Николай Секунов

Программирование на C++ в Linux

- Принципы создания приложений в интегрированной среде KDE
- Программирование в среде KDevelop
- Разработка пользовательского интерфейса
- Реализация многозадачности
- Расширенные возможности языка C++

Николай Секунов

Программирование на C++ в Linux

Санкт-Петербург
«БХВ-Петербург»
2004

УДК 681.3.068+800.92С++
ББК 32.973.26-018.1
С28

Секунов Н. Ю.

С28 Программирование на С++ в Linux. — СПб.: БХВ-Петербург, 2004. — 368 с.: ил.

ISBN 5-94157-355-3

Книга посвящена созданию приложений, написанных на языке С++, в среде разработки KDevelop. Дано описание способов взаимодействия компонентов приложений. Рассмотрена работа с утилитой Qt Designer и описаны основные элементы управления, используемые в диалоговых окнах, а также классы, созданные для работы с ними. Читатель знакомится с концепцией Документ/Представление и учится создавать элементы пользовательского интерфейса приложения. Кроме того, в отдельных главах разбираются вопросы вывода на экран различной информации, сохранения и восстановления ее из файла, создания текстовых редакторов, работы с шаблонами классов и функций и организации многозадачности в приложении на основе взаимодействуюших процессов. В завершение предоставляются рекомендации по созданию справочной системы приложения.

Для программистов

УДК 681.3.068+800.92С++
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Анатолий Адаменко*
Зав. редакцией	*Григорий Добин*
Редактор	*Наталья Сержантова*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Оформление серии	*Via Design*
Дизайн обложки	*Игоря Цырульникова*
Зав. производством	*Николай Тверских*

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 25.02.04.
Формат 70×100¹/₁₆. Печать офсетная. Усл. печ. л. 29,67.
Доп. тираж 4000 экз. Заказ № 113
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Гигиеническое заключение на продукцию, товар № 77.99.02.953.Д.001537.03.02 от 13.03.2002 г. выдано Департаментом ГСЭН Минздрава России.

Отпечатано с готовых диапозитивов
в Академической типографии "Наука" РАН
199034, Санкт-Петербург, 9 линия, 12.

ISBN 5-94157-355-3

© Секунов Н. Ю., 2003
© Оформление, издательство "БХВ-Петербург", 2003

Содержание

Введение 7

Для кого предназначена эта книга? 9
Структура книги 10
Соглашения, принятые в данной книге 13
Требования к аппаратным средствам и программному обеспечению 14

Глава 1. Взаимодействие компонентов приложения 17

Сигналы и приемники 17
    Посылка сигналов 18
    Реализация приемников 20
    Реализация соединения 21
Обработка событий 22
    Работа с окном 25
    Работа с фокусом ввода 26
    Работа с мышью 28
    Работа с клавиатурой 30
    Реализация перетаскивания 32
    Фильтры событий 33
    Синтетические события 34
    Последовательность обработки событий 36
Заключение 36

Глава 2. Диалоговые окна и простейшие элементы управления 38

Создание диалогового приложения 38
    Создание заготовки приложения 39
    Создание заготовки диалогового окна 40
    Завершение создания диалогового приложения 57
Создание специализированных диалоговых окон 66
    Создание диалогового окна с вкладками 66
Создание мастера 77

Глава 3. Классы элементов управления 88

Класс списка 88
Классы линейного регулятора и линейного индикатора 97
Работа с датой и временем 104

Глава 4. Классы приложений, документов и представлений 114

Многооконное приложение Qt 115
 Класс документа 117
 Класс представления 123
 Класс приложения 126
Многооконное приложение KDE 136
 Класс документа 137
 Класс представления 139
 Класс приложения 140

Глава 5. Создание элементов пользовательского интерфейса 146

Пользовательский интерфейс библиотеки Qt 147
 Внесение изменений в меню 147
 Настройка панели инструментов 152
 Работа со строкой состояния 162
Пользовательский интерфейс приложений KDE 168
 Внесение изменений в меню 169
 Настройка панели инструментов 174
 Работа со строкой состояния 183

Глава 6. Вывод информации на экран 187

Рисование фигур 187
Работа с кистью 193
 Перерисовка окна 197
 Синхронизация объектов представления 201
Вывод текста 203
Работа с битовыми образами 206
 Аппаратно-зависимые битовые образы 206
 Аппаратно-независимые битовые образы 211

Глава 7. Работа с файлами документов 214

Сохранение и восстановление информации в приложении 214
Настройка диалоговых окон 221
Внесение изменений в меню 225
Установка рабочего каталога 229

Глава 8. Работа с текстовыми документами 232

Создание простейшего текстового редактора 232
Создание более сложного редактора 235
Создание редактора KDE 250

Глава 9. Шаблоны и классы коллекций ... 270

Шаблоны ... 270
Понятие шаблона ... 271
Шаблоны функций ... 273
Шаблоны классов ... 275
Классы коллекций ... 278
Виды классов коллекций ... 278
Массивы ... 281
Связные списки ... 284
Карты отображений ... 290
Другие классы коллекций ... 294

Глава 10. Реализация многозадачности в приложении ... 299

Взаимодействие процессов ... 300
Создание клиента для простейшего сервера ... 300
Создание более сложного сервера ... 308
Создание клиента ... 314
Некоторые замечания ... 324

Глава 11. Справка в приложении ... 326

Формы представления справочной информации ... 327
Способы доступа к справочной системе ... 327
Способы представления справочной информации ... 328
Формы представления информации ... 330
Программирование контекстной справки ... 330
Вывод подсказок ... 331
Вывод справочной информации в строку состояния ... 332
Получение информации по конкретному элементу пользовательского интерфейса ... 333
Программирование командной справки ... 334
Формат файлов командной справки приложений Qt ... 335
Создание демонстрационного приложения Qt ... 340

Приложение 1. Что на CD ... 355

Приложение 2. Ресурсы Интернета ... 357

Предметный указатель ... 358

Введение

В настоящее время все большую популярность получает использование свободно распространяемого в исходных текстах программного обеспечения. Причем эта тенденция наблюдается по отношению не только к приложениям, используемым для домашних нужд, но и к полноценным коммерческим приложениям. Одним из наиболее успешных проектов по созданию подобного программного обеспечения (наряду с веб-сервером Apache, языком Perl и набором утилит GNU) является операционная система Linux — полностью 32-разрядная, защищенная, многоплатформенная, многопользовательская и многозадачная UNIX-подобная операционная система. В настоящее время она является одной из основных операционных систем для серверов Интернет и серверов локальных сетей, но в последние годы явно наметилась тенденция ее использования и на домашних компьютерах.

Основным недостатком UNIX-подобных операционных систем до недавнего времени являлась сложность их использования, связанная с необходимостью работы в командной строке. В настоящее время этот недостаток успешно исправляется и работа в интегрированных средах Linux, таких как KDE, уже мало чем отличается от работы в Windows. Однако набор программных средств этих оболочек пока еще уступает по качеству и разнообразию набору программных средств Windows.

Многие программы Linux создаются программистами в свободное время. Можно смело утверждать, что число появляющихся на рынке новых приложений Linux напрямую зависит от качества и удобства используемой ими среды программирования, поскольку в свое свободное время человек хочет получать удовольствие от того, чем он занимается. Поэтому многие программисты отдавали свое предпочтение языку Kylix, являющемуся потомком языка Pascal, созданного в середине 70-х годов для начального обучения школьников программированию.

Ясно, что, имея такое наследство, этот язык не может обеспечивать высокой эффективности создаваемых на нем приложений. По собственному опыту

могу сказать, что программа, написанная на языке Pascal, работает в 4 раза медленнее, чем та же программа, написанная на языке C++. Причем сравнение программ производилось в среде Windows. В среде Linux выигрыш должен быть намного большим, поскольку язык С является внутренним языком операционной системы, и все ее функции оптимизированы для данного языка.

Одним из наиболее распространенных инструментариев разработки приложений на языке C++ для интегрированной среды KDE является KDevelop — полноценная среда разработки, объединяющая в себе все необходимые средства для создания и отладки приложений:

- встроенный транслятор и компилятор языка C++;
- мастер создания приложений, позволяющий создавать работоспособные заготовки различных типов приложений;
- мастер создания классов, позволяющий создавать заготовки новых классов и включать их в существующие приложения;
- систему управления файлами проекта, упрощающую работу с файлами заголовков и реализации классов;
- систему создания документации по приложению с использованием языка SGML, включающую в себя автоматическую трансляцию созданных файлов в формат HTML;
- систему автоматического создания документации по классам и функциям приложения в формате HTML;
- поддержку локализации создаваемых приложений, существенно облегчающую создание версии приложения для нового языка;
- использование для создания пользовательского интерфейса приложения утилиты Qt Designer фирмы TrollTech.
- поддержку совместной работы нескольких разработчиков над одним проектом;
- поддержку, помимо своего внутреннего отладчика, отладчиков ddd и kdbg;
- использование для редактирования значков приложения утилиты KIconEdit.

Таким образом, среда разработки KDevelop объединяет в себе все необходимое для создания приложения и позволяет пользователю получить всю требующуюся ему информацию.

Данная книга посвящена созданию приложений, написанных на языке C++, в среде разработки KDevelop. В ней предпринята попытка описания как можно более широкого круга приложений. Поэтому каждое из описываемых типов приложений рассмотрено только в общих чертах, что, впро-

чем, создает хорошую базу для их последующего более детального изучения по другим источникам.

Конечно, в предлагаемой книге освещены далеко не все описанные выше возможности среды разработки, но представленного в ней материала достаточно для написания весьма сложных приложений и довольно полного ознакомления с данной средой.

Для кого предназначена эта книга?

Потенциальными читателями этой книги являются как уже достаточно квалифицированные программисты, имеющие большой опыт работы на языке C++ в среде Windows, так и начинающие. Для тех, кто имеет опыт создания приложений в среде разработки Microsoft Visual Studio, в книгу включены комментарии, демонстрирующие основные отличия KDevelop от этой среды, что существенно облегчит им изучение KDevelop. Начинающие программисты найдут в данной книге исчерпывающий материал по созданию основных типов приложений в среде Linux и организации их взаимодействия.

Поскольку операционная система Linux всегда считалась прибежищем наиболее продвинутых программистов, среди ее пользователей сформировалось мнение, что чем сложнее процесс создания программы, тем круче создавший ее программист. Поэтому некоторые разработчики приложений Linux намеренно отказывались от использования редакторов ресурсов и создавали свои диалоговые окна сразу в исходных текстах программ. Редактирование программ они производили в обычных текстовых редакторах, например в Emacs, а для трансляции и компоновки приложения использовали пакетные файлы.

Конечно, разработка программ указанным выше способом под силу далеко не каждому программисту. Дело в том, что даже очень опытный программист не может создать подобным образом достаточно сложного приложения, или на его создание уйдет столько времени, что на рынке появится уже следующее поколение аналогичных программ. Поэтому даже квалифицированным программистам, придерживающимся подобных взглядов, следует обратить внимание на среду разработки KDevelop и на предоставляемые ею возможности по автоматизации программирования. Тогда они смогут направить свои усилия не на создание простых приложений сложными методами, а на создание сложных приложений простыми методами.

Каждый раздел данной книги иллюстрируется несколькими демонстрационными приложениями, позволяющими читателю на примере работающей программы самостоятельно изучить особенности рассматриваемого раздела программирования в среде KDevelop. Тексты этих примеров будут размещены на прилагаемом к книге CD.

В предлагаемой книге не только описана методика создания различных компонентов приложения, но и указано, какие сложности при этом могут возникнуть, можно ли их обойти и как это сделать. При изложении материала соблюдалась максимальная объективность.

Структура книги

В данной книге рассмотрены принципы создания основных типов приложений, работающих в интегрированной среде KDE. В отличие от Windows, где, по крайней мере, в простейших приложениях разработчик избавлен от необходимости следить за взаимодействием их компонентов, в Linux он должен с самого начала разбираться в способах взаимодействия этих приложений. Поэтому книга начинается с описания способов взаимодействия компонентов приложений.

Одной из областей, где наиболее интенсивно используется взаимодействие компонентов приложения, являются диалоговые окна. Поэтому сразу же после описания способов взаимодействия компонентов их использование демонстрируется на примере специального вида приложений, называемых диалоговыми приложениями. На примере этих приложений рассмотрена работа с утилитой Qt Designer и описаны основные элементы управления, используемые в диалоговых окнах, а также классы, созданные для работы с ними.

Диалоговые приложения являются особым видом приложений, позволяющим с минимальными затратами создать оболочку для работы какого-либо алгоритма. Для создания более сложных приложений разработчику необходимо ознакомиться с концепцией Документ/Представление и научиться создавать элементы пользовательского интерфейса приложения.

После ознакомления с этими вопросами читатель переходит к рассмотрению основных видов приложений. Он учится выводить на экран различную информацию, а также сохранять эту информацию в файле и восстанавливать ее из файла. Особая глава посвящена вопросу создания текстовых редакторов, поскольку именно они считаются основным типом приложений в большинстве операционных систем.

После получения базовых знаний по использованию среды разработки KDevelop читатель может перейти к изучению более сложных вопросов программирования в ней. В специальных главах рассмотрена работа с шаблонами классов и функций, работа с классами коллекций и организация многозадачности в приложении на основе взаимодействующих процессов. В завершение предоставляется информация по созданию справочной системы приложения.

В приведенном ниже списке разделов и глав дано краткое их описание, позволяющее пользователю лучше ориентироваться в структуре книги.

Глава 1. Взаимодействие компонентов приложения

Данная глава посвящена принципам организации взаимодействия различных компонентов приложения. В ней дано подробное описание использующихся для этого сигналов и приемников, приведены сведения о системе обработки событий в приложениях KDevelop, а также представлены использующиеся для этого функции и классы.

Глава 2. Диалоговые окна и простейшие элементы управления

В настоящей главе рассмотрено создание диалоговых окон, диалоговых окон с вкладками и мастеров с использованием приложения Qt Designer фирмы TrollTech, описаны основные элементы управления диалогового окна и использующиеся для работы с ними классы. Здесь же продемонстрировано создание диалогового приложения из заготовки приложения KDE с минимальными возможностями.

Глава 3. Классы элементов управления

В данной главе описаны достаточно сложные многофункциональные элементы управления диалогового окна, для работы с которыми используется широкий набор функций поддерживающих их классов.

Глава 4. Классы приложений, документов и представлений

В этой главе рассмотрена концепция Документ/Представление и описаны реализующие ее класс документа, класс представления и класс приложения.

Глава 5. Создание элементов пользовательского интерфейса

В данной главе рассмотрены основные типы элементов пользовательского интерфейса приложений, включая меню, панель инструментов и строку состояния. Поскольку в библиотеке Qt, в отличие от библиотеки MFC, работа с элементами пользовательского интерфейса не скрывается от разработчика, процедура создания пользовательского интерфейса в среде разработки KDevelop требует от него большего внимания, чем в среде Visual Studio.

Глава 6. Вывод информации на экран

В настоящей главе рассмотрены принципы организации графического интерфейса приложений, использующих библиотеку Qt или библиотеку интегрированной среды KDE.

Глава 7. Работа с файлами документов

В данной главе рассмотрено сохранение информации приложения в файле и извлечение ее из файла в рамках концепции Документ/Представление.

Глава 8. Работа с текстовыми документами

Операционная система UNIX, как и большинство других операционных систем, создавалась для работы преимущественно с текстовыми приложениями и оптимизирована для выполнения данной задачи. Поэтому вопросы создания текстовых редакторов вынесены в отдельную главу.

Глава 9. Шаблоны и классы коллекций

В данной главе рассматриваются расширенные возможности языка C++, реализованные в библиотеке Qt. Шаблоны представляют собой метод задания функций обработки для переменных абстрактных типов. Это позволяет минимизировать число практически идентичных по тексту функций, выполняющих одну и ту же обработку для переменных различных типов. Использование шаблонов дает возможность сократить размер исходного текста программ и обеспечивает гарантированное внесение изменений во все функции обработки различных типов данных, выполняющих над ними одну и ту же операцию. Классы коллекций можно рассматривать как практическую реализацию идеи шаблонов, позволяющую разработчику создать простую и эффективную структуру хранения данных приложения.

Глава 10. Реализация многозадачности в приложении

Операционная система UNIX является истинно многозадачной, поскольку допускает одновременное исполнение нескольких независимых процессов. Во многих случаях возникает необходимость разбиения приложения на несколько взаимодействующих процессов, в каждом из которых производится своя обработка. Разбиение приложения на процессы позволяет существенно сократить его простои, связанные с ожиданием затребованных им ресурсов, т. к. в этом случае прекращает работу только один процесс приложения, а не все приложение сразу.

Глава 11. Справка в приложении

Любое серьезное приложение, независимо от того, создается ли оно для внутреннего использования или для продажи, должно содержать обширную справочную систему, позволяющую получить информацию по любому вопросу, связанному с данным приложением. Созданию справочной системы приложения и посвящена эта глава.

Приложение 1. Что на CD

Содержит описание приложений, помещенных на прилагаемый к книге CD.

Приложение 2. Ресурсы Интернета

Содержит адреса сайтов, на которых можно получить дополнительную информацию.

Соглашения, принятые в данной книге

В этой книге использовалось специальное форматирование для выделения некоторых текстов или фрагментов текста. Ниже приведены основные принципы выделения текста.

- Исходные тексты фрагментов программ, представляющие собой одну или более строк текста, выделяются специальным шрифтом. При этом комментарии, в том числе и вставленные мастером создания приложений, приводятся на русском языке:

```
/** Конструктор класса KDEEditView */
KDEEditView::KDEEditView(QWidget *parent, const char *name)
  : KEdit(parent, name)
{
 setBackgroundMode(PaletteBase);
}
```

- Когда на эти фрагменты программ имеются ссылки из различных фрагментов текста, они оформляются листингом, как показано ниже:

Листинг 2.1. Заголовок класса `TestDlgImpl`

```
/** Класс реализации диалогового окна */
class TestDlgImpl : public TestDlg
{
  Q_OBJECT
 public:
  TestDlgImpl(QWidget *parent=0, const char *name=0,
                  bool modal = true, WFlags fl = 0);
  ~TestDlgImpl();

 public slots:
  virtual void copyFlag(bool);
  virtual void boxCopy(const QString&);
  virtual void destination(int);

 private:
  bool copy;
  int  dest;
};
```

- Если в тексте встречается имя класса, функции, имя типа переменной или фрагмент программного кода длиной менее строки, он выделяется специальным шрифтом. Например: `int`, `QWidget`, `slotUpdateSave` и т. д.

- Имена функции обычно даются в полной форме. Это значит, что если функция является членом класса, то перед ней указывается имя класса, членом которого она является. При этом указывается имя того класса в иерархии, в котором впервые появилась данная функция. Например: `KAction::setEnabled`. Если перед именем функции не стоит имя класса, значит это либо глобальная функция, либо рассматриваемая в настоящее время функция пользовательского класса, либо часто встречающаяся функция, полное описание которой было только что приведено.
- Заголовок диалогового окна, имя кнопки или команды меню выделяется специальным жирным шрифтом. Например: кнопка **Open**, команда меню **File | New** и т. д.
- Имена клавиш помещаются в угловые скобки, например <Ctrl>, <F1> и т. д.
- Если требуется одновременно нажать несколько клавиш, они объединяются знаком (+). Например: <Ctrl>+<Alt>+<Del>.
- При первом появлении нового термина он выделяется курсивом. Например: *новый термин.*
- Примечания оформляются специальным стилем, как это показано ниже:

Примечание

Примечания содержат дополнительную информацию, которую можно и пропустить.

- Информация, на которую следует обратить особое внимание, оформляются следующим образом:

Внимание!

На эту информацию следует обратить особое внимание.

- Советы читателю оформляются так:

Совет

Это всего лишь совет.

Требования к аппаратным средствам и программному обеспечению

Для работы с описанной в данной книге средой программирования KDevelop 2.1.3 требуется следующее аппаратное и программное обеспечение:

- IBM-совместимый PC с процессором Intel Pentium II/III/Celeron или AMD Athlon/Duron;

- не менее 128 Мбайт оперативной памяти;
- свободное пространство на системном жестком диске не менее 2 Гбайт;
- дисплей SVGA и соответствующий видеоадаптер, обеспечивающие разрешение не менее 800×600 точек и 256 цветов (рекомендовано 65 536 цветов);
- дисковод CD-ROM;
- совместимая с Linux мышь;
- операционная система Linux с KDE 3.0.3 (например, Red Hat 8.0 Mandrake 9.0) и выше.

ГЛАВА 1

Взаимодействие компонентов приложения

Прежде чем перейти к описанию приложений, создаваемых в среде разработки KDevelop, необходимо сначала ознакомиться с тем, каким образом взаимодействуют друг с другом отдельные приложения и отдельные части одного и того же приложения.

Приложения операционной системы Linux взаимодействуют с ней посредством обмена событиями. События в объектах могут возникать по различным причинам, как вследствие взаимодействия пользователя с элементами пользовательского интерфейса, так и вследствие внутренних причин, например, завершения какой-то обработки. Единственное, что их объединяет, — это необходимость передать некоторую информацию другому объекту приложения.

Сигналы и приемники

В основе взаимодействия компонентов библиотеки Qt лежит концепция *сигналов* (signal) и *приемников* (slot), являющаяся альтернативой концепции использования для связи между объектами приложения указателей на функции, лежащей в основе X Window. Старая концепция существенно усложняла текст программы и не позволяла производить проверку типов возвращаемых объектов (поскольку эти функции имели тип возвращаемого значения `void*`). В основе концепции сигналов и приемников лежит простой принцип: при возникновении в некотором объекте события, могущего заинтересовать другие объекты, он посылает сигнал, не заботясь о том, будет ли он принят. В заинтересованном объекте устанавливается ловушка на данный сигнал, запускающая при его обнаружении функцию обработки возникшего события, называемую в библиотеке Qt *приемником*.

Сигналы и приемники не поддерживаются языком C++, однако их объявления включаются в структуры классов данного языка. Это кажущееся противоречие объясняется тем, что файлы заголовков и реализации классов, создаваемые разработчиком в среде KDevelop, не являются окончательными,

а будут перед своей компиляцией обработаны препроцессором moc (Meta Object Compiler), устраняющим все противоречия и создающим необходимые структуры для обработки событий.

Для использования механизма сигналов и приемников при обработке событий в своих приложениях разработчик должен:

1. Создавать сигналы и приемники только в классах, производных от класса `QObject`.
2. Добавить в начало заголовка класса макрос `Q_OBJECT` (без последующей точки с запятой).
3. Обработать файл заголовка препроцессором moc для получения компилируемого файла реализации.

Поскольку большинство приложений KDE и Qt используют automake и autoconf (в том числе и все приложения, созданные в среде KDevelop), то вызов препроцессора moc осуществляется в них автоматически, если в этом возникает необходимость. Для этого в заголовок класса и включается макрос `Q_OBJECT`.

Посылка сигналов

Как правило, посылаемые сигналы включаются в заголовок класса, объект которого будет их посылать. Для этого используется фиктивный модификатор прав доступа `signals`, который может использоваться только для функций, не возвращающих значения. Пример заголовка класса, посылающего сигналы, приведен в листинге 1.1.

Листинг 1.1. Заголовок класса, посылающего сигналы

```
class someClass : public QObject
{
 Q_OBJECT

public:
 someClass();

 void someFunction(int);

signals:
 void someSignal();
 void someParameterSignal(int);
};
```

Как видно из листинга 1.1, в качестве сигналов могут выступать как функции без аргументов, так и функции с аргументами, в которых передаются различные параметры посылаемого сигнала.

Посылка сигнала может производиться в любой функции приложения с использованием ключевого слова `emit`. Для транслятора языка C++ это ключевое слово определено как отсутствие какого-либо текста, поэтому для него посылка сигнала интерпретируется как простой вызов функции. Но препроцессор moc при обнаружении данного ключевого слова создает и инициализирует соответствующий метаобъект, реализующий сигнал как функцию члена класса в выходных файлах препроцессора. В листинге 1.2 приведен пример использования ключевого слова `emit` для посылки сигнала `someParameterSignal` из объекта описанного выше класса.

Листинг 1.2. Посылка сигнала

```
void someClass::someFunction(int i)
{
 emit someParameterSignal( i);
}
```

В данном случае функция `someFunction` используется исключительно для посылки сигнала и задания значения его аргумента, однако в этой функции может производиться и любая другая обработка, а посылка сигнала будет только одной из выполняемых ею задач.

Описанный выше метод посылки сигнала широко распространен, однако встречаются случаи, когда класс не может являться потомком класса `QObject`, но из его объекта требуется послать сигнал. В этом случае для посылки сигнала используется объект специального класса `QSignal`. Для использования этого класса в приложении в файл заголовка класса, из которого нужно посылать сигналы, прежде всего, следует включить файл заголовка qsignal.h, содержащий описание класса `QSignal`. После этого в заголовок данного класса добавляется указатель на объект типа `QSignal` и объявление метода `connect`, которое должно иметь следующий вид:

```
void connect(QObject* receiver, const char* member);
```

В конструкторе класса необходимо, используя оператор `new`, создать динамический объект класса `QSignal` и уничтожить его в деструкторе класса оператором `delete`.

После этого в классе необходимо реализовать его функцию `connect`. Для класса `someClass`, содержащего указатель на объект класса `QSignal` с именем `pQSignal`, эта реализация должна иметь следующий вид:

```
void connect(QObject* receiver, const char* member)
{
 pQSignal-> connect(receiver, member);
}
```

После этого для посылки сигнала следует вызывать функцию

```
pQSignal->activate();
```

Реализация приемников

В отличие от сигналов, являющихся функциями C++ только по своему синтаксису, приемники представляют собой полноценные функциональные члены класса, которые могут вызываться не только как обработчики сигналов, но и самостоятельно, из других функций приложения. Однако объявление приемника все-таки отличается от объявления любой другой функции класса добавлением в ее модификатор права доступа ключевого слова `slots`. Кроме того, функции приемников, так же как и функции соответствующих им сигналов, не могут возвращать никакого значения. Пример заголовка класса, содержащего приемники, приведен в листинге 1.3.

Листинг 1.3. Заголовок класса, содержащего приемники

```
class otherClass : public QObject
{
 Q_OBJECT

public:
 otherClass();

public slots:
 void someSlot1();
 void someParameterSlot(int);
};
```

При сравнении листингов 1.1 и 1.3 следует обратить внимание на то, что если сигналы не имеют других прав доступа, кроме фиктивного права доступа `signals`, то для приемников обязательно нужно указать модификатор права доступа к ним. Если этот приемник не должен наследоваться потомками данного класса, для него следует указать модификатор права доступа `private`. Для наследуемых приемников достаточно указать модификатор права доступа `protected`. Модификатор доступа `public`, если разобраться, определяет их уже не как приемники, а как обычные функции класса, поскольку разрешает доступ к ним из других классов, что никогда не происходит с приемниками.

Примечание

Хотя все приемники являются полноправными функциями приложения, настоятельно не рекомендуется помещать в этот раздел описания класса другие его функции, не являющиеся приемниками. В тех случаях, когда за описанием сиг-

налов должно следовать описание других членов класса, для них следует в явном виде указать модификатор прав доступа, что будет свидетельствовать о завершении описания приемников.

Реализация соединения

Для связи сигнала с приемником используется статическая функция `QObject::connect`, имеющая следующий синтаксис:

```
bool connect(const QObject* sender, const char* signal, const QObject*
receiver,
const char* member)
```

Первым аргументом данной функции является указатель на объект класса, посылающего сигнал, вторым аргументом — указатель на строку, идентифицирующую сигнал, третьим аргументом — указатель на объект класса, в котором будет производиться обработка данного сигнала, и четвертым — указатель на строку, идентифицирующую функцию обработки данного сигнала (приемник).

Для получения строки, идентифицирующей сигнал, как правило, используется макрос `SIGNAL`, в качестве аргумента которому передается сигнатура функции сигнала (его имя и типы аргументов), а для получения строки, идентифицирующей приемник, используется макрос `SLOT`, в качестве аргумента которому передается сигнатура функции приемника. Поскольку связывание сигнала с приемником можно трактовать как способ обеспечения вызова функции одного класса из объекта другого класса, списки формальных параметров функций сигнала и приемника должны совпадать.

Примечание

Допускается, чтобы список формальных параметров приемника был короче списка формальных параметров сигнала. В этом случае требуется, чтобы список формальных параметров приемника совпадал с началом списка формальных параметров сигнала. Допускается и полное отсутствие формальных параметров у приемника.

Во избежание появления в подобных случаях предупреждений о неиспользованных аргументах функции, в описании сигнала указываются только типы формальных параметров без их имен.

Допускается трансляция сигнала, т. е. использование в качестве приемника сигнала другого сигнала, который должен быть послан объектом-получателем при обнаружении исходного сигнала. В этом случае вместо текстового идентификатора приемника при вызове функции `connect` используется текстовое описание нового сигнала.

Поскольку функция `QObject::connect` полностью характеризует устанавливаемое соединение, она может располагаться в любой точке приложения,

в которой пересекаются области видимости источника сигнала и его приемника. Один и тот же сигнал могут обрабатывать сразу несколько приемников, но, в отличие от Windows, обработка будет происходить параллельно, а не последовательно. (Разумеется, на однопроцессорной машине все обработки будут выполняться последовательно, но вот порядок их вызова будет непредсказуем.)

