваемой области. Но область его применения крайне ограничена. Например, если вместо рамки попробовать нарисовать таким образом текст, то он будет очень трудно читаться. К тому же он не всегда гарантирует высокую контрастность, например, серый цвет средней интенсивности таковым и останется. И в довершение всех бед -- на платформе Mac OS X эта возможность не поддерживается вообще. Еще один из подходов к рисованию рамок -- создание анимированных пунктирных линий. Он часто используется в программах, занимающихся обработкой изображений, поскольку дает хороший контраст не зависимо от начального цвета пикселей, по которым проходит рамка. Для того, чтобы создать анимированную рамку в Qt, вам придется перекрыть обработчик события **QObject::timerEvent()**, в котором надо будет стирать рамку и опять рисовать ее, при этом всякий раз начинать рисование точек пунктира с новой позиции, что создаст иллюзию движения точек по линии.

```
void Plotter::refreshPixmap()
{
  pixmap.resize(size());
  pixmap.fill(this, 0, 0);
  QPainter painter(&pixmap, this);

  drawGrid(&painter);
```

```
  drawCurves(&painter);
  update();
}
```

Функция **refreshPixmap()** перерисовывает кривые графиков в буфере и затем обновляет изображение на экране. Сначала устанавливается размер буфера, чтобы он соответствовал размерам виджета. Затем он заполняется цветом фона, который был установлен в конструкторе, вызовом **setBackgroundMode()**.

Далее создается **QPainter** и с его помощью в буфере рисуются координатная сетка и кривые. В заключение вызывается **update()**, которая планирует событие "**paint**" для всего виджета в целом. Буфер будет скопирован в виджет -- в обработчике события **paintEvent()**.

```
void Plotter::drawGrid(QPainter *painter)
{
  QRect rect(Margin, Margin,
             width() - 2 * Margin, height() - 2 * Margin);
  PlotSettings settings = zoomStack[curZoom];
  QPen quiteDark = colorGroup().dark().light();
  QPen light = colorGroup().light();

  for (int i = 0; i <= settings.numXTicks; ++i) {
    int x = rect.left() + (i * (rect.width() - 1)
                             / settings.numXTicks);
    double label = settings.minX + (i * settings.spanX()
                                      / settings.numXTicks);
    painter->setPen(quiteDark);
    painter->drawLine(x, rect.top(), x, rect.bottom());
    painter->setPen(light);
    painter->drawLine(x, rect.bottom(), x, rect.bottom() + 5);
    painter->drawText(x - 50, rect.bottom() + 5, 100, 15,
                      AlignHCenter | AlignTop,
                      QString::number(label));
  }

  for (int j = 0; j <= settings.numYTicks; ++j) {
    int y = rect.bottom() - (j * (rect.height() - 1)
                               / settings.numYTicks);
    double label = settings.minY + (j * settings.spanY()
                                      / settings.numYTicks);
    painter->setPen(quiteDark);
    painter->drawLine(rect.left(), y, rect.right(), y);
    painter->setPen(light);
    painter->drawLine(rect.left() - 5, y, rect.left(), y);
    painter->drawText(rect.left() - Margin, y - 10,
                      Margin - 5, 20,
                      AlignRight | AlignVCenter,
                      QString::number(label));
  }
  painter->drawRect(rect);
}
```

Функция **drawGrid()** рисует координатную сетку, на фоне которой будут отображаться графики. Первый цикл for рисует вертикальные линии сетки и риски на оси OX. Второй -- горизонтальные линии сетки и риски на оси OY. Для рисования числовых значений, напротив рисок, и обозначений осей -- вызывается функция **drawText()**.
Функция **drawText()** имеет следующий синтаксис:

```
painter.drawText(x, y, w, h, alignment, text);
```

где (x, y, w, h) задают область рисования, **alignment** -- выравнивание текста внутри этой области, **text** -- собственно текст.

```
void Plotter::drawCurves(QPainter *painter)
{
  static const QColor colorForIds[6] = {
    red, green, blue, cyan, magenta, yellow
```

```
  };
  PlotSettings settings = zoomStack[curZoom];
  QRect rect(Margin, Margin,
             width() - 2 * Margin, height() - 2 * Margin);
  painter->setClipRect(rect.x() + 1, rect.y() + 1,
                       rect.width() - 2, rect.height() - 2);
  map<int, CurveData>::const_iterator it = curveMap.begin();
  while (it != curveMap.end()) {
    int id = (*it).first;
    const CurveData &data = (*it).second;
    int numPoints = 0;
    int maxPoints = data.size() / 2;
    QPointArray points(maxPoints);

    for (int i = 0; i < maxPoints; ++i) {
      double dx = data[2 * i] - settings.minX;
      double dy = data[2 * i + 1] - settings.minY;
      double x = rect.left() + (dx * (rect.width() - 1)
                                / settings.spanX());
      double y = rect.bottom() - (dy * (rect.height() - 1)
                                  / settings.spanY());
      if (fabs(x) < 32768 && fabs(y) < 32768) {
        points[numPoints] = QPoint((int)x, (int)y);
        ++numPoints;
      }
    }
    points.truncate(numPoints);
    painter->setPen(colorForIds[(uint)id % 6]);
    painter->drawPolyline(points);
    ++it;
  }
}
```

Функция **drawCurves()** рисует кривые графиков поверх координатной сетки. Начинается она с ограничения области рисования, вызовом **setClipRect()**. **QPainter** будет игнорировать попытки рисования за ее пределами.
Затем выполняется проход по всем кривым графика и для каждой из них -- по парам координат (x, y). Элемент итератора **first** дает нам ID (идентификатор) кривой, а **second** -- массив координат точек кривой.
Вложенный цикл **for** выполняет преобразование из системы координат графика в систему координат виджета и сохраняет результат в массив **points**, при условии, что они находятся в разумных пределах. По окончании выполнения преобразований координат для всех точек кривой, устанавливается цвет "чернил" (используя один из предопределенных цветов) и вызывается **drawPolyline()**, которая рисует ломаную линию, проходящую через заданные точки.
На этом завершается реализация класса **Plotter**. И нам остается рассмотреть еще ряд функций-членов класса **PlotSettings**.

```
PlotSettings::PlotSettings()
{
  minX = 0.0;
  maxX = 10.0;
  numXTicks = 5;
  minY = 0.0;
  maxY = 10.0;
  numYTicks = 5;
}
```

Конструктор инициализирует оси координат, с диапазоном измерения от 0 до 10 по каждой из них, и задает количество рисок на каждой из осей, равное 5.

```
void PlotSettings::scroll(int dx, int dy)
{
  double stepX = spanX() / numXTicks;
  minX += dx * stepX;
  maxX += dx * stepX;
  double stepY = spanY() / numYTicks;
  minY += dy * stepY;
  maxY += dy * stepY;
```

```
}
```

Функция **scroll()** увеличивает (или уменьшает) значения переменных **minX**, **maxX**, **minY** и **maxY**. Она реализует поддержку скроллинга и вызывается из **Plotter::keyPressEvent()**.

```
void PlotSettings::adjust()
{
  adjustAxis(minX, maxX, numXTicks);
  adjustAxis(minY, maxY, numYTicks);
}
```

Функция **adjust()** вызывается из **mouseReleaseEvent()**. Она округляет значения переменных **minX**, **maxX**, **minY** и **maxY** до "наилучших" и определяет значения рисок по каждой из осей. Обработка конкретной оси координат выполняется функцией **adjustAxis()**.

```
void PlotSettings::adjustAxis(double &min, double &max, int &numTicks)
{
  const int MinTicks = 4;
  double grossStep = (max - min) / MinTicks;
  double step = pow(10, floor(log10(grossStep)));

  if (5 * step < grossStep)
    step *= 5;
  else if (2 * step < grossStep)
    step *= 2;

  numTicks = (int)(ceil(max / step) - floor(min / step));
  min = floor(min / step) * step;
  max = ceil(max / step) * step;
}
```

Она округляет аргументы **min** и **max** до "наилучших" значений и определяет число рисок (**numTicks**) на оси, исходя из диапазона [min.. max]. Функция должна изменять фактические параметры (**minX**, **maxX**, **numXTicks**, и т.д.), поэтому они передаются по ссылке, а не по значению.

Большая часть кода функции служит для определения наиболее подходящего "расстояния" между соседними рисками ("шаг"). К выбору шага нужно подходить очень осторожно. Дробные значения шага, например 3.8, сложнее воспринимаются людьми, чем круглые. Для осей, которые имеют метки, записываемые в десятичной нотации, "наилучшими" значениями будут числа 10^n, 2*10^n или 5*10^n.

Поиск начинается с "большого шага", своего рода максимального значения для шага. Затем находится число, ближайшее (меньше или равно) к значению "большого шага", которое можно записать в форме 10^n: берется десятичный логарифм от "большого шага", округляется вниз до ближайшего целого и затем вычисляется степень 10-ти, с найденым числом в качестве показателя. Например, пусть "большой шаг" равен числу 236, в результате получаем: log 236 = 2.37291; округление дает число 2, а 10^2 = 100 -- кандидат для размера "наилучшего" шага.

Как только мы получили значение первого "кандидата" для шага оси, необходимо рассчитать еще два значения -- 2*10^n и 5*10^n. Для примера выше, два других кандидата -- это числа 200 и 500. Но число 500 значительно больше установленного нами максимума (236), а 200 -- меньше, поэтому в качестве шага оси принимается число 200.
Теперь, основываясь на значении шага, очень легко вычислить **min**, **max** и **numTicks**. Значение **min** получается за счет округления вниз начального **min**, до ближайшего множителя шага, а значение max -- за счет округления вверх, до ближайшего множителя шага. Величина **numTicks** -- это количество шагов, укладывающихся в интервал, между **min** и **max**. Например, если начальные значения **min** = 240, **max** = 1184, то новый диапазон значений оси будет составлять [200..1200], с 5 интервалами-шагами.
Этот алгоритм не всегда дает оптимальные значения. Более изощренный алгоритм вы найдете в статье Пауля Хекберта (Paul S. Heckbert) -- "Nice Numbers for Graph Labels", опубликованной в Graphics Gems (ISBN 0-12-286166-3). Кроме того, в ежеквартальнике Qt Quarterly имеется статья "Fast and Flicker-Free" (http://doc.trolltech.com/qq/qq06-flicker-free.html), которая рассматривает некоторые идеи по устранению эффекта мерцания.

Эта глава завершает первую часть книги. Здесь мы рассказали как настроить стандартные виджеты Qt и как создать свой виджет, используя в качестве базового класса **QWidget**. В Главе 2 мы видели, как можно "собрать" виджет из других виджетов, эта тема будет рассматриваться глубже в Главе 6.
К настоящему моменту, вы получили достаточно знаний, чтобы написать законченное приложение с графическим интерфейсом пользователя. Во второй части книги, мы перейдем к более глубокому изучению Qt, что позволит нам использовать всю мощь этой замечательной библиотеки.

ЧАСТЬ 2 - УГЛУБЛЕННЫЕ СВЕДЕНИЯ

6 Управление размещением виджетов

Каждый из виджетов, помещаемый на форму, должен быть размещен в нужном месте и с соответствующими размерами. Виджеты, размеры которых превышают размер формы, могут снабжаться полосами прокрутки, чтобы пользователь мог просмотреть все его содержимое. В этой главе мы рассмотрим различные способы размещения виджетов на форме и покажем, как реализовать отстыковываемые (**dockable**) окна и многодокументный интерфейс (MDI).

6.1 Основы компоновки виджетов

Qt предоставляет три основных способа управления размещением подчиненных виджетов на форме: абсолютное позиционирование, ручное управление размещением и менеджеры компоновки. Мы рассмотрим каждый из них, на примере диалога "**Find File**", показанный на рисунке 6.1.

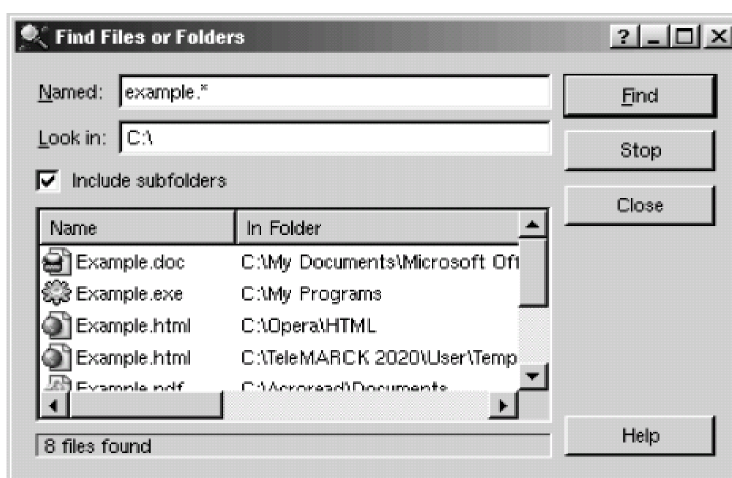


Рисунок 6.1. Диалог "Find File".

Абсолютное позиционирование -- это самый "неблагодарный" способ размещения виджетов. При таком подходе положение и размеры виджетов жестко зашиваются в программу, что, как правило, влечет за собой фиксированные размеры самой формы. Взглянем на конструктор диалога **FindFileDialog**, который строится по принципу абсолютного позиционирования:

```
FindFileDialog::FindFileDialog(QWidget *parent, const char *name)
    : QDialog(parent, name)
{
  ...
  namedLabel->setGeometry(10, 10, 50, 20);
  namedLineEdit->setGeometry(70, 10, 200, 20);
  lookInLabel->setGeometry(10, 35, 50, 20);
  lookInLineEdit->setGeometry(70, 35, 200, 20);
  subfoldersCheckBox->setGeometry(10, 60, 260, 20);
  listView->setGeometry(10, 85, 260, 100);
  messageLabel->setGeometry(10, 190, 260, 20);
  findButton->setGeometry(275, 10, 80, 25);
  stopButton->setGeometry(275, 40, 80, 25);
  closeButton->setGeometry(275, 70, 80, 25);
  helpButton->setGeometry(275, 185, 80, 25);
  setFixedSize(365, 220);
}
```

Абсолютное позиционирование имеет массу недостатков. Самый главный недостаток -- невозможность изменить размеры окна. Другой недостаток: текст меток может не умещаться в заданные размеры, если пользователь выбрал большой размер шрифта или, если интерфейс приложения был переведен на другой язык. Кроме того, этот подход требует от нас выполнения кропотливой работы по вычислению положения и размеров виджетов.

При ручном управлении размещением виджетов, мы по прежнему должны задавать положение компонентов на форме, но их размеры устанавливаются пропорционально размерам окна. Добиться этого можно за счет перекрытия обработчика события **resizeEvent()** формы, в котором можно пересчитывать и задавать новые размеры подчиненных виджетов:

```
FindFileDialog::FindFileDialog(QWidget *parent, const char *name)
    : QDialog(parent, name)
{
  ...
  setMinimumSize(215, 170);
  resize(365, 220);
}

void FindFileDialog::resizeEvent(QResizeEvent *)
{
  int extraWidth = width() - minimumWidth();
  int extraHeight = height() - minimumHeight();

  namedLabel->setGeometry(10, 10, 50, 20);
  namedLineEdit->setGeometry(70, 10, 50 + extraWidth, 20);
  lookInLabel->setGeometry(10, 35, 50, 20);
  lookInLineEdit->setGeometry(70, 35, 50 + extraWidth, 20);
  subfoldersCheckBox->setGeometry(10, 60, 110 + extraWidth, 20);
  listView->setGeometry(10, 85,
                        110 + extraWidth, 50 + extraHeight);
  messageLabel->setGeometry(10, 140 + extraHeight,
                            110 + extraWidth, 20);
  findButton->setGeometry(125 + extraWidth, 10, 80, 25);
  stopButton->setGeometry(125 + extraWidth, 40, 80, 25);
  closeButton->setGeometry(125 + extraWidth, 70, 80, 25);
  helpButton->setGeometry(125 + extraWidth, 135 + extraHeight,
                          80, 25);
}
```

В конструкторе мы установили минимальные размеры формы 215 X 170 и начальный размер 365 X 220. В обработчике **resizeEvent()** устанавливаются новые размеры виджетов при изменении размеров окна.

Как и в случае с абсолютным позиционированием, ручное управление размещением требует от программиста предварительного вычисления некоторых констант, которые потом жестко зашиваются в код программы. Написание таких программ очень утомительное занятие, особенно если потом потребуется внести изменения в дизайн формы. По прежнему сохраняется риск того, что какие-то надписи на форме не поместятся в отведенное им пространство. Избежать этого можно, если учитывать "идеальные" размеры виджетов, но это еще больше усложнит код.

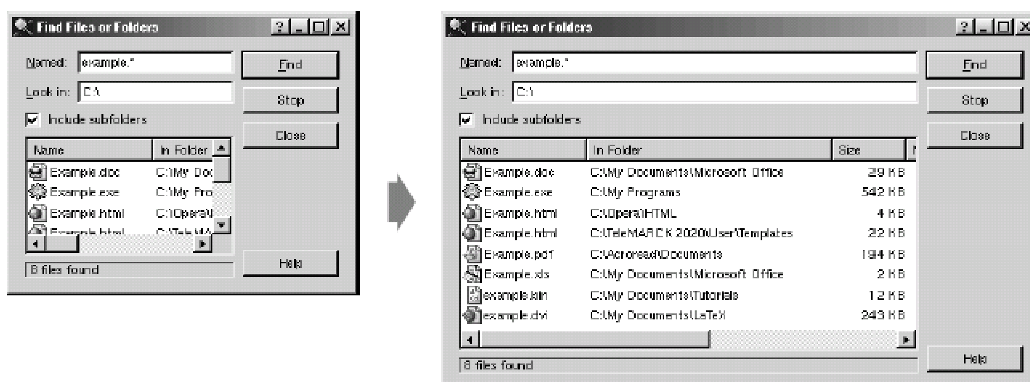


Рисунок 6.2. Диалог "Find File" с изменяемыми размерами.

Наилучшим решением размещения виджетов на форме считается использование менеджеров компоновки Qt. Они обеспечивают разумные размеры по-умолчанию для каждого типа виджетов и учитывают "идеальные" размеры каждого из них, которые, в свою очередь, зависят от выбранного размера шрифта, стиля отображения и объема содержимого. Кроме того, менеджеры компоновки учитываю минимальные и максимальные размеры, и автоматически корректируют расположение

виджетов, в ответ на изменение шрифта, содержимого или размеров окна.

В Qt имеется три вида менеджеров компоновки: **QHBoxLayout**, **QVBoxLayout** и **QGridLayout**. Это классы-потомки от **QLayout**, который реализует основные методы управления размещением. Все три класса полностью поддерживаются Qt Designer-ом, а так же могут использоваться при написании кода вручную. Оба варианта использования были рассмотрены в Главе 2.
Ниже приводится конструктор **FindFileDialog**, в котором используются менеджеры размещения:

```
FindFileDialog::FindFileDialog(QWidget *parent, const char *name)
    : QDialog(parent, name)
{
  ...
  QGridLayout *leftLayout = new QGridLayout;
  leftLayout->addWidget(namedLabel, 0, 0);
  leftLayout->addWidget(namedLineEdit, 0, 1);
  leftLayout->addWidget(lookInLabel, 1, 0);
  leftLayout->addWidget(lookInLineEdit, 1, 1);
  leftLayout->addMultiCellWidget(subfoldersCheckBox, 2, 2, 0, 1);
  leftLayout->addMultiCellWidget(listView, 3, 3, 0, 1);
  leftLayout->addMultiCellWidget(messageLabel, 4, 4, 0, 1);

  QVBoxLayout *rightLayout = new QVBoxLayout;
  rightLayout->addWidget(findButton);
  rightLayout->addWidget(stopButton);
  rightLayout->addWidget(closeButton);
  rightLayout->addStretch(1);
  rightLayout->addWidget(helpButton);

  QHBoxLayout *mainLayout = new QHBoxLayout(this);
  mainLayout->setMargin(11);
  mainLayout->setSpacing(6);
  mainLayout->addLayout(leftLayout);
  mainLayout->addLayout(rightLayout);
}
```

Размещением компонентов на форме управляют один **QHBoxLayout**, один **QGridLayout** и один **QVBoxLayout**. **QGridLayout** и **QVBoxLayout** расположены рядом друг с дружкой, внутри **QHBoxLayout**. Рамка вокруг формы имеет ширину 11 пикселей, промежутки между подчиненными виджетами -- 6 пикселей.

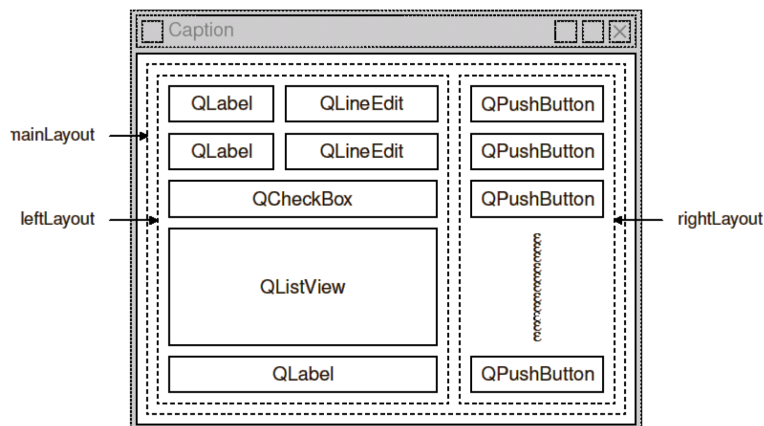


Рисунок 6.3. Раскладка диалога "Find File".

QGridLayout работает как плоская сетка ячеек. **QLabel**, в верхнем левом углу области, занимает ячейку (0, 0), а соответствующий ей **QLineEdit** -- (0, 1). **QCheckBox** объединяет две колонки и занимает ячейки (2, 0) и (2, 1). **QListView** и **QLabel**, расположенные снизу, так же занимают по две ячейки. Вызов **addMultiCellWidget()** имеет следующий синтаксис:

```
leftLayout->addMultiCellWidget(widget, row1, row2, col1, col2);
```

где **widget** -- это подчиненный виджет, передаваемый этому менеджеру компоновки, **row1**, **col1** -- верхняя левая ячейка, которую занимает виджет и **row2**, **col2** -- правая нижняя ячейка.

Тот же самый диалог может быть создан с помощью визуального построителя Qt Designer. Пример работы с визуальным построителем, мы рассматривали в Главе 2.

Использование менеджеров размещения дает определенные преимущества, которые мы уже обсуждали ранее. Если в область компоновки добавляется виджет или удаляется из нее, менеджер автоматически адаптируется под изменившиеся условия. То же самое применимо и к случаю, когда вызываются методы подчиненного компонента -- **hide()** и **show()**. Если подчиненный виджет изменит "идеальный" размер, то раскладка изменится, с учетом изменившихся обстоятельств. Кроме того, менеджеры размещения автоматически установят минимальный размер формы в целом, основываясь на минимальных и "идеальных" размерах дочерних виджетов.

Во всех примерах, которые мы до сих пор рассматривали, мы просто объединяли виджеты менеджерами размещения и добавляли дополнительные распорки, для утилизации свободного пространства. Но иногда, чтобы расположение компонентов полностью соответствовало нашим желаниям, этого бывает недостаточно. В таких ситуациях необходимо дополнительно настраивать политики изменения размеров и "идеальные" размеры виджетов.

Политика изменения размеров сообщает менеджеру компоновки, как виджет должен растягиваться или сжиматься. Qt по-умолчанию дает неплохие значения политики изменения размеров для всех стандартных виджетов, но никакое значение по-умолчанию не может идеально подходить под все случаи жизни. Поэтому, до сих пор обычной практикой считается дополнительная настройка политик изменения размеров для одного-двух виджетов на форме. Политика изменения размеров назначается для каждого из двух направлений (по вертикали и по горизонтали). Наиболее часто используются значения **Fixed**, **Minimum**, **Maximum**, **Preferred** и **Expanding**:

- **Fixed** -- виджет имеет фиксированные размеры, т.е. он не может ни растягиваться, ни сжиматься. Он всегда должен иметь "идеальный" (sizeHint()) размер.
- **Minimum** -- "идеальный" размер виджета, это минимально возможный его размер. Виджет не может сжиматься меньше этого размера, но может растягиваться и занимать все доступное пространство, если это потребуется.
- **Maximum** -- "идеальный" размер виджета, это максимально возможный его размер, т.е. виджет может сжиматься до минимально возможного размера, но не может растягиваться больше "идеального".
- **Preferred** -- "идеальный" размер виджета, это предпочтительный его размер, но в случае необходимости виджет может как растягиваться, так и сжиматься.
- **Expanding** -- виджет может и растягиваться, и сжиматься, но он предпочитает растягиваться.

Рисунок 6.4 подытоживает все, что было сказано выше о политиках изменения размеров, на примере **QLabel**, отображающей текст "**Some Text**".

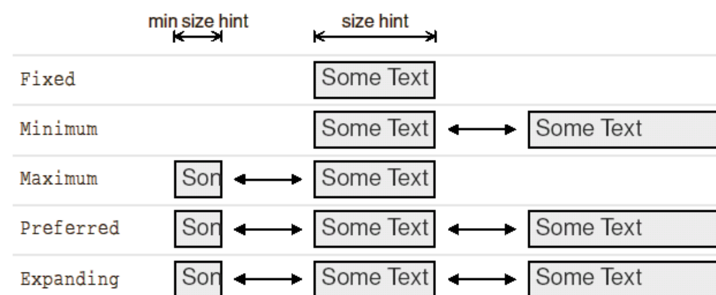


Рисунок 6.4. Различные политики изменения размеров.

Когда изменяется размер формы, которая включает в себя компоненты с политиками **Preferred** и **Expanding**, то дополнительное пространство отдается виджету с политикой **Expanding**, в то время, как виджет **Preferred** сохраняет "идеальные" размеры.

Существует еще две политики изменения размеров: **MinimumExpanding** и **Ignored**. Первая из них использовалась в ранних версиях Qt, хотя и довольно редко, в настоящее время не играет большой роли, поскольку лучший результат дает назначение политики **Expanding** и повторная реализация (перекрытие) метода **minimumSizeHint()**. Вторая -- во многом похожа на **Expanding**, но при этом игнорирует "идеальные" размеры виджета.

В дополнение к политикам изменения размера, горизонтальная и вертикальная составляющие визуального компонента, **QSizePolicy** хранят факторы растяжения. Они используются для задания степени растяжимости. Например, предположим, что на форме находятся **QListView**, а под ним -- **QTextEdit**. Нам необходимо, чтобы при растягивании формы **QTextEdit** рос в два раза быстрее, чем **QListView**. Для этого, фактор растягивания по вертикали (**verticalStretch**) компонента **QTextEdit** устанавливаем равным 2, а **QListView** -- 1.

Еще один способ воздействовать на порядок расположения -- изменять минимальный и максимальный размеры подчиненных виджетов. Менеджер компоновки будет учитывать значения этих параметров.

6.2 Разделители

Разделитель (**splitter**) -- это виджет, который используется для размещения других виджетов и их разделения вертикальной или горизонтальной полосой. Пользователь может изменять размеры виджетов, перемещая разделитель. Они зачастую используются вместо менеджеров размещения, чтобы дать пользователю возможность самому управлять размерами виджетов.
Разделители в Qt реализованы в виде класса **QSplitter**. Подчиненные виджеты автоматически размещаются друг за дружкой, в порядке их создания, в смежных областях, разделителя. Ниже приводится код, который создает окно, изображенное на рисунке 6.5.

```
#include <qapplication.h>
#include <qsplitter.h>
#include <qtextedit.h>

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

  QSplitter splitter(Qt::Horizontal);
  splitter.setCaption(QObject::tr("Splitter"));
  app.setMainWidget(&splitter);

  QTextEdit *firstEditor = new QTextEdit(&splitter);
  QTextEdit *secondEditor = new QTextEdit(&splitter);
  QTextEdit *thirdEditor = new QTextEdit(&splitter);

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

На форме находятся три компонента **QTextEdit**, выровненных по горизонтали виджетом **QSplitter**. В отличие от менеджера размещения, который отвечает только за размещение подчиненных виджетов, **QSplitter** является потомком класса **QWidget** и может использоваться как любой другой виджет.

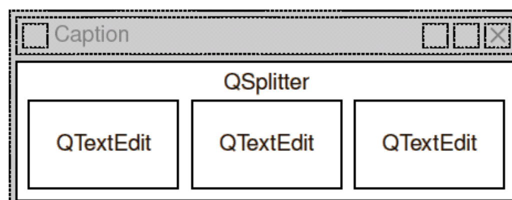


Рисунок 6.5. Разделитель в приложении.

QSplitter может размещать подчиненные виджеты как по горизонтали, так и по вертикали. За счет вкладывания одного разделителя в другой, могут быть достигнуты весьма замысловатые комбинации. Например, приложение - почтовый клиент, главное окно которого изображено на рисунке 6.6, содержит горизонтальный разделитель и, вложенный в него, вертикальный разделитель.
Ниже приводится код конструктора подкласса **QMainWindow**:

```
MailClient::MailClient(QWidget *parent, const char *name)
    : QMainWindow(parent, name)
{
  horizontalSplitter = new QSplitter(Horizontal, this);
  setCentralWidget(horizontalSplitter);

  foldersListView = new QListView(horizontalSplitter);
  foldersListView->addColumn(tr("Folders"));
  foldersListView->setResizeMode(QListView::AllColumns);

  verticalSplitter = new QSplitter(Vertical, horizontalSplitter);
  messagesListView = new QListView(verticalSplitter);
  messagesListView->addColumn(tr("Subject"));
  messagesListView->addColumn(tr("Sender"));
  messagesListView->addColumn(tr("Date"));
  messagesListView->setAllColumnsShowFocus(true);
  messagesListView->setShowSortIndicator(true);
  messagesListView->setResizeMode(QListView::AllColumns);

  textEdit = new QTextEdit(verticalSplitter);
  textEdit->setReadOnly(true);

  horizontalSplitter->setResizeMode(foldersListView,
                                    QSplitter::KeepSize);
  verticalSplitter->setResizeMode(messagesListView,
                                  QSplitter::KeepSize);
  ...
  readSettings();
}
```

Здесь сначала создается горизонтальный разделитель, после чего он назначается центральным виджетом. Затем создаются подчиненные виджеты.

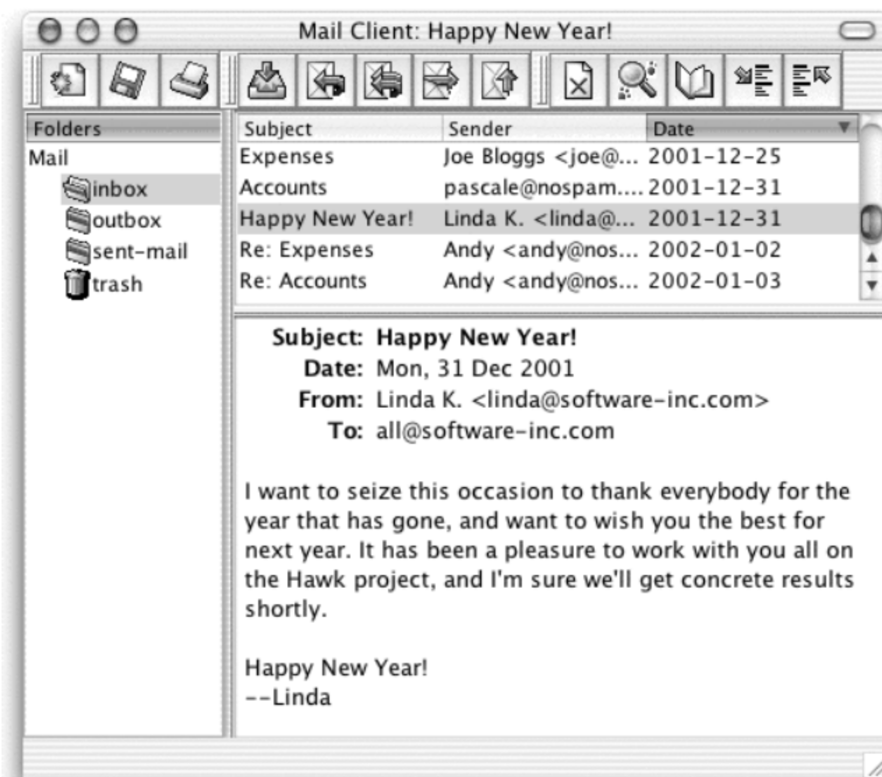


Рисунок 6.6. Почтовый клиент в Mac OS X.

Когда пользователь изменит размеры окна, **QSplitter** распределит пространство между подчиненными виджетами таким образом, что относительные их размеры останутся без изменения. Но в случае с почтовым клиентом нам необходимо, чтобы два **QListView** сохранили свои размеры, а все дополнительное пространство было отдано компоненту **QTextEdit**. Достигается это парой вызовов **setResizeMode()**.
На запуске приложения, **QSplitter** устанавливает размеры подчиненных виджетов, основываясь на их начальных размерах. Передвинуть разделитель можно не только вручную, но и программно, вызвав **QSplitter::setSizes()**. Кроме того, **QSplitter** предоставляет возможность сохранить свое положение, при завершении работы приложения, и восстановить его на следующем запуске. Ниже приводится функция, которая сохраняет настройки приложения - почтового клиента:

```
void MailClient::writeSettings()
{
  QSettings settings;
  settings.setPath("software-inc.com", "MailClient");
  settings.beginGroup("/MailClient");

  QString str;
  QTextOStream out1(&str);

  out1 << *horizontalSplitter;
  settings.writeEntry("/horizontalSplitter", str);
  QTextOStream out2(&str);
  out2 << *verticalSplitter;
  settings.writeEntry("/verticalSplitter", str);

  settings.endGroup();
}
```

И, соответствующая ей, функция **readSettings()**.

```
void MailClient::readSettings()
{
  QSettings settings;
  settings.setPath("software-inc.com", "MailClient");
  settings.beginGroup("/MailClient");

  QString str1 = settings.readEntry("/horizontalSplitter");
  QTextIStream in1(&str1);
  in1 >> *horizontalSplitter;
  QString str2 = settings.readEntry("/verticalSplitter");
  QTextIStream in2(&str2);
  in2 >> *verticalSplitter;

  settings.endGroup();
}
```

Вся файловые операции, в этих функциях, выполняются через классы **QTextIStream** и **QTextOStream** -- потомки класса **QTextStream**.

По-умолчанию, во время перетаскивания, разделитель отображается в виде рамки. А размеры виджетов, с обеих сторон разделителя, изменяют размер только тогда, когда пользователь отпустит кнопку мыши. Чтобы изменения размеров происходили в реальном времени, необходимо вызвать **setOpaqueResize(true)**.
Разделители **QSplitter** полностью поддерживаются визуальным построителем Qt Designer. Чтобы поместить виджеты в разделитель -- разместите подчиненные виджеты на форме примерно так, как вы желаете, затем выделите их и выберите пункт меню **Layout|Lay Out Horizontally** (**in Splitter**) или **Layout|Lay Out Vertically** (**in Splitter**).

6.3 Многостраничные виджеты

Еще один виджет, которые может оказаться полезным, в смысле компоновки -- это **QWidgetStack**. Он может содержать наборы виджетов, объединяемых в "страницы", и всегда показывает только одну

страницу, скрывая остальные. Нумерация страниц начинается с 0. Чтобы сделать определенный подчиненный виджет-страницу видимым, необходимо вызвать функцию **raiseWidget()**, передав ей либо номер страницы, либо указатель на подчиненный виджет.

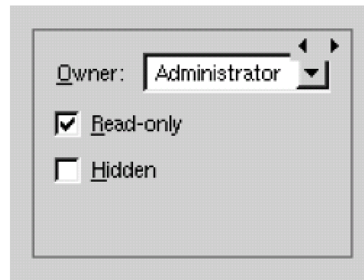


Рисунок 6.7. QWidgetStack.

Сам по себе **QWidgetStack** невидим и не предоставляет никаких дополнительных визуальных элементов, с помощью которых пользователь мог бы переходить от страницы к странице. Маленькие стрелочки и темно-серая рамка, которые вы можете наблюдать на рисунке 6.7, предоставляются визуальным построителем Qt Designer для удобства разработчика.

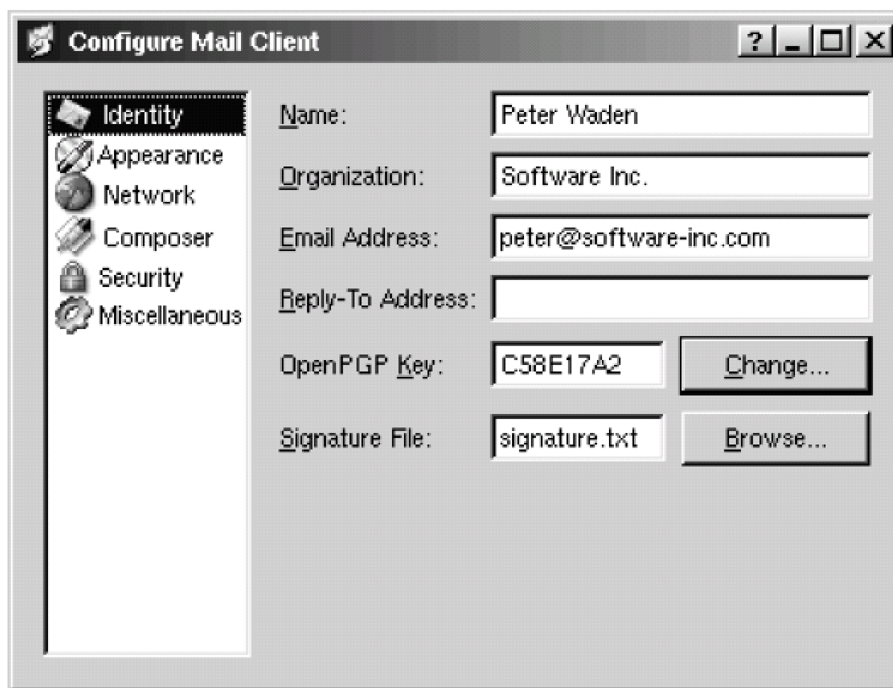


Рисунок 6.8. Диалог Configure.

Диалог **Configure**, изображенный на рисунке 6.8, может служить примером использования **QWidgetStack**. В левой части окна диалога находится **QListBox**, а в правой -- **QWidgetStack**. Каждому элементу в **QListBox** соответствует своя страница в **QWidgetStack**. Формы подобного рода очень просто создаются в Qt Designer:

- Создается новая форма из шаблона "**Dialog**" или "**Widget**".
- На форму добавляются **QListBox** и **QWidgetStack**.
- Каждая страница **QWidgetStack** заполняется необходимыми виджетами. (Чтобы создать новую страницу -- щелкните правой кнопкой мыши и выберите из контекстного меню пункт **Add Page**. Чтобы перейти к другой странице -- щелкните мышкой по одной из кнопок, расположенных в правом верхнем углу.)
- Объедините **QListBox** и **QWidgetStack** менеджером горизонтального размещения.
- Соедините сигнал **highlighted(int)**, от **QListBox**, со слотом **raiseWidget(int)**, компонента **QWidgetStack**.
- Установите свойство **currentItem** (**QListBox**) равным 0.

Поскольку мы реализовали механизм смены страниц стандартными сигналами и слотами, диалог будет способен выполнять корректный переход от страницы к странице, даже во время предварительного просмотра в Qt Designer.

6.4 Области просмотра с прокруткой

Класс **QScrollView** представляет собой область просмотра с двумя полосами прокрутки и "угловым" компонентом, находящимся в правом нижнем углу (обычно -- пустой **QWidget**). Если необходимо добавить полосы прокрутки к своему виджету, то намного проще воспользоваться готовым **QScrollView**, чем добавлять компоненты **QScrollBar** к своему виджету и писать код, реализующий их функциональность.

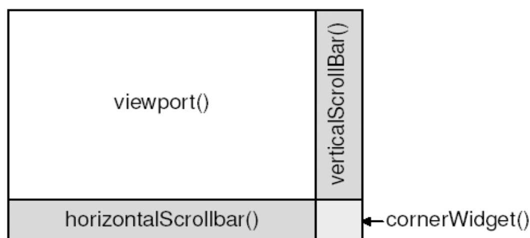


Рисунок 6.9. Виджеты, составляющие QScrollView.

Самый простой способ добавить визуальный компонент в **QScrollView** -- это вызвать метод **addChild()**, указав необходимый подчиненный виджет в качестве аргумента. **QScrollView** автоматически переподчинит визуальный компонент, став его владельцем. Например, пусть необходимо окружить компонент **IconEditor**, который был разработан нами в Главе 5, полосами прокрутки. Для этого можно было бы написать следующий код:

```
#include <qapplication.h>
#include <qscrollview.h>

#include "iconeditor.h"

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

  QScrollView scrollView;
  scrollView.setCaption(QObject::tr("Icon Editor"));
  app.setMainWidget(&scrollView);

  IconEditor *iconEditor = new IconEditor;
  scrollView.addChild(iconEditor);

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

По-умолчанию, полосы прокрутки отображаются только в том случае, когда подчиненный виджет не умещается в область просмотра (**viewport**). Однако, следующий код вынудит **QScrollView** всегда показывать их:

```
scrollView.setHScrollBarMode(QScrollView::AlwaysOn);
scrollView.setVScrollBarMode(QScrollView::AlwaysOn);
```

Когда изменяется "идеальный" размер подчиненного виджета, **QScrollView** автоматически адаптируется под новые условия.

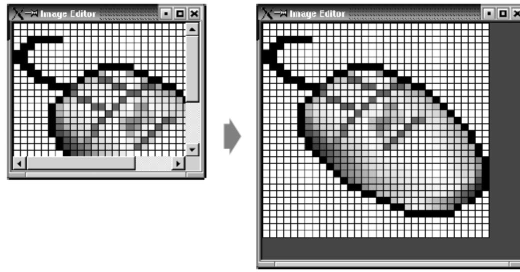


Рисунок 6.10. Изменение размеров QScrollView.

Еще один способ добавить полосы прокрутки к своему виджету -- использовать **QScrollView** в качестве класса-предка и перекрыть метод **drawContents()**. Такой подход реализован в классах **QIconView**, **QListBox**, **QListView**, **QTable** и **QTextEdit**. Если вашему виджету необходимы полосы прокрутки, то лучшим решением будет породить класс виджета от **QScrollView.**
Чтобы продемонстрировать это на примере, попробуем написать новую версию класса **IconEditor**, породив его от **QScrollView**. Назовем новый класс **ImageEditor**, поскольку полосы прокрутки дают нам возможность работать с изображениями большого размера.

```
#ifndef IMAGEEDITOR_H
#define IMAGEEDITOR_H

#include <qimage.h>
#include <qscrollview.h>

class ImageEditor : public QScrollView
{
  Q_OBJECT
  Q_PROPERTY(QColor penColor READ penColor WRITE setPenColor)
  Q_PROPERTY(QImage image READ image WRITE setImage)
  Q_PROPERTY(int zoomFactor READ zoomFactor WRITE setZoomFactor)

public:
  ImageEditor(QWidget *parent = 0, const char *name = 0);

  void setPenColor(const QColor &newColor);
  QColor penColor() const { return curColor; }
  void setZoomFactor(int newZoom);
  int zoomFactor() const { return zoom; }
  void setImage(const QImage &newImage); const
  QImage &image() const { return curImage; }

protected:
  void contentsMousePressEvent(QMouseEvent *event);
  void contentsMouseMoveEvent(QMouseEvent *event);
  void drawContents(QPainter *painter, int x, int y, int width, int height);

private:
  void drawImagePixel(QPainter *painter, int i, int j);
  void setImagePixel(const QPoint &pos, bool opaque);
  void resizeContents();
  QColor curColor;
  QImage curImage; int zoom;
};

#endif
```

Заголовочный файл очень похож на предыдущий. Основное отличие состоит в том, что теперь предком является не **QWidget**, а **QScrollView**. Другие, менее значимые отличия, мы рассмотрим в процессе описания реализации класса.

```
ImageEditor::ImageEditor(QWidget *parent, const char *name)
    : QScrollView(parent, name, WStaticContents | WNoAutoErase)
{
  curColor = black;
  zoom = 8;
  curImage.create(16, 16, 32);
```

```
  curImage.fill(qRgba(0, 0, 0, 0));
  curImage.setAlphaBuffer(true);
  resizeContents();
}
```

Родительскому конструктору передаются флаги **WStaticContents** и **WNoAutoErase**. Они необходимы для области просмотра. Мы не назначаем политики изменения размеров, поскольку значения по-умолчанию (**Expanding**, **Expanding**) нас вполне устраивают. В конструкторе ранней версии мы не вызывали **updateGeometry()**, поскольку начальные размеры виджета могли зависеть от действий менеджеров размещения. Однако в данном случае, нам необходимо задать начальные размеры компонента, что мы и делаем вызовом **resizeContents()**.

```
void ImageEditor::resizeContents()
{
  QSize size = zoom * curImage.size();
  if (zoom >= 3)
    size += QSize(1, 1);
  QScrollView::resizeContents(size.width(), size.height());
}
```

Приватная функция **resizeContents()** вызывает унаследованный метод **QScrollView::resizeContents()**, передавая ему начальные размеры содержимого **QScrollView**, который в свою очередь отображает полосы прокрутки, в зависимости от размеров содержимого и области просмотра.
Нам нет необходимости перекрывать функцию **sizeHint()**. Компонент **QScrollView** автоматически вычисляет "идеальный" размер, отталкиваясь от размера содержимого области просмотра.

```
void ImageEditor::setImage(const QImage &newImage)
{
  if (newImage != curImage) {
    curImage = newImage.convertDepth(32);
    curImage.detach();
    resizeContents();
    updateContents();
  }
}
```

В большинстве случаев, в оригинальном **IconEditor**, когда необходимо было послать компоненту событие "**paint**", мы вызывали методы **update()** и **updateGeometry()** -- чтобы объявить об изменении "идеальных" размеров. В новой версии, эти вызовы заменены на **updateContents()** и **resizeContents()**, соответственно.

```
void ImageEditor::drawContents(QPainter *painter, int, int, int, int)
{
  if (zoom >= 3) {
    painter->setPen(colorGroup().foreground());
    for (int i = 0; i <= curImage.width(); ++i)
      painter->drawLine(zoom * i, 0,
                        zoom * i, zoom * curImage.height());
    for (int j = 0; j <= curImage.height(); ++j)
      painter->drawLine(0, zoom * j,
                        zoom * curImage.width(), zoom * j);
  }

  for (int i = 0; i < curImage.width(); ++i) {
    for (int j = 0; j < curImage.height(); ++j)
      drawImagePixel(painter, i, j);
  }
}
```

QScrollView вызывает функцию **drawContents()**, чтобы перерисовать содержимое области просмотра. Объект **QPainter** уже инициализирован, в соответствии с позициями движков в полосах прокрутки, поэтому мы просто "рисуем", точно так же как в обработчике события **paintEvent()**.

Второй, третий, четвертый и пятый аргументы определяют координаты прямоугольника, который должен быть перерисован. Мы могли бы использовать их, чтобы перерисовывать только видимую часть изображения, но для упрощения примера мы перерисовыаем все изображение.

Функция **drawImagePixel()**, обращение к которой стоит в конце **drawContents()**, осталась без изменений (см. оригинальную версию), поэтому здесь мы ее рассматривать не будем.

```
void ImageEditor::contentsMousePressEvent(QMouseEvent *event)
{
  if (event->button() == LeftButton)
    setImagePixel(event->pos(), true);
  else if (event->button() == RightButton)
    setImagePixel(event->pos(), false);
}

void ImageEditor::contentsMouseMoveEvent(QMouseEvent *event)
{
  if (event->state() & LeftButton)
    setImagePixel(event->pos(), true);
  else if (event->state() & RightButton)
    setImagePixel(event->pos(), false);
}
```

События от мыши, направляемые содержимому **QScrollView**, обрабатываются специальными функциями обработчиками, имена которых начинаются со слова **contents**. Прежде, чем события будут переданы обработчикам, **QScrollView** выполнит преобразование координат из системы координат области просмотра в систему координат содержимого, поэтому у нас не возникает необходимости в написании дополнительного кода, выполняющего эти действия.

```
void ImageEditor::setImagePixel(const QPoint &pos, bool opaque)
{
  int i = pos.x() / zoom;
  int j = pos.y() / zoom;

  if (curImage.rect().contains(i, j)) {
    if (opaque)
      curImage.setPixel(i, j, penColor().rgb());
    else curImage.setPixel(i, j, qRgba(0, 0, 0, 0));

    QPainter painter(viewport());
    painter.translate(-contentsX(), -contentsY());
    drawImagePixel(&painter, i, j);
  }
}
```

Функция **setImagePixel()** вызывается из **contentsMousePressEvent()** и **contentsMouseMoveEvent()**, для закрашивания и очистки пикселей. Код функций, по большей части, остался без изменений, за исключением способа инициализации объекта **QPainter**. В данном случае, мы передаем ему **viewport()**, в качестве владельца, поскольку рисование будет производиться на поверхности области просмотра, а затем выполняем преобразование системы координат, чтобы учесть положение движков на полосах прокрутки.
Последние три строки, которые работают с **QPainter**, можно было бы заменить одной строкой:

```
updateContents(i * zoom, j * zoom, zoom, zoom);
```

Которая сообщила бы **QScrollView** о необходимости перерисовать один квадратик, который соответствует текущему пикселю. Но поскольку у нас функция **drawContents()** не оптимизирована, то приходится создавать **QPainter** и рисовать изображение пикселя самостоятельно.
Если теперь мы попробуем поработать с **ImageEditor**, то мы практически не заметим разницы с оригинальным **IconEditor**, вставленным в **QScrollView**. Однако, другие виджеты, порожденные от **QScrollView**, используют дополнительные преимущества родительского класса. Например, **QTextEdit** выполняет перенос текста по словам.

Обратите внимание: вам наверняка придется использовать класс **QScrollView**, в качестве предка, если размеры отображаемого содержимого очень велики, поскольку некоторые оконные подсистемы не в состоянии отобразить виджеты, размеры которых превышают величину 32767 пикселей.

Еще один важный момент, которого мы не коснулись здесь: мы можем вставлять подчиненные виджеты в область просмотра, вызовом функции **addWidget()**, и перемещать вызовом **moveWidget()**. Всякий раз, когда пользователь перемещается по области просмотра, с помощью полос прокрутки, **QScrollView** автоматически перемещает подчиненные виджеты на экране. (Если подчиненных виджетов слишком много, то прокрутка может существенно замедляться. Чтобы оптимизировать этот процесс, можно вызвать **enableClipper(true)**.) В качестве примера, использующего подобный подход, можно привести **web**-браузер, в котором большая часть содержимого может отрисовываться непосредственно в области просмотра, но кнопки и поля ввода на формах должны быть представлены в виде виджетов.

6.5 Стыкуемые окна

Стыкуемые окна -- это окна, которые могут отстыковываться и пристыковываться к специальным областям стыковки. Самый яркий пример, пожалуй, это панели инструментов.
Объекты класса **QMainWindow** предоставляют в распоряжение программиста четыре области стыковки: вверху, внизу, слева и справа от центрального виджета. Когда создаются экземпляры класса **QToolBar**, они автоматически пристыковываются к верхней области окна-владельца.

Рисунок 6.11. "Плавающие" пристыковываемые окна.

Каждое из таких окон имеет "рукоятку". Она отображается в виде двух серых линий в левой или в верхней части окна, как это показано на рисунке 6.12. Ухватив мышью за "рукоятку", пользователь может перемещать стыкуемые окна из одной области стыковки в другую. Он так же может отделять стыкуемые окна от главного окна приложения. Отстыкованные окна могут свободно перемещаться по всей поверхности экрана, они имеют свою полосу заголовка и могут иметь собственную кнопку закрытия окна. Отстыкованные окна всегда отображаются поверх родительского окна.

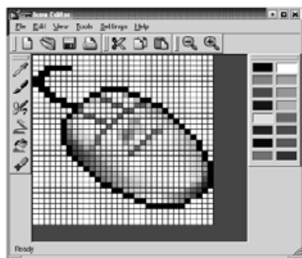


Рисунок 6.12. QMainWindow с пятью стыкуемыми окнами.

Чтобы кнопка закрытия отображалась на отстыкованном окне, необходимо вызвать **setCloseMode()**:

```
dockWindow->setCloseMode(QDockWindow::Undocked);
```

Область стыковки -- **QDockArea**, имеет свое контекстное меню, со списком всех пристыкованных окон и панелей инструментов. После того, как отстыкованное окно было закрыто пользователем, оно может быть восстановлено с помощью этого меню.

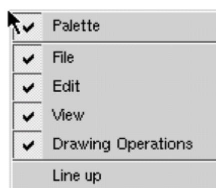


Рисунок 6.13. Контекстное меню QDockArea.

Стыкуемые окна должны быть потомками класса **QDockWindow**. Если вам нужна панель инструментов с кнопками и некоторыми другими виджетами, то для этой цели прекрасно подойдет **QToolBar**, который является наследником **QDockWindow**. Ниже приводится пример создания экземпляра класса **QToolBar**, на который помещаются **QComboBox**, **QSpinBox** и ряд дополнительных кнопок. Полученная панель инструментов размещается в нижней области стыковки:

```
QToolBar *toolBar = new QToolBar(tr("Font"), this);
QComboBox *fontComboBox = new QComboBox(true, toolBar);

QSpinBox *fontSize = new QSpinBox(toolBar);
boldAct->addTo(toolBar);
italicAct->addTo(toolBar);
underlineAct->addTo(toolBar);
moveDockWindow(toolBar, DockBottom);
```

Эта панель будет выглядеть просто отвратительно, если пользователь переместит ее в левую или правую область стыковки, из-за **QComboBox** и **QSpinBox**. Чтобы предотвратить такую возможность, мы можем запретить стыковку к левой и правой областям, вызовом **QMainWindow:: setDockEnabled()**:

```
setDockEnabled(toolBar, DockLeft, false);
setDockEnabled(toolBar, DockRight, false);
```

Если необходимо создать нечто более похожее на плавающее окно или палитру инструментов, то можно напрямую обращаться к **QDockWindow**, вызывая метод **setWidget()**, чтобы добавить виджет в окно. Если необходимо предоставить пользователю возможность изменять размеры пристыкованного окна, то для этого можно воспользоваться функцией **setResizeEnabled()**.

Если виджет должен изменять свой вид, в зависимости от того, к какой из областей стыковки он присоединен, то для этого необходимо перекрыть метод **QDockWindow:: setOrientation()** и выполять все необходимые действия в нем.

Если необходимо сохранять положение всех панелей инструментов и других стыкуемых окон, чтобы потом, на следующем запуске приложения восстанавливать его, можно написать код, который очень похож на тот, который мы разбирали ранее, используя оператор "**<<**" класса **QMainWindow**, для записи в файл, и "**>>**" -- для восстановления из файла.

Приложения, подобные Microsoft Visual Studio и Qt Designer очень широко используют стыкуемые окна, чтобы сделать интерфейс с пользователем более гибким.

6.6 Многодокументный интерфейс

Приложения, которые могут работать с несколькими документами, открываемыми в отдельных окнах и расположенных внутри главного окна, называют MDI-приложениями (MDI -- от англ. Multiple Document Interface). В Qt подобный интерфейс создается с помощью класса QWorkspace, назначаемого центральным виджетом. Каждое окно с открытым документом становится подчиненным, по отношению к **QWorkspace**.

В этом разделе мы создадим приложение **Editor** (текстовый редактор), изображенное на рисунке 6.14, чтобы продемонстрировать принципы создания MDI-приложений и оконных меню.

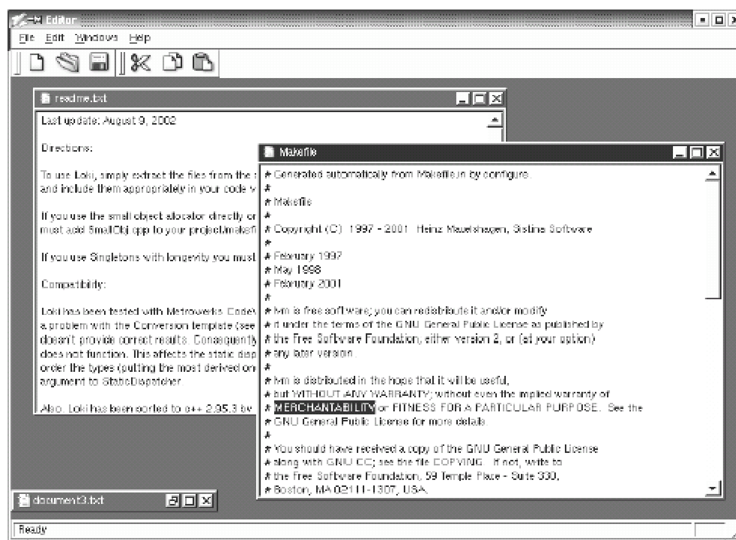


Рисунок 6.14. Внешний вид приложения Editor.

Приложение состоит из двух классов: **MainWindow** и **Editor**. Полный код приложения находится на CD, сопровождающем книгу, а поскольку он во многом похож на код, который мы писали в приложении **Spreadsheet** (в первой части книги), то мы будем описывать только ту часть реализации, которая является для нас еще незнакомой.

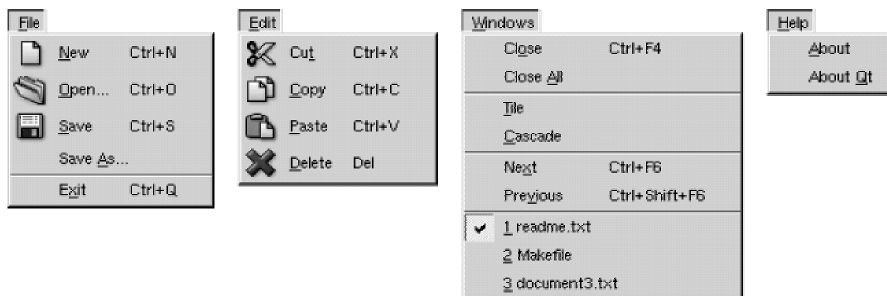


Рисунок 6.15. Меню приложения Editor.

Начнем с класса **MainWindow**.

```
MainWindow::MainWindow(QWidget *parent, const char *name)
    : QMainWindow(parent, name)
{
  workspace = new QWorkspace(this);
  setCentralWidget(workspace);
  connect(workspace, SIGNAL(windowActivated(QWidget *)),
          this, SLOT(updateMenus()));
  connect(workspace, SIGNAL(windowActivated(QWidget *)),
          this, SLOT(updateModIndicator()));
  createActions();
  createMenus();
  createToolBars();
  createStatusBar();

  setCaption(tr("Editor"));
  setIcon(QPixmap::fromMimeSource("icon.png"));
}
```

В конструкторе создается экземпляр класса **QWorkspace** и назначается центральным виджетом. Затем мы соединяем сигнал **windowActivated()**, класса **QWorkspace**, с двумя приватными слотами. Эти слоты гарантируют, что меню и строка состояния всегда будут соответствовать текущему активному окну.

```
void MainWindow::newFile()
{
  Editor *editor = createEditor();
```

```
  editor->newFile();
  editor->show();
}
```

Слот **newFile()** соответствует пункту меню **File|New**. Он создает новое окно (класса **Editor**) с документом, вызывая приватную функцию **createEditor()**.

```
Editor *MainWindow::createEditor()
{
  Editor *editor = new Editor(workspace);
  connect(editor, SIGNAL(copyAvailable(bool)),
          this, SLOT(copyAvailable(bool)));
  connect(editor, SIGNAL(modificationChanged(bool)),
          this, SLOT(updateModIndicator()));
  return editor;
}
```

Функция **createEditor()** создает виджет класса **Editor** и устанавливает два соединения типа сигнал-слот. Первое соответствует пунктам меню **Edit|Cut** и **Edit|Copy**. Доступность этих пунктов меню разрешается или запрещается, в зависимости от наличия выделенного текста. Второе соединение отвечает за обновление индикатора MOD (признак наличия в документе несохраненных изменений), который находится в строке состояния.
Поскольку мы имеем дело с многодокументным интерфейсом, то вполне возможно, что одновременно могут оказаться открытыми несколько окон с документами. Вас может обеспокоить этот факт, поскольку интерес для нас представляют сигналы **copyAvailable(bool)** и **modificationChanged()**, исходящие только от активного окна. На самом деле это не может служить причиной для беспокойства, поскольку сигналы могут подавать только активные окна.

```
void MainWindow::open()
{
  Editor *editor = createEditor();
  if (editor->open())
    editor->show();
  else
    editor->close();
}
```

Функция **open()** соответствует пункту меню **File|Open**. Она создает новое окно **Editor** и вызывает метод **Editor::open()**. Если функция **Editor::open()** завершается с ошибкой, то окно редактора просто закрывается, поскольку пользователь уже был извещен о возникших проблемах.

```
void MainWindow::save()
{
  if (activeEditor()) {
    activeEditor()->save();
    updateModIndicator();
  }
}
```

Слот **save()** вызывает функцию **save()** активного окна. Опять таки, весь код, который фактически сохраняет файл, находится в классе **Editor**.

```
Editor *MainWindow::activeEditor()
{
  return (Editor *)workspace->activeWindow();
}
```

Приватная функция **activeEditor()** возвращает указатель на активное окно редактора.

```
void MainWindow::cut()
{
  if (activeEditor())
    activeEditor()->cut();
}
```

Слот **cut()** вызывает функцию **cut()** активного окна. Слоты **copy()**, **paste()** и **del()** реализованы аналогичным образом.

```
void MainWindow::updateMenus()
{
  bool hasEditor = (activeEditor() != 0);
  saveAct->setEnabled(hasEditor);
  saveAsAct->setEnabled(hasEditor);
  pasteAct->setEnabled(hasEditor);
  deleteAct->setEnabled(hasEditor);
  copyAvailable(activeEditor()
              && activeEditor()->hasSelectedText());
  closeAct->setEnabled(hasEditor);
  closeAllAct->setEnabled(hasEditor);
  tileAct->setEnabled(hasEditor);
  cascadeAct->setEnabled(hasEditor);
  nextAct->setEnabled(hasEditor);
  previousAct->setEnabled(hasEditor);

  windowsMenu->clear();
  createWindowsMenu();
}
```

Слот **updateMenus()** вызывается всякий раз, когда активизируется другое окно (или когда закрывается последнее окно с документом), с целью обновления системы меню. Большинство из пунктов меню имеют смысл только при наличии активного дочернего окна, поэтому мы запрещаем некоторые пункты меню, если нет ни одного окна с открытым документом. Затем очищается меню **Windows** и вызывается функция **createWindowsMenu()**, которая обновляет список открытых дочерних окон.

```
void MainWindow::createWindowsMenu()
{
  closeAct->addTo(windowsMenu);
  closeAllAct->addTo(windowsMenu);
  windowsMenu->insertSeparator();
  tileAct->addTo(windowsMenu);
  cascadeAct->addTo(windowsMenu);
  windowsMenu->insertSeparator();
  nextAct->addTo(windowsMenu);
  previousAct->addTo(windowsMenu);

  if (activeEditor()) {
    windowsMenu->insertSeparator();
    windows = workspace->windowList();
    int numVisibleEditors = 0;

    for (int i = 0; i < (int)windows.count(); ++i) {
      QWidget *win = windows.at(i);
      if (!win->isHidden()) {
        QString text = tr("%1 %2")
                       .arg(numVisibleEditors + 1)
                       .arg(win->caption());
        if (numVisibleEditors < 9)
          text.prepend("&");
        int id = windowsMenu->insertItem(
                     text, this, SLOT(activateWindow(int)));
        bool isActive = (activeEditor() == win);
        windowsMenu->setItemChecked(id, isActive);
        windowsMenu->setItemParameter(id, i);
        ++numVisibleEditors;
      }
    }
  }
}
```

Приватная функция **createWindowsMenu()** заполняет меню **Windows** действиями (**action**) и дополняет списком открытых окон. Перечень пунктов типичен для меню подобного рода и соответствующие им действия легко реализуются с помощью слотов **QWorkspace -- closeActiveWindow()**, **closeAllWindows()**, **tile()** и **cascade()**.

Активное окно, в списке, отмечается маркером, напротив имени документа. Когда пользователь выбирает пункт меню, соответствующий открытому документу, вызывается слот **activateWindow()**, которому в качестве аргумента передается индекс в массиве **windows**. Это очень похоже на то, что мы делали в Главе 3, когда создавали список недавно открывавшихся документов.

Для первых девяти пунктов меню мы добавили символ амперсанда, перед порядковым номером пункта меню, чтобы можно было быстро перемещаться между открытыми документами, с помощью горячих клавиш.

```
void MainWindow::activateWindow(int param)
{
  QWidget *win = windows.at(param);
  win->show();
  win->setFocus();
}
```

Функция **activateWindow()** вызывается, когда пользователь выбирает какое либо окно с документом, из меню **Windows**. Параметр **param** -- это индекс выбранного окна, в массиве **windows**.

```
void MainWindow::copyAvailable(bool available)
{
  cutAct->setEnabled(available);
  copyAct->setEnabled(available);
}
```

Слот **copyAvailable()** вызывается, когда выделяется какой либо текст (или наоборот, когда выделение снимается) в окне редактора. Он так же вызывается из **updateMenus()**. И разрешает или запрещает пункты меню **Cut** и **Copy**.

```
void MainWindow::updateModIndicator()
{
  if (activeEditor() && activeEditor()->isModified())
    modLabel->setText(tr("MOD"));
  else
    modLabel->clear();
}
```

Функция **updateModIndicator()** обновляет индикатор MOD в строке состояния. Вызывается при любом изменении текста в окне редактора, а так же при активации другого окна.

```
void MainWindow::closeEvent(QCloseEvent *event)
{
  workspace->closeAllWindows();
  if (activeEditor())
    event->ignore();
  else
    event->accept();
}
```

Функция **closeEvent()** закрывает все дочерние окна. Если какое либо из окон "проигнорирует" событие "**close**" (например в том случае, когда пользователь отменил закрытие окна, имевшее несохраненные данные), то это событие так же игнорируется и главным окном приложения **MainWindow**. В противном случае событие "принимается" и Qt закрывает окно. Если не перекрыть этот обработчик, то у пользователя не будет возможности записать на диск несохраненные данные.

На этом мы завершаем обзор класса **MainWindow** и переходим к реализации класса **Editor**. Этот класс представляет собой одно дочернее окно. Он порожден от класса **QTextEdit**, который реализует всю необходимую функциональность по редактированию текста. Так же, как и любой другой виджет Qt, **QTextEdit** может использоваться как дочернее окно в рабочей области MDI.
Ниже приводится определение класса:

```
class Editor : public QTextEdit
{
  Q_OBJECT
```

```
public:
  Editor(QWidget *parent = 0, const char *name = 0);

  void newFile();
  bool open();
  bool openFile(const QString &fileName);
  bool save();
  bool saveAs();
  QSize sizeHint() const;

signals:
  void message(const QString &fileName, int delay);

protected:
  void closeEvent(QCloseEvent *event);

private:
  bool maybeSave();
  void saveFile(const QString &fileName);
  void setCurrentFile(const QString &fileName);
  QString strippedName(const QString &fullFileName);
  bool readFile(const QString &fileName);
  bool writeFile(const QString &fileName);

  QString curFile;
  bool isUntitled;
  QString fileFilters;
};
```

Четыре приватных функции, которые обсуждались нами при создании приложения **Spreadsheet**, аналогичным образом реализованы и в классе **Editor**. Это функции **maybeSave()**, **saveFile()**, **setCurrentFile()** и **strippedName()**.

```
Editor::Editor(QWidget *parent, const char *name)
    : QTextEdit(parent, name)
{
  setWFlags(WDestructiveClose);
  setIcon(QPixmap::fromMimeSource("document.png"));

  isUntitled = true;
  fileFilters = tr("Text files (*.txt)\n"
                   "All files (*)");
}
```

В конструкторе, с помощью функции **setWFlags()**, взводится флаг **WDestructiveClose**. Если конструктор класса не принимает флаги в качестве аргументов, как это имеет место быть в случае с **QTextEdit**, то мы можем установить флаги вызовом **setWFlags()**.