Если требуется разорвать связь сигнала и приемника, используется статическая функция `QObject::disconnect`, имеющая следующий синтаксис:

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

Она имеет тот же набор аргументов, что и установившая соединение функция `connect`.

> **Примечание**
>
> В классах, производных от класса `QObject`, можно использовать не только статическую версию данной функции, но и ее перегруженную версию, позволяющую, например, при добавлении команды меню связывать сигнал `activated` непосредственно с приемником.

Обработка событий

Для работы с элементами пользовательского интерфейса в приложениях X Window используется *обработка событий*. При нажатии пользователем, работающим с приложением X Window, клавиши на клавиатуре или щелчке кнопкой мыши в окне этого приложения протокол X11 гарантирует, что для обработки данного события будет вызвано нужное приложение. Для обработки событий в приложении организуется *главный цикл обработки событий* (main event loop), поддержание которого и является главной задачей приложения. Этот цикл продолжает работать до тех пор, пока в него не поступит обращение к приемнику `quit` или в какой-нибудь функции приложения не будет вызвана функция `exit`. Завершение работы главного цикла обработки сообщений приводит к завершению работы функции `exec`, вызываемой в функции `main` для инициализации работы этого цикла. Поскольку вызов функции `exec` является последним оператором функции `main`, завершение ее работы приводит к завершению работы всего приложения.

Согласно протоколу X11, для каждого окна определяется обрабатываемый им набор событий, как правило, включающий в себя такие события, как `XExposeEvent`, `XDestroyWindowEvent` и `XResizeRequestEven`. Если возникшее в процессе работы приложения событие включено в данный набор, оно помещается в очередь событий, где и ожидает своей обработки. Все остальные события просто игнорируются.

Непосредственная работа с циклом обработки сообщений не очень удобна и довольно опасна, поскольку разработчик может случайно изменить реакцию приложения на какое-нибудь жизненно важное событие, что способно привести к краху приложения. Поэтому в библиотеке Qt для работы с событиями создан специальный класс `QEvent`, объекты которого могут посылаться объектам других классов функцией `QApplication::notify`. В объектах классов, которые должны обработать данное событие, производится перегрузка виртуальной функции `QObject::event`, получающей и распознающей посланное событие. Для определения того, какие события будут обрабатываться данным классом, в нем устанавливается фильтр событий. Событие не будет обработано классом, содержащим функцию его обработки, если это будет запрещено фильтром.

Если событие получается и правильно распознается функцией `event` какого-либо класса, она возвращает значение `true`, которое, в свою очередь, является возвращаемым значением функции `notify`, пославшей это событие. Событие считается обработанным и удаляется из очереди. В противном случае возвращается значение `false`.

Внимание!

Обратите внимание на коренное отличие в обработке событий и сигналов: событие всегда обрабатывается одним обработчиком, сколько бы их ни было, а сигнал обрабатывается всеми настроенными на него приемниками.

Все события, возникающие при работе пользователя с элементами пользовательского интерфейса приложения, поступают в объект этого приложения, где они преобразуются в объекты класса `QEvent`, передаются объекту активного окна. Если в активном окне находится обработчик данного события, он возвращает объекту приложения значение `true`, извещая его о том, что событие обработано. После этого событие удаляется из очереди событий. Если в активном окне верхнего уровня не нашлось обработчика данного события, оно передается его активному дочернему окну. Если обработчик события не найден во всей иерархии активных окон, то это событие игнорируется и удаляется из очереди обработки событий.

Для определения набора событий, обрабатываемых классом окна, в нем устанавливается фильтр обработки событий. Подобный фильтр может быть установлен в любом классе, производном от класса `QObject`.

Обработка событий производится в перегруженной функции `QObject::event`, которой в качестве аргумента передается указатель на объект класса `QEvent`, содержащий информацию о событии. Для определения того, какое именно событие получено, можно использовать функцию `QEvent::type`, сравнив возвращенное ею значение с предопределенной константой. После определения истинного типа события можно произвести явное преобразование

типа объекта события, установив для него его истинный тип, а не тип его базового класса, и вызвать функции-члены класса события. Пример перегрузки функции `event` в некотором классе приведен в листинге 1.4.

Листинг 1.4. Перегрузка функции event

```
bool someClass::event( QEvent* event)
{

 if( event->type() == Event_FocusIn){
  if( (QFocusEvent*)event->reason() == Mouse){
   // Обработать сообщение
   return true;
  }
  else
   return false;
 }
 else
  return false;
}
```

Функция в листинге 1.4 будет обрабатывать только сообщение об активизации окна щелчком мыши. Сообщения обо всех остальных способах его активизации будут игнорироваться.

Примечание

Список предопределенных констант типов событий можно найти в файле q1xcompatibility.h. Эти константы образуются прибавлением префикса `Event_` к значению перечислимого типа `Type`, определенного в классе `QEvent`.

Приложения, использующие библиотеку Qt и являющиеся развитием приложения KDE, используют для работы с пользователем графический интерфейс. Базовым классом для любых графических объектов в библиотеке Qt является класс `QWidget`, в котором произведена перегрузка функции `QObject::event`, обеспечивающая более эффективную обработку событий, возникающих в графических приложениях.

Как уже говорилось ранее, информация обо всех действиях, производимых пользователем с окном, поступает в объект класса `QApplication`, создающий в ответ на полученную информацию объекты класса `QEvent` и производных от него классов, затем передающий эти объекты для обработки функции `QWidget::event`. В этой функции, аналогично тому, как это было сделано нами в листинге 1.4, производится анализ типа поступившего объекта события.

Прежде всего в нем вызываются фильтры событий, перегружающие установленные по умолчанию процедуры обработки сообщений и отсеивающие сообщения, которые не должны обрабатываться данным классом. После этого, с использованием функции `QEvent::type`, определяется тип поступившего события, и информация о нем помещается в объект одного из классов событий, которые будут описаны в следующих подразделах данного раздела. Указатель на этот объект передается в качестве аргумента виртуальной функции обработки данного события.

Необходимость создания объектов различных классов для передачи информации о поступившем событии связана с тем, что каждое из этих событий индивидуально и сложно создать информационную структуру, одинаково годную для хранения информации о перемещении фокуса ввода и нажатии пользователем клавиши. Поэтому в специализированных объектах классов событий создаются оптимальные для них структуры данных и в них включаются все необходимые методы для облегчения доступа к этим данным.

Поскольку существуют некоторые стандартные способы обработки определенных событий в окне, известные пользователю и, как правило, не изменяемые разработчиком, то виртуальные функции обработки событий в классе `QWidget` содержат реализацию этих методов, снимая эту задачу с разработчика. Если же разработчику нужно изменить реакцию на событие, то он может перегрузить соответствующую функцию в своем классе.

Виртуальные функции класса `QWidget` позволяют обрабатывать стандартные типы событий, описываемые стандартными классами, но ничто не мешает разработчику создать свой собственный класс события и определить для него функцию обработки.

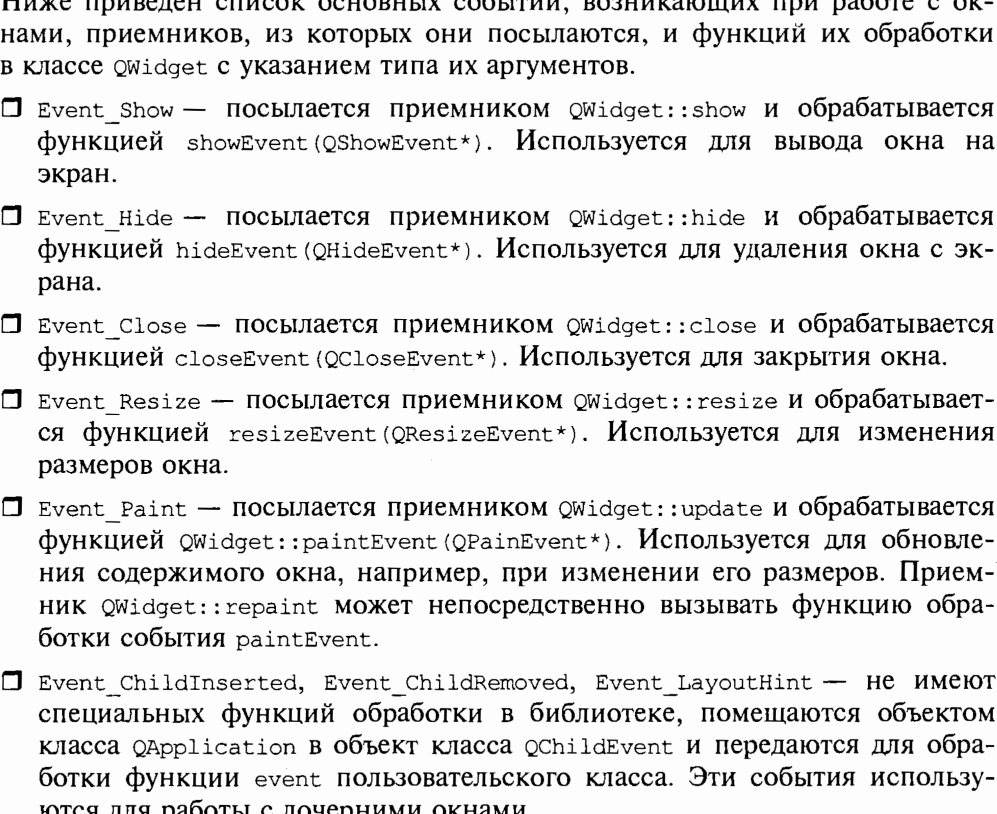
Внимание!

При перегрузке функций обработки событий настоятельно рекомендуется оставлять их виртуальными и не делать их закрытыми членами класса, что позволит использовать эти классы в качестве базовых при создании специфических элементов пользовательского интерфейса.

Работа с окном

Действия пользователя при работе с окном, как правило, не могут быть сразу оформлены в виде события. Этому обычно предшествует некоторая обработка, в результате которой и формируется соответствующее событие. Как правило, это событие, называемое *синтетическим событием*, формируется в одном из приемников класса окна. Для посылки событий окнам эти приемники используют функции с префиксом `qt_`. Обработка этих событий идет дальше обычным образом через перегруженную функцию. Такое поведение позволяет элементам окна, которые, в свою очередь, являются самостоятельными окнами, взаимодействовать друг с другом.

Ниже приведен список основных событий, возникающих при работе с окнами, приемников, из которых они посылаются, и функций их обработки в классе QWidget с указанием типа их аргументов.



- Event_Show — посылается приемником QWidget::show и обрабатывается функцией showEvent(QShowEvent*). Используется для вывода окна на экран.
- Event_Hide — посылается приемником QWidget::hide и обрабатывается функцией hideEvent(QHideEvent*). Используется для удаления окна с экрана.
- Event_Close — посылается приемником QWidget::close и обрабатывается функцией closeEvent(QCloseEvent*). Используется для закрытия окна.
- Event_Resize — посылается приемником QWidget::resize и обрабатывается функцией resizeEvent(QResizeEvent*). Используется для изменения размеров окна.
- Event_Paint — посылается приемником QWidget::update и обрабатывается функцией QWidget::paintEvent(QPainEvent*). Используется для обновления содержимого окна, например, при изменении его размеров. Приемник QWidget::repaint может непосредственно вызывать функцию обработки события paintEvent.
- Event_ChildInserted, Event_ChildRemoved, Event_LayoutHint — не имеют специальных функций обработки в библиотеке, помещаются объектом класса QApplication в объект класса QChildEvent и передаются для обработки функции event пользовательского класса. Эти события используются для работы с дочерними окнами.

Работа с фокусом ввода

Фокус ввода является атрибутом окна, и поэтому связанные с ним события можно было бы не выделять в отдельную группу, а отнести к предыдущей группе, однако обработка этих событий имеет свою специфику, которую следует рассмотреть отдельно.

Понятие фокуса ввода связано с тем, что в приложении, как правило, выстраивается целая иерархия окон, причем только одно окно в этой иерархии может быть активным. Активное окно обрабатывает события клавиатуры, и это можно считать наилучшим его отличительным признаком. Фокус ввода может устанавливаться щелчком левой кнопки мыши в пределах окна или же передаваться от окна к окну с использованием клавиатуры. Для определения последовательности передачи фокуса ввода между окнами формируется кольцевая очередь окон. Для перехода к следующему окну в очереди нажимается клавиша <Tab>, а для перехода к предыдущему окну — комбинация клавиш <Shift>+<Tab>. Использование клавиатуры для передачи фо-

куса ввода используется, в основном, в диалоговых окнах. При работе с многооконными приложениями, особенно ориентированными на текст, этот способ может быть и не реализован.

В некоторых случаях необходимо исключить активизацию некоторого окна. В качестве такого окна может выступать, например, кнопка панели инструментов или элемент управления диалогового окна, обращение к которым в данной конкретной ситуации недопустимо. Такое окно исключается из очереди активизируемых окон, не активизируется мышью. Как правило, такое окно еще и соответствующим образом выделяется (перерисовывается в серых тонах), чтобы сообщить пользователю о своей недоступности.

Для определения того, какие способы активизации будут доступны для окна, и будет ли окно вообще активизироваться, используется функция `QWidget::setFocusPolicy`, в качестве аргумента которой передается одно из значений перечислимого типа `QWidget::FocusPolicy`, приведенных в следующем списке:

- `QWidget::TabFocus` — для передачи фокуса ввода может использоваться клавиатура;
- `QWidget::ClickFocus` — активизация может производиться щелчком левой кнопки мыши;
- `QWidget::StrongFocus` — фокус ввода может устанавливаться мышью и передаваться клавиатурой;
- `QWidget::WheelFocus` — аналогичен `StrongFocus`, но для активизации окна может использоваться и колесико мыши;
- `QWidget::NoFocus` — окно не может быть активизировано.

При обнаружении события активизации или деактивации в перегруженной функции `QWidget::event` создается объект класса `QFocusEvent`, указатель на который передается в качестве аргумента одной из функций обработки событий: `QWidget::focusInEvent`, обрабатывающей событие активизации `Event_FocusIn`, или `QWidget::focusOutEvent`, обрабатывающей событие деактивации `Event_FocusOut`.

Методы класса `QWidget` позволяют вносить существенные изменения в процедуру передачи фокуса ввода, например, они позволяют изменять порядок его передачи от одного окна к другому. Однако, используя эту возможность, следует помнить о том, что изначально все окна были организованы в циклическую очередь, которая при подобных действиях может разорваться или из нее могут быть исключены некоторые окна. Порядок включения окон в очередь активизации определяется порядком инициализации дочерних окон в конструкторе родительского окна.

Разработчики среды KDevelop советуют пользователям заранее продумывать дизайн своих диалоговых окон и создавать в них элементы управления, рас-

полагая их слева направо и сверху вниз. Хороший совет, только не ясно от кого он исходит: от человека, не создавшего за всю свою жизнь ни одного полноценного проекта, или от гениального разработчика и дизайнера, способного с первого раза, без единой итерации, создать распространяемую версию приложения, к которой не будет нареканий со стороны ни одного из ее пользователей. Наверное, проще, все-таки, изменить порядок инициализации окон в конструкторе.

Примечание

Команды меню и кнопки панели инструментов, как правило, не получают фокуса ввода. Единственным распространенным исключением из этого правила является кнопка контекстной справки (**What's This**), которая, располагаясь в панели инструментов, может получить фокус ввода с клавиатуры.

Работа с мышью

Поскольку мышь используется пользователем для работы с конкретным окном, то и функции обработки возникающих при этом событий помещены в класс `QWidget`. Ниже приведен список этих событий и обрабатывающих их функций, с указанием типа их аргументов.

- `Event_MouseButtonPress` — свидетельствует о нажатии кнопки мыши. Обрабатывается функцией `mousePressEvent(QMouseEvent *)`.
- `Event_MouseButtonRelease` — свидетельствует об отпускании кнопки мыши. Обрабатывается функцией `mouseReleaseEvent(QMouseEvent *)`.
- `Event_MouseButtonDblClick` — свидетельствует о двойном щелчке кнопкой мыши. Обрабатывается функцией `mouseDoubleClickEvent(QMouseEvent *)`.
- `Event_MouseMove` — свидетельствует о перемещении мыши. Обрабатывается функцией `mouseMoveEvent(QMouseEvent *)`.
- `Event_Enter` — свидетельствует о перемещении указателя мыши в окно. Обрабатывается функцией `enterEvent(QEvent *)`.
- `Event_Leave` — свидетельствует о перемещении указателя мыши за пределы окна. Обрабатывается функцией `leaveEvent(QEvent *)`.
- `Event_Wheel` — свидетельствует о прокрутке колесика мыши. Обрабатывается функцией `wheelEvent(QWheelEvent*)`. Появилось в версии Qt 2.0.

Примечание

Двойной щелчок мыши многими понимается по-разному. Для того чтобы каждый пользователь мог настроить максимальный интервал между двумя щелчками, воспринимаемыми как один двойной щелчок, в соответствии со своими привычками, в классе `QApplication` существует функция `setDoubleClickInterval`, позволяющая задать его в миллисекундах. По умолчанию этот интервал составляет 400 мс.

Для простоты обработки двойной щелчок мыши по умолчанию обрабатывается так же, как и нажатие кнопки мыши.

Мышь является указательным устройством, поэтому большинство полученных событий будет невозможно обработать без точного знания положения указателя мыши. Кроме того, само событие не позволяет определить, какая из кнопок мыши была нажата. Поэтому объект класса `QMouseEvent` содержит большой объем дополнительной информации. Так, для определения того, какая кнопка мыши была нажата, используется функция `QMouseEvent::button`, а для определения того, какая функциональная клавиша была нажата при возникновении обрабатываемого события, применяется функция `QMouseEvent::state`. Обе эти функции возвращают значение перечислимого типа `ButtonState`, определенного в файле qnamespace.h следующим образом:

```
// Документировано в qevent.cpp
enum ButtonState {    // Значения состояния мыши/клавиатуры
NoButton = 0x0000,
LeftButton = 0x0001,
RightButton = 0x0002,
MidButton = 0x0004,
MouseButtonMask = 0x0007,
ShiftButton = 0x0008,
ControlButton  = 0x0010,
AltButton = 0x0020,
KeyButtonMask = 0x0038,
Keypad  = 0x4000
};
```

Поскольку значения кодов кнопок и клавиш не пересекаются, они могут объединяться операцией ЛОГИЧЕСКОГО ИЛИ.

Для определения местоположения мыши при возникновении обрабатываемого события могут быть использованы следующие функции (как они определены в заголовке класса):

```
const QPoint &pos() const { return p; }
const QPoint &globalPos() const { return g; }
int   x()  const { return p.x(); }
int   y()  const { return p.y(); }
int   globalX() const { return g.x(); }
int   globalY() const { return g.y(); }
```

Как видно, эти функции позволяют получить как глобальные, так и оконные координаты указателя мыши. Причем эта информация может быть получена как объект класса `QPoint` или как значения отдельных координат.

По умолчанию перемещение мыши обрабатывается только в том случае, если у нее нажата какая-либо кнопка. В противном случае это событие иг-

норируется функцией QWidget::event. Для того чтобы окно могло обрабатывать любые перемещения мыши, следует вызвать функцию QWidget::setMouseTracking с аргументом true.

Работа с клавиатурой

События, возникающие при работе пользователя с клавиатурой, обрабатываются объектом активного окна, т. е. окна, имеющего в данный момент фокус ввода. Поэтому функции обработки этих событий помещены в класс QWidget. При работе с клавиатурой могут возникнуть только два события:

- Event_KeyPress — свидетельствует о нажатии клавиши. Обрабатывается функцией keyPressEvent(QKeyEvent*);
- Event_KeyRelease — свидетельствует об отпускании клавиши. Обрабатывается функцией keyReleaseEvent(QKeyEvent*).

Примечание

Если текущее окно может передавать фокус ввода по нажатии клавиши <Tab>, нажатие этой клавиши и комбинации клавиш <Shift>+<Tab> отфильтровывается функцией QWidget::event и помещается не в объект класса QKeyEvent, как все остальные нажатия клавиш, а в объект класса QFocusEvent.

При работе с клавиатурой, в отличие от работы с мышью, пользователь не может непосредственно указать, к какому элементу управления он обращается. При работе с оконными интерфейсами, как правило, возникает достаточно сложная иерархия окон. Типичным примером такой иерархии является диалоговое окно, объектами которого являются элементы управления этого окна. Во многих случаях нажатие пользователем клавиши должно обрабатываться не активным окном, а его родительским окном или одним из окон, занимающим еще более высокое положение в иерархии.

Однако, для определенности, все события, возникающие при работе с клавиатурой, поступают сначала в активное окно. Объект этого окна может обработать эти события или сразу же передать их своему родительскому окну, для чего ему достаточно вызвать функцию QKeyEvent::ignore в поступившем объекте класса события.

Для определения того, какая клавиша была нажата, вызывается функция QKeyEvent::key, возвращающая значение перечислимого типа Key, определенного в файле qnamespace.h. Эта же задача может быть решена и вызовом функции QKeyEvent::ascii, возвращающей, как следует из ее названия, ASCII-код нажатой клавиши. Однако использовать последнюю функцию настоятельно не рекомендуется, поскольку, во-первых, ASCII-код не является общепризнанным стандартом и, во-вторых, пользоваться мнемоническими константами намного удобнее, чем числовыми кодами.

Поскольку пользователь мог нажать не одну клавишу, а их комбинацию, в объекте класса `QKeyEvent` так же, как и в объекте класса `QMouseEvent`, может быть вызвана функция `state`, возвращающая значение описанного выше перечислимого типа `ButtonState`.

Для облегчения обработки нажатия пользователем комбинаций клавиш в библиотеке Qt предусмотрен специальный класс `QAccel`, содержащий фильтры событий для зарегистрированных в нем комбинаций клавиш. Каждая из зарегистрированных комбинаций должна состоять из обычной, нефункциональной клавиши и любой из клавиш <Ctrl>, <Shift> или <Alt>.

Чтобы создать в окне фильтр комбинаций клавиш, в его объекте создается объект класса `QAccel`, конструктору которого в качестве аргумента передается указатель на объект класса окна, и для каждой из комбинаций клавиш вызывается функция `QAccel::insertItem`, первым аргументом которой является регистрируемая комбинация клавиш, а вторым — сопоставляемый ей цифровой идентификатор.

Совет

Несмотря на то, что второй аргумент функции `QAccel::insertItem` является необязательным, его настоятельно рекомендуется использовать, поскольку при его отсутствии комбинации клавиш автоматически назначается уникальный отрицательный цифровой идентификатор. Хотя значение присваиваемого идентификатора возвращается данной функцией и может быть сохранено в соответствующей переменной, лучше не пускать это дело на самотек.

Некоторым командам раскрывающихся меню уже сопоставлены стандартные комбинации клавиш, другим командам они могут быть сопоставлены пользователем с применением функции `QMenuData::setAccel`, первым аргументом которой является регистрируемая комбинация клавиш, а вторым — цифровой идентификатор команды меню. Подобная функция существует и для кнопок. Однако, поскольку каждой кнопке соответствует свой объект, функция `QButton::setAccel` имеет только один аргумент, в котором передается комбинация клавиш, инициирующая данную кнопку.

Для установки связи с другими объектами, в которых будет производиться обработка нажатия комбинации клавиш, используется функция `QAccel::connectItem`, первым аргументом которой является цифровой идентификатор комбинации клавиш, вторым — указатель на объект, содержащий приемник, а третьим — текстовая строка, идентифицирующая соответствующий приемник. Для посылки сигнала вызывается функция `QAccel::activated`, в качестве аргумента которой передается цифровой идентификатор комбинации клавиш.

При работе в среде KDE у разработчика появляются дополнительные возможности по обработке нажатий комбинаций клавиш. Для этого он должен вместо объектов класса `QAccel` создавать во многом аналогичные им по функциональным возможностям объекты класса `KAccel`, позволяющие поль-

зователю с помощью специальных диалоговых окон производить настройку комбинаций клавиш, используемых для инициализации той или иной процедуры.

Реализация перетаскивания

Одним из основных преимуществ оконных интерфейсов является реализация в них концепции перетаскивания, позволяющая пользователю выделить в окне некоторый объект и переместить его в пределах того же самого окна или в другое окно. Операции, производимые с объектом в процессе его перетаскивания, зависят от природы этого объекта. Например, перетаскивание значка некоторого документа на значок приложения, способного работать с данным типом документа, может быть интерпретировано как запрос на открытие данного документа в этом приложении.

Для реализации перетаскивания используется протокол XDND, генерирующий необходимые для этого события. Эти события должны генерироваться в двух оконных объектах, причем не требуется, чтобы все эти события могли бы в полном объеме генерироваться в каждом из этих объектов. Первый объект, в котором изначально находился перетаскиваемый объект, называется *источником перетаскивания* (dragsource). В этом объекте необходимо реализовать выделение перетаскиваемого объекта, его перемещение и его удаление при выходе перетаскиваемого объекта за границы окна. Второй объект, в который будет перемещен перетаскиваемый объект, называется *адресатом перетаскивания* (drop site или drop sink). В этом объекте необходимо реализовать перемещение перетаскиваемого объекта, его добавление в оконный объект и другие связанные с ним объекты при его отпускании.

При перемещении объекта в пределах окна источником перетаскивания и его адресатом является один и тот же оконный объект. В библиотеке Qt версии 2.0 протокол XDND реализован непосредственно в классе `QWidget`, и для включения в создаваемый оконный объект поддержки перетаскивания достаточно вызвать в конструкторе его класса функцию `QWidget::setAcceptDrops` с аргументом `true`. В предыдущей версии этот протокол был реализован в специальном классе `QDropSite`.

Оконный объект, поддерживающий перетаскивание, может обрабатывать следующие события:

- `Event_DragEnter` — свидетельствует о начале перетаскивания. Обрабатывается функцией `dragEnterEvent(QDragEnterEvent*)`;
- `Event_DragMove` — свидетельствует о перемещении перетаскиваемого объекта. Обрабатывается функцией `dragMoveEvent(QDragMoveEvent*)`;
- `Event_DragLeave` — свидетельствует о выходе перетаскиваемого объекта за пределы окна. Обрабатывается функцией `dragLeaveEvent(QDragLeaveEvent*)`;

- `Event_Drop` — свидетельствует о завершении операции перетаскивания. Обрабатывается функцией `dropEvent(QDropEvent*)`;
- `Event_DragResponse` — системное событие, обрабатываемое внутренними функциями объекта. Определяет, будет ли обрабатываться данное событие.

Внимание!

Реализация протокола DND в библиотеке Qt отличается от реализации Motif DND (используемой Netscape and GNOME).

Фильтры событий

Фильтры событий позволяют разработчику вмешиваться в установленный по умолчанию ход обработки событий. Как уже говорилось выше, процесс обработки событий достаточно сложен и для его автоматизации используется метод `QObject::event`, обеспечивающий для каждого события свою обработку. Но что делать в том случае, если разработчику нужно внести изменения в процесс обработки некоторых событий? Неужели ему придется отказаться от использования данного метода и писать свой обработчик событий? Конечно же, нет! Для этого ему достаточно написать соответствующий фильтр события, выделяющий его из потока всех остальных событий при внутреннем вызове метода `QObject::event`.

Существует два способа фильтрации событий: ее можно производить в том же объекте, в котором производится обработка событий, или в другом объекте. Процесс фильтрации событий производится в методе `QObject::eventFilter`. Установка фильтра в объекте класса, обрабатывающем событие, производится вызовом метода `QObject::installEventFilter`, а прекращение фильтрации событий производится вызовом метода `QObject::removeEventFilter`. Указанные методы имеют следующий синтаксис:

```
bool QObject::eventFilter ( QObject *, QEvent *) [virtual]
void QObject::installEventFilter ( const QObject * obj)
void QObject::removeEventFilter ( const QObject * obj)
```

В листинге 1.5 приведен пример перегрузки метода `QObject::eventFilter`.

Листинг 1.5. Реализация фильтра событий

```
bool SomeClass::eventFilter(QObject* object, QEvent* event){

 if(event->type() == Event_FocusIn)
 {
 ...
```

2 Зак. 113

```
 // Обработка события
 ...
  return true; // Событие перехвачено и обработано
 }
 else
 {
  return false; // Обработка события будет продолжена
         // в методе QObject::event()
 }
}
```

Если фильтрация производится в том же классе, в котором производится обработка события, то метод `QObject::installEventFilter`, как правило, вызывается в конструкторе данного класса. При этом устанавливаются сразу все фильтры. Аналогично, вызов метода `QObject::removeEventFilter` полностью прекращает фильтрацию событий в данном объекте класса.

Если фильтр событий создается в отдельном объекте, у разработчика появляется больше пространства для маневра: он может в любой момент включить или отключить фильтрацию того или иного события.

Примечание

Класс `KApplication` уже содержит фильтр для нажатия пользователем комбинации клавиш <Ctrl>+<Alt>+<F12>, вызывающей приложение KDebug.

Синтетические события

Синтетические события уже рассматривались нами при описании работы с окном. Здесь эти события будут рассмотрены подробнее.

Необходимость в создании синтетических событий связана с тем, что не каждое событие может быть распознано системой на основании первичной информации о действии пользователя. Например, нажатие на кнопку может быть распознано только после анализа текущего положения курсора и сравнения его с областями выведенных на экран кнопок. Возникают и другие ситуации, когда событие должно быть создано в процессе обработки первичного события. Конечно, можно все эти события свести к первичным событиям (нажатиям кнопок мыши, перемещениям ее указателя и т. д.) и создать для них сложные и разветвленные функции обработки, но это существенно снизит читабельность программы. Поэтому гораздо проще на основе анализа полученной информации программно формировать новые события, которые будут практически неотличимы от первичных событий и будут обрабатываться по тем же правилам.

Поскольку фантазия разработчиков программного обеспечения практически безгранична, и они очень болезненно реагируют на любую попытку ограни-

чить предоставленный им инструментарий, им дана возможность программно создавать практически любое событие. Для этого ему необходимо создать объект событий и послать его. Ниже приведены конструкторы основных типов событий с кратким описанием их аргументов.

- `QEvent(int type)` — перечень допустимых значений аргумента можно найти в файле qevent.h, где они объединены в перечислимый тип `Type`. В большинстве случаев значения этого типа совпадают с именами предопределенных констант событий, но у них отсутствует префикс `Event_`.
- `QCloseEvent()` — не имеет аргументов.
- `QFocusEvent(int type)` — аргумент может принимать значение `Event_FocusIn` или `Event_FocusOut`.
- `QKeyEvent(int type, int key, int ascii, int state)` — первый аргумент может принимать значение `Event_KeyPress` или `Event_KeyRelease`. Значения второго аргумента определены в файле qkeycode.h. Третий аргумент может содержать любую комбинацию флагов `ShiftButton`, `ControlButton` и `AltButton`.
- `QMouseEvent (int type, const QPoint & pos, int button, int state)` — первый аргумент может принимать значение `Event_MouseButtonPress`, `Event_MouseButtonRelease`, `Event_MouseButtonDblClick` или `Event_MouseMove`. Второй аргумент содержит позицию указателя мыши. Третий аргумент может принимать значение `LeftButton`, `RightButton`, `MidButton` или `NoButton`. Если первый аргумент принимает значение `Event_MouseButtonRelease`, четвертый аргумент может содержать любую комбинацию флагов `ShiftButton`, `ControlButton` и `AltButton`. Если первый аргумент принимает значение `Event_MouseButtonPress` или `Event_MouseButtonDblClick`, к набору флагов четвертого аргумента добавляются флаги `LeftButton`, `RightButton` и `MidButton`.
- `QMoveEvent(const QPoint & pos, const QPoint & oldPos)`— первый аргумент содержит новую позицию окна на экране, а второй аргумент — его старую позицию. Для получения значения второго аргумента можно использовать функцию `QWidget::pos`.
- `QPaintEvent(const QRect & paintRect)` — аргумент данного конструктора определяет область перерисовки.
- `QResizeEvent(const QSize & size, const QSize & oldSize)` — первый аргумент содержит новый размер окна, а второй аргумент — его старый размер. Для получения значения второго аргумента можно использовать функцию `QWidget::size`.

После создания события его нужно пустить на обработку. Поскольку событие является программным, эту операцию можно выполнить непосредственно, вызвав соответствующую функцию приложения. Для этого используется функция `QApplication::sendEvent`, первый аргумент которой содержит ука-

затель на объект получателя события, а второй — указатель на объект посылаемого события. Вызов этой функции остановит выполнение приложения источника на время обработки события, и после завершения этого процесса функция возвратит логическое значение, указывающее на отсутствие ошибок в процессе обработки.

Если нет необходимости дожидаться результатов обработки синтетического события, разработчик может использовать функцию `QApplication::postEvent`, имеющую тот же набор аргументов, но помещающую событие в очередь и немедленно возвращающую управление приложению источника. Посылаемый объект событий должен создаваться в куче (heap) и уничтожаться непосредственно после своей посылки.

Последовательность обработки событий

События в очереди, как и следовало ожидать, обрабатываются в порядке их поступления. Однако в некоторых случаях эту стратегию нельзя признать удачной. Наиболее явно ее недостатки проявляются тогда, когда обработка одного из событий в очереди занимает достаточно продолжительное время, препятствуя обработке событий, реализующих пользовательский интерфейс.

Для борьбы с этим недостатком в библиотеке Qt предусмотрена функция `QApplication::processEvents`, осуществляющая обработку ожидающих в очереди событий в течение указанного промежутка времени или до исчерпания очереди в зависимости от того, что скорее произойдет. Если промежуток времени в явном виде не указан, он принимается равным 3 секундам. Место и время вызова данной функции полностью лежит на совести разработчика.

Заключение

В данной главе помещен достаточно сложный материал, требующий от читателя способности охватить большой объем информации и проследить в нем отдельные взаимосвязи. Поэтому в заключение этой главы мне кажется не лишним привести список базовых понятий, на которые можно опираться при уяснении себе различных вопросов.

- Для взаимодействия отдельных компонентов приложения используются сигналы и приемники.
- При посылке сигнала отсутствует какая-либо проверка того, что он будет принят и обработан.
- Приемники представляют собой обычные методы классов, связываемые с сигналами и вызываемые при их поступлении.
- Приемники не могут содержать аргументы, значения которых устанавливаются по умолчанию, и не могут возвращать значения.

- Ничто не мешает вызывать приемники как обычные методы при наличии соответствующих прав доступа.
- Для передачи приложению информации от элементов пользовательского интерфейса используются события.
- Приложение преобразует полученные события в объекты класса `QEvents`.
- Для обработки событий может быть использован глобальный фильтр событий.
- Прошедшие фильтр события направляются объекту активного в настоящее время окна.
- Объект окна может иметь свой собственный фильтр событий, вызов которого является первой операцией в перегруженной функции `QObject::event`.
- Прошедшие фильтр события, в зависимости от своего типа, преобразуются в объекты классов, производных от класса `QEvents`.
- Для полученных объектов классов вызываются их фильтры событий, которые и осуществляют обработку событий.

Для внесения изменений в описанный выше процесс обработки событий разработчик может:

- перегрузить любой используемый в данном процессе виртуальный метод;
- создать синтетическое событие и послать его непосредственно или поместить в очередь событий;
- приостановить выполнение длинных операций для инициализации процесса обработки ожидающих в очереди событий;

Таким образом, механизм обработки сигналов и событий предоставляет разработчику мощный инструмент для обеспечения взаимодействия компонентов сложных приложений и реализации в них пользовательского интерфейса.

ГЛАВА 2

Диалоговые окна и простейшие элементы управления

После ознакомления с принципами взаимодействия компонентов в приложении можно непосредственно переходить к созданию последних. Лучше всего начать с создания диалоговых приложений. Этот тип приложений широко распространен в среде Windows, а в среде разработки Microsoft Visual Studio имеется специальный мастер для их создания. Такой мастер отсутствует в среде KDevelop, что, однако, не исключает возможности создания в ней подобных приложений.

Выбор диалоговых приложений в качестве исходной точки для изучения приложений среды разработки KDevelop продиктован не только тем, что без диалоговых окон практически невозможно написать какую-либо программу для работы в графической среде, но и минимальной поддержкой, обеспечиваемой данному типу приложений, что обуславливает минимальный размер заготовки проекта приложения.

Создание диалогового приложения

Полноценная программа в любой графической среде может иметь множество диалоговых окон, каждое из которых будет предназначено для обмена информацией или ее вывода в определенном формате. Во многих случаях все приложение может быть оформлено как одно диалоговое окно, элементы которого используются как для ввода исходных данных, так и для инициализации их обработки и вывода результатов этой обработки. Такие приложения, называемые *диалоговыми приложениями*, очень удобны для проверки работоспособности и отладки отдельных алгоритмов.

В отличие от Microsoft Visual Studio, помещающего описание ресурсов диалогового окна в текстовый файл с расширением rc, среда разработки

KDevelop создает файл с расширением ui и помещает в него описание диалогового окна в формате, совместимом с XML. Этот файл включается в makefile данного проекта, создающий на его основе соответствующие файлы заголовков и реализации класса диалогового окна.

Учитывая, что эти файлы будут заново создаваться при каждой перекомпиляции приложения, разработчик не может использовать этот класс непосредственно, а должен создать производный от него класс, поместив его заголовок и реализацию в другие файлы, и уже в них вносить все изменения, необходимые для обеспечения функциональности создаваемого диалогового окна.

Поскольку процесс создания диалогового приложения достаточно сложен, он будет разбит на несколько этапов, каждый из которых будет поясняться отдельно.

Создание заготовки приложения

Поскольку мастер создания приложения среды KDevelop не дает возможности непосредственно создать заготовку диалогового приложения, мы используем в качестве базового типа приложение **KDE Mini**, содержащее минимум текста, но обеспечивающее всю необходимую поддержку в работе с диалоговыми окнами. Текст созданного приложения можно найти на прилагаемом к книге CD.

Чтобы самостоятельно создать диалоговое приложение:

1. Выберите команду меню **Project | New** (Проект | Новый). Появится диалоговое окно **ApplicationWizard** (Мастер создания приложений).
2. В окне иерархического списка диалогового окна в папке **KDE** выделите строку **KDE Mini** (Приложение KDE с минимальными возможностями) и нажмите кнопку **Next** (Далее). Появится второе окно мастера **ApplicationWizard**.
3. В текстовое поле **Project name** (Имя проекта) введите имя проекта **Dialog**, заполните остальные текстовые поля корректной информацией (или оставьте их без изменений, поскольку этот проект не будет распространяться) и нажмите кнопку **Create** (Создать). Появится последнее окно мастера **ApplicationWizard** и начнется процесс создания заготовки приложения.
4. Как только станет доступной кнопка **Exit** (Выход), нажмите ее и выйдите из мастера создания приложений.

Эта процедура ничем не отличается от процедуры создания любого другого приложения в среде KDevelop и выделена в самостоятельный раздел только потому, что процедуры создания заготовки диалогового окна и его настройки не зависят от типа использующего его приложения.

Создание заготовки диалогового окна

Для создания заготовки диалогового окна среда разработки KDevelop использует приложение Qt Designer, доступ к которому можно получить, вызвав команду меню **View | Dialog Editor** (Вид | Редактор диалогового окна), однако для более тесной интеграции диалогового окна в приложение, существенно облегчающей последующую работу с ним, следует использовать описанную ниже процедуру.

Чтобы создать заготовку диалогового окна, в приложении Qt Designer:

1. Выберите команду меню **File | New** (Файл | Создать), нажмите комбинацию клавиш <Ctrl>+<N> или кнопку **New** в панели инструментов. Появится диалоговое окно **New File** (Новый файл), изображенное на рис. 2.1.

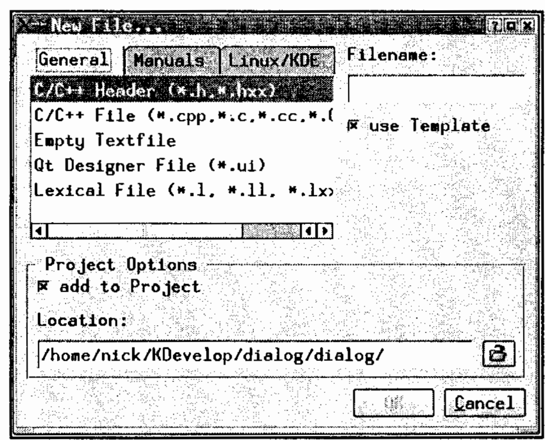


Рис. 2.1. Диалоговое окно **New File**

2. В окне списка раскрытой вкладки **General** (Общие свойства) выделите строку **Qt Designer File (*.ui)** (Файл приложения Qt Designer), в текстовое поле **Filename** введите имя файла `testdlg` (расширение ui будет добавлено к нему автоматически) и нажмите кнопку **OK**. Появится диалоговое окно **Load decision** (Тип загрузки), изображенное на рис. 2.2 и предлагающее загрузить файл в текстовом виде.

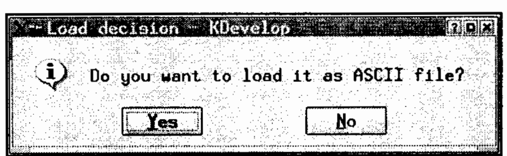


Рис. 2.2. Диалоговое окно **Load decision**

3. Нажмите кнопку **No** (Нет). Появится диалоговое окно **file** (Файл), изображенное на рис. 2.3.

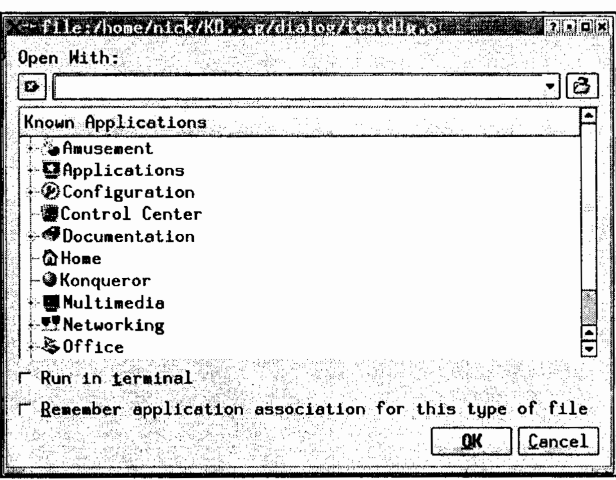


Рис. 2.3. Диалоговое окно **file**

4. Нажмите кнопку **Open File Dialog** (Диалоговое окно открытия файлов), расположенную справа от текстового поля **Open With** (Открыть с помощью), появится диалоговое окно **KDevelop**, изображенное на рис. 2.4.

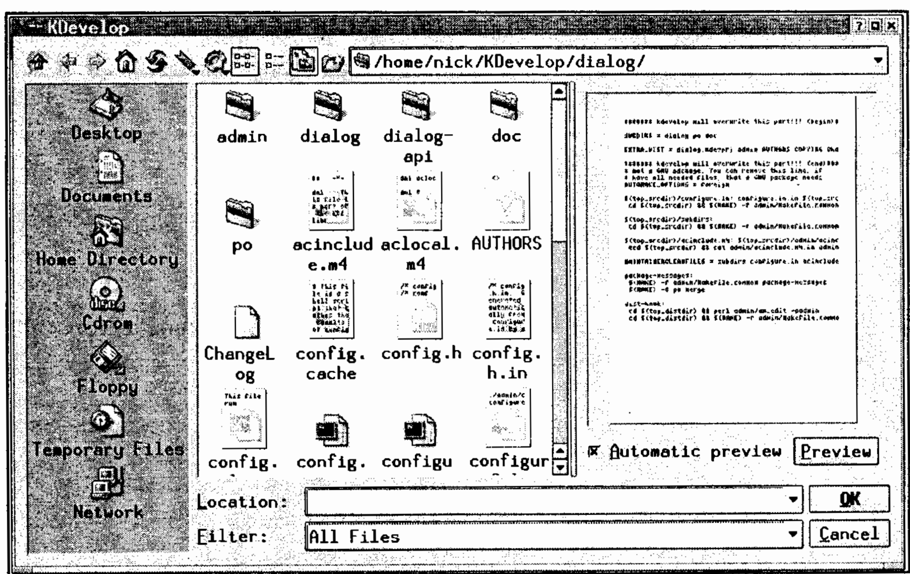


Рис. 2.4. Диалоговое окно **KDevelop**

5. Раскройте в нем каталог usr\bin, выделите в окне списка файл designer-qt3 и нажмите кнопку **OK**. Имя этого приложения появится в текстовом поле **Open With**.

6. В диалоговом окне **file** установите значок **Remember application association for this type of file** (Запомнить связь открываемого приложения с типом файла), чтобы это диалоговое окно больше не появлялось, и нажмите кнопку **OK**. Откроется окно приложения **Qt Designer by Trolltech**, изображенное на рис. 2.5.

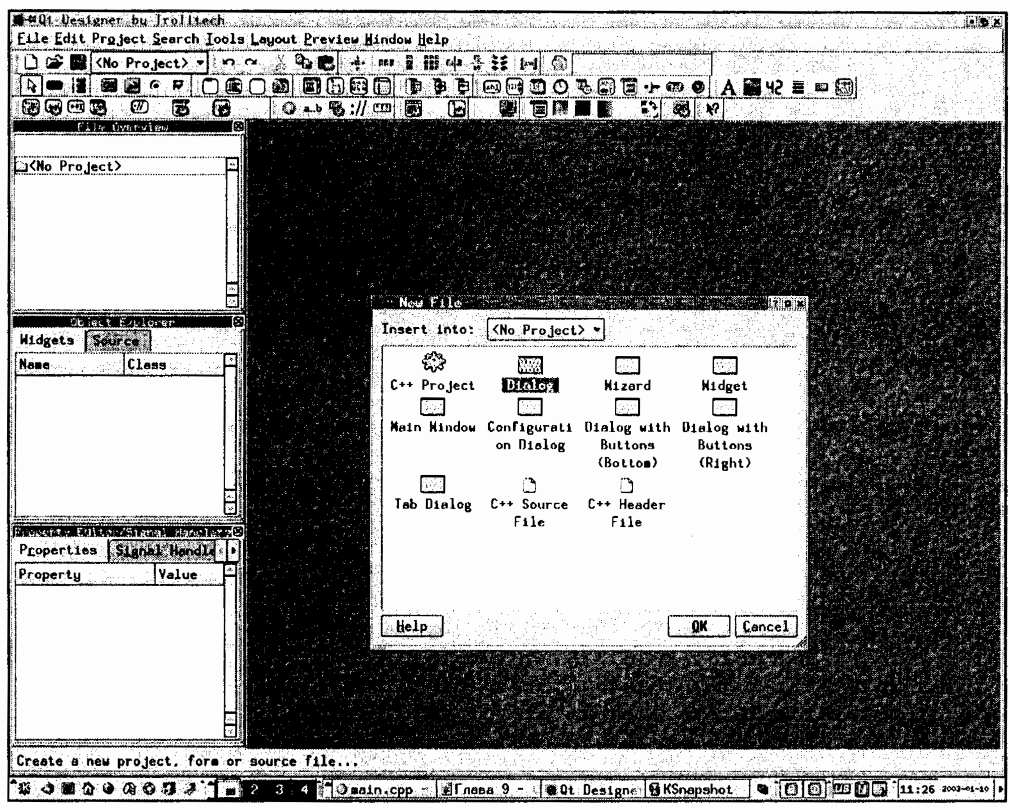


Рис. 2.5. Приложение Qt Designer by Trolltech

7. В окне списка появившегося при старте приложения диалогового окна **New File** (Новый файл) выделите значок **Dialog** (Диалоговое окно) и нажмите кнопку **OK**. В приложении появится пустая заготовка диалогового окна, а в панели **Property Editor/Signal Handlers** (Редактор свойств/Обработчики сигналов) появится список свойств созданного окна.

8. В панели **Property Editor/Signal Handlers** выделите строку **name** и замените содержащееся в нем имя диалогового окна **Form1** именем **TestDlg**.

Совет

Имя диалогового окна и имя файла, в который будет помещено его описание, должны совпадать, поскольку имя диалогового окна используется приложением KDevelop для именования создаваемого им класса диалогового окна, а имя файла — для именования файлов заголовка и реализации этого класса (все символы переводятся в нижний регистр). С другой стороны, при создании производного класса это приложение считает, что имена класса и имена его файлов реализации и заголовка совпадают.

9. В той же панели выделите строку **caption** (заголовок) и замените содержащийся в нем заголовок диалогового окна **Form1** заголовком **Qt Dialog**.
10. Выберите команду меню **Tools | Display | TextLabel** (Сервис | Изображения | Статический текст) или нажмите кнопку **Text Label** в панели инструментов **Display** (при этом кнопка должна "утопиться") и щелкните левой кнопкой мыши в левом верхнем углу заготовки диалогового окна. Появится рамка статического текста, изображенная на рис. 2.6.

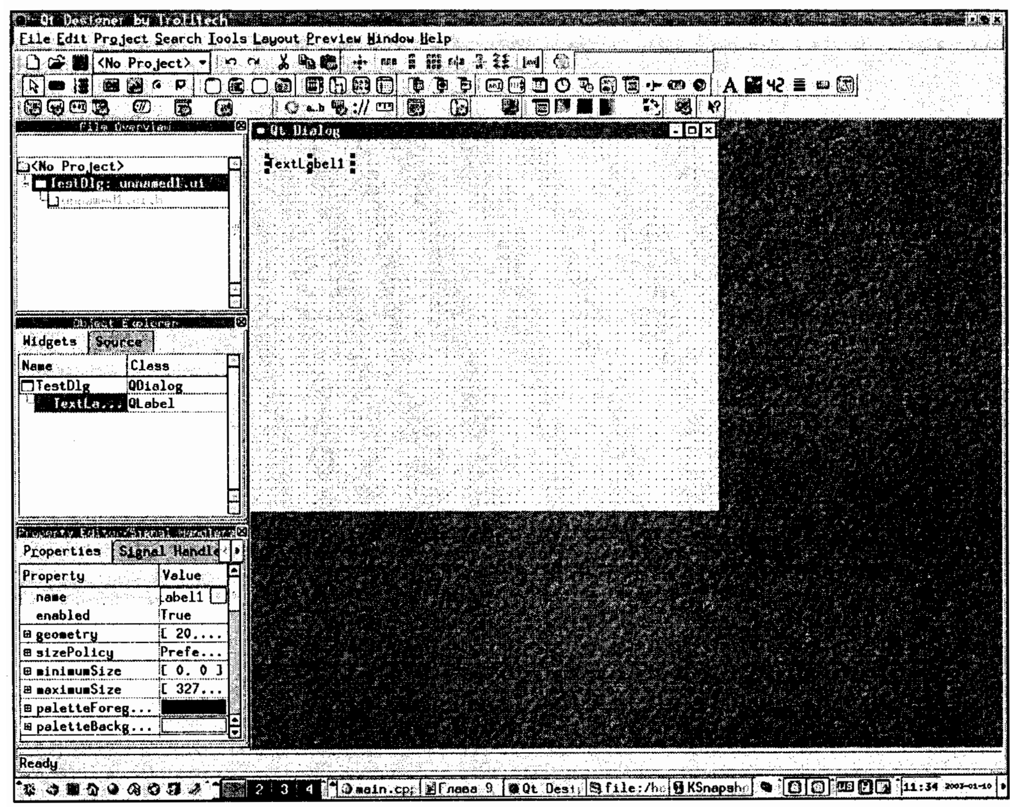


Рис. 2.6. Заготовка диалогового окна с рамкой статического текста

11. В панели **Property Editor/Signal Handlers** выделите строку **text** (текст) и введите в связанное с ней текстовое поле строку **LineEdit**.
12. Выберите команду меню **Tools | Input | LineEdit** (Сервис | Ввод | Текстовое поле) или нажмите кнопку **Line Edit** в панели инструментов **Input** и щелкните левой кнопкой мыши под только что введенным статическим текстом. На месте щелчка появится текстовое поле.
13. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя текстового поля на **LineEdit**.
14. Повторите пункты 10 и 11 для создания справа от заголовка текстового поля нового статического текста **SpinBox**.
15. Выберите команду меню **Tools | Input | SpinBox** (Сервис | Ввод | Инкрементный регулятор) или нажмите кнопку **Spin Box** в панели инструментов **Input** и щелкните левой кнопкой мыши под только что введенным статическим текстом. В заготовке диалогового окна появится инкрементный регулятор.
16. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя инкрементного регулятора на **SpinBox**.
17. Повторите пункты 10 и 11 для создания справа от заголовка инкрементного регулятора нового статического текста **Combo Box**.
18. Выберите команду меню **Tools | Input | ComboBox** (Сервис | Ввод | Раскрывающийся список) или нажмите кнопку **Combo Box** в панели инструментов **Input** и создайте под только что введенным статическим текстом раскрывающийся список.
19. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя раскрывающегося списка на **ComboBox**.
20. В той же панели выделите строку **editable** (Редактируемый) и выделите в связанном с ней раскрывающемся списке строку **True**. При этом заготовка диалогового окна примет вид, изображенный на рис. 2.7.
21. Выберите команду меню **Layout | Add Spacer** (Расположение | Добавить разделитель) или нажмите кнопку **Spacer** в панели инструментов **Layout**.
22. Щелкните левой кнопкой мыши слева от статического текста **Line Edit** и выберите в появившемся контекстном меню команду **Horizontal** (Горизонтальный разделитель). На месте щелчка появится горизонтальный разделитель (Spacer).
23. Повторите пункты 21 и 22, щелкнув левой кнопкой мыши справа от статического текста **Line Edit**. При этом заготовка диалогового окна примет вид, изображенный на рис. 2.8.
24. Поместите указатель мыши в левый верхний угол заготовки диалогового окна, нажмите левую кнопку мыши и, не отпуская ее, переместите указатель мыши. Появится рамка.

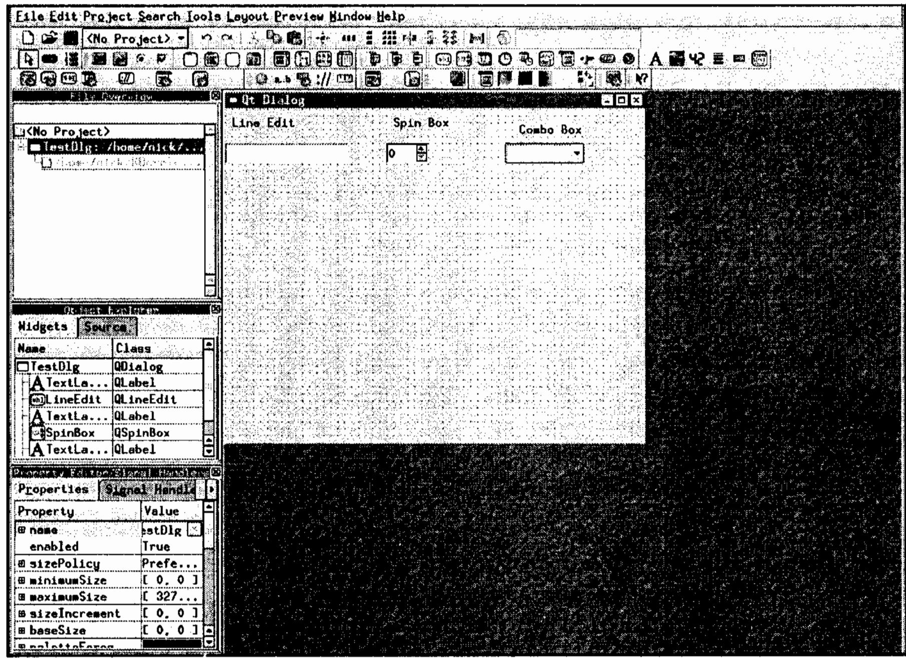


Рис. 2.7. Заготовка диалогового окна с раскрывающимся списком

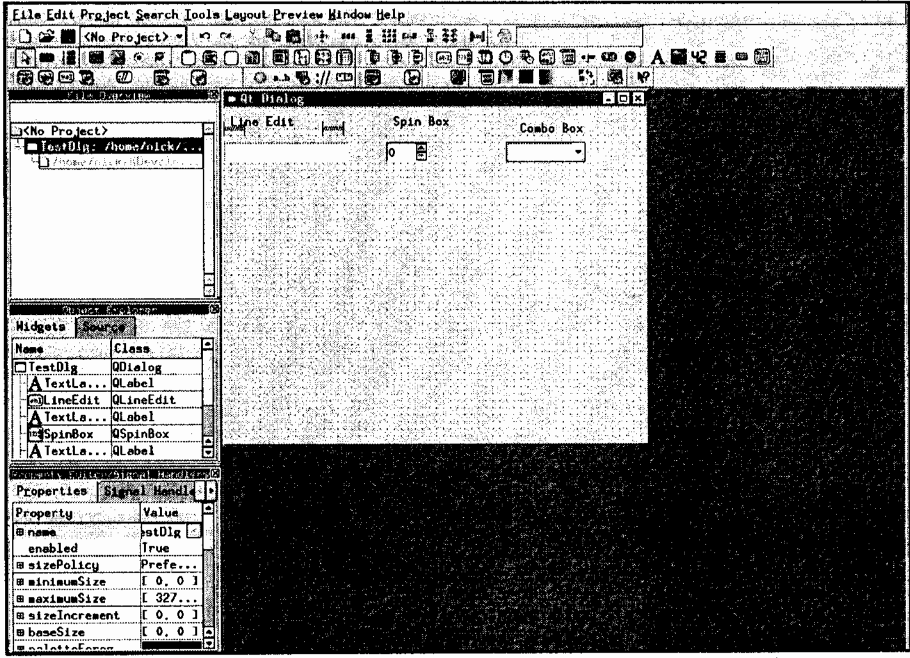


Рис. 2.8. Заготовка диалогового окна с разделителями

25. Охватите рамкой статический текст **Line Edit** и два разделителя. Отпустите левую кнопку мыши. Элементы управления, попавшие в рамку, будут выделены.

Совет

Если при выделении текстового поля и его заголовка в рамку попали и выделились другие элементы управления, отмените их выделение щелчком левой кнопки мыши при нажатой клавише <Shift>.

26. Выберите команду меню **Layout | Lay Out Horizontally** (Расположение | Расположить по горизонтали), нажмите комбинацию клавиш <Ctrl>+<H> или кнопку **Lay Out Horizontally** в панели инструментов **Layout**. Текстовое поле и два разделителя будут автоматически выровнены по горизонтали, как это показано на рис. 2.9.