Так как мы позволяем пользователям одновременно открывать несколько документов, необходимо предусмотреть какие либо характеристики окон, чтобы потом пользователи могли как-то их отличать между собой, до того, как вновь создаваемые документы будут сохранены. Самый распространенный способ -- присваивать документам имена по-умолчанию, которые включают в себя порядковый номер (например, **document1.txt**). Для этой цели мы используем переменную **isUntitled**, которая отличает имена документов, уже существующих, и имена документов, которым имя еще не было присвоено пользователем.
После вызова конструктора должна вызываться одна из двух функций -- либо **newFile()**, либо **open()**.

```
void Editor::newFile()
{
  static int documentNumber = 1;

  curFile = tr("document%1.txt").arg(documentNumber);
  setCaption(curFile);
  isUntitled = true;
  ++documentNumber;
}
```

Функция **newFile()** генерирует новое имя документа, например **document2.txt**. Этот код помещен в **newFile()**, а не в конструктор, потому что нет необходимости вести счетчик создаваемых документов для тех из них, которые после конструирования объекта будут открываться функцией **open()**. Поскольку переменная **documentNumber** объявлена как статическая, то она существует в единственном экземпляре, для всех объектов класса **Editor**.

```
bool Editor::open()
{
  QString fileName =
          QFileDialog::getOpenFileName(".", fileFilters, this);
  if (fileName.isEmpty())
    return false;

  return openFile(fileName);
}
```

Функция **open()** пытается открыть существующий файл, с помощью вызова **openFile()**.

```
bool Editor::save()
{
  if (isUntitled) {
    return saveAs();
  } else {
    saveFile(curFile);
    return true; }
}
```

Функция **save()** использует переменную **isUntitled**, чтобы определить -- какую функцию вызывать: **saveFile()** или **saveAs()**.

```
void Editor::closeEvent(QCloseEvent *event)
{
  if (maybeSave())
    event->accept();
  else
    event->ignore();
}
```

За счет перекрытия родительского метода **closeEvent()** мы даем пользователю возможность сохранить имеющиеся изменения. Логика сохранения реализована в функции **maybeSave()**, которая выводит запрос перед пользователем: "Желаете ли вы сохранить имеющиеся изменения?". Если она возвращает true, то событие "close" принимается, в противном случае оно игнорируется и окно останется открытым.

```
void Editor::setCurrentFile(const QString &fileName)
{
  curFile = fileName;
  setCaption(strippedName(curFile));
  isUntitled = false;
  setModified(false);
}
```

Функция **setCurrentFile()** вызывается из **openFile()** и **saveFile()**, чтобы изменить содержимое переменных **curFile** и **isUntitled**, обновить заголовок окна и сбросить признак "**modified**". Класс **Editor** наследует методы **setModified()** и **isModified()** от своего предка -- **QTextEdit**, поэтому у нас нет необходимости "тащить" свой признак модификации документа. Когда пользователь вносит какие либо изменения в документ, **QTextEdit** выдает сигнал **modificationChanged()** и устанавливает признак модификации.

```
QSize Editor::sizeHint() const
{
  return QSize(72 * fontMetrics().width( x ),
               25 * fontMetrics().lineSpacing());
}
```

Функция **sizeHint()** возвращает "идеальные" размеры виджета, основываясь на размере символа 'x'. Класс **QWorkspace** использует эти размеры, чтобы назначить начальные размеры для окна с документом.

И в заключение приведем исходный текст файла **main.cpp**:

```
#include <qapplication.h>

#include "mainwindow.h"

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  MainWindow mainWin;
  app.setMainWidget(&mainWin);
  if (argc > 1) {
    for (int i = 1; i < argc; ++i)
      mainWin.openFile(argv[i]);
    } else {
      mainWin.newFile();
    }

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

Если пользователь задаст имена документов в командной строке, то приложение попытается загрузить их. В противном случае приложение создает пустой документ. Специфические ключи командной строки, такие как **-style** и **-font**, будут автоматически исключены из списка аргументов, конструктором **QApplication**. Так что, если мы дадим такую команду:

```
editor -style=motif readme.txt
```

То приложение на запуске откроет один единственный документ **readme.txt**.
Многодокументный интерфейс -- один из способов одновременной работы с несколькими документами. Другой способ состоит в том, чтобы использовать несколько окон верхнего уровня. Он был описан в разделе Работа с несколькими документами одновременно Главы 3.

7 ОБРАБОТКА СОБЫТИЙ

Приложения с графическим интерфейсом управляются событиями: все, что происходит в приложении -- есть результат обработки тех или иных событий. При разработке программ под Qt, задумываться о событиях приходится довольно редко, поскольку виджеты Qt выдают сигналы, когда происходит нечто значительное. События приобретают значение в том случае, когда необходимо создать новый виджет или когда нужно расширить функциональность существующего виджета.
В этой главе мы рассмотрим существующую модель обработки событий, расскажем о фильтрации событий и в заключение исследуем цикл обработки событий, на предмет того, как уменьшить время отклика приложения на действия пользователя, во время длительной обработки данных.

7.1 Обработчики событий

События генерируются оконной системой или Qt, в ответ на различные ситуации. Когда нажимается или отпускается клавиша на клавиатуре или кнопка мыши, генерируется соответствующее событие. Когда перемещается одно окно и в результате этого перемещения открывается другое, лежавшее ниже, возникает событие, которое сообщает открывшемуся окну о необходимости перерисовать себя. События генерируются всякий раз, когда виджет теряет или получает фокус ввода. В большинстве своем, события генерируются в ответ на действия пользователя, но иногда, например события от таймера, они генерируются системой независимо от пользователя.

Не надо путать события с сигналами. Сигналы необходимы для организации взаимодействий между виджетами, тогда как события необходимы для организации взаимодействия между виджетом и системой. Например, когда мы используем **QPushButton**, нас больше интересует сигнал **clicked()**, нежели события от мыши или клавиатуры, которые стали причиной появления сигнала. Но если мы разрабатываем новый класс, на подобие **QPushButton**, то нам придется писать код, который будет обрабатывать события от мыши и клавиатуры, и выдавать сигнал **clicked()** по мере необходимости.

События поступают к объектам в функцию **event()**, унаследованную от **QObject**. Реализация функции **event()** в **QWidget** передает наиболее употребимые типы событий специализированным обработчикам, таким как **mousePressEvent()**, **keyPressEvent()** и **paintEvent()**, остальные события игнорируются.

В предыдущих главах мы уже сталкивались с обработкой событий, при создании классов **MainWindow**, **IconEditor**, **Plotter**, **ImageEditor** и **Editor**. Полный список типов событий вы найдете в сопроводительной документации к классу **QEvent**. Кроме того, за программистом сохраняется возможность создания и диспетчеризации своих собственных типов событий. Нестандартные типы событий широко применяются в многопоточных приложениях, но это тема отдельной главы. В этой главе мы рассмотрим два типа событий: события от клавиатуры и события от таймера.

События от клавиатуры обрабатываются функциями **keyPressEvent()** и **keyReleaseEvent()**. В примере с виджетом **Plotter**, мы перекрывали родительский обработчик **keyPressEvent()**. Обычно программиста интересует только **keyPressEvent()**, поскольку к моменту нажатия интересующей его клавиши уже нажаты клавиши-модификаторы, а к моменту отпускания нужной клавиши, клавиши-модификаторы могут быть уже отжаты. К клавишам-модификаторам относятся: **Ctrl**, **Shift** и **Alt**. Состояние этих клавиш может быть получено вызовом функции **state()**. Например, представим, что нам необходимо написать виджет **CodeEditor** и реализовать обработчик событий от клавиатуры, который различал бы комбинации клавиш **Home** и **Ctrl**+**Home**, в этом случае мы могли бы написать следующий код:

```
void CodeEditor::keyPressEvent(QKeyEvent *event)
{
  switch (event->key()) {
    case Key_Home:
        if (event->state() & ControlButton)
          goToBeginningOfDocument();
        else
          goToBeginningOfLine();
        break;
    case Key_End:
        ...
    default:
```

```
      QWidget::keyPressEvent(event);
  }
}
```

Комбинации **Tab** и **Backtab** (**Shift**+**Tab**) -- особый случай. Они обрабатываются в **QWidget::event()** до того, как событие попадет в **keyPressEvent()**. Смысл этой комбинации заключается в передаче фокуса от одного виджета к другому, в заданной последовательности. Как правило, такое поведение нас вполне устраивает, но что делать, если необходимо реализовать иную семантику для данных комбинаций, например, чтобы клавишей **Tab** можно было оформлять отступы в **CodeEditor**? Выход довольно прост, он заключается в перекрытии метода предка **event()**:

```
bool CodeEditor::event(QEvent *event)
{
  if (event->type() == QEvent::KeyPress) {
    QKeyEvent *keyEvent = (QKeyEvent *)event;
    if (keyEvent->key() == Key_Tab) {
      insertAtCurrentPosition( \t );
      return true;
    }
  }
  return QWidget::event(event);
}
```

Если событие пришло от клавиатуры, то объект типа **QEvent** приводится к типу **QKeyEvent** и выполняется определение нажатой клавиши. Если это клавиша **Tab**, то выполняются некоторые действия и функция возвращает результат **true**, сообщая Qt о том, что событие обработано. Если функция вернет **false**, то **Qt** попробует вызвать метод **event()** владельца.
Использование объектов **QAction** дает более высокий уровень обслуживания событий. Например, если предположить, что **CodeEditor** имеет два публичных слота **goToBeginningOfLine()** и **goToBeginningOfDocument()** и **CodeEditor** назначен центральным виджетом для класса **MainWindow**, то можно было бы обслуживать комбинации клавиш следующим образом:

```
MainWindow::MainWindow(QWidget *parent, const char *name)
    : QMainWindow(parent, name)
{
  editor = new CodeEditor(this);
  setCentralWidget(editor);

  goToBeginningOfLineAct = new QAction(tr("Go to Beginning of Line"),
                                       tr("Home"), this);
  connect(goToBeginningOfLineAct, SIGNAL(activated()),
          editor, SLOT(goToBeginningOfLine()));
  goToBeginningOfDocumentAct = new QAction(tr("Go to Beginning of Document"),
                                           tr("Ctrl+Home"), this);
  connect(goToBeginningOfDocumentAct, SIGNAL(activated()),
          editor, SLOT(goToBeginningOfDocument()));
  ...
}
```

Такой способ облегчает добавление пунктов в меню или кнопок на панель инструментов, но об этом мы уже говорили в Главе 3. Если в меню не появляются пункты, описанные через **QAction**, то необходимо заменить **QAction** на **QAccel** -- класс, который используется **QAction** для обработки нажатий на комбинации клавиш.

Разница между этими двумя подходами (перекрытие метода **keyPressEvent()** и использование **QAction** или **QAccel**) очень похожа на разницу между перекрытием метода **resizeEvent()** и использованием дочерних классов от **QLayout**. Если вы создаете свой виджет, порождая его от **QWidget**, то скорее всего вам подойдет первый вариант, связанный с написанием нескольких своих обработчиков, с жестко зашитым поведением. Но если вы предполагаете использовать уже готовый виджет, то более удобен высокоуровневый подход, связанный с использованием **QAction**.

Другой распространенный тип событий -- события от таймера. В то время, как большинство событий связаны с действиями пользователя, события от таймера генерируются системой и позволяют организовать обработку данных через определенные интервалы времени. Этот тип событий может

использоваться, например, для создания мигающего курсора или просто для обновления изображения на экране.
С целью демонстрации обслуживания событий от таймера, создадим виджет **Ticker**. Он будет выводить строку текста и прокручивать ее справа-налево на один пиксель каждые 30 миллисекунд. Если ширина виджета больше ширины текста, то заданный текст будет нарисован столько раз, сколько уместится на виджете.

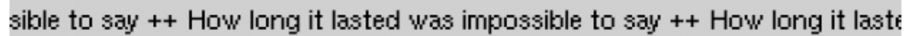


Рисунок 7.1. Внешний вид виджета Ticker.

Начнем с файла заголовка:

```
#ifndef TICKER_H
#define TICKER_H

#include <qwidget.h>

class Ticker : public QWidget
{
  Q_OBJECT
  Q_PROPERTY(QString text READ text WRITE setText)

public:
  Ticker(QWidget *parent = 0, const char *name = 0);

  void setText(const QString &newText);
  QString text() const { return myText; }
  QSize sizeHint() const;

protected:
  void paintEvent(QPaintEvent *event);
  void timerEvent(QTimerEvent *event);
  void showEvent(QShowEvent *event);
  void hideEvent(QHideEvent *event);

private:
  QString myText;
  int offset;
  int myTimerId;
};

#endif
```

Мы реализуем четыре обработчика событий, при чем с тремя из них (**timerEvent()**, **showEvent()** и **hideEvent()**) мы встречаемся впервые.
Перейдем к файлу с реализацией:

```
#include <qpainter.h>

#include "ticker.h"

Ticker::Ticker(QWidget *parent, const char *name)
    : QWidget(parent, name)
{
  offset = 0;
  myTimerId = 0;
}
```

Конструктор инициализирует переменную **offset** значением 0. Координата x, с которой будет выводится текст, получается из переменной **offset**.

```
void Ticker::setText(const QString &newText)
{
  myText = newText;
  update();
```

```
  updateGeometry();
}
```

Функция **setText()** запоминает текст, который должен выводиться на экран. Она вызывает **update()**, чтобы перерисовать виджет, а функцию **updateGeometry()** -- чтобы известить менеджер размещения об изменении "идеального" размера виджета.

```
QSize Ticker::sizeHint() const
{
  return fontMetrics().size(0, text());
}
```

Функция **sizeHint()** возвращает "идеальные" размеры области, которые необходимы для вывода текста. Функция **QWidget::fontMetrics()** возвращает экземпляр класса **QFontMetrics**, с помощью которого можно получить информацию об используемом шрифте. В данном случае он возвращает размеры области, в которую уместился бы заданный текст.

```
void Ticker::paintEvent(QPaintEvent *)
{
  QPainter painter(this);

  int textWidth = fontMetrics().width(text());
  if (textWidth < 1)
    return;
  int x = -offset;
  while (x < width()) {
    painter.drawText(x, 0, textWidth, height(),
                     AlignLeft | AlignVCenter, text());
    x += textWidth;
  }
}
```

Функция **paintEvent()** выводит текст, с помощью вызова **QPainter::drawText()**. С помощью **fontMetrics()** она определяет ширину текста и затем рисует его столько раз, сколько потребуется, чтобы заполнить виджет на всю ширину, учитывая значение переменной **offset**.

```
void Ticker::showEvent(QShowEvent *)
{
  myTimerId = startTimer(30);
}
```

Функция **showEvent()** запускает таймер. Функция **QObject::startTimer()** возвращает целое число, которое может быть использовано для идентификации таймера. Класс **QObject** может поддерживать несколько независимых таймеров, каждый со своим собственным временным интервалом. После вызова **startTimer()**, Qt будет автоматически генерировать события от таймера через интервалы времени, приблизительно равные 30-ти миллисекундам. Точность таймера зависит от операционной системы.
В принципе, **startTimer()** можно было бы вызвать и в конструкторе, но мы не сделали этого с целью экономии ресурсов системы, поскольку нет большого смысла в событиях от таймера, когда виджет невидим.

```
void Ticker::timerEvent(QTimerEvent *event)
{
  if (event->timerId() == myTimerId) {
    ++offset;
    if (offset >= fontMetrics().width(text()))
      offset = 0;
    scroll(-1, 0);
  } else {
    QWidget::timerEvent(event);
  }
}
```

Функция **timerEvent()** -- это обработчик событий от таймера и вызывается системой через заданные интервалы времени. Она увеличивает величину смещения на 1, чтобы создать эффект перемещения,

до тех пор, пока смещение не сравняется с шириной текста. Затем прокручивает содержимое виджета на 1 пиксель влево, вызовом функции **QWidget::scroll()**. Теоретически, вместо **scroll()** можно было бы вызвать **update()**, но функция **scroll()** более эффективна и к тому же предотвращает эффект мерцания, потому что она просто перемещает существующее на экране изображение и генерирует событие "**paint**" для очень узкой области, в данном случае область перерисовки имеет ширину в 1 пиксель. Если событие поступило не от того таймера, который нас интересует, то оно просто передается базовому классу.

```
void Ticker::hideEvent(QHideEvent *)
{
  killTimer(myTimerId);
}
```

Функция **hideEvent()** вызывает **QObject::killTimer()**, которая останавливает таймер.
Если необходимо создать несколько таймеров, то обработка событий от них может стать слишком громоздкой. В таких ситуациях проще создавать объекты класса **QTimer** для каждого таймера. **QTimer** выдает сигнал **timeout()** по истечении каждого интервала времени, кроме того, он предоставляет возможность создания таймеров-будильников, которые срабатывают один раз.

7.2 Установка фильтров событий

Одна из замечательных особенностей модели обработки событий в Qt -- возможность одного экземпляра **QObject** отслеживать события, предназначенные для другого экземпляра **QObject** до того, как последний получит их.
Предположим, что у нас имеется виджет **CustomerInfoDialog**, собранный из нескольких **QLineEdit**, и нам необходимо передавать фокус ввода, от одного к другому, нажатием на клавишу "пробел". Решение "в лоб" -- создать дочерний класс от **QLineEdit** и перекрыть обработчик события **keyPressEvent()**, в котором вызывать **focusNextPrevChild()**, примерно так:

```
void MyLineEdit::keyPressEvent(QKeyEvent *event)
{
  if (event->key() == Key_Space)
    focusNextPrevChild(true);
  else
    QLineEdit::keyPressEvent(event);
}
```

Однако это решение имеет массу недостатков. Поскольку **MyLineEdit** -- это нестандартный виджет, то нам придется приложить некоторые усилия, чтобы интегрировать его с Qt Designer, если захотим создавать формы с помощью визуального построителя. Кроме того, если потребуется, чтобы другие типы виджетов (такие как **QComboBoxe** и **QSpinBox**) так же поддерживали эту особенность, то мы будем вынуждены создать дочерние классы и для этих виджетов.

Более гибкое решение -- позволить **CustomerInfoDialog** отслеживать события, отправляемые подчиненным виджетам и реализовать необходимую функциональность. Делается это с помощью фильтров событий. Установка фильтра событий производится в два этапа:

- Регистрация фильтра событий, вызовом функции **installEventFilter()** того объекта, которому предназначены события.
- Создание обработчика перехваченных событий **eventFilter()**.

Регистрацию фильтра событий мы поместим в конструктор класса **CustomerInfoDialog**:

```
CustomerInfoDialog::CustomerInfoDialog(QWidget *parent,
                                       const char *name)
    : QDialog(parent, name)
{
  ...
  firstNameEdit->installEventFilter(this);
  lastNameEdit->installEventFilter(this);
  cityEdit->installEventFilter(this);
  phoneNumberEdit->installEventFilter(this);
}
```

После регистрации фильтра, все события, которые предназначены объектам **firstNameEdit**, **lastNameEdit**, **cityEdit** и **phoneNumberEdit**, сначала попадут в обработчик **CustomerInfoDialog::eventFilter()**. Ниже приводится исходный код функции **eventFilter()**:

```
bool CustomerInfoDialog::eventFilter(QObject *target, QEvent *event)
{
  if (target == firstNameEdit || target == lastNameEdit
          || target == cityEdit || target == phoneNumberEdit) {
    if (event->type() == QEvent::KeyPress) {
      QKeyEvent *keyEvent = (QKeyEvent *)event;
      if (keyEvent->key() == Key_Space) {
        focusNextPrevChild(true);
        return true;
      }
    }
  }
  return QDialog::eventFilter(target, event);
}
```

Прежде всего мы убеждаемся, что событие отправлено одному из **QLineEdit**. Не забывайте, что базовый класс **QDialog** может контролировать и другие виджеты. (В Qt 3.2 это не относится к **QDialog**. Однако, другие классы, такие как **QMainWindow**, отслеживают события некоторых из подчиненных виджетов по различным причинам.)

Если событие пришло от клавиатуры, то выполняется приведение к типу **QKeyEvent** и проверяется -- какая клавиша нажата. Если нажата клавиша "пробел", то вызывается функция **focusNextPrevChild()**, которая передает фокус вводв следующему виджету и возвращается результат **true**, сообщая Qt о том, что событие обработано. Если вернуть **false**, то Qt передаст событие объекту назначения.
Если событие порождено не клавишей "пробел", то управление передается функции **eventFilter()** базового класса.

В Qt предусмотрены пять уровней, на которых событие может быть перехвачено и обработано:
1 Обработка событий в функциях-обработчиках
Перекрытие обработчиков событий, таких как: **mousePressEvent()**, **keyPressEvent()** и **paintEvent()**, безусловно самый распространенный способ. Мы уже видели множество примеров тому.

2 Перекрытие метода QObject::event().
Внутри этого обработчика мы можем перехватывать события до того, как они попадут в специализированные функции-обработчики. Этот подход чаще всего используется для того, чтобы изменить реакцию виджета на клавишу табуляции, как это было показано ранее. Он так же используется для обработки событий, которые встречаются не так часто, например: **LayoutDirectionChange**. Если мы перекрываем функцию **event()**, то необходимо предусмотреть вызов обработчика **event()** базового класса, чтобы обработать события, которые нас не интересуют.

3 Установка фильтра событий для QObject.
После того, как фильтр будет зарегистрирован функцией **installEventFilter()**, все события, предназначающиеся указанному объекту, сначала будут попадать в обработчик **eventFilter()**. Такой способ мы использовали для перехвата событий от клавиши "пробел" в примере выше.

4 Установка фильтра событий объекта QApplication.
После регистрации фильтра, любое событие, предназначенное для любого объекта в приложении, будет сначала попадать в обработчик **eventFilter()**. Такой подход чаще всего используется в целях отладки и реализации в приложении скрытых сюрпризов (так называемых "пасхальных яиц").

5 Создание дочернего класса от QApplication и перекрытие метода notify().
Qt вызывает **QApplication::notify()**, чтобы передать событие приложению. Таким способом можно перехватить любое событие до того, как оно попадет в фильтр событий. Вообще фильтры событий более удобны, поскольку допускается одновременное существование любого количества фильтров, а функция **notify()** может быть только одна.

Большинство типов событий, включая события от мыши и клавиатуры, могут передаваться дальше. Если событие не было обработано по пути к объекту наначения, или самим объектом, то процесс обработки события повторяется, но на этот раз объектом назначения становится виджет-владелец. Так продолжается до тех пор, пока событие не будет обработано, либо пока событие не достигнет виджет самого верхнего уровня.

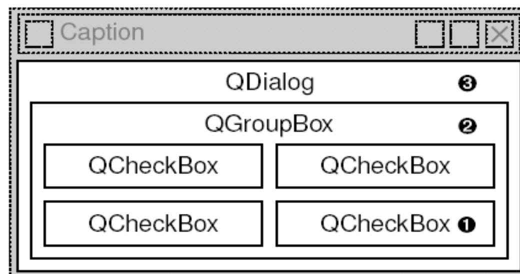


Рисунок 7.2. Обработка событий в окне диалога.

На рисунке 7.2 показан порядок передачи события от подчиненного виджета к владельцу. Когда пользователь нажимает какую либо клавишу, событие сначала передается виджету, который владеет фокусом ввода, в данном случае это **QCheckBox** в правом нижнем углу. Если виджет не обрабатывает событие, то оно передается виджету **QGroupBox** и затем **QDialog**.

7.3 Сокращение времени отклика при длительной обработке данных

С вызова функции **QApplication::exec()** начинается главный цикл обработки событий. Сначала Qt запускает несколько событий, чтобы отобразить и перерисовать виджеты. После этого в цикле постоянно выполняется проверка поступления новых событий и их передача виджетам приложения. Во время обработки одного события, в очередь могут поступать другие события. Если обработка какого либо события занимает продолжительное время, то это может отрицательно сказаться на времени отклика пользовательского интерфейса. Например, ни одно событие, поступившее от оконной системы во время сохранения файла на диск, не будет обработано до тех пор, пока файл не будет сохранен полностью. В течение времени, необходимого для сохранения файла, приложение никак не реагирует на запросы оконной системы.

Как одно из возможных решений данной проблемы -- создавать многопоточные приложения, в которых один поток будет отвечать за пользовательский интерфейс, а другой -- за дисковые операции (или любые другие действия, выполняющиеся продолжительное время). В этом случае приложение будет исправно откликаться на действия пользователя даже во время выполнения длительной обработки данных. Этот подход мы будем обсуждать в Главе 17.

Более простое решение -- вызывать **QApplication::processEvents()** как можно чаще, во время длительных операций. Эта функция выполняет обработку событий, ожидающих в очереди, и затем возвращает управление в вызвавшую функцию. Фактически, **QApplication::exec()** -- это не более чем цикл **while**, в котором вызывается функция **processEvents()**.
Ниже приводится пример того, как можно сократить время отклика приложения **Spreadsheet**, во время сохранения большого файла на диск (см. оригинальную версию):

```
bool Spreadsheet::writeFile(const QString &fileName)
{
  QFile file(fileName);
  ...
  for (int row = 0; row < NumRows; ++row) {
    for (int col = 0; col < NumCols; ++col) {
      QString str = formula(row, col);
      if (!str.isEmpty())
        out << (Q_UINT16)row << (Q_UINT16)col << str;
    }
    qApp->processEvents();
  }
  return true;
}
```

Однако в таких случаях существует одна опасность: пользователь может закрыть приложение до того, как файл будет сохранен, или даже может повторно вызывать процедуру сохранения файла. Эта проблема решается довольно просто -- нужно заменить вызов

```
qApp->processEvents();
```

на

```
qApp->eventLoop()->processEvents(QEventLoop::ExcludeUserInput);
```

который заставит Qt игнорировать события от мыши и клавиатуры.

Зачастую возникает необходимость вывести окно диалога, демонстрирующего ход выполнения длительной операции. Для подобных целей предназначен **QProgressDialog**, который имеет индикатор хода выполнения. У него так же имеется кнопка **Cancel**, с помощью которой пользователь может прервать операцию. Ниже представлен измененный вариант функции, которая демонстрирует пользователю ход операции сохранения файла:

```
bool Spreadsheet::writeFile(const QString &fileName)
{
  QFile file(fileName);
  ...
  QProgressDialog progress(tr("Saving file..."), tr("Cancel"),
                           NumRows);
  progress.setModal(true);
  for (int row = 0; row < NumRows; ++row) {
    progress.setProgress(row);
    qApp->processEvents();
    if (progress.wasCanceled()) {
      file.remove();
      return false;
    }
    for (int col = 0; col < NumCols; ++col) {
      QString str = formula(row, col);
      if (!str.isEmpty())
        out << (Q_UINT16)row << (Q_UINT16)col << str;
    }
  }
  return true;
}
```

На этот раз функция создает **QProgressDialog**, которому передает значение переменной **NumRows**, как общее число шагов. Затем, перед сохранением каждой строки, вызывается **setProgress()**, которая обновляет индикатор хода операции. Процент выполнения вычисляется компонентом **QProgressDialog** самостоятельно. Затем вызывается **QApplication::processEvents()**, чтобы обновить изображение на экране, а заодно и проверить -- не нажал ли пользователь на кнопку **Cancel**. Если кнопка **Cancel** была нажата, то операция сохранения прерывается и файл удаляется.

Мы не вызываем метод **show()** диалога, потому что он самостоятельно выполняет это действие. Если операция выполняется достаточно быстро, возможно потому что файл получился очень коротким, или потому что компьютер обладает очень высокой производительностью, **QProgressDialog** обнаружит это и вообще не будет выводить себя на экран.

Есть еще один способ выполнения длительных операций. Он сильно отличается от того, что был описан выше. Вместо того, чтобы в процессе длительных операций предусматривать обработку пользовательского интерфейса, можно наоборот, производить длительные операции, когда приложение простаивает. Этот способ пригоден в тех случаях, когда операция может быть безопасно прервана и затем опять продолжена.
В Qt такой вариант может быть реализован с помощью специального таймера -- таймера с нулевым интервалом. Такие таймеры генерируют события, когда очередь событий пуста. Ниже приводится пример реализации обработчика событий от такого таймера:

```
void Spreadsheet::timerEvent(QTimerEvent *event)
```

```
{
  if (event->timerId() == myTimerId) {
    while (step < MaxStep && !qApp->hasPendingEvents()) {
      performStep(step);
      ++step;
    }
  } else {
    QTable::timerEvent(event);
  }
}
```

Если функция **hasPendingEvents()** возвращает **true**, обработка данных приостанавливается и управление передается обратно в Qt. Обработка будет продолжена, когда Qt обслужит все события в очереди.

8 Двух- и трехмерная графика

В этой главе будут рассмотрены графические возможности Qt. Краеугольным камнем движка двухмерной графики в Qt является **QPainter**. Он может использоваться для рисования на поверхности виджета (на экране), во внутреннем буфере (**pixmap**) и на принтере. Кроме того, в состав Qt входит класс **QCanvas**, который позволяет создавать изображения из графических примитивов.
В качестве альтернативы **QPainter** и **QCanvas**, можно рассматривать библиотеку **OpenGL**. Она предоставляет механизмы создания трехмерной графики, но может использоваться и для рисования двухмерных изображений. Код, использующий **OpenGL** очень легко интегрируется в приложения Qt, мы продемонстрируем это на конкретных примерах.

8.1 Рисование средствами QPainter

Класс **QPainter** используется для создания изображений на "графических устройствах", таких как виджеты или карты пикселей (**pixmap**). Чаще всего он используется при создании нестандартных виджетов, для придания им уникального, ни на что не похожего, внешнего вида. Однако этот класс может использоваться и для вывода графики на принтер, более подробно мы коснемся этого вопроса немного ниже.

QPainter может рисовать простые геометрические фигуры: точки, линии, прямоугольники, эллипсы, дуги, сегменты круга, замкнутые ломаные (многоугольники) и кривые Безье. Он так же может отображать карты пикселей, рисунки и текст.
Когда конструктору **QPainter** передается устройство для рисования, он получает часть настроек от заданного устройства, оставшиеся параметры настройки заполняет значениями по-умолчанию. Эти настройки определяют способ рисования. Тремя наиболее важными характеристиками **QPainter** являются перо (**pen**), кисть (**brush**) и шрифт (**font**).

Перо используется для рисования линий и границ геометрических фигур. Оно характеризуется такими параметрами, как: цвет, толщина, стиль рисования линий, стиль оформления концов линий и стиль оформления углов.

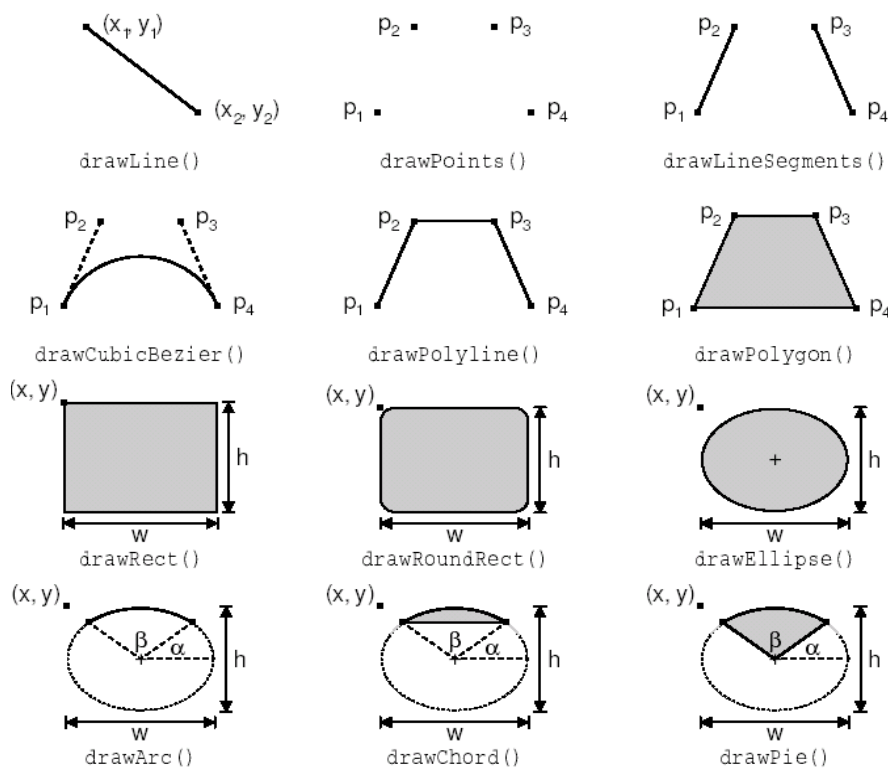


Рисунок 8.1. Методы класса QPainter, для рисования геометрических фигур.

Рисунок 8.2. Стили пера.

Кисть -- это шаблон, которым заполняются геометрические фигуры. Кисти характеризуются цветом и стилем.

Шрифт используется для рисования текста. Шрифт может иметь огромное количество атрибутов, среди них: название и размер.
Настройки этих характеристик могут быть выполнены с помощью функций **setPen()**, **setBrush()** и **setFont()**.

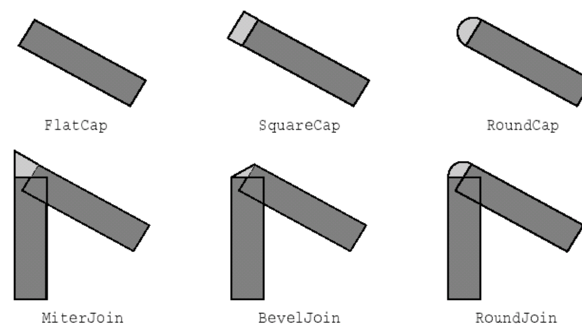


Рисунок 8.3. Стили оформления концов линий и углов.

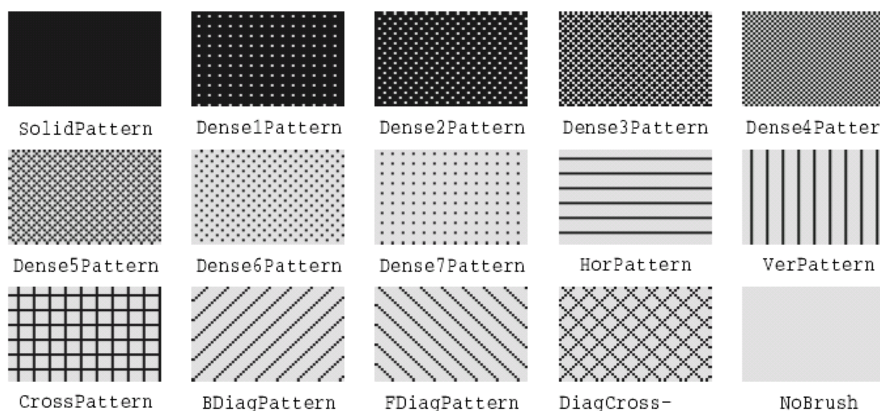


Рисунок 8.4. Стили кисти.

Ниже приводится код, который рисует эллипс, показанный на рисунке 8.5(а):

```
QPainter painter(this);
painter.setPen(QPen(black, 3, DashDotLine));
painter.setBrush(QBrush(red, SolidPattern));
painter.drawEllipse(20, 20, 100, 60);
```

Следующий код рисует сегмент круга, показанный на рисунке 8.5(б):

```
QPainter painter(this);
painter.setPen(QPen(black, 5, SolidLine));
painter.setBrush(QBrush(red, DiagCrossPattern));
painter.drawPie(20, 20, 100, 60, 60 * 16, 270 * 16);
```

Последние два аргумента **drawPie()** выражаются в 1/16 долях градуса.

(а) Эллипс (б) Сегмент круга (в) Кривая Безье

Рисунок 8.5 Примеры геометрических фигур

И наконец код, который рисует кривую Безье, показанную на рисунке 8.5(в):

```
QPainter painter(this);
QPointArray points(4);
points[0] = QPoint(20, 80);
points[1] = QPoint(50, 20);
points[2] = QPoint(80, 20);
points[3] = QPoint(120, 80);
painter.setPen(QPen(black, 3, SolidLine));
painter.drawCubicBezier(points);
```

Текущее состояние **QPainter** может быть сохранено на стеке, вызовом **save()** и восстановлено со стека, вызовом **restore()**. Это может потребоваться в том случае, когда необходимо на время изменить какие либо настройки, а затем восстановить их прежние значения.
Кроме перечисленных выше характеристик (перо, кисть и шрифт), **QPainter** имеет еще целый ряд параметров настройки:

- Цвет фона (**background color**), который используется для заливки геометрических фигур (под шаблоном, наносимым кистью), текста или рисунков, когда **background mode** имеет значение **OpaqueMode** (по-умолчанию: **TransparentMode**).
- Растровые операции (**raster operation**) определяют, как новое изображение должно накладываться на существующее. По-умолчанию: **CopyROP**, т.е. новое изображение (пиксели) просто копируется на устройство рисования, ранее находившееся там изображение игнорируется. В список растровых операций так же входят: **XorROP**, **NotROP**, **AndROP** и **NotAndROP**.
- Начальные координаты кисти (**brush origin**) задают начальную точку рисования шаблона кисти, обычно это левый верхний угол виджета.
- Врезка (**clip region**) определяет область устройства, на которой может производиться рисование. Операции рисования за пределами этой области -- игнорируются.
- Область просмотра (**viewport**), окно (**window**) и матрица преобразования (**world matrix**) определяют отношения между логической системой координат **QPainter** и системой координат физического устройства. Значения по-умолчанию принимаются таковыми, что эти две системы координат совпадают.

Рассмотрим подробнее систему координат, которая задается параметрами область просмотра (**viewport**), окно (**window**) и матрицей преобразования (**world matrix**). (В данном случае, термин "окно" не имеет ничего общего с виджетом самого верхнего уровня, а "область просмотра" -- с классом **QScrollView**.)

Понятия область просмотра и окно тесно связаны между собой. Область просмотра -- это произвольный прямоугольник, заданный физическими координатами. Окно -- описывает тот же самый прямоугольник, но уже в логических координатах. Когда выполняется рисование, то указываются логические координаты, которые затем преобразуются в физические.
По-умолчанию координаты области просмотра и окна совпадают с системой координат физического устройства. Например, если устройство отображения представляет из себя виджет, с размерами 320 X 200, то и область просмотра и окно имеют те же самые размеры. В данном случае логическая и физическая системы координат совпадают.
Подобный механизм дает возможность писать код, который не зависит от размера или разрешения устройства. Конечно же, мы и сами можем выполнять отображение логических координат в физические, но проще доверить эту работу классу **QPainter**. Например, представим, что нам необходимо работать в системе координат, ограниченной прямоугольником от (-50, -50) до (+50,

+50), когда точка с координатами (0, 0) находится в центре прямоугольника. В этом случае можно установить пареметры окна следующим образом:

```
painter.setWindow(QRect(-50, -50, 100, 100));
```

где первые два аргумента задают координаты верхнего левого угла (-50, -50), последние два аргумента (100, 100)-- ширину и высоту прямоугольника, соответственно. В данном случае, это означает, что логические координаты (-50, -50) соответствуют физическим координатам (0, 0), а логические координаты (+50, +50) -- физическим (320, 200). Изменять параметры области просмотра нет необходимости.

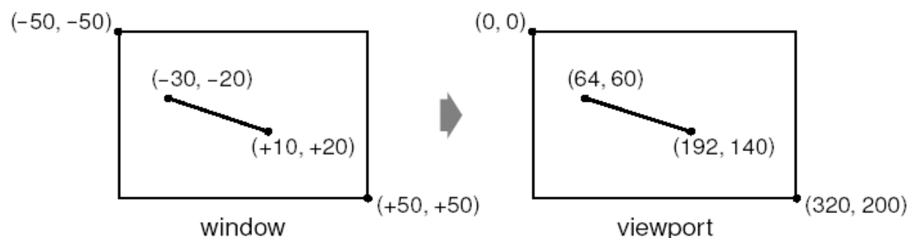


Рисунок 8.6. Преобразование логических координат в физические.

Теперь перейдем к матрице преобразований (**world matrix**). Она задает набор трансформаций, которые должны быть выполнены в дополнение к преобразованиям логических координат в физические. Это позволяет выполнять изменение масштаба, вращение и сдвиг рисуемых элементов. Например, если необходимо нарисовать текст под углом 45 градусов, то можно написать следующий код:

```
QWMatrix matrix;
matrix.rotate(45.0);
painter.setWorldMatrix(matrix);
painter.drawText(rect, AlignCenter, tr("Revenue"));
```

Здесь логические координаты, передаваемые в **drawText()**, сначала подвергаются трансформации, а затем отображаются в физические координаты.
Если указывается несколько трансформаций, то они применяются в порядке следования в исходном коде программы. Например, допустим, что необходимо повернуть изображение относительно точки с координатами (10, 20). Для этого можно задать следующий порядок трансформаций: сдвинуть окно так, чтобы центр вращения переместился в координаты (0, 0), повернуть изображение и затем выполнить обратный сдвиг:

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

Более простой способ -- воспользоваться методами 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, AlignCenter, tr("Revenue"));
```

Но если необходимо воспользоваться одним и тем же набором трансформаций несколько раз подряд, то вариант с **QWMatrix** даст значительный выигрыш по времени.
При необходимости, матрицу преобразований можно сохранить вызовом **saveWorldMatrix()** и затем восстановить вызовом **restoreWorldMatrix()**.

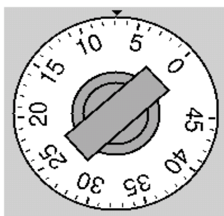


Рисунок 8.7. Внешний вид виджета OvenTimer.

С целью демонстрации использования преобразований, рассмотрим код виджета **OvenTimer** (таймер электропечи), который изображен на рисунке 8.7. Этот компонент моделирует поведение обычного таймера, которыми раньше, до появления цифровых часов и таймеров, снабжались электропечи. Пользователь может щелкнуть по риске на лимбе таймера, чтобы установить время ожидания, после чего ручка таймера начнет вращаться против часовой стрелки и по достижении нулевой отметки **OvenTimer** выдаст сигнал **timeout()**.

```
class OvenTimer : public QWidget
{
  Q_OBJECT

public:
  OvenTimer(QWidget *parent, const char *name = 0);

  void setDuration(int secs);
  int duration() const;
  void draw(QPainter *painter);

signals:
  void timeout();

protected:
  void paintEvent(QPaintEvent *event);
  void mousePressEvent(QMouseEvent *event);

private:
  QDateTime finishTime;
  QTimer *updateTimer;
  QTimer *finishTimer;
};
```

Класс **OvenTimer** порожден от класса **QWidget** и перекрывает два виртуальных метода предка: **paintEvent()** и **mousePressEvent()**.

```
#include <qpainter.h>
#include <qpixmap.h>
#include <qtimer.h>

#include <cmath>
using namespace std;

#include "oventimer.h"

const double DegreesPerMinute = 7.0;
const double DegreesPerSecond = DegreesPerMinute / 60;
const int MaxMinutes = 45;
const int MaxSeconds = MaxMinutes * 60;
const int UpdateInterval = 10;

OvenTimer::OvenTimer(QWidget *parent, const char *name)
    : QWidget(parent, name)
{
  finishTime = QDateTime::currentDateTime();
  updateTimer = new QTimer(this);
  finishTimer = new QTimer(this);
  connect(updateTimer, SIGNAL(timeout()), this, SLOT(update()));
  connect(finishTimer, SIGNAL(timeout()), this, SIGNAL(timeout()));
}
```

В конструкторе создаются два объекта **QTimer**: **updateTimer** -- для обновления изображения виджета, и **finishTimer** -- для выдачи сигнала **timeout()**, по достижении нулевой отметки.

```
void OvenTimer::setDuration(int secs)
{
  if (secs > MaxSeconds)
    secs = MaxSeconds;
  finishTime = QDateTime::currentDateTime().addSecs(secs);
  updateTimer->start(UpdateInterval * 1000, false);
  finishTimer->start(secs * 1000, true);
  update();
}
```

Функция **setDuration()** устанавливает продолжительность действия таймера в секундах. Аргумент **false**, передаваемый в функцию **dateTimer->start()** сообщает Qt о том, что это таймер с многократным срабатыванием. Период срабатывания таймера равен 10 секундам. Таймер **finishTimer** должен сработать всего один раз, поэтому в функцию **start()**, этого объекта, передается аргумент true. Конечное время работы таймера вычисляется сложением текущего времени, которое мы получаем вызовом **QDateTime::currentDateTime()** и времени ожидания.
Переменная **finishTime** имеет тип **QDateTime**, который в Qt отвечает за хранение даты и времени. Объекты этого типа становятся просто незаменимы в ситуациях, когда в отмеряемый интервал времени попадает граница суток.

```
int OvenTimer::duration() const
{
  int secs = QDateTime::currentDateTime().secsTo(finishTime);
  if (secs < 0)
    secs = 0;
  return secs;
}
```

Функция **duration()** возвращает число секунд, оставшихся до конца работы таймера.

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

Когда пользователь щелкает по лимбу таймера, вычисляется новый интервал действия таймера. Затем в очередь ставится событие "**paint**". Теперь, на вершине будет находиться выбранная пользователем риска.

```
void OvenTimer::paintEvent(QPaintEvent *)
{
  QPainter painter(this);
  int side = QMIN(width(), height());
  painter.setViewport((width() - side) / 2, (height() - side) / 2,
                      side, side);
  painter.setWindow(-50, -50, 100, 100);
  draw(&painter);
}
```

В обработчике **paintEvent()** устанавливается область просмотра (**viewport**), которая по своим размерам является наибольшей квадратной областью, которую можно разместить в виджете, а затем настраивается окно -- прямоугольник (-50, -50, 100, 100), с размерами 100 X 100. Макрос **QMIN()** возвращает наименьшее из двух аргументов.

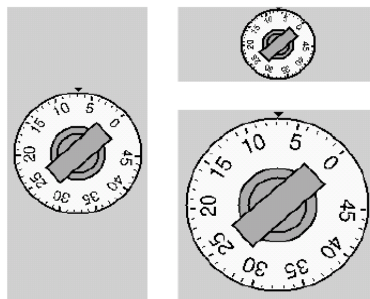


Рисунок 8.8. Внешний вид виджета OvenTimer с различными размерами.

Если область просмотра (**viewport**) не сделать квадратной, то лимб таймера будет рисоваться в виде эллипса, когда форма фиджета будет далека от квадратной. Вообще, во избежание подобного рода деформаций, необходимо задавать настройки области просмотра и окна с одинаковыми отношениями сторон.
Размеры окна (-50, -50, 100, 100) выбирались из следующих соображений:

- Функции рисования в **QPainter**, принимают значения координат в виде целых чисел. Если выбрать размер окна слишком маленьким, то координаты некоторых точек не смогут быть указаны достаточно точно, из-за возникающей проблемы округления.
- Если выбрать размер окна слишком большим, то при необходимости рисования текста функцией **drawText()**, нам придется выбирать шрифт большого размера.

В данной ситуации, выбор параметров (-50, -50, 100, 100) окна выглядит более предпочтительно, чем скажем (-5, -5, 10, 10) или (-2000, -2000, 4000, 4000).

Теперь перейдем к функции **draw()**:

```
void OvenTimer::draw(QPainter *painter)
{
  static const QCOORD triangle[3][2] = {
      { -2, -49 }, { +2, -49 }, { 0, -47 }
  };
  QPen thickPen(colorGroup().foreground(), 2);
  QPen thinPen(colorGroup().foreground(), 1);

  painter->setPen(thinPen);
  painter->setBrush(colorGroup().foreground());
  painter->drawConvexPolygon(QPointArray(3, &triangle[0][0]));
```

Рисование виджета начинается с маленького треугольника, который обозначает нулевую позицию вверху. Треугольник задается тремя, жестко зашитыми парами координат. Собственно рисование производится функцией **drawConvexPolygon()**. Треугольник можно было бы нарисовать функцией **drawPolygon()**, но если заранее известно, что многоугольник выпуклый, то вы можете сэкономить несколько микросекунд, за счет использования функции **drawConvexPolygon()**.
Одна из замечательных сторон механизма перехода от логических координат к физическим состоит в том, что мы можем жестко зашивать координаты точек в исходный код и при этом получать неплохие результаты, при изменении размеров виджета.

```
  painter->setPen(thickPen);
  painter->setBrush(colorGroup().light());
  painter->drawEllipse(-46, -46, 92, 92);
  painter->setBrush(colorGroup().mid());
  painter->drawEllipse(-20, -20, 40, 40);
  painter->drawEllipse(-15, -15, 30, 30);
```

Далее рисуются внешний и два внутренних круга. Внешний круг заполняется цветом "**light**" (обычно -- белый), Внутренние круги заполняются цветом "**mid**" (обычно -- серый).

```
  int secs = duration();
  painter->rotate(secs * DegreesPerSecond);
  painter->drawRect(-8, -25, 16, 50);

  for (int i = 0; i <= MaxMinutes; ++i) {
```

```
    if (i % 5 == 0) {
      painter->setPen(thickPen);
      painter->drawLine(0, -41, 0, -44);
      painter->drawText(-15, -41, 30, 25,
                        AlignHCenter | AlignTop,
                        QString::number(i));
      } else {
        painter->setPen(thinPen);
        painter->drawLine(0, -42, 0, -44);
      }
      painter->rotate(-DegreesPerMinute);
  }
}
```

Затем рисуются рукоятка и риски на лимбе. Напротив каждой пятой риски рисуется число, обозначающее количество минут. Функция **rotate()** вызывается для того, чтобы повернуть систему координат. В начальный момент, риска с отметкой "0" находилась вверху, теперь же она переместилась в точку, координаты которой зависят от оставшегося до срабатывания времени. Рукоятка рисуется после выполнения поворота, поскольку ее ориентация зависит от угла поворота.

В цикле **for**, по краю внешнего круга рисуются риски, а под ними -- числа, обозначающие количество минут, с шагом 5. В конце каждой итерации выполняется поворот системы координат по часовой стрелке на 7 градусов, что соответствует одной минуте. Таким образом, каждая следующая риска будет рисоваться на своем месте, хотя координаты в **drawLine()** и **drawText()** задаются одни и те же. Тут есть еще одна проблема, которую мы не учли -- мерцание. Виджет перерисовывается целиком каждые 10 секунд, что становится причиной появления эффекта подмаргивания изображения. Чтобы избавиться от нее, добавим двойную буферизацию. Для этого нужно передать родительскому конструктору флаг **WNoAutoErase** и изменить **paintEvent()** следующим образом:

```
void OvenTimer::paintEvent(QPaintEvent *event)
{
  static QPixmap pixmap;
  QRect rect = event->rect();

  QSize newSize = rect.size().expandedTo(pixmap.size());
  pixmap.resize(newSize);
  pixmap.fill(this, rect.topLeft());

  QPainter painter(&pixmap, this);
  int side = QMIN(width(), height());
  painter.setViewport((width() - side) / 2 - event->rect().x(),
                      (height() - side) / 2 - event->rect().y(),
                      side, side);
  painter.setWindow(-50, -50, 100, 100);
  draw(&painter);
  bitBlt(this, event->rect().topLeft(), &pixmap);
}
```

На этот раз все рисование производится в буфере. Сначала устанавливается размер будущего изображения, в соответствии с размером области, которую необходимо перерисовать. Затем настраиваются область просмотра и окно таким образом, что сам процесс рисования проходит точно так же, как и раньше. Благодаря этому нам не надо вносить изменения в функцию **draw()**. В завершение обработки события "**paint**", готовый буфер переносится на поверхность виджета, функцией **bitBlt()**.

Очень похоже на то, что мы описывали в разделе **Двойная буферизация**, но с одним важным отличием: в Главе 5, для выполнения сдвига, мы пользовались функцией **translate()**, теперь же, мы вычитаем координаты левого верхнего угла прямоугольника, требующего перерисовки, при настройке области просмотра. Использование **translate()** здесь было бы не очень удобным, поскольку преобразование должно быть выражено в логических координатах, в то время как событие поставляется с координатами физическими.

8.2 Рисование средствами QCanvas

QCanvas (Canvas -- холст, полотно, канва. прим. перев.) предоставляет более высокоуровневый интерфейс, чем **QPainter**. Он может включать в себя элементы любой формы и имеет внутреннюю реализацию двойной буферизации. Для приложений, которые занимаются визуализацией информации или двухмерных игр, выбор **QCanvas** может оказаться лучшим решением.

Элементы, которые может отображать **QCanvas**, являются экземплярами класса **QCanvasItem** или его потомков. Qt содержит неплохой набор предопределенных графических элементов: **QCanvasLine**, **QCanvasRectangle**, **QCanvasPolygon**, **QCanvasPolygonalItem**, **QCanvasEllipse**, **QCanvasSpline**, **QCanvasSprite** и **QCanvasText**.
Классы **QCanvas** и **QCanvasItem** -- просто данные, они не имеют визуального представления. Для отображения **QCanvas** и его элементов мы должны использовать виджет **QCanvasView**. Такое разделение данных и средств их отображения, позволяет отображать один и тот же **QCanvas** в нескольких **QCanvasView**, причем каждый из них может визуализировать свою собственную часть **QCanvas**, причем с применением различных матриц преобразования.
Класс **QCanvas** оптимизирован для работы с большим количеством элементов. Когда изменяется какой либо элемент, то перерисовывается только та часть, которая действительно изменилась. В нем так же заложен эффективный алгоритм проверки на пересечение. Поэтому, **QCanvas** можно смело рассматривать как неплохую альтернативу подходам, связанным с перекрытием родительских методов **paintEvent()** и **QScrollView::drawContents()**.

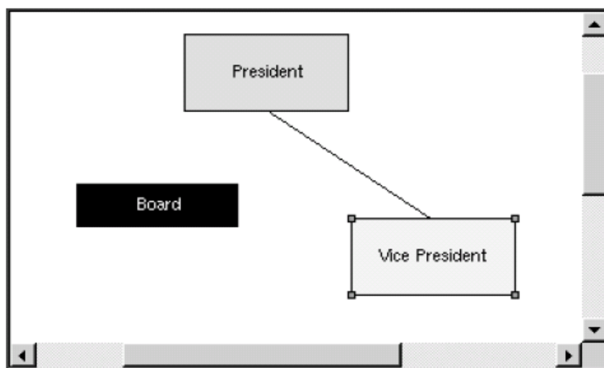


Рисунок 8.9. Внешний вид виджета DiagramView.

С целью демонстрации основных приемов работы с **QCanvas**, приведем исходный код виджета **DiagramView** -- редактора структурных диаграмм. Виджет поддерживает два типа фигур (прямоугольники и линии) и имеет контекстное меню, которое дает возможность вставить новый элемент в диаграмму, копировать элементы в буфер обмена, вставлять их из буфера обмена, удалять и изменять их свойства.

```
class DiagramView : public QCanvasView
{
  Q_OBJECT

public:
  DiagramView(QCanvas *canvas, QWidget *parent = 0, const char *name = 0);

public slots:
  void cut();
  void copy();
  void paste();
  void del();
  void properties();
  void addBox();
  void addLine();
  void bringToFront();
  void sendToBack();
```

Класс **DiagramView** порожден от класса **QCanvasView**, который в свою очередь ведет родословную от класса **QScrollView**. Он предоставляет массу публичных слотов, через которые возможно

взаимодействие с приложением. Эти слоты так же используются и самим виджетом, для обслуживания контекстного меню.

```
protected:
  void contentsContextMenuEvent(QContextMenuEvent *event);
  void contentsMousePressEvent(QMouseEvent *event);
  void contentsMouseMoveEvent(QMouseEvent *event);
  void contentsMouseDoubleClickEvent(QMouseEvent *event);

private:
  void createActions();
  void addItem(QCanvasItem *item);
  void setActiveItem(QCanvasItem *item);
  void showNewItem(QCanvasItem *item);

  QCanvasItem *pendingItem;
  QCanvasItem *activeItem;
  QPoint lastPos;
  int minZ;
  int maxZ;

  QAction *cutAct;
  QAction *copyAct;
  ...
  QAction *sendToBackAct;
};
```

Приватные и защищенные члены класса мы будем описывать очень коротко.

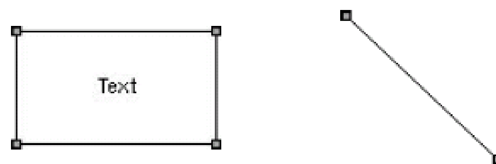


Рисунок 8.10. Элементы DiagramBox и DiagramLine.

Помимо класса **DiagramView**, нам необходимо определить два класса элеменов даграмм. Назовем эти классы как **DiagramBox** и **DiagramLine**.

```
class DiagramBox : public QCanvasRectangle
{
public:
  enum { RTTI = 1001 };

  DiagramBox(QCanvas *canvas);
  ~DiagramBox();

  void setText(const QString &newText);
  QString text() const { return str; }
  void drawShape(QPainter &painter);
  QRect boundingRect() const;
  int rtti() const { return RTTI; }

private:
  QString str;
};
```

Элемент диаграммы **DiagramBox** отображается в виде прямоугольника, с текстом внутри. Он наследует значительную часть функциональности от своего предка -- класса **QCanvasRectangle**, в который добавлена возможность рисования дополнительного текста и маленьких квадратиков по углам, для индикации активности элемента. В реальном приложении, эти квадратики можно было бы использовать для того, чтобы изменять размеры прямоугольника, но в данном случае, для упрощения примера, мы не будем этого делать.

Функция **rtti()** перекрывает родительский метод. Имя этой функции происходит от английского "run-time type identification" -- "идентификация типа во время исполнения". Возвращаемый ею результат будет сравниваться с константой RTTI, чтобы узнать -- является ли тот или иной элемент объектом класса **DiagramBox**. Эту же проверку можно было бы выполнить с использованием механизма C++ **dynamic_cast<T>()**, но это ограничило бы нас в выборе компилятора C++.
Число 1001 выбрано случайным образом. Приемлемо любое значение, большее 1000, единственное ограничение: в одном и том же приложении не должны использоваться разные классы с одинаковым значением RTTI.

```
class DiagramLine : public QCanvasLine
{
public:
  enum { RTTI = 1002 };

  DiagramLine(QCanvas *canvas);
  ~DiagramLine();

  QPoint offset() const { return QPoint((int)x(), (int)y()); }
  void drawShape(QPainter &painter);
  QPointArray areaPoints() const;
  int rtti() const { return RTTI; }
};
```

Элемент диаграммы **DiagramLine** отображается в виде линии. Этот класс наследует функциональность класса **QCanvasLine**, в который добавлена возможность отображения маленьких квадратиков на концах линии, для индикации активности элемента.

Перейдем к обзору реализации этих трех классов:

```
DiagramView::DiagramView(QCanvas *canvas, QWidget *parent,
                         const char *name)
    : QCanvasView(canvas, parent, name)
{
  pendingItem = 0;
  activeItem = 0;
  minZ = 0;
  maxZ = 0;
  createActions();
}
```

Конструктор **DiagramView** в первом аргументе получает указатель на **QCanvas** и передает его унаследованному конструктору.
В приватной функции **createActions()** создаются экземпляры **QAction**. Мы уже рассматривали подобные функции в примерах ранее, поэтому реализацию этой функции мы опустим.

```
void DiagramView::contentsContextMenuEvent(QContextMenuEvent *event)
{
  QPopupMenu contextMenu(this);
  if (activeItem) {
    cutAct->addTo(&contextMenu);
    copyAct->addTo(&contextMenu);
    deleteAct->addTo(&contextMenu);
    contextMenu.insertSeparator();
    bringToFrontAct->addTo(&contextMenu);
    sendToBackAct->addTo(&contextMenu);
    contextMenu.insertSeparator();
    propertiesAct->addTo(&contextMenu);
  } else {
    pasteAct->addTo(&contextMenu);
    contextMenu.insertSeparator();
    addBoxAct->addTo(&contextMenu);
    addLineAct->addTo(&contextMenu);
  }
  contextMenu.exec(event->globalPos());
}
```

Чтобы создать контекстное меню, мы перекрыли обработчик **contentsContextMenuEvent()** родительского класса **QScrollView**.

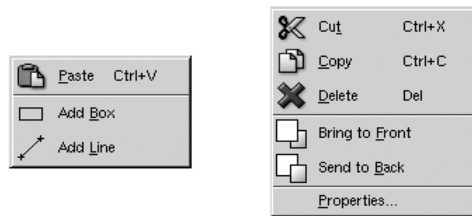


Рисунок 8.11. Контекстное меню виджета DiagramView.

Если к моменту поступления события был активизирован какой либо из элементов диаграммы, то меню будет содержать пункты, которые имеют отношение к выделенному элементу: **Cut**, **Copy**, **Delete**, **Bring to Front**, **Send to Back** и **Properties**. В противном случае меню будет состоять из трех пунктов: **Paste**, **Add Box** и **Add Line**.

```
void DiagramView::addBox()
{
  addItem(new DiagramBox(canvas()));
}

void DiagramView::addLine()
{
  addItem(new DiagramLine(canvas()));
}
```

Слоты **addBox()** и **addLine()** создают элементы диаграммы **DiagramBox** или **DiagramLine**, соответственно, которые затем добавляются в виджет, с помощью **addItem()**.

```
void DiagramView::addItem(QCanvasItem *item)
{
  delete pendingItem;
  pendingItem = item;
  setActiveItem(0);
  setCursor(crossCursor);
}
```

Приватная функция **addItem()** изменяет внешний вид указателя мыши на крестик и записывает в переменную pendingItem указатель на вновь созданный элемент. Этот элемент не будет видим на экране до тех пор, пока не будет вызван его метод **show()**.
Когда пользователь выбирает пункт контекстного меню **Add Box** или **Add Line**, изменяется внешний вид указателя мыши, но элемент будет добавлен только когда он щелкнет по канве.

```
void DiagramView::contentsMousePressEvent(QMouseEvent *event)
{
  if (event->button() == LeftButton << pendingItem) {
    pendingItem->move(event->pos().x(), event->pos().y());
    showNewItem(pendingItem);
    pendingItem = 0;
    unsetCursor();
  } else {
    QCanvasItemList items = canvas()->collisions(event->pos());
    if (items.empty())
      setActiveItem(0);
    else
      setActiveItem(*items.begin());
  }
  lastPos = event->pos();
}
```

Когда пользователь нажимает левую кнопку мыши и при этом курсор отображается в виде крестика, то вставляемый элемент диаграммы уже создан. Поэтому нам остается только вставить его в позицию курсора мыши, сделать видимым и вернуть внешний виж курсовра в первоначальное состояние.

Любой другой щелчок по канве интерпретируется как попытка выделить какой либо из элементов или наоборот, снять выделение. Функция **collisions()** возвращает список всех элементов, находящихся под указателем мыши. Первый из этого списка активизируется. Если список содержит несколько элементов, то первым в нем всегда будет стоять тот элемент, который отображается поверх других.

```
void DiagramView::contentsMouseMoveEvent(QMouseEvent *event)
{
  if (event->state() & LeftButton) {
    if (activeItem) {
      activeItem->moveBy(event->pos().x() - lastPos.x(),
                         event->pos().y() - lastPos.y());
      lastPos = event->pos();
      canvas()->update();
    }
  }
}
```

Пользоватeль может перемещать элементы диаграммы, удерживая их левой кнопкой мыши. Каждый раз, когда виджет получает событие, извещающее о перемещении мыши, мы сдвигаем элемент по горизонтали и вертикали, на полученные расстояния и вызываем **update()** канвы. Всякий раз, когда изменяется содержимое канвы, мы должны вызывать метод **update()**, чтобы перерисовать виджет.

```
void DiagramView::contentsMouseDoubleClickEvent(QMouseEvent *event)
{
  if (event->button() == LeftButton && activeItem
          && activeItem->rtti() == DiagramBox::RTTI) {
    DiagramBox *box = (DiagramBox *)activeItem;
    bool ok;

    QString newText = QInputDialog::getText(
            tr("Diagram"), tr("Enter new text:"),
            QLineEdit::Normal, box->text(), &ok, this);
    if (ok) {
      box->setText(newText);
      canvas()->update();
    }
  }
}
```

Когда пользователь выполняет двойной щелчок по элементу диаграммы, вызывается функция **rtti()**, а полученное от нее значение сравнивается с **DiagramBox::RTTI (1001)**.

Рисунок 8.12. Диалог изменения текста в элементе DiagramBox.

Если это действительно **DiagramBox**, то запускается **QInputDialog**, что позволяет пользователю изменить текст, отображаемый внутри прямоугольника.

```
void DiagramView::bringToFront()
{
  if (activeItem) {
    ++maxZ;
    activeItem->setZ(maxZ);
    canvas()->update();
  }
}
```

Слот **bringToFront()** перемещает выбранный элемент поверх других элементов диаграммы. Это достигается за счет записи значания, в координату z компонента, большего, чем у других. Если на

канве, в тех же самых координатах, находятся два или более компонентов, то тот, который имеет большее значение координаты z будет отображаться поверх остальных.

```
void DiagramView::sendToBack()
{
  if (activeItem) {
    --minZ;
    activeItem->setZ(minZ);
    canvas()->update();
  }
}
```

Слот **sendToBack()** перемещает выбранный элемент ниже других. Это достигается за счет записи значания, в координату z компонента, меньшего, чем у других.

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

Реализация слота **cut()** достаточно проста, и мы не будем его подробно описывать.

```
void DiagramView::copy()
{
  if (activeItem) {
    QString str;

    if (activeItem->rtti() == DiagramBox::RTTI) {
      DiagramBox *box = (DiagramBox *)activeItem;
      str = QString("DiagramBox %1 %2 %3 %4 %5")
            .arg(box->width())
            .arg(box->height())
            .arg(box->pen().color().name())
            .arg(box->brush().color().name())
            .arg(box->text());
    } else if (activeItem->rtti() == DiagramLine::RTTI) {
      DiagramLine *line = (DiagramLine *)activeItem;
      QPoint delta = line->endPoint() - line->startPoint();
      str = QString("DiagramLine %1 %2 %3")
            .arg(delta.x())
            .arg(delta.y())
            .arg(line->pen().color().name());
    }
    QApplication::clipboard()->setText(str);
  }
}
```

Слот **copy()** преобразует информацию об элементе в строку и копирует ее в буфер обмена. Строка содержит все необходимые сведения, чтобы потом можно было опять воссоздать элемент. Например, прямоугольник черного цвета, с текстом "My Left Foot" белого цвета, будет представлен в виде строки:

```
      DiagramBox 320 40 #000000 #ffffff My Left Foot
```

Нет необходимости беспокоиться о сохранении координат элемента. Когда элемент вынимается из буфера обмена, он просто вставляется в левый верхний угол канвы. Представление объекта в виде строки -- это самый простой способ добавить поддержку буфера обмена. Безусловно, буфер обмена может хранить и двоичные данные в произвольном формате, но об этом мы поговорим в Главе 9.

```
void DiagramView::paste()
{
  QString str = QApplication::clipboard()->text();
  QTextIStream in(&str);
  QString tag;

  in >> tag;
```

```
  if (tag == "DiagramBox") {
    int width;
    int height;
    QString lineColor;
    QString fillColor;
    QString text;

    in >> width >> height >> lineColor >> fillColor;
    text = in.read();

    DiagramBox *box = new DiagramBox(canvas());
    box->move(20, 20);
    box->setSize(width, height);
    box->setText(text);
    box->setPen(QColor(lineColor));
    box->setBrush(QColor(fillColor));
    showNewItem(box);
  } else if (tag == "DiagramLine") {
    int deltaX;
    int deltaY;
    QString lineColor;

    in >> deltaX >> deltaY >> lineColor;

    DiagramLine *line = new DiagramLine(canvas());
    line->move(20, 20);
    line->setPoints(0, 0, deltaX, deltaY);
    line->setPen(QColor(lineColor));
    showNewItem(line);
  }
}
```

Слот **paste()** пользуется услугами **QTextIStream**, для разбора содержимого строки из буфера обмена. **QTextIStream** отделяет поля в строке по символу пробела, точно так же, как и cin. Поля считываются оператором "**>>**", за исключением последнего, которое может содержать пробелы. Чтобы прочитать последнее поле используется метод **QTextStream::read()**, который возвращает остаток строки.

```
void DiagramView::del()
{
  if (activeItem) {
    QCanvasItem *item = activeItem;
    setActiveItem(0);
    delete item;
    canvas()->update();
  }
}
```

Слот **del()** удаляет активный элемент и перерисовывает канву.

```
void DiagramView::properties()
{
  if (activeItem) {
    PropertiesDialog dialog;
    dialog.exec(activeItem);
  }
}
```

Слот **properties()** запускает диалог изменения свойств активного элемента. Класс **PropertiesDialog** получает только указатель на элемент, и сам определяет -- какого типа элемент он получил, после чего выполняет все необходимые действия.

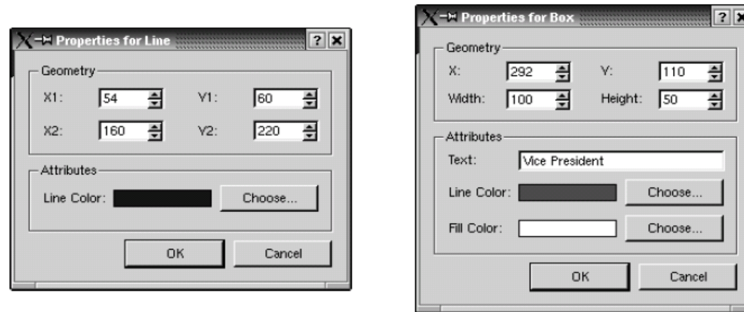


Рисунок 8.13. Два варианта отображения диалога PropertiesDialog.

Файлы **.ui** и **.ui.h** вы найдете на CD, сопровождающем книгу.

```
void DiagramView::showNewItem(QCanvasItem *item)
{
  setActiveItem(item);
  bringToFront();
  item->show();
  canvas()->update();
}
```

Функция **showNewItem()** активизирует элемент диаграммы и делает его видимым.

```
void DiagramView::setActiveItem(QCanvasItem *item)
{
  if (item != activeItem) {
    if (activeItem)
      activeItem->setActive(false);
    activeItem = item;
    if (activeItem)
      activeItem->setActive(true);
    canvas()->update();
  }
}
```

Последняя функция **setActiveItem()** сбрасывает признак активности у предыдущего активного элемента, запоминает указатель на новый активный элемент и активизирует его. Признак активности элемента хранится в классе **QCanvasItem**. Qt не использует его, но предоставляет такую возможность для удобства разработчика. Мы используем этот признак, поскольку в нашем случае активные элементы рисуются несколько иначе, чем неактивные.
Перейдем к рассмотрению реализации классов **DiagramBox** и **DiagramLine**.

```
const int Margin = 2;

void drawActiveHandle(QPainter &painter, const QPoint &center)
{
  painter.setPen(Qt::black);
  painter.setBrush(Qt::gray);
  painter.drawRect(center.x() - Margin, center.y() - Margin,
                   2 * Margin + 1, 2 * Margin + 1);
}
```

Функция **drawActiveHandle()** рисует маленькие квадратики, для индикации активности элемента диаграммы.

```
DiagramBox::DiagramBox(QCanvas *canvas)
    : QCanvasRectangle(canvas)
{
  setSize(100, 60);
  setPen(black);
  setBrush(white);
  str = "Text";
}
```

В конструкторе задаются начальные размеры прямоугольника 100 X 60, цвет пера (черный) и цвет кисти (белый). Цветом пера отображаются границы прямоугольника и текст, цветом кисти заливается внутреннее пространство прямоугольника.

```
DiagramBox::~DiagramBox()
{
  hide();
}
```

Деструктор скрывает элемент диаграммы, вызовом метода **hide()**. Это необходимо для любых классов, порожденных от **QCanvasPolygonalItem** (базовый класс для **QCanvasRectangle**).

```
void DiagramBox::setText(const QString &newText)
{
  str = newText;
  update();
}
```

Функция **setText()** записывает текст, который должен отображаться в прямоугольнике, и вызывает **QCanvasItem::update()**, чтобы отобразить изменения на экране.

```
void DiagramBox::drawShape(QPainter &painter)
{
  QCanvasRectangle::drawShape(painter);
  painter.drawText(rect(), AlignCenter, text());
  if (isActive()) {
    drawActiveHandle(painter, rect().topLeft());
    drawActiveHandle(painter, rect().topRight());
    drawActiveHandle(painter, rect().bottomLeft());
    drawActiveHandle(painter, rect().bottomRight());
  }
}
```

Функция **drawShape()** перекрывает метод класса **QCanvasPolygonalItem**, чтобы нарисовать текст и маленькие квадратики по углам, если данный элемент диаграммы активен. Сам прямоугольник рисуется родительским методом.

```
QRect DiagramBox::boundingRect() const
{
  return QRect((int)x() - Margin, (int)y() - Margin,
               width() + 2 * Margin, height() + 2 * Margin);
}
```

Функция **boundingRect()** перекрывает метод класса **QCanvasItem**. Она вызывается классом **QCanvas**, для проверки наложения одних элементов на другие и оптимизации перерисовки. Возвращаемые размеры должны быть не меньше тех, которые получает **drawShape()**.
Значение, которое возвращает родительский метод, нас не устраивает потому, что он не учитывает размеры маленьких квадратиков, рисуемых по углам активного элемента.

```
DiagramLine::DiagramLine(QCanvas *canvas)
    : QCanvasLine(canvas)
{
  setPoints(0, 0, 0, 99);
}
```

Конструктор **DiagramLine** задает координаты точек, между которыми будет нарисована линия: (0, 0) и (0, 99). В результате получается вертикальная линия, длиной в 100 пикселей.

```
DiagramLine::~DiagramLine()
{
  hide();
}
```

Опять же, в деструкторе необходимо скрыть элемент.

```
void DiagramLine::drawShape(QPainter &painter)
{
  QCanvasLine::drawShape(painter);
  if (isActive()) {
    drawActiveHandle(painter, startPoint() + offset());
    drawActiveHandle(painter, endPoint() + offset());
  }
}
```

Функция **drawShape()** перекрывает родительский метод, чтобы нарисовать маленькие квадратики на концах линии, если элемент активен. Сама линия рисуется средствами родительского класса. Реализация функции **offset()** находится внутри определения класса **DiagramLine**. Она возвращает положение элемента на канве.

```
QPointArray DiagramLine::areaPoints() const
{
  const int Extra = Margin + 1;
  QPointArray points(6);
  QPoint pointA = startPoint() + offset();
  QPoint pointB = endPoint() + offset();

  if (pointA.x() > pointB.x())
    swap(pointA, pointB);

  points[0] = pointA + QPoint(-Extra, -Extra);
  points[1] = pointA + QPoint(-Extra, +Extra);
  points[3] = pointB + QPoint(+Extra, +Extra);
  points[4] = pointB + QPoint(+Extra, -Extra);
  if (pointA.y() > pointB.y()) {
    points[2] = pointA + QPoint(+Extra, +Extra);
    points[5] = pointB + QPoint(-Extra, -Extra);
  } else {
    points[2] = pointB + QPoint(-Extra, +Extra);
    points[5] = pointA + QPoint(+Extra, -Extra);
  }
  return points;
}
```

Функция **areaPoints()** играет роль, аналогичную **boundingRect()** класса **DiagramBox**. Аппроксимация области, принадлежащей диагональной линии, прямоугольником будет слишком грубым приближением. Потому необходимо перекрыть родительский метод и вернуть более точные границы области рисования элемента. В принципе, реализация метода в классе **QCanvasLine** уже возвращает приемлемые границы, но она не учитывает маленькие квадратики, которые рисуются у активных элементов.

Первое, что делает функция -- сохраняет координаты точек во временных переменных **pointA** и **pointB**, а затем проверяет -- находится ли точка **pointA** левее точки **pointB** и меняет их местами, если это необходимо, с помощью функции **swap()** (определена в <**algorithm**>). После этого она выполняет различные действия для ниспадающих и восстающих линий.

Границы области рисования линии всегда представляются в виде 6 точек, но их координаты существенно зависят от того -- ниспадающая линия или восстающая. Однако, координаты 4-х точек из 6-ти (0, 1, 3 и 4) всегда одинаковы для обоих случаев. Например, точки 0 и 1 всегда определяют левый верхний и левый нижний углы конца A, а точка 2 задает правый нижний угол для восстающих линий на конце A и левый нижний угол для ниспадающих линий на конце B.

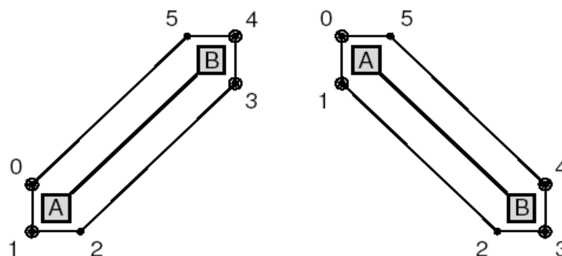


Рисунок 8.14. Границы области рисования линий DiagramLine.

При рассмотрении кода, который мы написали, вы наверняка заметили, что виджет **DiagramView** реализует достаточно большой объем функциональности, отвечающей за выделение элементов и их перемещение по канве, а так же предоставляет контекстное меню для взаимодействия с пользователем.

Одна деталь, которую мы опустили -- пользователь не может изменить размеры элемента, манипулируя маленькими квадратиками. Если бы мы хотели добавить такую возможность, то скорее всего нам пришлось бы сделать все немного иначе. Вместо того, чтобы рисовать квадратики в **drawShape()**, нам скорее всего пришлось бы сделать их самостоятельными элементами канвы. И изменять внешний вид указателя мыши, вызовом **setCursor()**, когда он находится над квадратиком, но для этого, сначала потребовалось бы вызвать **setMouseTracking(true)**, потому что обычно Qt передает события перемещения мыши только тогда, когда какая либо кнопка мыши удерживается в нажатом состоянии.

Кроме того, можно было бы расширить набор элементов диаграмм, сделать возможным выделение нескольких элементов диаграммы одновременно и добавить возможность объединения элементов в группы. Статья "Canvas Item Groupies", в ежеквартальнике Qt Quarterly (http://doc.trolltech.com/qq/qq05-canvasitemgrouping.html), описывает один из приемов реализации подобных возможностей.

В этом разделе мы предоставили пример работающего кода, использующего функциональность классов **QCanvas** и **QCanvasView**, но не раскрыли всех возможностей класса **QCanvas**. Например, элементы могут перемещаться по канве, если им указать скорость перемещения вызовом метода **setVelocity()**. За подробной информацией обращайтесь к сопроводительной документации.

8.3 Вывод на печать

Процедура вывода изображений на печать в Qt очень похожа на рисование по поверхности виджетов. Вкратце, процесс печати можно представить следующими шагами:

- Создается экземпляр класса **QPrinter**, который будет представлять "устройство для рисования".
- Вызывается функция **QPrinter::setup()**, которая покажет пользователю диалог выбора принтера.
- Создается экземпляр класса **QPainter**, который будет взаимодействовать с объектом **QPrinter**.
- Средствами **QPainter** рисуется изображение на странице.
- Вызывается метод **QPrinter::newPage()**, чтобы прокрутить страницу.
- Повторять действия, описанные в пунктах 4 и 5, пока не будут отпечатаны все страницы.

В операционных системах Windows и Mac OS X, **QPrinter** использует системные драйверы. В Unix страницы генерируются в формате PostScript и затем передаются устройству печати **lp** или **lpr** (или любой другой программе, которая будет назначена вызовом **QPrinter::setPrintProgram()**).

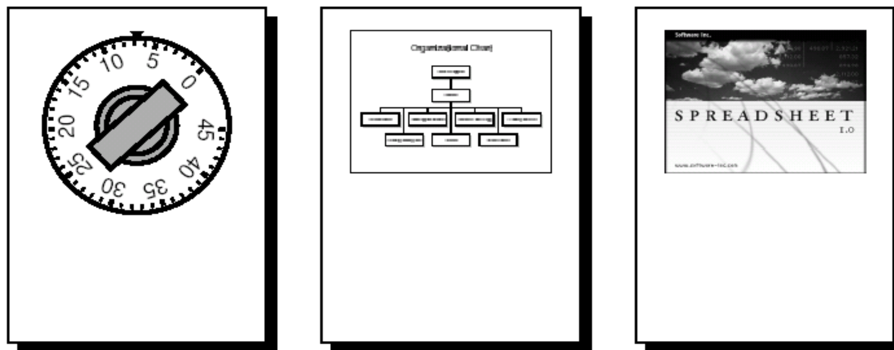


Рисунок 8.15. Пример вывода на печать виджетов OvenTimer, QCanvas и QImage.

Начнем обсуждение с простого примера, который печатает одну страницу. Для начала напечатаем виджет **OvenTimer**:

```
void PrintWindow::printOvenTimer(OvenTimer *ovenTimer)
{
```

```
  if (printer.setup(this)) {
    QPainter painter(&printer);
    QRect rect = painter.viewport();
    int side = QMIN(rect.width(),
    rect.height());
    painter.setViewport(0, 0, side, side);
    painter.setWindow(-50, -50, 100, 100);
    ovenTimer->draw(&painter);
  }
}
```

Здесь мы исходим из того, что класс **PrintWindow** содержит переменную-член **printer**, класса **QPrinter**. В противном случае можно было бы создать экземпляр **QPrinter** на стеке, но в этом случае у нас отсутствовала бы возможность сохранить пользовательские настройки принтера.
Мы вызываем **setup()**, чтобы запустить диалог выбора принтера. Она возвращает **true**, если пользователь нажал на кнопку **OK**. После вызова **setup()**, объект **QPrinter** готов к работе.

Далее создается **QPainter**, который будет рисовать на **QPrinter**. Потом настраивается область просмотра (**viewport**) и назначается система координат окна (-50, -50, 100, 100) -- прямоугольник, который ожидает получить **OvenTimer**, и в завершение выполняется рисование виджета, вызовом функции **draw()**. Если не установить размеры области просмотра, то виджет **OvenTimer** будет вытянут на всю высоту страницы.

По-умолчанию **QPainter** устанавливает размеры окна такими, чтобы они соответствовали разрешению экрана (обычно где-то между 72 и 100 точками на дюйм), но в данном случае это не имеет большого значения, так как мы сами установили систему координат окна.
Пример вывода на печать виджета **OvenTimer** не имеет особой практической ценности, потому что он предназначен, в первую очередь, для вывода на экран и взаимодействия с пользователем. Но для других виджетов, таких как **Plotter**, который был разработан нами в Главе 5, этот пример приобретает определенный смысл.
Более практичный пример -- вывод на печать **QCanvas**. Приложения, которые его используют, очень часто нуждаются в возможности вывода на печать того, что нарисует пользователь.

```
void PrintWindow::printCanvas(QCanvas *canvas)
{
  if (printer.setup(this)) {
    QPainter painter(&printer);
    QRect rect = painter.viewport();
    QSize size = canvas->size();
    size.scale(rect.size(), QSize::ScaleMin);
    painter.setViewport(rect.x(), rect.y(),
                        size.width(), size.height());
    painter.setWindow(canvas->rect());
    painter.drawRect(painter.window());
    painter.setClipRect(painter.viewport());

    QCanvasItemList items = canvas->collisions(canvas->rect());
    QCanvasItemList::const_iterator it = items.end();
    while (it != items.begin()) {
      --it;
      (*it)->draw(painter);
    }
  }
}
```

На этот раз мы установили систему координат окна в соответствии с размерами канвы и ограничили область просмотра тем же самым соотношением сторон. Для этого мы использовали функцию **QSize::scale()**, задав в качестве второго аргумента **ScaleMin**. Например, если канва имела размер 640 X 480, а область просмотра **QPainter** -- 5000 X 5000, в результате получится область просмотра с размерами 5000 X 3750.
Функция **collisions()** возвратит список видимых элементов канвы, отсортированный по значению координаты z. Список просматривается в цикле, начиная с конца, и выполняется рисование элементов списка вызовом **QCanvasItem::draw()**. Таким образом, чем выше в списке стоит элемент, тем позднее он будет нарисован.

Третий пример -- печать картинки из **QImage**.

```
void PrintWindow::printImage(const QImage &image)
{
  if (printer.setup(this)) {
    QPainter painter(&printer);
    QRect rect = painter.viewport();
    QSize size = image.size();
    size.scale(rect.size(), QSize::ScaleMin);
    painter.setViewport(rect.x(), rect.y(),
                        size.width(), size.height());
    painter.setWindow(image.rect());
    painter.drawImage(0, 0, image);
  }
}
```

Мы установили размеры окна в соответствии с размерами изображения и размеры области просмотра (viewport), чтобы соблюсти отношения сторон, после чего нарисовали изображение, начиная с позиции (0, 0).
Печать компонентов, которые занимают не более одной страницы, достаточно проста. Но нередко приходится сталкиваться с необходимостью вывода на печать многостраничных документов. В таких случаях нужно вывести на печать одну страницу, затем вызвать функцию **newPage()** и напечатать следующую страницу. Однако здесь возникает проблема определения окончания каждой из страниц.
Qt предлагает два варианта вывода на печать многостраничных документов:

- Можно "перегнать" документ в формат HTML и вывести его средствами **QSimpleRichText**.
- Можно выполнять перевод страниц вручную.

Далее мы рассмотрим оба варианта.

В качестве примера напечатаем справочник цветовода, который содержит названия цветов и их краткое описание. Каждая статья справочника хранится в виде "название: описание", например:

```
Miltonopsis santanae: Самая опасная разновидность орхидеи.
```

Поскольку каждая статья представлена одной строкой, то весь справочник можно представить как список строк -- **QStringList**.
Следующий фрагмент кода выводит на печать содержимое справочника, предварительно "перегнав" его в формат HTML:

```
void PrintWindow::printFlowerGuide(const QStringList &entries)
{
  QString str;
  QStringList::const_iterator it = entries.begin();
  while (it != entries.end()) {
    QStringList fields = QStringList::split(": ", *it);
    QString title = QStyleSheet::escape(fields[0]);
    QString body = QStyleSheet::escape(fields[1]);

    str += "<table width=\"100%\" border=1 cellspacing=0>\n"
           "<tr><td bgcolor=\"lightgray\"><font size=\"+1\">"
           "<b><i>" + title + "</i></b></font>\n<tr><td>"
           + body + "\n</table>\n<br>\n";
    ++it;
  }
  printRichText(str);
}
```

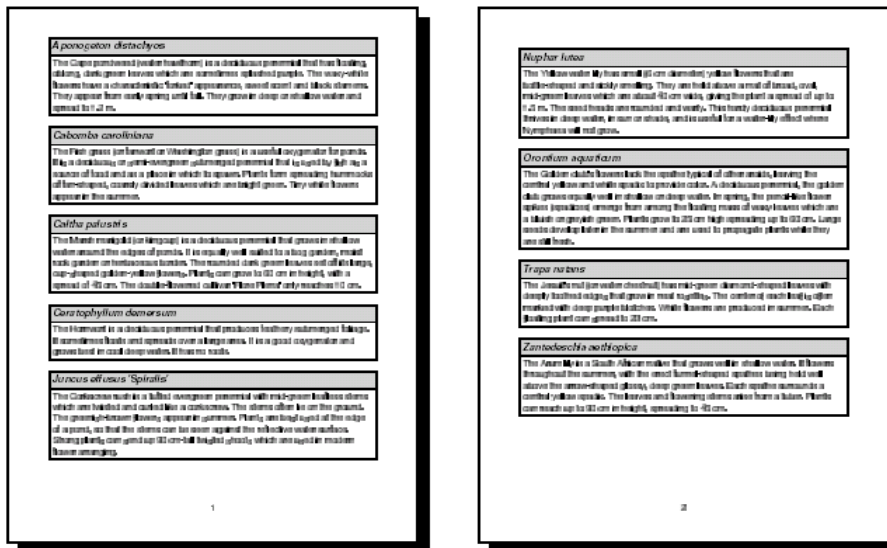


Рисунок 8.16. Пример вывода на печать справочника цветовода, с помощью QSimpleRichText.

На первом шаге выполняется преобразование справочника в формат HTML. Каждая статья представляется в виде HTML таблицы с двумя ячейками. Функция **QStyleSheet::escape()** заменяет специальные символы '**&**', '**<**', '**>**' их аналогами HTML ("**&**", "**<**", "**>**"). И в заключение выводим на печать то, что получилось, вызовом **printRichText()**.

```
const int LargeGap = 48;

void PrintWindow::printRichText(const QString &str)
{
  if (printer.setup(this)) {
    QPainter painter(&printer);
    int pageHeight = painter.window().height() - 2 * LargeGap;
    QSimpleRichText richText(str, bodyFont, "", 0, 0,
                             pageHeight);
    richText.setWidth(&painter, painter.window().width());
    int numPages = (int)ceil((double)richText.height()
                             / pageHeight);
    int index;

    for (int i = 0; i < (int)printer.numCopies(); ++i) {
      for (int j = 0; j < numPages; ++j) {
        if (i > 0 || j > 0)
          printer.newPage();
        if (printer.pageOrder()
               == QPrinter::LastPageFirst) {
          index = numPages - j - 1;
        } else {
          index = j;
        }
        printPage(&painter, richText, pageHeight, index);
      }
    }
  }
}
```

Сначала мы рассчитываем высоту одной страницы, отталкиваясь от размера окна и размера пространства, которое резервируется под нижний и верхний колонтитулы. Затем создается объект класса **QSimpleRichText**, содержащий HTML текст. Последний аргумент, в конструкторе **QSimpleRichText** -- это высота страницы. Класс **QSimpleRichText** использует эту величину, чтобы вставить разрывы страниц.

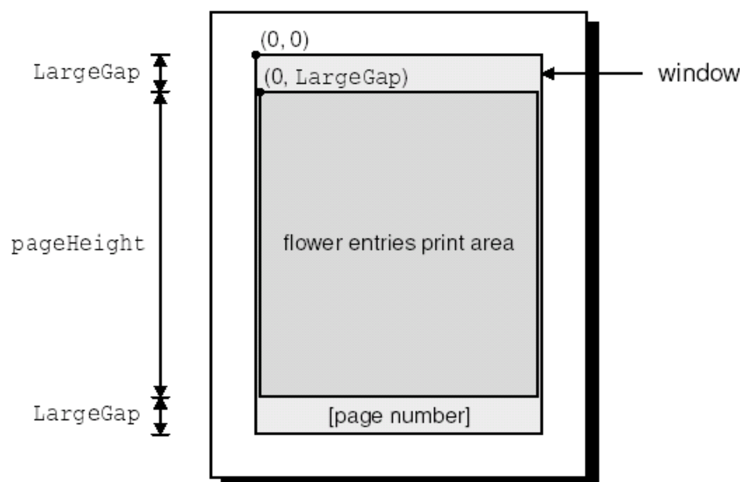


Рисунок 8.17. Раскладка страницы справочника цветовода.

После этого выполняется печать каждой страницы. Внешний цикл for отсчитывает количество копий, запрошенных пользователем. В большинстве своем, драйверы принтеров поддерживают печать нескольких копий документа, в этом случае функция **QPrinter::numCopies()** вернет 1, в противном случае -- количество копий, запрошенных пользователем. В предыдущих примерах, с целью упрощения кода, мы игнорировали этот параметр.
Внутренний цикл **for** отсчитывет страницы. если страница не является первой, то вызывается функция **newPage()**. Для вывода очередной страницы на печать вызывается функция **printPage()**. Диалог выбора принтера позволяет пользователю заказать печать страниц в обратном порядке, мы так же соблюдаем и это требование.
В данном примере предполагается, что **printer**, **bodyFont** и **footerFont** -- это переменные-члены класса **PrintWindow**.

```
void PrintWindow::printPage(QPainter *painter,
                            const QSimpleRichText &richText,
                            int pageHeight, int index)
{
  QRect rect(0, index * pageHeight + LargeGap,
             richText.width(), pageHeight);
  painter->saveWorldMatrix();
  painter->translate(0, -rect.y());
  richText.draw(painter, 0, LargeGap, rect, colorGroup());
  painter->restoreWorldMatrix();
  painter->setFont(footerFont);
  painter->drawText(painter->window(), AlignHCenter | AlignBottom,
                    QString::number(index + 1));
}
```

Функция **printPage()** выводит на печать (index + 1)-ую страницу. Она содержит HTML-код и номер страницы в нижнем колонтитуле.
Мы выполняем смещение системы координат и вызываем **draw()**, чтобы нарисовать текст, с нужной позиции. После этого, в нижнем колонтитуле, по центру страницы, выводится ее номер. Если бы нам потребовалось выводить что нибудь в верхнем колонтитуле, то мы добавили бы еще один вызов **drawText()**.
Константа **LargeGap** равна числу 48. Если исходить из предположения, что разрешение экрана срставляет 96 точек на дюйм, то число 48 соответствует половине дюйма (12.7 мм). Чтобы найти точное значение для константы, в каждом конкретном случае, можно воспользоваться услугами класса **QPaintDeviceMetrics**:

```
  QPaintDeviceMetrics metrics(&printer);
  int LargeGap = metrics.logicalDpiY() / 2;
```

Ниже приводится один из вариантов инициализации **bodyFont** и **footerFont** в конструкторе **PrintWindow**:

```
  bodyFont = QFont("Helvetica", 14);
```

```
  footerFont = bodyFont;
```

А теперь покажем, как напечатать справочник с помощью **QPainter**. Ниже приводится измененный вариант функции **printFlowerGuide()**:

```
void PrintWindow::printFlowerGuide(const QStringList &entries)
{
  if (printer.setup(this)) {
    QPainter painter(&printer);
    vector<QStringList> pages;
    int index;

    paginate(&painter, &pages, entries);
    for (int i = 0; i < (int)printer.numCopies(); ++i) {
      for (int j = 0; j < (int)pages.size(); ++j) {
        if (i > 0 || j > 0)
          printer.newPage();

        if (printer.pageOrder() == QPrinter::LastPageFirst) {
          index = pages.size() - j - 1;
        } else {
          index = j;
        }
        printPage(&painter, pages, index);
      }
    }
  }
}
```

Первое, что нужно сделать после настройки принтера и **QPainter** -- это вызвать вспомогательную функцию **paginate()**, чтобы определить разбивку справочника по страницам. Результат работы функции -- массив **QStringList**, в котором каждый из элементов хранит статьи справочника для одной страницы.

Например, допустим, что справочник содержит всего 6 статей, которые мы обозначим как A, B, C, D, E и F. Теперь предположим, что статьи A и B располагаются на первой странице, C, D и E -- на второй, а F -- на третьей. Таким образом, массив pages, в элементе с индексом 0, будет содержать статьи A и B, статьи C, D и E -- в элементе с индексом 1 и статью F -- в элементе с индексом 2.
В остальном, функция **printFlowerGuide()** практически идентична приведенному ранее варианту. Однако, функция **printPage()** имеет существенные отличия, но об этом немного позже.

```
void PrintWindow::paginate(QPainter *painter,
                           vector<QStringList> *pages,
                           const QStringList &entries)
{
  QStringList currentPage;
  int pageHeight = painter->window().height() - 2 * LargeGap;
  int y = 0;

  QStringList::const_iterator it = entries.begin();
  while (it != entries.end()) {
    int height = entryHeight(painter, *it);
    if (y + height > pageHeight && !currentPage.empty()) {
      pages->push_back(currentPage);
      currentPage.clear();
      y = 0;
    }
    currentPage.push_back(*it);
    y += height + MediumGap; ++it;
  }
  if (!currentPage.empty())
    pages->push_back(currentPage);
}
```

Функция **paginate()** распределяет статьи справочника по страницам, основываясь на результатах функции **entryHeight()**, которая вычисляет высоту одной статьи.

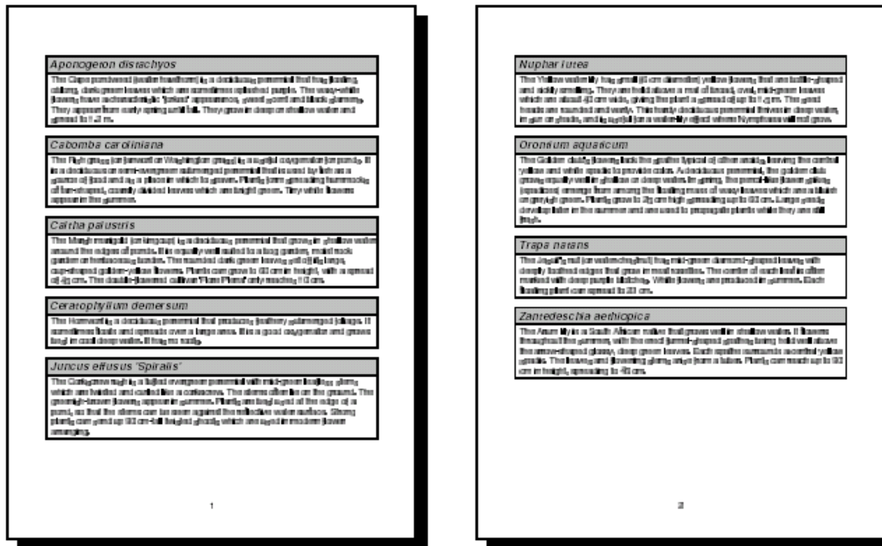


Рисунок 8.18. Вывод справочника цветовода с помощью QPainter.

Она в цикле проходит по всем статьям справочника и добавляет их в конец текущей страницы пока не закончится доступное пространство. После этого текущая страница добавляется в массив pages и начинается заполнение новой страницы.

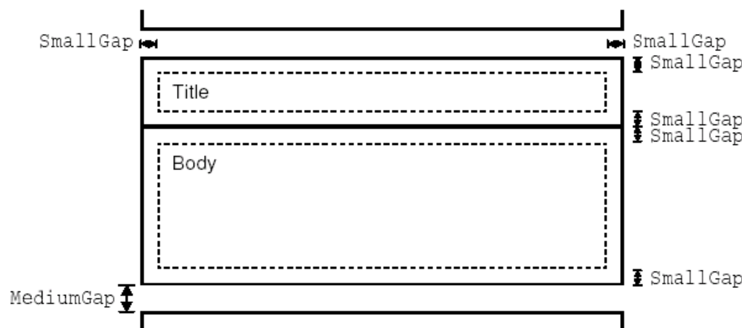


Рисунок 8.19. Раскладка одной статьи справочника.

```
int PrintWindow::entryHeight(QPainter *painter, const QString &entry) {
  QStringList fields = QStringList::split(": ", entry);
  QString title = fields[0];
  QString body = fields[1];
  int textWidth = painter->window().width() - 2 * SmallGap;
  int maxHeight = painter->window().height();
  painter->setFont(titleFont);
  QRect titleRect = painter->boundingRect(0, 0,
                                          textWidth, maxHeight,
                                          WordBreak, title);
  painter->setFont(bodyFont);
  QRect bodyRect = painter->boundingRect(0, 0,
                                         textWidth, maxHeight,
                                         WordBreak, body);
  return titleRect.height() + bodyRect.height() + 4 * SmallGap;
}
```

Функция **entryHeight()**, с помощью **QPainter::boundingRect()**, вычисляет высоту статьи на странице. На рисунке 8.19 показана раскладка статьи справочника и назначение констант **SmallGap** и **MediumGap**.

```
void PrintWindow::printPage(QPainter *painter,
                            const vector<QStringList> &pages,
                            int index)
{
  painter->saveWorldMatrix();
  painter->translate(0, LargeGap);
  QStringList::const_iterator it = pages[index].begin();
  while (it != pages[index].end()) {
```

```
    QStringList fields = QStringList::split(": ", *it);
    QString title = fields[0];
    QString body = fields[1];
    printBox(painter, titleFont, title, lightGray);
    printBox(painter, bodyFont, body, white);
    painter->translate(0, MediumGap);
    ++it;
  }
  painter->restoreWorldMatrix();
  painter->setFont(footerFont);
  painter->drawText(painter->window(), AlignHCenter | AlignBottom,
                    QString::number(index + 1));
}
```

Функция **printPage()** обходит в цикле все статьи справочника и печатает их в два приема: первый раз функция **printBox()** вызывается для печати заголовка статьи (название цветка) и второй раз -- для печати описания (тела статьи). В заключение печатается номер страницы, внизу по центру.

```
void PrintWindow::printBox(QPainter *painter, const QFont &font,
                           const QString &str, const QBrush &brush)
{
  painter->setFont(font);
  int boxWidth = painter->window().width();
  int textWidth = boxWidth - 2 * SmallGap;
  int maxHeight = painter->window().height();

  QRect textRect = painter->boundingRect(SmallGap, SmallGap,
                                         textWidth, maxHeight,
                                         WordBreak, str);
  int boxHeight = textRect.height() + 2 * SmallGap;
  painter->setPen(QPen(black, 2, SolidLine));
  painter->setBrush(brush);
  painter->drawRect(0, 0, boxWidth, boxHeight);
  painter->drawText(textRect, WordBreak, str);
  painter->translate(0, boxHeight);
}
```

Функция **printBox()** рисует прямоугольник, а затем внутри него -- текст.
Если на печать выводится большой документ, или пользователь заказал несколько копий одного документа, то неплохо было бы показать индикатор хода выполнения задания -- **QProgressDialog**. Ниже приводится модифицированный вариант функции **printFlowerGuide()**, которая выводит перед пользователем индикатор хода выполнения задания:

```
void PrintWindow::printFlowerGuide(const QStringList &entries)
{
  if (printer.setup(this)) {
    QPainter painter(&printer);
    vector<QStringList> pages;
    int index;

    paginate(&painter, &pages, entries);

    int numSteps = printer.numCopies() * pages.size();
    int step = 0;
    QProgressDialog progress(tr("Printing file..."),
                             tr("Cancel"), numSteps, this);
    progress.setModal(true);

    for (int i = 0; i < (int)printer.numCopies(); ++i) {
      for (int j = 0; j < (int)pages.size(); ++j) {
        progress.setProgress(step);
        qApp->processEvents();
        if (progress.wasCanceled()) {
          printer.abort();
          return;
        }
        ++step;

        if (i > 0 || j > 0)
```

```
          printer.newPage();
        if (printer.pageOrder() == QPrinter::LastPageFirst) {
          index = pages.size() - j - 1;
        } else {
          index = j;
        }
        printPage(&painter, pages, index);
      }
    }
  }
}
```

Когда пользователь нажимает на кнопку **Cancel** -- вызывается **QPrinter::abort()**, которая останавливает процесс печати.

8.4 Графика OpenGL

OpenGL -- это стандарт API, для отображения двух- и трехмерной графики. Приложения Qt могут использовать OpenGL, посредством модуля **QGL**. Мы полагаем, что вы уже имеете некоторое знакомство с OpenGL. Если это не так, то рекомендуем начать изучение с посещения сайта *http://www.opengl.org/*.
Рисование трехмерных объектов, с помощью OpenGL, не так сложно, как может показаться на первый взгляд. Все что вам нужно сделать -- создать дочерний класс от **QGLWidget**, перекрыть некоторые виртуальные методы предка и связать приложение с модулем **QGL** и библиотекой OpenGL. Поскольку **QGLWidget** ведет свою родословную от **QWidget**, то здесь вполне применимы знания, которые вы уже получили. Основное отличие здесь состоит в том, что теперь, вместо **QPainter**, вам придется использовать стандартные функции рисования из OpenGL.

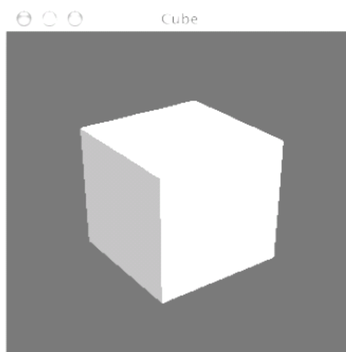


Рисунок 8.20. Приложение Cube.

Для демонстрации возможностей библиотеки OpenGL, напишем приложение **Cube**, изображенное на рисунке 8.20. Приложение рисует трехмерный куб, грани которого окрашены в различные цвета. Пользователь может вращать куб и перемещать его. Двойным щелчком мыши по грани куба, он сможет изменить ее цвет, с помощью диалога выбора цвета **QColorDialog**.

```
class Cube : public QGLWidget
{
public:
  Cube(QWidget *parent = 0, const char *name = 0);

protected:
  void initializeGL();
  void resizeGL(int width, int height);
  void paintGL();
  void mousePressEvent(QMouseEvent *event);
  void mouseMoveEvent(QMouseEvent *event);
  void mouseDoubleClickEvent(QMouseEvent *event);

private:
  void draw();
  int faceAtPosition(const QPoint &pos);
  GLfloat rotationX;
```

```
  GLfloat rotationY;
  GLfloat rotationZ;
  QColor faceColors[6];
  QPoint lastPos;
};
```

Класс **Cube** порожден от **QGLWidget**. Функции **initializeGL()**, **resizeGL()** и **paintGL()** перекрывают методы родительского класса **QGLWidget**. Обработчики событий от мыши перекрывают обработчики, унаследованные от **QWidget**. Определение класса **QGLWidget** находится в заголовке <**qgl.h**>.

```
Cube::Cube(QWidget *parent, const char *name)
    : QGLWidget(parent, name)
{
  setFormat(QGLFormat(DoubleBuffer | DepthBuffer));
  rotationX = 0;
  rotationY = 0;
  rotationZ = 0;
  faceColors[0] = red;
  faceColors[1] = green;
  faceColors[2] = blue;
  faceColors[3] = cyan;
  faceColors[4] = yellow;
  faceColors[5] = magenta;
}
```

В конструкторе вызывается **QGLWidget::setFormat()**, чтобы задать контекст устройства отображения OpenGL, и инициализируются приватные переменные-члены класса.

```
void Cube::initializeGL()
{
  qglClearColor(black);
  glShadeModel(GL_FLAT);
  glEnable(GL_DEPTH_TEST);
  glEnable(GL_CULL_FACE);
}
```

Функция **initializeGL()** вызывается один раз, перед вызовом **paintGL()**. Здесь выполняется настройка контекста отображения.
Все функции являются стандартными вызовами из библиотеки OpenGL, за исключением **qglClearColor()** -- метода класса **QGLWidget**. Если задаться целью, до конца следовать стандарту OpenGL, то мы могли бы вызвать функцию **glClearColor()**, в режиме RGBA, или **glClearIndex()**, в режиме индексированных цветов.

```
void Cube::resizeGL(int width, int height)
{
  glViewport(0, 0, width, height);
  glMatrixMode(GL_PROJECTION);
  glLoadIdentity();
  GLfloat x = (GLfloat)width / height;
  glFrustum(-x, x, -1.0, 1.0, 4.0, 15.0);
  glMatrixMode(GL_MODELVIEW);
}
```

Функция **resizeGL()** вызывается один раз, перед **paintGL()**, но после того, как будет вызвана функция **initializeGL()**. Здесь настраивается область просмотра (viewport), проекция и прочие настройки, которые зависят от размера виджета.

```
void Cube::paintGL()
{
  glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
  draw();
}
```

Функция **paintGL()** вызывается всякий раз, когда возникает необходимость перерисовать содержимое виджета. Она напоминает обработчик события **QWidget::paintEvent()**, только вместо **QPainter** здесь

используются обращения к функциям OpenGL. Собственно рисование выполняется внутри приватной функции **draw()**:

```
void Cube::draw()
{
  static const GLfloat coords[6][4][3] = {
      { { +1.0, -1.0, +1.0 }, { +1.0, -1.0, -1.0 },
        { +1.0, +1.0, -1.0 }, { +1.0, +1.0, +1.0 } },
      { { -1.0, -1.0, -1.0 }, { -1.0, -1.0, +1.0 },
        { -1.0, +1.0, +1.0 }, { -1.0, +1.0, -1.0 } },
      { { +1.0, -1.0, -1.0 }, { -1.0, -1.0, -1.0 },
        { -1.0, +1.0, -1.0 }, { +1.0, +1.0, -1.0 } },
      { { -1.0, -1.0, +1.0 }, { +1.0, -1.0, +1.0 },
        { +1.0, +1.0, +1.0 }, { -1.0, +1.0, +1.0 } },
      { { -1.0, -1.0, -1.0 }, { +1.0, -1.0, -1.0 },
        { +1.0, -1.0, +1.0 }, { -1.0, -1.0, +1.0 } },
      { { -1.0, +1.0, +1.0 }, { +1.0, +1.0, +1.0 },
        { +1.0, +1.0, -1.0 }, { -1.0, +1.0, -1.0 } }
  };

  glMatrixMode(GL_MODELVIEW);
  glLoadIdentity();
  glTranslatef(0.0, 0.0, -10.0);
  glRotatef(rotationX, 1.0, 0.0, 0.0);
  glRotatef(rotationY, 0.0, 1.0, 0.0);
  glRotatef(rotationZ, 0.0, 0.0, 1.0);

  for (int i = 0; i < 6; ++i) {
    glLoadName(i);
    glBegin(GL_QUADS);
    qglColor(faceColors[i]);
    for (int j = 0; j < 4; ++j) {
      glVertex3f(coords[i][j][0], coords[i][j][1], coords[i][j][2]);
    }
    glEnd();
  }
}
```

Внутри функции **draw()** выполняется рисование куба, с учетом вращения по осям x, y и z и цветов граней, находящихся в массиве **faceColors**. Все вызовы являются стандартными для OpenGL, за исключением **qglColor()**. Мы могли бы использовать вместо нее стандартные функции OpenGL **glColor3d()** или **glIndex()**, в зависимости от выбранного режима цветопередачи.

```
void Cube::mousePressEvent(QMouseEvent *event)
{
  lastPos = event->pos();
}

void Cube::mouseMoveEvent(QMouseEvent *event)
{
  GLfloat dx = (GLfloat)(event->x() - lastPos.x()) / width();
  GLfloat dy = (GLfloat)(event->y() - lastPos.y()) / height();

  if (event->state() & LeftButton) {
    rotationX += 180 * dy;
    rotationY += 180 * dx;
    updateGL();
  } else if (event->state() & RightButton) {
    rotationX += 180 * dy;
    rotationZ += 180 * dx;
    updateGL();
  }
  lastPos = event->pos();
}
```

Функции **mousePressEvent()** и **mouseMoveEvent()** позволяют пользователю вращать куб и перемещать его по поверхности экрана. Левой кнопкой мыши выполняется вращение по осям x и y, правой -- по осям x и z.

После изменения переменных **rotationX** и/или **rotationY** и **rotationZ**, вызывается функция **updateGL()**, которая перерисовывает изображение.

```
void Cube::mouseDoubleClickEvent(QMouseEvent *event)
{
  int face = faceAtPosition(event->pos());
  if (face != -1) {
    QColor color = QColorDialog::getColor(faceColors[face],
                                          this);
    if (color.isValid()) {
      faceColors[face] = color;
      updateGL();
    }
  }
}
```

Обработчик **mouseDoubleClickEvent()** позволяет пользователю изменить цвет грани по двойному щелчку мыши. Для определения номера грани вызывается функция **faceAtPosition()**. Если под указателем мыши действительно находится какая либо грань куба, вызывается **QColorDialog::getColor()**, чтобы получить от пользователя новый цвет грани. Затем он заносится в массив **faceColors** и вызывается **updateGL()**, чтобы перерисовать изображение.

```
int Cube::faceAtPosition(const QPoint &pos)
{
  const int MaxSize = 512;
  GLuint buffer[MaxSize];
  GLint viewport[4];

  glGetIntegerv(GL_VIEWPORT, viewport);
  glSelectBuffer(MaxSize, buffer);
  glRenderMode(GL_SELECT);
  glInitNames();
  glPushName(0);
  glMatrixMode(GL_PROJECTION);
  glPushMatrix();
  glLoadIdentity();
  gluPickMatrix((GLdouble)pos.x(),
                (GLdouble)(viewport[3] - pos.y()),
                5.0, 5.0, viewport);
  GLfloat x = (GLfloat)width() / height();
  glFrustum(-x, x, -1.0, 1.0, 4.0, 15.0);
  draw();
  glMatrixMode(GL_PROJECTION);
  glPopMatrix();

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

Функция **faceAtPosition()** возвращает либо номер грани, находящейся в заданных координатах, либо -1, если точка с заданными координатами не входит ни в одну из граней. Код, выполняющий проверку, достаточно сложен. По сути -- он переводит сцену в режим **GL_SELECT**, чтобы мы могли воспользоваться дополнительными возможностями OpenGL, и отыскивает номер грани ("**name**"). Далее приводится содержимое файла **main.cpp**:

```
#include <qapplication.h>

#include "cube.h"

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  if (!QGLFormat::hasOpenGL())
    qFatal("This system has no OpenGL support");

  Cube cube;
  cube.setCaption(QObject::tr("Cube"));
```

```
  cube.resize(300, 300);
  app.setMainWidget(&cube);
  cube.show();
  return app.exec();
}
```

Если система не поддерживает OpenGL, то, с помощью вызова **qFatal()**, приложение выводит сообщение об ошибке и завершает работу.
Чтобы связать приложение **Cube** с модулем **QGL** и библиотекой OpenGL, в файл .pro нужно добавить строчку:

```
CONFIG += opengl
```

За дополнительной информацией о модуле **QGL**, обращайтесь к сопроводительной документации по классам **QGLWidget**, **QGLFormat**, **QGLContext** и **QGLColormap**.

9 DRAG AND DROP

"Drag and Drop" (от англ. "Перетащил и бросил") -- современный интуитивно понятный способ перемещения информации внутри приложения или между приложениями. Он часто реализуется как дополнение к поддержке буфера обмена.

В этой главе мы покажем как добавить в приложение поддержку механизма "перетащил и бросил". Затем мы будем использовать код "drag and drop" для реализации поддержки буфера обмена. Это возможно по той простой причине, что в основе обоих механизмов лежит один абстрактный класс **QMimeSource**, который может хранить данные в различных форматах.

9.1 Реализация механизма 'drag and drop' в приложениях

В основе механизма "перетащил и бросил" лежат две операции: операция перетаскивания и операция сброса. Виджет может быть как источником, откуда начинается перетаскивание, так и местом, куда может производиться сбрасывание.
Это очень мощный механизм, позволяющий перетаскивать данные из одного приложения в другое. Однако, в некоторых случаях, можно реализовать некоторое подобие механизма "перетащил и бросил", не прибегая к специальным возможностям Qt. Если все, что вам нужно -- это перетащить какие либо данные внутри одного виджета, то гораздо проще это выполняется перекрытием обработчиков событий от мыши. Подобный подход мы рассматривали в Главе 8, при разработке виджета **DiagramView**.

В нашем первом примере мы рассмотрим -- как заставить Qt приложение принимать данные, перетаскиваемые из других приложений. Приложение представлено окном, где в качестве центрального, используется виджет **QTextEdit**. Когда пользователь перемещает какой либо файл с рабочего стола или из программы-обозревателя, то наше приложение будет загружать его в **QTextEdit**. Ниже приводится определение класса **MainWindow**:

```
class MainWindow : public QMainWindow
{
  Q_OBJECT
public:
  MainWindow(QWidget *parent = 0, const char *name = 0);

protected:
  void dragEnterEvent(QDragEnterEvent *event);
  void dropEvent(QDropEvent *event);

private:
  bool readFile(const QString &fileName);
  QString strippedName(const QString &fullFileName);

  QTextEdit *textEdit;
};
```

Класс **MainWindow** перекрывает методы предка (**QWidget**) **dragEnterEvent()** и **dropEvent()**. Так как целью данного примера является демонстрация работы механизма "drag and drop", ту часть реализации класса **MainWindow**, которая не имеет отношения к этому механизму, мы приводить не будем.

```
MainWindow::MainWindow(QWidget *parent, const char *name)
    : QMainWindow(parent, name)
{
  setCaption(tr("Drag File"));
  textEdit = new QTextEdit(this);
  setCentralWidget(textEdit);
  textEdit->viewport()->setAcceptDrops(false);
  setAcceptDrops(true);
}
```

В конструкторе создается объект класса **QTextEdit** и назначается центральным виджетом приложения. Далее запрещается "сброс" в область **QTextEdit** и разрешается для главного окна приложения.

Запрет на сброс для **QTextEdit** накладывается из-за того, что обработка перетаскивания и сброса будет реализована в классе главного окна приложения. По-умолчанию **QTextEdit** может принимать "сбрасываемые" текстовые данные, перетянутые из другого приложения, так например, если пользователь перетащит файл в область **QTextEdit**, то в виджет будет вставлено имя файла. Но это не совсем то, что нам надо. Мы собираемся открыть файл и поместить его содержимое в центральный виджет, а не вставить его имя в текст. Поэтому мы не будем использовать возможности класса **QTextEdit**, а реализуем свои собственные методы в классе главного окна приложения.
Поскольку события, в случае отказа от обработки, переходят от подчиненного виджета -- виджету владельцу, то главное окно получит событие "сброса" даже в том случае, если сброс осуществлялся в области **QTextEdit**.

```
void MainWindow::dragEnterEvent(QDragEnterEvent *event)
{
  event->accept(QUriDrag::canDecode(event));
}
```

Функция **dragEnterEvent()** вызывается, когда пользователь перемещает некий объект в область виджета. Если вызывается **accept(true)**, то это говорит о том, что пользователь может сбросить перетаскиваемый объект на виджет. Если вызывается **accept(false)** -- перетаскиваемый объект не может быть принят виджетом. Qt автоматически изменяет внешний вид указателя мыши, показывая пользователю -- может или не может быть сброшен данный объект в этом месте.

В нашем примере предполагается, что пользователь может сбросить в область приложения только имена файлов. Поэтому мы воспользовались услугами класса **QUriDrag**, который обслуживает перетаскивание файлов, для опознания перетаскиваемого объекта. Этот класс может использоваться для опознания Универсальных Идентификаторов Ресурсов (URI -- Universal Resource Identifier), таких как пути FTP или HTTP.

```
void MainWindow::dropEvent(QDropEvent *event)
{
  QStringList fileNames;
  if (QUriDrag::decodeLocalFiles(event, fileNames)) {
    if (readFile(fileNames[0]))
      setCaption(tr("%1 - Drag File")
                 .arg(strippedName(fileNames[0])));
  }
}
```

Функция **dropEvent()** вызывается в момент сброса объекта на виджет. Функция **QUriDrag::decodeLocalFiles()** возвращает список имен файлов, которые перетаскивает пользователь. Из этого списка мы вынимаем первый файл. Обычно пользователь перетаскивает файлы по одному, но возможна ситуация, когда перетаскивается несколько выделенных файлов.
Кроме того, класс **QWidget** предоставляет методы **dragMoveEvent()** и **dragLeaveEvent()**, но в большинстве приложений эти методы не используются.
Второй пример показывает -- как начать перетаскивание и как принять сбрасываемый объект. С этой целью мы создадим подкласс от **QListBox**, и реализуем в нем поддержку механизма "перетащил и бросил". Этот класс мы будем использовать в приложении "Project Chooser", показанном на рисунке 9.1.

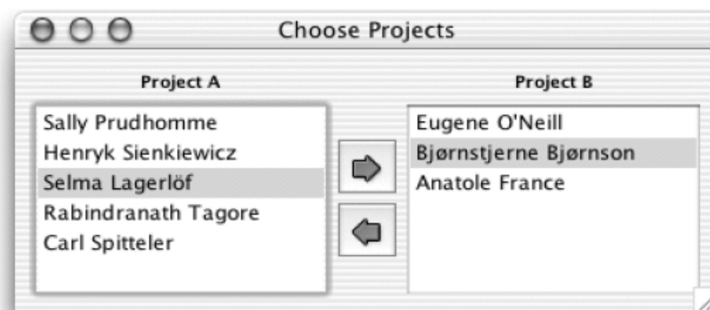


Рисунок 9.1. Внешний вид приложения "Project Chooser".

Окно приложения состоит из двух списков -- участников проектов. Каждый из списков отвечает за отдельный проект. Пользователь может перемещать имена участников проектов из одного списка в другой.
Вся реализация будет размещаться в единственном классе, потомке **QListBox**. Ниже приводится определение класса:

```
class ProjectView : public QListBox
{
  Q_OBJECT
public:
  ProjectView(QWidget *parent, const char *name = 0);

protected:
  void contentsMousePressEvent(QMouseEvent *event);
  void contentsMouseMoveEvent(QMouseEvent *event);
  void contentsDragEnterEvent(QDragEnterEvent *event);
  void contentsDropEvent(QDropEvent *event);

private:
  void startDrag();
  QPoint dragPos;
};
```

Класс **ProjectView** реализует четыре обработчика событий, объявленных в **QScrollView** (базовый класс для **QListBox**).

```
ProjectView::ProjectView(QWidget *parent, const char *name)
    : QListBox(parent, name)
{
  viewport()->setAcceptDrops(true);
}
```

В конструкторе мы разрешаем прием сбрасываемых объектов в область списка.

```
void ProjectView::contentsMousePressEvent(QMouseEvent *event)
{
  if (event->button() == LeftButton)
    dragPos = event->pos();
    QListBox::contentsMousePressEvent(event);
}
```

Когда пользователь нажимает левую кнопку мыши, позиция указателя запоминается в приватной переменной **dragPos** и вызывается метод предка **contentsMousePressEvent()**, чтобы обработать нажатие кнопки в обычном порядке.

```
void ProjectView::contentsMouseMoveEvent(QMouseEvent *event)
{
  if (event->state() & LeftButton) {
    int distance = (event->pos() - dragPos).manhattanLength();
    if (distance > QApplication::startDragDistance())
      startDrag();
  }
  QListBox::contentsMouseMoveEvent(event);
}
```

Когда пользователь перемещает указатель мыши, при удерживаемой левой кнопке, мы полагаем, что началось перетаскивание объекта. Далее вычисляется расстояние между текущим положением указателя мыши и точкой начала перетаскивания.
Если это расстояние больше, чем рекомендуемое классом **QApplication** (обычно 4 пикселя), после которого перемещение мыши действительно начинает рассматриваться как перетаскивание объекта, вызывается **startDrag()**, которая отмечает начало перетаскивания. Это дает возможность избежать ложного запуска процесса перетаскивания из-за дрожжания руки пользователя.

```
void ProjectView::startDrag()
{
```

```
  QString person = currentText();
  if (!person.isEmpty()) {
    QTextDrag *drag = new QTextDrag(person, this);
    drag->setSubtype("x-person");
    drag->setPixmap(QPixmap::fromMimeSource("person.png"));
    drag->drag();
  }
}
```

В **startDrag()** создается объект класса **QTextDrag**. Этот класс представляет перетаскиваемый объект, который содержит перемещаемый текст. Это один из нескольких предопределенных типов, которые предоставляет Qt для перетаскиваемых объектов. Кроме него можно еще назвать **QImageDrag**, **QColorDrag** и **QUriDrag**. Дополнительно, в соответствие перетаскиваемому объекту, мы ставим небольшую картинку, которая будет перемещаться вслед за указателем мыши, изображая перетаскиваемый объект.

Затем вызывается **setSubtype()**, которая устанавливает подтип объекта -- x-person. После этого полный тип объекта **MIME** приобретает значение **text/x-person**. Если этого не сделать, то перетаскиваемый объект будет иметь тип **MIME -- text/plain**.

Стандартные типы **MIME** определены IANA (Internet Assigned Numbers Authority). Полный **MIME** тип состоит из названия типа и подтипа, разделенных символом слэша. Когда создается нестандартный тип, рекомендуется предварять название подтипа префиксом x-. Типы **MIME** используются буфером обмена и механизмом "drag and drop" для идентификации различных типов данных.

Функция **drag()** отмечает начало операции перетаскивания. После этого Qt принимает на себя обязательства по владению перетаскиваемым объектом, пока перемещение не будет завершено. Она сама удалит объект, когда нужда в нем отпадет, даже если он так и не достигнет места назначения.

```
void ProjectView::contentsDragEnterEvent(QDragEnterEvent *event)
{
  event->accept(event->provides("text/x-person"));
}
```

Виджет класса **ProjectView** может не только начать перетаскивание объекта, типа **text/x-person**, но так же может принимать сбрасываемые объекты этого типа. Когда перемещаемый объект попадает в область виджета, выполняется проверка на корректность типа **MIME**.

```
void ProjectView::contentsDropEvent(QDropEvent *event)
{
  QString person;

  if (QTextDrag::decode(event, person)) {
    QWidget *fromWidget = event->source();
    if (fromWidget && fromWidget != this
            && fromWidget->inherits("ProjectView")) {
      ProjectView *fromProject = (ProjectView *)fromWidget;
      QListBoxItem *item = fromProject->findItem(person, ExactMatch);
      delete item;
      insertItem(person);
    }
  }
}
```

В функции **contentsDropEvent()**, с помощью **QTextDrag::decode()**, из перетаскиваемого объекта извлекается текстовая строка. Функция **QDropEvent::source()** возвращает указатель на виджет, в котором была начата операция перетаскивания, но только в том случае, если виджет принадлежит тому же самому приложению. Если виджет-приемник и виджет-источник -- это не одно и то же, и виджет-источник принадлежит классу **ProjectView**, то элемент списка удаляется из виджета-источника (вызовом **delete**) и вставляется в виджет-приемник.

9.2 Поддержка нестандартных типов данных при перетаскивании

До сих пор мы имели дело с предопределенными типами перетаскиваемых объектов. Например, мы использовали **QUriDrag**, для перетаскивания файлов, и **QTextDrag** -- для текста. Оба этих класса являются наследниками **QDragObject**, который служит базой для всех перемещаемых объектов. В свою очередь, класс **QDragObject** наследует свойства абстрактного класса **QMimeSource**, предназначенного для хранения данных различных типов.

Если вы пожелаете перемещать объекты с текстовой информацией, с изображениями, с именами файлов или с информацией о цвете, то можно использовать предопределенные классы Qt: **QTextDrag**, **QImageDrag**, **QUriDrag** и **QColorDrag**. Но если вам необходимо перемещать нестандартные типы данных, то у вас есть два пути:

- Сохранить информацию, в двоичном представлении, в объекте класса **QStoredDrag**.
- Создать свой собственный класс перетаскиваемых объектов, породив его от **QDragObject** и перекрыв соответствующие виртуальные методы.

Класс **QStoredDrag** может хранить любые двоичные данные, что позволяет использовать его для любых типов **MIME**. Например, если вам потребуется перетащить некоторые данные, хранящиеся в файле формата (фиктивного) ASDF, то можно рекомендовать примерно такой код:

```
void MyWidget::startDrag()
{
  QByteArray data = toAsdf();
  if (!data.isEmpty()) {
    QStoredDrag *drag = new QStoredDrag("octet-stream/x-asdf", this);
    drag->setEncodedData(data);
    drag->setPixmap(QPixmap::fromMimeSource("asdf.png"));
    drag->drag();
  }
}
```

Однако, **QStoredDrag** имеет ряд неудобств. Одно из них заключается в том, что он может хранить только один **MIME** тип. Если мы предполагаем использовать механизм "drag and drop" только в пределах одного приложения, то это не является большой проблемой. Но когда необходимо реализовать взаимодействие между различными приложениями, то одного **MIME** типа, как правило бывает недостаточно.

Другое неудобство состоит в необходимости преобразования данных в **QByteArray**, даже если приемник не может принимать данные этого типа. При достаточно большом объеме данных, это может привести к неоправданной потере производительности. Было бы намного удобнее, если бы преобразование выполнялось в момент сброса перетаскиваемого объекта.
Решение этих двух проблем заключается в создании дочернего класса от **QDragObject** и реализации двух виртуальных методов **format()** и **encodedData()**, используемых Qt для получения сведений о перетаскиваемых объектах. Чтобы показать -- как это можно сделать, мы создадим класс **CellDrag**, который будет хранить данные из одной или нескольких ячеек таблицы **QTable**.

```
class CellDrag : public QDragObject
{
public:
  CellDrag(const QString &text, QWidget *parent = 0,
           const char *name = 0);
  const char *format(int index) const;
  QByteArray encodedData(const char *format) const;

  static bool canDecode(const QMimeSource *source);
  static bool decode(const QMimeSource *source, QString &str);

private:
  QString toCsv() const;
  QString toHtml() const;
  QString plainText;
};
```

Класс **CellDrag** порожден от класса **QDragObject**. В нем только две функции имеют прямое отношение к механизму "drag and drop" -- это **format()** и **encodedData()**. Дополнительно, только лишь для удобства, он предоставляет в распоряжение программиста статические функции **canDecode()** и **decode()**, которые извлекают данные в момент сброса.

```
CellDrag::CellDrag(const QString &text, QWidget *parent,
                   const char *name)
    : QDragObject(parent, name)
{
  plainText = text;
}
```

Конструктору передается строка в текстовом виде, которая будет перемещаться. Это обычный текст, который может содержать символы табуляции и перевода строки. Этот текстовый тип мы использовали в Главе 4, когда добавляли в приложение Spreadsheet поддержку буфера обмена (см. раздел Реализация меню **Edit**).

```
const char *CellDrag::format(int index) const
{
  switch (index) {
    case 0:
      return "text/csv";
    case 1:
      return "text/html";
    case 2:
      return "text/plain";
    default:
      return 0;
  }
}
```

Функция **format()** перекрывает метод родительского класса **QMimeSource** и возвращает различные **MIME** типы, поддерживаемые объектом при перетаскивании. В нашем примере поддерживаются три типа данных: CSV (от англ. Comma-Separated Values -- Данные, Разделенные Запятыми), HTML и простой текст.

Когда Qt пытается определить -- какой **MIME** тип поддерживается перетаскиваемым объектом, она вызывает **format()** с аргументом **index**, равным 0, 1, 2... и так до тех пор, пока **format()** не вернет пустой указатель. Типы **MIME** для CSV и HTML были взяты из официального списка, который вы найдете по адресу: *http://www.iana.org/assignments/media-types/* .

Порядок следования форматов не имеет значения, однако, хорошей практикой считается помещать наиболее предпочтительные форматы в начало. Приложения, которые поддерживают несколько форматов, зачастую останавливаются на первом подходящем формате.

```
QByteArray CellDrag::encodedData(const char *format) const
{
  QByteArray data;
  QTextOStream out(data);

  if (qstrcmp(format, "text/csv") == 0) {
    out << toCsv();
  } else if (qstrcmp(format, "text/html") == 0) {
    out << toHtml();
  } else if (qstrcmp(format, "text/plain") == 0) {
    out << plainText;
  }
  return data;
}
```

Функция **encodedData()** возвращает данные в заказанном формате. Аргумент **format**, обычно содержит одну из строк, которую возвращает функция **format()**, но мы не можем безоговорочно утверждать это, поскольку не все приложения проверяют тип **MIME** вызовом **format()**. В приложениях Qt такая проверка обычно выполняется вызовом **provides()** внутри **QDragEnterEvent** и **QDragMoveEvent** (как мы это видели ранее).

Для преобразования **QString** в **QByteArray**, лучше использовать **QTextStream**.

```
QString CellDrag::toCsv() const
{
  QString out = plainText;
  out.replace("\\", "\\\\");
  out.replace("\"", "\\\"");
  out.replace("\t", "\", \"");
  out.replace("\n", "\"\n\"");
  out.prepend("\"");
  out.append("\"");
  return out;
}

QString CellDrag::toHtml() const
{
  QString out = QStyleSheet::escape(plainText);
  out.replace("\t", "<td>");
  out.replace("\n", "\n<tr><td>");
  out.prepend("<table>\n<tr><td>");
  out.append("\n</table>");
  return out;
}
```

Функции **toCsv()** и **toHtml()** выполняют преобразование символов табуляции и перевода строки в соответствующие элементы формата CSV и HTML. Например, данные

```
Red    Green    Blue
Cyan   Yellow   Magenta
```

будут преобразованы 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 простой заменой одних символов другими, с помощью **QString::replace()**. Для экранирования специальных символов HTML используется статическая функция **QStyleSheet::escape()**.

```
bool CellDrag::canDecode(const QMimeSource *source)
{
  return source->provides("text/plain");
}
```

Функция **canDecode()** возвращает **true**, если перетаскиваемые данные могут быть декодированы, в противном случае возвращается **false**.
Хотя мы и предусматриваем поддержку трех форматов для перетаскиваемых данных, мы будем принимать только данные в простом текстовом виде, поскольку для наших нужд этого будет более чем достаточно. Если пользователь попытается переместить ячейки из **QTable** в HTML-редактор, то данные будут преобразованы в HTML-таблицу. Но если пользователь попробует переместить произвольную HTML-таблицу (например, из браузера) в **QTable**, то эти данные не будут восприняты приложением.