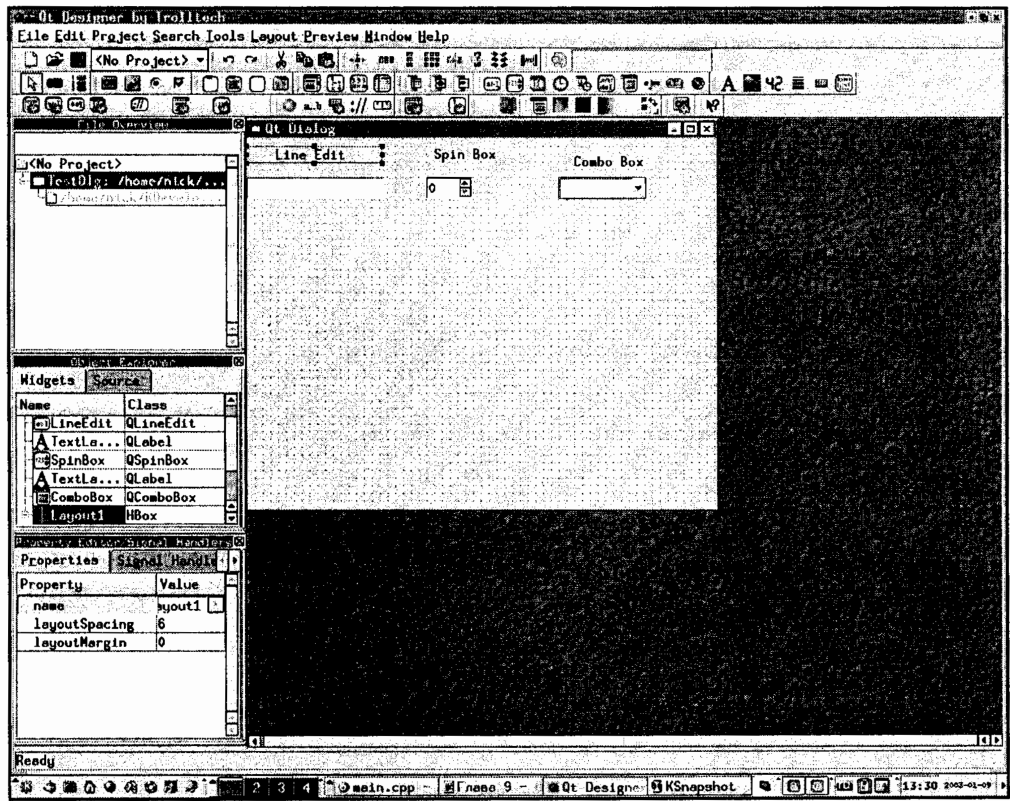


Рис. 2.9. Выравнивание по горизонтали

27. Поместите указатель мыши в левый верхний угол заготовки диалогового окна, нажмите левую кнопку мыши и, не отпуская ее, обведите полу-

чаемой при перемещении указателя мыши рамкой текстовое поле с его заголовком. Текстовое поле и его заголовок будут выделены.

28. Выберите команду меню **Layout | Lay Out Vertically** (Расположение | Расположить по вертикали), нажмите комбинацию клавиш <Ctrl>+<L> или кнопку **Lay Out Vertically** в панели инструментов **Layout**. Текстовое поле и его заголовок будут автоматически выровнены по вертикали, как это показано на рис. 2.10.

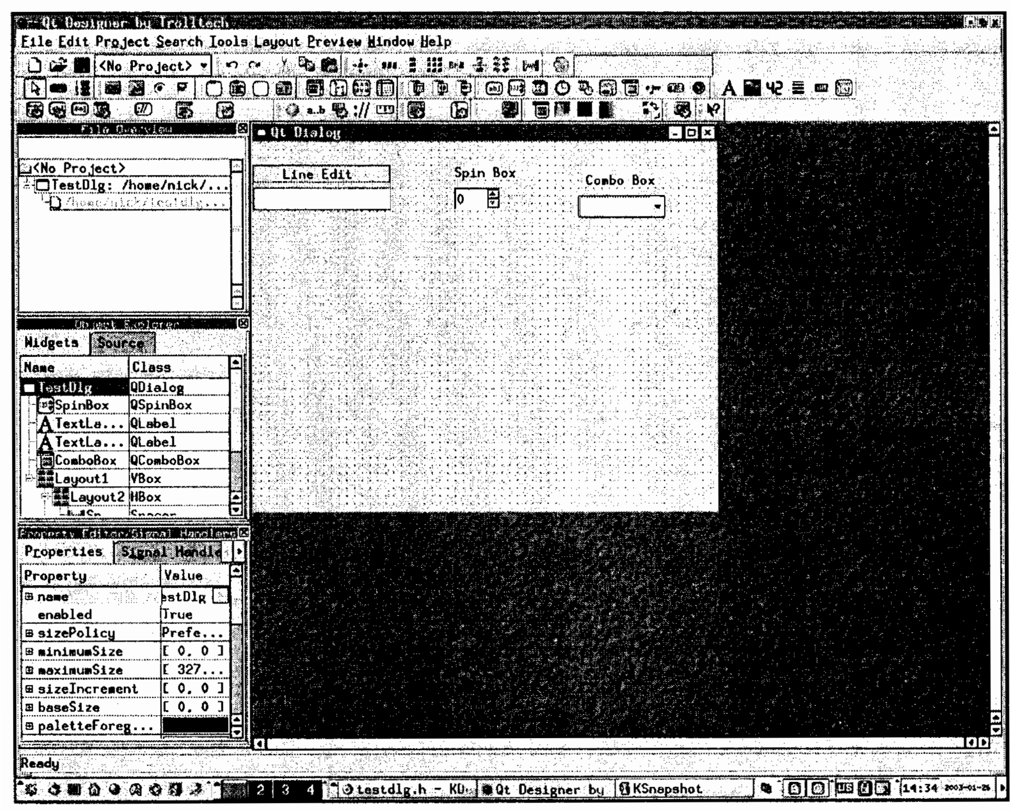


Рис. 2.10. Выравнивание по вертикали

29. Повторите пункты 21—28 для инкрементного регулятора и его заголовка.

30. Повторите пункты 21—28 для раскрывающегося списка и его заголовка.

31. Поместите горизонтальные разделители справа от текстового поля, слева от раскрывающегося списка и по обеим сторонам инкрементного регулятора, выделите все элементы управления и разделители и выровняйте их по горизонтали. В результате описанных выше действий заготовка диалогового окна примет вид, изображенный на рис. 2.11.

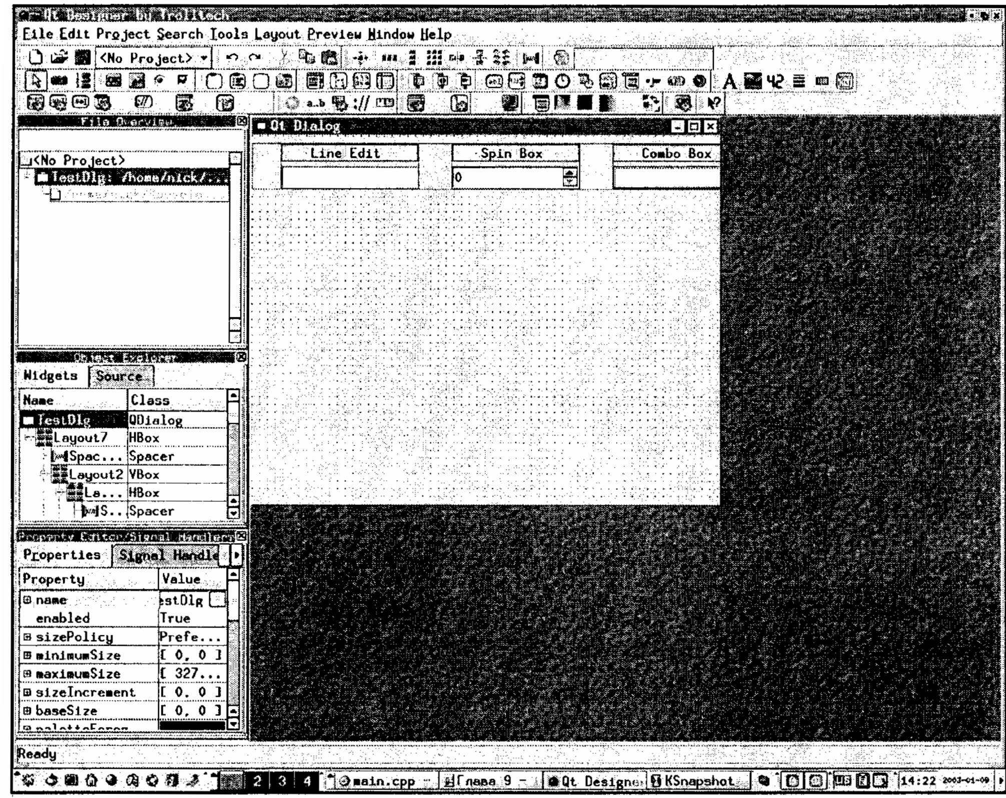


Рис. 2.11. Полное выравнивание строки элементов управления

Примечание

То, что выравнивание элементов управления привело к их выходу за пределы заготовки диалогового окна, не имеет на данный момент никакого значения, поскольку это будет исправлено впоследствии.

32. Выберите команду меню **Tools | Containers | ButtonGroup** (Сервис | Контейнер | Групповая рамка кнопок) или нажмите кнопку **Button Group** в панели инструментов **Containers** и создайте групповую рамку кнопок под текстовым полем.
33. В панели **Property Editor/Signal Handlers** выделите строку **title** (заголовок) и измените заголовок группы переключателей на **Destination**.
34. В той же панели выделите строку **name** и измените имя группы переключателей на **DestGroup**.
35. Выберите команду меню **Tools | Buttons | RadioButton** (Сервис | Кнопки | Переключатель) или нажмите кнопку **Radio Button** в панели инструмен-

тов **Buttons**, по описанной выше методике создайте в верхней части групповой рамки переключатель.

36. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя переключателя на **EditSwitch**.

37. В той же панели выделите строку **text** и введите в связанное с ней текстовое поле строку **Line Edit**.

38. В панели **Property Editor/Signal Handlers** выделите строку **checked** (отмеченный) и выделите в связанном с ней раскрывающемся списке строку **True**.

39. Повторите пункты 35—37, поместив под только что введенным переключателем новый переключатель с именем **SpinSwitch** и текстом **Spin Box**.

40. Выделите элемент управления групповой рамки и выберите команду меню **Layout | Lay Out Vertically**, нажмите комбинацию клавиш <Ctrl>+ +<L> или кнопку **Lay Out Vertically** в панели инструментов **Layout**. Групповая рамка расширится, а переключатели в ней выровняются по вертикали. При этом заготовка диалогового окна примет вид, изображенный на рис. 2.12.

41. Выберите команду меню **Tools | Buttons | CheckBox** (Сервис | Кнопки | Флажок) или нажмите кнопку **Check Box** в панели инструментов **Buttons** и создайте флажок справа от группы переключателей.

42. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя флажка на **CopyCheck**.

43. В той же панели выделите строку **text** и введите в связанное с ней текстовое поле строку **Copy?** (Копировать?).

44. Выберите команду меню **Tools | Buttons | PushButton** (Сервис | Кнопки | Кнопка) или нажмите кнопку **Push Button** в панели инструментов **Buttons** и создайте кнопку справа от флажка.

45. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя кнопки на **OKButton**.

46. В той же панели выделите строку **text** и введите в связанное с ней текстовое поле строку **OK**.

47. Поместите горизонтальные разделители слева от групповой рамки, справа от кнопки **OK** и по обеим сторонам флажка.

48. Выделите групповую рамку, флажок, кнопку **OK** и введенные в предыдущем пункте разделители и выровняйте их по горизонтали. При этом заготовка диалогового окна примет вид, изображенный на рис. 2.13.

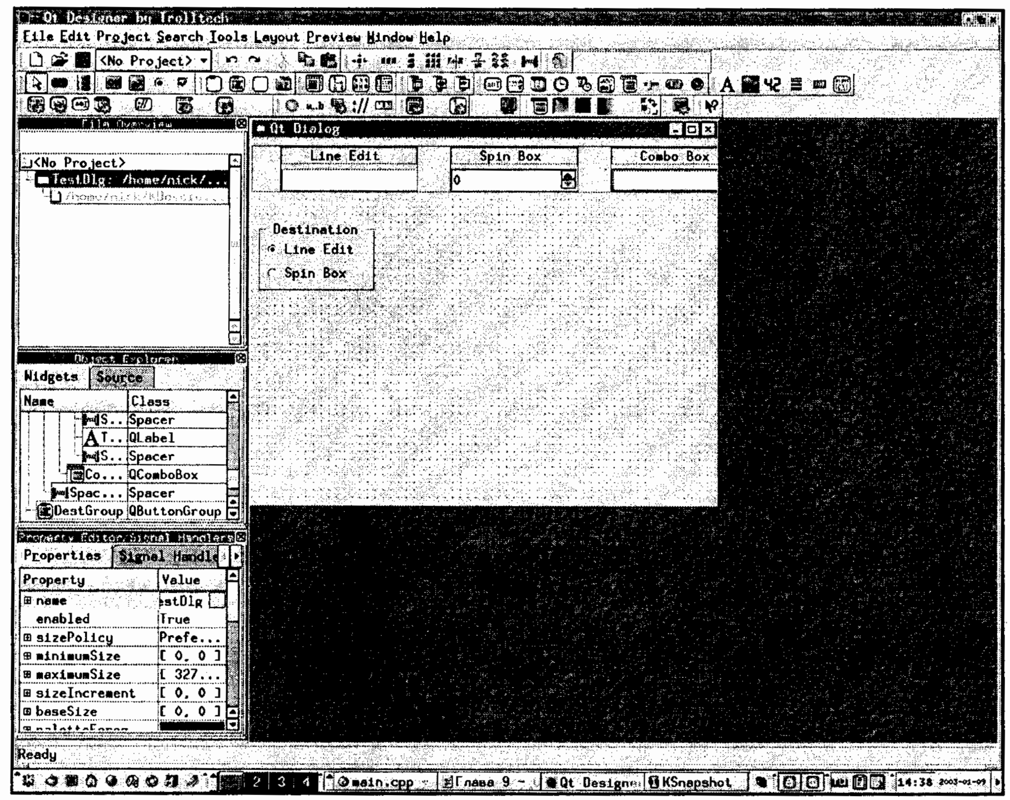


Рис. 2.12. Заготовка диалогового окна с группой переключателей

49. Выберите команду меню **Layout | Add Spacer** или нажмите кнопку **Spacer** в панели инструментов **Layout**.

50. Щелкните левой кнопкой мыши над строкой заголовков и выберите в появившемся контекстном меню команду **Vertical** (Вертикальный разделитель). На месте щелчка появится вертикальный разделитель.

51. Повторите пункты 49 и 50, щелкнув левой кнопкой мыши выше и ниже флажка.

52. Щелкните левой кнопкой мыши в заготовке диалогового окна за пределами элементов управления и выберите команду меню **Layout | Lay Out Vertically**, нажмите комбинацию клавиш <Ctrl>+<L> или кнопку **Lay Out Vertically** в панели инструментов **Layout**. Элементы управления диалогового окна будут выровнены по вертикали.

53. Выберите команду меню **Layout | Adjust Size** (Расположение | Настроить размер), нажмите комбинацию клавиш <Ctrl>+<J> или кнопку **Adjust Size** в панели инструментов **Layout**. Размеры заготовки диалогового окна будут приведены в соответствие с его содержимым.

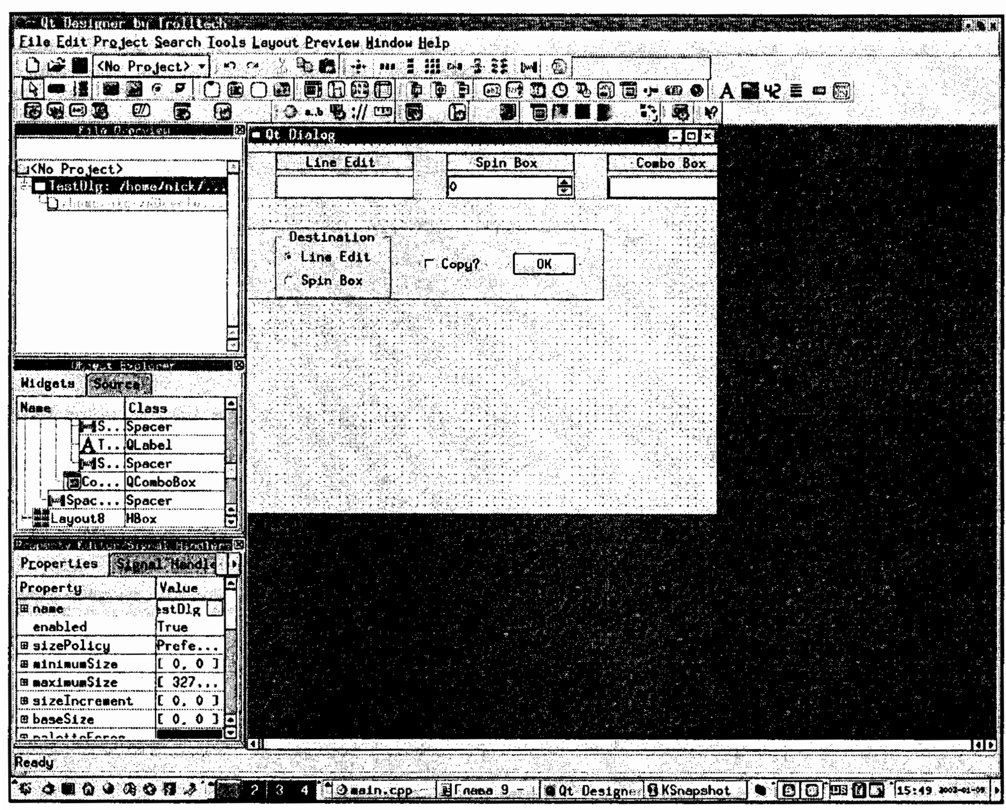


Рис. 2.13. Выравнивание по горизонтали завершено

54. Выберите команду меню **Tools | Tab Order**, нажмите клавишу <F4> или кнопку **Tab Order** в панели инструментов **Tools**. Кнопка **Tab Order** в панели инструментов "утопится", и заготовка диалогового окна примет вид, изображенный на рис. 2.14.

55. Щелкните левой кнопкой мыши по синему кружку с номером **7**, расположенным около кнопки **OK**. В этом кружке появится номер **1**, а номера в остальных кружках будут увеличены на единицу.

56. Нажмите клавишу <Esc>. Порядковые номера элементов управления исчезнут.

57. Выберите команду меню **Preview | Preview Form** (Предварительный просмотр | Предварительный просмотр формы) или нажмите комбинацию клавиш <Ctrl>+<T>. Появится макет созданного нами диалогового окна, изображенный на рис. 2.15.

58. Измените размеры макета диалогового окна. Его элементы управления будут плавно перемещаться по его поверхности и его размер не может

быть сделан меньшим, чем это необходимо для размещения всех его элементов управления.

59. Нажмите кнопку **Закрыть**. Макет диалогового окна закроется.

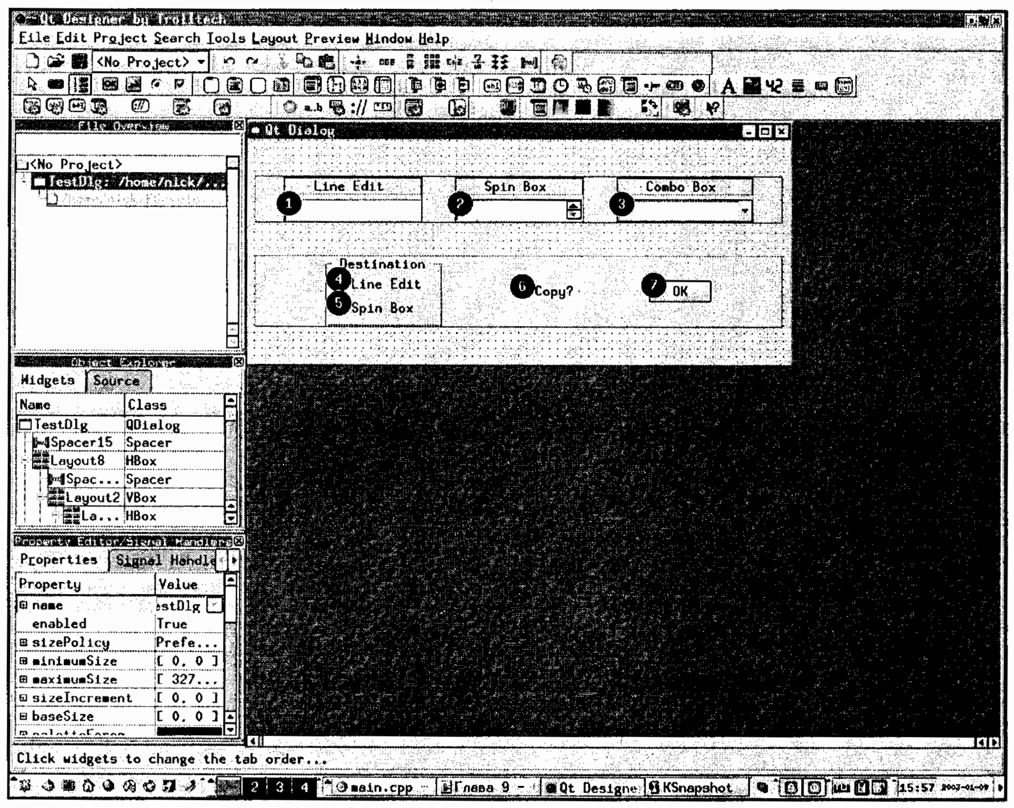


Рис. 2.14. Отображение порядковых номеров элементов управления

Хотя работа с приложением Qt Designer еще не закончена, у нас набралось достаточно материала для обсуждения.

Разработчику, имеющему представление о среде программирования Visual Studio, описанный выше процесс во многом покажется знакомым. Основным отличием в процессе включения в заготовку диалогового окна новых элементов управления является то, что они берутся не из панели, а из меню или из различных панелей инструментов, а также то, что эти элементы управления помещаются в заготовку диалогового окна не перетаскиванием, а щелчком левой кнопки мыши. Следует признать, что это не такие уж большие отличия, к которым легко привыкнуть.

Среди разработчиков Linux еще бытует стойкое убеждение в том, что нормальные программисты создают свои диалоговые окна исключительно в

текстовых редакторах, а все эти графические редакторы диалоговых окон созданы исключительно для новичков. Однако мне, привыкшему работать с графическими редакторами, с трудом верится, что человек может, глядя на ряды скупых цифр, полностью воссоздать в своем воображении диалоговое окно и понять, какие значения следует скорректировать для того, чтобы это окно приняло надлежащий вид. Если же вы не относитесь к подобным уникумам, вам все равно придется сначала нарисовать диалоговое окно на бумаге, затем определить по этому наброску координаты элементов диалогового окна, написать соответствующую программу, запустить ее на исполнение и, увидев, как ужасно все получилось, срочно вносить в нее исправления.

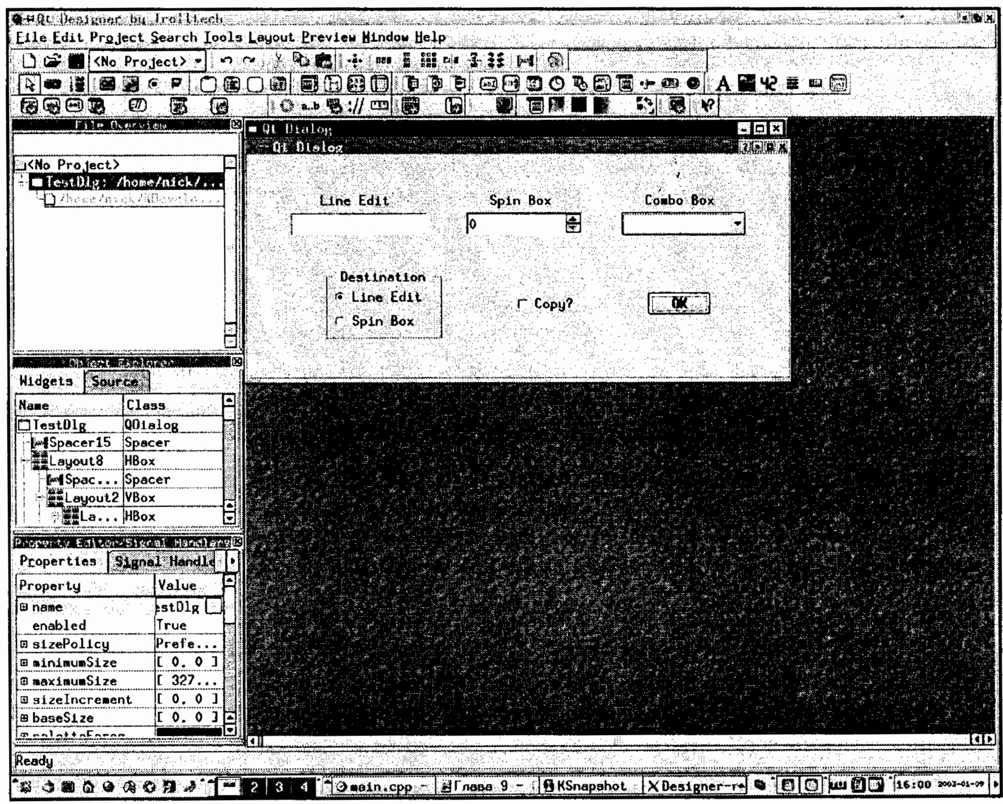


Рис. 2.15. Предварительный просмотр диалогового окна

Конечно, если у вас мазохистские наклонности и масса свободного времени для их удовлетворения, то — вперед и с песней, однако большинство, я думаю, предпочтет сразу видеть результат своих трудов и иметь возможность наиболее простыми способами корректировать этот результат.

Диалоговые окна, создаваемые в приложении Qt Designer, имеют одно существенное отличие от диалоговых окон Windows — возможность изменения их размеров в процессе работы приложения и связанное с этим автоматическое выравнивание их элементов управления. Возможность динамического изменения размеров диалогового окна является очень полезным свойством использующих их приложений, особенно если при этом обеспечена возможность изменения размеров некоторых элементов управления данного диалогового окна.

В среде программирования Visual Studio существуют некоторые средства для выравнивания положения элементов управления в диалоговом окне, однако эти средства являются статическими, т. е. работают только в редакторе диалогового окна. Это связано с тем, что диалоговые окна Windows имеют фиксированные размеры и в них отсутствует необходимость в использовании динамического выравнивания элементов управления.

Учитывая, что среди разработчиков Linux все еще находятся приверженцы программирования в текстовых редакторах, принципиально исключающих использование средств выравнивания положения элементов управления в диалоговых окнах, ниже будут приведены аргументы в пользу как статического, так и динамического выравнивания элементов.

- С увеличением числа элементов в диалоговом окне задача красивого и рационального их размещения существенно усложняется и перемещение одного какого-либо элемента или изменение размеров диалогового окна часто приводит к необходимости изменения размеров и положения других элементов управления. Статическое выравнивание позволяет существенно сократить число необходимых для этого операций.
- Изменение размеров формы без динамического выравнивания элементов приводит к существенному ухудшению их внешнего вида, поскольку все неперемещаемые элементы управления формы остаются в левом верхнем ее углу, а в самой форме появляются пустые места.
- Динамическое выравнивание элементов управления позволяет также решить очень серьезную проблему, возникающую при локализации приложений. Как правило, большинство локализуемых приложений изначально разрабатываются на английском языке, отличающемся особой краткостью. Поэтому при переводе большинства надписей на другие языки их размер увеличивается. Поскольку каждому элементу управления в диалоговом окне отводится определенное место, увеличение размера надписи может привести к ее усечению или, при отсутствии такового, к налезанию ее на другой элемент управления. Даже в том случае, если новая надпись помещается полностью, изменение ее размера приводит к нарушению симметрии диалогового окна.
- Та же самая проблема возникнет и в том случае, когда пользователю предоставляется возможность выбора используемых в приложении шрифтов.

- Динамическое выравнивание элементов позволяет выводить одно и то же диалоговое окно с использованием различных стилей.

Однако за все приходится платить: используемый в приложении Qt Designer метод автоматического выравнивания не позволяет разработчику установить свои размеры элементов управления в создаваемой заготовке.

Теперь, когда мы убедились, что динамическое выравнивание элементов управления полезно, разберемся в том, как оно реализуется. Основным средством, используемым для выравнивания элементов управления, является разделитель. Этот элемент управления является невидимым и может рассматриваться как пружина, перемещающая разделяемые им элементы управления как можно дальше друг от друга. Если элемент управления с обеих сторон окружен разделителями, то он центрируется относительно окружающих его элементов управления, в качестве которых могут выступать и границы родительского окна. Как мы уже видели, разделители бывают горизонтальные и вертикальные.

Совет

При выравнивании элементов управления диалогового окна в большинстве случаев сначала производится выравнивание по строкам (т. е. выравнивание по горизонтали), а затем выровненные строки выравниваются по столбцам (т. е. по вертикали).

При описании принципов работы разделителя было упомянуто, что одной из его "точек упора" может быть граница родительского окна выравниваемого элемента управления. Дело в том, что диалоговое окно Linux, в отличие от диалогового окна Windows, имеет не двухуровневую, а многоуровневую структуру. Если в Windows все элементы управления являются дочерними окнами диалогового окна, то в Linux некоторые элементы управления, например переключатели, могут быть дочерними окнами других элементов управления, например рамки переключателей, и глубина этой иерархии не ограничена (в разумных пределах). Убедиться в этом можно, исследовав иерархию объектов в панели **Object Explorer** (Панель объектов), изображенной на рис. 2.16, как отдельное окно.