```
bool CellDrag::decode(const QMimeSource *source, QString &str)
{
  QByteArray data = source->encodedData("text/plain");
  str = QString::fromLocal8Bit((const char *)data, data.size());
  return !str.isEmpty();
}
```

И, наконец, функция **decode()** преобразует **text/plain** данные в **QString**. Здесь мы предполагаем, что используется 8-ми битная кодировка символов.
Если вы пожелаете точно указывать кодировку символов, для перемещаемых данных, вы можете задать параметр **charset** формата **text/plain**, напимер:

```
text/plain;charset=US-ASCII
text/plain;charset=ISO-8859-1
text/plain;charset=Shift_JIS
```

Итак. Мы закончили описание реализации класса **CellDrag**. Нам осталось только интегрировать его с **QTable**. Оказывается, класс **QTable** уже выполняет почти все, что нам нужно. Единственное, что нам остается сделать -- это вызвать **setDragEnabled(true)** в конструкторе и перекрыть метод **QTable::dragObject()**, который будет возвращать **CellDrag**:

```
QDragObject *MyTable::dragObject()
{
  return new CellDrag(selectionAsString(), this);
}
```

Мы не приводим текст функции **selectionAsString()**, поскольку он почти полностью совпадает с текстом функции **Spreadsheet::copy()**.
Чтобы добавить поддержку приема данных, сбрасываемых на таблицу, необходимо перекрыть методы **contentsDragEnterEvent()** и **contentsDropEvent()** точно так же, как мы это делали в приложении "Project Chooser".

9.3 Расширенные возможности буфера обмена

Большинство приложений используют внутренние механизмы Qt, при работе с буфером обмена. Например, класс **QTextEdit** включает в себя поддержку комбинаций клавиш **Ctrl+X**, **Ctrl+C** и **Ctrl+V**, которые соответствуют слотам **cut()**, **copy()** и **paste()**. В результате этого, от программиста не требуется написания специального кода, отвечающего за работу с буфером обмена.

При написании собственных классов, вы можете получить доступ к буферу обмена с помощью функции **QApplication::clipboard()**, которая возвращает указатель на объект класса **QClipboard**. Работа с буфером обмена на удивление проста и незатейлива! Чтобы поместить в него данные нужно лишь вызвать метод **setText()**, **setImage()** или **setPixmap()**. Чтобы получить данные из буфера -- **text()**, **image()** или **pixmap()**. В Главе 4 мы уже пробовали работать с буфером обмена, при разработке приложения **Spreadsheet**.

Однако, в некоторых случаях, встроенной поддержки буфера обмена может оказаться недостаточно. Например, может потребоваться обеспечить поддержку данных, которые не являются ни текстом, ни рисунком. Или, с целью повышения совместимости с другими приложениями, необходимо будет организовать обмен данными в нескольких форматах. Проблема очень напоминает то, с чем мы уже столкнулись чуть выше, поэтому и решение ее практически аналогичное: необходимо создать дочерний класс от **QMimeSource** и перекрыть методы родительского класса **format()** и **encodedData()**.

Если в приложение включена поддержка механизма "drag and drop", то вы можете просто использовать уже существующий потомок класса **QDragObject**, помещая объекты этого типа в буфер обмена, вызовом **setData()**. Поскольку **QDragObject** ведет свою родословную от **QMimeSource**, а буфер обмена умеет взаимодействовать с классом **QMimeSource**, то все будет работать без особых проблем. Рассмотрим на примере, как можно реализовать функцию copy() для потомка класса QTable:

```
void MyTable::copy()
{
  QApplication::clipboard()->setData(dragObject());
}
```

В конце предыдущего раздела мы реализовали функцию **dragObject()**, которая возвращает **CellDrag**, предназначенный для хранения содержимого выделенных ячеек.

Чтобы извлечь данные из буфера обмена, необходимо обратиться к методу **data()**. Ниже приводится текст функции **paste()** для потомка класса **QTable**:

```
void MyTable::paste()
{
  QMimeSource *source = QApplication::clipboard()->data();
  if (CellDrag::canDecode(source)) {
    QString str;
    CellDrag::decode(source, str);
    performPaste(str);
  }
}
```

Функция **performPaste()** -- практически полный аналог функции **Spreadsheet::paste()** из Главы 4.

Это практически все, что необходимо для расширения возможностей при работе с буфером обмена. Буфер обмена X11, предоставляет дополнительные возможности, которые недоступны в операционных системах Windows и Mac OS X. В X11, обычно имеется возможность вставки выделенной области, щелчком средней кнопки трехкнопочной мыши, благодаря наличию отдельного буфера "выделения". Если вы желаете добавить поддержку этого буфера обмена в свои виджеты, вам придется добавить дополнительный аргумент **QClipboard::Selection** во все вызовы, обращающиеся к буферу обмена. Например, вот как можно реализовать обработчик события **mouseReleaseEvent()** в текстовом редакторе, который должен поддерживать вставку блоков текста по щелчку средней кнопки мыши:

```
void MyTextEditor::mouseReleaseEvent(QMouseEvent *event)
{
  QClipboard *clipboard = QApplication::clipboard();
  if (event->button() == MidButton
          && clipboard->supportsSelection()) {
    QString text = clipboard->text(QClipboard::Selection); pasteText(text);
  }
}
```

На платформе X11 функция **supportsSelection()** возвращает **true**, на других -- **false**.

10 Ввод/вывод

Темой обсуждения этой главы будут -- чтение и запись файлов, навигация по файловой системе и взаимодействие с внешними приложениями.

Qt предоставляет в ваше распоряжение два замечательных класса: **QDataStream** и **QTextStream**, которые значительно упрощают операции чтения-записи файлов. Они берут на себя хлопоты о порядке следования байт и кодировке текста, обеспечивая полную совместимость приложений на разных платформах.

Во многих приложениях необходимо реализовать возможность обхода файловой системы или предоставления сведений о файлах. Классы **QDir** и **QFileInfo** возьмут на себя эту "черную" и "неблагодарную" работу.

Иногда возникает необходимость запускать другие программы из нашего приложения. Класс **QProcess** сможет выполнить это в асинхронном режиме, не "замораживая" интерфейс с пользователем.

10.1 Чтение и запись двоичных данных

Чтение и запись данных произвольного формата, с помощью **QDataStream** -- это самый простой способ организовать сохранение и загрузку данных в Qt-приложении. Он поддерживает огромное количество типов данных Qt, включая **QByteArray**, **QFont**, **QImage**, **QMap<K, T>**, **QPixmap**, **QString**, **QValueList<T>** и **QVariant**. Перечень типов данных, поддерживаемых **QDataStream** вы найдете по адресу *http://doc.trolltech.com/3.2/datastreamformat.html* .
Чтобы продемонстрировать основные приемы работы с двоичными данными, мы напишем два класса: **Drawing** и **Gallery**. Первый будет хранить основные сведения о картине (имя художника, название и год создания), второй -- список картин.

Начнем с класса **Gallery**.

```
class Gallery : public QObject
{
public:
  bool loadBinary(const QString &fileName);
  bool saveBinary(const QString &fileName);
  ...

private:
  enum { MagicNumber = 0x98c58f26 };

  void writeToStream(QDataStream &out);
  void readFromStream(QDataStream &in);
  void error(const QFile &file, const QString &message);
  void ioError(const QFile &file, const QString &message);

  QByteArray getData();
  void setData(const QByteArray &data);
  QString toString();

  std::list<Drawing> drawings;
};
```

Он содержит публичные функции, которые сохраняют и загружают данные. Данные -- это список картин. Каждый элемент списка -- это объект класса **Drawing**. Приватные функции мы будем рассматривать по мере необходимости.
Ниже приводится исходный текст функции, сохраняющей список картин в двоичном виде:

```
bool Gallery::saveBinary(const QString &fileName)
{
  QFile file(fileName);
  if (!file.open(IO_WriteOnly)) {
    ioError(file, tr("Cannot open file %1 for writing"));
```

```
    return false;
  }

  QDataStream out(&file);
  out.setVersion(5);
  out << (Q_UINT32)MagicNumber;
  writeToStream(out);
  if (file.status() != IO_Ok) {
    ioError(file, tr("Error writing to file %1"));
    return false;
  }
  return true;
}
```

Сначала мы открываем файл. Затем устанавливаем версию **QDataStream**. Номер версии определяет способ сохранения различных типов данных. Базовые типы языка C++ всегда сохраняются в неизменном виде.
Далее в файл выводится сигнатура (число), которая идентифицирует файлы галереи. Чтобы обеспечить совместимость с другими платформами, мы приводим **MagicNumber** к типу **Q_UINT32**.
Список картин выводится в файл приватной функцией **writeToStream()**. Нет необходимости явно закрывать файл -- это будет сделано автоматически, когда объект **QFile** выйдет из области видимости по завершении функции.
После вызова **writeToStream()** проверяется статус устройства **QFile**. Если возникла ошибка -- вызывается **ioError()**, которая выводит окно с сообщением и вызывающей программе возвращается значение **false**.

```
void Gallery::ioError(const QFile &file, const QString &message)
{
  error(file, message + ": " + file.errorString());
}
```

Функция **ioError()** вызывает более универсальную функцию **error()**:

```
void Gallery::error(const QFile &file, const QString &message)
{
  QMessageBox::warning(0, tr("Gallery"), message.arg(file.name()));
}
```

Теперь рассмотрим функцию **writeToStream()**:

```
void Gallery::writeToStream(QDataStream &out)
{
  list<Drawing>::const_iterator it = drawings.begin();
  while (it != drawings.end()) {
    out << *it;
    ++it;
  }
}
```

Она последовательно проходит по списку картин и сохраняет их одну за другой в поток, который был передан в качестве аргумента. Если бы мы, вместо **list<Drawing>** использовали определение **QValueList<Drawing>**, мы могли бы обойтись без цикла, просто записав:

```
  out << drawings;
```

Когда **QValueList<T>** помещается в поток, то каждый элемент списка записывается посредством его собственного оператора "**<<**".

```
QDataStream &operator<<(QDataStream &out, const Drawing &drawing)
{
  out << drawing.myTitle << drawing.myArtist << drawing.myYear;
  return out;
}
```

Вывод объекта **Drawing** осуществляется простой записью трех его переменных-членов: **myTitle**, **myArtist** и **myYear**. Перегруженный оператор **operator<<()** должен быть объявлен как "дружественный" (**friend**). В заключение функция возвращает поток. Это общепринятая в языке C++ идиома программирования, которая позволяет объединять операторы "**<<**" в цепочки, например:

```
out << drawing1 << drawing2 << drawing3;
```

Ниже приводится определение класса **Drawing**:

```
class Drawing
{
  friend QDataStream &operator<<(QDataStream &, const Drawing &);
  friend QDataStream &operator>>(QDataStream &, Drawing &);

public:
  Drawing() { myYear = 0; }
  Drawing(const QString &title, const QString &artist, int year)
  { myTitle = title; myArtist = artist; myYear = year; }

  QString title() const { return myTitle; }
  void setTitle(const QString &title) { myTitle = title; }
  QString artist() const { return myArtist; }
  void setArtist(const QString &artist) { myArtist = artist; }
  int year() const { return myYear; }
  void setYear(int year) { myYear = year; }

private:
  QString myTitle;
  QString myArtist;
  int myYear;
};
```

Рассмотрим функцию, которая читает файл со списком картин:

```
bool Gallery::loadBinary(const QString &fileName)
{
  QFile file(fileName);
  if (!file.open(IO_ReadOnly)) {
    ioError(file, tr("Cannot open file %1 for reading"));
    return false;
  }

  QDataStream in(&file);
  in.setVersion(5);

  Q_UINT32 magic;
  in >> magic;
  if (magic != MagicNumber) {
    error(file, tr("File %1 is not a Gallery file"));
    return false;
  }

  readFromStream(in);

  if (file.status() != IO_Ok) {
    ioError(file, tr("Error reading from file %1"));
    return false;
  }
  return true;
}
```

Файл открывается на чтение и создается объект **QDataStream**, который будет читать данные из файла. Мы установили версию 5 для **QDataStream**, поскольку в этой версии была произведена запись в файл. Использование фиксированного номера версии -- 5, гарантирует, что приложение всегда сможет читать и записывать данные, если оно собрано с Qt 3.2 или более поздней.
Работа с файлом начинается со считывания сигнатуры (числа) **MagicNumber**. Это дает нам уверенность, что мы работаем с файлом, содержащим список картин, а не что-то иное. Затем список считывается функцией **readFromStream()**.

```
void Gallery::readFromStream(QDataStream &in)
{
  drawings.clear();
  while (!in.atEnd()) {
    Drawing drawing;
    in >> drawing;
    drawings.push_back(drawing);
  }
}
```

Функция начинается с очистки ранее находившихся в списке данных. Затем в цикле производится считывание всех описаний картин, одного за другим. Если бы мы, вместо **list<Drawing>** использовали определение **QValueList<Drawing>**, мы могли бы обойтись без цикла, просто записав:

```
  in >> drawings;
```

Когда **QValueList<T>** получает данные из потока, то каждый элемент списка читается посредством его собственного оператора "**>>**".

```
QDataStream &operator>>(QDataStream &in, Drawing &drawing)
{
  in >> drawing.myTitle >> drawing.myArtist >> drawing.myYear;
  return in;
}
```

Реализация оператора "**>>**" является зеркальным отражением оператора "**<<**". При использовании **QDataStream** у нас не возникает необходимости производить синтаксический анализ в любом его проявлении.

При желании, читать и записывать любые двоичные данные в необработанном виде, можно с помощью функций **readRawBytes()** и **writeRawBytes()**.
Чтение и запись данных базовых типов (таких как **Q_UINT16** или **float**), может производиться как операторами "**<<**" и "**>>**", так и с помощью функций **readRawBytes()** и **writeRawBytes()**. По-умолчанию, порядок следования байт, используемый **QDataStream** -- "**big-endian**". Для того, чтобы изменить его на "**little-endian**" (храктерный для платформы Intel), необходимо указывать его явно:

```
  stream.setByteOrder(QDataStream::LittleEndian);
```

В случае чтения/записи базовых типов языка C++, указывать версию, через вызов **setVersion()**, необязательно.
Если необходимо записать/прочитать файл, что называется "за один присест", то можно воспользоваться методами класса **QFile** -- **writeBlock()** и **readAll()**, например:

```
  file.writeBlock(getData());
```

Данные, записанные таким образом, находятся в файле в виде простой последовательности байт. Однако, в этом случае, вся ответственность за структурирование и идентификацию данных при считывании, полностью ложится на плечи разработчика. За создание списка **QByteArray** и заполнение его данными, в классе **Gallery** отвечает приватная функция **getData()**. Чтение блока данных из файла выглядит не менее просто, чем запись:

```
  setData(file.readAll());
```

За извлечение данных из **QByteArray**, в классе **Gallery** отвечает приватная функция **setData()**.
Сохранение всех данных, в виде **QByteArray**, может потребовать значительного объема памяти, но такой способ имеет свои преимущества. Например, мы можем сжать данные, с помощью **qCompress()**, при записи в файл:

```
  file.writeBlock(qCompress(getData()));
```

И разархивировать при считывании:

```
  setData(qUncompress(file.readAll()));
```

Ниже приводится один из возможных вариантов реализации функций **getData()** и **setData()**:

```
QByteArray Gallery::getData()
{
  QByteArray data;
  QDataStream out(data, IO_WriteOnly);
  writeToStream(out);
  return data;
}
```

Здесь создается поток **QDataStream**, которому в качестве устройства вывода, вместо **QFile**, назначается **QByteArray**. После этого массив заполняется двоичными данными, вызовом **writeToStream()**.
Аналогичным образом, функция **setData()** обращается к **readFromStream()**, для чтения ранее записанных данных:

```
void Gallery::setData(const QByteArray &data)
{
  QDataStream in(data, IO_ReadOnly);
  readFromStream(in);
}
```

В примерах выше, мы сохраняли и считывали данные, жестко задавая номер версии для **QDataStream**. Такой подход достаточно прост и надежен, но он имеет один маленький недостаток: мы не сможем работать с файлами, записанными с новыми версиями. Например, если в последующих версиях Qt, в класс **QFont** будут добавлены новые элементы, то мы лишимся возможности сохранять и загружать компоненты этого типа, используя более старую версию **QDataStream**.
Как одно из возможных решений этой проблемы -- записывать в файл номер версии:

```
  QDataStream out(&file);
  out << (Q_UINT32)MagicNumber;
  out << (Q_UINT16)out.version();
  writeToStream(out);
```

Этот код будет выполнять запись данных, с использованием самой последней версии **QDataStream**.
При чтении таких файлов, сначала будет считываться сигнатура файла и номер версии **QDataStream**:

```
  QDataStream in(&file);

  Q_UINT32 magic;
  Q_UINT16 streamVersion;
  in >> magic >> streamVersion;

  if (magic != MagicNumber) {
    error(file, tr("File %1 is not a Gallery file"));
    return false;
  } else if ((int)streamVersion > in.version()) {
    error(file, tr("File %1 is from a more recent version of the "
                   "application"));
    return false;
  }
  in.setVersion(streamVersion);
  readFromStream(in);
```

Чтение данных будет возможно в том случае, если номер версии будет меньше или равен версии, используемой приложением. В противном случае чтение завершится сообщением об ошибке.
Вместо версии **QDataStream** можно использовать версию приложения. Например, допустим, что некий формат файла соответствует версии 1.3 приложения. Тогда мы могли бы записать следующий код:

```
  QDataStream out(&file);
  out.setVersion(5);
  out << (Q_UINT32)MagicNumber;
  out << (Q_UINT16)0x0103;
  writeToStream(out);
```

При чтении такого файла можно определять версию **QDataStream**, основываясь на версии приложения:

```
QDataStream in(&file);

Q_UINT32 magic;
Q_UINT16 appVersion;
in >> magic >> appVersion;

if (magic != MagicNumber) {
  error(file, tr("File %1 is not a Gallery file"));
  return false;
} else if (appVersion > 0x0103) {
  error(file, tr("File %1 is from a more recent version of the "
                 "application"));
  return false;
}

if (appVersion <= 0x0102) {
  in.setVersion(4);
} else {
  in.setVersion(5);
}
readFromStream(in);
```

Этот код говорит, что для чтения данных из файла, созданного приложением с версией 1.2 или более ранней, должна использоваться 4-я версия **QDataStream**, для чтения данных из файла, созданного приложением с версией 1.3 -- 5-я версия **QDataStream**.
Как только мы получаем в руки механизм определения версии **QDataStream**, процедура чтения и записи двоичных данных становится простой и надежной.

10.2 Чтение и запись текста

Для чтения и записи текстовых данных, Qt предоставляет класс **QTextStream**. Он может использоваться как для чтения/записи простого текста, так и для файлов с другими текстовыми форматами, такими как HTML, XML и файлов с исходными текстами программ. Он принимает на себя обязательства по преобразованию кодировки символов между Unicode и 8-ми битными кодировками, а так же по разному обрабатывает признак окончания строки, в соответствии с соглашениями, принятыми в различных операционных системах.
В качестве фундаментального типа данных, **QTextStream** использует **QChar**. В дополнение к символьным и строковым данным, **QTextStream** поддерживает базовые числовые типы языка C++, конвертируя их в/из строки.
С целью демонстрации возможностей **QTextStream**, продолжим рассмотрение реализации класса **Gallery**. Ниже приводится исходный текст функции **saveText()**, которая сохраняет список картин в простой текстовый файл:

```
bool Gallery::saveText(const QString &fileName)
{
  QFile file(fileName);
  if (!file.open(IO_WriteOnly | IO_Translate)) {
    ioError(file, tr("Cannot open file %1 for writing"));
    return false;
  }

  QTextStream out(&file);
  out.setEncoding(QTextStream::UnicodeUTF8);

  list<Drawing>::const_iterator it = drawings.begin();
  while (it != drawings.end()) {
    out << *it;
    ++it;
  }
  if (file.status() != IO_Ok) {
    ioError(file, tr("Error writing to file %1"));
    return false;
```

```
  }
  return true;
}
```

При открытии файла используется флаг **IO_Translate**, чтобы корректным образом перевести символ перевода строки в последовательность символов, которая соответствует используемой операционной системе ("**/r/n**" -- для Windows, "**/r**" -- для Mac OS X). Затем устанавливается кодировка символов UTF-8, совместимая с ASCII. (За дополнительной информацией об Unicode, см. Главу 15.) После этого, в цикле, в файл выводятся описания картин, с помощью перегруженного оператора "**<<**":

```
QTextStream &operator<<(QTextStream &out, const Drawing &drawing)
{
  out << drawing.myTitle << ":" << drawing.myArtist << ":"
      << drawing.myYear << endl;
  return out;
}
```

При записи сведений о картине, в качестве разделителя полей, используется символ двоеточия. Каждая запись в файле завершается символом перевода строки. При этом мы исходим из предположения, что ни имя художника, ни название картины не содержат символов двоеточия или перевода строки.
Ниже показан пример содержимого файла, созданного функцией **saveText()**:

```
      The False Shepherds:Hans Bol:1576
      Panoramic Landscape:Jan Brueghel the Younger:1619
      Dune Landscape:Jan van Goyen:1630
      River Delta:Jan van Goyen:1653
```

Теперь перейдем к функции чтения файла:

```
bool Gallery::loadText(const QString &fileName)
{
  QFile file(fileName);
  if (!file.open(IO_ReadOnly | IO_Translate)) {
    ioError(file, tr("Cannot open file %1 for reading"));
    return false;
  }

  drawings.clear();
  QTextStream in(&file);
  in.setEncoding(QTextStream::UnicodeUTF8);

  while (!in.atEnd()) {
    Drawing drawing;
    in >> drawing;
    drawings.push_back(drawing);
  }

  if (file.status() != IO_Ok) {
    ioError(file, tr("Error reading from file %1"));
    return false;
  }
  return true;
}
```

Все самое интересное в этой функции, заключено внутри цикла **while**. Он выполняет чтение данных, с помощью оператора "**>>**", до тех пор, пока не будет достигнут конец файла.
Реализация оператора "**>>**" не так тривиальна, поскольку представление текстовых данных не так однозначно. Рассмотрим следующий пример:

```
  out << "alpha" << "bravo";
```

Если исходить из того, что **out** -- это экземпляр класса **QTextStream**, то в файл фактически будет записана одна строка "**alphabravo**". Мы не сможем прочитать данные, просто написав:

```
  in >> str1 >> str2;
```

Фактически, в переменную **str1** будет записана строка "**alphabravo**", а в переменную **str2** -- ничего.

Если записываемый текст состоит из отдельных слов, мы можем вставлять пробелы между ними и затем читать этот текст слово за словом. (Этот подход был реализован в функциях **DiagramView::copy()** и **DiagramView::paste()**, в Главе 8.) Но в данном случае этот вариант не подходит, поскольку имя художника и название картины могут состоять более чем из одного слова. Поэтому, за один раз читается целая строка и затем разбивается на элементы, с помощью функции **QStringList::split()**:

```
QTextStream &operator>>(QTextStream &in, Drawing &drawing)
{
  QString str = in.readLine();
  QStringList fields = QStringList::split(":", str);
  if (fields.size() == 3) {
    drawing.myTitle = fields[0];
    drawing.myArtist = fields[1];
    drawing.myYear = fields[2].toInt();
  }
  return in;
}
```

Текстовые файлы могут читаться за один прием, с помощью **QTextStream::read()**:

```
  QString wholeFile = in.read();
```

В переменной, конец каждой строки будет отмечен символом **'\n'**, независимо от используемой операционной системы.
Считывание файла за раз может оказаться удобным решением, если данные должны пройти предварительную обработку, например:

```
  wholeFile.replace("&", "&amp;");
  wholeFile.replace("<", "&lt;");
  wholeFile.replace(">", "&gt;");
```

Чтобы записать данные в файл за одно обращение, можно сначала разместить их в переменной, а затем вывести на диск:

```
QString Gallery::saveToString()
{
  QString result;
  QTextOStream out(&result);
  list<Drawing>::const_iterator it = drawings.begin();
  while (it != drawings.end()) {
    out << *it;
    ++it;
  }
  return result;
}
```

Связать поток со строковой переменной так же просто, как и связать поток с файлом.

```
void Gallery::readFromString(const QString &data)
{
  QString string = data;
  drawings.clear();
  QTextIStream in(&string);
  while (!in.atEnd()) {
    Drawing drawing;
    in >> drawing;
    drawings.push_back(drawing);
  }
}
```

Запись текстовых данных -- довольно простая операция, а вот чтение их может оказаться довольно сложной задачей. В случае использования сложных форматов может потребоваться написать

полноценный синтаксический анализатор. Как правило, подобные анализаторы считывают текст символ за символом, с помощью оператора "**>>**" в переменную типа **QChar** или построчно, с помощью **readLine()** и затем анализируют полученную строку.

10.3 Работа с файлами и каталогами

Класс **QDir** дает возможность навигации по файловой системе и получать информацию о файлах, независимо от типа операционной системы. Чтобы показать некоторые особенности класса **QDir**, напишем небольшое консольное приложение, которое подсчитывает суммарный объем всех файлов с изображениями в заданном каталоге и вложенных подкаталогах.
Основу приложения составляет функция **imageSpace()**, которая суммирует размеры файлов в заданном каталоге:

```
int imageSpace(const QString &path)
{
  QDir dir(path);
  QStringList::Iterator it;
  int size = 0;

  QStringList files = dir.entryList("*.png *.jpg *.jpeg",
                                    QDir::Files);
  it = files.begin();
  while (it != files.end()) {
    size += QFileInfo(path, *it).size();
    ++it;
  }

  QStringList dirs = dir.entryList(QDir::Dirs);
  it = dirs.begin();
  while (it != dirs.end()) {
    if (*it != "." && *it != "..")
      size += imageSpace(path + "/" + *it);
    ++it;
  }
  return size;
}
```

Начинается она с создания экземпляра класса **QDir**, с заданным полным именем каталога. Затем вызывается функция **entryList()**, которой передаются два аргумента. Первый из них -- это список шаблонов имен файлов, разделенных пробелами. В шаблонах допускается указывать символы подстановки '*****' и '**?**'. В данном примере будут учитываться только файлы изображений, в форматах JPEG и PNG. Второй аргумент определяет тип элементов результирующего списка (обычные файлы, каталоги, устройства и пр.).

Затем, в цикле, осуществляется проход по списку файлов и суммируются их размеры. Класс **QFileInfo** позволяет получить доступ к таким характеристикам файла, как размер, права доступа, владелец и время (создания, последнего обращения, последнего изменения).

Вторым обращением к **entryList()** создается список вложенных подкаталогов. После чего, в цикле, выполняется проход по подкаталогам, с рекурсивным вызовом **imageSpace()** для каждого из них. Полный путь к вложенным подкаталогам "собирается" из полного пути к текущему каталогу, символа слэша и имени подкаталога (***it**). Класс **QDir** интерпретирует символ "**/**" как разделитель имен каталогов независимо от используемой операционной системы. Перед выводом полного пути перед пользователем, можно вызвать функцию **QDir::convertSeparators()**, которая преобразует символ "**/**" в корректное представление, в зависимости от используемой платформы.

Добавим в нашу программу функцию **main()**:

```
int main(int argc, char *argv[])
{
  QString path = QDir::currentDirPath();
  if (argc > 1)
    path = argv[1];
```

```
  cerr << "Space used by images in " << endl
       << path.ascii() << endl
       << "and its subdirectories is "
       << (imageSpace(path) / 1024) << " KB" << endl;
  return 0;
}
```

В этом примере мы не создавали объект класса **QApplication**, потому что мы воспользовались только инструментальными классами, не имеющими отношения к графическому интерфейсу. Полный список таких классов вы найдете по адресу: *http://doc.trolltech.com/3.2/tools.html*.

Для начальной инициализации переменной **path** была использована функция **QDir::currentDirPath()**, которая возвращает полное имя текущего каталога. В качестве альтернативы можно было бы использовать функцию **QDir::homeDirPath()**, возвращающую полный путь к домашнему каталогу пользователя. Если путь к каталогу задается пользователем из командной строки, то он замещает значение по-умолчанию. В заключение вызывается функция **imageSpace()**, которая подсчитывает суммарный размер всех файлов с изображениями.
Класс **QDir** предоставляет ряд других функций, для работы с каталогами и файлами, среди них: **rename()**, **exists()**, **mkdir()** и **rmdir()**.

10.4 Взаимодействия между процессами

Класс **QProcess** позволяет запускать и взаимодействовать с другими программами. Экземпляры класса работают асинхронно, выполняя всю работу в фоновом режиме, что не приводит к "замораживанию" пользовательского интерфейса. **QProcess** может известить приложение о завершении запущенной им программы или о наличии данных, полученных от нее, выдавая соответствующие сигналы.
В демонстрационных целях напишем небольшое приложение, которое предоставит пользователю интерфейс с внешней программой преобразования графических файлов. Для данного примера будет использоваться программа **convert** из пакета **ImageMagick**, которая доступна для большинства платформ.

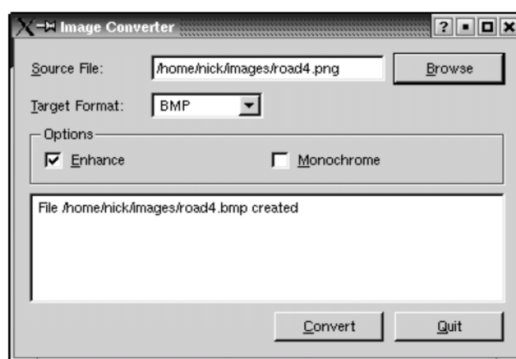


Рисунок 10.1. Внешний вид приложения Image Converter.

Форма приложения была разработана в среде визуального построителя интерфейсов Qt Designer. Соответствующий **.ui** находится на CD, сопровождающем книгу. Здесь же мы сконцентрируем все свое внимание на содержимом файла **.ui.h**, который содержит исходный код. Обратите внимание: переменные **process** и **fileFilters** были объявлены в Qt Designer, на вкладке **Members** следующим образом:

```
    QProcess *process;
    QString fileFilters;
```

Утилита **uic** добавляет эти переменные в класс **ConvertDialog**.

```
void ConvertDialog::init()
{
  process = 0;
  QStringList imageFormats = QImage::outputFormatList();
  targetFormatComboBox->insertStringList(imageFormats);
```

```
  fileFilters = tr("Images") + " (*." +
                imageFormats.join(" *.").lower() + ")";
}
```

Переменная **fileFilters** содержит текст описания и один, или более, шаблонов имен файлов (например, "Text files (***.txt**)"). Функция **QImage::outputFormatList()** возвращает список форматов изображений, поддерживаемых Qt. Этот список тесно связан с опциями, выбранными при установке библиотки.

```
void ConvertDialog::browse()
{
  QString initialName = sourceFileEdit->text();
  if (initialName.isEmpty())
    initialName = QDir::homeDirPath();
  QString fileName =
          QFileDialog::getOpenFileName(initialName, fileFilters,
                                       this);
  fileName = QDir::convertSeparators(fileName);
  if (!fileName.isEmpty()) {
    sourceFileEdit->setText(fileName);
    convertButton->setEnabled(true);
  }
}
```

Кнопка **Browse** связана со слотом **browse()**. Если ранее пользователь уже выбирал файл, то путь поиска, для диалога выбора файла, назначается исходя из полного имени предыдущего файла, в противном случае, открывается домашний каталог.

```
void ConvertDialog::convert()
{
  QString sourceFile = sourceFileEdit->text();
  targetFile = QFileInfo(sourceFile).dirPath() + QDir::separator()
               + QFileInfo(sourceFile).baseName();
  targetFile += ".";
  targetFile += targetFormatComboBox->currentText().lower();
  convertButton->setEnabled(false);
  outputTextEdit->clear();
  process = new QProcess(this);
  process->addArgument("convert");
  if (enhanceCheckBox->isChecked())
    process->addArgument("-enhance");
  if (monochromeCheckBox->isChecked())
    process->addArgument("-monochrome");
  process->addArgument(sourceFile);
  process->addArgument(targetFile);
  connect(process, SIGNAL(readyReadStderr()),
          this, SLOT(updateOutputTextEdit()));
  connect(process, SIGNAL(processExited()),
          this, SLOT(processExited()));
  process->start();
}
```

Кнопка **Convert** связана со слотом **convert()**. По сигналу от кнопки собирается имя целевого файла, из имени исходного файла и расширения, соответствующего заданному формату.
Затем создается экземпляр класса **QProcess**. После этого собирается список аргументов командной строки, с помощью функции **addArgument()**. Первым идет имя файла внешней программы. Далее следуют аргументы, которые будут ей передаваться.
После создания списка аргументов производится соединение сигнала **readyReadStderr()**, класса **QProcess**, со слотом **updateOutputTextEdit()** диалогового окна, чтобы выводить в **QTextEdit** сообщения от внешней программы, по мере их поступления. И затем соединяется сигнал **processExited()**, класса **QProcess**, со слотом **processExited()** диалогового окна.

```
void ConvertDialog::updateOutputTextEdit()
{
  QByteArray data = process->readStderr();
  QString text = outputTextEdit->text() + QString(data);
  outputTextEdit->setText(text);
}
```

Как только внешняя программа выдаст что нибудь на stderr, будет вызван слот **updateOutputTextEdit()**. Сообщение будет прочитано и записано в **QTextEdit**.

```
void ConvertDialog::processExited()
{
  if (process->normalExit()) {
    outputTextEdit->append(tr("File %1 created")
                           .arg(targetFile));
  } else {
    outputTextEdit->append(tr("Conversion failed"));
  }
  delete process;
  process = 0;
  convertButton->setEnabled(true);
}
```

По завершении внешнего процесса перед пользователем выводится соответствующее сообщение, после чего процесс удаляется.
Создание графического интерфейса, для консольных приложений, подобным образом, может оказаться очень полезным, потому что позволяет использовать функциональность, заложенную в уже существующие программы и нам не нужно ломать голову над собственной реализацией.

11 Контейнерные классы

Контейнерные классы -- это универсальные шаблонные классы, предназначенные для хранения элементов заданного типа в смежных областях памяти. Стандарт C++ уже включает в себя большое количество контейнеров, как часть STL (Standard Template Library -- Стандартная Библиотека Шаблонов).

Qt имеет свой набор шаблонных классов. Таким образом, при создании программ разработчик может использовать как контейнерные классы из библиотеки Qt, так и классы из STL. Если вы уже знакомы с контейнерами из STL, то мы не видим веских причин для того, чтобы насильно заставлять себя переходить на использование контейнеров из Qt.

В этой главе мы рассмотрим наиболее важные контейнеры из STL и Qt. Мы так же поближе рассмотрим классы **QString** и **QVariant**, которые имеют много общего с контейнерами и в отдельных случаях могут использоваться как альтернатива контейнерам.
Начальные сведения о классах и функциях STL вы найдете по адресу: *http://www.sgi.com/tech/stl/*.

11.1 Векторы

Классы векторов, списков и словарей (**map**) -- это шаблонные классы, параметризуемые типом объектов, которые предполагается хранить в контейнере. Значения, которые хранятся в контейнерах, могут быть базового типа (например **int** или **double**), указателями или классами, которые имеют конструктор по-умолчанию (конструктор, у которого нет входных аргументов или все входные аргументы имеют значения по-умолчанию), конструктор копирования и перегруженный оператор присваивания. Среди классов, которые отвечают этим требованиям, можно назвать **QDateTime**, **QRegExp**, **QString** и **QVariant**. Классы Qt, которые наследуют **QObject**, не могут быть помещены в контейнеры, поскольку у них нет конструктора копирования и оператора присваивания. Однако, это не является большой проблемой, поскольку сохраняется возможность помещать в контейнеры указатели этих типов.

В этом разделе мы рассмотрим наиболее общие операции над векторами, а в следующих двух разделах расскажем о списках и словарях (**map**). Большая часть примеров, рассматриваемых в этой главе, будет основана на классе **Film**, который хранит название фильма и его продолжительность. (Мы отказались от более подходящего для этого случая названия **Movie**, потому что это имя очень похоже на **QMovie** -- класс Qt, который предназначен для показа анимированных изображений.) Ниже приводится определение класса **Film**:

```
class Film
{
public:
  Film(int id = 0, const QString &title = "", int duration = 0);

  int id() const { return myId; }
  void setId(int catalogId) { myId = catalogId; }
  QString title() const { return myTitle; }
  void setTitle(const QString &title) { myTitle = title; }
  int duration() const { return myDuration; }
  void setDuration(int minutes) { myDuration = minutes; }

private:
  int myId;
  QString myTitle;
  int myDuration;
};

int operator==(const Film &film1, const Film &film2);
int operator<(const Film &film1, const Film &film2);
```

Мы не включили в класс явное определение конструктора копирования и оператора присваивания, потому что они предоставляются C++ автоматически. Если бы наш класс выполнял дополнительное резервирование памяти, под данные-члены, тогда нам пришлось бы включить в него явную реализацию конструктора копирования и оператора присваивания.

В дополнение к классу мы реализовали два оператора сравнения -- "равно" и "меньше". Оператор "равно" используется для поиска элемента в контейнере. Оператор "меньше" -- используется для нужд сортировки. В данной ситуации нет необходимости реализовать четыре других оператора сравнения ("**!=**", "**<=**", "**>**", "**>=**"), поскольку STL никогда ими не пользуется.
Ниже приводится исходный код трех функций:

```
Film::Film(int id, const QString &title, int duration)
{
  myId = id;
  myTitle = title;
  myDuration = duration;
}

int operator==(const Film &film1, const Film &film2)
{
  return film1.id() == film2.id();
}

int operator<(const Film &film1, const Film &film2)
{
  return film1.id() < film2.id();
}
```

При сравнивании экземпляров **Film**, используются их числовые идентификаторы, а не названия, поскольку к названию фильма не предъявляется требование уникальности.

Рисунок 11.1. Вектор экземпляров класса Film.

Вектор -- это структура данных, которая хранит элементы, подобно обычному массиву. Главное отличие вектора от массива C++ состоит в том, что вектор всегда "знает", сколько элементов он хранит, и может динамически изменять свой размер. Добавление новых элементов в конец вектора выполняется очень быстро, но операция по вставке новых элементов в начало или в середину вектора требует значительных затрат по времени.
В STL, класс вектора носит имя **std::vector<T>** и определен в заголовке <**vector**>. Объявить вектор, который будет хранить массив экземпляров класса **Film**, можно так:

```
vector<Film> films;
```

Эквивалентное объявление, использующее класс Qt -- **QValueVector<T>**:

```
QValueVector<Film> films;
```

Вектор, созданный подобным образом, изначально имеет размер 0. Если заранее известно количество элементов в векторе, можно явно указать начальный размер в определении и с помощью оператора "**[ ]**" присвоить значения его элементам.
Очень удобно заполнять вектор с помощью функции **push_back()**. Она добавляет указанный элемент в конец вектора, увеличивая его размер на 1:

```
films.push_back(Film(4812, "A Hard Day's Night", 85));
films.push_back(Film(5051, "Seven Days to Noon", 94));
films.push_back(Film(1301, "Day of Wrath", 105));
films.push_back(Film(9227, "A Special Day", 110));
films.push_back(Film(1817, "Day for Night", 116));
```

Как правило, Qt предоставляет функции с теми же именами, что и STL, но в некоторых случаях Qt добавляет к классам дополнительные методы, с более интуитивно понятными именами. Например, классы Qt могут добавлять элементы как с помощью метода **push_back()**, так и с помощью дополнительного метода **append()**.
Еще один способ заполнения вектора состоит в том, чтобы задать при объявлении его начальный размер, а потом выполнить инициализацию отдельных элементов:

```
vector<Film> films(5);

films[0] = Film(4812, "A Hard Day's Night", 85);
films[1] = Film(5051, "Seven Days to Noon", 94);
films[2] = Film(1301, "Day of Wrath", 105);
films[3] = Film(9227, "A Special Day", 110);
films[4] = Film(1817, "Day for Night", 116);
```

Элементы вектора, которые не были инициализированы явно, приобретают значения, присвоенные конструктором по-умолчанию. В случае базовых типов языка C++ и указателей, начальные значения элементов вектора не определены, аналогично локальным переменным, размещаемым на стеке.
Векторы допускают обход всех элементов в цикле, с использованием оператора "**[ ]**":

```
  for (int i = 0; i < (int)films.size(); ++i)
    cerr << films[i].title().ascii() << endl;
```

В качестве альтернативы -- можно использовать итератор:

```
  vector<Film>::const_iterator it = films.begin();
  while (it != films.end()) {
    cerr << (*it).title().ascii() << endl;
    ++it;
  }
```

Каждый контейнерный класс имеет два типа итераторов: **iterator** и **const_iterator**. Различие между ними заключается в том, что **const_iterator** не позволяет модифицировать элементы вектора.
Функция-член контейнера -- **begin()** возвращает итератор, который ссылается на первый элемент в контейнере (например, **films[0]**). Функция-член контейнера -- **end()** возвращает итератор, который ссылается на элемент "следующий за последним" (например, **films[5]**). Если контейнер пуст, значения, возвращаемые функциями **begin()** и **end()**, эквивалентны. Это обстоятельство может использоваться для проверки наличия элементов в контейнере, хотя для этой цели гораздо удобнее использовать функцию **empty()**.

Итераторы обладают интуитивно понятным синтаксисом, который напоминает синтаксис указателей языка C++. Для перемещения к следующему или предыдущему элементу, можно использовать операторы "**++**" и "**--**", а унарный "*****" -- для получения доступа к элементу контейнера, находящемуся в позиции итератора.
Если необходимо отыскать некоторый элемент в векторе, можно воспользоваться функцией STL -- **find()**:

```
  vector<Film>::iterator it = find(films.begin(), films.end(),
                                   Film(4812));
  if (it != films.end())
    films.erase(it);
```

Она возвращает итератор, указывающий на первый встретившийся элемент вектора, отвечающий критериям поиска (элементы контейнера сравниваются перегруженным **operator==()** с последним аргументом функции). Определение функции находится в заголовке <algorithm>, где вы найдете множество других шаблонных функций. Qt предоставляет аналоги некоторых из них, правда под другими именами (например, **qFind()**). Вы можете использовать их, если не желаете пользоваться библиотекой STL.
Сортировка элементов вектора может быть произведена функцией **sort()**:

```
  sort(films.begin(), films.end());
```

Для сравнения элементов вектора она использует оператор "**<**", если явно не указывается другая функция сравнения. На отсортированных векторах, для поиска некоторого элемента может использоваться функция **binary_search()**. Она дает результат, аналогичный **find()** (при условии, что в векторе нет двух фильмов с одинаковыми числовыми идентификаторами), но при этом работает намного быстрее.

```
int id = 1817;
if (binary_search(films.begin(), films.end(), Film(id)))
  cerr << "Found " << id << endl;
```

В позицию итератора, с помощью функции **insert()**, может быть вставлен новый элемент или удален существующий, с помощью функции **erase()**:

```
films.erase(it);
```

Элементы, которые следуют за удаляемым будут перемещены на одну позицию влево (или выше, если хотите) и размер вектора будет уменьшен на 1 элемент.

11.2 Списки

Список (если быть более точным -- связанный список) -- это структура данных, которая может хранить элементы списка в областях памяти с произвольным размещением. В отличие от векторов, списки не предоставляют такого количества способов произвольного доступа к своим элементам, как векторы, но с другой стороны, функции **insert()** и **erase()** исполняются очень быстро.
Большинство алгоритмов работы с векторами не применимы к спискам, в особенности это относится к функциям **sort()** и **binary_search()**, по причине того, что списки не обладают возможностью быстрого доступа к произвольному элементу. Сортировка STL-списков выполняется функцией **sort()**.

Рисунок 11.2. Список экземпляров класса Film.

Класс списка в STL называется как **std::list<T>**, и определен в заголовке <**list**>. Например:

```
list<Film> films;
```

Эквивалентный класс в Qt -- **QValueList<T>**;:

```
QValueList<Film> films;
```

Новый элемент может быть добавлен в список вызовом функции **push_back()** или **insert()**. В отличие от векторов, вставка элемента в начало или в середину списка выполняется так же быстро, как и добавление элемента в конец списка.
В STL, списки не имеют оператора "**[ ]**", поэтому, для выбора нужного элемента приходится использовать итераторы. (Списки Qt поддерживают оператор "**[ ]**", но на больших списках он может работать очень медленно.) Синтаксис и порядок использования аналогичен векторам, например:

```
list<Film>::const_iterator it = films.begin();
while (it != films.end()) {
  cerr << (*it).title().ascii() << endl;
  ++it;
```

```
  }
```

Списки предоставляют практически тот же набор функций, что и векторы, включая **empty()**, **size()**, **erase()** и **clear()**. Функция **find()** так же имеется.
Некоторые функции Qt возвращают **QValueList<T>**. Если есть необходимость пройти в цикле по списку, то нужно создать копию списка и выполнить проход по копии. Ниже представлен пример корректной работы со списком **QValueList<int>**, который возвращает **QSplitter::sizes()**:

```
QValueList<int> list = splitter->sizes();
QValueList<int>::const_iterator it = list.begin();
while (it != list.end()) {
  do_something(*it);
  ++it;
}
```

Следующий код -- неправильный:

```
// НЕВЕРНО!
QValueList<int>::const_iterator it = splitter->sizes().begin();
while (it != splitter->sizes().end()) {
  do_something(*it);
  ++it;
}
```

Это происходит потому, что **QSplitter::sizes()** возвращает результат по значению. Если не сохранить его копию, C++ автоматически удалит его еще до того как начнется итерация. Всегда создавайте копию контейнера, возвращаемого по значению, когда требуется получить для него итератор.
На первый взгляд, операция создания копии может показаться ресурсоемкой, но это не так, благодаря тому, что Qt использует оптимизацию, которая называется **implicit sharing** (неявное совместное использование данных). Суть оптимизации заключается в том, что фактически, операция копирования может и не производиться, не смотря на то, что программа запросила ее.
Класс **QStringList**, широко используемый в Qt, это дочерний класс **QValueList<QString>**. Он расширяет набор методов предка своими, дополнительными функциями, которые делают этот класс очень мощным инструментом. Более подробно мы расскажем о нем в последнем разделе этой главы.

11.3 Словари (map)

Словари предназначены для хранения произвольного количества элементов, в виде пар "ключ-значение". Причем к "ключам" предъявляется требование уникальности. Словари обладают широкими возможностями доступа к произвольным элементам и незначительными накладными расходами на операцию добавления нового элемента. Если в словарь вставляется новое значение по существующему ключу, то оно затирает старое значение в паре "ключ-значение".

Рисунок 11.3. Словарь экземпляров класса Film.

Поскольку словари хранят элементы в виде "ключ-значение", то принципы работы со словарями несколько отличаются от тех, что используются при работе со списками и векторами. Ниже приводится версия класса **Film**, которая будет использоваться для иллюстрации работы со словарем:

```
class Film
{
public:
  Film(const QString &title = "", int duration = 0);
```

```
  QString title() const { return myTitle; }
  void setTitle(const QString &title) { myTitle = title; }
  int duration() const { return myDuration; }
  void setDuration(int minutes) { myDuration = minutes; }

private:
  QString myTitle;
  int myDuration;
};

Film::Film(const QString &title, int duration)
{
  myTitle = title;
  myDuration = duration;
}
```

В этой версии отсутствует числовой идентификатор фильма, поскольку теперь он будет использоваться в качестве "ключа" в словаре. Кроме того, здесь отсутствуют операторы сравнения -- словари изначально упорядочивают элементы по ключу, но не по значению.
Класс словаря в STL определен под именем **std::map<K, T>**, в файле заголовка <**map**>. Ниже приводится пример объявления словаря, с целыми значениями в качестве ключей и **Film** -- в качестве значения:

```
  map<int, Film> films;
```

Эквивалент в Qt -- **QMap<K, T>**:

```
  QMap<int, Film> films;
```

Наиболее естесственный способ заполнения словарей -- присваивать значение по заданному ключу:

```
  films[4812] = Film("A Hard Day's Night", 85);
  films[5051] = Film("Seven Days to Noon", 94);
  films[1301] = Film("Day of Wrath", 105);
  films[9227] = Film("A Special Day", 110);
  films[1817] = Film("Day for Night", 116);
```

Итератор словаря предоставляет возможность доступа к паре "ключ-значение". Ключ извлекается с помощью **(*it).first**, а значение -- **(*it).second**:

```
  map<int, Film>::const_iterator it = films.begin();
  while (it != films.end()) {
    cerr << (*it).first << ": "
         << (*it).second.title().ascii() << endl;
    ++it;
  }
```

Большинство компиляторов допускают запись в виде **it->first** и **it->second**, но более переносимый вариант, все таки: **(*it).first** и **(*it).second**.
Итераторы словарей в Qt несколько отличаются от итераторов словарей в STL. В Qt ключ можно получить с помощью **it.key()**, а значение -- **it.data()**:

```
  QMap<int, Film>::const_iterator it = films.begin();
  while (it != films.end()) {
    cerr << it.key() << ": " << it.data().title().ascii() < endl;
    ++it;
  }
```

При обходе словаря в цикле, элементы словаря всегда упорядочены по значению ключа.
Для доступа к значениям словаря и их изменения может использоваться оператор "**[ ]**", однако, при попытке получить значение по несуществующему в словаре ключу, будет создан новый элемент словаря с заданным ключом и пустым значением. Чтобы избежать случайного создания пустых элементов, используйте функцию **find()**, чтобы получить искомый элемент:

```
  map<int, Film>::const_iterator it = films.find(1817);
```

```
  if (it != films.end())
    cerr << "Found " << (*it).second.title().ascii() << endl;
```

Эта функция вернет итератор **end()**, если ключ отсутствует в словаре.
В примере выше, в качестве ключа использовались целые числа, однако, для этих целей могут использоваться и другие типы. Наиболее популярный -- **QString**, например:

```
map<QString, QString> actorToNationality;
actorToNationality["Doris Day"] = "American";
actorToNationality["Greta Garbo"] = "Swedish";
```

Если необходимо хранить несколько значений с одинаковыми ключами, используйте **multimap<K, T>**.
Если необходимо хранить одни только ключи, используйте **set<K>** или **multiset<K>**. Qt не имеет классов, эквивалентных приведенным.
Класс **QMap<K, T>** имеет несколько дополнительных функций, особенно удобных при работе с небольшими наборами данных. Функции **QMap<K, T>::keys()** и **QMap<K, T>::values()** возвращают списки **QValueList** ключей и значений словаря.

11.4 Контейнеры указателей

Кроме STL-подобных контейнеров, Qt предоставляет еще целый ряд контейнерных классов. Они были разработаны в начале 90-х годов прошлого века для Qt 1.0, еще до того, как STL стала частью C++, и потому имеют свой характерный синтаксис. Поскольку эти классы оперируют указателями на объекты, их часто называют контейнерами указателей (pointer-based containers), в противоположность более современным контейнерам значений (value-based containers) Qt и STL. В Qt 4 контейнеры указателей еще останутся, для сохранения совместимости, но их использование не будет приветствоваться.

Контейнеры указателей сохраняют свою актуальность лишь благодаря тому, что в Qt 3 еще имеется ряд немаловажных функций, которые работают с ними. Один пример мы приводили в Главе 3, когда выполняли итерации по виджетам, второй -- в Главе 6, когда выполняли итерации по окнам в MDI-приложении.

Основными контейнерами указателей являются классы **QPtrVector<T>**, **QPtrList<T>**, **QDict<T>**, **QAsciiDict<T>**, **QIntDict<T>** и **QPtrDict<T>**.
Класс **QPtrVector<T>** предназначен для хранения вектора указателей. Ниже приводится пример создания **QPtrVector<Film>** с пятью элементами:

```
QPtrVector<Film> films(5);
films.setAutoDelete(true);
films.insert(0, new Film(4812, "A Hard Day's Night", 85));
films.insert(1, new Film(5051, "Seven Days to Noon", 94));
films.insert(2, new Film(1301, "Day of Wrath", 105));
films.insert(3, new Film(9227, "A Special Day", 110));
films.insert(4, new Film(1817, "Day for Night", 116));
```

Класс **QPtrVector<T>** не имеет функции **append()**, поэтому приходится явно указывать индекс для добавляемых элементов. В этом примере использована первая версия класса **Film**, которая содержит переменную-член -- числовой идентификатор фильма.
Контейнеры указателей в Qt обладают одним замечательным свойством -- "auto-delete" (автоматическое удаление). Если автоудаление разрешено, Qt становится владельцем всех объектов, вставляемых в контейнер и удаляет их автоматически, когда удаляется контейнер (или при вызове методов **remove()** и **clear()**).
Для исключения элемента из вектора, должна вызываться функция **remove()**, с указанием индекса удаляемого элемента:

```
films.remove(2);
```

Эта функция не изменяет размер вектора, она просто обнуляет указатель с заданным индексом. Если разрешено автоудаление, то автоматически удаляется объект, на который указывал элемент вектора.
Чтобы обойти все элементы вектора в цикле, можно просто использовать индексы:

```
for (int i = 0; i < (int)films.count(); ++i) {
  if (films[i])
    cerr << films[i]->title().ascii() << endl;
}
```

В данном примере сначала выполняется проверка указателя (указатель не должен быть пустым), а затем выполняются все необходимые действия над указателем.
Класс **QPtrList<T>** предназначен для хранения списка указателей. Добавление новых элементов в **QPtrList<T>** производится функциями **append()**, **prepend()** и **insert()**:

```
QPtrList<Film> films;
films.setAutoDelete(true);
films.append(new Film(4812, "A Hard Day's Night", 85));
films.append(new Film(5051, "Seven Days to Noon", 94));
```

Список указателей имеет "текущий" элемент, значение которого изменяется функциями навигации по списку, такими как **first()**, **next()**, **prev()** и **last()**. Один из способов выполнения прохода по списку:

```
Film *film = films.first();
while (film) {
  cerr << film->title().ascii() << endl;
  film = films.next();
}
```

Однако списки допускают доступ к элементам по индексу:

```
for (int i = 0; i < (int)films.count(); ++i)
  cerr << films.at(i)->title().ascii() << endl;
```

Третий возможный вариант обхода списка, заключается в использовании **QPtrListIterator<T>**.
Классы **QDict<T>**, **QAsciiDict<T>**, **QIntDict<T>** и **QPtrDict<T>** являются близкими эквивалентами **map<K, T>**. Эти классы так же хранят пары "ключ-значение". Ключ в них может быть представлен одним из четырех типов: **QString**, **const char ***, **int и void ***, в зависимости от типа используемого класса. Поскольку все четыре класса предоставляют одинаковую функциональность, мы рассмотрим только один из них -- **QIntDict<T>**.
Для демонстрации воспользуемся второй версией класса **Film**, которая использовалась ранее, совместно с классом **map<K, T>**.

```
QIntDict<Film> films(101);
films.setAutoDelete(true);
```

Конструктору передается число, используемое классом для определения количества памяти, которую нужно выделить под элементы словаря. Для улучшения производительности, это число должно быть простым и немного больше, чем количество элементов, которое предполагается вставить в словарь. Список простых чисел, меньших 10 000, вы найдете по адресу: *http://doc.trolltech.com/3.2/primes.html*.
Вставка нового элемента выполняется функцией **insert()**, которой передаются ключ и значение:

```
films.insert(4812, new Film("A Hard Day's Night", 85));
films.insert(5051, new Film("Seven Days to Noon", 94));
```

Для доступа к элементу словаря можно использовать функцию **find()** или оператор "**[ ]**". Для удаления элемента -- функцию **remove()**. Для изменения значения, ассоциированного с заданным ключом -- **replace()**.
Если функция **insert()** вызывается несколько раз с одним и тем же ключом, доступ будет иметься только к значению, которое было вставлено последним. При вызове **remove()**, элементы удаляются в обратном порядке. Чтобы избежать вставки нескольких значений с одим и тем же ключом, используйте **replace()** вместо **insert()**.
Обход элементов контейнера может быть выполнен с помощью итератора:

```
QIntDictIterator<Film> it(films);
while (it.current()) {
```

```
    cerr << it.currentKey() << ": "
         << it.current()->title().ascii() << endl;
    ++it;
  }
```

Текущий ключ итератора может быть получен вызовом **currentKey()**, а текущее значение -- функцией **current()**. Порядок следования элементов в словаре не определен.
Для хранения элементов базовых типов языка C++ (**int**, **double** и т.п) и структур, Qt предоставляет специальный, вектор-подобный класс **QMemArray<T>**. В некоторых приложениях он может использоваться напрямую, однако, чаще используются два производных класса **QByteArray** (**QMemArray<char>**) и **QPointArray** (**QMemArray<QPoint>**). Мы уже использовали их несколько раз в предыдущих главах.
Ниже приводится пример создания **QByteArray**:

```
  QByteArray bytes(4);
  bytes[0] = 'A';
  bytes[1] = 'C';
  bytes[2] = 'D';
  bytes[3] = 'C';
```

При создании экземпляра **QMemArray<T>**, необходимо либо сразу указать начальный размер будущего массива, либо вызвать функцию **resize()** после создания. Доступ к элементам массива выполняется с помощью оператора "**[ ]**":

```
  for (int i = 0; i < (int)bytes.size(); ++i)
    cerr << bytes[i] << endl;
```

Поиск элемента в массиве осуществляется с помощью функции **QMemArray<T>::find()**:

```
  if (bytes.find( A ) != -1)
    cerr << "Found" < endl;
```

Иногда программисты забывают об одной особенности класса **QMemArray<T>** и его производных -- они используют то, что называется explicitly shared (явное совместное использование данных). Это означает, что созданные копии объекта (с помощью конструктора копирования или оператором присваивания) ссылаются на одни и те же данные. Когда данные модифицируются с помощью одного объекта, изменения будут видны в другом. Не следует путать явное совместное использование данных (explicitly shared) с неявным совместным использованием данных (implicitly shared), которое лишено данной проблемы.
Избежать описанной проблемы несложно, для этого достаточно выполнить полное копирование объекта вызовом **copy()**:

```
  duplicate = bytes.copy();
```

Теперь два объекта будут ссылаться на различные наборы данных.
Скорее всего, в Qt 4, предпочтение будет отдано классу **QValueVector<T>**, а классы **QByteArray** и **QPointArray** станут его производными.

11.5 Классы QString и QVariant

Строки используются практически во всех программах ничуть не реже других типов.
Язык C++ предоставляет два типа строк: традиционные строки языка C -- массивы символов, завершающиеся символом **'\0'** и класс **string**. Qt предоставляет гораздо более мощный класс **QString**. Он предназначен для хранения строк с 16-ти битными символами Unicode. Unicode содержит наборы символов ASCII и Latin-1 с их обычными числовыми значениями. Но поскольку каждый символ в **QString** представлен 16-ю битами, он может содержать тысячи других символов. Дополнительную информацию об Unicode вы найдете в Главе 15.

Конкатенация двух строк **QString** может выполняться двухместным оператором "**+**" или оператором "**+=**". Ниже приводится пример использования обоих операторов:

```
QString str = "User: ";
str += userName + "\n";
```

Кроме того, имеется функция **QString::append()**, которая идентична по своему действию оператору "**+=**":

```
str = "User: ";
str.append(userName);
str.append("\n");
```

И совершенно иной подход к объединению строк состоит в использовании функции **QString::sprintf()**:

```
str.sprintf("%s %.1f%%", "perfect competition", 100.0);
```

Она поддерживает тот же набор спецификаторов формата, что и библиотечная функция **sprintf()**. В примере выше, в строку **str** будет записана строка "perfect competition 100.0%".

Еще один способ "сборки" строки из других строк и чисел -- использовать **arg()**:

```
str = QString("%1 %2 (%3s-%4s)")
      .arg("permissive").arg("society").arg(1950).arg(1970);
```

В этом примере "%1" будет заменено словом "permissive", "%2" -- "society", "%3" -- "1950" и "%4" -- "1970". В результате получится строка "permissive society (1950s-1970s)". Класс имеет несколько перегруженных функций **arg()** для обработки различных типов данных. Некоторые из них имеют дополнительные параметры, управляющие длиной выходной строки, базой системы счисления и точностью представления чисел с плавающей точкой. В большинстве случаев **arg()** представляет лучшее решение, чем **sprintf()**, потому что она более безопасна, полностью поддерживает Unicode и позволяет переводчикам изменять порядок следования параметров "**%n**".
QString позволяет преобразовывать числа в их строковое представление, с помощью статической функции **QString::number()**:

```
str = QString::number(59.6);
```

или с помощью **QString::setNum()**:

```
str.setNum(59.6);
```

Обратное преобразование может быть выполнено функциями **toInt()**, **toLongLong()**, **toDouble()** и т.д., например:

```
bool ok;
double d = str.toDouble(&ok);
```

Эти функции могут принимать необязательный аргумент типа **bool**, в котором возвращается признак успеха преобразования. Если преобразование не может быть выполнено, они всегда возвращают **0**.
Зачастую возникает ситуация, когда необходимо извлечь часть строки. Функция **mid()** возвращает подстроку заданной длины, начиная с заданной позиции в исходной строке. Например, следующий код выводит строку "pays":

```
QString str = "polluter pays principle";
cerr << str.mid(9, 4).ascii() << endl;
```

Если опустить второй аргумент (или передать в качестве второго аргумента число -1), функция вернет подстроку, начиная с заданной позиции и до конца исходной строки. Например, следующий код выведет строку "pays principle":

```
QString str = "polluter pays principle";
cerr << str.mid(9).ascii() << endl;
```

Дополнительно имеются функции **left()** и **right()**. Они обе принимают количество символов **n** и возвращают первые или последние **n** символов исходной строки, соответственно. Например, следующий код выведет строку "polluter principle":

```
QString str = "polluter pays principle";
cerr << str.left(8).ascii() << " " << str.right(9).ascii()
     << endl;
```

Если нужно выполнить проверку -- начинается ли или заканчивается ли строка определенной комбинацией символов, для этих целей существуют функции **startsWith()** и **endsWith()**:

```
if (uri.startsWith("http:") && uri.endsWith(".png"))
  ...
```

Это гораздо быстрее и проще, чем:

```
if (uri.left(5) == "http:" && uri.right(4) == ".png")
  ...
```

Оператор сравнения строк "**==**" чувствителен к регистру символов. Для выполнения регистронезависимого сравнения, можно воспользоваться функциями **upper()** или **lower()**, например:

```
if (fileName.lower() == "readme.txt")
  ...
```

Для замены одной подстроки в строке другой подстрокой, используйте функцию **replace()**:

```
QString str = "a sunny day";
str.replace(2, 5, "cloudy");
```

в результате получится строка "a cloudy day". То же самое действие может выполнено с помощью функций **remove()** и **insert()**:

```
str.remove(2, 5);
str.insert(2, "cloudy");
```

В первой строке удаляется пять символов, начиная со 2-й позиции, в результате получается строка "a day" (с двумя пробелами), затем, во второую позицию вставляется слово "cloudy".

Существуют перегруженные версии функции **replace()**, которые заменяют все вхождения первого аргумента на второй. Например, чтобы заменить все символы '**&**' в строке на "**&**":

```
str.replace("&", "&amp;");
```

Очень часто возникает необходимость выбросить из начала и конца строки все лишние пробельные символы (такие как: пробелы, символы табуляции, символы перевода строки). Для этой цели существует функция **stripWhiteSpace()**:

```
QString str = " BOB \t THE \nDOG \n";
cerr << str.stripWhiteSpace().ascii() << endl;
```

Строка **str** может быть изображена как:

			B	O	B		\t		T	H	E			\n	D	O	G		\n

А результат, возвращаемый функцией **stripWhiteSpace()**, как:

B	O	B		\t		T	H	E			\n	D	O	G

Для удаления лишних пробельных символов, как на концах строки, так и внутри, предназначена функция **simplifyWhiteSpace()**:

```
QString str = " BOB \t THE \nDOG \n";
cerr << str.simplifyWhiteSpace().ascii() << endl;
```

Результат работы функции будет выглядеть так:

B	O	B		T	H	E		D	O	G

Строки могут быть разбиты на подстроки с помощью функции **QStringList::split()**:

```
QString str = "polluter pays principle";
QStringList words = QStringList::split(" ", str);
```

В этом примере, строка "polluter pays principle" разбивается на три подстроки; "polluter", "pays" и "principle". Функция **split()** может принимать третий необязательный параметр типа **bool**, который определяет -- должны ли игнорироваться пустые подстроки (по-умолчанию) или нет.
Элементы списка **QStringList** могут быть объединены в одну строку, с помощью функции **join()**. В качестве аргумента ей передается строка, которая должна быть вставлена между объединяемыми строками. Например, следующий код демонстрирует, как можно объединить все строки в списке, отсортированном по алфавиту, в единую строку, причем подстроки отделяются друг от друга символом перевода строки:

```
words.sort();
str = words.join("\n");
```

Еще одна немаловажная операция над строками -- определение длины строки. Для этого предназначена функция **length()** и, как вариант, **isEmpty()**, которая возвращает true, если длина строки равна **0**.
QString различает пустые строки и несуществующие (**NULL**) строки. Эти различия корнями уходят в язык программирования C. Чтобы проверить -- существует ли строка, можно вызывать функцию **isNull()**. Для большинства приложений очень важно знать -- содержит ли строка хотя бы один символ. Функция **isEmpty()** вернет **true**, если строка не содержит ни одного символа (пустая или несуществующая строка).
Преобразования, между **const char *** и **QString**, в большинстве случаев выполняются автоматически:

```
str += " (1870)";
```

Этот код добавляет строку типа **const char *** к строке типа **QString**.

В некоторых ситуациях возникает необходимость явно выполнять преобразование между **const char *** и **QString**. Чтобы преобразовать строку **QString** в **const char ***, используйте функцию **ascii()** или **latin1()**. Обратное преобразование может быть выполнено за счет операции приведения типа.

Когда вызываются функции **ascii()** или **latin1()**, или когда выполняется автоматическое преобразование к типу **const char ***, возвращаемая строка принадлежит объекту **QString**. Это означает, что нас не должна беспокоить проблема утечки памяти -- Qt самостоятельно утилизирует память, по мере необходимости. С другой стороны, необходимо проявлять большую осторожность при работе с указателями. Например, если оригинальная версия строки **QString** будет изменена, то ранее полученный указатель на **const char *** может оказаться недопустимым. Если же необходимо сохранить предыдущий вариант строки, то для этих целей можно воспользоваться услугами класса **QByteArray** или **QCString**. Они хранят полную копию данных.
Класс **QString** поддерживает **implicit sharing** (неявное совместное использование данных). Это означает, что на копирование строки уходит времени не больше, чем необходимо для копирования указателя на строку. Собственно копирование производится только тогда, когда выполняется попытка изменить одну из копий. Все это делается автоматически и незаметно для нас.
Вся прелесть неявного совместного использования данных состоит в том, что таким образом оптимизируется скорость выполнения операций и при этом нам не нужно постоянно помнить об этом -- это просто работает!

Qt использует это метод оптимизации и для других классов, включая: **QBrush**, **QFont**, **QPen**, **QPixmap**, **QMap<K, T>**, **QValueList<T>** и **QValueVector<T>**. Что повышает эффективность передачи экземпляров классов по значению, как в виде аргументов функций, так и в виде возвращаемых значений.
C++ -- это строго типизированный язык, однако, иногда возникает необходимость сохранять данные в более общем виде. Самый простой способ -- использовать строки. Например, строки могут хранить текстовые или числовые данные. Qt предоставляет более простой способ работы с переменными -- класс **QVariant**.
Класс **QVariant** может хранить значения многих типов Qt: **QBrush**, **QColor**, **QCursor**, **QDateTime**, **QFont**, **QKeySequence**, **QPalette**, **QPen**, **QPixmap**, **QPoint**, **QRect**, **QRegion**, **QSize** и **QString**. Он так же может хранить контейнеры: **QMap<QString, QVariant>**, **QStringList и QValueList<QVariant>**. Мы уже использовали **QVariant**, когда разрабатывали приложение **Spreadsheet**, в Главе 4, для хранения содержимого ячейки.

Одно из обычных применений класса **QVariant** -- создание словарей (**map**), в которых в качестве ключа используются строки, а в качестве значений -- экземпляры класса **QVariant**. Как правило, информация о конфигурации приложения сохраняется и загружается с помощью **QSettings**, но иногда приложения могут обслуживать настройки напрямую, например, сохраняя их в базе данных. **QMap<QString, QVariant>** идеально подходит в таких ситуациях:

```
QMap<QString, QVariant> config;
config["Width"] = 890;
config["Height"] = 645;
config["ForegroundColor"] = black;
config["BackgroundColor"] = lightGray;
config["SavedDate"] = QDateTime::currentDateTime();
QStringList files;
files << "2003-05.dat" << "2003-06.dat" << "2003-07.dat";
config["RecentFiles"] = files;
```

11.5.1 Принцип Действия Неявного Совместного Использования Данных

Механизм неявного совместного использования данных работает полностью автоматически и незаметно для нас. Таким образом, когда мы используем классы, поддерживающие этот механизм, мы не должны писать дополнительный код, который поддерживал бы его работу. Но знать принцип его действия вам все таки необходимо, поэтому рассмотрим простенький пример и исследуем -- что же скрыто от наших глаз.

```
QString str1 = "Humpty";
QString str2 = str1;
```

Здесь в переменную **str1** записывается строка "Humpty" и затем выполняется присваивание переменной **str2**. С этого момента обе переменные указывают на одну и ту же структуру данных в памяти (типа **QStringData**). Вместе с символами строки она хранит счетчик ссылок, который содержит количество объектов, ссылающихся на нее. Поскольку и **str1**, и **str2** ссылаются на одни и те же данные, то счетчик ссылок равен 2.

```
str2[0] = 'D';
```

Когда выполняется изменение содержимого переменной **str2**, то прежде всего создается полная копия данных, таким образом, теперь **str1** и **str2** ссылаются на различные структуры и все изменения будут производиться над их собственными копиями данных. Счетчик ссылок переменной **str1** ("Humpty") теперь стал равен **1** и счетчик ссылок переменной **str2** ("Dumpty") так же стал равен **1**. Когда счетчик ссылок равен **1**, это означает, что данные используются только одним объектом.

```
str2.truncate(4);
```

Если теперь выполнить модификацию переменной **str2**, то никакого копирования производиться уже не будет, потому что счетчик ссылок равен **1**. Функция **truncate()** будет оперировать с данными, принадлежащими переменной **str2**, и счетчик ссылок останется равным **1**.

str1 = str2;

После такого присваивания, счетчик ссылок переменной **str1** станет равным **0**, это означает, что строка "Humpty" больше не нужна. В этом случае память, ранее занимаемая переменной **str1**, будет освобождена. Теперь обе переменные будут ссылаться на строку "Dump", а счетчик ссылок станет равным **2**.Создание классов, использующих оптимизацию неявного совместного использования данных, выполняется довольно просто. В ежеквартальнике Qt Quarterly, в статье "Data Sharing with Class" (*http://doc.trolltech.com/qq/qq02-data-sharing-with-class.html*) описывается -- как это сделать. Обход в цикле элементов словаря, который хранит значения в вариантном виде, может оказаться не такой простой задачей, если некоторые из значений являются контейнерами. Чтобы проверить тип вариантного значения вам придется воспользоваться функцией **type()**:

```
QMap<QString, QVariant>::const_iterator it = config.begin();
while (it != config.end()) {
  QString str;
  if (it.data().type() == QVariant::StringList)
    str = it.data().toStringList().join(", ");
  else
    str = it.data().toString();
  cerr << it.key().ascii() << ": " << str.ascii() << endl;
  ++it;
}
```

С помощью **QVariant** можно создавать довольно сложные структуры данных, хранящие значения контейнерного типа:

```
QMap<QString, QVariant> price;
price["Orange"] = 2.10;
price["Pear"].asMap()["Standard"] = 1.95;
price["Pear"].asMap()["Organic"] = 2.25;
price["Pineapple"] = 3.85;
```

В этом примере был создан словарь со строковыми ключами (название продукта) и значениями типа **double** (цена) или типа **QMap**. Словарь верхнего уровня содержит три ключа: "Orange", "Pear" и "Pineapple". Значение, связанные с ключом "Pear" -- это словарь с двумя ключами ("Standard" и "Organic").

Возможность создания структур данных, подобных этой, может показаться очень соблазнительной, так как можно структурировать данные по своему усмотрению. Но удобство **QVariant** -- вещь дорогостоящая. Ради удобочитаемости придется пожертвовать быстродействием.

12 Базы данных

Модуль SQL, в библиотеке Qt, предоставляет независимый от типа платформы и базы данных интерфейс, для доступа к базам данных SQL, и набор классов, обеспечивающих взаимодействие пользовательского интерфейса с базами данных.
Глава начинается с демонстрационного примера, который показывает, как установить соединение с базой данных и как выполнить произвольный SQL-код. Во втором и третьем разделах мы подробнее остановимся на том, как предоставить пользователю возможность просматривать и изменять наборы данных, используя **QDataTable** -- для просмотра данных в табличном виде, и **QSqlForm** -- в виде формы.

12.1 Установление соединения и выполнение запроса

Прежде чем выполнить запрос к базе данных, для начала необходимо установить с ней соединение. Как правило, установление соединения с базой данных выполняется в виде отдельной функции, которую приложение вызывает на запуске, например:

```
bool createConnection()
{
  QSqlDatabase *db = QSqlDatabase::addDatabase("QOCI8");
  db->setHostName("mozart.konkordia.edu");
  db->setDatabaseName("musicdb");
  db->setUserName("gbatstone");
  db->setPassword("T17aV44");
  if (!db->open()) {
    db->lastError().showMessage();
    return false;
  }
  return true;
}
```

Первым делом, вызовом **QSqlDatabase::addDatabase()**, создается экземпляр класса **QSqlDatabase**. Аргумент функции определяет драйвер базы данных, используемый для доступа к ней. В данном случае -- это драйвер Oracle. Коммерческая версия Qt включает в себя следующий набор драйверов: QODBC3 (ODBC), QOCI8 (Oracle), QTDS7 (Sybase Adaptive Server), QPSQL7 (PostgreSQL), QMYSQL3 (MySQL), and QDB2 (IBM DB2). В некоммерческие версии Qt входит только часть этого набора. [7]

Затем указывается сетевое имя сервера баз данных, имя базы данных, имя пользователя и пароль, после чего выполняется попытка установить соединение. Если функция **open()** завершилась неудачей -- выводится сообщение об ошибке, с помощью **QSqlError::showMessage()**.
Обычно функция, подобная **createConnection()** вызывается из функции **main()**:

```
int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  if (!createConnection())
    return 1;
  ...
  return app.exec();
}
```

После установления соединения, посредством **QSqlQuery**, можно выполнять SQL-запросы к базе данных. Например, следующий код выполняет SQL-предложение -- **SELECT**:

```
  QSqlQuery query;
  query.exec("SELECT title, year FROM cd WHERE year >= 1998");
```

После вызова функции **exec()**, можно просматривать полученный набор данных:

```
  while (query.next()) {
    QString title = query.value(0).toString();
    int year = query.value(1).toInt();
```

```
    cerr << title.ascii() << ": " << year << endl;
  }
```

Первый вызов **next()** позиционирует **QSqlQuery** на первую запись в наборе данных. Последующие вызовы **next()** передвигают указатель на следующую запись и так до тех пор, пока не будет достигнут конец набора. В этой точке **next()** вернет **false**.

Функция **value()** возвращает значение поля в виде **QVariant**. Поля нумеруются, начиная с **0**, в порядке их следования в предложении **SELECT**. Класс **QVariant** может хранить огромное количество типов языка C++ и Qt, в том числе **int** и **QString**. Различные типы данных, которые могут храниться в базе данных переводятся в соответствующие типы C++ и Qt, и сохраняются в виде **QVariant**. Например, **VARCHAR** представляется в виде **QString**, а **DATETIME** -- как **QDateTime**.

Класс **QSqlQuery** предоставляет целый набор функций для навигации по набору данных: **first()**, **last()**, **prev()**, **seek()** и **at()**. Они очень удобны в использовании, но на некоторых базах данных могут оказаться довольно медлительными и ресурсоемкими. С целью оптимизациии, при работе с большими наборами данных, можно вызвать **QSqlQuery::setForwardOnly(true)**, перед **exec()**, а затем выполнять просмотр набора данных с помощью **next()**, правда в этом случае мы получаем, так называемые, однонаправленные наборы данных, т.е. такие наборы, навигация по которым может осуществляться только вперед, с помощью **next()**.

Чуть выше говорилось о том, что SQL-запрос передается как аргумент функции **exec()**, но текст запроса может передаваться напрямую, конструктору **QSqlQuery**:

```
  QSqlQuery query("SELECT title, year FROM cd WHERE year >= 1998");
```

Проверка на наличие ошибок и выдача сообщения могут быть выполнены таким образом:

```
  if (!query.isActive())
    query.lastError().showMessage();
```

Выполнение предложения **INSERT** ничуть не сложнее, чем **SELECT**:

```
  QSqlQuery query("INSERT INTO cd (id, artistid, title, year) "
                  "VALUES (203, 102, 'Living in America', 2002)");
```

После выполнения такого запроса, **QSqlQuery::numRowsAffected()** возвращает количество записей, подвергшихся изменению (или -1, если база данных не предусматривает поставку такой информации).
В случае необходимости вставить в запрос значения переменных или когда нежелательно, или невозможно перевести аргументы предложения **INSERT** в строковый вид, можно построить параметризованный запрос, с помощью функции **prepare()**. Текст параметризованного запроса, вместо реальных значений содержит параметры, которые заполняются фактическими значениями после создания запроса. Qt поддерживает Oracle-подобный и ODBC-подобный стили именования параметров для всех типов баз данных. В примере ниже показано использование Oracle-подобного стиля именования:

```
  QSqlQuery query(db);
  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", QString("Living in America"));
  query.bindValue(":year", 2002);
  query.exec();
```

Теперь тот же самый пример, но в стиле ODBC:

```
  QSqlQuery query(db);
  query.prepare("INSERT INTO cd (id, artistid, title, year) "
                "VALUES (?, ?, ?, ?)");
  query.addBindValue(203);
```

```
query.addBindValue(102);
query.addBindValue(QString("Living in America"));
query.addBindValue(2002);
query.exec();
```

После создания запроса, вызовом **prepare()**, параметры запроса заполняются фактическими значениями, с помощью функции **bindValue()** или **addBindValue()**, после чего запрос исполняется вызовом **exec()**. Параметризованные запросы можно выполнять в цикле. Перед началом цикла создается запрос, а в теле цикла производится заполнение параметров новыми значениями и исполнение запроса.
Параметризованные запросы очень часто используются в тех случаях, когда в базу данных нужно записать двоичные данные или строки, которые содержат символы из наборов, не принадлежащих диапазону **ASCII** или **Latin-1**. Для баз данных, которые поддерживают **Unicode**, Qt использует эту кодировку символов, в других случаях выполняется преобразование строк в соответствующую кодировку.
Qt поддерживает механизм транзакций для баз данных, в которых он присутствует. Для запуска транзакции вызывается метод объекта **QSqlDatabase -- transaction()**. Для завершения транзакции вызывается либо функция **commit()**, либо **rollback()**. Например, выполним поиск по внешнему ключу и вставим запись в таблицу в рамках транзакции:

```
QSqlDatabase::database()->transaction();
QSqlQuery query;
query.exec("SELECT id FROM artist WHERE name = 'Gluecifer'");
if (query.next()) {
  int artistId = query.value(0).toInt();
  query.exec("INSERT INTO cd (id, artistid, title, year) "
             "VALUES (201, " + QString::number(artistId)
             + ", 'Riding the Tiger', 1997)");
}
QSqlDatabase::database()->commit();
```

Функция **QSqlDatabase::database()** возвращает указатель на объект **QSqlDatabase**, который был создан в **createConnection()**. Если транзакция не может быть запущена, **QSqlDatabase::transaction()** возвращает **false**.
Некоторые базы данных не поддерживают механизм транзакций. В этом случае, функции **transaction()**, **commit()** и **rollback()** не выполняют никаких действий. Наличие поддержки механизма транзакций, той или иной базой данных, можно проверить с помощью метода **hasFeature()**, объекта **QSqlDriver**, ассоциированного с базой данных:

```
QSqlDriver *driver = QSqlDatabase::database()->driver();
if (driver->hasFeature(QSqlDriver::Transactions))
  ...
```

В примерах выше рассматривались случаи с единственным подключением к базе данных. Однако ничто не мешает нам создать и второе, и третье и т.д. соединения. В этом случае необходимо просто передать имя соединения, вторым аргументом в функцию **addDatabase()**:

```
QSqlDatabase *db = QSqlDatabase::addDatabase("QPSQL7", "OTHER");
db->setHostName("saturn.mcmanamy.edu");
db->setDatabaseName("starsdb");
db->setUserName("gilbert");
db->setPassword("ixtapa6");
```

Чтобы потом получить указатель на объект **QSqlDatabase**, достаточно просто передать имя соединения в функцию **QSqlDatabase::database()**:

```
QSqlDatabase *db = QSqlDatabase::database("OTHER");
```

Для исполнения запросов через эти соединения, необходимо передать объект **QSqlDatabase** конструктору **QSqlQuery**:

```
QSqlQuery query(db);
query.exec("SELECT id FROM artist WHERE name = 'Mando Diao'");
```

Каждое соединение с базой данных может поддерживать только одну активную транзакцию, поэтому множественные подключения могут оказаться полезными в том случае, когда необходимо одновременно запустить несколько транзакций. При использовании нескольких соединений, в приложении по прежнему имеется одно неименованное соединение, которое используется по-умолчанию объектами **QSqlQuery**, если им явно не указать с каким соединением они должны работать.
В дополнение к **QSqlQuery**, Qt предоставляет класс **QSqlCursor**, производный от **QSqlQuery**. Этот класс расширяет функциональность предка большим числом дополнительных методов, которые позволяют отказаться от написания SQL-запросов для наиболее употребимых SQL-операций, таких как: **SELECT**, **INSERT**, **UPDATE** и **DELETE**. Кроме того **QSqlCursor** выступает в роли посредника между **QDataTable** и базой данных. Далее, в этом разделе мы будем говорить о **QSqlCursor**, а в следующем разделе покажем, как можно использовать **QDataTable**, для представления наборов данных в табличной форме.
Следующий пример демонстрирует выполнение SQL-запроса -- **SELECT**:

```
QSqlCursor cursor("cd");
cursor.select("year >= 1998");
```

Эквивалентный вариант с использованием **QSqlQuery**:

```
QSqlQuery query("SELECT id, artistid, title, year FROM cd "
                "WHERE year >= 1998");
```

Навигация по набору данных выполняется точно так же, как и в **QSqlQuery**, за одним маленьким исключением -- теперь, вместо порядкового номера поля, функции **value()** можно передать его имя:

```
while (cursor.next()) {
  QString title = cursor.value("title").toString();
  int year = cursor.value("year").toInt();
  cerr << title.ascii() << ": " << year << endl;
}
```

Для вставки записи в таблицу, предварительно нужно создать новую запись **QSqlRecord**, вызовом **primeInsert()**, а затем, для каждого из полей, вызвать **setValue()**. После всего этого можно выполнить вставку функцией **insert()**:

```
QSqlCursor cursor("cd");
QSqlRecord *buffer = cursor.primeInsert();
buffer->setValue("id", 113);
buffer->setValue("artistid", 224);
buffer->setValue("title", "Shanghai My Heart");
buffer->setValue("year", 2003);
cursor.insert();
```

Чтобы изменить запись -- нужно позиционировать **QSqlCursor** на запись, которая должна подвергнуться изменениям (например, с помощью **select()** и **next()**). Получить указатель на **QSqlRecord**, вызовом **primeUpdate()**. После этого записать новые значения функцией **setValue()** и вызвать **update()**, чтобы отправить сделанные изменения в базу данных:

```
QSqlCursor cursor("cd");
cursor.select("id = 125");
if (cursor.next()) {
  QSqlRecord *buffer = cursor.primeUpdate();
  buffer->setValue("title", "Melody A.M.");
  buffer->setValue("year", buffer->value("year").toInt() + 1);
  cursor.update();
}
```

Процедура удаления записи похожа на процедуру изменения:

```
QSqlCursor cursor("cd");
cursor.select("id = 128");
```

```
if (cursor.next()) {
  cursor.primeDelete();
  cursor.del();
}
```

Классы **QSqlQuery** и **QSqlCursor** реализуют интерфейс между Qt и базами данных SQL. В следующих двух разделах мы покажем как они могут использоваться в приложениях с графическим интерфейсом, которые позволяют пользователю просматривать и изменять наборы данных, хранящиеся в базе.

12.2 Представление данных в табличной форме

Класс **QDataTable** -- это ориентированный на работу с базами данных виджет, наследник **QTable**. Взаимодействие **QDataTable** с базой данных осуществляется посредством **QSqlCursor**. В этом разделе мы рассмотрим два диалога, которые используют виджет **QDataTable**. Диалоги, работающие с **QSqlForm** будут представлены в следующем разделе.
Приложение, рассматриваемое здесь, работает с тремя таблицами, которые определены следующим образом:

```
CREATE TABLE artist (
    id INTEGER PRIMARY KEY,
    name VARCHAR(40) NOT NULL,
    country VARCHAR(40));

CREATE TABLE cd (
    id INTEGER PRIMARY KEY,
    artistid INTEGER NOT NULL,
    title VARCHAR(40) NOT NULL,
    year INTEGER NOT NULL,
    FOREIGN KEY (artistid) REFERENCES artist);

CREATE TABLE track (
    id INTEGER PRIMARY KEY,
    cdid INTEGER NOT NULL,
    number INTEGER NOT NULL,
    title VARCHAR(40) NOT NULL,
    duration INTEGER NOT NULL,
    FOREIGN KEY (cdid) REFERENCES cd);
```

Некоторые базы данных не поддерживают внешние ключи. В этом случае вам следует удалить предложения **FOREIGN KEY**. Пример останется работоспособным, но база данных не сможет соблюдать ссылочную целостность данных.

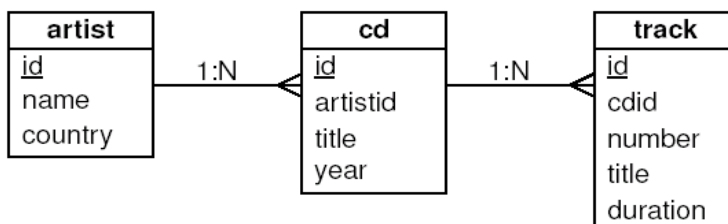


Рисунок 12.1. Таблицы приложения CD Collection.

Первым будет класс диалога, который позволит пользователю редактировать список исполнителей. С его помощью пользователь сможет добавлять, изменять или удалять сведения об исполнителях, выбирая соответствующие пункты контекстного меню **QDataTable**. Внесенные изменения будут записываться в базу данных, по нажатию кнопки **Update**.

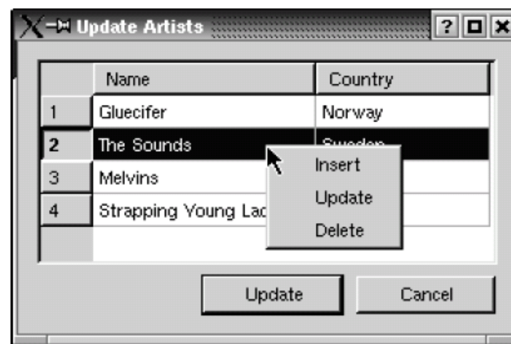


Рисунок 12.2. Диалог ArtistForm.

Определение класса диалога:

```
class ArtistForm : public QDialog
{
  Q_OBJECT
public:
  ArtistForm(QWidget *parent = 0, const char *name = 0);

protected slots:
  void accept();
  void reject();

private slots:
  void primeInsertArtist(QSqlRecord *buffer);
  void beforeInsertArtist(QSqlRecord *buffer);
  void beforeDeleteArtist(QSqlRecord *buffer);

private:
  QSqlDatabase *db;
  QDataTable *artistTable;
  QPushButton *updateButton;
  QPushButton *cancelButton;
};
```

Слоты **accept()** и **reject()** унаследованы от **QDialog**.

```
ArtistForm::ArtistForm(QWidget *parent, const char *name)
    : QDialog(parent, name)
{
  setCaption(tr("Update Artists"));

  db = QSqlDatabase::database("ARTIST");
  db->transaction();

  QSqlCursor *artistCursor = new QSqlCursor("artist", true, db);
  artistTable = new QDataTable(artistCursor, false, this);
  artistTable->addColumn("name", tr("Name"));
  artistTable->addColumn("country", tr("Country"));
  artistTable->setAutoDelete(true);
  artistTable->setConfirmDelete(true);
  artistTable->setSorting(true);
  artistTable->refresh();

  updateButton = new QPushButton(tr("Update"), this);
  updateButton->setDefault(true);
  cancelButton = new QPushButton(tr("Cancel"), this);
```

В конструкторе **ArtistForm** запускается транзакция для соединения под именем "**ARTIST**". Затем создается **QSqlCursor**, для таблицы artist в базе данных, и **QDataTable**, которая будет отображать содержимое таблицы.
Второй аргумент конструктора **QSqlCursor** -- это флаг "автозаполнение". Если в этом аргументе передать **true**, **QSqlCursor** будет загружать информацию о каждом из полей в таблице.

Второй аргумент конструктора **QDataTable** -- так же флаг "автозаполнение". В случае **true**, **QDataTable** будет автоматически создавать колонки для каждого из полей в **QSqlCursor**. В нашем примере этот флаг передается со значением **false** и с помощью **addColumn()** в виджет добавляются две колонки, соответствующие полям name и country.

Владение объектом **QSqlCursor** передается виджету **QDataTable**. Вызовом **setAutoDelete()** устанавливается режим автоматического удаления записей, средствами **QDataTable**, таким образом нам не нужно будет писать дополнительный код, удаляющий записи из таблицы. Вызовом **setConfirmDelete()** устанавливается режим подтверждения удаления, теперь **QDataTable** будет выкидывать перед пользователем окно с запросом на подтверждение выполнения операции удаления. Функция **setSorting(true)** позволит пользователю выполнять сортировку данных в виджете, щелчком мыши по заголовкам колонок. В заключение вызывается функция **refresh()**, которая заполняет **QDataTable** данными.
Затем создаются кнопки **Update** и **Cancel**.

```
  connect(artistTable, SIGNAL(beforeDelete(QSqlRecord *)),
          this, SLOT(beforeDeleteArtist(QSqlRecord *)));
  connect(artistTable, SIGNAL(primeInsert(QSqlRecord *)),
          this, SLOT(primeInsertArtist(QSqlRecord *)));
  connect(artistTable, SIGNAL(beforeInsert(QSqlRecord *)),
          this, SLOT(beforeInsertArtist(QSqlRecord *)));
  connect(updateButton, SIGNAL(clicked()),
          this, SLOT(accept()));
  connect(cancelButton, SIGNAL(clicked()),
          this, SLOT(reject()));
```

Здесь подключаются три сигнала от **QDataTable** к трем приватным слотам. Кнопка **Update** соединяется со слотом **accept()**, кнопка **Cancel** -- со слотом **reject()**.

```
  QHBoxLayout *buttonLayout = new QHBoxLayout;
  buttonLayout->addStretch(1);
  buttonLayout->addWidget(updateButton);
  buttonLayout->addWidget(cancelButton);
  QVBoxLayout *mainLayout = new QVBoxLayout(this);
  mainLayout->setMargin(11);
  mainLayout->setSpacing(6);
  mainLayout->addWidget(artistTable);
  mainLayout->addLayout(buttonLayout);
}
```

В заключение кнопки передаются менеджеру размещения по горизонтали, а **QDataTable** и менеджер размещения по горизонтали -- менеджеру размещения по вертикали.

```
void ArtistForm::accept()
{
  db->commit();
  QDialog::accept();
}
```

Когда пользователь нажимает кнопку **Update**, выполняется подтверждение транзакции и вызывается унаследованный метод **accept()** предка.

```
void ArtistForm::reject()
{
  db->rollback();
  QDialog::reject();
}
```

Когда пользователь нажимает кнопку **Cancel**, выполняется откат транзакции и вызывается унаследованный метод **reject()** предка.

```
void ArtistForm::beforeDeleteArtist(QSqlRecord *buffer)
{
  QSqlQuery query(db);
  query.exec("DELETE FROM track WHERE track.id IN "
```

```
              "(SELECT track.id FROM track, cd "
              "WHERE track.cdid = cd.id AND cd.artistid = "
              + buffer->value("id").toString() + ")");
  query.exec("DELETE FROM cd WHERE artistid = "
              + buffer->value("id").toString());
}
```

Слот **beforeDeleteArtist()** связан с сигналом **beforeDelete()**, виджета **QDataTable**, который выдается непосредственно перед удалением записи. Здесь выполняется каскадное удаление записей, запуском двух запросов: первый удаляет все записи о дорожках на CD по исполнителю, второй -- все CD по исполнителю. Эти операции не нарушают целостность базы данных, потому что выполняются в контексте одной транзакции, которая была запущена в конструкторе формы.

```
void ArtistForm::primeInsertArtist(QSqlRecord *buffer)
{
  buffer->setValue("country", "USA");
}
```

Слот **primeInsertArtist()** связан с сигналом **primeInsert()**, виджета **QDataTable**, который выдается непосредственно перед созданием новой записи. Здесь устанавливается значение по-умолчанию для поля **country**.
Это один из способов установки значений по-умолчанию. Другой способ состоит в создании производного класса от **QSqlCursor** и перекрытии метода **primeInsert()**, но такая метода имеет смысл только в том случае, если один и тот же класс **QSqlCursor** используется в нескольких местах в приложении и обеспечивает непротиворечивость интерфейса. Третий вариант -- сделать это на уровне базы данных, с помощью предложения **DEFAULT** в блоке **CREATE TABLE**.

```
void ArtistForm::beforeInsertArtist(QSqlRecord *buffer)
{
  buffer->setValue("id", generateId("artist", db));
}
```

Слот **beforeInsertArtist()** связан с сигналом **beforeInsert()**, виджета **QDataTable**, который выдается в тот момент, когда пользователь завершает редактирование записи и нажимает клавишу **Enter**, чтобы подтвердить изменения. Здесь устанавливается значение поля **id**. Функция **generateId()** генерирует уникальное значение для первичного ключа.
Так как эта функция будет использоваться в разных местах приложения, она определена как **inline** в заголовочном файле , который будет подключаться к файлам с исходными текстами по мере необходимости. Ниже приводится быстрый (но малоэффективный) вариант функции:

```
inline int generateId(const QString &table, QSqlDatabase *db)
{
  QSqlQuery query(db);
  query.exec("SELECT max(id) FROM " + table);
  query.next();
  return query.value(0).toInt() + 1;
}
```

Функция **generateId()** гарантирует корректную работу только в контексте той же самой транзакции, где исполняется соответствующее выражение **INSERT**.
Некоторые типы баз данных поддерживают автоматическую генерацию значений полей. В этом случае нужно просто настроить базу данных на автоматическую генерацию значений поля **id** и вызвать **setGenerated("id", false)** класса **QSqlCursor**, чтобы сообщить ему, что не нужно генерировать значения для поля **id**.

Теперь рассмотрим другой диалог, который использует **QDataTable**. Этот диалог реализует просмотр таблиц, связанных отношением "мастер-деталь". Мастер-таблица -- это список компакт дисков (CD). Деталь-таблица -- список дорожек на текущем диске. Этот диалог является главным окном приложения **CD Collection**.
На этот раз, вместо контекстного меню, на форму диалога положены кнопки **Add**, **Edit** и **Delete**, которые позволяют пользователю вносить изменения в список компакт дисков. Когда пользователь

нажимает на кнопку **Add** или **Edit**, перед ним появляется диалог **CDForm**. (Описание формы будет приведено в следующем разделе.)

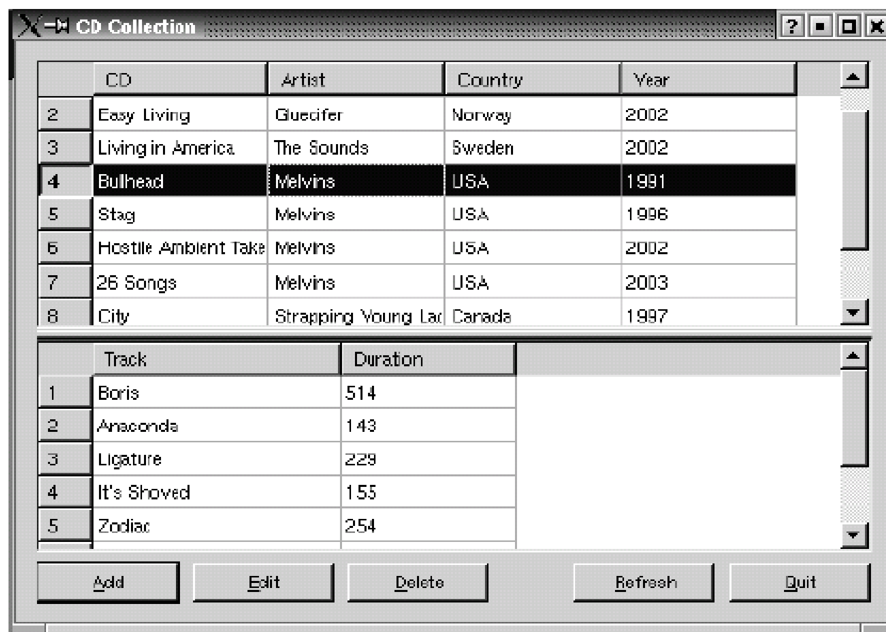


Рисунок 12.3. Диалог MainForm.

Еще одно отличие этого примера от предыдущего заключается в том, что теперь придется работать с внешними ключами, чтобы вместо числового идентификатора исполнителя вывести его имя и название страны. Чтобы добиться этого, необходимо использовать класс **QSqlSelectCursor**, производный от класса **QSqlCursor**.
Определение класса главного окна:

```
class MainForm : public QDialog
{
  Q_OBJECT
public:
  MainForm(QWidget *parent = 0, const char *name = 0);

private slots:
  void addCd();
  void editCd();
  void deleteCd();
  void currentCdChanged(QSqlRecord *record);

private:
  QSplitter *splitter;
  QDataTable *cdTable;
  QDataTable *trackTable;
  QPushButton *addButton;
  ...
  QPushButton *quitButton;
};
```

Класс **MainForm** -- производный от класса **QDialog**.

```
MainForm::MainForm(QWidget *parent, const char *name)
    : QDialog(parent, name)
{
  setCaption(tr("CD Collection"));

  splitter = new QSplitter(Vertical, this);

  QSqlSelectCursor *cdCursor = new QSqlSelectCursor(
                "SELECT cd.id, title, name, country, year "
                "FROM cd, artist WHERE cd.artistid = artist.id");
  if (!cdCursor->isActive()) {
```

```
    QMessageBox::critical(this, tr("CD Collection"),
            tr("The database has not been created.\n"
            "Run the cdtables example to create a sample "
            "database, then copy cdcollection.dat into "
            "this directory and restart this application."));
    qApp->quit();
  }

  cdTable = new QDataTable(cdCursor, false, splitter);
  cdTable->addColumn("title", tr("CD"));
  cdTable->addColumn("name", tr("Artist"));
  cdTable->addColumn("country", tr("Country"));
  cdTable->addColumn("year", tr("Year"));
  cdTable->setAutoDelete(true);
  cdTable->refresh();
```

В конструкторе создается **QDataTable** для таблицы **cd** и связанный с ней курсор. Курсор основан на запросе, который соединяет таблицы **cd** и **artist**. **QDataTable** работает в режиме "только для чтения", потому что взаимодействует с объектом класса **QSqlSelectCursor**. Виджет таблицы, работающий "только на чтение" не имеет контекстного меню.

Если попытка выполнения запроса терпит неудачу, перед пользователем выводится окно, с сообщением об ошибке, и на этом работа приложения завершается.

```
  QSqlCursor *trackCursor = new QSqlCursor("track");
  trackCursor->setMode(QSqlCursor::ReadOnly);
  trackTable = new QDataTable(trackCursor, false, splitter);
  trackTable->setSort(trackCursor->index("number"));
  trackTable->addColumn("title", tr("Track"));
  trackTable->addColumn("duration", tr("Duration"));
```

Здесь создается второй виджет **QDataTable** и его курсор. Вызовом **setMode(QSqlCursor::ReadOnly)** таблица переводится в режим "только для чтения", а вызовом **setSort()** выполняется сортировка по полю с номером дорожки.

```
  addButton = new QPushButton(tr("&Add"), this);
  editButton = new QPushButton(tr("&Edit"), this);
  deleteButton = new QPushButton(tr("&Delete"), this);
  refreshButton = new QPushButton(tr("&Refresh"), this);
  quitButton = new QPushButton(tr("&Quit"), this);

  connect(addButton, SIGNAL(clicked()),
          this, SLOT(addCd()));
  ...
  connect(quitButton, SIGNAL(clicked()),
          this, SLOT(close()));
  connect(cdTable, SIGNAL(currentChanged(QSqlRecord *)),
          this, SLOT(currentCdChanged(QSqlRecord *)));
  connect(cdTable, SIGNAL(doubleClicked(int, int, int, const QPoint &)),
          this, SLOT(editCd()));
  ...
}
```

Здесь настраивается остальная часть пользовательского интерфейса и создаются необходимые соединения "сигнал-слот".

```
void MainForm::addCd()
{
  CdForm form(this);
  if (form.exec()) {
    cdTable->refresh();
    trackTable->refresh();
  }
}
```

Когда пользователь нажимает на кнопку **Add**, вызывается модальный диалог **CdForm** и, если пользователь в этом диалоге нажмет на кнопку **Update**, выполняется обновление таблиц **QDataTable**.

```
void MainForm::editCd()
{
  QSqlRecord *record = cdTable->currentRecord();
  if (record) {
    CdForm form(record->value("id").toInt(), this);
    if (form.exec()) {
      cdTable->refresh();
      trackTable->refresh();
    }
  }
}
```

Когда пользователь нажимает на кнопку **Edit**, вызывается модальный диалог **CdForm**, конструктору которого, передается идентификатор текущего компакт диска. В этом случае диалог запускается с заполненными полями, соответствующими заданному CD.
При таком варианте параметризации формы диалога, возможна ситуация, когда к моменту появления окна диалога, идентификатор диска уже будет отсутствовать в базе данных. Например, пользователь мог нажать кнопку **Edit** за доли секунды до того, как другой пользователь удалил запрашиваемый компакт диск из базы данных. Для решения этой проблемы мы могли бы в **CdForm** выполнить запрос **SELECT** по заданному идентификатору диска и продолжать работу только в том случае, если диск еще присутствует в базе. Однако здесь мы полностью полагаемся на сообщение об ошибке от базы данных.

```
void MainForm::deleteCd()
{
  QSqlRecord *record = cdTable->currentRecord();

  if (record) {
    QSqlQuery query;
    query.exec("DELETE FROM track WHERE cdid = "
               + record->value("id").toString());
    query.exec("DELETE FROM cd WHERE id = "
               + record->value("id").toString());
    cdTable->refresh();
    trackTable->refresh();
  }
}
```

Когда пользователь нажимает на кнопку **Delete**, выполняется удаление всех дорожек диска из таблицы **track**, после чего удаляется запись из таблицы **cd**. В завершение обновляются обе таблицы-виджеты.

```
void MainForm::currentCdChanged(QSqlRecord *record)
{
  trackTable->setFilter("cdid = "
                        + record->value("id").toString());
  trackTable->refresh();
}
```

Слот **currentCdChanged()** связан с сигналом **currentChanged()** объекта **cdTable**, который выдается, когда пользователь вносит изменения в текущую запись о CD или перемещается к другой записи. Всякий раз, когда это происходит, вызывается функция **setFilter()** и обновляется таблица **trackTable**. Таким образом она всегда будет отображать только те дорожки, которые относятся к текущему CD.
По сути -- это весь код, реализующий функциональность **MainForm**. Однако тут следует упомянуть об одном небольшом улучшении, которое можно добавить. Суть его заключается в том, чтобы показывать длительность звучания дорожки не в секундах (например, "155"), как это делается сейчас, а в минутах и секундах (например, "02:35"). С этой целью необходимо создать производный класс от **QSqlCursor** и перекрыть в нем метод **calculateField()**, для преобразования значения поля **duration** в **QString** с заданным форматом представления:

```
QVariant TrackSqlCursor::calculateField(const QString &name)
{
  if (name == "duration") {
    int duration = value("duration").toInt();
```

```
    return QString("%1:%2").arg(duration / 60, 2)
                           .arg(duration % 60, 2);
  }
  return QVariant();
}
```

Кроме того, в этом случае необходимо вызвать метод курсора **setCalculated("duration", true)**, чтобы **QDataTable** использовала значение поля duration, возвращаемое функцией **calculateField()**.

12.3 Разработка форм, ориентированных на работу с базами данных

Qt предлагает инновационный подход к организации взаимодействий форм с базами данных. Вместо того, чтобы создавать отдельные версии встроенных виджетов, для работы с данными, Qt делает любой виджет ориентированным на работу с базами данных, с помощью класов **QSqlForm** и **QSqlPropertyMap**, которые выступают посредниками между базой данных и виджетами. Любой виджет, не зависимо от того -- стандартный он или нет, сможет работать с базами данных, при использовании этих классов.

Класс **QSqlForm**, производный от **QObject**, облегчает создание форм для просмотра и изменения отдельных записей. В общем случае, порядок создания формы диалога выглядит следующим образом:
- Создаются виджеты-редакторы (**QLineEdit**, **QComboBox**, **QSpinBox** и т.п.) для каждого из полей записи.
- Создается экземпляр **QSqlCursor**.
- Создается экземпляр **QSqlForm**.
- Выполняется настройка **QSqlForm**, которая заключается в связывании полей записи с виджетами.
- Вызывается метод **QSqlForm::readFields()**, который заполняет виджеты данными.
- Диалог выводится перед пользователем.
- По завершении работы диалога, Вызывается метод **QSqlForm::writeFields()**, чтобы скопировать измененные значения обратно в базу данных.

Все эти действия мы продемонстрируем на примере диалога **CdForm**. Он предназначен для создания и изменения записи о компакт диске. Пользователь может задать название диска, исполнителя и год выпуска, а так же название и продолжительность каждого произведения на диске.

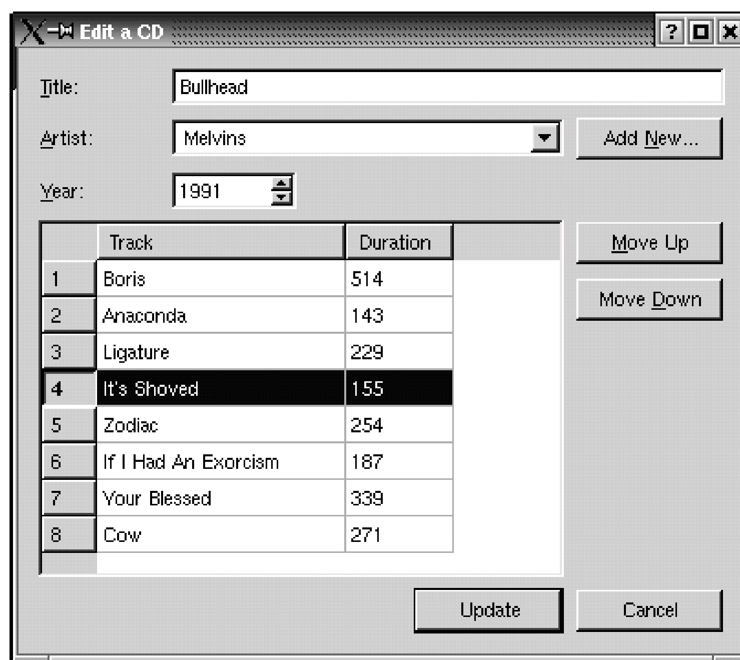


Рисунок 12.4. Диалог CdForm.

Начнем с определения класса:

```
class CdForm : public QDialog
{
  Q_OBJECT
public:
  CdForm(QWidget *parent = 0, const char *name = 0);
  CdForm(int id, QWidget *parent = 0, const char *name = 0);
  ~CdForm();

protected slots:
  void accept();
  void reject();

private slots:
  void addNewArtist();
  void moveTrackUp();
  void moveTrackDown();
  void beforeInsertTrack(QSqlRecord *buffer);
  void beforeDeleteTrack(QSqlRecord *buffer);

private:
  void init();
  void createNewRecord();
  void swapTracks(int trackA, int trackB);

  QLabel *titleLabel;
  QLabel *artistLabel;
  ...
  QDataTable *trackTable;
  QSqlForm *sqlForm;
  QSqlCursor *cdCursor;
  QSqlCursor *trackCursor;
  int cdId;
  bool newCd;
};
```

В классе объявлены два конструктора: один вставляет новую запись в базу данных, другой -- обновляет существующую запись. Слоты **accept()** и **reject()** унаследованы от **QDialog**.

```
CdForm::CdForm(QWidget *parent, const char *name)
    : QDialog(parent, name)
{
  setCaption(tr("Add a CD"));
  cdId = -1;
  init();
}
```

Первый конструктор записывает в заголовок диалога строку "Add a CD" ("добавить диск") и вызывает приватную функцию **init()**, которая выполняет остальную часть работы.

```
CdForm::CdForm(int id, QWidget *parent, const char *name)
    : QDialog(parent, name)
{
  setCaption(tr("Edit a CD"));
  cdId = id;
  init();
}
```

Второй конструктор записывает в заголовок диалога строку "Edit a CD" ("изменить сведения о диске") и так же вызывает функцию **init()**.

```
void CdForm::init()
{
  db = QSqlDatabase::database("CD");
  db->transaction();
  if (cdId == -1)
    createNewRecord();
```

В функции **init()** запускается транзакция для соединения под именем "CD". Для диалогов **CdForm** и **ArtistForm** используются различные соединения, поскольку оба они могут отображаться одновременно и при этом нельзя допустить, чтобы операция **Cancel** в одной форме выполняла откат транзакции, запущенной в другой форме.

Если идентификатор диска не задан, вызывается функция **createNewRecord()**, которая вставляет пустую запись в таблицу. Это позволит использовать **cdId** как внешний ключ для **QDataTable** с дорожками. Если пользователь нажмет на кнопку **Cancel**, все изменения, произведенные в контексте транзакции, будут отменены, в том числе и операция вставки новой записи.

В этом диалоге используется еще одно соединение с базой данных, отличное от того, что используется в **ArtistForm**. Сделано это так потому, что в соединении, активной может быть только одна транзакция, а в данном приложении может сложиться ситуация, когда потребуется иметь две активных транзакции одновременно, например, в случае, когда пользователь нажимает кнопку **Add New** и открывает диалог **ArtistForm**.

```
titleLabel = new QLabel(tr("&Title:"), this);
artistLabel = new QLabel(tr("&Artist:"), this);
yearLabel = new QLabel(tr("&Year:"), this);
titleLineEdit = new QLineEdit(this);
yearSpinBox = new QSpinBox(this);
yearSpinBox->setRange(1900, 2100);
yearSpinBox->setValue(QDate::currentDate().year());
artistComboBox = new ArtistComboBox(db, this);
artistButton = new QPushButton(tr("Add &New..."), this);
...
cancelButton = new QPushButton(tr("Cancel"), this);
```

На форме диалога размещаются текстовые метки, поле ввода, счетчик, выпадающий список и кнопки. Выпадающий список принадлежит к классу **ArtistComboBox**, о котором мы поговорим немного ниже.

```
trackCursor = new QSqlCursor("track", true, db);
trackTable = new QDataTable(trackCursor, false, this);
trackTable->setFilter("cdid = " + QString::number(cdId));
trackTable->setSort(trackCursor->index("number"));
trackTable->addColumn("title", tr("Track"));
trackTable->addColumn("duration", tr("Duration"));
trackTable->refresh();
```

Далее создается и настраивается **QDataTable**, которая позволит пользователю просматривать и изменять сведения о дорожках на текущем компакт диске. Очень напоминает то, что мы делали с классом **ArtistForm**, в предыдущем разделе.

```
cdCursor = new QSqlCursor("cd", true, db);
cdCursor->select("id = " + QString::number(cdId));
cdCursor->next();
```

Создается **QSqlCursor** и текущей, для него, делается запись, содержащая идентификатор требуемого диска.

```
QSqlPropertyMap *propertyMap = new QSqlPropertyMap;
propertyMap->insert("ArtistComboBox", "artistId");
sqlForm = new QSqlForm(this);
sqlForm->installPropertyMap(propertyMap);
sqlForm->setRecord(cdCursor->primeUpdate());
sqlForm->insert(titleLineEdit, "title");
sqlForm->insert(artistComboBox, "artistid");
sqlForm->insert(yearSpinBox, "year");
sqlForm->readFields();
```

Класс **QSqlPropertyMap** хранит сведения, благодаря которым **QSqlForm** "знает" -- значения какого типа, в каком свойстве, может хранить тот или иной виджет-редактор. Класс **QSqlForm** уже "знает", что **QLineEdit** запоминает свое значение в свойстве **text**, а **QSpinBox** -- в свойстве **value**. Но он ничего не знает о нестандартных виджетах, коим является **ArtistComboBox**. Поэтому мы должны вставить

название класса и имя свойства класса ("ArtistComboBox", "artistId") в карту свойств и вызвать **installPropertyMap()**, чтобы указать **QSqlForm**, что при работе с виджетом класса **ArtistComboBox** следует использовать свойство **artistId**.
Кроме того экземпляру класса **QSqlForm** нужно передать буфер, с которым он будет работать, а так же сообщить о том, какой виджет, какому полю в таблице соответствует. В заключение, вызовом **readFields()**, данные считываются из базы и переносятся в виджеты.

```
  connect(artistButton, SIGNAL(clicked()),
          this, SLOT(addNewArtist()));
  connect(moveUpButton, SIGNAL(clicked()),
          this, SLOT(moveTrackUp()));
  connect(moveDownButton, SIGNAL(clicked()),
          this, SLOT(moveTrackDown()));
  connect(updateButton, SIGNAL(clicked()),
          this, SLOT(accept()));
  connect(cancelButton, SIGNAL(clicked()),
          this, SLOT(reject()));
  connect(trackTable, SIGNAL(beforeInsert(QSqlRecord *)),
          this, SLOT(beforeInsertTrack(QSqlRecord *)));
  ...
}
```

На последней стадии выполнения функции производится соединение сигналов и слотов, которые будут описаны несколько ниже.

```
void CdForm::accept()
{
  sqlForm->writeFields();
  cdCursor->update();
  db->commit();
  QDialog::accept();
}
```

Когда пользователь нажимает на кнопку **Update**, производится запись новых значений полей в буфер редактирования объекта **QSqlCursor**. Затем выполняется SQL-предложение **UPDATE**, вызовом функции **update()**, вызовом **commit()** подтверждается транзакция и в заключение вызывается метод **accept()**, унаследованный от **QDialog**.

```
void CdForm::reject()
{
  db->rollback();
  QDialog::reject();
}
```

Когда пользователь нажимает на кнопку **Cancel**, производится откат произведенных изменений и форма диалога закрывается.

```
void CdForm::addNewArtist()
{
  ArtistForm form(this);
  if (form.exec()) {
    artistComboBox->refresh();
    updateButton->setEnabled(artistComboBox->count() > 0);
  }
}
```

Когда пользователь нажимает на кнопку **Add New**, запускается модальный диалог **ArtistForm**. Этот диалог позволяет добавлять нового исполнителя в базу данных, удалять или изменять сведения об исполнителях. Если пользователь нажмет кнопку **Update**, будет вызвана функция **ArtistComboBox::refresh()**, которая обновит список исполнителей в виджете.
Если в списке нет ни одного исполнителя, кнопка **Update** будет запрещена, поскольку необходимо избежать создания записи о CD, без указания имени исполнителя.

```
void CdForm::beforeInsertTrack(QSqlRecord *buffer)
{
```

```
  buffer->setValue("id", generateId("track", db));
  buffer->setValue("number", trackCursor->size() + 1);
  buffer->setValue("cdid", cdId);
}
```

Слот **beforeInsertTrack()** связан с сигналом **beforeInsert()**. Он заполняет поля **id**, **number** и **cdid**.

```
void CdForm::beforeDeleteTrack(QSqlRecord *buffer)
{
  QSqlQuery query(db);
  query.exec("UPDATE track SET number = number - 1 "
             "WHERE track.number > "
             + buffer->value("number").toString());
}
```

Слот **beforeDeleteTrack()** связан с сигналом **beforeDelete()**. Он выполняет перенумерацию дорожек на диске, чьи номера больше номера удаляемой дорожки, чтобы сохранить неразрывность последовательности номеров дорожек. Например, допустим, что диск содержит 6 дорожек и пользователь удаляет 4-ю, тогда 5-я дорожка получит номер 4, а 6-я -- 5.

Имеется еще 4 функции, описание которых мы не привели: **moveTrackUp()**, **moveTrackDown()**, **swapTracks()** и **createNewRecord()**. Они совершенно необходимы, чтобы сделать приложение более-менее удобным, но их реализация не содержит ничего нового для вас, поэтому мы не будем их рассматривать. Исходные тексты функций вы найдете на CD, сопровождающем книгу.
Теперь, после того как мы рассмотрели все классы диалогов в приложении, можно перейти к описанию нестандартного класса **ArtistComboBox**. Как обычно, начнем с определения класса:

```
class ArtistComboBox : public QComboBox
{
  Q_OBJECT
  Q_PROPERTY(int artistId READ artistId WRITE setArtistId)
public:
  ArtistComboBox(QSqlDatabase *database, QWidget *parent = 0,
                 const char *name = 0);
  void refresh();
  int artistId() const;
  void setArtistId(int id);

private:
  void populate();

  QSqlDatabase *db;
  QMap<int, int> idFromIndex;
  QMap<int, int> indexFromId;
};
```

Класс **ArtistComboBox** порожден от класса **QComboBox**. В него добавлено свойство **artistId** и несколько функций.
В приватной секции объявлены две переменные-члены типа **QMap<int, int>**. Первая отвечает за соответствие идентификатора исполнителя индексу в списке виджета. Вторая отвечает за соответствие индекса в списке -- идентификатору исполнителя.

```
ArtistComboBox::ArtistComboBox(QSqlDatabase *database,
                               QWidget *parent, const char *name)
    : QComboBox(parent, name)
{
  db = database;
  populate();
}
```

В конструкторе вызывается функция **populate()**, которая заполняет список виджета именами и идентификаторами из таблицы **artist**.

```
void ArtistComboBox::refresh()
{
  int oldArtistId = artistId();
```

```
  clear();
  idFromIndex.clear();
  indexFromId.clear();
  populate();
  setArtistId(oldArtistId);
}
```

Функция **refresh()** очищает и повторно заполняет список виджета самыми свежими данными из базы. При этом, после обновления списка выбранным остается тот же исполнитель, который был выбран до обновления.

```
void ArtistComboBox::populate()
{
  QSqlCursor cursor("artist", true, db);
  cursor.select(cursor.index("name"));

  int index = 0;
  while (cursor.next()) {
    int id = cursor.value("id").toInt();
    insertItem(cursor.value("name").toString(), index);
    idFromIndex[index] = id;
    indexFromId[id] = index;
    ++index;
  }
}
```

Функция **populate()** переносит список исполнителей из базы данных в список виджета, попутно обновляя словари **idFromIndex** и **indexFromId**.

```
int ArtistComboBox::artistId() const
{
  return idFromIndex[currentItem()];
}
```

Функция **artistId()** возвращает идентификатор текущего исполнителя.

```
void ArtistComboBox::setArtistId(int id)
{
  if (indexFromId.contains(id))
    setCurrentItem(indexFromId[id]);
}
```

Функция **setArtistId()** делает текущим исполнителя с заданным идентификатором.
В приложении, часто использующем виджет выпадающего списка, который предназначен для отображения внешнего ключа, вероятно было бы более удобным создать универсальный **DatabaseComboBox**, в конструктор которого можно было бы передать имя таблицы, имя поля, которое должно отображаться в списке и имя поля, которое должно использоваться в качестве идентификатора.
Закончим обзор приложения "CD Collection" рассмотрением реализации функций **createConnections()** и **main()**.

```
inline bool createOneConnection(const QString &name)
{
  QSqlDatabase *db;
  if (name.isEmpty())
    db = QSqlDatabase::addDatabase("QSQLITEX");
  else
    db = QSqlDatabase::addDatabase("QSQLITEX", name);
  db->setDatabaseName("cdcollection.dat");
  if (!db->open()) {
    db->lastError().showMessage();
    return false;
  }
  return true;
}

inline bool createConnections()
```

```
{
  return createOneConnection("")
          && createOneConnection("ARTIST")
          && createOneConnection("CD");
}
```

Функция **createConnections()** создает три идентичных соединения с базой данных. Первое соединение создается безымянным, оно будет использоваться по-умолчанию, когда имя соединения не задано явно. Два других соединения создаются с именами "**ARTIST**" и "**CD**" -- они используются диалогами **ArtistForm** и **CdForm**, соответственно.

```
int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  if (!createConnections())
    return 1;

  MainForm mainForm;
  app.setMainWidget(&mainForm);
  mainForm.resize(480, 320);
  mainForm.show();
  return app.exec();
}
```

Функция **main()** практически ничем не отличается от аналогичных функций, которые мы до сих пор видели, за одним маленьким исключением -- она вызывает **createConnections()**.

Как мы уже говорили в конце предыдущего раздела, приложение можно было бы несколько улучшить, если показывать длительность звучания дорожки не в секундах, а в минутах и секундах. Но, кроме перекрытия метода **QSqlCursor::calculateField()**, это повлекло бы за собой еще и создание класса -- редактора времени звучания дорожки, производного от **QSqlEditorFactory**. А затем необходимо было бы с помощью **QSqlPropertyMap** сообщить **QDataTable** о том, как получить измененное значение от класса-редактора. За дополнительной информацией обращайтесь к сопроводительной документации по функциям **installEditorFactory()** и **installPropertyMap()**, класса **QDataTable**.

Еще одно из возможных улучшений приложения -- добавить возможность хранения в базе данных изображения обложки диска и отображения его в **CdForm**. Реализовать это можно за счет добавления в базу данных поля типа **BLOB**, в котором можно хранить изображения. Получать изображения из базы данных можно в виде **QByteArray** и затем передавать их в конструктор **QImage**.

13 РАБОТА С СЕТЬЮ

Для работы с протоколами FTP и HTTP в библиотеке Qt имеются классы **QFtp** и **QHttp**. Они достаточно удобны для организации обмена файлами по сети.

Классы **QFtp** и **QHttp** основаны на низкоуровневом классе **QSocket**, который реализует представление сокетов TCP. Протокол TCP работает в терминах потоков данных, передаваемых между узлами сети. Класс **QSocket**, в свою очередь, реализован поверх **QSocketDevice** -- тонкой "обертки" вокруг платформо-зависимого сетевого API операционной системы. Класс **QSocketDevice** поддерживает протоколы TCP и UDP.

В этой главе мы будем говорить об этих 4-х, и некоторых других классах, и покажем -- как с ними работать. Расскажем, как организовать обмен файлами по сети. Протокол TCP будет использоваться нами при написании приложений-серверов и соответствующих им приложений-клиентов. Аналогично, протокол UDP будет использоваться для написания передающей и принимающей части приложений. Понимание принципов работы классов **QFtp** и **QHttp** обычно ни у кого не вызывает затруднений, даже у новичков, однако, для понимания принципов работы с классами **QSocket** и **QSocketDevice**, желательно иметь некоторый опыт работы с сетями.

13.1 Класс QFtp

Класс **QFtp** предназначен для создания клиентских приложений, работающих с протоколом FTP. Он реализует набор функций, для выполнения наиболее распространенных операций этого протокола, включая **get()**, **put()**, **remove()** и **mkdir()**.

Все операции выполняются асинхронно. Когда вызывается функция, такая как **get()** или **put()**, управление сразу же возвращается программе, а собственно передача данных начинает производиться, когда управление опять переходит в цикл обработки событий Qt. Благодаря этому, во время исполнения FTP-команд, не возникает эффекта "замораживания" интерфейса с пользователем. Демонстрацию возсможностей **QFtp** начнем с показа того, как скачать файл с сервера, используя функцию **get()**. Предположим, что основной класс приложения **MainWindow** должен скачать прейскурант с FTP-сайта.

```
class MainWindow : public QMainWindow
{
  Q_OBJECT
public:
  MainWindow(QWidget *parent = 0, const char *name = 0);

  void getPriceList();
  ...

private slots:
  void ftpDone(bool error);

private:
  QFtp ftp;
  QFile file;
  ...
};
```

Класс определяет публичную функцию **getPriceList()**, которая отвечает за получение файла с прейскурантом, и приватный слот **ftpDone(bool)**, который вызывается по окончании приема файла. Так же в классе определены две переменные: **ftp**, ответственную за взаимодействие с FTP-сервером, и **file**, используемую для записи файла на диск.

```
MainWindow::MainWindow(QWidget *parent, const char *name)
    : QMainWindow(parent, name)
{
  ...
  connect(&ftp, SIGNAL(done(bool)), this, SLOT(ftpDone(bool)));
}
```

В конструкторе выполняется соединение между сигналом **done(bool)**, экземпляра класса **QFtp**, и слотом **ftpDone(bool)**. Объекты класса **QFtp** выдают этот сигнал по завершении обработки всех запросов. Параметр **bool** указывает на наличие возможных ошибок.

```
void MainWindow::getPriceList()
{
  file.setName("price-list.csv");
  if (!file.open(IO_WriteOnly)) {
    QMessageBox::warning(this, tr("Sales Pro"),
                         tr("Cannot write file %1\n%2.")
                         .arg(file.name())
                         .arg(file.errorString()));
    return;
  }

  ftp.connectToHost("ftp.trolltech.com");
  ftp.login();
  ftp.cd("/topsecret/csv");
  ftp.get("price-list.csv", &file);
  ftp.close();
}
```

Функция **getPriceList()** загружает файл **ftp://ftp.trolltech.com/topsecret/csv/price-list.csv** и сохраняет его под именем **price-list.csv** в текущем каталоге.

Начинается функция с попытки открыть на запись файл в текущем каталоге. Затем выполняется последовательность из пяти FTP-команд. Второй аргумент функции **get()** задает устройство, в которое будет осуществляться запись принимаемых данных.
Команды FTP ставятся в очередь и исполняются в цикле обработки событий Qt. По завершении обработки всех команд, объект **QFtp** выдает сигнал **done(bool)**, который подключен к слоту **ftpDone(bool)**.

```
void MainWindow::ftpDone(bool error)
{
  if (error)
    QMessageBox::warning(this, tr("Sales Pro"),
                         tr("Error while retrieving file with " "FTP: %1.")
                         .arg(ftp.errorString()));
  file.close();
}
```

После выполнения FTP-команд, файл закрывается. Если возникла какая либо ошибка, перед пользователем выводится соответствующее сообщение.

Класс **QFtp** предоставляет следующие операции: **connectToHost()**, **login()**, **close()**, **list()**, **cd()**, **get()**, **put()**, **remove()**, **mkdir()**, **rmdir()** и **rename()**. Все эти функции ставят соответствующие команды в очередь и возвращают идентификационный номер команды. Любые команды FTP могут быть испольнены с помощью функции **rawCommand()**. Например, так выглядит исполнение команды **SITE CHMOD**:

```
    ftp.rawCommand("SITE CHMOD 755 fortune");
```

Объекты **QFtp**, перед исполнением команды, выдают сигнал **commandStarted(int)**, а по завершении -- **commandFinished(int, bool)**. Аргумент **int** -- это идентификационный номер команды. Если вас интересует ход выполнения отдельных команд, то вам придется сохранять их идентификационные номера, при вызове соответствующей функции. Благодаря этому появится возможность предоставить пользователю более детальную информацию о ходе процесса. Например:

```
void MainWindow::getPriceList()
{
  ...
  connectId = ftp.connectToHost("ftp.trolltech.com");
  loginId = ftp.login();
  cdId = ftp.cd("/topsecret/csv");
  getId = ftp.get("price-list.csv", &file);
```

```
  closeId = ftp.close();
}

void MainWindow::commandStarted(int id)
{
  if (id == connectId) {
    statusBar()->message(tr("Connecting..."));
  } else if (id == loginId) {
    statusBar()->message(tr("Logging in..."));
  ...
}
```

Другой способ обеспечения пользователя обратной связью с процессом -- использовать сигнал **stateChanged()**.
Однако, в большинстве приложений нас интересует только результат выполнения всей последовательности команд. В этом случае мы просто соединяемся с сигналом **done(bool)**, который выдается послк выполнения последней команды в последовательности.
При возникновении ошибки, **QFtp** автоматически очищает очередь команд. Это означает, что если ошибка произошла во время установления соединения или во время авторизации на сервере, следующие за ними команды никогда не будут выполнены. Но если после возникновения ошибки в очередь будут поставлены другие команды, то они будут исполнены как ни в чем не бывало.
Теперь рассмотрим более сложный пример:

```
class Downloader : public QObject
{
  Q_OBJECT
public:
  Downloader(const QUrl &url);

signals:
  void finished();

private slots:
  void ftpDone(bool error);
  void listInfo(const QUrlInfo &urlInfo);

private:
  QFtp ftp;
  std::vector<QFile *> openedFiles;
};
```

Экземпляр класса **Downloader** попытается скачать все файлы из каталога FTP. Имя каталога задается как **QUrl**, при вызове конструктора. Класс **QUrl** -- это стандартный класс из библиотеки Qt, который реализует интерфейс для работы со строками URL и выделения их них отдельных частей, таких как имя файла, путь к файлу, протокол и порт.

```
Downloader::Downloader(const QUrl &url)
{
  if (url.protocol() != "ftp") {
    QMessageBox::warning(0, tr("Downloader"),
                         tr("Protocol must be  ftp ."));
    emit finished();
    return;
  }

  int port = 21;
  if (url.hasPort())
    port = url.port();

  connect(&ftp, SIGNAL(done(bool)),
          this, SLOT(ftpDone(bool)));
  connect(&ftp, SIGNAL(listInfo(const QUrlInfo &)),
          this, SLOT(listInfo(const QUrlInfo &)));

  ftp.connectToHost(url.host(), port);
  ftp.login(url.user(), url.password());
  ftp.cd(url.path());
  ftp.list(); }
```

В конструкторе прежде всего выполняется проверка строки URL -- она должна начинаться с комбинации символов: "**ftp:**". Затем из URL извлекается номер порта, если порт не указан, то предполагается использование стандартного FTP-порта -- 21.
Затем выполняются соединения сигнал-слот и в очередь помещаются 4 FTP-команды. Последняя из них запрашивает у сервера список файлов и выдает сигнал **listInfo(const QUrlInfo &)**, когда от сервера приходит очередное имя файла. Этот сигнал связан со слотом **listInfo()**, отвечающим за скачивание файла.

```
void Downloader::listInfo(const QUrlInfo &urlInfo)
{
  if (urlInfo.isFile() && urlInfo.isReadable()) {
    QFile *file = new QFile(urlInfo.name());
    if (!file->open(IO_WriteOnly)) {
      QMessageBox::warning(0, tr("Downloader"),
                           tr("Error: Cannot open file " "%1:\n%2.")
                           .arg(file->name())
                           .arg(file->errorString()));
      emit finished();
      return;
    }
    ftp.get(urlInfo.name(), file);
    openedFiles.push_back(file);
  }
}
```

Аргумент типа **QUrlInfo** предоставляет подробную информацию о файле. Если это обычный файл (не каталог) и доступен на чтение, то производится попытка скачать его, вызовом **get()**. Объект **QFile** используется для сохранения локальной копии файла, он создается оператором new, а указатель на него сохраняется в динамическом массиве (векторе) **openedFiles**.

```
void Downloader::ftpDone(bool error)
{
  if (error)
    QMessageBox::warning(0, tr("Downloader"),
                         tr("Error: %1.")
                         .arg(ftp.errorString()));

  for (int i = 0; i < (int)openedFiles.size(); ++i)
    delete openedFiles[i];
  emit finished();
}
```

Слот **ftpDone()** вызывается по завершении выполнения последовательности команд или в случае возникновения ошибки. Функция удаляет все объекты **QFile**, попутно закрывая все файлы. (Файлы закрываются автоматически деструктором класса **QFile**.)
Если ошибок не возникло, то порядок выполнения команд и выдачи сигналов будет следующим:

```
  connectToHost(host)
  login()
  cd(path)
  list()
    emit listInfo(file_1)
      get(file_1)
    emit listInfo(file_2)
      get(file_2)
    ...
    emit listInfo(file_N)
      get(file_N)
  emit done()
```

Если ошибка возникла, например, во время скачивания пятого файла из двадцати имевшихся, то оставшиеся пятнадуать файлов не будут скачиваться. Если вас это не устраивает, то можно попробовать выполнять скачивание файлов по одному -- запускать команду **GET**, ждать появления сигнала **done(bool)** и только после этого запускать **GET** для очередного файла. А в функции **listInfo()** --

просто создавать список имен файлов в каталоге сервера. В этом случае порядок выполнения команд и выдачи сигналов будет следующим:

```
connectToHost(host)
login()
cd(path)
list()
  emit listInfo(file_1)
  emit listInfo(file_2)
  ...
  emit listInfo(file_N)
emit done()
get(file_1)
  emit done()
get(file_2)
  emit done()
...
get(file_N)
  emit done()
```

Другой вариант решения проблемы состоит в использовании отдельного объекта **QFtp** для каждого из файлов. Это позволит выполнять параллельную загрузку нескольких файлов через различные FTP-соединения.

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

  QUrl url("ftp://ftp.example.com/");
  if (argc >= 2)
    url = argv[1];
  Downloader downloader(url);
  QObject::connect(&downloader, SIGNAL(finished()),
                   &app, SLOT(quit()));
  return app.exec();
}
```

Реализацией функции **main()** мы завершаем рассмотрение программы. Если пользователь указывает URL в командной строке, то файлы скачиваются из указанного каталога, в противном случае -- из каталога **ftp://ftp.example.com/**.
В обоих вышеприведенных примерах, файл скачивается с помощью функции **get()** и записывается на диск посредством объекта **QFile**. В случае же, если принятый файл нужно сохранить в памяти, то для этого прекрасно подойдет класс **QBuffer**, производный от класса **QIODevice** -- обертки вокруг класса **QByteArray**. Например:

```
QBuffer *buffer = new QBuffer(byteArray);
buffer->open(IO_WriteOnly);
ftp.get(urlInfo.name(), buffer);
```

В функции **get()** мы могли бы опустить второй аргумент или передать в место него "пустой" (NULL) указатель. В этом случае **QFtp** будет выдавать сигнал **readyRead()** всякий раз, при поступлении очередной порции данных, которые могут быть прочитаны вызовом **readBlock()** или **readAll()**.
Если необходимо отображать ход выполнения скачивания файла, то можно связать сигнал **dataTransferProgress(int, int)**, класса **QFtp**, со слотом **setProgress(int, int)** класса **QProgressBar** или **QProgressDialog**. Кроме того, можно привязать сигнал **canceled()**, класса **QProgressBar** или **QProgressDialog** со слотом **abort()**, класса **QFtp**.

13.2 Класс QHttp

Класс **QHttp** предназначен для создания клиентских приложений, работающих с протоколом HTTP. Он реализует набор функций, для выполнения наиболее распространенных операций этого протокола, включая **get()** и **post()**. Если вы прочитали предыдущий раздел, то обнаружите, что класс **QHttp** очень похож на **QFtp**.

Как и **QFtp**, объекты класса **QHttp** работают асинхронно. Функции **get()** и **post()** сразу же возвращают управление вызывающей программе, а собственно передача данных осуществляется в цикле обработки событий.
Рассмотрим принцип действия **QHttp** на примере приложения, которое пытается получить HTML-файл с сайта **Trolltech**. Мы не будем здесь приводить содержимое заголовочного файла, т.к. он очень похож на тот, который использовался в предыдущем разделе. Различие состоит лишь в том, что теперь приватный слот объявлен под именем **httpDone(bool)** и изменено объявление одной приватной переменной (**http** типа **QHttp**).

```
MainWindow::MainWindow(QWidget *parent, const char *name)
    : QMainWindow(parent, name)
{
  ...
  connect(&http, SIGNAL(done(bool)), this, SLOT(httpDone(bool)));
}
```

В конструкторе выполняется соединение сигнала **done(bool)**, объекта **QHttp**, со слотом главного окна -- **httpDone(bool)**.

```
void MainWindow::getFile()
{
  file.setName("aboutqt.html");
  if (!file.open(IO_WriteOnly)) {
    QMessageBox::warning(this, tr("HTTP Get"),
                         tr("Cannot write file %1\n%2.")
                         .arg(file.name())
                         .arg(file.errorString()));
    return;
  }

  http.setHost("doc.trolltech.com");
  http.get("/3.2/aboutqt.html", &file);
  http.closeConnection();
}
```

Функция **getFile()** загружает файл **http://doc.trolltech.com/3.2/aboutqt.html** и сохраняет его в текущем каталоге, под именем **aboutqt.html**.
Объект **QFile** пытается открыть файл на запись, после этого в очередь помещается последовательность из трех HTTP-команд. Второй аргумент функции **get()** определяет устройство, куда будут записаны полученные данные.