Для вывода данной панели на экран или удаления ее с экрана используется контекстное меню, появляющееся при щелчке правой кнопкой мыши по любой панели приложения (включая ее панели инструментов).

Если бы выравнивание элементов управления в данном диалоговом окне производилось после включения в него всех элементов управления, то до начала этой операции в окне существовала бы трехуровневая иерархия объектов. Причем нижний уровень иерархии образовывали бы два переключателя, являвшиеся дочерними окнами групповой рамки. Подобная иерархия объектов автоматически создается приложением Qt Designer при рисовании групповой рамки, охватывающей группу флажков или переключателей, и при помещении этих элементов управления в групповую рамку.

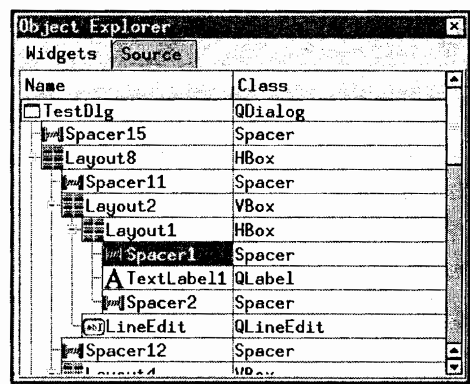


Рис. 2.16. Диалоговое окно **Qbject Explorer**

Выравнивание элементов управления производится всегда в пределах их родительского окна. Поэтому для выравнивания переключателей по вертикали выделялась групповая рамка, являющаяся их родительским окном. Выравнивание переключателей в групповой рамке было произведено без использования разделителей, поскольку нас устраивал ее минимальный размер.

Все остальные выравниваемые нами элементы управления являлись непосредственными потомками диалогового окна. Поскольку для всех элементов не мог быть выбран один из стандартных методов выравнивания, для их автоматического размещения была использована другая стратегия: среди них были образованы группы, для которых можно было указать единый способ выравнивания, и каждая из этих групп была выровнена отдельно.

Так как общий принцип выравнивания в пределах родительского окна остался неизменным, при выборе соответствующей команды меню или при нажатии кнопки в панели инструментов, инициирующей процедуру выравнивания, вокруг выделенных элементов управления сначала формируется невидимая рамка выравнивания, все выделенные элементы управления становятся ее дочерними окнами и только потом для этих окон инициируется процедура выравнивания. Таким образом, после выравнивания элементов управления в диалоговом окне сформировалась пятиуровневая иерархия объектов.

Примечание

Обратите внимание на то, что при выравнивании элементов в одном направлении (например, по горизонтали) они одновременно выравниваются между собой и в другом направлении (в данном случае, по вертикали). Так, например, флажок **Copy?** центрируется по вертикали в пределах рамки выравнивания.

Теперь нам остается рассмотреть еще один важный вопрос, связанный с созданием заготовки диалогового окна — установление последовательности

передачи фокуса ввода элементам управления диалогового окна при нажатии пользователем клавиши <Tab> или комбинации клавиш <Shift>+<Tab>.

Несмотря на несомненные преимущества работы с мышью, в некоторых случаях пользователю удобно не отрываться от клавиатуры и осуществлять с нее передачу фокуса ввода элементам управления диалогового окна. Для этого все элементы управления диалогового окна, у которых свойство `focusPolicy` имеет значение `TabFocus`, объединяются в циклический список. Для передачи фокуса ввода следующему элементу этого списка используется клавиша <Tab>, а для передачи его предыдущему элементу списка — комбинации клавиш <Shift>+<Tab>.

Примечание

Если элементы управления диалогового окна еще не имеют фокуса ввода, то при первом нажатии клавиши <Tab> фокус ввода передается первому элементу списка, а при нажатии комбинации клавиш <Shift>+<Tab> — последнему элементу списка.

Диалоговые окна часто вызываются по ошибке, поэтому всегда полезно предоставить пользователю возможность закрыть только что открытое окно с применением минимальных усилий. Для этого в них могли бы быть назначены кнопки, выбираемые по умолчанию, но эти кнопки перехватывают сообщение о нажатии клавиши <Enter> независимо от наличия у них фокуса ввода, что может негативно сказаться на работе с другими элементами управления. Поэтому мы просто поставили кнопку выхода из диалогового окна первой в списке элементов управления.

Завершение создания диалогового приложения

В предыдущем разделе нами была создана заготовка диалогового окна и даже было проведено ее тестирование, но все равно эта заготовка является пока что не больше, чем красивой картинкой. Для превращения ее в полноценное диалоговое окно необходимо осуществить связь элементов управления друг с другом и с самим диалоговым окном.

Чтобы завершить процедуру создания диалогового приложения:

1. Выберите команду меню **Tools | Connect Signal/Slots** (Сервис | Связать Сигналы/Приемники), нажмите клавишу <F3> или кнопку **Connect Signal/Slots** в панели инструментов **Tools** (она должна "утопиться").
2. Поместите указатель мыши на кнопку **OK**, нажмите левую кнопку мыши и, не отпуская ее, переместите указатель мыши на свободное место в заготовке диалогового окна (за пределы рамок выравнивания). Появится диалоговое окно **Edit Connections**, изображенное на рис. 2.17.

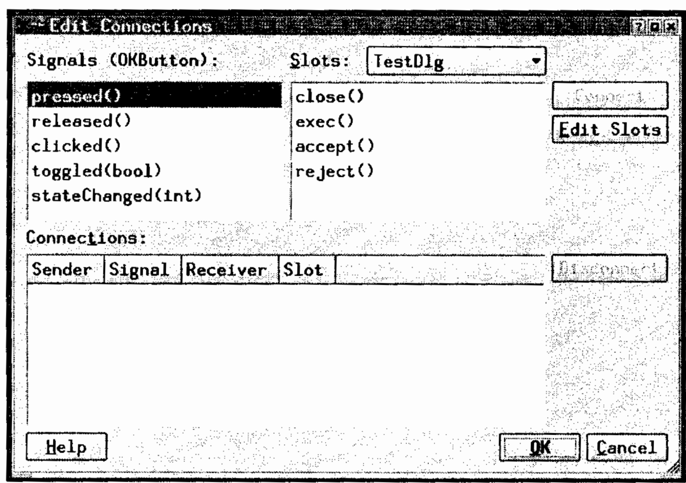


Рис. 2.17. Диалоговое окно **Edit Connections**

3. В окне списка **Signals** (Сигналы) выделите сигнал `clicked()`, в окне списка **Slots** (Приемники) — приемник `accept()`. В окне списка **Connections** (Соединения) появится информация о новом соединении.
4. Нажмите кнопку **OK**. Диалоговое окно **Edit Connections** закроется.
5. Повторите пункты 1 и 2 для групповой рамки переключателей.
6. Нажмите кнопку **Edit Slots** (Редактировать приемники). Появится диалоговое окно **Edit Slots**, изображенное на рис. 2.18.

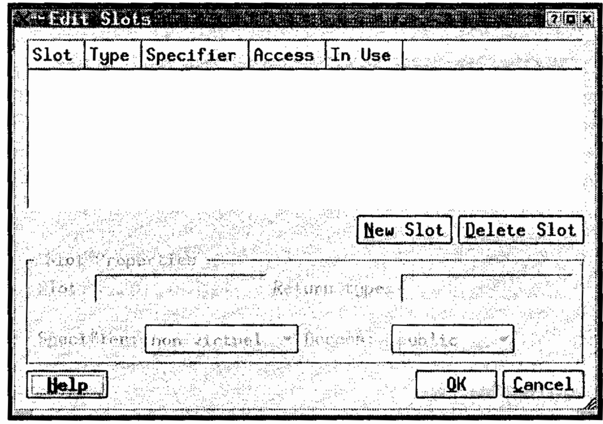


Рис. 2.18. Диалоговое окно **Edit Slots**

7. Нажмите кнопку **New Slot** (Новый приемник). В списке приемников появится новый приемник с именем `new_slot`, который сразу же будет

выделен (его имя появится в текстовом поле **Slot** группы **Slot Properties** (Свойства приемника)).

8. В текстовое поле **Slot** введите сигнатуру нового приемника `destination(int)` и нажмите кнопку **OK**. Диалоговое окно **Edit Slots** закроется и в списке **Slots** диалогового окна **Edit Connections** появится новый приемник.
9. В окне списка **Signals** выделите сигнал `clicked(int)`, в окне списка **Slots** — приемник `destination(int)`. В окне списка **Connections** появится информация о новом соединении.
10. Нажмите кнопку **OK**. Диалоговое окно **Edit Connections** закроется.
11. Повторите пункты 5—10 для флажка **Copy?**, сопоставив его сигналу `toggled(bool)` приемник диалогового окна `copyFlag(bool)`.
12. Повторите пункты 5—10 для раскрывающегося списка, сопоставив его сигналу `textChanged(const QString&)` приемник `boxCopy(const QString&)`.
13. Выберите команду меню **File | Save** (Файл | Сохранить), нажмите комбинацию клавиш <Ctrl>+<S> или нажмите кнопку **Save** в панели инструментов **File**. Появится диалоговое окно **Save Form 'TestDlg' As** (Сохранить форму TestDlg как), изображенное на рис. 2.19.

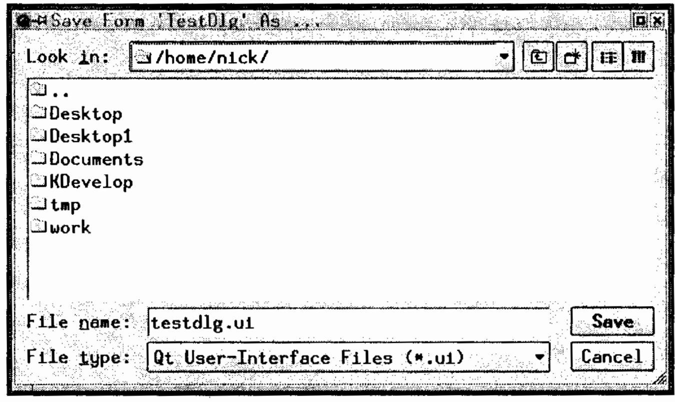


Рис. 2.19. Диалоговое окно **Save Form 'TestDlg' As**

14. Перейдите в каталог, содержащий ваш файл описания пользовательского интерфейса, и нажмите кнопку **Save** (Сохранить). Появится окно сообщения **File Already Exists**, предлагающее переписать существующий файл.
15. Нажмите кнопку **Yes**. Заготовка диалогового окна будет сохранена.
16. Закройте приложение Qt Designer by Trolltech.

17. В среде разработки KDevelop выберите команду меню **Build | Make** (Создать | Компилировать), нажмите клавишу <F8> или кнопку **Make** в панели инструментов. Приложение будет откомпилировано.

18. В окне иерархических списков (**Tree Tool Views**) раскройте вкладку **Classes** (Классы), щелкните правой кнопкой мыши на каталоге **Classes** и выберите в появившемся контекстном меню команду **New class** (Новый класс). Появится диалоговое окно **Class Generator**, изображенное на рис. 2.20.

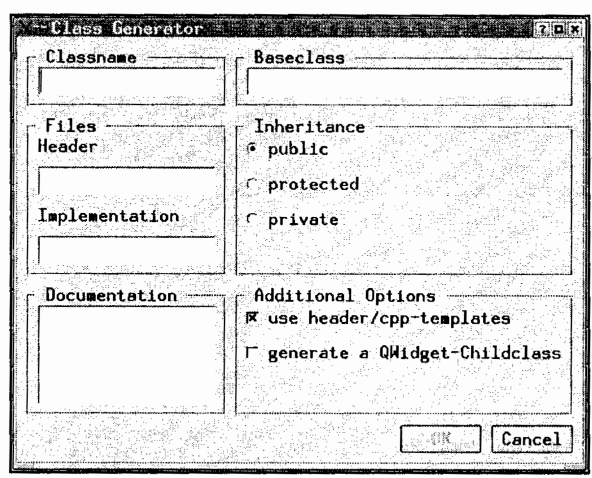


Рис. 2.20. Диалоговое окно **Class Generator**

19. В текстовое поле **Classname** (Имя класса) введите имя класса `TestDlgImpl`, в текстовое поле **Baseclass** (Имя базового класса) — имя базового класса `TestDlg`, в текстовое поле **Documentation** (Документация) введите комментарий `User dialog class`, в группе **Additional Options** (Дополнительные свойства) установите флажок **generate a QWidget-Childclass** (Создать потомка класса `QWidget`) и нажмите кнопку **OK**. В проект будут добавлены файлы заголовка и реализации, содержащие заготовку нового класса, и будут открыты окна редактирования этих файлов.

20. В появившемся после создания класса окне редактирования файла testdlgimpl.h измените заголовок класса `TestDlgImpl` в соответствии с текстом листинга 2.1.

Листинг 2.1. Заголовок класса TestDlgImpl

```
class TestDlgImpl : public TestDlg
{
  Q_OBJECT
```

```
public:
 TestDlgImpl(QWidget *parent=0, const char *name=0,
                  bool modal = true, WFlags fl = 0);
 ~TestDlgImpl();

public slots:
 virtual void copyFlag(bool);
 virtual void boxCopy(const QString&);
 virtual void destination(int);

private:
 bool copy;
 int  dest;
};
```

21. Откройте окно редактирования файла testdlgimpl.cpp и измените реализацию класса `TestDlgImpl` в соответствии с текстом листинга 2.2.

Листинг 2.2. Реализация класса `TestDlgImpl`

```
#include <qlineedit.h>
#include <qspinbox.h>
#include <qcombobox.h>

TestDlgImpl::TestDlgImpl( QWidget *parent, const char *name, bool
modal, WFlags fl)
  : TestDlg( parent, name, modal, fl)
{
 LineEdit->setText("32");
 SpinBox->setValue(24);
 ComboBox->insertItem("8");
 ComboBox->insertItem("32");
 ComboBox->insertItem("256");

 copy = false;
 dest = 0;
}
TestDlgImpl::~TestDlgImpl()
{
}

void TestDlgImpl::copyFlag(bool b)
{
 copy = b;
}
```

```
/** Копирует информацию из раскрывающегося списка */
void TestDlgImpl::boxCopy(const QString& sz)
{
 if( copy)
 {
  switch( dest)
  {
   case 0: LineEdit->setText(sz);
            break;
   case 1: SpinBox->setValue(sz.toInt());
          break;
  }
 }
}

/** Определяет получателя информации */
void TestDlgImpl::destination(int button)
{
 dest = button;
}
```

22. Откройте окно редактирования файла main.cpp и после строки `#include "dialog.h"` поместите строку

```
#include "testdlgimpl.h"
```

23. Измените главную функцию приложения в соответствии с текстом листинга 2.3.

Листинг 2.3. Функция `main`

```
int main(int argc, char *argv[])
{
 KAboutData aboutData( "dialog", I18N_NOOP("Dialog"),
  VERSION, description, KAboutData::License_GPL,
  "(c) 2002, Nick", 0, 0, "Nick@localhost");
 aboutData.addAuthor("Nick",0, "Nick@localhost");
 KCmdLineArgs::init( argc, argv, &aboutData);
 KCmdLineArgs::addCmdLineOptions( options); // Добавьте собственные
                                             // опции

 KApplication a;

 TestDlgImpl dlg;
 a.setMainWidget( &dlg);

 return dlg.exec();
}
```

24. Выберите команду меню **Debug | Start** (Отладка | Пуск) или нажмите кнопку **Debug** в панели инструментов. Появится окно приложения, изображенное на рис. 2.21.

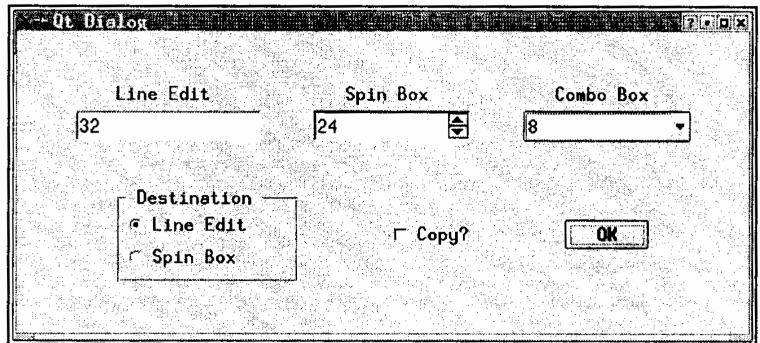


Рис. 2.21. Окно диалогового приложения

25. В раскрывающемся списке **Combo Box** выделите строку **256**. Она появится в текстовом поле раскрывающегося списка.
26. Измените значение в текстовом поле раскрывающегося списка.
27. Установите флажок **Copy?** и повторите пункт 24. Новое значение появится не только в текстовом поле раскрывающегося списка, но и в текстовом поле **Line Edit**.
28. Установите переключатель **Spin Box** в группе **Destination** и повторите пункт 24, выделив в раскрывающемся списке строку **32**. Новое значение появится не только в текстовом поле раскрывающегося списка, но и в текстовом поле инкрементного регулятора.
29. Нажмите на верхнюю кнопку инкрементного регулятора. Значение в его текстовом поле увеличится на 1.
30. Нажмите на нижнюю кнопку инкрементного регулятора. Значение в его текстовом поле уменьшится на 1.
31. Нажмите кнопку **OK**. Диалоговое окно и приложение закроются.

Концепция сигналов и приемников была уже подробно рассмотрена нами в *главе 1*, поэтому мы не будем здесь на ней подробно останавливаться. Единственное, что здесь нужно понять, это то, что приложение Qt Designer позволяет устанавливать связи как между отдельными элементами управления диалогового окна, так и между каждым из них и самим диалоговым окном.

Поскольку элементы управления диалогового окна описываются стандартными классами, то пользователь не может изменить набор их приемников. Поэтому при связи элементов управления диалогового окна друг с другом или с самим собой могут быть использованы только стандартные приемни-

ки. Класс диалогового окна создается вместе с самим диалоговым окном. Поэтому в него могут быть включены любые необходимые разработчику приемники. Добавление в класс диалогового окна сигналов бессмысленно, поскольку элементы управления диалогового окна не имеют для них приемников.

Взаимодействие между элементами управления диалогового окна демонстрируется использованием элемента управления групповой рамки кнопок, организующей взаимодействие помещенных в нее переключателей и формирующей специальные сигналы, предоставляющие интегрированную информацию об их состоянии.

В результате работы приложения Qt Designer создается файл описания интерфейса (файл с расширением ui), который не может быть непосредственно включен в приложение C++. Этот файл необходимо откомпилировать утилитой uic, формирующей на его основе файлы заголовка и реализации класса диалогового окна.

Поскольку эти файлы будут создаваться заново при каждой перекомпиляции приложения, нет никакого смысла вносить в них какие-либо изменения. Эти файлы даже не включаются в проект. Тем не менее вносить изменения в класс диалогового окна нужно, т. к. в него включены, но не реализованы несколько приемников. Для разрешения этого противоречия в приложении создается класс, производный от класса диалогового окна, созданного приложением Qt Designer. В этом классе и реализуются все приемники класса диалогового окна.

Совет

При разработке больших проектов требуется обеспечить единообразие именования классов, существенно повышающее его читабельность. Разработчики приложения Qt Designer предлагают формировать имя класса, в котором реализуются приемники исходного класса, путем конкатенации имени исходного класса и суффикса Impl, как это сделано в рассматриваемом приложении.

Утилита uic при создании файла заголовка диалогового окна не включает в него файлы заголовков для его элементов управления. Она включает только объявления используемых ею классов элементов управления. Это связано с принятым в приложениях KDevelop стилем программирования, требующего всегда, когда это возможно, избегать включения файлов заголовков в другие файлы заголовков. Такое требование объясняется тем, что при внесении изменений в файлы заголовков необходимо перекомпилировать все файлы, включающие эти файлы заголовков. Кроме того, такой стиль программирования позволяет избежать включения в файлы реализации ненужных ему файлов заголовков.

Файл реализации диалогового окна создавался нами для перегрузки виртуальных приемников класса диалогового окна, созданного утилитой uic. По-

этому в заголовок созданного нами класса необходимо добавить объявления перегружаемых приемников и переменные, необходимые для их работы.

Совет

Файлы заголовка и реализации производного класса содержат только его конструктор и деструктор, к тому же конструктор класса, по непонятным причинам, имеет сокращенный набор аргументов, не позволяющий сделать диалоговое окно модальным. Поэтому большинство разработчиков предпочитают при внесении изменений в эти файлы сначала скопировать в них информацию, содержащуюся в файлах заголовка и реализации базового класса. Эти файлы содержатся в каталоге dialog во вкладке **Files** окна раскрывающихся списков (в этом каталоге хранятся файлы реализации и заголовков всех классов приложения).

В класс реализации диалогового окна включаются две переменные, первая из которых имеет логический тип и служит для хранения текущего состояния флажка **Copy?**, а вторая — имеет целочисленный тип и служит для хранения номера отмеченного переключателя. Значение первой переменной изменяется в приемнике `TestDlgImpl::copyFlag`, принимающем сигнал от флажка, передающий информацию о его новом состоянии, а значение второй переменной — в приемнике `TestDlgImpl::destination`, принимающем сигнал от групповой рамки окон.

Приемник `TestDlgImpl::boxCopy` принимает сигнал об изменении содержимого текстового поля раскрывающегося списка, передающий новое его содержимое. В этом приемнике проверяется, следует ли копировать это содержимое куда-нибудь, если следует, то проверяет куда именно и производит копирование. Для копирования информации в текстовое поле используется функция `QLineEdit::setText`, а для копирования в текстовое поле инкрементного регулятора — функция `QSpinBox::setValue`. Поскольку аргумент функции `setValue` имеет целочисленный тип, выводимая строка преобразуется к этому типу вызовом функции `QString::toInt`.

Для того чтобы пользователь мог сразу же приступить к работе с приложением, все элементы управления диалогового окна инициализируются в конструкторе его класса. При этом строка помещается в текстовое поле вызовом функции `setText`, начальное значение инкрементного регулятора устанавливается вызовом функции `setValue`, а раскрывающийся список заполняется вызовом функции `QComboBox::insertItem`.

Внимание!

Обратите внимание на то, что сигнатура конструктора класса была изменена, и в ней были восстановлены все аргументы конструктора класса `QDialog`. Кроме того, в заголовке класса у третьего аргумента конструктора было изменено используемое по умолчанию значение. Теперь по умолчанию создается модальное диалоговое окно.

3 Зак. 113

Теперь диалоговое окно готово к работе, но остались еще вопросы с его вызовом. Поскольку нами создается диалоговое приложение, то вызов диалогового окна будет производиться в главной функции приложения, и оно станет его основным окном.

В главной функции приложения, после создания никому не нужного класса диалогового окна **About** и чтения аргументов командной строки, создается объект класса `KApplication`, являющийся главным объектом класса приложения. После этого создается объект класса нашего диалогового окна. Конструктор этого класса не имеет аргументов, поскольку все их значения задаются по умолчанию.

Вызовом функции `QApplication::setMainWidget` объект класса диалогового окна назначается объектом класса главного окна приложения и вызовом функции `QDialog::exec` диалоговое окно вводится на экран. Поскольку это диалоговое окно является модальным, то его функция `exec` не возвратит значения до тех пор, пока это диалоговое окно не будет закрыто. Возвращаемое значение этой функции является возвращаемым значением приложения.

Внимание!

В отличие от диалоговых окон Windows, являющихся в большинстве своем модальными диалоговыми окнами, диалоговые окна библиотеки Qt являются по умолчанию немодальными диалоговыми окнами, а при создании дочерних классов от классов диалогового окна, созданных утилитой uic, среда программирования KDevelop вообще исключает возможность создания модальных диалоговых окон. Для вызова этого диалогового окна как модального разработчику необходимо восстановить, по крайней мере, третий аргумент конструктора класса реализации диалогового окна.

Создание специализированных диалоговых окон

В диалоговом окне **New File** появляющегося при запуске приложения Qt Designer by Trolltech из среды разработки KDevelop, помимо использованного нами шаблона **Dialog**, существуют и другие шаблоны диалоговых окон. Некоторые из них просто добавляют в создаваемое диалоговое окно стандартные кнопки и связывают их сигналы со стандартными приемниками диалогового окна. Другие же шаблоны используются для создания специализированных диалоговых окон, которые и будут рассмотрены в этом разделе.

Создание диалогового окна с вкладками

Пользователям операционных систем, имеющих графический интерфейс, хорошо знакомы диалоговые окна с вкладками, позволяющими разработчи-

ку сгруппировать предоставляемую пользователю информацию по темам, что существенно повышает объем информации, который может быть передан пользователем через одно диалоговое окно.

Для разъяснения принципов разработки таких окон нами будет создано демонстрационное приложение, текст которого можно найти на прилагаемом к книге CD.

Чтобы самостоятельно создать диалоговое приложение с вкладками:

1. Выберите команду меню **Project | New**. Появится диалоговое окно **ApplicationWizard**.
2. В окне иерархического списка диалогового окна в папке **KDE** выделите строку **KDE Mini** и нажмите кнопку **Next**. Появится второе окно мастера **ApplicationWizard**.
3. В текстовое поле **Project name** введите имя проекта **TabDialog** и нажмите кнопку **Create**. Появится последнее окно мастера **ApplicationWizard** и начнется процесс создания заготовки приложения.
4. Как только станет доступной кнопка **Exit**, нажмите ее и выйдите из мастера создания приложений.
5. Выберите команду меню **File | New**, нажмите комбинацию клавиш <Ctrl>+<N> или кнопку **New** в панели инструментов. Появится диалоговое окно **New File**.
6. В окне списка раскрытой вкладки **General** выделите строку **Qt Designer File (*.ui)**, в текстовое поле **Filename** введите имя файла `tabdlg` и нажмите кнопку **OK**. Появится диалоговое окно **Load decision**.
7. Нажмите кнопку **No**. Откроется окно приложения Qt Designer by Trolltech.
8. В окне списка появившегося при старте приложения диалогового окна **New File** выделите значок **Tab Dialog** и нажмите кнопку **OK**. В приложении появится пустая заготовка диалогового окна с двумя вкладками, а в панели **Property Editor/Signal Handlers** появится список свойств созданного окна.
9. В панели **Property Editor/Signal Handlers** выделите строку **name** и замените содержащееся в нем имя диалогового окна **Form1** именем **TabtDlg**.
10. В той же панели выделите строку **caption** и замените содержащийся в нем заголовок диалогового окна **Form1** заголовком **Tab Dialog**.
11. Щелкните левой кнопкой мыши на раскрытой вкладке, выделите в панели **Property Editor/Signal Handlers** строку **pageTitle** и замените содержащийся в нем заголовок вкладки **Tab** заголовком **First**.
12. В той же панели выделите строку **pageName** и введите в нее имя вкладки **firstTab**.

13. Раскройте вторую вкладку и повторите в ней пункты 11 и 12, установив для нее заголовок **Second** и имя **secondTab**.
14. В панели **Object Explorer** раскройте вкладку **Widgets** (Окна) (если она еще не раскрыта), щелкните правой кнопкой мыши по каталогу **tabWidget** и выберите в появившемся контекстном меню команду **Add Page**. В каталоге и в заготовке диалогового окна появится новая раскрытая вкладка.
15. Повторите в ней пункты 11 и 12, установив для новой вкладки заголовок **Third** и имя **thirdTab**.
16. Раскройте первую вкладку в заготовке диалогового окна, выберите команду меню **Tools | Display | TextLabel** или нажмите кнопку **Text Label** в панели инструментов **Display** и щелкните левой кнопкой мыши в центре заготовки диалогового окна. Появится рамка статического текста.
17. В панели **Property Editor/Signal Handlers** выделите строку **text** и введите в связанное с ней текстовое поле строку **Enter some text** (Введите какой-нибудь текст).
18. Выберите команду меню **Tools | Input | LineEdit** или нажмите кнопку **LineEdit** в панели инструментов **Input** и щелкните левой кнопкой мыши под только что введенным статическим текстом. На месте щелчка появится текстовое поле.
19. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя текстового поля на **firstLineEdit**.
20. Выберите команду меню **Layout | Add Spacer** или нажмите кнопку **Spacer** в панели инструментов **Layout**.
21. Щелкните левой кнопкой мыши слева от статического текста **Enter some text** и выберите в появившемся контекстном меню команду **Horizontal**. На месте щелчка появится горизонтальный разделитель.
22. Повторите пункты 20 и 21, щелкнув левой кнопкой мыши справа от статического текста **Enter some text**.
23. Щелкните левой кнопкой мыши на левом горизонтальном разделителе. Он будет выделен.
24. Нажмите клавишу <Shift> и, удерживая ее, щелкните левой кнопкой мыши на статическом тексте и правом разделителе. Оба разделителя и статический текст будут выделены.

Примечание

При работе с вкладкой диалогового окна выделение рамкой невозможно, поскольку при попытке рисования рамки выделяется сама вкладка.

25. Выберите команду меню **Layout | Lay Out Horizontally**, нажмите комбинацию клавиш <Ctrl>+<H> или кнопку **Lay Out Horizontally** в панели

инструментов **Layout**. Текстовое поле и два разделителя будут автоматически выровнены по горизонтали.

26. Повторите пункты 23 и 24 для одновременного выделения группы статического текста и текстового поля.
27. Выберите команду меню **Layout | Lay Out Vertically**, нажмите комбинацию клавиш <Ctrl>+<L> или кнопку **Lay Out Vertically** в панели инструментов **Layout**. Текстовое поле и его заголовок будут автоматически выровнены по вертикали.
28. Установите вертикальные разделители над заголовком текстового поля и под текстовым полем.
29. Установите горизонтальные разделители слева и справа от текстового поля, щелкните левой кнопкой мыши за пределами рамок элементов управления и выберите команду меню **Layout | Lay Out in a Grid** (Расположение | Расположить в сетке), нажмите комбинацию клавиш <Ctrl>+ +<G> или кнопку **Lay Out in a Grid** в панели инструментов **Layout**. Текстовое поле и его заголовок будут помещены в центр вкладки, как это показано на рис. 2.22.
30. Раскройте вторую вкладку и повторите для нее пункты 16—29, присвоив текстовому полю имя **secondLineEdit**.
31. Раскройте третью вкладку и повторите для нее пункты 16—29, присвоив текстовому полю имя **thirdLineEdit**.
32. Щелкните левой кнопкой мыши на свободном поле заготовки диалогового окна и выберите команду меню **Layout | Break Layout** (Расположение | Отменить расположение), нажмите комбинацию клавиш <Ctrl>+ +<B> или кнопку **Break Layout** в панели инструментов **Layout**.
33. Выделите вкладку, поместите указатель мыши на синий прямоугольник, расположенный на ее нижней границе, нажмите левую кнопку мыши и переместите вверх нижнюю границу вкладок.
34. Выберите команду меню **Tools | Buttons | CheckBox** или нажмите кнопку **Check Box** в панели инструментов **Buttons** и создайте флажок под вкладками.
35. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя флажка на **syncCheck**.
36. В той же панели выделите строку **text** и введите в связанное с ней текстовое поле строку **Synchronize?** (Синхронизировать?).
37. Поместите горизонтальные разделители слева и справа от флажка, выделите флажок и разделители и выберите команду меню **Layout | Lay Out Horizontally**, нажмите комбинацию клавиш <Ctrl>+<H> или кнопку **Lay Out Horizontally** в панели инструментов **Layout**. Флажок и два разделителя будут автоматически выровнены по горизонтали.

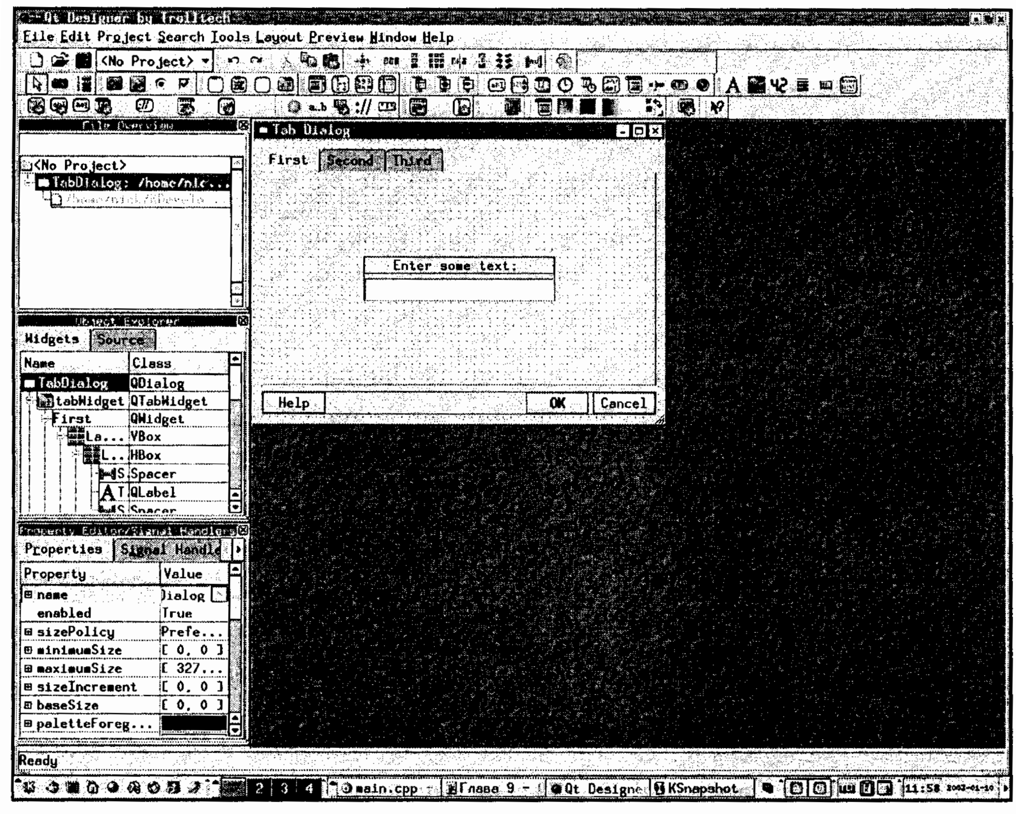


Рис. 2.22. Заготовка первой вкладки

38. Щелкните левой кнопкой мыши на свободном поле заготовки диалогового окна и выберите команду меню **Layout | Lay Out Vertically**, нажмите комбинацию клавиш <Ctrl>+<L> или кнопку **Lay Out Vertically** в панели инструментов **Layout**. Элементы диалогового окна будут выровнены по вертикали, как это показано на рис. 2.23.

39. Раскройте первую вкладку заготовки диалогового окна и выберите команду меню **Tools | Connect Signal/Slots**, нажмите клавишу <F3> или кнопку **Connect Signal/Slots** в панели инструментов **Tools**.

40. Поместите указатель мыши на текстовое поле, нажмите левую кнопку мыши и, не отпуская ее, переместите указатель мыши на свободное место в заготовке диалогового окна (за пределами вкладки). Появится диалоговое окно **Edit Connections**.

41. Нажмите кнопку **Edit Slots**. Появится диалоговое окно **Edit Slots**.

42. Нажмите кнопку **New Slot**, введите в текстовое поле **Slot** имя нового приемника `textChanged(const QString&)` и нажмите кнопку **OK**. Диало-

говое окно **Edit Slots** закроется и в списке **Slots** диалогового окна окно **Edit Connections** появится новый приемник.

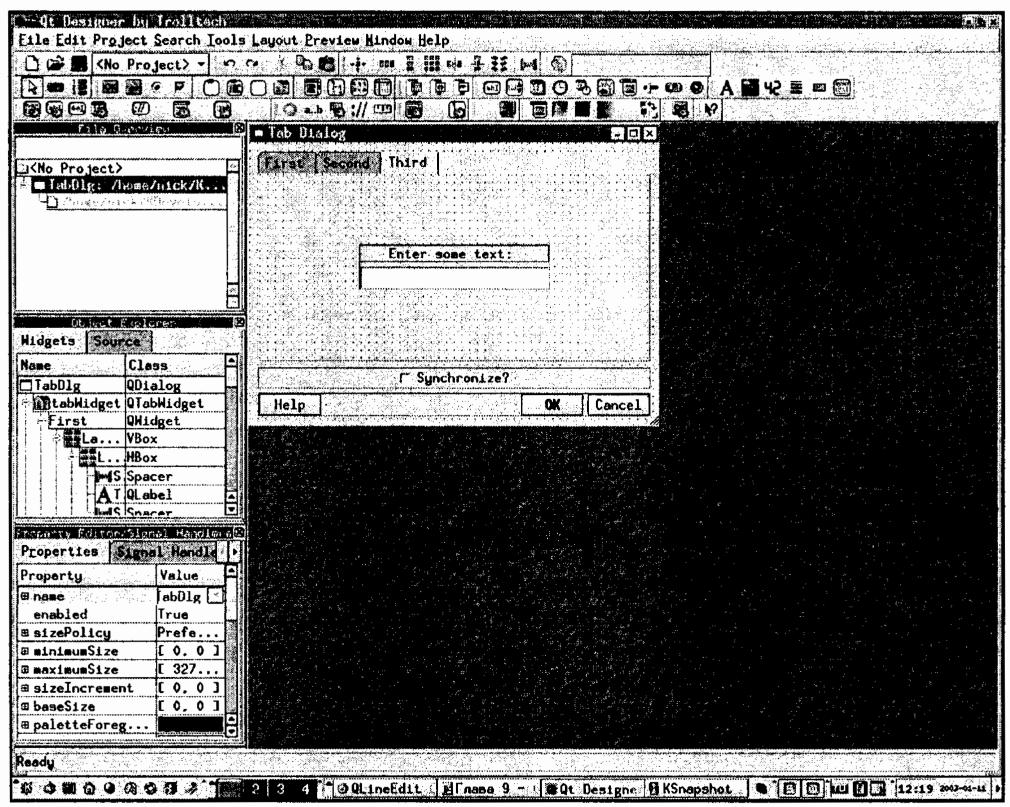


Рис. 2.23. Заготовка диалогового окна с вкладками

43. В окне списка **Signals** выделите сигнал `textChanged(const QString&)`, в окне списка **Slots** выделите приемник `textChanged(const QString&)`. В окне списка **Connections** появится информация о новом соединении.
44. Нажмите кнопку **OK**. Диалоговое окно **Edit Connections** закроется.
45. Откройте вторую вкладку заготовки диалогового окна и повторите для нее пункты 39, 40, 43 и 44.
46. Откройте третью вкладку заготовки диалогового окна и повторите для нее пункты 39, 40, 43 и 44.
47. Выберите команду меню **Tools | Connect Signal/Slots**, нажмите клавишу <F3> или кнопку **Connect Signal/Slots** в панели инструментов **Tools**.
48. Поместите указатель мыши на свободное поле вкладки, нажмите левую кнопку мыши и, не отпуская ее, переместите указатель мыши на сво-

бодное место в заготовке диалогового окна. Появится диалоговое окно **Edit Connections**.

49. Нажмите кнопку **Edit Slots**. Появится диалоговое окно **Edit Slots**.

50. Нажмите кнопку **New Slot**, введите в текстовое поле **Slot** имя нового приемника `pageSelected(const QString&)` и нажмите кнопку **OK**. Диалоговое окно **Edit Slots** закроется и в списке **Slots** диалогового окна **Edit Connections** появится новый приемник.

51. В окне списка **Signals** выделите сигнал `selected(const QString&)`, в окне списка **Slots** — приемник `pageSelected(const QString&)`. В окне списка **Connections** появится информация о новом соединении.

52. Нажмите кнопку **OK**. Диалоговое окно **Edit Connections** закроется.

53. Сохраните созданную заготовку в файле описания пользовательского интерфейса приложения и закройте приложение Qt Designer by Trolltech.

54. В окне иерархических списков среды разработки KDevelop раскройте вкладку **Classes**, щелкните правой кнопкой мыши на каталоге **Classes** и выберите в появившемся контекстном меню команду **New class**. Появится диалоговое окно **Class Generator**.

55. В текстовое поле **Classname** введите имя класса `TabDlgImpl`, в текстовое поле **Baseclass** — имя базового класса `TabDlg`, в текстовое поле **Documentation** — комментарий `Dialog implementation class` (Класс реализации диалогового окна), в группе **Additional Options** установите флажок **generate a QWidget-Childclass** и нажмите кнопку **OK**. В проект будут добавлены файлы заголовка и реализации, содержащие заготовку нового класса, и будут открыты окна редактирования этих файлов.

56. В появившемся после создания класса окне редактирования файла testdlgimpl.h замените строку `TabDlgImpl(QWidget *parent=0, const char *name=0);` строкой

```
TabDlgImpl(QWidget *parent=0, const char *name=0, bool modal = false);
```

57. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `TabDlgImpl` и выберите в появившемся контекстном меню команду **Add slot**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Slots**, как это показано на рис. 2.24.

58. В текстовое поле **Declaration** (Объявление) введите сигнатуру приемника `textChanged(const QString&)`, установите флажок **Virtual** (Виртуальный) в группе **Modifiers** (Модификаторы), в текстовое поле **Documentation** введите комментарий `Saves user input` (Сохраняет введенную информацию) и нажмите кнопку **Apply** (Применить). В класс будет добавлен новый приемник.

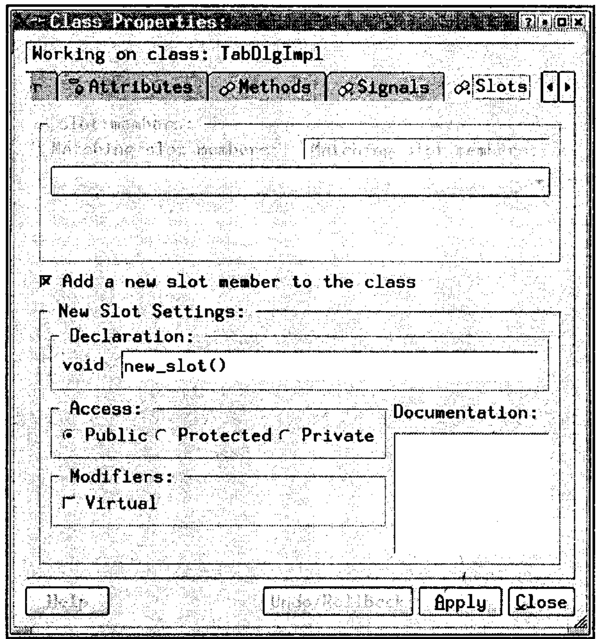


Рис. 2.24. Добавление приемника

59. Повторите пункты 57 и 58 для добавления в класс приемника `pageSelected(const QString&)`, снабдив его комментарием `Updates line edits` (Изменяет содержимое текстовых полей).

60. В открывшемся после добавления приемников окне редактирования файла tabdlgimpl.cpp измените реализацию класса `TabDlgImpl` в соответствии с текстом листинга 2.4.

Листинг 2.4. Реализация класса `TabDlgImpl`

```
#include "tabdlgimpl.h"
#include <qlineedit.h>
#include <qcheckbox.h>

TabDlgImpl::TabDlgImpl(QWidget *parent, const char *name, bool modal)
  : TabDlg(parent,name, modal)
{
}

TabDlgImpl::~TabDlgImpl()
{
}
```

```
/** Сохраняет введенную информацию */
void TabDlgImpl::textChanged(const QString& sz)
{
 szTemp = sz;
}
/** Изменяет содержимое текстовых полей */
void TabDlgImpl::pageSelected(const QString& pageName)
{
 if( syncCheck-> isChecked())
  if( pageName == "First")
   firstLineEdit-> setText( szTemp);
  else
   if( pageName == "Second")
    secondLineEdit-> setText( szTemp);
   else
    if( pageName == "Third")
     thirdLineEdit-> setText( szTemp);
}
```

61. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `TabDlgImpl` и выберите в появившемся контекстном меню команду **Add member variable**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes**, как это показано на рис. 2.25.

62. В текстовое поле **Type** введите тип переменной `QString`, в текстовое поле **Name** введите идентификатор переменной `szTemp`, установите переключатель **Private** в группе **Access**, в текстовое поле **Documentation** введите комментарий `Temporary string` (Строка для хранения временной информации) и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.

63. Откройте окно редактирования файла main.cpp и после строки `#include "tabdialog.h"` вставьте строку

    ```
    #include "tabdlgimpl.h"
    ```

64. Измените функцию `main` в соответствии с текстом листинга 2.5.

Листинг 2.5. Функция main

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

 KAboutData aboutData( "tabdialog", I18N_NOOP("TabDialog"),
  VERSION, description, KAboutData::License_GPL,
  "(c) 2003, ", 0, 0, "");
```

```
    aboutData.addAuthor("",0, "");
    KCmdLineArgs::init( argc, argv, &aboutData);
    KCmdLineArgs::addCmdLineOptions( options); // Добавьте собственные
                                                // опции

    KApplication a;

    TabDlgImpl dlg;
    a.setMainWidget( &dlg);

    return dlg.exec();
}
```

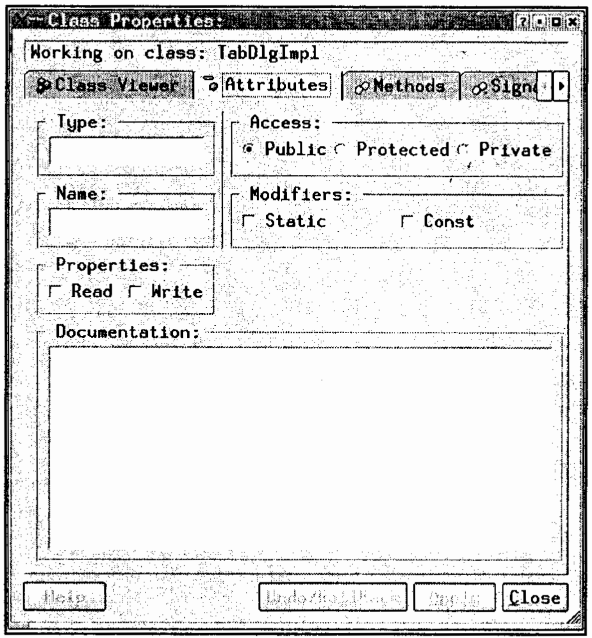


Рис. 2.25. Добавление переменной

65. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Появится окно приложения, изображенное на рис. 2.26.
66. В текстовое поле **Enter some text** первой вкладки введите какой-нибудь текст и раскройте вторую вкладку. Ее текстовое поле будет пусто.
67. Установите флажок **Synchronize?** и раскройте третью вкладку. В ее текстовом поле будет выведен тест, введенный в первой вкладке.

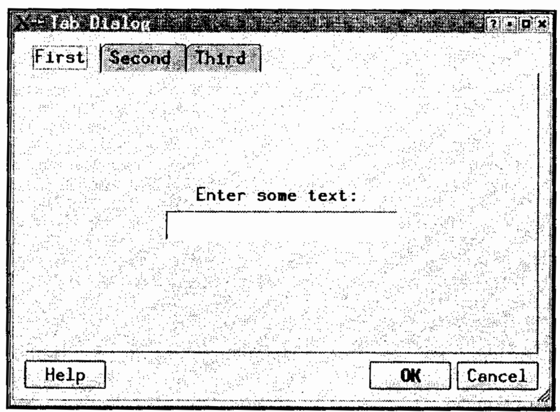


Рис. 2.26. Окно диалогового приложения с вкладками

68. Измените текст в текстовом поле **Enter some text** и раскройте вторую вкладку. Текст в текстовом поле не изменится.
69. Закройте приложение.

Поскольку работа с диалоговым окном, имеющим вкладки, мало чем отличается от работы с обычным диалоговым окном, а в каждую вкладку нужно поместить какой-нибудь элемент управления, то все вкладки сделаны одинаковыми.

В отличие от диалоговых окон Visual Studio (я намеренно не употребляю здесь термин "диалоговые окна Windows", поскольку приложения KDevelop могут работать и под управлением этой операционной системы), в среде KDevelop для работы с панелями диалоговых окон, имеющих вкладки, не используются специальные объекты классов, а все элементы управления принадлежат непосредственно диалоговому окну, что существенно затрудняет работу с диалоговыми окнами, имеющими несколько вкладок, каждая из которых содержит множество элементов управления. Однако это имеет и свои преимущества. Поскольку для реализации вкладок используется специальный элемент управления, то в диалоговое окно, помимо него и стандартных кнопок, могут быть добавлены другие элементы управления, что невозможно в приложениях Visual Studio.

Так как все элементы управления принадлежат непосредственно диалоговому окну, то и приемники посылаемых ими сигналов располагаются в объекте класса диалогового окна. Для простоты, сигналы всех текстовых полей обрабатываются одним и тем же приемником.

Приемник `TabDlgImpl::textChanged` вызывается при внесении пользователем изменений в текстовое поле, расположенное на любой из вкладок диалогового окна. В этом приемнике измененный пользователем текст сохраняется в закрытой переменной класса.

Приемник `TabDlgImpl::pageSelected` вызывается при переходе пользователя на другую вкладку диалогового окна. В качестве аргумента этому приемнику передается имя открываемой вкладки. Прежде всего, здесь, с использованием функции `QCheckBox::isChecked`, производится проверка состояния флажка **Synchronize?**. Если этот флажок установлен, значение аргумента приемника проверяется для определения того, в какое текстовое поле следует вывести сохраненную в объекте класса строку. Для вывода текста в текстовое поле используется функция `QLineEdit::setText`.

Изменения, внесенные в функцию `main` и в содержащий ее файл, аналогичны изменениям, внесенным в эту функцию в рассмотренном ранее диалоговом приложении, и направлены на то, чтобы сделать диалоговое окно главным окном приложения.

Создание мастера

Говоря о диалоговых окнах нельзя обойти вниманием их особую разновидность, называемую *мастером* (wizard), представляющую собой последовательность диалоговых окон, переход между которыми осуществляется с помощью кнопок **Next** (Далее) и **Back** (Назад). Этот тип диалогового окна хорошо знаком пользователям, работающим в графических оболочках, таких как Windows и KDE.

Процесс создания мастера во многом аналогичен процессу создания обычного диалогового окна и диалогового окна с вкладками, поэтому в рассмотренном в этом разделе приложении мы сосредоточимся только на тех аспектах, которые их различают. Текст приложения мастера можно найти на прилагаемом к книге CD.

Чтобы самостоятельно создать это приложение:

1. По методике, описанной в начале этой главы для диалогового приложения, создайте приложение **Wizard**.
2. Выберите команду меню **File | New**, нажмите комбинацию клавиш <Ctrl>+<N> или кнопку **New** в панели инструментов. Появится диалоговое окно **New File**.
3. В окне списка раскрытой вкладки **General** выделите строку **Qt Designer File (*.ui)**, в текстовое поле **Filename** введите имя файла `wizarddlg` (расширение ui будет добавлено к нему автоматически) и нажмите кнопку **OK**. Появится диалоговое окно **Load decision**.
4. Нажмите кнопку **No**. Откроется окно приложения Qt Designer by Trolltech.
5. В окне списка открывшегося при вызове приложения диалогового окна **New File** выделите значок **Wizard** и нажмите кнопку **OK**. В приложении

появится пустая заготовка окна мастера, а в панели **Property Editor/Signal Handlers** появится список свойств созданного окна.

6. В панели **Property Editor/Signal Handlers** выделите строку **name** и введите в связанное с ней текстовое поле имя диалогового окна **WizardDlg**.
7. В той же панели выделите строку **caption** и введите в связанное с ней текстовое поле заголовок мастера **Qt Wizard**.
8. Там же выделите строку **pageTitle** и введите в нее заголовок вкладки **Page 1**.
9. Выделите строку **pageName** и введите в нее имя вкладки **firstPage**.
10. Выберите команду меню **Tools | Containers | ButtonGroup** или нажмите кнопку **Button Group** в панели инструментов **Containers** и создайте групповую рамку кнопок в центре заготовки панели мастера.
11. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя групповой рамки на **nextGroup**.
12. В той же панели выделите строку **text** и введите в связанное с ней текстовое поле строку **Next**.
13. Выберите команду меню **Tools | Buttons | RadioButton** или нажмите кнопку **Radio Button** в панели инструментов **Buttons** и создайте переключатель в групповой рамке кнопок.
14. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя переключателя на **secondButton**.
15. В той же панели выделите строку **text** и введите в связанное с ней текстовое поле строку **Page 2**.
16. Там же выделите строку **checked** и выделите в связанном с ней раскрывающемся списке строку **True**.
17. Повторите пункты 13—15, поместив под только что введенный переключатель новый переключатель с именем **thirdButton** и заголовком **Page 3**.
18. Выделите групповую рамку кнопок и выберите команду меню **Layout | Lay Out Vertically**, нажмите комбинацию клавиш <Ctrl>+<L> или кнопку **Lay Out Vertically** в панели инструментов **Layout**. Переключатели в групповой рамке будут выровнены по вертикали.
19. Поместите горизонтальные разделители слева и справа от групповой рамки.
20. Поместите вертикальные разделители выше и ниже групповой рамки.
21. Щелкните левой кнопкой мыши на свободном пространстве панели и выберите команду меню **Layout | Lay Out in a Grid**, нажмите комбинацию клавиш <Ctrl>+<G> или кнопку **Lay Out in a Grid** в панели инст-

рументов **Layout**. Групповая рамка кнопок будет помещена в центр панели.

Совет

Если результат не достигнут, выберите команду меню **Layout | Break Layout**, нажмите комбинацию клавиш <Ctrl>+<B> или кнопку **Break Layout** в панели инструментов для отмены автоматического размещения, перетащите групповую рамку кнопок на пересечение разделителей и повторите пункт 21.

22. Выберите команду меню **Layout | Adjust Size**, нажмите комбинацию клавиш <Ctrl>+<J> или кнопку **Adjust Size** в панели инструментов **Layout**. Размер панели будет приведен в соответствие с размерами групповой рамки. В результате произведенных действий заготовка панели мастера примет вид, изображенный на рис. 2.27.

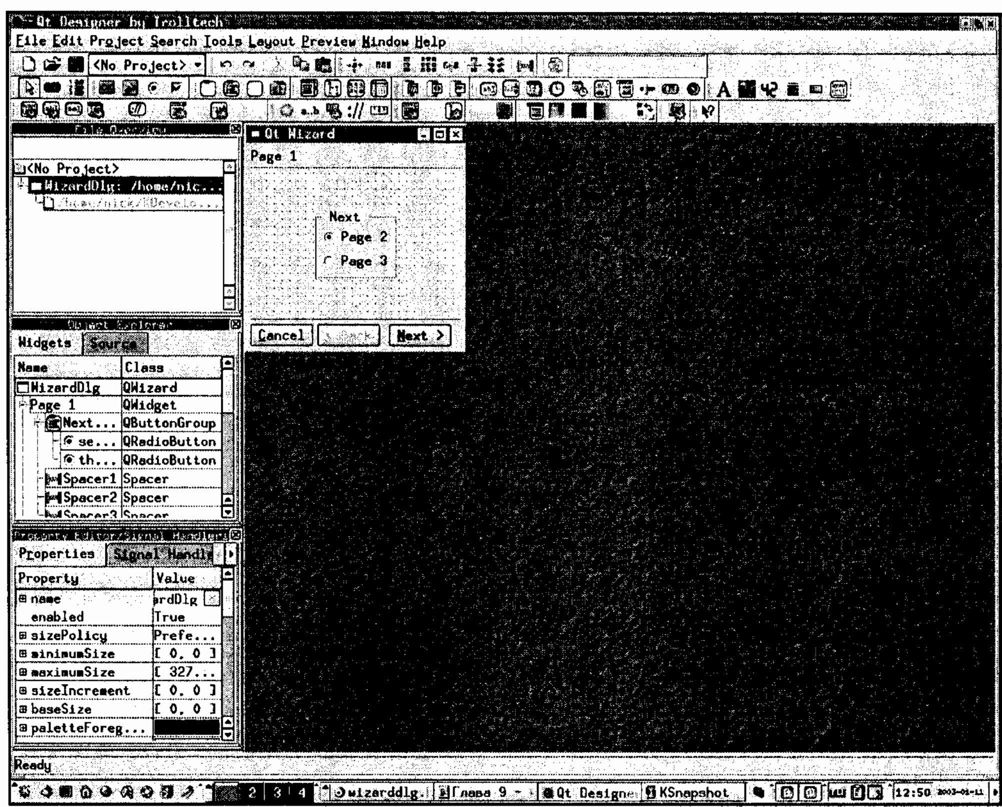


Рис. 2.27. Заготовка панели мастера

23. В панели **Object Explorer** щелкните правой кнопкой мыши по каталогу **WizardDlg** и выберите в появившемся контекстном меню команду **Add**

Page. В каталоге появится новый подкаталог с заголовком **Page**, а в окне редактирования ресурсов появится заготовка новой панели.

24. В панели **Property Editor/Signal Handlers** выделите строку **pageTitle** и введите в нее заголовок вкладки **Page 2**.
25. В той же панели выделите строку **pageName** и введите в нее имя вкладки **secondPage**.
26. Выберите команду меню **Tools | Buttons | CheckBox** или нажмите кнопку **Check Box** в панели инструментов **Buttons** и создайте флажок в центре заготовки панели мастера.
27. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя флажка на **secondExitCheck**.
28. В той же панели выделите строку **text** и введите в связанное с ней текстовое поле строку **Exit?** (Завершить работу?).
29. Поместите горизонтальные разделители слева и справа от флажка.
30. Поместите вертикальные разделители выше и ниже флажка.
31. Щелкните левой кнопкой мыши на свободном пространстве панели и выберите команду меню **Layout | Lay Out in a Grid**, нажмите комбинацию клавиш <Ctrl>+<G> или кнопку **Lay Out in a Grid** в панели инструментов **Layout**. Флажок будет помещен в центр панели.
32. Повторите пункты 23—31, создав новую панель с заголовком **Page 3** и именем **thirdPage**, поместив в нее флажок с именем **thirdExitCheck**.
33. В заготовке мастера дважды нажмите кнопку **Back**. Появится первая панель мастера.
34. Выберите команду меню **Tools | Connect Signal/Slots**, нажмите клавишу <F3> или кнопку **Connect Signal/Slots** в панели инструментов **Tools**.
35. Поместите указатель мыши на групповую рамку кнопок, нажмите левую кнопку мыши и, не отпуская ее, переместите указатель мыши на свободное место в заготовке диалогового окна. Появится диалоговое окно **Edit Connections**.
36. Нажмите кнопку **Edit Slots**. Появится диалоговое окно **Edit Slots**.
37. Нажмите кнопку **New Slot**, введите в текстовое поле **Slot** сигнатуру нового приемника `nextPage(int)` и нажмите кнопку **OK**. Диалоговое окно **Edit Slots** закроется, а в окне **Slots** диалогового окна **Edit Connections** появится сигнатура нового разъема.
38. В окне списка **Signals** выделите сигнал `clicked(int)`, а в окне списка **Slots** — приемник `nextPage(int)`. В окне списка **Connections** появится информация о новом соединении.
39. Нажмите кнопку **OK**. Диалоговое окно **Edit Connections** закроется.

40. В заготовке мастера нажмите кнопку **Next**. Раскроется заготовка второй панели мастера.
41. Повторите пункты 34—39 для флажка второй панели, сопоставив его сигналу `toggled(bool)` приемник `exitCheck2(bool)`.
42. Повторите пункты 40 и 41 для флажка третьей панели, сопоставив его сигналу `toggled(bool)` приемник `exitCheck3(bool)`.
43. Сохраните созданную заготовку мастера в файле описания пользовательского интерфейса приложения и закройте приложение Qt Designer by Trolltech.
44. В окне иерархических списков среды разработки KDevelop раскройте вкладку **Classes**, щелкните правой кнопкой мыши на каталоге **Classes** и выберите в появившемся контекстном меню команду **New class**. Появится диалоговое окно **Class Generator**.
45. В текстовое поле **Classname** введите имя класса `WizardDlgImpl`, в текстовое поле **Baseclass** — имя базового класса `WizardDlg`, в текстовое поле **Documentation** — комментарий `Implementation of wizard class` (Реализация класса мастера), в группе **Additional Options** установите флажок **generate a QWidget-Childclass** и нажмите кнопку **OK**. В проект будут добавлены файлы заголовка и реализации, содержащие заготовку нового класса.
46. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `WizardDlgImpl` и выберите в появившемся контекстном меню команду **Add slot**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Slots**.
47. В текстовое поле **Declaration** введите сигнатуру перегружаемого приемника `nextPage(int)`, установите флажок **Virtual** в группе **Modifiers**, в текстовое поле **Documentation** введите комментарий `Sets up the next page` (Выбирает следующую панель) и нажмите кнопку **Apply**. В класс будет добавлен новый приемник.
48. Повторите пункты 46 и 47 для добавления в класс `WizardDlgImpl` нового приемника, имеющего сигнатуру `exitCheck2(bool)`, снабдив его комментарием `Enables exit from the page` (Разрешает завершение работы на данной панели).
49. Повторите пункты 46 и 47 для добавления в класс `WizardDlgImpl` нового приемника, имеющего сигнатуру `exitCheck3(bool)`, снабдив его комментарием `Enables exit from the page`.
50. Повторите пункты 46 и 47 для добавления в класс `WizardDlgImpl` нового приемника, имеющего сигнатуру `back()`, снабдив его комментарием `Handles the Back button press` (Обрабатывает нажатие кнопки Назад).

51. Повторите пункты 46 и 47 для добавления в класс `WizardDlgImpl` нового приемника, имеющего сигнатуру `next()`, снабдив его комментарием `Handles the Next button press` (Обрабатывает нажатие кнопки Далее).

52. В открывшемся после добавления приемников окне редактирования файла wizarddlgimpl.cpp измените реализацию класса `WizardDlgImpl` в соответствии с текстом листинга 2.6.

Листинг 2.6. Реализация класса `WizardDlgImpl`