HTTP-запросы исполняются в цикле обработки событий. По завершении заданной последовательности команд, объект **Qhttp** выдает сигнал **done(bool)**, который поступает в слот **httpDone(bool)**.

```
void MainWindow::httpDone(bool error)
{
  if (error)
    QMessageBox::warning(this, tr("HTTP Get"),
                         tr("Error while fetching file with "
                           "HTTP: %1.")
                         .arg(http.errorString()));
  file.close();
}
```

После того, как запрос будет выполнен, файл закрывается. При возникновении ошибки перед пользователем выводится соответствующее сообщение.
Среди всего прочего, **QHttp** предоставляет в распоряжение программиста следующие функции: **setHost()**, **get()**, **post()** и **head()**. Ниже приводится пример передачи списка пар "имя = значение" в CGI-скрипт:

```
    http.setHost("www.example.com");
    http.post("/cgi/somescript.py", QCString("x=200&y=320"), &file);
```

Для выполнения произвольных HTTP-запросов можно использовать более универсальную функцию **request()**, 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, QCString("qt-interest=on&search=opengl"));
```

Перед началом выполнения очередной операции, **QHttp** выдает сигнал **requestStarted(int)**, а после окончания -- **requestFinished(int, bool)**. Аргумент **int** определяет идентификационный номер запроса. Если вас интересует ход выполнения отдельных команд, то вам придется сохранять их идентификационные номера, при вызове соответствующей функции. Благодаря этому появится возможность предоставить пользователю более детальную информацию о ходе процесса. Однако, в большинстве приложений нас интересует только результат выполнения всей последовательности команд. В этом случае мы просто соединяемся с сигналом **done(bool)**, который выдается после выполнения последней команды в последовательности.

При возникновении ошибки, очередь команд автоматически очищается. Это означает, что все последующие команды никогда не будут выполнены. Но если в очередь будут поставлены другие команды уже после возникновения ошибки, то они будут исполнены как ни в чем не бывало. Подобно **QFtp**, объекты класса **QHttp** имеют в своем распоряжении сигнал **readyRead()**, и функции **readBlock()** и **readAll()**, которые могут использоваться в том случае, когда функции **get()** не передается устройство для записи. Кроме того, этот класс так же имет сигнал **dataTransferProgress(int, int)**, который может быть напрвлен в слот **setProgress(int, int)** класса **QProgressBar** или **QProgressDialog**.

13.3 Класс QSocket

Класс **QSocket** может использоваться при разработке приложений серверов и клиентов, работающих по протоколу TCP. TCP -- это протокол транспортного уровня, который является базой для множества других протоколов Интернет, включая FTP и HTTP, а так же может служить основой для разработки нестандартных протоколов обмена данными.

Протокол TCP ориентирован на потоки. Протоколы более высокого уровня, работающие поверх TCP, обычно подразделяются на строко-ориентированные и блочно-ориентированные:

- Строко-ориентированные протоколы передают данные в виде текстовых строк, каждая из которых завершается символом перевода строки.
- Блочно-ориентированные протоколы передают данные в виде блоков. Размер каждого блока содержится в отдельном поле, внутри блока.

Класс **QSocket** порожден от класса **QIODevice**, поэтому он в состоянии читать и писать данные из/в экземпляры классов **QDataStream** или **QTextStream**. Одно важное отличие чтения данных из сети от чтения данных из файла состои в том, что перед вызовом оператора "**>>**" необходимо убедиться в том, что от удаленного хоста получены все данные. В случае ошибки мы можем получить непредсказуемый результат.

В этом разделе мы рассмотрим исходный код клиентского и серверного приложений, которые используют собственный, блочно-ориентированный протокол обмена. Приложение-клиент называется **Trip Planner**. Оно позволяет пользователю планировать поездку по железной дороге. Приложение-сервер называется **Trip Server**. Оно предоставляет клиенту информацию о расписании движения поездов. Начнем с приложения **Trip Planner**.

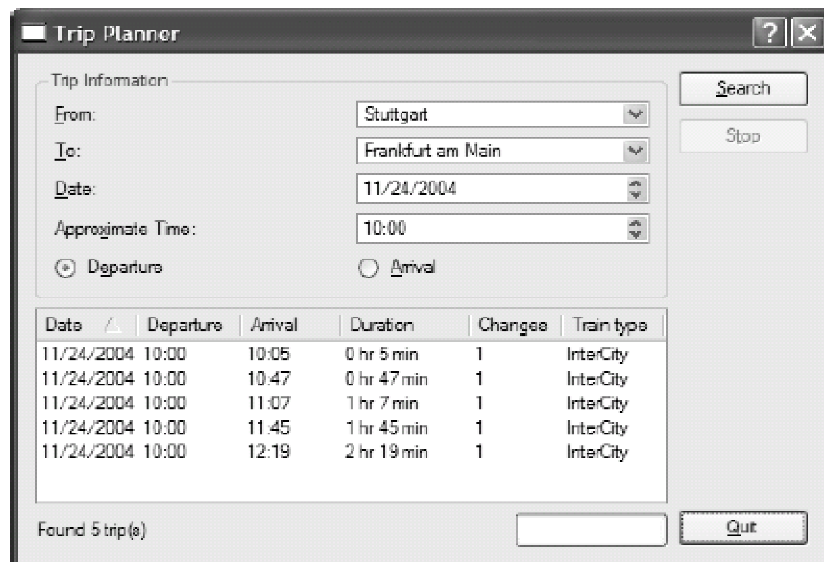


Рисунок 13.1. Внешний вид приложения Trip Planner.

На форме приложения находятся поля ввода **From** (Из), **To** (В), **Date** (Дата), **Approximate Time** (Примерное время) и две кнопки с зависимой фиксацией (**radio buttons**), которые уточняют смысл поля **pproximate Time** -- время отправления или время прибытия. Когда пользователь нажимает кнопку **Search**, приложение передает запрос серверу и получает список поездов, которые отвечают заданным критериям. Этот список отображается в виджете **QListView**. В самом низу формы находятся **QLabel**, для отображения результатов выполнения последнего запроса, и **QProgressBar**.

льзовательский интерфейс приложения был разработан в среде визуального построителя Qt Designer. Поэтому, все свое внимание мы сконцентрируем на содержимом файла **.ui.h**. Обратите внимание: следующие четыре переменные-члены были объявлены на вкладке **Members**, в построителе Qt Designer, как:

```
QSocket socket;
QTimer connectionTimer;
QTimer progressBarTimer;
Q_UINT16 blockSize;
```

Переменная **socket** отвечает за работу с TCP-соединением. Переменная **connectionTimer** используется для отслеживания тайм аута соединения. Переменная **progressBarTimer** предназначена для периодического обновления индикатора хода выполнения запроса. И наконец переменная **blockSize** используется при анализе блока данных, полученных от сервера.

```
void TripPlanner::init()
{
  connect(&socket, SIGNAL(connected()),
          this, SLOT(sendRequest()));
  connect(&socket, SIGNAL(connectionClosed()),
          this, SLOT(connectionClosedByServer()));
  connect(&socket, SIGNAL(readyRead()),
          this, SLOT(updateListView()));
  connect(&socket, SIGNAL(error(int)),
          this, SLOT(error(int)));
  connect(&connectionTimer, SIGNAL(timeout()),
          this, SLOT(connectionTimeout()));
  connect(&progressBarTimer, SIGNAL(timeout()),
          this, SLOT(advanceProgressBar()));

  QDateTime dateTime = QDateTime::currentDateTime();
  dateEdit->setDate(dateTime.date());
  timeEdit->setTime(QTime(dateTime.time().hour(), 0));
}
```

Функция **init()** связывает сигналы объекта **QSocket** -- **connected()**, **connectionClosed()**, **readyRead()** и **error(int)**, и сигналы **timeout()** от таймеров, с соответствующими слотами. Поля ввода **Date** и **Approximate Time** заполняются значениями по-умолчанию -- текущими датой и временем.

```
void TripPlanner::advanceProgressBar()
{
  progressBar->setProgress(progressBar->progress() + 2);
}
```

Слот advanceProgressBar() связан с сигналом timeout(), объекта progressBarTimer.

```
void TripPlanner::connectToServer()
{
  listView->clear();

  socket.connectToHost("tripserver.zugbahn.de", 6178);

  searchButton->setEnabled(false);
  stopButton->setEnabled(true);
  statusLabel->setText(tr("Connecting to server..."));
  connectionTimer.start(30 * 1000, true);
  progressBarTimer.start(200, false);
  blockSize = 0;
}
```

Слот **connectToServer()** вызывается по нажатию на кнопку **Search**. Функция вызывает **connectToHost()** для установления соединения с мифическим сервером **tripserver.zugbahn.de**, который ожидает поступления запросов на порту с номером 6178. (Если вы планируете опробовать пример на своей машине, замените имя удаленного сервера на localhost.) Функция **connectToHost()** работает асинхронно -- она всегда сразу же возвращает управление вызывающей программе. Само соединение устанавливается немного позже, в этот момент **QSocket** выдает сигнал **connected()**. В случае возникновении ошибки, выдается сигнал **error(int)** (с кодом ошибки).

После этого обновляется интерфейсная часть приложения и запускаются два таймера. Первый из них, **connectionTimer** -- это таймер с однократным срабатыванием. Он выдает сигнал **timeout()** через 30 секунд после запуска. Второй таймер, **progressBarTimer**, отрабатывает через каждые 200 миллисекунд. С его помощью выполняется обновление индикатора хода процесса.
И в заключении в переменную **blockSize** записывается значение 0. Она хранит размер очередного блока данных, принятого от сервера.

```
void TripPlanner::sendRequest()
{
  QByteArray block;
  QDataStream out(block, IO_WriteOnly);
  out.setVersion(5);
  out << (Q_UINT16)0 << (Q_UINT8)'S'
      << fromComboBox->currentText()
      << toComboBox->currentText()
      << dateEdit->date()
      << timeEdit->time();
  if (departureRadioButton->isOn())
    out << (Q_UINT8)'D';
  else
    out << (Q_UINT8)'A';
  out.device()->at(0);
  out << (Q_UINT16)(block.size() - sizeof(Q_UINT16));
  socket.writeBlock(block.data(), block.size());

  statusLabel->setText(tr("Sending request..."));
}
```

Слот **sendRequest()** связан с сигнвлом **connected()**, объекта **QSocket**. При появлении сигнала, слот генерирует запрос серверу, передавая информацию, введенную пользователем.

Блок запроса имеет следующую структуру:
Q_UINT16 Размер блока в байтах (исключая это поле)

Q_UINT8 Тип запроса (всегда 'S')
QString Пункт отправления
QString Пункт прибытия
QDate Дата
QTime Примерное время
Q_UINT8 Тип поля "Примерное время": 'D' -- отправление, 'A' -- прибытие.

Сначала данные записываются в объект **QByteArray**, который называется **block**. Записать данные напрямую в **QSocket** не представляется возможным, потому что размер блока заранее не известен. Изначально, в поле **size** записывается число **0**. Затем, после записи в блок всех данных, производится переход к началу блока, вызовом функции **at(0)** и записывается корректное значение размера передаваемого блока данных. После этого блок передается серверу, вызовом **writeBlock()**.

```
void TripPlanner::updateListView()
{
  connectionTimer.start(30 * 1000, true);

  QDataStream in(&socket);
  in.setVersion(5);
  for (;;) {
    if (blockSize == 0) {
      if (socket.bytesAvailable() < sizeof(Q_UINT16))
        break;
      in >> blockSize;
    }

    if (blockSize == 0xFFFF) {
      closeConnection();
      statusLabel->setText(tr("Found %1 trip(s)")
                           .arg(listView->childCount()));
      break;
    }

    if (socket.bytesAvailable() < blockSize)
      break;

    QDate date;
    QTime departureTime;
    QTime arrivalTime;
    Q_UINT16 duration;
    Q_UINT8 changes;
    QString trainType;

    in >> date >> departureTime >> duration >> changes
       >> trainType;
    arrivalTime = departureTime.addSecs(duration * 60);

    new QListViewItem(listView,
                      date.toString(LocalDate),
                      departureTime.toString(tr("hh:mm")),
                      arrivalTime.toString(tr("hh:mm")),
                      tr("%1 hr %2 min").arg(duration / 60)
                                        .arg(duration % 60),
                         QString::number(changes),
                         trainType);
    blockSize = 0;
  }
}
```

Слот **updateListView()** реагирует на сигнал **readyRead()**, **объекта QSocket**, который выдается при получении новых данных от сервера. Первое, что необходимо сделать -- это перезапустить таймер с однократным срабатыванием, отслеживающий тайм аут соединения. Всякий раз, когда от сервера приходит очередная порция данных, необходимо продлить срок "жизни" соединения еще на 30 секунд.
Сервер передает расписание движения поездов, которые удовлетворяют заданным критериям. Каждая строка расписания передается в виде отдельного блока и каждый блок начинается полем, содержащим размер блока. Сложность обработки данных в цикле **for** заключается в том, что от

сервера не все данные приходят одновременно. Мы можем получить блок целиком, или только часть блока, или полтора блока, или даже все блоки сразу.

Рисунок 13.2. Поток данных от Trip Server, разбитый на блоки.

Так как же работает цикл **for**? Если значение переменной **blockSize** равно 0, это означает, что размер очередного блока еще не прочитан. Значение 0xFFFF используется для индикации окончания передачи, поэтому, прочитав это значение, можно быть уверенным, что новых данных больше не поступит.
Если размер блока меньше 0xFFFF, то выполняется попытка прочитать блок, но прежде всего проверяется -- получен ли блок полностью. Если это не так, то цикл прерывается. При поступлении новой порции данных, снова будет выдан сигнал **readyRead()** и тогда можно будет повторить попытку.

После того как блок будет получен целиком, можно безопасно прочитать его оператором "**>>**", выделить нужную информацию и записать ее в объект класса **QListViewItem**. Блок, поступающий от сервера имеет следующую структуру:

Q_UINT16	Размер блока в байтах (исключая это поле)
QDate	Дата отправления
QTime	Время отправления
Q_UINT16	Время в пути (в минутах)
Q_UINT8	Количество остановок
QString	Тип поезда

Завершив разбор блока данных, функция записывает значение 0 в переменную **blockSize**, говоря о том, что размер очередного блока данных неизвестен.

```
void TripPlanner::closeConnection()
{
  socket.close();
  searchButton->setEnabled(true);
  stopButton->setEnabled(false);
  connectionTimer.stop();
  progressBarTimer.stop();
  progressBar->setProgress(0);
}
```

Функция **closeConnection()** закрывает соединение с сервером, обновляет интерфейс с пользователем и останавливает таймеры. Она вызывается из **updateListView()**, когда будет получен блок с размером 0xFFFF, и из некоторых других слотов, которые будут описаны чуть ниже.

```
void TripPlanner::stopSearch()
{
  statusLabel->setText(tr("Search stopped"));
  closeConnection();
}
```

Слот **stopSearch()** реагирует на нажите кнопки **Stop**. Суть его состоит в закрытии соединения вызовом функции **closeConnection()**.

```
void TripPlanner::connectionTimeout()
{
  statusLabel->setText(tr("Error: Connection timed out"));
  closeConnection();
}
```

Слот **connectionTimeout()** отрабатывает по истечении тайм аута соединения.

```
void TripPlanner::connectionClosedByServer()
```

```
{
  if (blockSize != 0xFFFF)
    statusLabel->setText(tr("Error: Connection closed by "
                            "server"));
    closeConnection();
  }
```

Слот **connectionClosedByServer()** реагирует на сигнал **connectionClosed()**, объекта **socket**. Если сервер закрыл соединение до того, как был получен маркер конца передачи (0xFFFF), пользователю выводится сообщение об ошибке. Затем вызывается **closeConnection()**, чтобы обновить интерфейс и остановить таймеры.

```
void TripPlanner::error(int code)
{
  QString message;

  switch (code) {
    case QSocket::ErrConnectionRefused:
        message = tr("Error: Connection refused");
        break;
    case QSocket::ErrHostNotFound:
        message = tr("Error: Server not found");
        break;
    case QSocket::ErrSocketRead:
    default:
        message = tr("Error: Data transfer failed");
  }
  statusLabel->setText(message);
  closeConnection();
}
```

Слот **error(int)** связан с сигналом **error(int)** сокета. Он генерирует текст сообщения, соответствующий полученному коду ошибки.

Функция **main()** не содержит ничего нового:

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

  TripPlanner tripPlanner;
  app.setMainWidget(&tripPlanner);
  tripPlanner.show();
  return app.exec();
}
```

Перейдем к реализации приложения-сервера. Сервер состоит из двух классов: **TripServer** и **ClientSocket**. Первый порожден от **QServerSocket** и предназначен для приема входящих соединений. Второй -- наследник **QSocket** предназначен для обслуживания одиночного соединения с клиентом. В каждый конкретный момент времени, в памяти приложения будет находиться столько экземпляров **ClientSocket**, сколько клиентов подключено к серверу.

```
class TripServer : public QServerSocket
{
public:
  TripServer(QObject *parent = 0, const char *name = 0);

  void newConnection(int socket);
};
```

В классе **TripServer** перекрыт родительский метод **newConnection()**. Эта функция вызывается всякий раз, когда сервер обнаруживает попытку соединения с ним.

```
TripServer::TripServer(QObject *parent, const char *name)
    : QServerSocket(6178, 1, parent, name)
{
}
```

Здесь, родительскому конструктору передается номер порта (6178). Второй аргумент, 1, это количество подключений, ожидающих обработки.

```
void TripServer::newConnection(int socketId)
{
  ClientSocket *socket = new ClientSocket(this);
  socket->setSocket(socketId);
}
```

В функции **newConnection()** создается новый объект класса **ClientSocket**, которому присваивается заданный идентификационный номер.

```
class ClientSocket : public QSocket
{
  Q_OBJECT
public:
  ClientSocket(QObject *parent = 0, const char *name = 0);

private slots:
  void readClient();

private:
  void generateRandomTrip(const QString &from, const QString &to,
                          const QDate &date, const QTime &time);

  Q_UINT16 blockSize;
};
```

Класс **ClientSocket** порожден от класса **QSocket** и отвечает за обслуживание одиночного соединения с клиентом.

```
ClientSocket::ClientSocket(QObject *parent, const char *name)
    : QSocket(parent, name)
{
  connect(this, SIGNAL(readyRead()),
          this, SLOT(readClient()));
  connect(this, SIGNAL(connectionClosed()),
          this, SLOT(deleteLater()));
  connect(this, SIGNAL(delayedCloseFinished()),
          this, SLOT(deleteLater()));
  blockSize = 0;
}
```

В конструкторе устанавливаются все необходимые соединения между сигналами и слотами, и записывается значение 0 в переменную **blockSize**.
Сигналы **connectionClosed()** и **delayedCloseFinished()** соединены со слотом **deleteLater()**. Эта функция унаследована от **QObject**. Она удаляет объект, когда управление переходит в цикл обработки событий. Она обеспечивает удаление экземпляров **ClientSocket** при закрытии соединения.

```
void ClientSocket::readClient()
{
  QDataStream in(this);
  in.setVersion(5);

  if (blockSize == 0) {
    if (bytesAvailable() < sizeof(Q_UINT16))
      return;
    in >> blockSize;
  }
  if (bytesAvailable() < blockSize)
    return;

  Q_UINT8 requestType;
  QString from;
  QString to;
  QDate date;
  QTime time;
```

```
  Q_UINT8 flag;
  in >> requestType;

  if (requestType == 'S') {
    in >> from >> to >> date >> time >> flag;

    srand(time.hour() * 60 + time.minute());
    int numTrips = rand() % 8;
    for (int i = 0; i < numTrips; ++i)
      generateRandomTrip(from, to, date, time);

    QDataStream out(this);
    out << (Q_UINT16)0xFFFF;
  }
  close();
  if (state() == Idle)
    deleteLater();
}
```

Слот **readClient()** связан с сигналом **readyRead()** сокета. Если переменная **blockSize** содержит 0, то выполняется попытка прочитать размер очередного блока данных, в противном случае предполагается, что размер уже прочитан и необходимо проверить -- поступил ли блок данных полностью. Если блок данных поступил целиком, то выполняется чтение блока. Чтение производится с помощью **QDataStream** напрямую из сокета (аргумент **this**).

После того как блок запроса прочитан, можно приступать к формированию ответа. Если бы это было реальное приложение, все необходимые сведения можно было бы брать из базы данных. Но здесь мы будем довольствоваться функцией **generateRandomTrip()**, которая генерирует расписание случайным образом. Функция будет вызываться случайное число раз и в конце передачи будет отправляться маркер конца передачи (0xFFFF).
В заключение -- соединение закрывается. Если выходной буфер сокета пуст, то соединение закрывается немедленно и можно вызвать **deleteLater()**, чтобы удалить сокет, когда управление попадет в цикл обработки событий. (Вполне безопасно было бы вызвать **delete this**.) В противном случае, сокет продолжит передачу данных и затем закроет соединение по сигналу **delayedCloseFinished()**.

```
void ClientSocket::generateRandomTrip(const QString &,
        const QString &, const QDate &date, const QTime &time)
{
  QByteArray block;
  QDataStream out(block, IO_WriteOnly);
  out.setVersion(5);
  Q_UINT16 duration = rand() % 200;
  out << (Q_UINT16)0 << date << time << duration
      << (Q_UINT8)1 << QString("InterCity");
  out.device()->at(0);
  out << (Q_UINT16)(block.size() - sizeof(Q_UINT16));

  writeBlock(block.data(), block.size());
}
```

Функция **generateRandomTrip()** показывает, как можно отправить блок данных через TCP-соединение. Это очень похоже на то, что мы уже видели в клиентском приложении (функция **sendRequest()**). Опять же, чтобы определить размер блока, данные сначала записываются в **QByteArray**, а затем передаются сокету вызовом **writeBlock()**.

```
int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  TripServer server;
  if (!server.ok()) {
    qWarning("Failed to bind to port");
    return 1;
  }
  QPushButton quitButton(QObject::tr("&Quit"), 0);
  quitButton.setCaption(QObject::tr("Trip Server"));
  app.setMainWidget(&quitButton);
```

```
  QObject::connect(&quitButton, SIGNAL(clicked()),
                   &app, SLOT(quit()));
  quitButton.show();
  return app.exec();
}
```

В функции **main()** создается экземпляр класса **TripServer** и кнопка **QPushButton**, с помощью которой пользователь может остановить сервер.
На этом мы завершаем рассмотрение примера построения клиентского и серверного приложений. В данном случае мы реализовали обмен по своему, блочно-ориентированному протоколу, что позволило нам использовать **QDataStream** для чтения и записи данных. Если бы мы занялись реализацией строково-ориентированного протокола, то в самом простейшем случае мы могли бы воспользоваться функциями класса **QSocket** -- **canReadLine()** и **readLine()**, при получении сигнала **readyRead()**:

```
    QStringList lines;
    while (socket.canReadLine())
      lines.append(socket.readLine());
```

После этого можно былобы обработать каждую прочитанную строку. Передача текстовых строк могла бы быть выполнена с помощью **QTextStream**, связанного с **QSocket**.
Серверное приложение, в данной реализации, довольно плохо масштабируется, при наличии большого числа подключений. Проблема состоит в том, что когда обслуживается одно подключение, приложение не в состоянии обслужить другие соединения. Более масштабируемый подход заключается в создании отдельного потока для каждого соединения. Но экземпляры класса **QSocket** могут использоваться только в том потоке, который содержит цикл обработки событий (запускаемый вызовом **QApplication::exec()**), по причинам, которые более подробно будут описаны в Главе 17. Решение проблемы заключается в использовании низкоуровневого класса **QSocketDevice**, который работает независимо от цикла обработки событий.

13.4 Протокол UDP и класс QSocketDevice

Класс **QSocketDevice** реализует низкоуровневый интерфейс для работы с протоколами UDP и TCP. В большинстве TCP-приложений, достаточно будет функциональности, заложенной в **QSocket**, но если в приложении необходимо работать с протоколом UDP, то тогда вам придется обратить свой взор на класс **QSocketDevice**.
UDP -- это протокол с негарантированной доставкой сообщений, ориентированный на работу с датаграммами. Некоторые нестандартные протоколы работают поверх UDP, поскольку он не такой "тяжелый" как TCP. По протоколу UDP, данные передаются в виде отдельных пакетов - датаграмм. В этом протоколе отсутствует понятие "соединения", если UDP-пакет теряется где-то в недрах сети, то система не получит уведомления об ошибке.

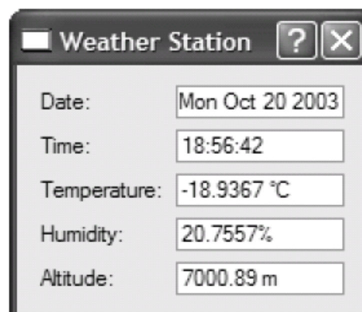


Рисунок 13.3. Внешний вид приложения TheWeather Station.

Рассмотрим принципы организации сетевых взаимодействий по протоколу UDP, на примере приложений **Weather Balloon** и **Weather Station**. Приложение **Weather Balloon** не имеет графического интерфейса. Его задача -- раз в 5 секунд отсылать UDP-датаграмму, которая содержит информацию о погодных условиях. Приложение **Weather Station** будет принимать эти датаграммы и отображать полученные сведения на экране. Начнем с приложения **Weather Balloon**.

```
class WeatherBalloon : public QPushButton
{
  Q_OBJECT
public:
  WeatherBalloon(QWidget *parent = 0, const char *name = 0);

  double temperature() const;
  double humidity() const;
  double altitude() const;

protected:
  void timerEvent(QTimerEvent *event);

private:
  QSocketDevice socketDevice;
  int myTimerId;
};
```

Класс **WeatherBalloon** порожден от **QPushButton**. Для взаимодествия с **Weather Station** он использует **QSocketDevice**.

```
WeatherBalloon::WeatherBalloon(QWidget *parent, const char *name)
    : QPushButton(tr("Quit"), parent, name),
        socketDevice(QSocketDevice::Datagram)
{
  socketDevice.setBlocking(false);
  myTimerId = startTimer(5 * 1000);
}
```

В списке инициализаторов конструктора, вызовом **QSocketDevice::Datagram**, создается устройство **QSocketDevice**. В теле конструктора, созданное устройство переводится в асинхронный режим работы, вызовом **setBlocking(false)**. (По-умолчанию, **QSocketDevice** работают в синхронном режиме.)

```
void WeatherBalloon::timerEvent(QTimerEvent *event)
{
  if (event->timerId() == myTimerId) {
    QByteArray datagram;
    QDataStream out(datagram, IO_WriteOnly);
    out.setVersion(5);
    out << QDateTime::currentDateTime() << temperature()
        << humidity() << altitude();
    socketDevice.writeBlock(datagram, datagram.size(),
                            0x7F000001, 5824);
  } else {
    QPushButton::timerEvent(event);
  }
}
```

В обработчике событий от таймера генерируется датаграмма, содержащая текущие дату, время, температуру воздуха, влажность и высоту над уровнем моря:

QDateTime	Дата и время измерений
double	Температура (в градусах Цельсия)
double	Влажность (в %)
double	Высота над уровнем моря (в метрах)

Передача датаграммы осуществляется вызовом **writeBlock()**. Третий и четвертый аргументы функции **writeBlock()** -- это IP-адрес и номер порта клиентского узла (**Weather Station**). В данном случае мы исходим из предположения, что оба приложения работают на одной машине, поэтому IP-адрес -- 127.0.0.1 (0x7F000001). В отличие от **QSocket**, экземпляры класса **QSocketDevice** не работает с сетевыми именами компьютеров, ему нужны IP-адреса. Если вам необходимо преобразовать имя хоста в его IP-адрес, это можно сделать с помощью класса **QDns**.

```
int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  WeatherBalloon balloon;
  balloon.setCaption(QObject::tr("Weather Balloon"));
```

```
  app.setMainWidget(&balloon);
  QObject::connect(&balloon, SIGNAL(clicked()),
                   &app, SLOT(quit()));
  balloon.show();
  return app.exec();
}
```

Функция **main()** просто создает объект **WeatherBalloon**, который выступает в двух ипостасях: как UDP-узел и как кнопка **QPushButton** на экране.
Теперь перейдем к приложению **Weather Station**.

```
class WeatherStation : public QDialog
{
  Q_OBJECT
public:
  WeatherStation(QWidget *parent = 0, const char *name = 0);

private slots:
  void dataReceived();

private:
  QSocketDevice socketDevice;
  QSocketNotifier *socketNotifier;

  QLabel *dateLabel;
  QLabel *timeLabel;
  ...
  QLineEdit *altitudeLineEdit;
};
```

Класс **WeatherStation** порожден от **QDialog**. Его назначение -- ожидать поступления датаграмм на определенном порту, производить разбор поступившей информации (от **Weather Balloon**) и отображать ее в пяти компонентах **QLineEdit**.
Класс содержит две приватные переменные, которые представляют для нас наибольший интерес: **socketDevice** и **socketNotifier**. Первая переменная имеет тип **QSocketDevice**. Она используется для приема датаграмм. Вторая переменная имеет тип **QSocketNotifier**. Она используется для того, чтобы известить приложение о поступлении датаграммы.

```
WeatherStation::WeatherStation(QWidget *parent, const char *name)
    : QDialog(parent, name), socketDevice(QSocketDevice::Datagram)
{
  socketDevice.setBlocking(false);
  socketDevice.bind(QHostAddress(), 5824);
  socketNotifier = new QSocketNotifier(socketDevice.socket(),
                                        QSocketNotifier::Read,
                                        this);
  connect(socketNotifier, SIGNAL(activated(int)),
          this, SLOT(dataReceived()));
  ...
}
```

В списке инициализаторов конструктора создается устройство **QSocketDevice**, вызовом **QSocketDevice::Datagram**. В теле конструктора, вызовом **setBlocking(false)**, оно переводится в асинхронный режим работы. Вызовом функции **bind()**, сокету назначаются IP-адрес и номер порта. Вызовом функции **QHostAddress()** мы сообщаем, что будем принимать датаграммы, отправляемые на любой IP-адрес, который принадлежит компьютеру с запущенным приложением **Weather Station**. Затем создается объект **QSocketNotifier**, который будет "следить" за сокетом. Объект **QSocketNotifier** будет выдавать сигнал **activated(int)** в момент поступления датаграммы. Этот сигнал соединяется со слотом **dataReceived()**.

```
void WeatherStation::dataReceived()
{
  QDateTime dateTime;
  double temperature;
  double humidity;
  double altitude;
```

```
  QByteArray datagram(socketDevice.bytesAvailable());
  socketDevice.readBlock(datagram.data(), datagram.size());

  QDataStream in(datagram, IO_ReadOnly);
  in.setVersion(5);
  in >> dateTime >> temperature >> humidity >> altitude;

  dateLineEdit->setText(dateTime.date().toString());
  timeLineEdit->setText(dateTime.time().toString());
  temperatureLineEdit->setText(tr("%1 C").arg(temperature));
  humidityLineEdit->setText(tr("%1%").arg(humidity));
  altitudeLineEdit->setText(tr("%1 m").arg(altitude));
}
```

В функции **dataReceived()** производится чтение датаграммы, вызовом **readBlock()**. Функция **QByteArray::data()** возвращает указатель на данные в **QByteArray**, по которому **readBlock()** запишет полученные сведения. Затем производится извлечение отдельных значений, которые заносятся в визуальные компоненты для отображения на экране. С точки зрения приложения, датаграмма всегда передается и принимается как единый блок данных. Это означает, что если доступен хотя бы один байт, то следовательно датаграмма получена целиком.

```
int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  WeatherStation station;
  app.setMainWidget(&station);
  station.show();
  return app.exec();
}
```

В заключение, в функции **main()** создается объект **WeatherStation** и назначается главным виджетом приложения.
На этом мы заканчиваем рассмотрение принципов работы с UDP. Мы постарались создать приемное и передающее приложения настолько простыми, насколько это возможно. В большинстве реальных применений, приложения должны как передавать, так и принимать датаграммы. В составе класса **QSocketDevice** имеются функции **peerAddress()** и **peerPort()**, которые могут использоваться для определения IP-адреса и номера порта, на которые нужно послать ответ.

14 XML

XML (от англ. Extensible Markup Language -- Расширяемый Язык Разметки) -- популярный формат файлов, используемый для обмена и хранения данных в текстовом виде.

Для работы с XML документами, Qt поддерживает два различных API:

- SAX (от англ. Simple API for XML -- Простейший Прикладной Интерфейс для работы с XML) -- используется для выполнения синтаксического анализа методом обработки событий разбора прямо в приложении, с помощью виртуальных функций.
- DOM (от англ. Document Object Model -- Объектная Модель представления Документов) -- преобразует XML-документ в древовидную структуру, в результате приложение получает возможность навигации по ней.

В каждом конкретном случае, при выборе того или иного API, необходимо учитывать множество факторов. SAX -- более быстрый, он больше подходит для выполнения простых задач (например, чтобы найти все вхождения заданного тега в документе), и для работы с XML-файлами огромного размера, которые могут не уместиться в памяти целиком. DOM -- более удобен, в большинстве приложений, фактор удобства перевешивает быстроту и нетребовательность SAX.
В этой главе мы покажем, как работать с XML-файлами посредством обоих API.

14.1 Чтение XML-документов с помощью SAX

SAX -- это (де-факто) Java API стандарт для чтения XML-документов. Классы SAX, в библиотеке Qt, моделируют реализацию SAX2 Java, с небольшими отличиями в именованиях. Дополнительную информацию о SAX вы найдете по адресу: *http://www.saxproject.org/*.

Qt предоставляет SAX-парсер **QXmlSimpleReader**. Он распознает правильно оформленные XML-документы и поддерживает пространства имен XML. Во время анализа документа вызываются виртуальные функции классов-обработчиков событий разбора. (В данном случае, понятие "событие разбора" никак не пересекается с понятием событий в Qt.) Например, предположим, что парсер анализирует XML-документ со следующим содержимым:

```
<doc>
  <quote>Errare humanum est</quote>
</doc>
```

В этом случае парсер мог бы вызвать следующие обработчики событий разбора:

```
startDocument()
startElement("doc")
startElement("quote")
characters("Errare humanum est")
endElement("quote")
endElement("doc")
endDocument()
```

Все вышеприведенные функции определены в классе **QXmlContentHandler**. С целью упрощения примера мы не приводим некоторые аргументы в функциях **startElement()** и **endElement()**.

Класс **QXmlContentHandler** -- лишь один из многих, которые могут работать совместно с **QXmlSimpleReader**. Среди других классов можно назвать: **QXmlEntityResolver**, **QXmlDTDHandler**, **QXmlErrorHandler**, **QXmlDeclHandler** и **QXmlLexicalHandler**. Они реализуют исключительно виртуальные функции и предоставляют сведения о различного типа событиях разбора. В большинстве приложений используются только два класса: **QXmlContentHandler** и **QXmlErrorHandler**.

Для большего удобства, Qt так же предоставляет класс **QXmlDefaultHandler**, который наследует (через множественное наследование) и реализует все виртуальные функции других классов-обработчиков. Такая архитектура, со множеством абстрактных классов и единственным классом-наследником, довольно необычна для Qt, однако, она была принята в соответствии с моделью реализации, принятой в Java.

Рассмотрим на примере, как можно использовать классы **QXmlSimpleReader** и **QXmlDefaultHandler** для разбора XML-файла и отображения его содержимого в QListView. Наш класс, производный от класса **QXmlDefaultHandler**, будет называться **SaxHandler**. В его задачи будет входить разбор XML-документа, представляющего собой список терминов, использовавшихся в книге.

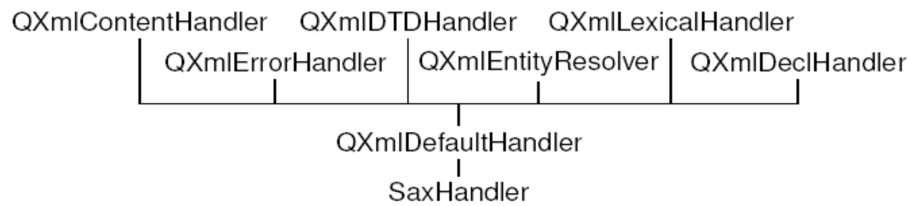


Рисунок 14.1. Дерево наследования класса SaxHandler.

Ниже приведен XML-файл, содержимое которого отображается в **QListView**, на рисунке 14.2.

```
<?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>
```

Terms	Pages
sidebearings	10, 34-35, 307-308
subtraction	
of pictures	115, 244
of vectors	9

Рисунок 14.2. Файл со списком терминов, использованных в книге, загруженный в QListView.

Сначала создадим определение класса-обработчика:

```
class SaxHandler : public QXmlDefaultHandler
{
public:
  SaxHandler(QListView *view);

  bool startElement(const QString &namespaceURI,
                    const QString &localName,
                    const QString &qName,
                    const QXmlAttributes &attribs);
  bool endElement(const QString &namespaceURI,
                  const QString &localName,
                  const QString &qName);
  bool characters(const QString &str);
  bool fatalError(const QXmlParseException &exception);

private:
  QListView *listView;
  QListViewItem *currentItem;
  QString currentText;
};
```

Класс **SaxHandler** порожден от класса **QXmlDefaultHandler** и перекрывает четыре метода родителя: **startElement()**, **endElement()**, **characters()** и **fatalError()**. Первые три функции объявлены в классе **QXmlContentHandler**, последняя функция -- в **QXmlErrorHandler**.

```
SaxHandler::SaxHandler(QListView *view)
{
  listView = view;
  currentItem = 0;
}
```

Конструктор получает указатель на **QListView**, который будет заполняться информацией из XML-файла.

```
bool SaxHandler::startElement(const QString &, const QString &,
                              const QString &qName,
                              const QXmlAttributes &attribs)
{
  if (qName == "entry") {
    if (currentItem) {
      currentItem = new QListViewItem(currentItem);
    } else {
      currentItem = new QListViewItem(listView);
    }
    currentItem->setOpen(true);
    currentItem->setText(0, attribs.value("term"));
  } else if (qName == "page") {
    currentText = "";
  }
  return true;
}
```

Функция **startElement()** вызывается, когда парсер встречает новый открывающий тег. Третий аргумент -- это имя тега. Четвертый -- список атрибутов. В данном примере мы будем игнорировать первый и второй аргументы. Они предназначены для работы с XML-файлами, которые используют механизм пространств имен.

Если это тег **<entry>**, создается новый элемент списка **QListView**. Если анализируемый тег вложен в другой тег **<entry>**, создается вложенный подэлемент списка -- **QListViewItem**. В противном случае создается элемент списка верхнего уровня. Функция **setOpen(true)** вызывается для того, чтобы открыть вложенные подэлементы данного элемента. Функция **setText()** записывает текст (значение атрибута **term**), который будет отображаться на экране в первой колонке списка.

Если это тег **<page>**, то в **currentText** записывается пустая строка. Переменная **currentText** служит своего рода аккумулятором для текста, размещаемого между тегами **<page>** и **</page>**.
В заключение, в вызывающую программу возвращается **true**, чтобы сообщить парсеру SAX о том, что он может продолжить разбор файла. В случае неопознанного тега, можно вернуть **false**, чтобы известить парсер об ошибке. В этом случае необходимо тогда перекрыть метод **errorString()**, унаследованный от **QXmlDefaultHandler**, чтобы вернуть соответствующее сообщение об ошибке.

```
bool SaxHandler::characters(const QString &str)
{
  currentText += str;
  return true;
}
```

Функция **characters()** вызывается для передачи символьных данных из XML-файла. В нашем случае мы просто добавляем их в конец переменной **currentText**.

```
bool SaxHandler::endElement(const QString &, const QString &,
                            const QString &qName)
{
  if (qName == "entry") {
    currentItem = currentItem->parent();
  } else if (qName == "page") {
    if (currentItem) {
```

```
      QString allPages = currentItem->text(1);
      if (!allPages.isEmpty())
        allPages += ", ";
      allPages += currentText;
      currentItem->setText(1, allPages);
    }
  }
  return true;
}
```

Функция **endElement()** вызывается, когда парсер встречает закрывающий тег. Аналогично функции **startElement()**, третьим аргументом ей передается имя тега.
Если это тег **</entry>**, то текущим назначается элемент более высокого уровня. Таким образом восстанавливается значение переменной, которое предшествовало открывающему тегу **<entry>**.

Если это тег **</page>**, производится добавление номеров в список страниц, которые отображаются во второй колонке списка.

```
bool SaxHandler::fatalError(const QXmlParseException &exception) {
  qWarning("Line %d, column %d: %s", exception.lineNumber(),
           exception.columnNumber(), exception.message().ascii());
  return false;
}
```

Функция **fatalError()** вызывается, когда парсер не может продолжить разбор XML-файла. Тогда мы просто выводим сообщение, с указанием номера строки и позиции в строке, где была обнаружена ошибка.

На этом мы завершаем обзор реализации класса **SaxHandler** и переходим к демонстрации практического его применения:

```
bool parseFile(const QString &fileName)
{
  QListView *listView = new QListView(0);
  listView->setCaption(QObject::tr("SAX Handler"));
  listView->setRootIsDecorated(true);
  listView->setResizeMode(QListView::AllColumns);
  listView->addColumn(QObject::tr("Terms"));
  listView->addColumn(QObject::tr("Pages"));
  listView->show();

  QFile file(fileName);
  QXmlSimpleReader reader;

  SaxHandler handler(listView);
  reader.setContentHandler(&handler);
  reader.setErrorHandler(&handler);
  return reader.parse(&file);
}
```

Сначала создается виджет **QListView** с двумя колонками. Затем создаются объект **QFile**, посредством которого будет выполняться работа с файлом XML-документа, и **QXmlSimpleReader** -- сам парсер. У нас нет необходимости открывать файл -- за нас это сделает сама библиотека Qt.

В заключение создается объект **SaxHandler**. Мы передаем его парсеру, как обработчик событий разбора и как обработчик ошибок. И наконец запускаем процесс разбора, вызовом **parse()**.

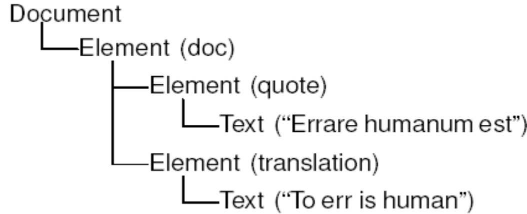
14.2 Чтение XML-документов с помощью DOM

DOM -- это стандарт API, для разбора XML-документов, разработанный в недрах World Wide Web Consortium (W3C). Qt предоставляет реализацию DOM Level 2 для чтения, изменения и записи XML-документов.

DOM представляет XML-файл в памяти, в виде древовидной структуры. У приложения имеется возможность перемещаться по этой структуре, как ему заблагорассудится. Программа может изменить содержимое дерева и сохранить его обратно в файл.
Рассмотрим следующий XML-документ:

```
<doc>
  <quote>Errare humanum est</quote>
  <translation>To err is human</translation>
</doc>
```

Соответствующее ему дерево DOM:



Дерево состоит из узлов различного типа. Например, узел **Element** соответствует открывающему и парному закрывающему тегам. Все, что находится между ними, отображается в виде дочерних узлов. В Qt, имена классов узлов начинаются с префикса **QDom**. Таким образом, класс **QDomElement** представляет узел **Element**, а **QDomText** -- узел **Text**.

Различные типы узлов могут включать в себя различные типы дочерних узлов. Например, узел **Element** может содержать другие узлы типа **Element**, а так же **Entity Reference**, **Text**, **CDATA Section**, **Processing Instruction** и **Comment**. На рисунке 14.3 показано, какие типы узлов, в состав каких типов могут входить.

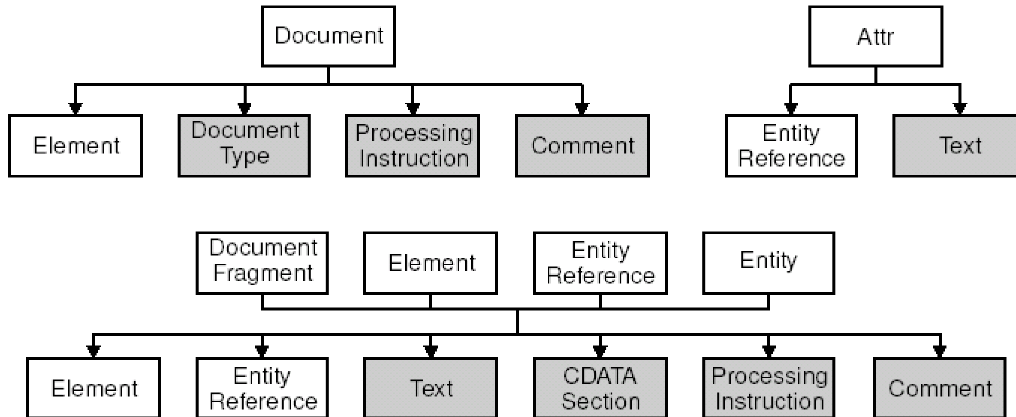


Рисунок 14.3. Взаимоотношения между типами узлов в DOM.

Для демонстрации работы с XML-документами через API DOM, мы напишем парсер, который будет анализировать XML-файл, представленный выше.

```
class DomParser
{
public:
  DomParser(QIODevice *device, QListView *view);

private:
  void parseEntry(const QDomElement &element,
                  QListViewItem *parent);

  QListView *listView;
};
```

Наш класс **DomParser** будет производить анализ XML-файла и выводить его содержимое в **QListView**. Этот класс не имеет предка.

```
DomParser::DomParser(QIODevice *device, QListView *view)
{
  listView = view;

  QString errorStr;
  int errorLine;
  int errorColumn;
  QDomDocument doc;
  if (!doc.setContent(device, true, &errorStr, &errorLine,
                      &errorColumn)) {
    qWarning("Line %d, column %d: %s", errorLine, errorColumn,
             errorStr.ascii());
    return;
  }

  QDomElement root = doc.documentElement();
  if (root.tagName() != "bookindex") {
    qWarning("The file is not a bookindex file");
    return;
  }

  QDomNode node = root.firstChild();
  while (!node.isNull()) {
    if (node.toElement().tagName() == "entry")
      parseEntry(node.toElement(), 0);
    node = node.nextSibling();
  }
}
```

В конструкторе создается объект **QDomDocument** и вызывается его метод **setContent()**, чтобы прочитать XML-документ из **QIODevice**. Она автоматически открывает устройство. Затем вызывается **documentElement()**, чтобы получить корневой узел (со всеми дочерними узлами), и проверяется -- является ли этот элемент тегом **<bookindex>**. После этого выполняются итерации по всем дочерним узлам и если встречен тег **<entry>**, вызывается **parseEntry()** для его анализа.
Класс **QDomNode** может хранить узлы любого типа. Если вы собираетесь обрабатывать узлы какого-то конкретного типа, нужно сначала выполнить соответствующее преобразование. В этом примере нас интересуют только узлы типа **Element**, поэтому мы выполняем преобразование вызовом метода **toElement()**, чтобы получить узел типа QDomElement, и затем вызываем **tagName()**, чтобы прочитать имя тега. Если узел относится к другому типу, то функция **toElement()** вернет пустой объект **QDomElement**, с пустым именем тега.

```
void DomParser::parseEntry(const QDomElement &element,
                           QListViewItem *parent)
{
  QListViewItem *item;
  if (parent) {
    item = new QListViewItem(parent);
  } else {
    item = new QListViewItem(listView);
  }
  item->setOpen(true);
  item->setText(0, element.attribute("term"));

  QDomNode node = element.firstChild();
  while (!node.isNull()) {
    if (node.toElement().tagName() == "entry") {
      parseEntry(node.toElement(), item);
    } else if (node.toElement().tagName() == "page") {
      QDomNode childNode = node.firstChild();
      while (!childNode.isNull()) {
        if (childNode.nodeType() == QDomNode::TextNode) {
          QString page = childNode.toText().data();
          QString allPages = item->text(1);
          if (!allPages.isEmpty())
            allPages += ", ";
```

```
          allPages += page;
          item->setText(1, allPages);
          break;
        }
        childNode = childNode.nextSibling();
      }
    }
    node = node.nextSibling();
  }
}
```

В функции **parseEntry()** создается элемент списка **QListView**. Если тег вложен в другой тег **<entry>**, то создается вложенный элемент списка. В противном случае создается элемент списка верхнего уровня. Чтобы открыть элемент списка, вызывается **setOpen(true)** и затем в него записывается текст, отображаемый в первой колонке списка, вызовом функции **setText()** (содержимое атрибута **term**). После инициализации **QListViewItem**, выполняются итерации по всем вложенным узлам, соответствующим данному тегу **<entry>**.
Если встречен тег **<entry>**, вызывается функция **parseEntry()**, которой, в качестве второго аргумента передается, текущий элемент списка. В результате будет создан новый элемент списка, вложенный в текущий.
Если встречен тег **<page>**, выполняется поиск узла **Text**. После того как он будет найден, выполняется преобразование узла, функцией **toText()**, в **QDomText** и из него извлекается текст в виде **QString**. Полученный таким образом текст добавляется в список номеров страниц, который отображается во второй колонке **QListViewItem**.
Теперь покажем, как можно использовать полученный класс **DomParser**:

```
void parseFile(const QString &fileName) {
  QListView *listView = new QListView(0);
  listView->setCaption(QObject::tr("DOM Parser"));
  listView->setRootIsDecorated(true);
  listView->setResizeMode(QListView::AllColumns);
  listView->addColumn(QObject::tr("Terms"));
  listView->addColumn(QObject::tr("Pages"));
  listView->show();

  QFile file(fileName);
  DomParser(&file, listView);
}
```

Сначала создается и настраивается **QListView**. Затем создаются **QFile** и **DomParser**. Во время создания, **DomParser** выполняет разбор XML-документа и заполняет список.
Как показывает пример, навигация по DOM-дереву может оказаться весьма громоздкой. Простое извлечение текста, заключенного между тегами **<page>** и **</page>** потребовало от нас выполнения итераций по всему списку дочерних узлов, с помощью функций **firstChild()** и **nextSibling()**.
Программисты, которые часто сталкиваются с необходимостью выполнения синтаксического анализа XML-документов, нередко пишут свои высокоуровневые классы-обертки, упрощающие выполнение наиболее часто используемых операций, таких как извлечение текста, заключенного между тегами.

14.3 Запись в XML-документы

Существует два основных подхода создания XML-файлов в приложениях Qt:
- Можно построить дерево DOM и затем вызвать метод **save()**.
- Можно вручную создать XML-файл

Выбор того или иного метода зачастую сильно зависит от используемого парсера -- SAX или DOM.
Ниже приводится отрывок кода, который создает дерево DOM и записывает его в файл, с помощью **QTextStream**:

```
  const int Indent = 4;

  QDomDocument doc;
  QDomElement root = doc.createElement("doc");
  QDomElement quote = doc.createElement("quote");
  QDomElement translation = doc.createElement("translation");
```

```
  QDomText quoteText = doc.createTextNode("Errare humanum est");
  QDomText translationText = doc.createTextNode("To err is human");

  doc.appendChild(root);
  root.appendChild(quote);
  root.appendChild(translation);
  quote.appendChild(quoteText);
  translation.appendChild(translationText);

  QTextStream out(&file);
  doc.save(out, Indent);
```

Вторым аргументом функции **save()** передается размер отступов. Ненулевое значение обеспечивает более удобочитаемый вид файла:

```
<doc>
    <quote>Errare humanum est</quote>
    <translation>To err is human</translation>
</doc>
```

Другой вариант применим в приложениях, которые используют структуру DOM-дерева для внутренней организации данных. Такие приложения, как правило, читают XML-документы с помощью DOM-парсера в память, затем изменяют содержимое дерева и сохраняют изменения с помощью **save()** в XML-файл.
В примерах выше использовалась кодировка UTF-8, однако существует возможность сохранения данных в других кодировках. Для этого достаточно добавить в начало XML-файла соответствующую кодировку:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
```

Следующий отрывок кода показывает, как это можно сделать:

```
  QTextStream out(&file);
  QDomNode xmlNode = doc.createProcessingInstruction("xml",
                          "version=\"1.0\" encoding=\"ISO-8859-1\"");
  doc.insertBefore(xmlNode, doc.firstChild());
  doc.save(out, Indent);
```

Создание XML-файлов вручную выполняется ничуть не сложнее. Для этого можно воспользоваться классом **QTextStream** и записать в него строки, как в обычный текстовый файл. Самое сложное в этом случае -- выполнить правильное экранирование служебных символов и значений атрибутов. Сделать это можно с помощью отдельной функции:

```
QString escapeXml(const QString &str)
{
  QString xml = str;
  xml.replace("&", "&amp;");
  xml.replace("<", "&lt;");
  xml.replace(">", "&gt;");
  xml.replace(" ", "&apos;");
  xml.replace("\"", "&quot;");
  return xml;
}
```

Ниже приводится пример использования этой функции:

```
  QTextStream out(&file);
  out.setEncoding(QTextStream::UnicodeUTF8);
  out << "<doc>\n"
      << "    <quote>" << escapeXml(quoteText) << "</quote>\n"
      << "    <translation>" << escapeXml(translationText)
      << "</translation>\n"
      << "</doc>\n";
```

В ежеквартальнике Qt Quarterly вы найдете статью "Generating XML" (по адресу: *http://doc.trolltech.com/qq/qq05-generating-xml.html*, в которой приводится пример простого класса,

создающего XML-файлы. Класс сам обслуживает экранирование служебных символов, вставляет отступы и задает кодировку символов, предоставляя нам возможность сконцентрироваться на содержимом XML-файла.

15 Интернационализация

В этой главе мы коснемся проблем интернационализации приложений, написаных с помощью библиотеки Qt. Первый раздел полностью посвящен Unicode -- кодировке символов, принятой в Qt. Сведения, которые содержатся в этом разделе, будут полезны всем Qt-разработчикам, поскольку даже если приложение разрабатывалось исключительно с англоязычным интерфейсом, рано или поздно оно может оказаться запущенным на машине грека или японца.
Во втором разделе будет показано, как разрабатывать приложения, которые потом можно будет перевести на другие языки. Этот процесс настолько прост, что не стоит отказываться от такой возможности, даже если вы не планируете выпускать локализованные версии приложения.
В третьем разделе будет рассказываться об истинно интернациональных приложениях. Здесь же мы покажем, как изменить язык интерфейса прямо во время исполнения приложения.
В последнем разделе мы опишем процесс перевода приложений на другие языки. А так же покажем, как программист и переводчик могут совместно работать над проектом, пользуясь утилитой Qt Linguist и другим инструментарием.

15.1 Unicode

Unicode -- это стандарт кодировки символов, который поддерживает большинство систем записи символов. Первоначально, идея Unicode состояла в том. чтобы каждый из символов кодировался не 8-ю, а 16-ю битами, что дает возможность определить 65536 символов, вместо 256. Наборы символов ASCII и ISO 8859-1 (Latin-1) являются поднаборами Unicode, и сохранили числовые значения своих символов. Например, Символ 'A' имеет значение 0x41 в ASCII, Latin-1 и Unicode.

Класс **QString** хранит строки как Unicode. Каждый символ в QString является 16-ти битным **QChar**, а не 8-ми битным **char**. Ниже приводится два способа записи символа 'A' в строку:

```
str[0] = 'A';
str[0] = QChar(0x41);
```

Мы можем записать любой из символов Unicode, по его числовому значению. Например, так можно записать символ греческого алфавита ('Σ') и знак евро ('€')

```
str[0] = QChar(0x3A3);
str[0] = QChar(0x20AC);
```

Числовые значения всех символов Unicode вы найдете по адресу *http://www.unicode.org/unicode/standard/standard.html*. Если у вас не возникает необходимости использовать не Latin-1 символы, вам достаточно будет ознакомиться с кодами символов в общих чертах. С другой стороны Qt предлагает более удобный способ ввода символов Unicode в программах, как -- будет описано немного ниже.

Текстовый движок Qt 3.2 поддерживает следующие наборы символов на всех платформах: Арабский, Китайский, Кириллический, Греческий, Иврит, Японский, Корейский, Лао, Латинский, Тайский и Вьетнамский. Кроме этого, на платформах X11 с Xft и Windows NT, дополнительно поддерживаются следующие наборы символов: Бенгальский, Девангари, Гуджарати, Каннада, Кхмерский, Сирийский, Тамильский, Телугу и Тана. На X11 поддерживаются еще Малайский и Тибетский наборы символов, а на Windows XP дополнительно поддерживается Дивехи. Если исходить из предположения, что в системе установлены соответствующие шрифты, Qt будет в состоянии отображать все символы из этих наборов.

Принципы работы с **QChar**, в программах, несколько отличается от принципов работы с **char**. Чтобы получить числовое значение символа **QChar**, нужно вызвать метод этого класса **unicode()**. Чтобы получить код символа ASCII или Latin-1, класса **QChar** нужно вызвать метод этого класса **latin1()**. Если символ не относится к поднабору Latin-1, **latin1()** вернет 0.
Если заранее известно, что программа будет работать исключительно с символами ASCII или Latin-1, можно использовать функции из <**cctype**>, такие как: **isalpha()**, **isdigit()** и **isspace()**. Они будут работать безотказно, потому что символы **QChar** автоматически преобразуются в **char**, в данном контексте, так же как и **QString** автоматически преобразуется в **const char ***. Однако, в любом случае лучше

пользоваться функциями-членами класса **QChar**, для выполнения подобных операций, поскольку они будут корректно работать с любыми символами Unicode. Среди функций, которые предоставляются классом **QChar**, можно назвать: **isPrint()**, **isPunct()**, **isSpace()**, **isMark()**, **isLetter()**, **isNumber()**, **isLetterOrNumber()**, **isDigit()**, **isSymbol()**, **lower()** и **upper()**. Например, так можно проверить -- является ли символ цифрой или символом верхнего регистра:

```
    if (ch.isDigit() || ch != ch.lower())
      ...
```

Функция **lower()** возвращает версию символа в нижнем регистре. Если результат функции отличается от оригинального символа, значит это символ верхнего регистра (или заглавный символ). Этот отрывок кода справедлив для языков, которые различают регистр символов (прописные и строчные), включая латиницу, кириллицу и набор греческих символов.

Как только мы начинаем работать с Unicode-строками, у нас появляется возможность использовать их повсюду, где Qt API ожидает получить **QString**. В свою очередь, Qt берет на себя ответственность по корректному отображению символов строки и преобразованию в другие кодировки, если в этом возникает необходимость.

Особую осторожность нужно проявлять при работе с текстовыми файлами. Они могут содержать текст в самых разнообразных кодировках, определить которую, зачастую практически невозможно. По-умолчанию **QTextStream** использует системную 8-ми битную кодировку символов (**QTextCodec::codecForLocale()**), как для записи, так и для чтения файлов.

Если у вас есть желание писать в файлы любые Unicode символы, можно предложить сохранять данные как Unicode, для этого, непосредственно перед записью данных, с помощью **QTextStream**, нужно вызвать функцию **setEncoding(QTextStream::Unicode)**. В результате текст будет записан в файл в кодировке UTF-16, где каждый символ представлен двумя байтами. Формат UTF-16 очень близок к представлению **QString** в памяти, поэтому чтение/запись строк UTF-16 выполняется очень быстро. Однако, этот формат довольно расточителен в случае символов ASCII, для хранения которых достаточно одного байта.

При чтении данных из файла, **QTextStream** обычно автоматически определяет Unicode, но для полной уверенности, перед выполнением процедуры чтения, лучше все-таки вызвать **setEncoding(QTextStream::Unicode)**.

Еще одна кодировка, которая поддерживает весь набор символов Unicode -- это UTF-8. Ее основное преимущество перед UTF-16 состоит в том, что для хранения символов ASCII (символы в диапазоне 0x00..0x7F) она использует всего один байт. Все остальные символы, включая символы Latin-1, числовые значения которых лежат выше 0x7F, представлены многобайтными последовательностями. Для хранения текста, состоящего преимущественно из ASCII-символов, в формате UTF-8 потребуется практически в два раза меньше пространства, чем в UTF-16. Чтобы использовать для записи/чтения текстовых файлов формат UTF-8, предварительно нужно вызвать **setEncoding(QTextStream::UnicodeUTF8)**.

Если предполагается использование исключительно кодировки Latin-1, вне зависимости от настроек локали пользователя, можно вызвать **setEncoding(QTextStream::Latin1)**.

Другие виды кодировки могут быть установлены с помощью вызова функции **setCodec()**, передав ей соответствующий **QTextCodec**. Класс **QTextCodec** выполняет преобразование между Unicode и заданной кодировкой. Экземпляры этого класса очень широко используются в библиотеке Qt. Они используются для поддержки шрифтов, методов ввода, буфера обмена, механизма "drag-and-drop" и именования файлов.

Рассмотрим такой пример: допустим, что нам необходимо прочитать файл, записанный в кодировке EUC-KR, тогда мы могли бы написать такой код:

```
  QTextStream in(&file);
  QTextCodec *koreanCodec = QTextCodec::codecForName("EUC-KR");
```

```
  if (koreanCodec)
    in.setCodec(koreanCodec);
```

Некоторые форматы файлов могут содержать указание о кодировке символов в области заголовка. В данном случае, заголовок -- это некая область в начале файла, которая содержит исключительно ASCII-символы, чтобы иметь гарантированную возможность их чтения, независимо от настроек локали. Наиболее типичный пример -- файлы формата XML, которые, как правило, используют кодировку UTF-8 или UTF-16. Самый правильный способ настройки **QTextStream**, перед работой с XML-файлами, это вызвать **setEncoding(QTextStream::UnicodeUTF8)**. Если файл ранее был сохранен в UTF-16, **QTextStream** автоматически определит это обстоятельство и скорректирует свои настройки. Заголовок XML-файла, иногда может содержать описание кодировки в заголовке **<?xml?>** (аргумент **encoding**), например:

```
<?xml version="1.0" encoding="EUC-KR"?>
```

Поскольку **QTextStream** не допускает изменения настройки кодировки после начала процедуры чтения, то наиболее правильный подход заключается в том, чтобы начать чтение файла заново, после того как будет задана правильная кодировка (может быть получена с помощью **QTextCodec::codecForName()**).
Но не стоит забывать, что в случае XML, мы можем использовать специализированные классы Qt, предназначенные для работы с данными файлами (см. Главу 14), что избавит нас от необходимости беспокоиться о кодировке файлов.

Еще одна область применения **QTextCodec** -- задание кодировки для строк, размещаемых в исходном тексте программ. Рассмотрим такой пример: группа японских программистов разрабатывают приложение, предназначенное, в первую очередь, для внутреннего рынка. Наиболее вероятно, что исходные тексты набираются в редакторе, в кодировке EUC-JP или Shift-JIS, что позволяет им вставлять японские иероглифы прямо в текст программы, примерно так:

```
QPushButton *button = new QPushButton(tr("日諾"), 0);
```

По-умолчанию, Qt интерпретирует аргументы функции **tr()** как Latin-1. Чтобы установить иную кодировку, нужно вызвать статическую функцию **QTextCodec::setCodecForTr()**, например:

```
QTextCodec *japaneseCodec = QTextCodec::codecForName("EUC-JP");
QTextCodec::setCodecForTr(japaneseCodec);
```

Это должно быть сделано перед самым первым вызовом функции **tr()**. Как правило это делается в функции **main()**, после создания объекта **QApplication**.
Но все остальные строки в программе, по прежнему будут интерпретироваться как Latin-1. Если программист хочет записать японские иероглифы в строковую переменную, он должен выполнить явное преобразование в Unicode:

```
QString text = japaneseCodec->toUnicode("海鮮料理");
```

Как альтернатива -- установить соответствующий кодек для выполнения преобразований между **const char *** и **QString**, вызовом **QTextCodec::setCodecForCStrings()**:

```
QTextCodec::setCodecForCStrings(japaneseCodec);
```

Техника, описанная выше, может применяться к любой кодировке, не являющейся Latin-1, включая Китайскую, Греческую, Корейскую и Русскую. Ниже приводится список кодировок, поддерживаемых библиотекой Qt 3.2:

- Apple Roman
- Big5-HKSCS
- CP874
- CP1250
- CP1251
- CP1252
- CP1253
- CP1254
- CP1258
- EUC-JP
- EUC-KR
- GB2312
- GB18030
- GBK
- IBM-850
- IBM-866
- ISO 8859-4
- ISO 8859-5
- ISO 8859-6
- ISO 8859-7
- ISO 8859-8
- ISO 8859-8-I
- ISO 8859-9
- ISO 8859-10
- ISO 8859-15
- ISO 10646 UCS-2
- JIS7
- KOI8-R
- KOI8-U
- Shift-JIS
- TIS-620
- TSCII

·	CP1255 ·	ISO 8859-1	ISO 8859-11 ·	UTF-8
·	CP1256 ·	ISO 8859-2	ISO 8859-13	
·	CP1257 ·	ISO 8859-3	ISO 8859-14	

Для каждой из них, **QTextCodec::codecForName()** возвращает правильное значение. Поддержка других кодировок может быть реализована либо путем создания производного класса от **QTextCodec**, либо созданием файла-карты (**charmap**) и последующим использованием **QTextCodec::loadCharmapFile()**.

15.2 Разработка приложений, подготовленных к переводу

Если необходимо предусмотреть возможность перевода приложения на разные языки, следует соблюдать следующие положения:

- Весь текст, который будет отображаться перед пользователем, должен пропускаться через функцию **tr()**.
- На запуске, приложение должно подгружать файл с переводом (**.qm**).

Эти условия не являются обязательными для приложений, которые никогда не будут переведены на другие языки. Однако, использование функции **tr()** не настолько обременительно, чтобы отказываться от нее. К тому же, тем самым вы оставляете открытой возможность локализации приложения в будущем.

Функция **tr()** -- статическая, она определена в классе **QObject** и перекрывается в каждом классе-потомке, который включает в свое определение макрос **Q_OBJECT**. Она возвращает перевод заданной строки, если он существует, или оригинальную версию строки -- в противном случае.

Для подготовки файла перевода необходимо запустить утилиту Qt -- **lupdate**. Она извлечет из исходного текста программы все строки, которые передаются функции **tr()** и создаст файл перевода. Этот файл может быть передан переводчику, который добавит в него перевод для каждой из строк. Более подробно процесс перевода описан в разделе Перевод существующих приложений.

Функция **tr()** имеет следующий синтаксис вызова:

```
  Context::tr(sourceText, comment)
```

Часть имени **Context** -- это имя класса, производного от **QObject**. Если функция вызывается в контексте класса, то указание имени класса не обязательно. **sourceText** -- это строка символов, которая должна быть переведена. comment -- не обязательный аргумент, может использоваться для предоставления дополнительной информации переводчику.

Еще один пример:

```
BlueWidget::BlueWidget(QWidget *parent, const char *name)
    : QWidget(parent, name)
{
  QString str1 = tr("Legal");
  QString str2 = BlueWidget::tr("Legal");
  QString str3 = YellowDialog::tr("Legal");
  QString str4 = YellowDialog::tr("Legal", "US paper size");
}
```

Первые два вызова производятся в контексте класса **BlueWidget**, последние два -- **YellowDialog**. Все четыре вызова получают строку "**Legal**" в качестве исходной, кроме того, последний из них имеет дополнительный комментарий, который поможет переводчику понять смысл исходной строки.

Перевод строк в разных контекстах выполняется независимо друг от друга. Переводчики обычно учитывают контекст в своей работе, часто выполняя пробные запуски приложения и оценивая качество перевода.

При вызове **tr()** из глобальных функций, необходимо явно указывать контекст, в качестве которого может использоваться любой класс, наследник от **QObject**. Если в приложении нет ничего подходящего, всегда можно прибегнуть к услугам самого **QObject**, например:

```
int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  ...
  QPushButton button(QObject::tr("Hello Qt!"), 0);
  app.setMainWidget(&button);
```

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

Очень часто полезной оказывается следующая методика, которая может быть применена к переводу названия приложения: вместо того, чтобы всякий раз набивать строки с именем приложения и вынуждать переводчика переводить их для каждого из контекстов, в котором они используются, более удобным будет определить его в виде макроса **APPNAME**, поместить макрос в заголовочный файл и использовать его по мере необходимости:

```
#define APPNAME MainWindow::tr("OpenDrawer 2D")
```

До сих пор, в качестве контекста мы рассматривали имя класса. Это довольно удобно, поскольку в большинстве случаев мы можем не указывать контекст перевода явно, при вызове функции **tr()**. Но более универсальный способ подготовки строк к переводу состоит в использовании функции **QApplication:: translate()**, которая принимает три аргумента: контекст, исходный текст и необязательный комментарий. Например, еще один способ определения макроса **APPNAME**:

```
#define APPNAME qApp->translate("Global Stuff", "OpenDrawer 2D")
```

На этот раз текст помещен в контекст "**Global Stuff**".
Функции **tr()** и **translate()** имеют двойное назначение: они служат маркерами для утилиты **lupdate** и в то же самое время -- это обычные функции C++, которые выполняют перевод текста. Такая двойственность накладывает некоторые ограничения на то, как записывается исходный код. Например, следующий отрывок не будет выполнять перевод строки на другой язык:

```
  // НЕВЕРНО
  const char *appName = "OpenDrawer 2D";
  QString translated = tr(appName);
```

Проблема состоит в том, что **lupdate** не сможет отыскать строку "OpenDrawer 2D", поскольку она явно не передается функции **tr()**. Эта проблема очень часто проявляется при работе с динамическими строками:

```
  // НЕВЕРНО
  statusBar()->message(tr("Host " + hostName + " found"));
```

Здесь строка изменяется динамически, в зависимости от значения переменной **hostName**, таким образом мы не можем требовать от **tr()** корректного перевода.
Как одно из решений проблемы -- используйте **QString::arg()**:

```
  statusBar()->message(tr("Host %1 found").arg(hostName));
```

Остановимся в этом месте чуть подробнее: функции **tr()** передается строка символов "Host %1 found". Допустим, что приложение загрузило файл с русским переводом, тогда функция **tr()** должна вернуть примерно такую строку: "Обнаружен узел сети %1". После этого аргумент '**%1**' замещается содержимым переменной **hostName**. В результате мы получаем вполне корректный перевод сообщения, которое демонстрируется русскоговорящему пользователю.
В случае, когда необходимо записать перевод строки в переменную, следует использовать макрос **QT_TR_NOOP()**. Чаще всего этот прием используется при создании статических массивов строк, например:

```
void OrderForm::init()
{
  static const char * const flowers[] = {
      QT_TR_NOOP("Medium Stem Pink Roses"),
      QT_TR_NOOP("One Dozen Boxed Roses"),
      QT_TR_NOOP("Calypso Orchid"),
      QT_TR_NOOP("Dried Red Rose Bouquet"),
      QT_TR_NOOP("Mixed Peonies Bouquet"),
      0
  };
```

```
  int i = 0;
  while (flowers[i]) {
    comboBox->insertItem(tr(flowers[i]));
    ++i;
  }
}
```

Макрос **QT_TR_NOOP()** фактически ничего не делает, но он служит маркером для **lupdate**. Строки, передаваемые этому макросу попадут в файл перевода и затем **tr()** переведет содержимое переменной обычным образом. Как видите, даже не смотря на то, что функции **tr()** передается не текст, а переменная, перевод будет выполнен корректно.
Есть еще один макрос -- **QT_TRANSLATE_NOOP()**, который похож на **QT_TR_NOOP()**, только в отличие от последнего, ему можно задать контекст перевода. Этот макрос найдет применение, когда необходимо инициализировать переменные за пределами 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
};
```

причем контекст должен совпадать с контекстом вызова функции **tr()**, которая будет выполнять перевод этих строк.
При использовании **tr()** в приложении не так уж и сложно забыть заключить какие нибудь строки в вызов этой функции, особенно если вы еще новичок. Эти досадные промахи будут проявляться в локализованных приложениях в виде непереведенных сообщений или надписей, вызывая чувство недовольства у пользователя. Чтобы избежать этой проблемы, мы можем запретить неявное преобразование из **const char *** в **QString**, определив символ препроцессора **QT_NO_CAST_ASCII**, перед директивой подключения заголовочного файла <**qstring.h**>. Самый простой способ определить этот символ -- поместить в файл .pro следующую строку:

```
DEFINES += QT_NO_CAST_ASCII
```

В результате, каждая строка, которая не пропускается через вызов **tr()** или **QString:: fromAscii()** (в зависимости от того, должна строка подвергаться переводу или нет), будет вызывать ошибку времени компиляции.
После того, как все строки будут "завернуты" в вызовы **tr()**, остается соблюсти еще одно важное условие -- загрузить на запуске файл с переводом. Обычно это делается в функции **main()**. Например, следующий код загрузит файл с переводом, с учетом региональных настроек пользователя:

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

  QTranslator appTranslator;
  appTranslator.load(QString("app_") + QTextCodec::locale(),
                     qApp->applicationDirPath());
  app.installTranslator(&appTranslator);
  ...
  return app.exec();
}
```

Функция **QTextCodec::locale()** возвращает строку -- имя локали пользователя, запустившего приложение. Локаль может быть определена более или менее точно, например, **ru** определяет русскую локаль, **ru_RU** -- русскую локаль для России, **ru_RU.KOI8-R** -- русскую локаль для России, с кодировкой символов KOI8-R.
Предположим, что приложение получило строку с именем локали -- **ru_RU.KOI8-R**, тогда **load()** попытается сначала загрузить файл **app_ru_RU.KOI8-R.qm**. Если этот файл отсутствует, то **load()** попытается загрузить файл **app_ru_RU.qm**, затем **app_ru.qm** и наконец **app.qm**. Обычно, в таких случаях достаточно создать один файл, с именем **app_ru.qm**. Однако, если перевод предполагает более точный

учет региональных настроек, как например в случае **fr_FR** (французский язык для Франции) и **fr_CA** (французский язык для Канады), то может потребоваться создать отдельные файлы с переводом для каждого из регионов.
Второй аргумент функции **load()** -- это каталог, где находится файл с переводом. Компания Trolltech предоставляет файлы с французским и немецким переводами Qt в каталоге translations. (Переводы на некоторые другие языки так же могут поставляться вместе с библиотекой, но все они выполняются командами добровольцев и официально не поддерживаются.) Так же должен подгружаться библиотечный файл с переводом:

```
QTranslator qtTranslator;
qtTranslator.load(QString("qt_") + QTextCodec::locale(),
                  qApp->applicationDirPath());
app.installTranslator(&qtTranslator);
```

Экземпляр класса **QTranslator** может хранить только один файл с переводом, поэтому следует использовать различные **QTranslator**. Но это не является большой проблемой, так как мы можем создать столько экземпляров класса **QTranslator**, сколько потребуется. Все они будут использоваться приложением при поиске перевода.
В некоторых языках, таких как арабский и иврит, строки пишутся справа-налево. В этих случаях приложению необходимо сообщить о порядке вывода строк вызовом **QApplication::setReverseLayout(true).** Для таких языков, файл перевода должен содержать специальный маркер -- "**LTR**", который обеспечивает корректный вывод переведенных строк.

Для пользователей программы может оказаться более удобным вариант, когда файлы перевода внедряются в тело исполняемого файла программы. Мало того, что этот прием уменьшает количество файлов, которые придется распространять вместе сприложением, но это так же сведет к минимуму риск случайной потери файлов с переводами. Для реализации этой возможности, в составе Qt распространяется утилита qembed, которая преобразует файлы с переводами в массивы C++, которые могут передаваться функции **QTranslator::load()**.

Мы описали все, что необходимо сделать, чтобы подготовить приложение к интернационализации. Но язык и направление письма это еще не все, что отличает страны и культуры. Интернационализированная программа должна принимать во внимание формат представления даты, времени, национальной валюты, чисел и порядок сортировки строк. Для этого в Qt 3.2 не существует никаких специальных функций, но мы можем использовать стандартные функции **setlocale()** и **localeconv()**. [8]

Некоторые функции и классы Qt адаптируют свое поведение под настройки локали:

- Функция **QString::localeAwareCompare()** выполняет сравнение строк в зависимости от настроек локали. Она используется классами **QIconView** и **QListView** для выполнения сортировки своих элементов.
- Функция **toString()** используется классами **QDate**, **QTime** и **QDateTime**, возвращающими локализованное представление даты и времени, когда вызываются с аргументом **Qt::LocalDate**.
- По-умолчанию **QDateEdit**, **QTimeEdit** и **QDateTimeEdit** представляют дату и время в локализованном виде.

Наконец, вместе с переводом, приложение может использовать разные наборы иконок для разных языков. Например, для языков, в которых письмо осуществляется справа-налево, в web-браузере логичнее было бы поменять местами кнопки "Назад" и "Вперед". Сделать это можно следующим образом:

```
if (QApplication::reverseLayout()) {
  backAct->setIconSet(forwardIcon);
  forwardAct->setIconSet(backIcon);
} else {
  backAct->setIconSet(backIcon);
  forwardAct->setIconSet(forwardIcon);
}
```

Иконки, изображение на которых соответствует алфавитным символам, очень часто должны быть адаптированы, в соответствии с конкретными языковыми настройками. Например, в текстовых

процессорах, иконка с изображением символа "**I**" (что означает "**Italic**" -- Курсив) должна быть заменена на "**C**" для Испании (**Cursivo**) или на "**К**" -- для России (**Курсив**). Самый простой способ:

```
if (tr("Italic")[0] == 'C') {
  italicAct->setIconSet(iconC);
} else if (tr("Italic")[0] == 'K') {
  italicAct->setIconSet(iconK);
} else {
  italicAct->setIconSet(iconI);
}
```

15.3 Динамическое переключение языков

В большинстве приложений, определение предпочитаемого пользователем языка производится в функции **main()** и затем выполняется загрузка соответствующего файла с переводом. Чаще всего этого бывает достаточно, но иногда возникают ситуации, когда нужно быстро переключить интерфейс приложения на другой язык, без перезапуска приложения. В качестве примера можно привести приложения для операторов международных телефонных центров, для переводчиков, выполняющих синхронный перевод или для операторов отделений банков, когда становится насущной необходимость в быстром переключении между языками.
Чтобы приложение позволяло выполнять быстрый переход от языка к языку, нужно сделать немного больше, чем просто загрузить единственный файл с переводом, но это не так сложно, как может показаться на первый взгляд. Чтобы реализовать возможность выбора языка во время исполнения приложения, нужно:

- Обеспечить пользователя средствами выбора языка.
- Для каждого виджета или диалога, все строки, подвергающиеся переводу, необходимо разместить в отдельной функции (которую часто называют как **retranslateStrings()**) и вызывать ее при смене языка.

Рассмотрим отдельные части исходного кода приложения, которое поддерживает возможность быстрой смены языка интерфейса. Язык по-умолчанию -- Английский.

Рисунок 15.1. Меню Language.

Поскольку заранее не известно, какой из языков предпочтет пользователь, приложение запускается с языком по-умолчанию. Загрузка файла с переводом будет производиться динамически, по мере необходимости. Таким образом, весь код, поддерживающий перевод, должен размещаться в классах главного окна и диалогов. Рассмотрим класс главного окна приложения-примера (**Call Center**), производного от класса **QMainWindow**:

```
MainWindow::MainWindow(QWidget *parent, const char *name)
    : QMainWindow(parent, name)
{
  journalView = new JournalView(this);
  setCentralWidget(journalView);

  qmPath = qApp->applicationDirPath() + "/translations";
  appTranslator = new QTranslator(this);
  qtTranslator = new QTranslator(this);
  qApp->installTranslator(appTranslator);
  qApp->installTranslator(qtTranslator);

  createActions();
```

```
  createMenus();
  retranslateStrings();
}
```

В конструкторе, виджет **JournalView** (наследник класса **QListView**) назначается центральным. Затем настраиваются переменные-члены, которые имеют отношение к переводу:

- Переменная **qmPath**, типа **QString**, определяет путь к каталогу, где находятся файлы с переводами.
- Переменная **appTranslator** -- это указатель на объект **QTranslator**, который используется для хранения текущего перевода.
- Переменная **qtTranslator** -- это указатель на объект **QTranslator**, который используется для хранения текущего библиотечного перевода.

В конце вызываются **createActions()** и **createMenus()**, которые создают систему меню. И наконец вызывается функция **retranslateStrings()**, которая настраивает первоначальный перевод по-умолчанию.

```
void MainWindow::createActions()
{
  newAct = new QAction(this);
  connect(newAct, SIGNAL(activated()), this, SLOT(newFile()));
  ...
  aboutQtAct = new QAction(this);
  connect(aboutQtAct, SIGNAL(activated()), qApp, SLOT(aboutQt()));
}
```

Функция **createActions()** создает обычные объекты **QAction**, правда без указания надписей и горячих комбинаций клавиш. Эти действия будут выполнены в функции **retranslateStrings()**.

```
void MainWindow::createMenus()
{
  fileMenu = new QPopupMenu(this);
  newAct->addTo(fileMenu);
  openAct->addTo(fileMenu);
  saveAct->addTo(fileMenu);
  exitAct->addTo(fileMenu);
  ...
  createLanguageMenu();
}
```

Функция **createMenus()** создает меню, но не вставляет их в полосу меню. Эти действия так же будут выполнены в функции **retranslateStrings()**. В конце функции вызывается **createLanguageMenu()**, которая заполняет меню списком поддерживаемых языков. Мы вернемся к этой функции чуть позже, а сейчас заглянем в исходный код функции **retranslateStrings()**:

```
void MainWindow::retranslateStrings()
{
  setCaption(tr("Call Center"));

  newAct->setMenuText(tr("&New"));
  newAct->setAccel(tr("Ctrl+N"));
  newAct->setStatusTip(tr("Create a new journal"));
  ...
  aboutQtAct->setMenuText(tr("About &Qt"));
  aboutQtAct->setStatusTip(tr("Show the Qt library's About box"));

  menuBar()->clear();
  menuBar()->insertItem(tr("&File"), fileMenu);
  menuBar()->insertItem(tr("&Edit"), editMenu);
  menuBar()->insertItem(tr("&Reports"), reportsMenu);
  menuBar()->insertItem(tr("&Language"), languageMenu);
  menuBar()->insertItem(tr("&Help"), helpMenu);
}
```

В функции **retranslateStrings()** сосредоточены все вызовы **tr()** для класса **MainWindow**. Она вызывается из конструктора класса, а так же всякий раз, когда пользователь изменяет язык интерфейса приложения, из меню **Language**.

Здесь в пункты меню записывается текст и строки подсказки. Затем меню вставляются в полосу меню, с уже переведенными надписями.
Функция **createMenus()**, которая упоминалась выше, вызывает функцию **createLanguageMenu()**, чтобы заполнить меню **Language** списком поддерживаемых языков:

```
void MainWindow::createLanguageMenu()
{
  QDir dir(qmPath);
  QStringList fileNames = dir.entryList("callcenter_*.qm");

  for (int i = 0; i < (int)fileNames.size(); ++i) {
    QTranslator translator;
    translator.load(fileNames[i], qmPath);

    QTranslatorMessage message =
            translator.findMessage("MainWindow", "English");
    QString language = message.translation();

    int id = languageMenu->insertItem(
                    tr("&%1 %2").arg(i + 1).arg(language),
                    this, SLOT(switchToLanguage(int)));
    languageMenu->setItemParameter(id, i);
    if (language == "English")
      languageMenu->setItemChecked(id, true);

    QString locale = fileNames[i];
    locale = locale.mid(locale.find('_') + 1);
    locale.truncate(locale.find('.'));
    locales.push_back(locale);
  }
}
```

Вместо того, чтобы жестко зашивать список языков в приложение, здесь создается один пункт меню для каждого файла с переводом (**.qm**). Для простоты примера, предполагается, что англоязычный вариант так же находится в отдельном файле **.qm**. Как альтернатива -- когда пользователь выбирает пункт меню **English**, очищать **QTranslator** методом **clear()**.

Единственная сложность тут состоит в том, чтобы представить названия языков в меню в достаточно удобочитаемом виде. Если просто показывать **en**, вместо **English**, или **de**, вместо **Deutsch**, то мы можем привести в замешательство отдельных пользователей. Поэтому, **createLanguageMenu()** проверяет перевод строки "**English**" в контексте "**MainWindow**". Эта строка должна быть переведена как "**Deutsch**" в немецком переводе, как "**Francais**" -- во французским и как "日本語" -- в японском. Пункты меню создаются вызовом **QPopupMenu::insertItem()**. Они соединяются со слотом **switchToLanguage(int)** главного окна, который мы рассмотрим чуть ниже. Аргумент слота **switchToLanguage(int)** -- это значение, установленное функцией **setItemParameter()**. Примерно то же самое мы делали в Главе 3, когда в приложении **Spreadsheet** создавали пункты меню, соответствующие названиям недавно открывавшихся документов.
В конце, название локали добавляется в список **locales**, который будет использоваться функцией **switchToLanguage()**.

```
void MainWindow::switchToLanguage(int param)
{
  appTranslator->load("callcenter_" + locales[param], qmPath);
  qtTranslator->load("qt_" + locales[param], qmPath);

  for (int i = 0; i < (int)languageMenu->count(); ++i)
    languageMenu->setItemChecked(languageMenu->idAt(i),
                                 i == param);
  retranslateStrings();
}
```

Слот **switchToLanguage()** отрабатывает, когда пользователь выбирает какой либо пункт в меню **Language**. Сначала слот загружат файлы с переводами для приложения и для библиотеки Qt. Затем обновляет состояние маркеров пунктов меню. И наконец вызывает **retranslateStrings()**, которая выполняет перевод всех надписей в главном окне.

В Microsoft Windows, дополнительно к обслуживанию меню **Language**, можно предусмотреть реакцию приложения на событие **LocaleChange**, которое возникает, когда изменяются региональные настройки среды. Этот тип событий существует во всех версиях Qt, но актуален только для Microsoft Windows. Для обработки этого события необходимо перекрыть обработчик **QObject::event()** следующим образом:

```
bool MainWindow::event(QEvent *event)
{
  if (event->type() == QEvent::LocaleChange) {
    appTranslator->load(QString("callcenter_")
                        + QTextCodec::locale(),
                        qmPath);
    qtTranslator->load(QString("qt_") + QTextCodec::locale(),
                       qmPath);
    retranslateStrings();
  }
  return QMainWindow::event(event);
}
```

Если пользователь изменит региональные настройки во время работы приложения, то будет произведена попытка загрузить соответствующий файл с переводом и последующим вызовом **retranslateStrings()** будет обновлен интерфейс приложения.

В любом случае, событие передается обработчику событий предка, поскольку базовый класс так же может быть заинтересован в получении события **LocaleChange**.

На этом мы завершаем обзор класса **MainWindow** и переходим к одному из виджетов главного окна -- **JournalView**, чтобы показать, какие изменения необходимо внести, чтобы он так же поддерживал возможность изменения языка во время работы приложения.

```
JournalView::JournalView(QWidget *parent, const char *name)
    : QListView(parent, name)
{
  ...
  retranslateStrings();
}
```

Класс **JournalView** порожден от класса **QListView**. В конце конструктора класса вызывается его метод **retranslateStrings()**. Это очень похоже на то, что мы делали в классе главного окна.

```
bool JournalView::event(QEvent *event)
{
  if (event->type() == QEvent::LanguageChange)
    retranslateStrings();
  return QListView::event(event);
}
```

Обработчик событий виджета вызывает **retranslateStrings()**, при поступлении события **LanguageChange**.

Qt генерирует событие **LanguageChange**, при изменении содержимого **QTranslator**. В нашем приложении эта ситуация возникает, когда вызывается функция **load()**, для загрузки файлов перевода в **appTranslator** и **qtTranslator**.

Не следует путать события **LanguageChange** и **LocaleChange**. Событие **LocaleChange** как бы говорит приложению: "Необходимо загрузить другой файл с переводом", а событие **LanguageChange**: "Необходимо выполнить перевод всех строк".
В классе **MainWindow** у нас не возникало необходимости реагировать на событие **LanguageChange**, поскольку функция **retranslateStrings()** итак вызывалась сразу же вслед за загрузкой нового файла перевода.

```
void JournalView::retranslateStrings()
{
```

```
  for (int i = columns() - 1; i >= 0; --i)
    removeColumn(i);
  addColumn(tr("Time"));
  addColumn(tr("Priority"));
  addColumn(tr("Phone Number"));
  addColumn(tr("Subject"));
}
```

Функция **retranslateStrings()** пересоздает заголовки столбцов в **QListView**, с новыми надписями в них. Для этого, сначала все заголовки удаляются, а потом создаются новые. Эта операция воздействует только на заголовки столбцов и никак не влияет на содержимое **QListView**.

Для диалогов и виджетов, создаваемых в визуальном построителе Qt Designer, утилита uic сама создает функции, похожие на **retranslateStrings()**, которая автоматически вызывается в ответ на событие **LanguageChange**. Все что нам остается сделать -- это загрузить соответствующий файл с переводом, когда пользователь изменяет язык приложения.

15.4 Перевод существующих приложений

Перевод Qt-приложений, которые содержат в себе вызовы **tr()**, выполняется в три приема;
1. Утилитой *lupdate* извлекаются все строки из исходного текста приложения.
2. Выполняется перевод строк, с помощью утилиты Qt *Linguist*.
3. С помощью утилиты *lrelease* создается двоичный файл .qm с переводом, который потом может быть загружен приложением.

Пункты 1 и 3 выполняются разработчиком приложения. Пункт 2 -- переводчиком. Этот процесс может повторяться неоднократно, в ходе разработки и эксплуатации приложения.

В качестве примера, рассмотрим процесс перевода приложения **Spreadsheet**, которое было написано нами в Главе 3. Оно уже содержит все необходимые вызовы **tr()**.

Прежде всего, необходимо внести изменения в файл проекта .pro, чтобы указать -- какие языки будут поддерживаться приложением. Допустим, что мы собираемся включить поддержку немецкого, французского и русского языков, дополнительно к английскому, тогда необходимо в файл **spreadsheet.pro** добавить раздел **TRANSLATIONS**:

```
TRANSLATIONS = spreadsheet_de.ts \
               spreadsheet_fr.ts \
               spreadsheet_ru.ts
```

Здесь мы указали три файла переводов: для немецкого, французского и русского языков. Эти файлы будут созданы при первом запуске утилиты **lupdate**, а на последующих запусках будут просто дополняться.
Обычно исходные файлы с переводом имеют расширение **.ts**. Они записываются в формате XML и потому занимают больше места на диске, чем скомпилированные файлы с переводом .qm. Для тех, кому это интересно -- .ts означает "translation source" (исходный текст перевода), а **.qm** -- "Qt message".
Допустим, что мы уже находимся в каталоге с исходными текстами приложения **Spreadsheet**. Теперь запускаем **lupdate** из командной строки:

```
  lupdate -verbose spreadsheet.pro
```

Ключ **-verbose** -- необязательный. Он просто заставляет **lupdate** выводить более подробную информацию в ходе своей работы. Ниже приведен примерный вывод, полученный во время работы утилиты:

```
Updating  spreadsheet_de.ts ...
  0 known, 101 new and 0 obsoleted messages
Updating  spreadsheet_fr.ts ...
  0 known, 101 new and 0 obsoleted messages
Updating  spreadsheet_ru.ts ...
  0 known, 101 new and 0 obsoleted messages
```

Каждая строка, которая "завернута" в вызов **tr()**, заносится в **.ts**, с пустым местом для перевода. Строки, которые находятся в файле **.ui**, так же включаются в исходный файл перевода.

По-умолчанию, **lupdate** предполагает, что все строки, завернутые в вызовы **tr()**, набраны в кодировке Latin-1. Если это не так, необходимо указать элемент **CODEC** в файле **.pro**, например так:

```
CODEC       = EUC-JP
```

Это необходимо делать в дополнение к вызову **QTextCodec::setCodecForTr()** в приложении.
Перевод, в файлы **spreadsheet_de.ts**, **spreadsheet_fr.ts** и **spreadsheet_ru.ts**, добавляется переводчиком, с помощью утилиты Qt **Linguist**.
Чтобы запустить Qt **Linguist**, в среде **Windows**, выберите пункт **Qt 3.2.x | Qt Linguist** в меню **Пуск**, в среде Unix -- наберите команду **linguist**. Затем, с помощью меню **File|Open**, откройте файл с исходным текстом перевода.

С левой стороны главного окна Qt Linguist находится список контекстов переводов. Для **Spreadsheet** существуют следующие контексты: "**FindDialog**", "**GoToCellDialog**", "**MainWindow**", "**SortDialog**" и "**Spreadsheet**". В верхней части с правой стороны находится список строк для текущего контекста. Каждая строка отображается вместе с переводом и флагом **Done** ("Готово"). В средней области, с правой стороны, вводится текст перевода для текущей строки. И внизу находится список переводов, автоматически предлагаемых утилитой Qt Linguist.
По окончании работы над переводом, файл .ts необходимо преобразовать в файл **.qm**. Для этого, в приложении Qt Linguist выберите пункт меню **File|Release**. Обычно, после перевода нескольких строк, выполняются пробные запуски приложения, с созданным файлом .qm, чтобы визуально оценить качество перевода.

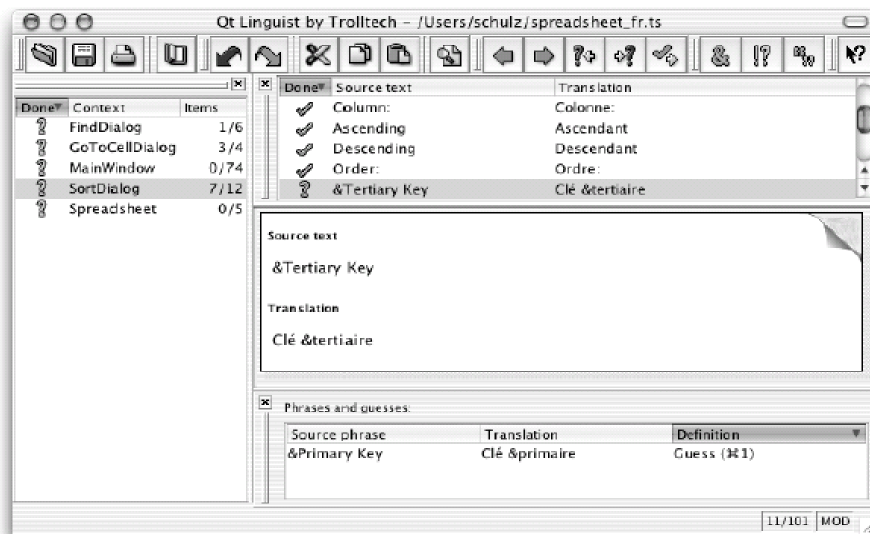


Рисунок 15.2. Qt Linguist в действии.

Чтобы перегенерировать все файлы **.qm** сразу, необходимо запустить утилиту командной строки **lrelease**:

```
lrelease -verbose spreadsheet.pro
```

Предположим, что мы сделали перевод на русский язык 19-ти строк, причем установили признак **Done** для 17-ти из них. В этом случае мы получим от **lrelease** примерно такой вывод:

```
Updating  spreadsheet_de.qm ...
  0 finished, 0 unfinished and 101 untranslated messages
Updating  spreadsheet_fr.qm ...
  0 finished, 0 unfinished and 101 untranslated messages
Updating  spreadsheet_ru.qm ...
  17 finished, 2 unfinished and 82 untranslated messages
```

Непереведенные строки, при пробном запуске приложения, будут отображаться на языке оригинала. Флаг Done никак не используется утилитой **lrelease**, он предназначен исключительно для переводчика, чтобы напоминать о том, какие строки имеют окончательный перевод, а какие требуют уточнения.

В случае внесения изменений в исходный код приложения, содержимое файлов **.ts** может "устареть". Чтобы этого не происходило нужно всякий раз запускать утилиту **lupdate**, добавлять перевод для вновь появляющихся строк и пересобирать файлы **.qm**. Некоторые команды разработчиков синхронизируют перевод так часто, насколько это только возможно, другие предпочитают дождаться окончательного релиза приложения и только тогда приступают к переводу.

Утилиты lupdate и Qt Linguist достаточно "умны". Переведенные строки, необходимость в которых уже отпала, все равно сохраняются в исходных файлах с переводами, на тот случай, если они могут понадобиться в будущем. При обновлении файлов **.ts**, утилита **lupdate** использует интеллектуальный алгоритм объединения, который помогает избежать лишней работы по переводу одинаковых строк в различных контекстах.

За дополнительной информацией о программах Qt Linguist, **lupdate** и **lrelease**, обращайтесь к справочному руководству Qt Linguist, которое доступно по адресу: *http://doc.trolltech.com/3.2/linguist-manual.html*. Это руководство содержит полное описание пользовательского интерфейса программы и пошаговый самоучитель для программистов.

16 Разработка справочной системы приложения

Большинство приложений включают в себя справочную систему. В некоторых случаях она ограничивается предоставлением всплывающих подсказок, подсказок, выводимых в строке состояния справки типа: "What's This?" ("Что это?"). Qt полностью поддерживает отображение справочной информации такого рода. В других приложениях объем справочной информации может быть очень велик и содержать огромное количество текста. Для отображения такого рода справочной информации можно использовать компонент **QTextBrowser**, а можно воспользоваться услугами Qt Assistant или любого другого обозревателя HTML-страниц.

16.1 Всплывающие подсказки и справка "What's This?"

Всплывающие подсказки -- это небольшие текстовые сообщения, которые появляются на экране, когда указатель мыши останавливается над виджетом на некоторое время. Текст таких подсказок выводится черным шрифтом на желтом фоне. Изначально всплывающие подсказки предназначались для того, чтобы дать краткое описание кнопок на панели инструментов. Однако мы можем добавить всплывающие подсказки к любому из виджетов. Например:

```
QToolTip::add(findButton, tr("Find next"));
```

Чтобы добавить подсказку к кнопке в панели инструментов, которой соответствует объект **QAction**, можно просто вызвать метод **setToolTip()** требуемого объекта **QAction**:

```
newAct = new QAction(tr("&New"), tr("Ctrl+N"), this);
newAct->setToolTip(tr("New file"));
```

Если текст подсказки не установить явно, он будет сгенерирован автоматически, из текста, заданного при создании объекта, и горячей комбинации клавиш, например "**New (Ctrl+N)**".
Подсказка в строке состояния -- это небольшой по объему текст, обычно немного больше, чем текст всплывающей подсказки, который отображается в строке состояния приложения. Для добавления такого текста нужно вызвать функцию **setStatusTip()**:

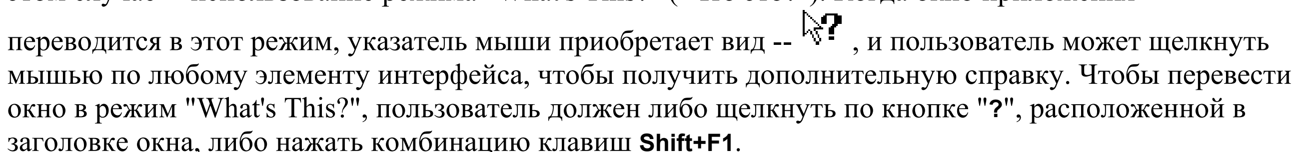
```
newAct->setStatusTip(tr("Create a new file"));
```

При отсутствии текста подсказки для строки состояния, **QAction** будет использовать текст всплывающей подсказки.

Для других виджетов необходимо передать в функцию **QToolTip::add()**, третьим и четвертым аргументами, объект класса **QToolTipGroup** и текст подсказки:

```
QToolTip::add(findButton, tr("Find next"), toolTipGroup,
              tr("Find the next occurrence of the search text"));
```

В визуальном построителе Qt Designer текст подсказок можно задать через свойства виджета **toolTip** и **statusTip**.

В некоторых случаях возникает необходимость в предоставлении большего количества справочной информации о виджете, чем это может уместиться в подсказки. Например, у нас может появиться желание добавить более подробный текст описания для каждого из полей в сложном диалоге, не заставляя при этом пользователя вызывать отдельное окно с текстом справки. Идеальное решение в этом случае -- использование режима "What's This?" ("Что это?"). Когда окно приложения переводится в этот режим, указатель мыши приобретает вид --  , и пользователь может щелкнуть мышью по любому элементу интерфейса, чтобы получить дополнительную справку. Чтобы перевести окно в режим "What's This?", пользователь должен либо щелкнуть по кнопке "**?**", расположенной в заголовке окна, либо нажать комбинацию клавиш **Shift+F1**.

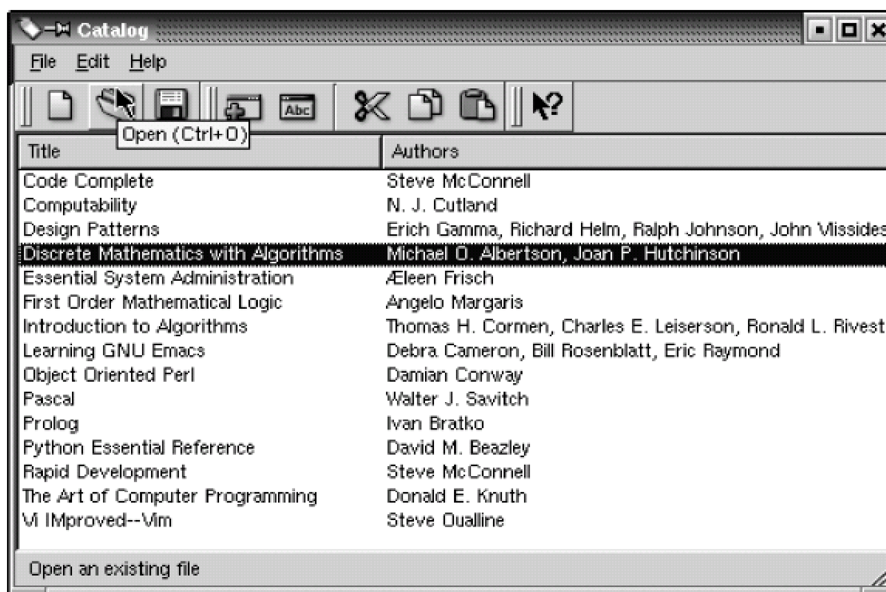


Рисунок 16.1. Приложение, отображающее всплывающую подсказку и подсказку в строке состояния.

Текст справки может быть зада вызовом **QWhatsThis::add()**. Например:

```
QWhatsThis::add(sourceLineEdit,
    tr("<img src=\"icon.png\">"
    " The meaning of the Source field depends on the "
    "Type field:"
    "<ul>"
    "<li><b>Books</b> have a Publisher</li>"
    "<li><b>Articles</b> have a Journal name with volume and "
    "issue number</li>"
    "<li><b>Thesis</b> have an Institution name and a "
    "department name</li>"
    "</ul>"));
```

Здесь, как и во многих других виджетах Qt, мы можем форматировать текст подсказки с помощью HTML-тегов. В этом примере мы добавили рисунок в текст подсказки (который описан в разделе **IMAGE**, в файле **.pro**), список сделали маркированным, а некоторые слова выделили жирным шрифтом. Теги HTML, которые поддерживает Qt, перечислены в описании класса **QStyleSheet**.

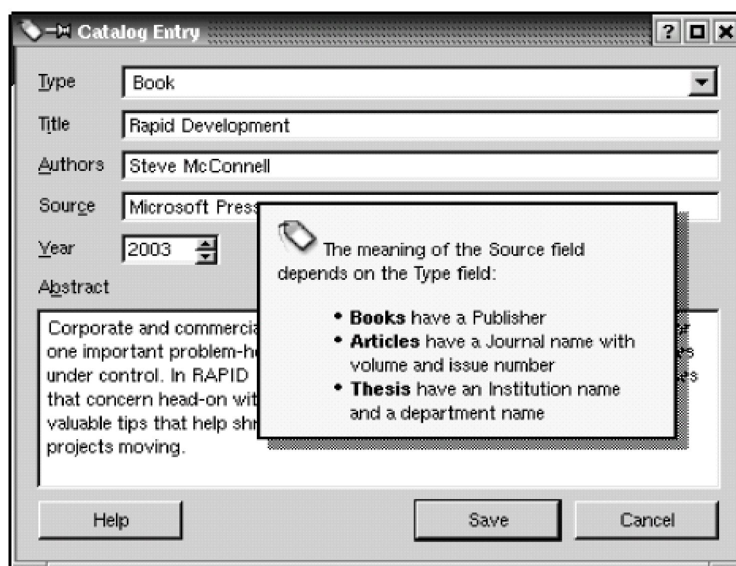


Рисунок 16.2. Диалог, отображающий текст подсказки "What's This?".

Текст подсказки "What's This?" может быть задан и для объектов **QAction**:

```
openAct->setWhatsThis(tr("<img src=open.png> "
                         "Click this option to open an "
                         "existing file."));
```

Текст подсказки будет отображаться после щелчка мышью по пунктам меню или кнопкам на панели инструментов, когда окно будет находиться в режиме "What's This?". В визуальном построителе Qt Designer текст справки можно задать через свойство **whatsThis**.

Если приложение предусматривает переход в режим "What's This?", то общепринятым считается добавление в меню **Help** пункта **What's This?** и кнопки **What's This?** на панель инструментов. Это может быть сделано за счет создания объекта **QAction**, у которого сигнал **activated()** будет связан со слотом **whatsThis()** главного окна приложения.

16.2 Использование QTextBrowser для отображения текста справки

Большие серьезные приложения могут содержать объем справочной информации, намного превышающий возможности всплывающих подсказок и даже подсказок типа "What's This?". Самое простое решение в этом случае -- создать обозреватель справочной системы. Приложение может открывать окно обозревателя справки при выборе пункта **Help**, в меню **Help**, или при нажатии на кнопку **Help** в панели инструментов.

В этом разделе мы рассмотрим простейший обозреватель справочной системы, внешний вид которого представлен на рисунке 16.3, и опишем, как его использовать в приложении. Для отображения текста справки используется **QTextBrowser**, который может обрабатывать некоторые теги HTML и идеально подходит под заданные условия.

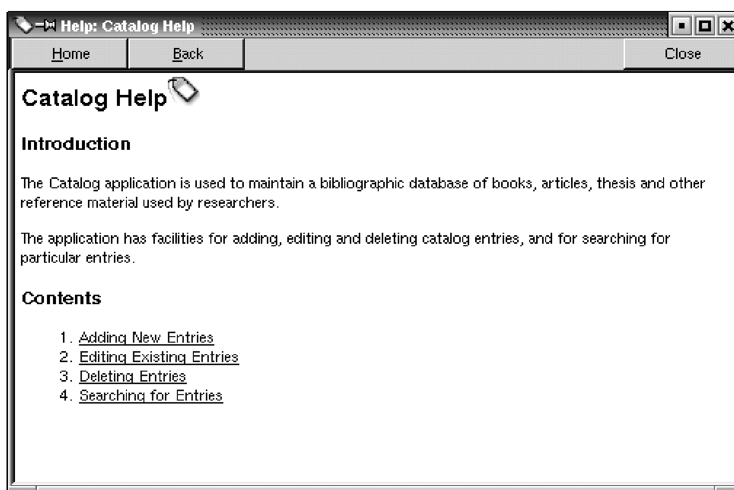


Рисунок 16.3. Виджет HelpBrowser.

Как обычно, начнем с заголовочного файла:

```
#include <qwidget.h>

class QPushButton;
class QTextBrowser;

class HelpBrowser : public QWidget
{
  Q_OBJECT
public:
  HelpBrowser(const QString &path, const QString &page,
              QWidget *parent = 0, const char *name = 0);

  static void showPage(const QString &page);

private slots:
  void updateCaption();
```

```
private:
  QTextBrowser *textBrowser;
  QPushButton *homeButton;
  QPushButton *backButton;
  QPushButton *closeButton;
};
```

HelpBrowser имеет статическую функцию, которая может вызываться из любого места в приложении. Она создает окно обозревателя и показывает запрошенную страницу.

Теперь исходный код файла реализации:

```
#include <qapplication.h>
#include <qlayout.h>
#include <qpushbutton.h>
#include <qtextbrowser.h>

#include "helpbrowser.h"

HelpBrowser::HelpBrowser(const QString &path, const QString &page,
                         QWidget *parent, const char *name)
    : QWidget(parent, name, WGroupLeader | WDestructiveClose)
{
  textBrowser = new QTextBrowser(this);
  homeButton = new QPushButton(tr("&Home"), this);
  backButton = new QPushButton(tr("&Back"), this);
  closeButton = new QPushButton(tr("Close"), this);
  closeButton->setAccel(tr("Esc"));

  QVBoxLayout *mainLayout = new QVBoxLayout(this);
  QHBoxLayout *buttonLayout = new QHBoxLayout(mainLayout);
  buttonLayout->addWidget(homeButton);
  buttonLayout->addWidget(backButton);
  buttonLayout->addStretch(1);
  buttonLayout->addWidget(closeButton);
  mainLayout->addWidget(textBrowser);

  connect(homeButton, SIGNAL(clicked()),
          textBrowser, SLOT(home()));
  connect(backButton, SIGNAL(clicked()),
          textBrowser, SLOT(backward()));
  connect(closeButton, SIGNAL(clicked()),
          this, SLOT(close()));
  connect(textBrowser, SIGNAL(sourceChanged(const QString &)),
          this, SLOT(updateCaption()));

  textBrowser->mimeSourceFactory()->addFilePath(path);
  textBrowser->setSource(page);
}
```

Размещение компонентов в окне более чем простое: ряд кнопок находится над **QTextBrowser**. Аргумент **path** -- это путь к каталогу, где находятся файлы с текстом справки. Аргумент page -- имя файла справки, с необязательным названием темы (в терминах HTML -- **anchor**, или имя ссылки).

Мы передаем конструктору флаг **WGroupLeader**, потому что окно **HelpBrowser** может открываться из модальных диалогов. Обычно модальные диалоги не позволяют пользователю взаимодействовать с другими окнами приложения. Однако, в данном случае, после того как пользователь запросил помощь, он должен иметь возможность работать как с окном модального диалога, так и с окном обозревателя справочной системы. Флаг **WGroupLeader** обеспечивает такую возможность.

```
void HelpBrowser::updateCaption()
{
  setCaption(tr("Help: %1").arg(textBrowser->documentTitle()));
}
```

Всякий раз, при переходе на другую страницу, вызывается слот **updateCaption()**. Функция **documentTitle()** возвращает текст, заданный в теге <**title**>.

```
void HelpBrowser::showPage(const QString &page)
{
  QString path = qApp->applicationDirPath() + "/doc";
  HelpBrowser *browser = new HelpBrowser(path, page);
  browser->resize(500, 400);
  browser->show();
}
```

В функции **showPage()** создается окно обозревателя и затем выводится на экран. Окно будет уничтожено автоматически, когда пользователь закроет его, поскольку в конструкторе был установлен флаг **WDestructiveClose**.

В данном примере мы исходим из предположения, что файлы справки находятся в подкаталоге **doc**. Теперь можно вызвать обозреватель из приложения. Для этого, в главном окне приложения, мы создадим объект **QAction -- Help** и соединим его со слотом **help()**:

```
void MainWindow::help()
{
  HelpBrowser::showPage("index.html");
}
```

Мы полагаем, что основной файл справки называется **index.html**. Чтобы вызвать обозреватель из диалога, нужно связать соответствующую кнопку **Help** со слотом **help()**:

```
void EntryDialog::help()
{
  HelpBrowser::showPage("dialogs.html#entrydialog");
}
```

Здесь мы обращаемся уже к другому файлу справки **-- dialogs.html** и выполняем переход к ссылке **entrydialog**.

Еще одно место, откуда можно вызвать обозреватель справочной системы -- текст справки типа "What's This?". Для этого достаточно вставить в текст справки "What's This?" тег HTML <**a href="..."**>.

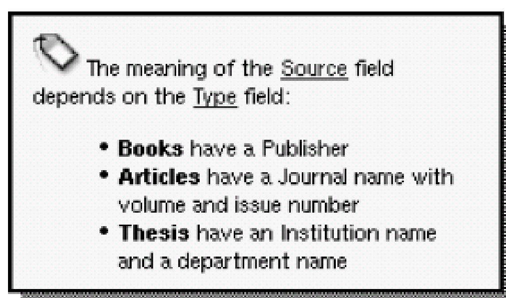


Рисунок 16.4. Текст справки "What's This?" со ссылкой на файл справки.

Чтобы такая гипертекстовая ссылка работала, мы должны использовать класс, производный от **QWhatsThis**, который будет "знать", как вызвать обозреватель справочной системы. Для этого надо в классе-потомке перекрыть метод **clicked()**, в котором вызвать **HelpBrowser:: showPage()**. Ниже приводится определение класса:

```
class MyWhatsThis : public QWhatsThis
{
public:
  MyWhatsThis(QWidget *widget, const QString &text);

  QString text(const QPoint &point);
  bool clicked(const QString &page);

private:
  QString myText;
};
```

Где **text()** и **clicked()** -- это методы предка, перекрываемые потомком.

```
MyWhatsThis::MyWhatsThis(QWidget *widget, const QString &text)
    : QWhatsThis(widget)
{
  myText = text;
}
```

Конструктор получает указатель на виджет и текст справки для этого виджета. Мы передаем указатель на виджет базовому конструктору и сохраняем текст справки в приватной переменной.

```
QString MyWhatsThis::text(const QPoint &)
{
  return myText;
}
```

Функция **text()** возвращает текст справки виджета, для заданных координат указателя мыши. Некоторые виджеты могут возвращать разный текст справки, в зависимости от того, где был произведен щелчок мышью, но в данном примере мы всегда будем возвращать один и тот же текст.

```
bool MyWhatsThis::clicked(const QString &page)
{
  if (page.isEmpty()) {
    return true;
  } else {
    HelpBrowser::showPage(page);
    return false;
  }
}
```

Функция **clicked()** вызывается в момент щелчка мышью по виджету, когда окно находится в режиме "What's This?". Если пользователь щелкает по гиперссылке, то функция получит название страницы в аргументе **page**. В противном случае в **page** будет пустая строка.

Возвращаемое значение используется базовым классом **QWhatsThis** для того, чтобы определить, что делать дальше -- скрыть (**true**) подсказку "What's This?" или оставить ее видимой (**false**). В данной ситуации мы хотим, чтобы текст "What's This?" оставался видимым на экране, поэтому возвращаем **false**. Когда пользователь щелкает по любому другому месту в тексте "What's This?", мы возвращаем **true**.
Ниже показан пример использования класса **MyWhatsThis**:

```
  new MyWhatsThis(sourceLineEdit,
      tr("<img src=\"icon.png\">"
      " The meaning of the "
      "<a href=\"fields.html#source\">Source</a> field depends on "
      "the <a href=\"fields.html#type\">Type</a> field:"
      "<ul>"
      "<li><b>Books</b> have a Publisher</li>"
      "<li><b>Articles</b> have a Journal name with volume and "
      "issue number</li>"
      "<li><b>Thesis</b> have an Institution name and a department "
      "name</li>"
      "</ul>"));
```

На этот раз, вместо вызова **QWhatsThis::add()**, мы создаем экземпляр класса **MyWhatsThis** для виджета, с текстом подсказки. Теперь пользователь может щелкнуть по гипертекстовой ссылке и вызвать обозреватель справочной системы.

Выглядит немного странно, так как мы создаем объект, но не связываем его с какой бы то ни было переменной. Но это только на первый взгляд, потому что Qt сама следит за создаваемыми объектами класса **QWhatsThis** и удаляет их, когда необходимость в них отпадает.

16.3 Использование Qt Assistant для отображения текста справки

Qt Assistant -- это приложение, которое вы можете свободно распространять совместно со своей программой, предоставляемое компанией **Trolltech**. Основные его достоинства -- поддержка индексации, поиск по все справочной системе и возможность поддержки нескольких наборов документации для различных приложений.
Для использования Qt Assistant в своих приложениях, необходимо добавить некоторый код, который добавит свою справочную систему в базу Qt Assistant.
Взаимодействие между приложением и Qt Assistant обеспечивает класс **QAssistantClient**, размещенный в отдельной библиотеке. Чтобы связать приложение с этой библиотекой, надо добавить следующую строку в файл проекта **.pro**:

```
LIBS += -lqassistantclient
```

Создадим новый класс **HelpBrowser**, который будет использовать Qt Assistant для отображения текста справки. Определение класса:

```
#ifndef HELPBROWSER_H
#define HELPBROWSER_H

class QAssistantClient;

class HelpBrowser
{
public:
  static void showPage(const QString &page);

private:
  static QAssistantClient *assistant;
};

#endif
```

И содержимое файла **helpbrowser.cpp**:

```
#include <qassistantclient.h>

#include "helpbrowser.h"

QAssistantClient *HelpBrowser::assistant = 0;

void HelpBrowser::showPage(const QString &page)
{
  if (!assistant)
    assistant = new QAssistantClient("");
  assistant->showPage(page);
}
```

Конструктор **QAssistantClient** получает в первом аргументе имя каталога, который используется для поиска исполняемого файла утилиты Qt Assistant. Когда передается пустая строка, то поиск производится в каталогах, указанных в переменной окружения **PATH**. Класс **QAssistantClient** имеет свою собственную функцию **showPage()**, которая принимает имя HTML-файла, с необязательным именем ссылки -- точь-в-точь как было сделано нами при создании класса **HelpBrowser**, производного от класса **QTextBrowser**.
Следующий шаг, который необходимо сделать -- сообщить Qt Assistant, где размещается документация к программе. Делается это путем создания профайла и файла **.dcf**, который содержит все необходимые сведения о документации. Как это делается описано в сопроводительной документации к утилите Qt Assistant, поэтому здесь мы не будем дублировать эти сведения.

17 Многопоточность

Обычно, приложения с графическим интерфейсом исполняются в рамках одного потока. Если такое приложение начинает выполнять продолжительную по времени операцию, то возникает эффект "замораживания" интерфейса, который длится до тех пор, пока длительная операция не будет завершена. В Главе 7 был приведен один из вариантов решения этой проблемы. Другой вариант -- многопоточность.

В многопоточных приложениях, обслуживание интерфейса производится в отдельном потоке, а обработка данных -- в другом (одном или нескольких) потоке. В результате приложение сохраняет возможность откликаться на действия пользователя даже во время интенсивной обработки данных. Еще одно преимущество многопоточности -- на многопроцессорных системах различные потоки могут выполняться на различных процессорах одновременно, что несомненно увеличивает скорость исполнения.

В этой главе мы поговорим о классе **QThread** и покажем, как синхронизировать потоки с помощью классов **QMutex**, **QSemaphore** и **QWaitCondition**. Затем коснемся темы взаимодействия между потоками и завершим главу перечислением классов Qt, которые могут использоваться за пределами главного потока приложения, где исполняется цикл обработки событий Qt.

Многопоточность -- очень объемная тема. Ей посвящено огромное количество книг. Здесь мы будем исходить из предположения, что вы уже знакомы с основными принципами разработки многопоточных приложений, и все наше внимание сконцентрируем не на использовании многопоточности как таковой, а на основных положениях создания многопоточных Qt-приложений.

17.1 Потоки

Добавить несколько потоков в приложение, написанное с использованием библиотеки Qt, очень просто. Для этого нужно лишь создать дочерний класс от **QThread** и перекрыть метод **run()**. В качестве примера мы напишем простой класс, потомок класса **QThread**, который выводит текст на консоль.

```
class Thread : public QThread
{
public:
  Thread();

  void setMessage(const QString &message);
  void run();
  void stop();

private:
  QString messageStr;
  volatile bool stopped;
};
```

Этот класс перекрывает метод родителя **run()** и добавляет еще две функции: **setMessage()** и **stop()**. Ключевое слово **volatile**, которое присутствует в объявлении переменной **stopped**, означает, что доступ к ней может производиться из других потоков и поэтому необходимо выполнять чтение значения переменной всякий раз, когда это необходимо. Если опустить этот спецификатор, компилятор может выполнить оптимизацию кода обращения к переменной, что в некоторых ситуациях приведет к неправильным результатам.

```
Thread::Thread()
{
  stopped = false;
}
```

В конструкторе, переменной **stopped** присваивается значение **false**.

```
void Thread::run()
{
  while (!stopped)
```

```
    cerr << messageStr.ascii();
  stopped = false;
  cerr << endl;
}
```

Функция **run()** предназначена для запуска потока. Поток будет исполняться до тех пор, пока в переменную **stopped** не будет записано значение **true**. В процессе исполнения потока, на консоль будет выводиться заданный текст сообщения. Поток завершит свою работу, как только функция **run()** вернет управление.

```
void Thread::stop()
{
  stopped = true;
}
```

Функция **stop()** записывает в переменную **stopped** значение **true** и тем самым останавливает исполнение потока. Эта функция может быть вызвана из другого потока в любой момент времени. В данном примере мы исходим из предположения, что присваивание значения булевой переменной является атомарной (в смысле -- непрерываемой) операцией. Чуть ниже, в этом же разделе, мы покажем, как обеспечить атомарность операции присваивания, с помощью экземпляра класса **QMutex**. Класс **QThread** имеет метод **terminate()**, который завершает работу потока. Однако мы не рекомендуем использовать его, поскольку этот метод может остановить поток в любой точке его исполнения, не давая ему возможность самому корректно завершить свою работу. Более безопасный способ остановки потока -- с помощью функции **stop()**, как это делается в нашем примере.

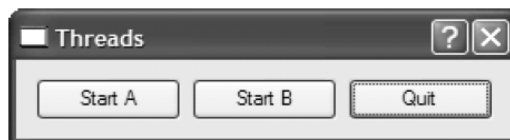


Рисунок 17.1. Внешний вид приложения Threads.

Теперь напишем небольшое приложение, которое будет запускать два дополнительных потока -- **A** и **B**.

```
class ThreadForm : public QDialog
{
  Q_OBJECT
public:
  ThreadForm(QWidget *parent = 0, const char *name = 0);

protected:
  void closeEvent(QCloseEvent *event);

private slots:
  void startOrStopThreadA();
  void startOrStopThreadB();

private:
  Thread threadA;
  Thread threadB;
  QPushButton *threadAButton;
  QPushButton *threadBButton;
  QPushButton *quitButton;
};
```

В классе **ThreadForm** объявлены две приватных переменных типа **Thread** и несколько кнопок, которыми выполняется управление приложением.

```
ThreadForm::ThreadForm(QWidget *parent, const char *name)
    : QDialog(parent, name)
{
  setCaption(tr("Threads"));

  threadA.setMessage("A");
```

```
  threadB.setMessage("B");

  threadAButton = new QPushButton(tr("Start A"), this);
  threadBButton = new QPushButton(tr("Start B"), this);
  quitButton = new QPushButton(tr("Quit"), this);
  quitButton->setDefault(true);

  connect(threadAButton, SIGNAL(clicked()),
          this, SLOT(startOrStopThreadA()));
  connect(threadBButton, SIGNAL(clicked()),
          this, SLOT(startOrStopThreadB()));
  connect(quitButton, SIGNAL(clicked()),
          this, SLOT(close()));
  ...
}
```

В конструкторе формы, с помощью вызовов функций **setMessage()**, потокам назначается текст для вывода на консоль. Таким образом, поток **A** будет печатать букву "**A**", а поток **B** -- букву "**B**".

```
void ThreadForm::startOrStopThreadA()
{
  if (threadA.running()) {
    threadA.stop();
    threadAButton->setText(tr("Start A"));
  } else {
    threadA.start();
    threadAButton->setText(tr("Stop A"));
  }
}
```

Когда пользователь нажимает кнопку, которая управляет потоком **A**, вызывается функция-слот **startOrStopThreadA()**. Она останавливает работу потока, если он запущен, и запускает -- в противном случае. Кроме того, функция так же изменяет надпись на кнопке.

```
void ThreadForm::startOrStopThreadB()
{
  if (threadB.running()) {
    threadB.stop();
    threadBButton->setText(tr("Start B"));
  } else {
    threadB.start();
    threadBButton->setText(tr("Stop B"));
  }
}
```

Код функции **startOrStopThreadB()** практически один-в-один повторяет код функции **startOrStopThreadA()**.

```
void ThreadForm::closeEvent(QCloseEvent *event)
{
  threadA.stop();
  threadB.stop();
  threadA.wait();
  threadB.wait();
  event->accept();
}
```

Если пользователь закрывает окно приложения, то потокам посылается команда на останов, после чего программа ждет (с помощью функции **QThread::wait()**), пока они не завершат свою работу. Затем вызывается **QCloseEvent:: accept()**.
Для сборки приложения, необходимо добавить следующую строку в файл **.pro**:

```
CONFIG += thread
```

Она сообщает **qmake** о том, что для сборки приложения должна использоваться версия Qt, совместимая с потоками. Чтобы собрать потоко-совместимую версию библиотеки Qt, необходимо передать ключ **-thread** скрипту **configure**, во время установки Qt в Unix или Mac OS X. В Windows,

библиотека Qt является потоко-совместимой по-умолчанию. Для данного примера необходимо добавить еще и опцию **console**, поскольку необходимо обеспечить вывод текста на консоль в операционной системе Windows:

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

Теперь, если запустить приложение и нажать на кнопку **Start A**, на консоли будет печататься последовательность символов "**A**". После нажатия на кнопку **Start B** последовательность символов "**A**" будет перемежаться символами "**B**". После нажатия на кнопку **Stop A**, будут выводиться одни символы "**B**".

Обычно в многопоточных приложениях возникает проблема синхронизации потоков. Для этих целей в Qt имеются классы **QMutex**, **QMutexLocker**, **QSemaphore** и **QWaitCondition**.

Класс **QMutex** являет собой средство защиты. С его помощью можно исключить возможность доступа к переменным или участкам кода из нескольких потоков одновременно. Класс имеет функцию **lock()**, которая "запирает" мьютекс. Если мьютекс не заперт, то текущий поток захватывает его и тут же "запирает", в противном случае, поток, который попытался захватить запертый мьютекс, блокируется до тех пор пока мьютекс не освободится. Когда функция **lock()** возвратит управление в поток, он станет держателем мьютекса до того момента, пока не будет вызвана функция **unlock()**. Кроме функции **lock()**, класс **QMutex** имеет функцию **tryLock()**, которая возвращает управление потоку немедленно, даже если мьютекс уже "заперт" другим потоком.
Для примере предположим, что необходимо защитить мьютексом переменную **stopped**. Для этого нужно добавить новый член класса **Thread**:

```
  QMutex mutex;
```

Функция **run()** теперь будет выглядеть так:

```
void Thread::run()
{
  for (;;) {
    mutex.lock();
    if (stopped) {
      stopped = false;
      mutex.unlock();
      break;
    }
    mutex.unlock();

    cerr << messageStr.ascii();
  }
  cerr << endl;
}
```

А функция **stop()** так:

```
void Thread::stop()
{
  mutex.lock();
  stopped = true;
  mutex.unlock();
}
```

В сложных функциях, особенно при использовании исключений C++, легко можно ошибиться при выполнении последовательностей операций по запиранию/отпиранию мьютексов. Поэтому, в состав Qt включен класс **QMutexLocker**, который значительно упрощает работу с мьютексами. Конструктор класса **QMutexLocker** принимает объект **QMutex** в виде аргумента и запирает его. Деструктор класса **QMutexLocker** -- отпирает мьютекс. Например, с использованием класса **QMutexLocker**, функция **stop()** могла бы быть переписана следующим образом:

```
void Thread::stop()
{
```

```
  QMutexLocker locker(&mutex);
  stopped = true;
}
```

Семафоры в Qt реализованы в виде класса **QSemaphore**. Семафоры являются дальнейшим обобщением мьютексов и могут использоваться для защиты от одновременного доступа к нескольким идентичным ресурсам.
В следующей таблице приведено соответствие между **QSemaphore** и **QMutex**:

```
      QSemaphore semaphore(1);      |      QMutex mutex;
      semaphore++;                  |      mutex.lock();
      semaphore--;                  |      mutex.unlock();
```

Постфиксные операторы "**++**" и "**--**" захватывают и отпускают один ресурс, доступ к которому защищен семафором. Аргумент 1, который передается конструктору, указывает, что семафор обслуживает один единственный ресурс. Основное преимущество семафора состоит в том, что с его помощью можно захватить сразу несколько ресурсов, последовательно вызывая "**++**" несколько раз. Типичное применение семафоров -- операции обмена данными (**DataSize**) между потоками, с помощью общего циклического буфера некоторого объема (**BufferSize**):

```
  const int DataSize = 100000;
  const int BufferSize = 4096;
  char buffer[BufferSize];
```

Поток-источник пишет данные в буфер до тех пор, пока не заполнит его и затем продолжает запись данных с начала буфера, затирая данные, записанные ранее. Поток-приемник читает данные по мере их поступления. Рисунок 17.2 иллюстрирует процесс записи/чтения в/из буфер(а) размером в 16 байт.

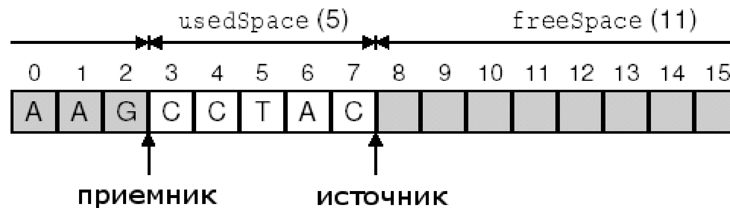


Рисунок 17.2. Модель источник/приемник.

Потребность в синхронизации очевидна: если поток-источник будет писать данные слишком быстро, то он затрет данные, которые еще не прочитаны потоком-приемником, и наоборот, если поток-приемник будет читать слишком быстро, то он "обгонит" поток-источник и начнет считывать неверные данные.

Самое простое решение этих проблем -- поток-источник заполняет весь буфер целиком, затем ожидает, пока поток-приемник не прочитает его, и так далее. Однако, на многопроцессорных системах такой прием приведет к снижению производительности. Более гибкое решение -- позволить потокам работать с разными частями буфера одновременно.
Как один из вариантов реализации подобного подхода -- использовать два семафора:

```
QSemaphore freeSpace(BufferSize);
QSemaphore usedSpace(BufferSize);
```

Семафор **freeSpace** управляет частью буфера, которая заполняется потоком-источником, а семафор **usedSpace** -- областью, которая доступна на чтение потоку-приемнику. Эти области не пересекаются между собой. Обе имеют размер BufferSize (4096).

В данном случае каждый байт области считается отдельным ресурсом. В реальных приложениях часто используются более крупные единицы измерения (например 64-х или 256-ти байтные блоки), чтобы уменьшить количество обращений к семафорам.

```
void acquire(QSemaphore &semaphore)
{
  semaphore++;
```

```
}
```

Функция **acquire()** предпринимает попытку захватить один ресурс (один байт в буфере). Для этих целей класс **QSemaphore** использует постфиксный оператор "**++**", но в нашем конкретном случае более удобным будет использовать функцию с именем **acquire()**.

```
void release(QSemaphore &semaphore)
{
  semaphore--;
}
```

Аналогичным образом реализована функция **release()**, являющаяся синонимом постфиксного оператора "**--**".

```
void Producer::run()
{
  for (int i = 0; i < DataSize; ++i) {
    acquire(freeSpace);
    buffer[i % BufferSize] = "ACGT"[(uint)rand() % 4];
    release(usedSpace);
  }
}
```

Источник захватывает один "свободный" байт. Если буфер заполнен данными, которые еще не прочитаны потоком-приемником, то вызов **acquire()** заблокирует работу источника до тех пор, пока приемник не начнет чтение данных. Как только байт будет захвачен, в него записывается случайный символ ('**A**', '**C**', '**G**' или '**T**'), после чего байт отпускается как "используемый", благодаря этому он становится доступен приемнику.

```
void Consumer::run()
{
  for (int i = 0; i < DataSize; ++i) {
    acquire(usedSpace);
    cerr << buffer[i % BufferSize];
    release(freeSpace);
  }
  cerr << endl;
}
```

Приемник захватывает один "используемый" байт. Если в буфере нет данных, доступных для чтения, то функция **acquire()** заблокирует работу приемника до тех пор, пока источник не запишет какие нибудь данные в буфер. Как только байт будет захвачен, символ выводится на консоль, после чего байт освобождается как "свободный", благодаря этому он становится доступен источнику.

```
int main()
{
  usedSpace += BufferSize;

  Producer producer;
  Consumer consumer;
  producer.start();
  consumer.start();
  producer.wait();
  consumer.wait();
  return 0;
}
```

Функция **main()** захватывает все байты как "используемые" (с помощью оператора "**+=**" класса **QSemaphore**), чтобы предотвратить преждевременное чтение данных приемником. Затем запускается поток-источник и вслед за ним -- поток приемник. В результате, поток-источник будет записывать данные, а вслед за ним, поток-приемник -- читать их.
После запуска программа будет выводить на консоль символы '**A**', '**C**', '**G**' и '**T**' в случайном порядке. После того, как программа выведет 100 000 символов, она завершит свою работу. Чтобы окончательно прояснить порядок работы потоков, попробуйте запретить вывод символов, принимаемых потоком-источником и выводите символ 'P', когда источник записывает один байт в

буфер, и символ 'c' -- когда поток-приемник читает один байт из буфера. Чтобы еще больше упростить понимание, можно уменьшить **DataSize** и **BufferSize**.

Для случая, когда **DataSize** == 10, а **BufferSize** == 4, вполне возможен результат: PcPcPcPcPcPcPcPcPcPc. Он говорит о том, что поток-приемник читает данные из буфера по мере их поступления -- оба потока работают с одинаковой скоростью. Возможен вариант, когда поток-источник успевает заполнить буфер целиком, до того как поток-приемник начнет чтение: PPPPccccPPPPccccPPcc. Существует масса других вариантов. Семафоры предоставляют большую свободу выбора системному планировщику, который может "изучать" поведение потоков и выбирать для них наиболее оптимальную политику планирования.

Еще один вариант синхронизации потоков может быть реализован на классах **QWaitCondition** и **QMutex**. Класс **QWaitCondition** дает потоку возможность возобновлять работу других потоков, при наступлении некоторого состояния. Что позволяет более точно управлять потоками, чем это возможно только на одних мьютексах. Чтобы продемонстрировать это на примере, мы опять вернемся к связке источник-приемник и реализуем тот же самый обмен данными с помощью классов **QWaitCondition** и **QMutex**.

```
const int DataSize = 100000;
const int BufferSize = 4096;
char buffer[BufferSize];

QWaitCondition bufferIsNotFull;
QWaitCondition bufferIsNotEmpty;
QMutex mutex;
int usedSpace = 0;
```

В дополнение к буферу обмена, мы объявили два экземпляра класса **QWaitCondition**, один экземпляр **QMutex** и одну переменную, которая хранит количество "используемых" байт.

```
void Producer::run()
{
  for (int i = 0; i < DataSize; ++i) {
    mutex.lock();
    while (usedSpace == BufferSize)
      bufferIsNotFull.wait(&mutex);
    buffer[i % BufferSize] = "ACGT"[(uint)rand() % 4];
    ++usedSpace;
    bufferIsNotEmpty.wakeAll();
    mutex.unlock();
  }
}
```

Работа потока-источника начинается с проверки -- не заполнен ли буфер. Если буфер заполнен, то он ждет, пока не наступит состояние "буфер не полон". Затем в буфер записывается один байт, содержимое переменной **usedSpace** увеличивается на 1 и возобновляются ("пробуждаются") все потоки, которые ожидают наступления состояния "буфер не пуст".

В данном примере мьютекс используется для предотвращения одновременного доступа к переменной **usedSpace**. Функция **QWaitCondition::wait()** может принимать первым аргументом запертый мьютекс, который отпирается, перед блокировкой вызвавшего ее потока, и опять запирается, перед тем как функция вернет управление.

В этом примере, цикл **while**

```
  while (usedSpace == BufferSize)
    bufferIsNotFull.wait(&mutex);
```

может быть заменен условным оператором:

```
  if (usedSpace == BufferSize) {
    mutex.unlock();
    bufferIsNotFull.wait();
```

```
    mutex.lock();
  }
```

Однако такой вариант неприемлем для случая, когда одновременно будут работать несколько потоков-источников, так как любой из потоков-источников может захватить мьютекс после выхода из **wait()**, и сделать ложным условие "буфер не полон".