```
WizardDlgImpl::WizardDlgImpl( QWidget* parent, const char* name,
bool modal, WFlags fl)
  : WizardDlg( parent, name, modal, fl)
{
 Next  = 0;
}

WizardDlgImpl::~WizardDlgImpl()
{
}

/** Выбирает следующую панель */
void WizardDlgImpl::nextPage(int page)
{
 Next = page;
}

/** Разрешает завершение работы на данной панели */
void WizardDlgImpl::exitCheck2(bool b)
{
 setFinishEnabled( SecondPage, b);
}

/** Разрешает завершение работы на данной панели */
void WizardDlgImpl::exitCheck3(bool b)
{
 setFinishEnabled( ThirdPage, b);
}

/** Обрабатывает нажатие кнопки Назад */
void WizardDlgImpl::back()
{
 if( currentPage() == ThirdPage)
  showPage( FirstPage);
```

```
  else
  {
   WizardDlg::back();
   setNextEnabled( SecondPage, true);
  }

}
/** Обрабатывает нажатие кнопки Далее */
void WizardDlgImpl::next()
{
  if( currentPage() == FirstPage)
  {
   switch( Next)
   {
    case 0: WizardDlg::next();
            setNextEnabled( SecondPage, false);
            break;
    case 1: showPage( SecondPage);
             WizardDlg::next();
             break;
   }
  }
  else
    WizardDlg::next();
}
```

53. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `WizardDlgImpl` и выберите в появившемся контекстном меню команду **Add member variable**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes**.
54. В текстовое поле **Type** введите тип переменной `int`, в текстовое поле **Name** — идентификатор переменной `Next`, установите переключатель **Private** в группе **Access**, в текстовое поле **Documentation** введите комментарий `Index of next page` (Индекс следующей вкладки) и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.
55. В открывшемся после добавления переменной окне редактирования файла wizarddlgimpl.h замените строку `WizardDlgImpl(QWidget* parent = 0, const char* name = 0);` строкой

```
WizardDlgImpl( QWidget* parent = 0, const char* name = 0,
bool modal = false, WFlags fl = 0);
```

56. Откройте окно редактирования файла main.cpp и после строки `#include "wizard.h"` поместите строку

```
#include "wizarddlgimpl.h"
```

57. Измените главную функцию приложения в соответствии с текстом листинга 2.7.

Листинг 2.7. Функция `main`

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

 KAboutData aboutData( "wizard", I18N_NOOP("Wizard"),
  VERSION, description, KAboutData::License_GPL,
  "(c) 2003, ", 0, 0, "");
 aboutData.addAuthor("",0, "");
 KCmdLineArgs::init( argc, argv, &aboutData);
 KCmdLineArgs::addCmdLineOptions( options); // Добавьте собственные
                                            // опции

 KApplication a;

 WizardDlgImpl dlg;
 a.setMainWidget(&dlg);

 return dlg.exec();
}
```

58. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Появится окно мастера, раскрытое на своей первой панели, как это показано на рис. 2.28.

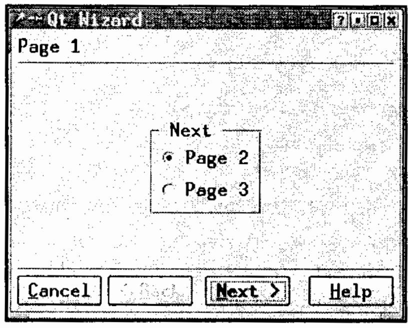


Рис. 2.28. Первая панель мастера

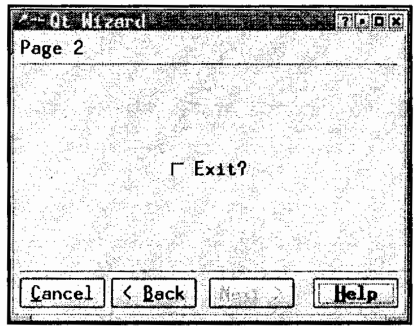


Рис. 2.29. Вторая панель мастера

59. Нажмите кнопку **Next**. Откроется вторая панель мастера, в которой будет недоступна кнопка **Next**, как это показано на рис. 2.29.

60. Установите флажок **Exit?**. В панели мастера появится кнопка **Finish**, как это показано на рис. 2.30.

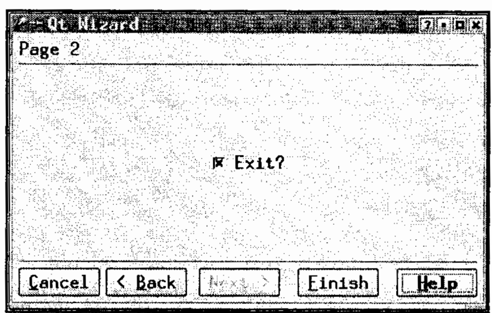


Рис. 2.30. Вторая панель мастера с кнопкой **Finish**

61. Сбросьте флажок **Exit?**. Кнопка **Finish** станет недоступной, но не исчезнет.
62. Нажмите кнопку **Back**. Откроется первая панель мастера, но он сохранит свои увеличенные после добавления в него кнопки **Finish** размеры.
63. Установите переключатель **Page 3** в группе **Next** и нажмите кнопку **Next**. Откроется третья панель мастера, изображенная на рис. 2.31. Эта панель уже имеет кнопку **Finish**, но она пока недоступна.

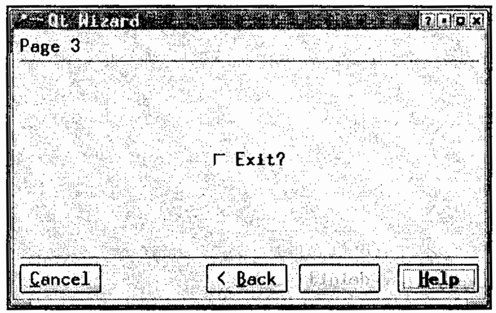


Рис. 2.31. Третья панель мастера

64. Нажмите кнопку **Back**. Откроется первая панель мастера.
65. Нажмите кнопку **Next**. Откроется третья панель мастера.
66. Установите в ней флажок **Exit?**. Кнопка **Finish** станет доступной.
67. Нажмите кнопку **Finish**. Приложение закроется.

Изначально предполагалось, что мастера будут использоваться только для вывода линейной последовательности диалоговых окон, когда для каждого окна, кроме первого и последнего, существует единственное предшествующее окно и единственное последующее окно. Однако во многих случаях возникает необходимость нарушения этой наперед заданной последовательности окон в зависимости от выбора пользователя. В этом случае для одно-

го из окон последовательности может быть определено сразу несколько окон, в которые может быть осуществлен переход из него. Именно этот случай и был реализован в рассматриваемом приложении.

Из первой страницы приложения в зависимости от выбора пользователя может быть осуществлен переход ко второй или к третьей странице. При нажатии кнопки **Back** на каждой из этих двух страниц переход будет осуществлен к первой странице, поскольку именно с нее пользователь получил к ним доступ. На каждой из этих страниц установлен флажок, позволяющий получить доступ к кнопке **Finish**, как правило, используемой для успешного завершения работы мастера. Необходимость обусловить доступ к этой кнопке какими-то действиями пользователя связана с тем, что обычно мастера используются для получения от пользователя некоторой информации и ее последующей обработки. Поэтому работа мастера не может быть успешно завершена при отсутствии всей необходимой для этой обработки информации.

Как следует из текстов программ, созданных средой разработки KDevelop на основе файла описания пользовательского интерфейса, созданного приложением Qt Designer, в библиотеке Qt, в отличие от Windows, отсутствует специальный класс для работы с панелями мастера. Поэтому вся информация об элементах управления всех панелей мастера помещается в общий класс мастера, что нельзя признать хорошим решением. Во-первых, это не позволяет использовать одинаковые имена для схожих элементов управления различных панелей, во-вторых, затрудняет работу с мастерами, содержащими большое число панелей, и, в-третьих, приводит к необходимости помещения всех операций, связанных с переходом от одной панели к другой, всего в две функции (представьте себе, как будет выглядеть перегрузка приемника `next` в мастере, осуществляющем различные обработки сигнала, когда при переходе к панелям обработки нужно считать состояние элементов управления текущей панели, преобразовать их в параметры, передаваемые процессу обработки, и запустить этот процесс).

Работа с элементами управления мастера ничем принципиально не отличается от описанной в предыдущем разделе работы с элементами управления диалогового окна. Поэтому здесь мы не будем подробно останавливаться на этом вопросе, а сразу перейдем к реализации переходов между панелями и управлению доступностью кнопок мастера.

Переход к следующей панели мастера осуществляется нажатием на кнопку **Next**, посылающей сигнал приемнику `QWizard::next`. Для изменения установленного по умолчанию порядка перехода между панелями этот приемник следует перегрузить. Перегрузка этого приемника выполнена по полной форме, хотя в данном случае она является излишней, поскольку кнопка **Next** доступна в рассматриваемом приложении только на первой панели. Несмотря на это, в перегруженном приемнике производится проверка того,

что эта кнопка была нажата именно в первой панели, для чего указатель на объект класса текущей панели, возвращаемый функцией `QWizard::currentPage`, сравнивается со значением указателя на первую панель, и в противном случае вызывается приемник базового класса.

Если кнопка **Next** была нажата в первой панели, вызывается оператор-переключатель, аргументом которого является текущее положение переключателей группы **Next**. Если необходимо произвести переход ко второй панели мастера, то сначала вызывается приемник базового класса, а затем в этой панели вызовом функции `QWizard::setNextEnabled` делается недоступной кнопка **Next**. Если же нужно перейти к третьей панели, сначала вызовом функции `QWizard::showPage` выводится на экран вторая панель мастера, а затем из нее производится переход к третьей панели вызовом приемника базового класса.

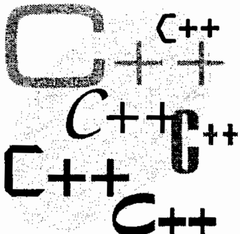
Переход к предыдущей панели мастера осуществляется нажатием на кнопку **Back**, посылающей сигнал приемнику `QWizard::back`. При перегрузке этой функции производится проверка того, что кнопка **Back** была нажата в третьей панели и, если это так, вызывается функция `showPage`, выводящая первую панель мастера. В противном случае вызывается приемник базового класса, и кнопка **Next** во второй панели делается доступной.

При сравнении приемников `next` и `back` в глаза сразу бросаются две особенности: переход из первой панели к третьей осуществляется в два этапа, а обратный переход происходит в один этап, и кнопка **Next** во второй панели делается доступной только на время ее отображения на экране. Это связано с тем, что разработчики где-то перемудрили с доступностью кнопки **Back**: при первом переходе из первой панели она становится доступной только после вызова приемника базового класса и в том случае, если в данную панель возможен переход из предыдущей панели последовательности (последнее правило действует не только при первом переходе).

Вывод на экран или обеспечение доступности или недоступности (если она уже выведена на экран) кнопки **Finish** осуществляется вызовом функции `QWizard::setFinishEnabled`, первый аргумент которой содержит указатель на объект панели, в которой следует сделать доступной или недоступной данную кнопку, а второй — определяет новое состояние кнопки. Эти функции вызываются одновременно в приемниках, связанных с сигналами, посылаемыми при переключении флажков.

Изменения, внесенные в главную функцию приложения, полностью аналогичны изменениям, внесенным в эту функцию рассмотренного ранее диалогового приложения.

ГЛАВА 3



Классы элементов управления

В *главе 2* было описано создание класса диалогового окна и процесс включения в него объектов классов простейших элементов управления. Рассмотренные в этой главе элементы управления выполняют какую-либо одну элементарную операцию и работа с ними понятна на интуитивном уровне. Однако существуют элементы управления, реализация которых описывается достаточно сложными классами, каждый из которых нуждается в отдельном описании. Некоторые из этих классов будут рассмотрены в данной главе.

Класс списка

При работе с диалоговыми окнами пользователю часто приходится работать со списками объектов: выделять объекты из списка, добавлять их в список или удалять их из него. Для выполнения этих операций в библиотеку Qt включен специальный класс, обеспечивающий работу связанного с ним элемента управления. Для демонстрации возможностей элемента управления `ListBox` нами создано приложение **ListBox**, текст которого можно найти на прилагаемом к книге CD.

Чтобы самостоятельно создать это приложение:

1. Выберите команду меню **Project | New** (Проект | Новый). Появится диалоговое окно **ApplicationWizard** (Мастер создания приложений).
2. В окне иерархического списка диалогового окна в папке **KDE** выделите строку **KDE Mini** (Приложение KDE с минимальными возможностями) и нажмите кнопку **Next** (Далее). Появится второе окно мастера **ApplicationWizard**.
3. В текстовое поле **Project name** (Имя проекта) введите имя проекта **ListBox**, заполните остальные текстовые поля корректной информацией (или оставьте их без изменений, поскольку этот проект не будет распро-

страняться) и нажмите кнопку **Create** (Создать). Появится последнее окно мастера **ApplicationWizard** и начнется процесс создания заготовки приложения.

4. Как только станет доступной кнопка **Exit** (Выход), нажмите ее и выйдите из мастера создания приложений.
5. Выберите команду меню **File | New** (Файл | Создать). Появится диалоговое окно **New File** (Создание файла).
6. В окне списка раскрытой вкладки **General** (Общие типы) выделите строку **Qt Designer File (*.ui)** (Файл Qt Designer), в текстовое поле **Filename** (Имя файла) введите имя файла `listboxdlg` (расширение ui будет добавлено к нему автоматически) и нажмите кнопку **OK**. Появится диалоговое окно **Load decision**, предлагающее открыть файл в текстовом виде.
7. Нажмите кнопку **No** (Нет). Откроется окно приложения Qt Designer by Trolltech.
8. В окне списка появившегося при старте приложения диалогового окна **New File** (Новый файл) выделите значок **Dialog with Buttons (Right)** (Диалоговое окно с кнопками, расположенными справа) и нажмите кнопку **OK**. В приложении появится пустая заготовка диалогового окна, а в панели **Property Editor/Signal Handlers** (Редактор свойств/Обработчики сигналов) появится список свойств созданного окна.
9. В панели **Property Editor/Signal Handlers** выделите строку **name** (имя) и введите в связанное с ней текстовое поле имя диалогового окна **ListBoxDlg**.
10. В той же панели выделите строку **caption** (заголовок) и введите в связанное с ней текстовое поле заголовок диалогового окна **List Box Dialog**.
11. Выберите команду меню **Tools | Display | TextLabel** (Сервис | Изображения | Статический текст) или нажмите кнопку **Text Label** в панели инструментов **Display** (при этом кнопка должна "утопиться") и щелкните левой кнопкой мыши в левом верхнем углу заготовки диалогового окна. На месте щелчка появится рамка статического текста.
12. В панели **Property Editor/Signal Handlers** выделите строку **text** (текст) и введите в связанное с ней текстовое поле строку **List Box**.
13. Выберите команду меню **Tools | Views | ListBox** (Сервис | Представления | Список) или нажмите кнопку **List Box** в панели инструментов **Views** и щелкните левой кнопкой мыши под только что введенным статическим текстом. На месте щелчка появится окно списка, как это показано на рис. 3.1.
14. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя окна списка на **ListBox**.

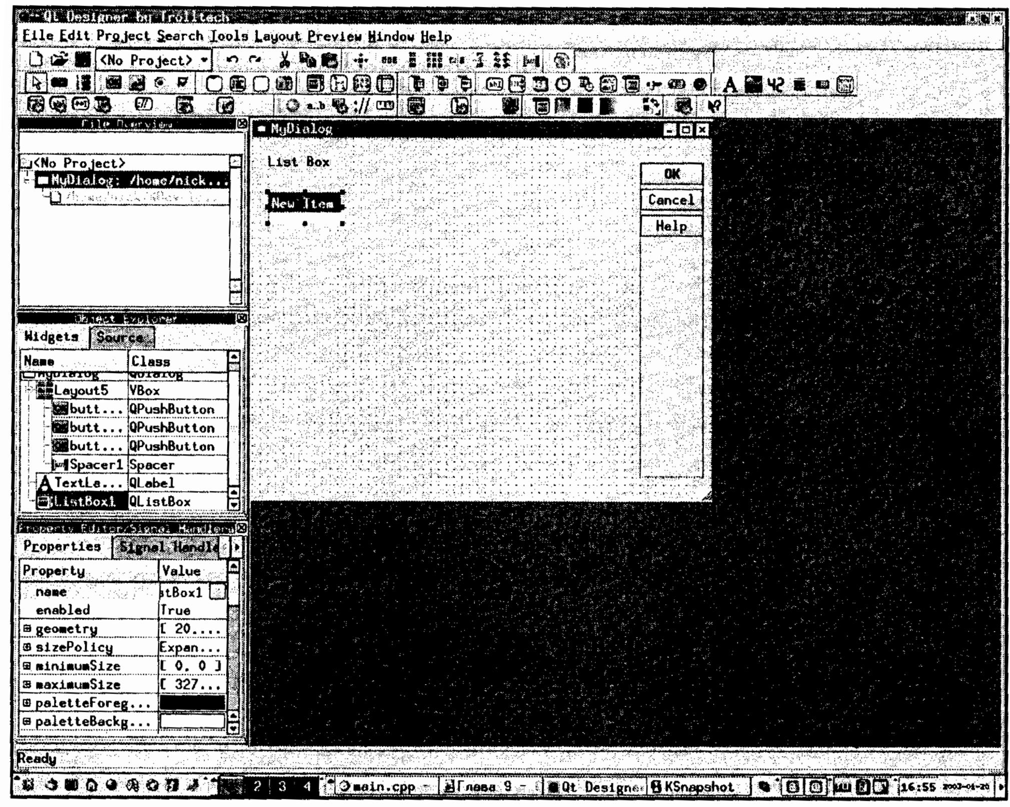


Рис. 3.1. Заготовка диалогового окна с окном списка

15. Повторите пункты 10—12 для создания справа от заголовка окна списка нового заголовка **Item**.

16. Выберите команду меню **Tools | Input | LineEdit** (Сервис | Ввод | Текстовое поле) или нажмите кнопку **Line Edit** в панели инструментов **Input** и щелкните левой кнопкой мыши под только что введенным статическим текстом. На месте щелчка появится текстовое поле.

17. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя текстового поля на **LineEdit**.

18. Выберите команду меню **Tools | Buttons | PushButton** (Сервис | Кнопки | Кнопка) или нажмите кнопку **Push Button** в панели инструментов **Buttons** и щелкните левой кнопкой мыши под только что введенным текстовым полем. На месте щелчка появится кнопка.

19. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя кнопки на **AddButton**.

20. В той же панели выделите строку **text** и введите в связанное с ней текстовое поле текст **Add** (Добавить).

21. Повторите пункты 18—20 для создания справа от только что созданной кнопки новой кнопки с именем **DeleteButton** и текстом **Delete**.

22. Выберите команду меню **Layout | Add Spacer** (Расположение | Добавить разделитель) или нажмите кнопку **Spacer** в панели инструментов **Layout** и щелкните левой кнопкой мыши слева от статического текста **List Box**.

23. В появившемся контекстном меню выберите команду **Horizontal** (Горизонтальный). На месте щелчка появится горизонтальный разделитель.

24. Повторите пункты 22 и 23 для установки горизонтального разделителя справа от статического текста **List Box**.

25. Выделите статический текст **List Box** и горизонтальные разделители и выберите команду меню **Layout | Lay Out Horizontally** (Расположить | Расположить по горизонтали), нажмите комбинацию клавиш <Ctrl>+<H> или кнопку **Lay Out Horizontally** в панели инструментов **Layout**. Статический текст будет выровнен по горизонтали.

26. Выделите статический текст **List Box** с разделителями и окно списка и выберите команду меню **Layout | Lay Out Vertically** (Расположить | Расположить по вертикали), нажмите комбинацию клавиш <Ctrl>+<L> или кнопку **Lay Out Vertically** в панели инструментов **Layout**. Окно списка со своим заголовком будет выровнено по вертикали.

27. Повторите пункты 22—25 для статического текста **Item**.

28. Поместите горизонтальный разделитель между кнопками **Add** и **Delete** и выровняйте их по горизонтали.

29. Повторите пункт 22, щелкнув левой кнопкой мыши между группой кнопок **Add** и **Delete** и текстовым полем и выбрав в появившемся контекстном меню команду **Vertical** (Вертикальный). На месте щелчка появится вертикальный разделитель.

30. Выделите статический текст **Item** с разделителями, текстовое поле, вертикальный разделитель, кнопки **Add** и **Delete** и выровняйте их по вертикали.

31. Поместите вертикальные разделители над каждой из трех кнопок и под кнопкой **OK**.

32. Поместите горизонтальные разделители слева и справа от каждой из кнопок.

33. Выделите кнопки и окружающие их разделители и выберите команду меню **Layout | Layout in a Grid** (Расположение | Расположить в сетке), нажмите комбинацию клавиш <Ctrl>+<G> или кнопку **Layout in a Grid** в панели инструментов **Layout**. Кнопки и разделители будут выровнены в сетке.

34. Выделите текстовое поле с его заголовком и кнопки и выровняйте их по вертикали.

35. Выделите всю заготовку диалогового окна и выровняйте ее по горизонтали.

36. Выберите команду меню **Layout | Adjust Size** (Расположение | Настроить размер), нажмите комбинацию клавиш <Ctrl>+<J> или кнопку **Adjust Size** в панели инструментов **Layout**. Размер заготовки диалогового окна будет приведен в соответствие с его содержимым. В результате описанных выше действий заготовка диалогового окна примет вид, изображенный на рис. 3.2.

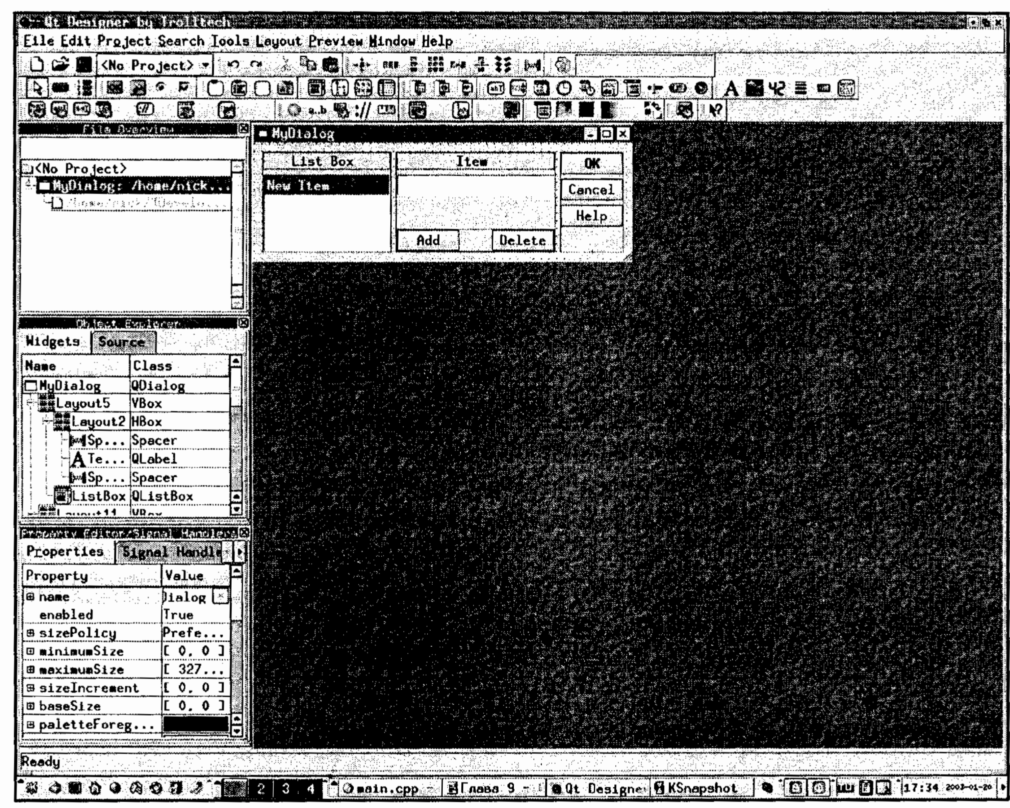


Рис. 3.2. Окончательный вид заготовки диалогового окна

37. Выберите команду меню **Tools | Connect Signal/Slots** (Сервис | Связать сигналы с приемниками), нажмите клавишу <F3> или кнопку **Connect Signal/Slots** в панели инструментов **Tools**. Кнопка **Connect Signal/Slots** в панели инструментов "утопится".

38. Поместите указатель мыши в окно списка, нажмите левую кнопку мыши и, не отпуская ее, переместите указатель мыши в свободную область заготовки диалогового окна. При отпускании кнопки мыши появится диалоговое окно **Edit Connections**.
39. Нажмите кнопку **Edit Slots**. Появится диалоговое окно **Edit Slots**.
40. Нажмите кнопку **New Slot**, введите в ставшее после этого доступным текстовое поле **Slot** сигнатуру приемника `onLbnDblclkList(QListBoxItem*)` и нажмите кнопку **OK**. В классе диалогового окна появится новый приемник.
41. В окне списка **Signals** диалогового окна **Edit Connections** выделите сигнатуру сигнала `doubleClicked(QListBoxItem*)`, а в окне списка **Slots** — сигнатуру приемника `onLbnDblclkList(QListBoxItem*)`. Информация о новом соединении появится в окне списка **Connections**.
42. Нажмите кнопку **OK**. Информация о новом соединении будет добавлена в заготовку диалогового окна.
43. Повторите пункты 37—42 для сопоставления сигналу `clicked()` кнопки **Add** приемника `onAdd()` класса диалогового окна.
44. Повторите пункты 37—42 для сопоставления сигналу `clicked()` кнопки **Delete** приемника `onDelete()` класса диалогового окна.
45. Сохраните файл описания ресурсов и выйдите из приложения Qt Designer.
46. В приложении KDevelop раскройте вкладку **Classes** (Классы) в окне иерархических списков, щелкните правой кнопкой мыши по каталогу **Classes** и выберите в появившемся контекстном меню команду **New class** (Новый класс). Появится диалоговое окно **Class Generator**.
47. В текстовое поле **Classname** (Имя класса) введите имя класса `ListBoxDlgImpl`, в текстовое поле **Baseclass** (Имя базового класса) введите имя базового класса `ListBoxDlg`, в группе **Additional Options** (Дополнительные параметры) установите флажок **generate a QWidget-Childclass**, в текстовое поле **Documentation** (Документация) введите комментарий `Class for dialog implementation` (Класс реализации диалогового окна) и нажмите кнопку **OK**. В приложение будет добавлен новый класс и будут открыты окна редактирования его файлов заголовка и реализации.
48. В открывшемся после добавления класса окне редактирования файла listboxdlgimpl.h измените заголовок класса `ListBoxDlgImpl` в соответствии с текстом листинга 3.1.