```
void Consumer::run()
{
  for (int i = 0; i < DataSize; ++i) {
    mutex.lock();
    while (usedSpace == 0)
      bufferIsNotEmpty.wait(&mutex);
    cerr << buffer[i % BufferSize];
    --usedSpace;
    bufferIsNotFull.wakeAll();
    mutex.unlock();
  } cerr << endl;
}
```

Поток-приемник являет собой полную противоположность потоку-источнику. Он ожидает наступления состояния "буфер не пуст" и пробуждает все потоки, которые ожидают наступления состояния "буфер не полон".

Во всех наших примерах, потоки обращались к одним и тем же глобальным переменным. Но в некоторых случаях возникает необходимость хранить в глобальной переменной различные значения для различных потоков. Это часто называют как "механизм хранения локальных данных потока" (thread-local storage -- TLS). Добиться этого можно с помощью словаря, где в качестве ключа будет выступать числовой идентификатор потока (возвращаемый функцией **QThread::currentThread()**), но лучшее решение -- использовать специализированный класс **QThreadStorage<T>**.

Обычно экземпляр класса **QThreadStorage<T>** используют в качестве буфера. При наличии отдельных буферов для каждого из потоков, отпадает необходимость в постоянном запирании, отпирании и, возможно, ожидании мьютекса. Например:

```
QThreadStorage<QMap<int, double> *> cache;

void insertIntoCache(int id, double value)
{
  if (!cache.hasLocalData())
    cache.setLocalData(new QMap<int, double>);
  cache.localData()->insert(id, value);
}

void removeFromCache(int id)
{
  if (cache.hasLocalData())
    cache.localData()->remove(id);
}
```

Переменная **cache** хранит по одному указателю на **QMap<int, double>**, для каждого из потоков. (Из-за ограничений, накладываемых некоторыми компиляторам, шаблонный тип в **QThreadStorage<T>** должен быть указателем.) При первом обращении к буферу из некоторого потока, когда функция **hasLocalData()** возвращает **false**, мы создаем экземпляр класса **QMap<int, double>**.
В дополнение к буферам, класс **QThreadStorage<T>** может использоваться для создания глобальных переменных, которые хранят код последней ошибки (подобных переменной **errno**) и предотвращают взаимовлияние потоков друг на друга.

17.2 Взаимодействие с главным потоком приложения

На запуске программы, написанной с использованием библиотеки Qt, стартует главный поток приложения. Это единственный поток, в котором допускается создание экземпляра класса **QApplication** и вызов его метода **exec()**. Всвязи с этим, главный поток приложения часто называют

GUI-потоком. После вызова функции **exec()** этот поток либо ждет поступления события, либо обрабатывает какое нибудь событие.

Главный поток может запускать другие потоки, с помощью экземпляров классов, производных от **QThread**. Новые потоки могут обмениваться информацией между собой через глобальные переменные, с использованием мьютексов, семафоров или ожидая наступления определенного состояния. Но это совершенно не подходит для организации взаимодействий между главным и второстепенными потоками в приложении, поскольку вышеперечисленные методики могут блокировать главный поток, "замораживая" тем самым интерфейс с пользователем.
Для этих целей обычно используется механизм событий Qt, который допускает создание нестандартных типов событий и их передачу через вызов метода **QApplication::postEvent()**. Кроме того, функция **postEvent()** является потоко-безопасной, поэтому она может использоваться для передачи событий в главный поток из любого другого потока.

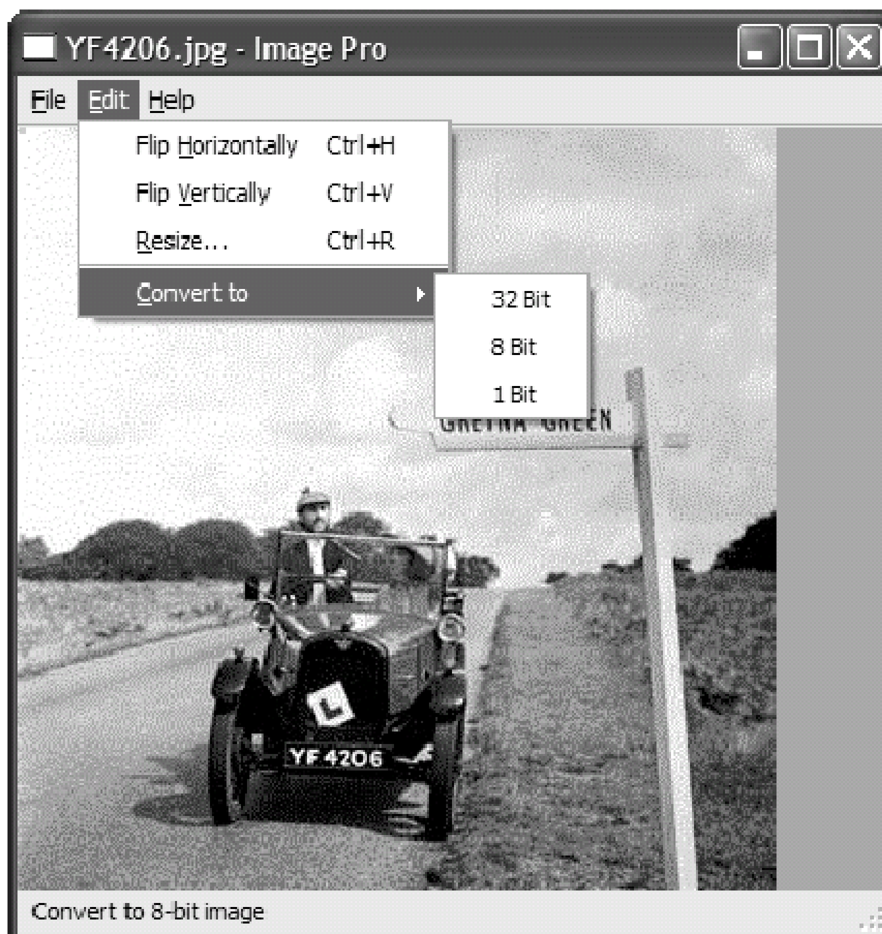


Рисунок 17.3. Внешний вид приложения Image Pro.

Принципы организации обмена событиями между потоками, мы будем рассматривать на примере приложения **Image Pro**. Оно предназначено для работы с изображениями и позволяет вращать их, изменять размеры и глубину цветопередачи. Длительные операции будут выполняться в дополнительном потоке, чтобы избежать блокировки главного цикла обработки событий приложения. Это особенно актуально при работе с большими изображениями. Второстепенный поток имеет список заданий, или "транзакций", которые необходимо выполнить и посылает события главному потоку, чтобы проинформировать его о ходе выполнения операции.

```
ImageWindow::ImageWindow(QWidget *parent, const char *name)
    : QMainWindow(parent, name)
{
  thread.setTargetWidget(this);
  ...
}
```

В конструкторе назначается виджет-получатель событий. События второстепенного потока будут отправляться этому виджету. Переменная **thread** относится к классу **TransactionThread**, который мы опишем чуть ниже.

```
void ImageWindow::flipHorizontally()
{
  addTransaction(new FlipTransaction(Horizontal));
}
```

Слот **flipHorizontally()** создает транзакцию "**flip**" ("отобразить") и регистрирует ее вызовом функции **addTransaction()**. Аналогичным образом реализованы функции **flipVertical()**, **resizeImage()**, **convertTo32Bit()**, **convertTo8Bit()** и **convertTo1Bit()**.

```
void ImageWindow::addTransaction(Transaction *transact)
{
  thread.addTransaction(transact);
  openAct->setEnabled(false);
  saveAct->setEnabled(false);
  saveAsAct->setEnabled(false);
}
```

Функция **addTransaction()** добавляет транзакцию в очередь заданий второстепенного потока и запрещает операции **Open**, **Save** и **Save As** на время ее выполнения.

```
void ImageWindow::customEvent(QCustomEvent *event)
{
  if ((int)event->type() == TransactionStart) {
    TransactionStartEvent *startEvent =
            (TransactionStartEvent *)event;
    infoLabel->setText(startEvent->message);
  } else if ((int)event->type() == AllTransactionsDone) {
    openAct->setEnabled(true);
    saveAct->setEnabled(true);
    saveAsAct->setEnabled(true);
    imageLabel->setPixmap(QPixmap(thread.image()));
    infoLabel->setText(tr("Ready"));
    modLabel->setText(tr("MOD"));
    modified = true;
    statusBar()->message(tr("Done"), 2000);
  } else {
    QMainWindow::customEvent(event);
  }
}
```

Функция **customEvent()** объявлена в классе **QObject** и предназначена для обработки нестандартных событий. Константы **TransactionStart** и **AllTransactionsDone** определены в **transactionthread.h**, как:

```
  enum { TransactionStart = 1001, AllTransactionsDone = 1002 };
```

Стандартные события Qt имеют значения ниже 1000, поэтому более высокие значения могут свободно использоваться для создания своих, нестандартных событий.
Нестандартные события создаются как экземпляры класса **QCustomEvent**, производного от **QEvent**, которые, кроме типа события, могут хранить дополнительный указатель типа **void**. Класс события **TransactionStart** порожден от **QCustomEvent** и имеет одну дополнительную переменную-член:

```
class TransactionStartEvent : public QCustomEvent
{
public:
  TransactionStartEvent();

  QString message;
};

TransactionStartEvent::TransactionStartEvent()
    : QCustomEvent(TransactionStart)
{
}
```

В конструкторе класса мы передаем константу **TransactionStart** унаследованному конструктору, инициализируя таким образом тип события.

Теперь перейдем к классу **TransactionThread**:

```
class TransactionThread : public QThread
{
public:
  void run();
  void setTargetWidget(QWidget *widget);
  void addTransaction(Transaction *transact);
  void setImage(const QImage &image);
  QImage image();

private:
  QWidget *targetWidget;
  QMutex mutex;
  QImage currentImage;
  std::list<Transaction *> transactions;
};
```

Класс **TransactionThread** имеет список заданий (транзакций), которые исполняются в порядке очередности поступления.

```
void TransactionThread::addTransaction(Transaction *transact)
{
  QMutexLocker locker(&mutex);
  transactions.push_back(transact);
  if (!running())
    start();
}
```

Функция **addTransaction()** добавляет новое задание в очередь транзакций и запускает поток на исполнение, если он еще не запущен.

```
void TransactionThread::run()
{
  Transaction *transact;

  for (;;) {
    mutex.lock();
    if (transactions.empty()) {
      mutex.unlock();
      break;
    }
    QImage oldImage = currentImage;
    transact = *transactions.begin();
    transactions.pop_front();
    mutex.unlock();

    TransactionStartEvent *event = new TransactionStartEvent;
    event->message = transact->messageStr();
    QApplication::postEvent(targetWidget, event);

    QImage newImage = transact->apply(oldImage);
    delete transact;

    mutex.lock();
    currentImage = newImage;
    mutex.unlock();
  }
  QApplication::postEvent(targetWidget,
                          new QCustomEvent(AllTransactionsDone));
}
```

Функция **run()** обходит список заданий и выполняет их (вызовом **apply()**). Доступ к объектам **transactions** и **currentImage** осуществляется под защитой мьютекса.

Когда транзакция запускается, в приложение, выбранному виджету (**ImageWindow**), посылается событие **TransactionStart**. После выполнения всех транзакций -- событие **AllTransactionsDone**.

```
class Transaction
{
public:
  virtual QImage apply(const QImage &image) = 0;
  virtual QString messageStr() = 0;
};
```

Класс **Transaction** -- это абстрактный класс, который служит основой для создания классов, выполняющих определенные действия над изображением. В нашем примере, это классы-потомки: **FlipTransaction**, **ResizeTransaction** и **ConvertDepthTransaction**. Мы рассмотрим только **FlipTransaction**, остальные два класса реализованы аналогичным образом.

```
class FlipTransaction : public Transaction
{
public:
  FlipTransaction(Qt::Orientation orient);

  QImage apply(const QImage &image);
  QString messageStr();

private:
  Qt::Orientation orientation;
};
```

Конструктору класса передается один аргумент, который определяет направление (ориентацию) отражения (**Horizontal** или **Vertical**).

```
QImage FlipTransaction::apply(const QImage &image)
{
  return image.mirror(orientation == Qt::Horizontal,
                      orientation == Qt::Vertical);
}
```

Для того, чтобы отразить изображение, функция **apply()** обращается к методу **QImage::mirror()** и возвращает полученный результат.

```
QString FlipTransaction::messageStr()
{
  if (orientation == Qt::Horizontal)
    return QObject::tr("Flipping image horizontally...");
  else
    return QObject::tr("Flipping image vertically...");
}
```

Функция **messageStr()** возвращает текст сообщения, которое будет отображаться в строке состояния приложения во время выполнения транзакции. Эта функция вызывается из **ImageWindow::customEvent()**, в контексте главного потока приложения.

Для длительных операций можно предусмотреть передачу сведений о ходе выполнения. Для этого нужно создать дополнительное событие и с его помощью посылать приложению процент выполнения задания.

17.3 Работа с классами Qt вне главного потока

Функция называется потоко-безопасной (**thread-safe**), когда она может свободно вызываться из нескольких потоков одновременно. Если две потоко-безопасные функции, вызываемые из разных потоков одновременно, работают с одними и теми же данными, то результат выполнения таких функций всегда предсказуем. Распространив это определение на классы можно сказать, что класс является потоко-безопасным, когда все его методы могут одновременно вызываться из нескольких потоков, без появления непредсказуемых побочных эффектов, даже если они взаимодействуют с оним и тем же объектом.

Среди потоко-безопасных классов в Qt можно назвать: **QThread**, **QMutex**, **QMutexLocker**, **QSemaphore**, **QThreadStorage<T>** и **QWaitCondition**. Кроме того, следующие функции-члены являются потоко-безопасными: **QApplication::postEvent()**, **QApplication::removePostedEvent()**, QEventLoop::wakeUp() и **QApplication::removePostedEvents()**.

Большинство невизуальных классов Qt соответствуют менее строгому требованию -- реентерабельности. Класс называется реентерабельным, если он допускает одновременное существование нескольких экземпляров в различных потоках. Однако, одновременный доступ к реентерабельным объектам из нескольких потоков может оказаться далеко не безопасен и потому должен выполняться под защитой мьютексов. Как правило, любой класс C++, который не использует глобальные или иные разделяемые данные, является реентерабельным.

Класс **QObject** -- реентерабельный, но ни один из его потомков в Qt не является таковым. Как следствие -- мы не можем напрямую обращаться к виджетам вне контекста главного потока приложения. Если, скажем, нужно изменить текст в **QLabel** из второстепенного потока, то необходимо послать нестандартное событие в главный поток, посредством которого изменить текст надписи.

Операция удаления объекта **QObject**, с помощью **delete**, не является реентерабельной. Поэтому, при необходимости удаления объекта **QObject** из другого потока, нам придется вызвать метод **QObject::deleteLater()**, который посылает событие "**deferred delete**".

В контексте любого потока допускается использование механизма сигналов и слотов. При выдаче сигнала, связанный с ним слот исполняется в контексте того же потока, а не в потоке, где был создан объект-приемник. Таким образом сигналы и слоты не могут использоваться для организации взаимодействий между потоками.

Класс **QTimer**, и классы для работы с сетью **QFtp**, **QHttp**, **QSocket** и **QSocketNotifier**, целиком зависят от цикла обработки событий, поэтому они не могут использоваться за пределами главного потока. Единственный сетевой класс, который не зависит от цикла обработки событий -- это **QSocketDevice**, являющийся "оберткой" вокруг платформо-зависимого сетевого API. Некоторые программисты считают, что использование **QSocketDevice** в синхронном режиме, вне контекста главного потока, дает более простой код, нежели использование **QSocket** (который работает асинхронно), а благодаря работе вне главного потока -- он не блокирует цикл обработки событий.

Модули SQL и OpenGL так же могут использоваться в многопоточных приложениях, но имеют свои собственные ограничения, которые отличаются от системы к системе. За более подробной информацией обращайтесь по адресу: *http://doc.trolltech.com/3.2/sql-driver.html*, а так же к статье "Glimpsing the Third Dimension", в ежеквартальнике Qt Quarterly: *http://doc.trolltech.com/qq/qq06-glimpsing.html*.

Многие из невизуальных классов Qt, включая **QImage**, **QString** и другие, используют явные и неявные методы оптимизации, связанные с разделением данных между объектами. Эти классы являются реентерабельными, за исключением конструкторов копирования и операторов присваивания. Когда создается копия объекта, то копируются только указатели на данные. Это может привести к непредсказуемым последствиям, если копии объекта попытаются одновременно, из нескольких потоков, изменить данные. В подобных ситуациях можно прибегнуть к услугам класса **QDeepCopy<T>**, например:

```
QString password;
QMutex mutex;

void setPassword(const QString &str)
{
  mutex.lock();
  password = QDeepCopy<QString>(str);
  mutex.unlock();
}
```

Возможно, Qt 4 будет иметь более широкую поддержку потоков. Так, среди всего прочего ожидается, что механизм сигналов и слотов будет расширен до поддержки установления связей через границы потоков, и позволит отказаться от необходимости создания нестандартных событий для взаимодействия с главным потоком. Ожидается так же, что невизуальные классы, подобные **QSocket** и **QTimer**, смогут использоваться вне контекста главного потока, и что необходимость в использовании класса **QDeepCopy<T>** отпадет.

18 Платформо-зависимые особенности

В этой главе будут рассмотрены некоторые из платформо-зависимых особенностей библиотеки Qt. Начнем с описания доступа к системному API: Win32 -- для Windows, Core Graphics -- для Mac OS X и Xlib -- для X11. Затем перейдем к рассмотрению расширения ActiveQt и покажем, как добавить поддержку ActiveX в свои приложения, для платформы Windows. В последнем разделе главы опишем, как можно добиться взаимодействия приложения с менеджером сессии на платформе X11. Помимо расширений, представленных здесь, в состав версии Qt Enterprise Edition включено расширение Qt/Motif, которое облегчает миграцию приложений от Motif и Xt -- к Qt. Аналогичное расширение, для Tcl/Tk приложений предоставляет froglogic, и конвертер ресурсов Microsoft Windows -- Klaralvdalens Datakonsult. Для разработчиков устройств, Trolltech предоставляет среду исполнения приложений -- Qtopia. За дополнительной информацией обращайтесь по адресам:
http://doc.trolltech.com/3.2/motif-extension.html
http://www.froglogic.com/
http://www.klaralvdalens-datakonsult.se/?page=products'=knut
http://www.trolltech.com/products/qtopia/

18.1 Взаимодействие с API операционной системы

Qt предоставляет достаточно мощный API, которого вполне достаточно для большинства приложений на любых платформах. Но в некоторых случаях невозможно обойтись без обращения к средствам, предоставляемым самой операционной системой. В этом разделе мы покажем, как можно организовать взаимодействие приложений с API операционной системы для выполнения специфических задач.

На всех, поддерживаемых библиотекой Qt, платформах, класс **QWidget** реализует функцию **winId()**, которая возвращает числовой идентификатор окна (HWND -- в терминах Windows). Коме того, **QWidget** предоставляет статическую функцию **find()**, которая возвращает **QWidget** по заданному идентификатору окна. Этот идентификатор может быть передан функциям API операционной системы для достижения эффектов, которые зависят от платформы. Например, следующий код использует **winId()**, чтобы добиться эффекта полупрозрачности, при отображении визуального компонента **QLabel** в Mac OS X, используя для этого функции графического ядра "Core Graphics". [9]

```
#include <qapplication.h>
#include <qlabel.h>
#include <qt_mac.h>

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  QLabel *label = new QLabel("Hello Qt!", 0);
  app.setMainWidget(label);

  CGSWindowRef winRef =
      GetNativeWindowFromWindowRef((WindowRef)label->winId());
  CGSSetWindowAlpha(_CGSDefaultConnection(), winRef, 0.5);

  label->show();
  return app.exec();
}
```

Следующий код добивается того же самого эффекта на платформе Windows, используя для этого Win32 API:

```
#define _WIN32_WINNT 0x0501

#include <qapplication.h>
#include <qt_windows.h>

int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  QLabel *label = new QLabel("Hello Qt!", 0);
```

```
  app.setMainWidget(label);

  int exstyle = GetWindowLong(label->winId(), GWL_EXSTYLE);
  exstyle |= WS_EX_LAYERED;
  SetWindowLong(label->winId(), GWL_EXSTYLE, exstyle);
  SetLayeredWindowAttributes(label->winId(), 0, 128,
                             LWA_ALPHA);

  label->show();
  return app.exec();
}
```

Этот код будет корректно работать в среде Windows 2000/XP. Если вы желаете собрать и запустить приложение в более ранних версиях Windows, которые не поддерживают полупрозрачность, вам придется использовать библиотеку **QLibrary**, которая определяет функцию **SetLayeredWindowAttributes** во время исполнения, а не во время сборки:

```
typedef BOOL (__stdcall *PSetLayeredWindowAttributes)
        (HWND, COLORREF, BYTE, DWORD);
PSetLayeredWindowAttributes pSetLayeredWindowAttributes =
        (PSetLayeredWindowAttributes) QLibrary::resolve("user32",
                "SetLayeredWindowAttributes");

  if (pSetLayeredWindowAttributes) {
    int exstyle = GetWindowLong(label->winId(), GWL_EXSTYLE);
    exstyle |= WS_EX_LAYERED;
    SetWindowLong(label->winId(), GWL_EXSTYLE, exstyle);
    pSetLayeredWindowAttributes(label->winId(), 0, 128,
                                LWA_ALPHA);
}
```

Qt/Windows использует такой подход, чтобы гарантировать возможность использования расширенных особенностей, таких как поддержка Unicode и преобразования шрифтов, везде, где это возможно, допуская при этом возможность работы приложений на старых версиях Windows.
В X11 нет стандартного способа добиться эффекта полупрозрачности. Однако, в этой среде у нас есть возможность изменить свойства окна:

```
  Atom atom = XInternAtom(win->x11Display(), "MY_PROPERTY", False);
  long data = 1;
  XChangeProperty(win->x11Display(), win->winId(), atom, atom,
                  32, PropModeReplace, (unsigned char *)&data, 1);
```

Qt/Embedded отличается от всех остальных версий Qt, где все это реализуется прямо поверх буфера изображений (frame buffer) Linux, без помощи промежуточного API. Она так же предоставляет свою собственную оконную систему -- QWS. За дополнительной информацией по Qt/Embedded обращайтесь по адресам: http://www.trolltech.com/products/embedded/ и http://doc.trolltech.com/3.2/winsystem.html.

При необходимости, можно использовать платформо-зависимые особенности не в ущерб переносимости, заключая специфический код в условные операторы препроцессора **#if** и **#endif**. Например:

```
#if defined(Q_WS_MAC)
  CGSWindowRef winRef =
      GetNativeWindowFromWindowRef((WindowRef)label->winId());
  CGSSetWindowAlpha(_CGSDefaultConnection(), winRef, 0.5);
#endif
```

Для каждой из платформ, Qt определяет один из следующих символов: **Q_WS_WIN**, **Q_WS_X11**, **Q_WS_MAC** или **Q_WS_QWS**. Перед обращением к любому из них, исходный код приложения должен подключить хотя бы один заголовочный файл библиотеки. Кроме того, Qt предоставляет следующие символы препроцессора, для определения типа операционной системы:

- Q_OS_WIN32
- Q_OS_WIN64
- Q_OS_CYGWIN
- Q_OS_DGUX
- Q_OS_DYNIX
- Q_OS_FREEBSD
- Q_OS_LINUX
- Q_OS_LYNX
- Q_OS_NETBSD
- Q_OS_QNX6
- Q_OS_RELIANT
- Q_OS_SCO

- Q_OS_MAC
- Q_OS_AIX
- Q_OS_BSDI
- Q_OS_HPUX
- Q_OS_HURD
- Q_OS_IRIX
- Q_OS_HPUX
- Q_OS_OSF
- Q_OS_QNX
- Q_OS_SOLARIS
- Q_OS_ULTRIX
- Q_OS_UNIXWARE

Для большего удобства, Qt определяет символ **Q_OS_WIN** для Win32 и Win64, и **Q_OS_UNIX** для всех Unix-подобных систем, включая Mac OS X. Для уточнения номера версии операционной системы, во время исполнения, можно воспользоваться функциями **QApplication::winVersion()** (95, 98 и т.д.) и **QApplication::macVersion()** (10.0, 10.1 и т.д.).
Некоторые классы визуальных компонентов предоставляют платформозависимую функцию **handle()**. На рисунке 18.1 перечислены типы значений, возвращаемых функцией **handle()**, в зависимости от типа платформы.

	Windows	X11	Mac OS X	Embedded
QCursor	HCURSOR	Cursor	int	int
QFont	HFONT	Font	FMFontFamily	FontID
QPaintDevice	HDC	Drawable	GWorldPtr	N/A
QPainter	HDC	Drawable	GWorldPtr	N/A
QRegion	HRGN	Region	RgnHandle	void *
QSessionManager	N/A	SmcConn	N/A	N/A

Рисунок 18.1. Типы результата, возвращаемого функцией handle().

Некоторые классы, такие как **QWidget**, **QPixmap**, **QPrinter** и **QPicture**, ведут свою родословную от класса **QPaintDevice**. В X11 и Mac OS X, **handle()** означает то же самое самое, что и **winId()** класса **QWidget**. В Windows функция **handle()** возвращает контекст устройства, в то время как **winId()** -- дескриптор окна. Похожим образом, функция **hbm()**, класса **QPixmap**, в среде Windows, возвращает дескриптор растра (HBITMAP).

В X11, класс **QPaintDevice** предоставляет множество функций, которые возвращают различные указатели и дескрипторы, включая **x11Display()** и **x11Screen()**. Они могут использоваться для настройки графического контекста X11 в объектах **QWidget** или **QPixmap**.

Нередки ситуации, когда приложения, написанные с использованием библиотеки Qt, должны получить доступ к низкоуровневым событиям (**XEvents** -- в X11, **MSG** -- в Windows и Mac OS X, **QWSEvents** -- в Qt/Embedded) прежде, чем они будут преобразованы в **QEvent**. Добиться этого можно, породив свой класс от **QApplication** и перекрыв соответствующие фильтры событий: **winEventFilter()**, **x11EventFilter()**, **macEventFilter()** и **qwsEventFilter()**.

Можно получить доступ к специфическим для платформы событиям, посылаемым заданному виджету, перекрыв одну из функций: **winEvent()**, **x11Event()**, **macEvent()** или **qwsEvent()**. Этот прием может оказаться единственно возможным, для обработки событий, которые обычно игнорируются библиотекой, например: событий от джойстика.

За дополнительной информацией, касающейся платформо-зависимых проблем, включая вопросы программирования с использованием Qt/Embedded и распространения готовых приложений для различных платформ, обращайтесь по адресу: *http://doc.trolltech.com/3.2/winsystem.html*.

18.2 ActiveX

Технология Microsoft ActiveX позволяет приложениям включать интерфейсные компоненты, предоставляемые другими приложениями и библиотеками. Она базируется на технологии Microsoft COM и определяет один набор интерфейсов для приложений, которые которые используют внешние компоненты, и другой набор -- для приложений и библиотек, которые могут предоставить эти компоненты.
Версия библиотеки Qt/Windows Enterprise Edition, включает в себя **ActiveQt framework**, которая позволяет объединить ActiveX и Qt. ActiveQt состоит из двух модулей:

- **QAxContainer** -- который позволяет использовать COM-объекты и встраивать в приложение элементы управления ActiveX.

- **QAxServer** -- позволяет экспортировать наши собственные COM-объекты и элементы управления ActiveX, написанные с помощью Qt.

В нашем первом примере мы встроим Windows Media Player в приложение Qt, используя для этого **QAxContainer**.

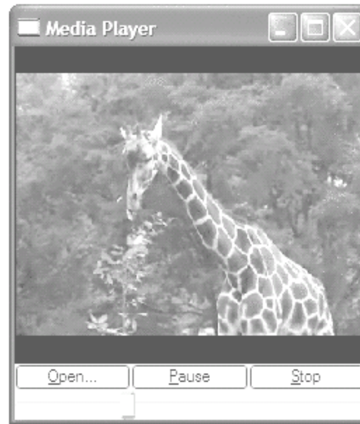


Рисунок 18.2. Внешний вид приложения Media Player.

Главное окно приложения относится к классу **PlayerWindow**:

```
class PlayerWindow : public QWidget
{
  Q_OBJECT
  Q_ENUMS(ReadyStateConstants)
public:
  enum PlayStateConstants { Stopped = 0, Paused = 1, Playing = 2 };
  enum ReadyStateConstants { Uninitialized = 0, Loading = 1,
                             Interactive = 3, Complete = 4 };

  PlayerWindow(QWidget *parent = 0, const char *name = 0);

protected:
  void timerEvent(QTimerEvent *event);

private slots:
  void onPlayStateChange(int oldState, int newState);
  void onReadyStateChange(ReadyStateConstants readyState);
  void onPositionChange(double oldPos, double newPos);
  void sliderValueChanged(int newValue);
  void openFile();
```

Класс **PlayerWindow** порожден от класса **QWidget**. Макрос **Q_ENUMS()** используется для того, чтобы сообщить утилите moc о том, что тип **ReadyStateConstants**, используемый слотом **onReadyStateChange()**, относится к перечислениям.

```
private:
  QAxWidget *wmp;
  QToolButton *openButton;
  QToolButton *playPauseButton;
  QToolButton *stopButton;
  QSlider *seekSlider;
  QString fileFilters;
  int updateTimer;
};
```

В приватной секции объявлена переменная-член типа **QAxWidget ***.

```
PlayerWindow::PlayerWindow(QWidget *parent, const char *name)
    : QWidget(parent, name)
{
  ...
  wmp = new QAxWidget(this);
  wmp->setControl("{22D6F312-B0F6-11D0-94AB-0080C74C7E95}");
```

В конструкторе создается объект **QAxWidget**, в который будет внедрен элемент управления ActiveX -- Windows Media Player. Модуль **QAxContainer** содержит три класса: **QAxObject** -- инкапсулирующий COM-объект, **QAxWidget** -- инкапсулирующий элемент ActiveX и **QAxBase**, реализующий базовую функциональность классов **QAxObject** и **QAxWidget**.

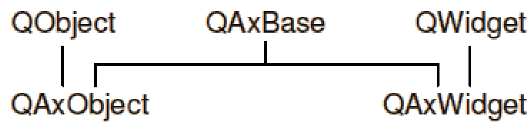


Рисунок 18.3. Дерево наследования в модуле QAxContainer.

Методу **setControl()**, объекта **QAxWidget**, передается идентификатор (GUID) элемента управления Windows Media Player 6.4. С его помощью будет создан экземпляр требуемого компонента, после чего все свойства, события и методы элемента ActiveX станут доступны как обычные свойства, сигналы и слоты Qt объекта **QAxWidget**.

Типы COM автоматически конвертируются в соответствующие типы Qt, в соответствии с таблицей, приведенной на рисунке 18.4. Так например, входной параметр типа **VARIANT_BOOL** преобразуется в тип **bool**, а выходной **VARIANT_BOOL** -- в **bool &**. Если результатом преобразования является класс Qt (**QString**, **QDateTime** и т.п.), то входной параметр получает тип константной ссылки на этот класс (например **const QString &**).

COM types	Qt type
VARIANT_BOOL	bool
char, short, int, long	int
unsigned char, unsigned short, unsigned int, unsigned long	uint
float, double	double
CY	Q_LLONG
BSTR	QString
DATE	QDateTime
OLE_COLOR	QColor
SAFEARRAY(VARIANT)	QValueList<QVariant>
SAFEARRAY(BYTE)	QByteArray
VARIANT	QVariant
IFontDisp *	QFont
IPictureDisp *	QPixmap

Рисунок 18.4. Отношения между типами COM и Qt.

Чтобы получить список всех свойств, сигналов и слотов, доступных в **QAxObject** или **QAxWidget**, вызовите **generateDocumentation()** или воспользуйтесь утилитой командной строки **dumpdoc**, размещенной в подкаталоге **extensions\activeqt\example**.

```
wmp->setProperty("ShowControls", QVariant(false, 0));
wmp->setSizePolicy(QSizePolicy::Expanding,
                   QSizePolicy::Expanding);
connect(wmp, SIGNAL(PlayStateChange(int, int)),
        this, SLOT(onPlayStateChange(int, int)));
connect(wmp, SIGNAL(ReadyStateChange(ReadyStateConstants)),
        this, SLOT(onReadyStateChange(ReadyStateConstants)));
connect(wmp, SIGNAL(PositionChange(double, double)),
        this, SLOT(onPositionChange(double, double)));
```

Вслед за функцией **setControl()**, в конструкторе **PlayerWindow**, вызывается **setProperty()**, чтобы записать значение **false** в свойство **ShowControls** плеера, поскольку приложение предоставляет свои элементы управления. Функция **setProperty()** реализована в классе **QObject** и может использоваться для установки значений свойств как в COM-объектах, так и в обычных классах Qt. Ее второй аргумент

имеет тип **QVariant**. Поскольку некоторые компиляторы C++ до сих пор не поддерживают должным образом тип **bool**, приходится передавать конструктору **QVariant** заготовку типа **int**. Для типов отличных от **bool**, преобразование проходит автоматически.
Вслед за нею вызывается **setSizePolicy()**, отдавая компоненту все доступное на форме пространство. После чего выполняется подключение трех событий **ActiveX** плеера к слотам приложения.
Оставшаяся часть конструктора выполняет обычные, для любого Qt класса, действия, за исключением, разве что, подключения трех Qt-сигналов к слотам COM-объекта (**Play()**, **Pause()** и **Stop()**).
Оставим конструктор в покое и рассмотрим функцию **timerEvent()**:

```
void PlayerWindow::timerEvent(QTimerEvent *event)
{
  if (event->timerId() == updateTimer) {
    double curPos = wmp->property("CurrentPosition").toDouble();
    onPositionChange(-1, curPos);
  } else {
    QWidget::timerEvent(event);
  }
}
```

Эта функция регулярно вызывается через определенные интервалы времени, во время проигрывания. С ее помощью устанавливается положение движка. Значение положения движка определяется путем чтения свойства **CurrentPosition** элемента ActiveX, с помощью функции **property()**.
Мы не привели остальную часть кода конструктора по той причине, что он напрямую не связан с ActiveX и не демонстрирует ничего такого, о чем бы мы не говорили ранее.
Чтобы связать приложение с модулем **QAxContainer**, необходимо в файл проекта добавить строку:

```
LIBS += -lqaxcontainer
```

Иногда возникает необходимость прямого вызова методов COM-объекта (без соединения с сигналом Qt). Самый простой способ -- вызвать функцию **dynamicCall()**, которой в качестве первого аргумента передать имя и сигнатуру вызываемого метода, а аргументы метода -- в виде последующих входных параметров, например:

```
  wmp->dynamicCall("TitlePlay(uint)", 6);
```

Функция **dynamicCall()** может принимать до восьми аргументов типа **QVariant** и возвращает значение типа **QVariant**. Если методу нужно передать аргумент типа **IDispatch *** или **IUnknown ***, то можно сначала инкапсулировать его в объект **QAxObject**, а затем вызвать метод **asVariant()**, чтобы преобразовать его в тип **QVariant**. Если метод COM-объекта возвращает значение типа **IDispatch *** или **IUnknown**, или если нужно получить доступ к свойству COM-объекта одного из этих типов, то нам придется воспользоваться функцией **querySubObject()**:

```
  QAxObject *session = outlook.querySubObject("Session");
  QAxObject *defaultContacts =
          session->querySubObject("GetDefaultFolder(OlDefaultFolders)",
                                  "olFolderContacts");
```

Если вызываемая функция имеет аргументы неподдерживаемых типов, вы должны сначала получить COM-интерфейс, вызовом функции **QAxBase::queryInterface()**, а затем вызвать нужный метод напрямую. Не забывайте вызвать **Release()**, когда надобность в интерфейсе отпадет.

Если необходимость в вызове подобных функций возникает достаточно часто, вы можете создать класс-потомок от **QAxObject** или от **QAxWidget**, и добавить функции-члены для работы с COM-интерфейсом. При этом вы должны помнить, что потомки классов **QAxObject** и **QAxWidget** не могут определять новые свойства, сигналы и слоты.
Теперь перейдем к обзору модуля **QAxServer**. Этот модуль позволяет превратить обычную Qt-программу в ActiveX-сервер. Сервер может быть выполнен либо в виде динамической библиотеки, либо в виде автономного приложения. Серверы, собранные в виде динамической библиотеки часто называют "внутренними" (in-process) серверами, а автономные приложения -- "внешними" (out-of-process) серверами.

Наш первый пример будет собран в виде "внутреннего" сервера, который реализует виджет, отображающий прыгающий шарик. Мы так же покажем, как встроить виджет в Internet Explorer.

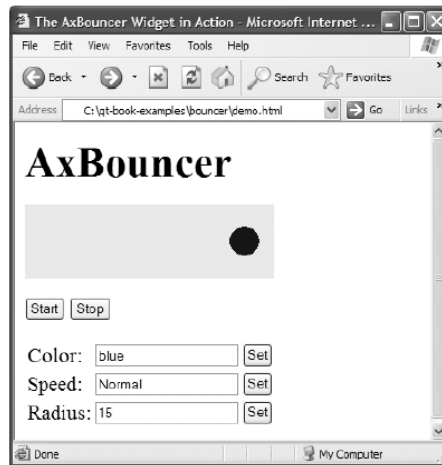


Рисунок 18.5. Виджет AxBouncer в Internet Explorer.

Начнем с определения класса виджета **AxBouncer**:

```
class AxBouncer : public QWidget, public QAxBindable
{
  Q_OBJECT
  Q_ENUMS(Speed)
  Q_PROPERTY(QColor color READ color WRITE setColor)
  Q_PROPERTY(Speed speed READ speed WRITE setSpeed)
  Q_PROPERTY(int radius READ radius WRITE setRadius)
  Q_PROPERTY(bool running READ isRunning)
```

Класс **AxBouncer** является потомком сразу двух классов: **QWidget** и **QAxBindable**. Класс **QAxBindable** реализует интерфейс между виджетом и ActiveX-клиентом. Любой виджет может быть экспортирован как элемент ActiveX, но порождая его от **QAxBindable**, мы получаем возможность извещать клиента при изменении значений свойств виджета, а так же реализовать COM-интерфейсы, в дополнение к тем, что уже реализованы в **QAxServer**.
При использовании множественного наследования, с участием класса **QObject**, вы всегда должны указывать этот класс первым в списке предков, чтобы обеспечить корректную работу утилиты **moc**.
В нашем определении мы объявили три свойства, доступные для чтения/записи и одно свойство, доступное только для чтения. Макрос **Q_ENUMS()** сообщает утилите moc, что **Speed** -- это перечисление, которое объявлено в публичной секции

```
public:
  enum Speed { Slow, Normal, Fast };

  AxBouncer(QWidget *parent = 0, const char *name = 0);

  void setSpeed(Speed newSpeed);
  Speed speed() const { return ballSpeed; }
  void setRadius(int newRadius);
  int radius() const { return ballRadius; }
  void setColor(const QColor &newColor);
  QColor color() const { return ballColor; }
  bool isRunning() const { return myTimerId != 0; }
  QSize sizeHint() const;
  QAxAggregated *createAggregate();

public slots:
  void start();
  void stop();

signals:
  void bouncing();
```

В конструкторе **AxBouncer** нет ничего необычного, это стандартный конструктор виджета, с аргументами **parent** и **name**. Макрос **QAXFACTORY_DEFAULT()**, который используется для экспорта компонента, принимает в виде параметра конструктор именно с такой сигнатурой.
Функция **createAggregate()** перекрывает метод родительского класса **QAxBindable**. К ее описанию мы вскоре вернемся.

```
protected:
  void paintEvent(QPaintEvent *event);
  void timerEvent(QTimerEvent *event);

private:
  int intervalInMilliseconds() const;

  QColor ballColor;
  Speed ballSpeed;
  int ballRadius;
  int myTimerId;
  int x;
  int delta;
};
```

Защищенная и приватная секции класса не содержат ничего необычного, что отличало бы их от привычных виджетов Qt.

```
AxBouncer::AxBouncer(QWidget *parent, const char *name)
    : QWidget(parent, name, WNoAutoErase)
{
  ballColor = blue;
  ballSpeed = Normal;
  ballRadius = 15;
  myTimerId = 0;
  x = 20;
  delta = 2;
}
```

Конструктор выполняет инициализацию приватных переменных.

```
void AxBouncer::setColor(const QColor &newColor)
{
  if (newColor != ballColor amp;amp; requestPropertyChange("color")) {
    ballColor = newColor;
    update();
    propertyChanged("color");
  }
}
```

Функция **setColor()** записывает новое значение цвета в переменную-член **color** и перерисовывает виджет, вызовом функции **update()**.
Необычными моментами здесь являются обращения к функциям **requestPropertyChange()** и **propertyChanged()**. Эти функции унаследованы от класса **QAxBindable**. В идеале эти функции должны вызываться в паре всякий раз, когда необходимо изменить значение свойства. Функция **requestPropertyChange()** запрашивает права клиента на изменение свойства и возвращает **true**, если клиенту позволено это делать. Функция **propertyChanged()** извещает клиента о том, что изменение произведено.

Функции **setSpeed()** и **setRadius()**, которые так же устанавливают новые значения свойств, следуют тому же шаблону. Аналогичные действия выполняют слоты **start()** и **stop()**, потому что они изменяют свойство **running**.
Осталась еще одна функция класса **AxBouncer**, которая представляет для нас интерес:

```
QAxAggregated *AxBouncer::createAggregate()
{
  return new ObjectSafetyImpl;
}
```

Функция **createAggregate()** перекрывает метод предка -- **QAxBindable**. Она позволяет реализовать COM-интерфейс(ы), которые не реализованы в модуле **QAxServer**, или обойти COM-интерфейсы по-умолчанию. В данном случае возвращается интерфейс **IObjectSafety**, который используется в Internet Explorer, для безопасного доступа к элементам компонента. Это обычный трюк, помогающий избежать сообщения об ошибке: "Object not safe for scripting".

Ниже приводится определение класса, который реализует интерфейс **IObjectSafety**:

```
class ObjectSafetyImpl : public QAxAggregated, public IObjectSafety
{
public:
  long queryInterface(const QUuid &iid, void **iface);

  QAXAGG_IUNKNOWN

  HRESULT WINAPI GetInterfaceSafetyOptions(REFIID riid,
          DWORD *pdwSupportedOptions, DWORD *pdwEnabledOptions);
  HRESULT WINAPI SetInterfaceSafetyOptions(REFIID riid,
          DWORD pdwSupportedOptions, DWORD pdwEnabledOptions);
};
```

Кдасс **ObjectSafetyImpl** порожден от **QAxAggregated** и **IObjectSafety**. Класс **QAxAggregated** -- это абстрактный класс, использующийся в качестве базового, для реализации дополнительных COM-интерфейсов. Доступ к COM-объекту, который является расширением **QAxAggregated**, может быть получен вызовом функции **controllingUnknown()**. Этот COM-объект негласно создается самим модулем **QAxServer**.
Макрос **QAXAGG_IUNKNOWN** подставляет стандартную реализацию функций **QueryInterface()**, **AddRef()** и **Release()**.

```
long ObjectSafetyImpl::queryInterface(const QUuid &iid, void **iface)
{
  *iface = 0;
  if (iid == IID_IObjectSafety)
    *iface = (IObjectSafety *)this;
  else
    return E_NOINTERFACE;

  AddRef();
  return S_OK;
}
```

Функция **queryInterface()** вызывается программой-клиентом, управляющей COM-объектом, для того, чтобы получить доступ к интерфейсу, реализуемому классом-наследником от **QAxAggregated**. Для интерфейсов, реализация которых отсутствует, следует возвращать значение **E_NOINTERFACE**.

```
HRESULT WINAPI ObjectSafetyImpl::GetInterfaceSafetyOptions(
            REFIID riid, DWORD *pdwSupportedOptions,
            DWORD *pdwEnabledOptions)
{
  *pdwSupportedOptions = INTERFACESAFE_FOR_UNTRUSTED_DATA
                         | INTERFACESAFE_FOR_UNTRUSTED_CALLER;
  *pdwEnabledOptions = *pdwSupportedOptions;
  return S_OK;
}

HRESULT WINAPI ObjectSafetyImpl::SetInterfaceSafetyOptions(REFIID,
            DWORD, DWORD)
{
  return S_OK;
}
```

Функции **GetInterfaceSafetyOptions()** и **SetInterfaceSafetyOptions()** объявлены в **IObjectSafety**. Мы выполняем их реализацию для того, чтобы сообщить вызывающей программе о том, что объект безопасен.

Теперь перейдем к содержимому файла **main.cpp**:

```
#include <qaxfactory.h>

#include "axbouncer.h"

QAXFACTORY_DEFAULT(AxBouncer,
                   "{5e2461aa-a3e8-4f7a-8b04-307459a4c08c}",
                   "{533af11f-4899-43de-8b7f-2ddf588d1015}",
                   "{772c14a5-a840-4023-b79d-19549ece0cd9}",
                   "{dbce1e56-70dd-4f74-85e0-95c65d86254d}",
                   "{3f3db5e0-78ff-4e35-8a5d-3d3b96c83e09}")

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

Макрос **QAXFACTORY_DEFAULT()** экспортирует компонент ActiveX. Этот макрос используется в случаях, когда сервер экспортирует единственный элемент управления ActiveX. В противном случае, сервер должен перекрыть класс **QAxFactory** и использовать макрос **QAXFACTORY_EXPORT()**.

Следующий пример в этом разделе продемонстрирует, как это делается.

Первый аргумент **QAXFACTORY_DEFAULT()** -- имя экспортируемого класса Qt. Остальные пять аргументов -- это GUID класса, GUID интерфейса, GUID интерфейса событий, GUID библиотеки типов и GUID приложения. Для генерации этих значений могут использоваться стандартные утилиты **guidgen** и **uuidgen**.
Поскольку наш сервер является библиотекой, мы не предусматриваем никаких действий в функции **main()**. Однако она нужна нам, чтобы "умиротворить" программу-компоновщик (linker).
Файл проекта для **.pro** нашего сервера:

```
TEMPLATE     = lib
CONFIG      += activeqt dll
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** могут быть скопированы из подкаталога библиотеки Qt: **extensions\activeqt\control**.
Файл **makefile** или файл проекта Visual C++, созданные **qmake**, уже содержат правила регистрации сервера в реестре Windows. Чтобы зарегистрировать сервер на другой машине, можно воспользоваться утилитой **regsvr32**, которая входит в состав ОС Windows.
После этого мы можем включить компонент Bouncer в HTML-страничку, с помощью тега <**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>
```

А для управления объектом -- разместить кнопки:

```
<input type="button" value="Start" onClick="AxBouncer.start()">
<input type="button" value="Stop" onClick="AxBouncer.stop()">
```

Этим объектом можно управлять из JavaScript или VBScript точно так же, как и любым другим элементом управления ActiveX.

В нашем последнем примере мы рассмотрим реализацию приложения **Address Book**. Это приложение может использоваться как обычная Qt/Windows программа или как "внешний" ActiveX сервер, например для программ, написанных на Visual Basic.

```
class AddressBook : public QMainWindow
{
  Q_OBJECT
  Q_PROPERTY(int count READ count)
public:
  AddressBook(QWidget *parent = 0, const char *name = 0);
  ~AddressBook();

  int count() const;

public slots:
  ABItem *createEntry(const QString &contact);
  ABItem *findEntry(const QString &contact) const;
  ABItem *entryAt(int index) const;
  ...
};
```

Класс **AddressBook** -- это виджет главного окна приложения.

```
class ABItem : public QObject, public QListViewItem
{
  Q_OBJECT
  Q_PROPERTY(QString contact READ contact WRITE setContact)
  Q_PROPERTY(QString address READ address WRITE setAddress)
  Q_PROPERTY(QString phoneNumber READ phoneNumber
             WRITE setPhoneNumber)
public:
  ABItem(QListView *listView);

  void setContact(const QString &contact);
  QString contact() const { return text(0); }
  void setAddress(const QString &address);
  QString address() const { return text(1); }
  void setPhoneNumber(const QString &number);
  QString phoneNumber() const { return text(2); }

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

Класс **ABItem** представляет одну запись в адресной книге. Он является производным от класса **QListViewItem**, что позволяет отображать его в **QListView**. А поскольку в число его предков входит класс **QObject**, то это позволяет экспортировать его как COM-объект.

```
int main(int argc, char *argv[])
{
  QApplication app(argc, argv);
  if (!QAxFactory::isServer()) {
    AddressBook addressBook;
    app.setMainWidget(&addressBook);
    addressBook.show();
    return app.exec();
  }
  return app.exec();
}
```

В функции **main()** выполняется проверка -- запущено ли приложение как автономная программа, или как сервер. Если программа запускается как автономное приложение, то создается главный виджет и дальше все идет как в обычных Qt-приложениях. Чтобы запустить программу как сервер, нужно передать ей ключ командной строки: **-activex**.
В дополнение к ключу **-activex**, серверы ActiveX могут принимать следующие ключи:

- **-regserver** -- регистрирует сервер в реестре Windows.
- **-unregserver** -- удаляет регистрацию сервера из реестра Windows.

- **-dumpidl file** -- записывает IDL сервера в указанный файл.

Если приложение будет использоваться как сервер, необходимо экспортировать классы **AddressBook** и **ABItem** как COM-компоненты:

```
QAXFACTORY_EXPORT(ABFactory,
                  "{2b2b6f3e-86cf-4c49-9df5-80483b47f17b}",
                  "{8e827b25-148b-4307-ba7d-23f275244818}")
```

Макрос **QAXFACTORY_EXPORT()** экспортирует фабрику COM-объектов. Поскольку наше приложение экспортирует два COM-объекта, то мы уже не можем воспользоваться макросом **QAXFACTORY_DEFAULT()**, как это было сделано в предыдущем примере.

Первый аргумент макроса **QAXFACTORY_EXPORT()** -- имя класса-наследника **QAxFactory**, который реализует COM-объект. Другие два аргумента -- это GUID библиотеки типов и приложения.

```
class ABFactory : public QAxFactory {
public:
  ABFactory(const QUuid &lib, const QUuid &app);
  QStringList featureList() const;
  QWidget *create(const QString &key, QWidget *parent,
                  const char *name);
  QUuid classID(const QString &key) const;
  QUuid interfaceID(const QString &key) const;
  QUuid eventsID(const QString &key) const;
  QString exposeToSuperClass(const QString &key) const;
};
```

Класс **ABFactory** порожден от класса **QAxFactory**, он реализует виртуальную функцию, экспортирующую класс **AddressBook**, как элемент управления ActiveX, и класс **ABItem**, как COM-объект.

```
ABFactory::ABFactory(const QUuid &lib, const QUuid &app)
    : QAxFactory(lib, app)
{
}
```

Конструктор класса **ABFactory** просто передает два аргумента конструктору родительского класса.

```
QStringList ABFactory::featureList() const
{
  return QStringList() << "AddressBook" << "ABItem";
}
```

Функция **featureList()** возвращает список COM-объектов, которые могут быть созданы фабрикой.

```
QWidget *ABFactory::create(const QString &key, QWidget *parent,
                           const char *name)
{
  if (key == "AddressBook")
    return new AddressBook(parent, name);
  else
    return 0;
}
```

Функция **create()** создает экземпляр элемента управления ActiveX. Для случая **ABItem** возвращается пустой указатель, поскольку пользователь не должен иметь возможность создавать объекты этого типа.

```
QUuid ABFactory::classID(const QString &key) const
{
  if (key == "AddressBook")
    return QUuid("{588141ef-110d-4beb-95ab-ee6a478b576d}");
  else if (key == "ABItem")
    return QUuid("{bc82730e-5f39-4e5c-96be-461c2cd0d282}");
  else
```

```
    return QUuid();
}
```

Функция **classId()** возвращает идентификаторы классов, которые могут быть экспортированы фабрикой.

```
QUuid ABFactory::interfaceID(const QString &key) const
{
  if (key == "AddressBook")
    return QUuid("{718780ec-b30c-4d88-83b3-79b3d9e78502}");
  else if (key == "ABItem")
    return QUuid("{c8bc1656-870e-48a9-9937-fbe1ceff8b2e}");
  else
    return QUuid();
}
```

Функция **interfaceId()** возвращает идентификаторы интерфейсов классов, экспортируемых фабрикой.

```
QUuid ABFactory::eventsID(const QString &key) const
{
  if (key == "AddressBook")
    return QUuid("{0a06546f-9f02-4f14-a269-d6d56ffeb861}");
  else if (key == "ABItem")
    return QUuid("{105c6b0a-3fc7-460b-ae59-746d9d4b1724}");
  else
    return QUuid();
}
```

Функция **eventsId()** возвращает идентификаторы интерфейсов событий, для экспортируемых классов.

```
QString ABFactory::exposeToSuperClass(const QString &key) const
{
  return key;
}
```

По-умолчанию, элементы управления ActiveX поставляют клиентам не только свои собственные свойства, сигналы и слоты, но и свойства, сигналы и слоты своих базовых классов, вплоть до **QWidget**. Мы можем перекрыть метод **exposeToSuperClass()**, чтобы ограничить верхнюю границу (в дереве наследования) поставляемых классов.
Здесь, в виде верхней границы, мы просто возвращаем имя класса компонента ("**AddressBook**" или "**ABItem**"). Это означает, что свойства, сигналы и слоты классов-предков для **AddressBook** и **ABItem** поставляться не будут.
Ниже приводится содержимое файла проекта для нашего "внешнего" сервера ActiveX:

```
CONFIG      += activeqt
HEADERS      = abfactory.h \
               abitem.h \
               addressbook.h \
               editdialog.h
SOURCES      = abfactory.cpp \
               abitem.cpp \
               addressbook.cpp \
               editdialog.cpp \
               main.cpp
RC_FILE      = qaxserver.rc
```

Файл **qaxserver.rc** может быть скопирован из подкаталога **extensions\activeqt\control**.
На этом мы завершаем краткий обзор ActiveQt framework. Дистрибутив библиотеки Qt включает в себя ряд дополнительных примеров, которые содержат сведения о модулях **QAxContainer** и **QAxServer** и решения наиболее общих проблем совместимости.

18.3 Управление сеансами

В момент завершения сеанса X11, некоторые оконные менеджеры выдают запрос на подтверждение завершения сеанса. Если мы подтверждаем завершение сессии, то приложения, которые работали в

этот момент, будут автоматически запущены в начале следующего сеанса, с теми же экранными координатами и, в идеале, в том же самом состоянии.

Компонент X11, который управляет сохранением и восстановлением сеанса называется менеджер сеанса (или, если хотите, менеджер сессии). Чтобы добавить в Qt-приложение возможность сохранения своего состояния, в момент завершения сессии, необходимо перекрыть метод **QApplication::saveState()**, в котором выполнять сохранение всех необходимых параметров.

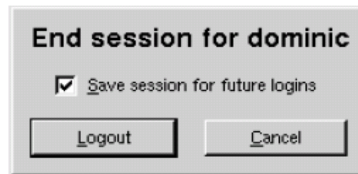


Рисунок 18.6. Окно запроса подтверждения завершения сеанса в KDE.

Операционные системы Windows 2000/XP (и некоторые Unix-системы) предлагают иной механизм сохранения сеансов, который носит название гибернация (**hibernation**). Когда пользователь переводит систему в режим гибернации, то она просто скидывает дамп памяти на диск и на следующем запуске загружает его. В этом случае приложениям ничего не надо делать, им даже нет необходимости что либо "знать" о гибернации.

Момент завершения работы может быть перехвачен приложением, для этого надо перекрыть метод **QApplication::commitData()**. Это позволит сохранить любые несохраненные данные и запросить что либо у пользователя, если в этом возникнет необходимость. Это поведение реализуется одинаково на обеих платформах: X11 и Windows.
Мы исследуем поведение приложения, которое может взаимодействовать с менеджером сеанса, на примере программы "Крестики-нолики". Сначала рассмотрим функцию **main()**:

```
int main(int argc, char *argv[])
{
  Application app(argc, argv);
  TicTacToe tic(0, "tic");
  app.setTicTacToe(&tic);
  tic.show();
  return app.exec();
}
```

Здесь создается экземпляр класса **Application**, производный от класса **QApplication**. Этот класс перекрывает методы предка -- **commitData()** и **saveState()**.
Затем создается виджет **TicTacToe** и выводится на экран. Виджету **TicTacToe** было присвоено имя "**tic**". Если вы хотите обеспечить взаимодействие программы с менеджером сеанса, то всем виджетам верхнего уровня должны быть присвоены уникальные имена.

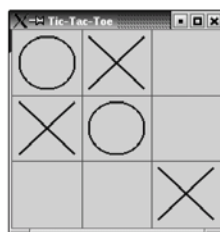


Рисунок 18.7. Внешний вид приложения "Крестики-нолики".

Ниже приводится определение класса **Application**:

```
class Application : public QApplication
{
  Q_OBJECT
public:
  Application(int &argc, char *argv[]);
```

```
  void setTicTacToe(TicTacToe *tic);
  void commitData(QSessionManager &sessionManager);
  void saveState(QSessionManager &sessionManager);

private:
  TicTacToe *ticTacToe;
};
```

Класс **Application** хранит указатель на виджет **TicTacToe** в приватной переменной.

```
void Application::saveState(QSessionManager &sessionManager)
{
  QString fileName = ticTacToe->saveState();

  QStringList discardCommand;
  discardCommand << "rm" << fileName;
  sessionManager.setDiscardCommand(discardCommand);
}
```

На платформе X11, менеджер сеансов вызывает функцию **saveState()**, для сохранения состояния приложения. Она доступна и на других платформах, но никогда не вызывается. Аргумент типа **QSessionManager** позволяет взаимодействовать с менеджером сеансов.
Функция начинается с сохранения состояния виджета **TicTacToe** в файл. Затем менеджеру сеанса передается команда удаления. Команда удаления -- это команда, которая будет использована менеджером сеанса для удаления любой информации, имеющей отношение к текущему состоянию приложения. В данном случае команда выглядит как:

```
rm file
```

где **file** -- это имя файла, в котором сохраняется информация о текущем состоянии, а **rm** -- это стандартная команда Unix, выполняющая удаление файлов.
Менеджеру сеанса может быть передана команда восстановления, которая будет выполнена менеджером для перезапуска приложения. По-умолчанию, Qt устанавливает команду восстановления:

```
appname -session id_key
```

где **appname** берется из **argv[0]**, **id** -- идентификатор сессии, поставляемый самим менеджером. Менеджер сеансов гарантирует уникальность идентификатора для каждого экземпляра приложения. И наконец **key** -- дополнительная информация, содержащая время сохранения состояния приложения. По различным причинам, функция **saveState()** может вызываться несколько раз, на протяжении одной сессии, таким образом пара **id** и **key** гарантируют уникальность каждого из сохраненных состояний.

Из-за ограничений, существующих в менеджерах сеансов, путь к исполняемому файлу приложения должен быть прописан в переменной **PATH**. В данном конкретном случае, если вы пожелаете испытать приложение "Крестики-нолики", вы должны переписать исполняемый файл программы в каталог, скажем, **/usr/bin**, и запустить ее командой **tictactoe**.
Для простых приложений, таких как "Крестики-нолики", состояние может быть сохранено в виде аргумента командной строки, которая перезапускает приложение в начале следующей сессии, например:

```
tictactoe -state OX-XO-X-O
```

В этом случае отпадает необходимость сохранения информации в отдельный файл и установки команды удаления файла.

```
void Application::commitData(QSessionManager &sessionManager)
{
  if (ticTacToe->gameInProgress()
          && sessionManager.allowsInteraction()) {
    int ret = QMessageBox::warning(ticTacToe, tr("Tic-Tac-Toe"),
                tr("The game hasn t finished.\n"
                   "Do you really want to quit?"),
```

```
                QMessageBox::Yes | QMessageBox::Default,
                QMessageBox::No | QMessageBox::Escape);
    if (ret == QMessageBox::Yes)
      sessionManager.release();
    else
      sessionManager.cancel();
  }
}
```

Функция **commitData()** вызывается в момент завершения сеанса. Здесь выводится запрос на подтверждение завершения приложения, чтобы предотвратить потерю данных. По-умолчанию она закрывает все виджеты верхнего уровня, точно так же, как и в случае завершения приложения нажатием на кнопку "**X**", в заголовке окна. В Главе 3 мы уже демонстрировали, как перекрыть метод **closeEvent()** и вывести запрос на подтверждение.

В данном примере, мы перекрыли метод **commitData()** и выводим запрос на подтверждение из него, если менеджер сеанса позволяет это сделать. Когда пользователь щелкает по кнопке **Yes**, то вызывается функция **release()**, которая сообщает менеджеру сеанса о том, что он может продолжить процедуру завершения сессии. В противном случае процедура завершения сеанса будет остановлена, вызовом метода **cancel()**.

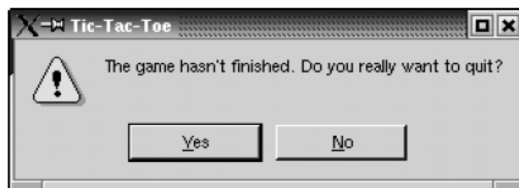


Рисунок 18.8. Запрос на подтверждение завершения работы программы.

Теперь перейдем к рассмотрению класса **TicTacToe**:

```
class TicTacToe : public QWidget
{
  Q_OBJECT
public:
  TicTacToe(QWidget *parent = 0, const char *name = 0);

  QSize sizeHint() const;
  bool gameInProgress() const;
  QString saveState() const;

protected:
  void paintEvent(QPaintEvent *event);
  void mousePressEvent(QMouseEvent *event);

private:
  enum { Empty = '-', Cross = 'X', Nought = 'O' };
  void clearBoard();
  void restoreState();
  QString sessionFileName() const;
  QRect cellRect(int row, int col) const;
  int cellWidth() const { return width() / 3; }
  int cellHeight() const { return height() / 3; }

  char board[3][3];
  int turnNumber;
};
```

Класс **TicTacToe** порожден от **QWidget** и перекрывает методы предка: **sizeHint()**, **paintEvent()** и **mousePressEvent()**. Он так же реализует новые методы: **gameInProgress()** и **saveState()**, которые используются классом **Application**.

```
TicTacToe::TicTacToe(QWidget *parent, const char *name)
    : QWidget(parent, name)
{
  setCaption(tr("Tic-Tac-Toe"));
```

```
  clearBoard();
  if (qApp->isSessionRestored())
    restoreState();
}
```

В конструкторе производится очистка игрового поля и, если приложение вызвано с ключом **-session**, то вызывается приватная функция **restoreState()**.

```
void TicTacToe::clearBoard()
{
  for (int row = 0; row < 3; ++row) {
    for (int col = 0; col < 3; ++col) {
      board[row][col] = Empty;
    }
  }
  turnNumber = 0;
}
```

Функция **clearBoard()** выполняет очистку ячеек игрового поля и записывает значение 0 в переменную **turnNumber** (номер хода).

```
QString TicTacToe::saveState() const
{
  QFile file(sessionFileName());
  if (file.open(IO_WriteOnly)) {
    QTextStream out(&file);
    for (int row = 0; row < 3; ++row) {
      for (int col = 0; col < 3; ++col) {
        out << board[row][col];
      }
    }
  }
  return file.name();
}
```

В функции **saveState()** производится сохранение состояния игрового поля в файл. Формат файла очень прост -- на место крестика записывается символ '**X**', на место нолика -- '**O**' и на место пустой ячейки -- '**-**'.

```
QString TicTacToe::sessionFileName() const
{
  return QDir::homeDirPath() + "/.tictactoe_"
         + qApp->sessionId() + "_" + qApp->sessionKey();
}
```

Функция **sessionFileName()** возвращает имя файла, которое соответствует текущему идентификатору и ключу сеанса. Эта функция вызывается как из **saveState()**, так и из **restoreState()**.

```
void TicTacToe::restoreState()
{
  QFile file(sessionFileName());
  if (file.open(IO_ReadOnly)) {
    QTextStream in(&file);
    for (int row = 0; row < 3; ++row) {
      for (int col = 0; col < 3; ++col) {
        in >> board[row][col];
        if (board[row][col] != Empty)
          ++turnNumber;
      }
    }
  }
  repaint();
}
```

Функция **restoreState()** загружвет файл, в котором было сохранено предыдущее состояние приложения и заполняет игровое поле. Номер хода рассчитывается как сумма крестиков и ноликов на игровом поле.

Функция **restoreState()** вызывается в конструкторе класса **TicTacToe**, если **QApplication::isSessionRestored()** возвращает **true**. В этом случае, функции **sessionId()** и **sessionKey()** возвращают те же значения, с которыми было сохранено предыдущее состояние приложения. Отсюда и **sessionFileName()** вернет имя файла, соответствующее этой сессии.

Отладка взаимодействия с менеджером сеанса может оказаться занятием нудным и трудоемким, поскольку придется неоднократно перезапускать сессию. К счастью, в состав X11 входит утилита **xsm**. На запуске, эта утилита откроет окно менеджера сеанса и терминал. Приложения, запускаемые из терминала, будут использовать **xsm**, в качестве менеджера сеанса. После этого мы можем завершать и перезапускать сессии и следить за поведением отлаживаемого приложения. Дополнительные сведения по этой теме вы найдете по адресу: *http://doc.trolltech.com/3.2/session.html.*

ОБ АВТОРАХ

Jasmin Blanchette
Жасмен получил высшее образование в области информатики в 2001 году, в Университете города Шербрук, штат Квебек, Канада, и был награжден медалью Фернанда Сегуина (Fernand Seguin) за успехи в обучении. Летом 2000 года он поступил в компанию Trolltech, на стажировку, на должность инженера-программиста и затем, в начале 2001 года, был принят на постоянную работу. Теперь он работает ведущим инженером-программистом. Он ведет проект Qt Linguist и отвечает за ежеквартальные выпуски Qt Quarterly -- технический информационный бюллетень компании Trolltech. В свободное время увлекается написанием романов на норвежском и шведском языках. Живет в Осло со своей подругой Энни Лин (Anne-Lene).

Mark Summerfield
Марк получил высшее образование в области информатики в 1993 году, в Уэльсском Университете. Там же он закончил аспирантуру. Прежде чем перейти на работу в Trolltech, он много лет, проработал инженером-программистом в различных фирмах. На протяжении последних нескольких лет работает управляющим отдела документации, в его багаже более 1500 страниц справочной документации к библиотек Qt и материалов в ежеквартальнике Qt Quarterly. В свободное время он занимается созданием свободно распространяемых программ. Живет в Южном Уэльсе, Великобритания, с супругой Андреа (Andrea).

ПРИМЕЧАНИЯ

[1] К настоящему моменту Реджинальд уже переехал в Германию, где стал одним из соучредителей компании froglogic, занимающейся оказанием консалтинговых услуг в сфере программного обеспечения.
[2] Несколько лет тому назад, Арнт оставил компанию и продолжил свою карьеру в Германии.
[3] Сигналы, в терминологии Qt -- совершенно не то же самое, что в терминологии Unix. На протяжении всей книги мы будем говорить исключительно о Qt-сигналах.
[4] Если во время компиляции возникла ошибка в строке 8, то это означает, что у вас установлена более старая версия Qt, чем требуется. Напоминаем еще раз, что для прогона примеров из этой книги вы должны установить Qt 3.2 или более поздний выпуск Qt 3.
[5] Qt предоставляет в распоряжение программиста два макроопределения: TRUE и FALSE, которые могут использоваться взамен стандартных true и false. Не смотря на это, мы не видим достаточных причин, чтобы использовать макросы Qt, разве только в том случае, когда компилятор не поддерживает ключевые слова true и false.
[6] Если вы проживаете в стране, чье законодательство признает патенты на программное обеспечение и где компания Unisys имеет зарегистрированный патент на алгоритм сжатия LZW, то Unisys может потребовать от вас приобрести лицензию на использование GIF. По этой причине, поддержка формата GIF в Qt по-умолчанию запрещена. Мы полагаем, что срок действия этого патента по всему миру истечет к концу 2004 года.
[7] От переводчика: кроме вышеперечисленных, Qt 3.2 включает в себя еще один драйвер, который уважаемые авторы, видимо по забывчивости, не указали -- это QIBASE (Interbase/Firebird).
[8] Вероятно, в состав Qt 3.3 будет включен класс QLocale, который будет обслуживать представление числовых форматов.
[9] Возможно в Qt 3.3 будет включена возможность достигать этого эффекта независимо от типа операционной системы.