Листинг 3.1. Заголовок класса `ListBoxDlgImpl`

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

```
class ListBoxDlgImpl : public ListBoxDlg
{
  Q_OBJECT

public:
 ListBoxDlgImpl(QWidget *parent=0, const char *name=0, bool modal = true);
 ~ListBoxDlgImpl();

public slots:
  /* Обрабатывает нажатие кнопки Delete */
  virtual void onDelete();
  /* Обрабатывает нажатие кнопки Add */
  virtual void onAdd();
  /* Обрабатывает двойной щелчок мыши в окне списка */
  virtual void onLbnDblclkList(QListBoxItem*);
};
```

49. Щелкните правой кнопкой мыши в окне редактирования файла listboxdlgimpl.h и выберите в появившемся контекстном меню команду **Switch Header/Source** (Переключение файлов заголовка и реализации).
50. В открывшемся окне редактирования файла listboxdlgimpl.cpp измените реализацию класса `ListBoxDlgImpl` в соответствии с текстом листинга 3.2.

Листинг 3.2. Реализация класса `ListBoxDlgImpl`

```
ListBoxDlgImpl::ListBoxDlgImpl(QWidget *parent, const char *name, bool
modal)
  : ListBoxDlg(parent,name,modal)
{
 ListBox->clear();
}

ListBoxDlgImpl::~ListBoxDlgImpl()
{
}

/* Обрабатывает нажатие кнопки Delete */
void ListBoxDlgImpl::onDelete()
{
 ListBox->removeItem( ListBox->currentItem());
}

/* Обрабатывает нажатие кнопки Add */
void ListBoxDlgImpl::onAdd()
```

```
{
 ListBox->insertItem( LineEdit->text());
 LineEdit->clear();
}

/* Обрабатывает двойной щелчок мыши в окне списка */
void ListBoxDlgImpl::onLbnDblclkList(QListBoxItem* item)
{
 LineEdit->setText( item->text());
}
```

51. Откройте окно редактирования файла main.cpp и после строки `#include "listbox.h"` вставьте строку

```
#include "listboxdlgimpl.h"
```

52. Измените главную функцию приложения в соответствии с текстом листинга 3.3.

Листинг 3.3. Функция `main`

```
int main(int argc, char *argv[])
{
 KAboutData aboutData( "listbox", I18N_NOOP("ListBox"),
  VERSION, description, KAboutData::License_GPL,
  "(c) 2003, ", 0, 0, "");
 aboutData.addAuthor("",0, "");
 KCmdLineArgs::init( argc, argv, &aboutData);
 KCmdLineArgs::addCmdLineOptions( options); // Добавьте свои опции.

 KApplication a;
 ListBoxDlgImpl dlg;
 a.setMainWidget( &dlg);

 return dlg.exec();
}
```

53. Выберите команду меню **Debug | Start** (Отладка | Пуск) или нажмите кнопку **Debug** в панели инструментов. Приложение будет запущено на исполнение.
54. В текстовое поле **Item** введите текст **First** и нажмите кнопку **Add**. Текст появится в окне списка и исчезнет из текстового поля.
55. Повторите пункт 54 для текста **Second**.
56. Повторите пункт 54 для текста **Third**. Окно приложения примет вид, изображенный на рис. 3.3.

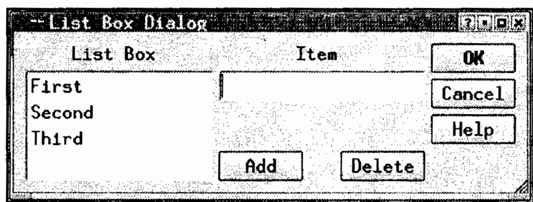


Рис. 3.3. Окно приложения **ListBox**

57. Дважды щелкните левой кнопкой мыши по строке **Second** в окне списка. Текст появится в текстовом поле **Item**.
58. Нажмите кнопку **Delete**. Выделенный текст исчезнет из окна списка, но сохраниться в текстовом поле, как это показано на рис. 3.4.
59. Нажмите кнопку **OK** и закройте приложение.

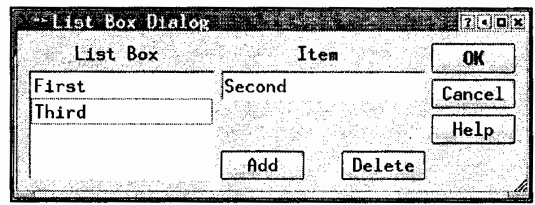


Рис. 3.4. Удаление элемента из списка

Работа с приложением Qt Designer и процесс создания класса реализации диалогового окна уже были нами подробно рассмотрены в *главе 2*, поэтому здесь мы сразу перейдем к внесенным в этот класс изменениям.

Поскольку от нас требуется, чтобы главное окно диалогового приложения могло быть сделано модальным, но мы не вносим никакие изменения в установленный по умолчанию набор флагов окна, то в конструктор класса реализации диалогового окна добавлен только третий аргумент. Значение четвертого аргумента подставляется конструктором базового класса по умолчанию. Кроме того, чтобы иметь возможность запускать конструктор создаваемого класса без аргументов, добавляемому в него третьему аргументу по умолчанию присваивается значение `true`, указывающее на то, что будет создаваться модальное диалоговое окно.

В класс реализации диалогового окна не добавляется никаких новых переменных и функций, а только производится перегрузка методов базового класса.

Создаваемый по умолчанию объект класса `QListBox` имеет одну неприятную особенность: в нем по умолчанию сразу же выводится строка **New Item**, что в большинстве случаев является излишним. Чтобы устранить эту особен-

ность, в конструкторе диалогового окна для этого объекта сразу же вызывается функция `QListBox::clear`, очищающая его содержимое.

Добавление строки в окно списка производится в функции `ListBoxDlgImpl::onAdd`, обрабатывающей нажатие кнопки **Add**. В ней с помощью функции `QLineEdit::text` считывается содержимое текстового поля **Item**, которое тут же, вызовом функции `QListBox::insertItem`, добавляется в конец списка. Для удобства работы пользователя текстовое поле после этого очищается вызовом функции `QLineEdit::clear`, позволяя вводить в него следующую строку.

Удаление выделенной строки производится в функции `ListBoxDlgImpl::onDelete`, обрабатывающей нажатие на кнопку **Delete**. Эта операция производится вызовом функции `QListBox::removeItem`, которой в качестве аргумента следует передать индекс удаляемой строки. Значение этого индекса для выделенной строки возвращается функцией `QListBox::currentItem`.

Функция `ListBoxDlgImpl::onLbnDblclkList` обрабатывает двойной щелчок левой кнопкой мыши по элементу списка. Если двойной щелчок производится просто в окне списка, данная функция не вызывается. Поэтому в ней нет необходимости в проверке наличия в списке выделенного элемента, что существенно упрощает структуру этой функции. В качестве аргумента данной функции передается указатель на объект класса `QListBoxItem`, содержащий описание выделенного элемента списка. Для получения содержащегося в нем текста используется функция `QListBoxItem::text`, возвращаемое значение которой передается в качестве аргумента функции `QLineEdit::setText`, выводящей этот текст в текстовое поле.

Классы линейного регулятора и линейного индикатора

При создании диалоговых окон часто возникает необходимость задания некоторой величины, изменяющейся в определенных пределах. Часто не столь важна точность задания этой величины, как удобство ее изменения. Это относится, например, к регуляторам уровня громкости воспроизводимого сигнала. При работе с аналоговой аудиоаппаратурой пользователи привыкли использовать с этой целью переменные резисторы ползункового типа. Для решения подобных задач в графических интерфейсах используются линейные регуляторы, позволяющие преобразовывать перемещения своего бегунка в числовые значения.

Во многих случаях пользователю необходимо представить информацию о степени завершенности некоторого процесса, например процесса записи файла на диск. Для этого чаще всего используются линейные индикаторы, в которых степень завершенности процесса индицируется заполнением поля элемента управления каким-либо цветом или орнаментом.

4 Зак. 113

Для демонстрации возможностей линейного регулятора и линейного индикатора было создано демонстрационное приложение **Progress**, текст которого можно найти на прилагаемом к книге CD-диске.

Чтобы самостоятельно создать это приложение:

1. По описанной в предыдущем разделе методике создайте заготовку диалогового приложения **Progress** и добавьте в него файл описания ресурсов progressdlg.ui, открыв тем самым приложение Qt Designer.
2. В окне списка появившегося при старте приложения диалогового окна **New File** выделите значок **Dialog** (Диалоговое окно) и нажмите кнопку **OK**. В приложении появится пустая заготовка диалогового окна, а в панели **Property Editor/Signal Handlers** — список свойств созданного окна.
3. В панели **Property Editor/Signal Handlers** выделите строку **name** и введите в связанное с ней текстовое поле имя диалогового окна **ProgressDlg**.
4. В той же панели выделите строку **caption** и введите в связанное с ней текстовое поле заголовок диалогового окна **Progress Dialog**.
5. Выберите команду меню **Tools | Display | ProgressBar** (Сервис | Изображения | Линейный индикатор) или нажмите кнопку **Progress Bar** в панели инструментов **Display** (при этом кнопка должна "утопиться") и щелкните левой кнопкой мыши в верхней части заготовки диалогового окна. На месте щелчка появится линейный индикатор.
6. В панели **Property Editor/Signal Handlers** выделите строку **name** и введите в связанное с ней текстовое поле строку **ProgressBar**.
7. В той же панели выделите строку **frameShape** (форма рамки) и выделите в связанном с ней раскрывающемся списке строку **Panel** (Панель).
8. Там же выделите строку **frameShadow** (обрамление) и выделите в связанном с ней раскрывающемся списке строку **Sunken** (Углубленное).
9. Выберите команду меню **Tools | Input | Slider** (Сервис | Ввод | Линейный регулятор) или нажмите кнопку **Slider** в панели инструментов **Input** (при этом кнопка должна "утопиться") и щелкните левой кнопкой мыши под линейным индикатором.
10. В появившемся контекстном меню выберите команду **Horizontal**. На месте щелчка появится горизонтальный линейный регулятор.
11. В панели **Property Editor/Signal Handlers** выделите строку **name** и введите в связанное с ней текстовое поле строку **Slider**.
12. В той же панели выделите строку **tickmarks** (метки на шкале регулятора) и выделите в связанном с ней раскрывающемся списке строку **Both** (с обеих сторон).
13. Там же выделите строку **tickInterval** (интервал меток) и установите в тестовом поле связанного с ней линейного регулятора значение 10.

14. В той же панели выделите строку **maxValue** (максимальное значение) и введите в связанное с ней текстовое поле значение `100`.
15. Выберите команду меню **Tools | Containers | Frame** (Сервис | Контейнеры | Рамка) или нажмите кнопку **Frame** в панели инструментов **Containers**. Кнопка **Frame** должна "утопиться".
16. Поместите указатель мыши над левым верхним углом линейного регулятора, нажмите левую кнопку мыши и, не отпуская ее, переместите указатель мыши под правый нижний угол линейного регулятора. Линейный регулятор будет охвачен рамкой.
17. В панели **Property Editor/Signal Handlers** выделите строку **frameShadow** и выделите в связанном с ней раскрывающемся списке строку **Sunken** (Углубленное).
18. Выберите команду меню **Tools | Buttons | PushButton** (Сервис | Кнопки | Кнопка) или нажмите кнопку **Push Button** в панели инструментов **Buttons** (при этом кнопка должна "утопиться") и щелкните левой кнопкой мыши в нижней части заготовки диалогового окна. На месте щелчка появится кнопка.
19. В панели **Property Editor/Signal Handlers** выделите строку **name** и введите в связанное с ней текстовое поле строку **ResetButton**.
20. В той же панели выделите строку **text** и введите в связанное с ней текстовое поле строку **Reset**.
21. Повторите пункты 18—20, поместив справа от только что введенной кнопки новую кнопку с именем **OKButton** и текстом **OK**.
22. Поместите горизонтальные разделители слева и справа от кнопки **Reset** и справа от кнопки **OK**.
23. Выделите разделители и кнопки и выберите команду меню **Layout | Lay Out Horizontally**, нажмите комбинацию клавиш <Ctrl>+<H> или кнопку **Lay Out Horizontally** в панели инструментов **Layout**. Кнопки будут выровнены по горизонтали.
24. Выделите рамку и выберите команду меню **Layout | Lay Out Horizontally**, нажмите комбинацию клавиш <Ctrl>+<H> или кнопку **Lay Out Horizontally** в панели инструментов **Layout**. Размеры линейного регулятора будут связаны с размерами окружающей его рамки.
25. Поместите вертикальные разделители выше и ниже линейного индикатора и выше и ниже кнопок.
26. Щелкните левой кнопкой мыши в свободной области заготовки диалогового окна и выберите команду меню **Layout | Lay Out Vertically**, нажмите комбинацию клавиш <Ctrl>+<L> или кнопку **Lay Out Vertically** в панели инструментов. Элементы управления диалогового окна будут выровнены по вертикали.

27. Выберите команду меню **Layout | Adjust Size**, нажмите комбинацию клавиш <Ctrl>+<J> или кнопку **Adjust Size** в панели инструментов. Размер заготовки диалогового окна будет приведен в соответствие с его содержимым, как это показано на рис. 3.5.

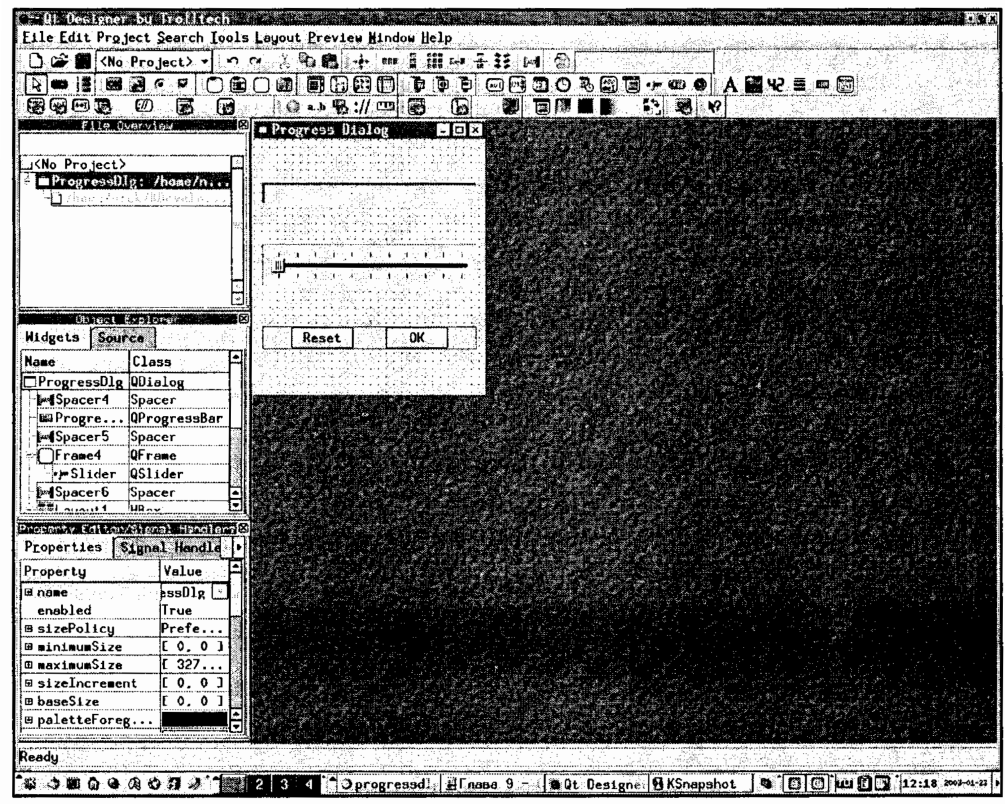


Рис. 3.5. Окончательный вид заготовки диалогового окна

28. Выберите команду меню **Tools | Connect Signal/Slots**, нажмите клавишу <F3> или кнопку **Connect Signal/Slots** в панели инструментов. Кнопка **Connect Signal/Slots** в панели инструментов "утопится".

29. Поместите указатель мыши на линейный регулятор, нажмите левую кнопку мыши и, не отпуская ее, переместите указатель мыши на свободное пространство заготовки диалогового окна. Отпустите левую кнопку мыши. Появится диалоговое окно **Edit Connections**.

30. Нажмите кнопку **Edit Slots**. Появится диалоговое окно **Edit Slots**.

31. Нажмите кнопку **New Slot** и введите в текстовое поле **Slot**, расположенное в группе **Slot Properties**, сигнатуру приемника `valueChanged(int)`.

32. Нажмите кнопку **OK**. В списке приемников класса диалогового окна появится новый приемник.
33. В окне списка **Signals** выделите сигнатуру сигнала `valueChanged(int)`, а в окне списка **Slots** — сигнатуру приемника `valueChanged(int)` и нажмите кнопку **OK**. В класс диалогового окна будет добавлена информация о новом соединении.
34. Повторите пункт 28, поместите указатель мыши на кнопку **Reset**, нажмите левую кнопку мыши и, не отпуская ее, переместите указатель мыши на линейный индикатор. Отпустите левую кнопку мыши. Появится диалоговое окно **Edit Connections**.
35. В окне списка **Signals** выделите сигнатуру сигнала `pressed()`, а в окне списка **Slots** — сигнатуру приемника `reset()` и нажмите кнопку **OK**. В класс диалогового окна будет добавлена информация о новом соединении.
36. Повторите пункты 28, 29 и 33 для связывания сигнала `pressed()` кнопки **OKButton** со стандартным приемником `accept()` диалогового окна.
37. Сохраните информацию о заготовке диалогового окна в файле описания ресурсов.
38. Закройте приложение Qt Designer.
39. В приложении KDevelop раскройте вкладку **Classes** в окне иерархических списков, щелкните правой кнопкой мыши по каталогу **Classes** и выберите в появившемся контекстном меню команду **New class**. Появится диалоговое окно **Class Generator**.
40. В текстовое поле **Classname** введите имя класса `ProgressDlgImpl`, в текстовое поле **Baseclass** — имя базового класса `ProgressDlg`, в группе **Additional Options** установите флажок **generate a QWidget-Childclass**, в текстовое поле **Documentation** введите комментарий `Class of dialog implementation` и нажмите кнопку **OK**. В приложение будет добавлен новый класс и будут открыты окна редактирования его файлов заголовка и реализации.
41. В открывшемся после добавления класса файле progressdlgimpl.h измените заголовок класса `ProgressDlgImpl` в соответствии с текстом листинга 3.4.

Листинг 3.4. Заголовок класса `ProgressDlgImpl`

```
#include <qprogressbar.h>
class ProgressDlgImpl : public ProgressDlg
{
  Q_OBJECT
```

```
public:
 ProgressDlgImpl(QWidget *parent=0, const char *name=0,
 bool modal = true);
 ~ProgressDlgImpl();

public slots:
  /** Изменилось положение бегунка линейного регулятора */
  virtual void valueChanged(int);
};
```

42. В файле progressdlgimpl.cpp измените реализацию класса `ProgressDlgImpl` в соответствии с текстом листинга 3.5.

Листинг 3.5. Реализация класса `ProgressDlgImpl`

```
ProgressDlgImpl::ProgressDlgImpl(QWidget *parent, const char *name,
bool modal)
  : ProgressDlg(parent,name,modal)
{
}

ProgressDlgImpl::~ProgressDlgImpl()
{
}

/** Изменилось положение бегунка линейного регулятора */
void ProgressDlgImpl::valueChanged(int pos)
{
 if( ProgressBar->progress() < pos)
  ProgressBar->setProgress(pos);
}
```

43. Откройте окно редактирования файла main.cpp и после строки `#include "progress.h"` вставьте строку

```
#include "progressdlgimpl.h"
```

44. Измените главную функцию приложения в соответствии с текстом листинга 3.6.

Листинг 3.6. Функция `main`

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

 KAboutData aboutData( "progress", I18N_NOOP("Progress"),
  VERSION, description, KAboutData::License_GPL,
  "(c) 2003, ", 0, 0, "");
```

```
    aboutData.addAuthor("",0, "");
    KCmdLineArgs::init( argc, argv, &aboutData);
    KCmdLineArgs::addCmdLineOptions( options); // Добавьте свои опции

    KApplication a;
    ProgressDlgImpl dlg;
    a.setMainWidget( &dlg);

    return dlg.exec();
}
```

45. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Приложение будет запущено на исполнение.
46. Переместите линейный регулятор вправо. Его перемещение будет отражено в линейном индикаторе, как это показано на рис. 3.6.

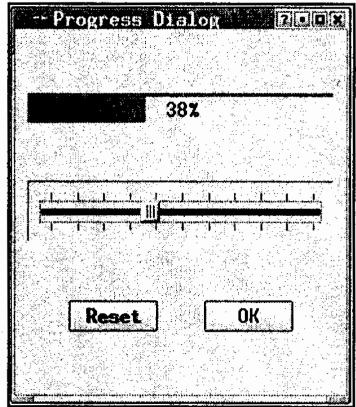


Рис. 3.6. Окно приложения **Progress**

47. Переместите линейный регулятор влево. Линейный индикатор останется без изменений.
48. Переместите линейный регулятор вправо за его предыдущую крайнюю позицию. Его перемещение будет отражено в линейном индикаторе.
49. Нажмите кнопку **Reset**. Линейный индикатор очистится.
50. Нажмите кнопку **OK** и закройте приложение.

Программирование работы с линейным регулятором и с линейным индикатором в библиотеке Qt намного проще, чем в Windows, поскольку в ней допускается настройка основных параметров этих элементов управления непосредственно при их включении в заготовку диалогового окна. Некоторые из этих параметров были нами изменены при создании рассматриваемого диалогового приложения. Например, диапазон отображения линейного ин-

дикатора составлял 100 единиц, а диапазон регулирования линейного регулятора — 99 единиц. С целью согласования этих диапазонов для линейного регулятора нами был установлен диапазон в 100 единиц.

Кроме того, для линейного регулятора нами были установлены метки шкалы по обеим его сторонам и их шаг был выбран равным шагу страницы, т. е. шагу, на который перемещается бегунок линейного регулятора при щелчке мыши со стороны требуемого его перемещения. Другим важным параметром линейного регулятора является размер строки, определяющий минимальный шаг перемещения его бегунка.

Примечание

Все параметры линейного регулятора и линейного индикатора (включая их ориентацию в пространстве) могут изменяться в процессе выполнения приложения с использованием для этого соответствующих методов классов `QProgressBar` и `QSlider`.

Как следует из рассмотрения работы приложения, линейный индикатор хранит максимальное значение, выставленное линейным регулятором. Для этого в приемнике `ProgressDlgImpl::valueChanged`, осуществляющем связь линейного регулятора с линейным индикатором, сначала производится сравнение текущего положения линейного индикатора, возвращаемого функцией `QProgressBar::progress`, с текущим положением линейного регулятора, передаваемым приемнику в качестве аргумента. И только в том случае, если значение аргумента приемника превышает значение текущего положения линейного индикатора, аргумент приемника передается в качестве аргумента функции `QProgressBar::setProgress`, устанавливающей линейный индикатор в новое положение.

Линейный индикатор может перемещаться только вправо, он когда-нибудь полностью заполнится и перестанет функционировать. Для того чтобы пользователь имел возможность продолжить работу с диалоговым окном, в нем предусмотрена кнопка **Reset**, сбрасывающая линейный индикатор. Поскольку сброс линейного индикатора осуществляется его стандартным приемником, то у нас пока еще отсутствует необходимость вовлечения в процесс связи кнопки и линейного индикатора объекта диалогового окна.

Работа с датой и временем

Во многих приложениях от пользователя требуется ввести в диалоговое окно дату или время, причем сделать это надо так, чтобы введенные сведения сразу же были бы преобразованы во внутренний формат представления даты и времени. Поэтому для выполнения этой задачи в библиотеку Qt включены соответствующие классы, обеспечивающие работу связанных с ними элементов управления.

Для демонстрации принципов работы с датой и временем в диалоговых окнах нами было создано приложение **DateTime**, текст которого можно найти на прилагаемом к книге CD-диске.

Чтобы самостоятельно создать это приложение:

1. По описанной в предыдущем разделе методике создайте заготовку диалогового приложения **DateTime** и добавьте в него файл описания ресурсов datetimedlg.ui, открыв тем самым приложение Qt Designer.
2. В окне списка появившегося при старте приложения диалогового окна **New File** выделите значок **Dialog with Buttons (Bottom)** (Диалоговое окно с кнопками, расположенными снизу) и нажмите кнопку **OK**. В приложении появится пустая заготовка диалогового окна, а в панели **Property Editor/Signal Handlers** — список свойств созданного окна.
3. В панели **Property Editor/Signal Handlers** выделите строку **name** и введите в связанное с ней текстовое поле имя диалогового окна **DateTimeDlg**.
4. В той же панели выделите строку **caption** и введите в связанное с ней текстовое поле заголовок диалогового окна **Date and Time Dialog**.
5. Выберите команду меню **Tools | Display | TextLabel** или нажмите кнопку **Text Label** в панели инструментов **Display** и щелкните левой кнопкой мыши в левом верхнем углу заготовки диалогового окна. На месте щелчка появится рамка статического текста.
6. В панели **Property Editor/Signal Handlers** выделите строку **text** и введите в связанное с ней текстовое поле строку **Date Edit** (Редактирование даты).
7. Выберите команду меню **Tools | Input | DateEdit** (Сервис | Ввод | Редактирование даты) или нажмите кнопку **Date Edit** в панели инструментов **Input** (при этом кнопка должна "утопиться") и щелкните левой кнопкой мыши под статическим текстом. На месте щелчка появится редактор даты.
8. В панели **Property Editor/Signal Handlers** выделите строку **name** и введите в связанное с ней текстовое поле имя редактора даты **DateEdit**.
9. Выделите редактор даты и его заголовок и выровняйте их по вертикали.
10. Повторите пункты 5 и 6, создав под редактором даты статический текст **Time Edit** (Редактор времени).
11. Выберите команду меню **Tools | Input | TimeEdit** (Сервис | Ввод | Редактирование времени) или нажмите кнопку **Time Edit** в панели инструментов **Input** (при этом кнопка должна "утопиться") и щелкните левой кнопкой мыши под только что введенным статическим текстом. На месте щелчка появится редактор времени.
12. В панели **Property Editor/Signal Handlers** выделите строку **name** и введите в связанное с ней текстовое поле имя редактора времени **TimeEdit**.

13. Выделите редактор даты и его заголовок и выровняйте их по вертикали.
14. Повторите пункты 5 и 6, создав в правом верхнем углу заготовки диалогового окна статический текст **Date and Time Edit** (Редактор даты и времени).
15. Выберите команду меню **Tools | Input | DateTimeEdit** (Сервис | Ввод | Редактирование даты и времени) или нажмите кнопку **Date Time Edit** в панели инструментов **Input** (при этом кнопка должна "утопиться") и щелкните левой кнопкой мыши под только что введенным статическим текстом. На месте щелчка появится редактор даты и времени.
16. В панели **Property Editor/Signal Handlers** выделите строку **name** и введите в связанное с ней текстовое поле имя редактора времени **DateTimeEdit**.
17. Выделите редактор даты и его заголовок и выровняйте их по вертикали.
18. Повторите пункты 5 и 6, создав под редактором даты и времени статический текст **Date preview** (Предварительный просмотр даты).
19. Выберите команду меню **Tools | Input (KDE) | KDateWidget** (Сервис | Ввод (KDE) | KDateWidget) или нажмите кнопку **Date preview (KDE)** в панели инструментов **Input (KDE)** (при этом кнопка должна "утопиться") и щелкните левой кнопкой мыши под только что введенным статическим текстом. На месте щелчка появится модифицированный редактор даты.
20. В панели **Property Editor/Signal Handlers** выделите строку **name** и введите в связанное с ней текстовое поле имя модифицированного редактора даты **ModDateWidget**.
21. Выделите модифицированный редактор даты и его заголовок и выровняйте их по вертикали.
22. Щелкните левой кнопкой мыши на свободной области заготовки диалогового окна и выберите команду меню **Layout | Layout in a Grid**, нажмите комбинацию клавиш <Ctrl>+<G> или кнопку **Layout in a Grid** в панели инструментов **Layout**. Элементы управления диалогового окна будут выровнены в сетке.
23. Выберите команду меню **Layout | Adjust Size**, нажмите комбинацию клавиш <Ctrl>+<J> или кнопку **Adjust Size** в панели инструментов **Layout**. Размер заготовки диалогового окна будет приведен в соответствие с его содержимым. В результате описанных выше действий заготовка диалогового окна примет вид, изображенный на рис. 3.7.
24. Выберите команду меню **Tools | Connect Signal/Slots**, нажмите клавишу <F3> или кнопку **Connect Signal/Slots** в панели инструментов. Кнопка **Connect Signal/Slots** в панели инструментов "утопится".

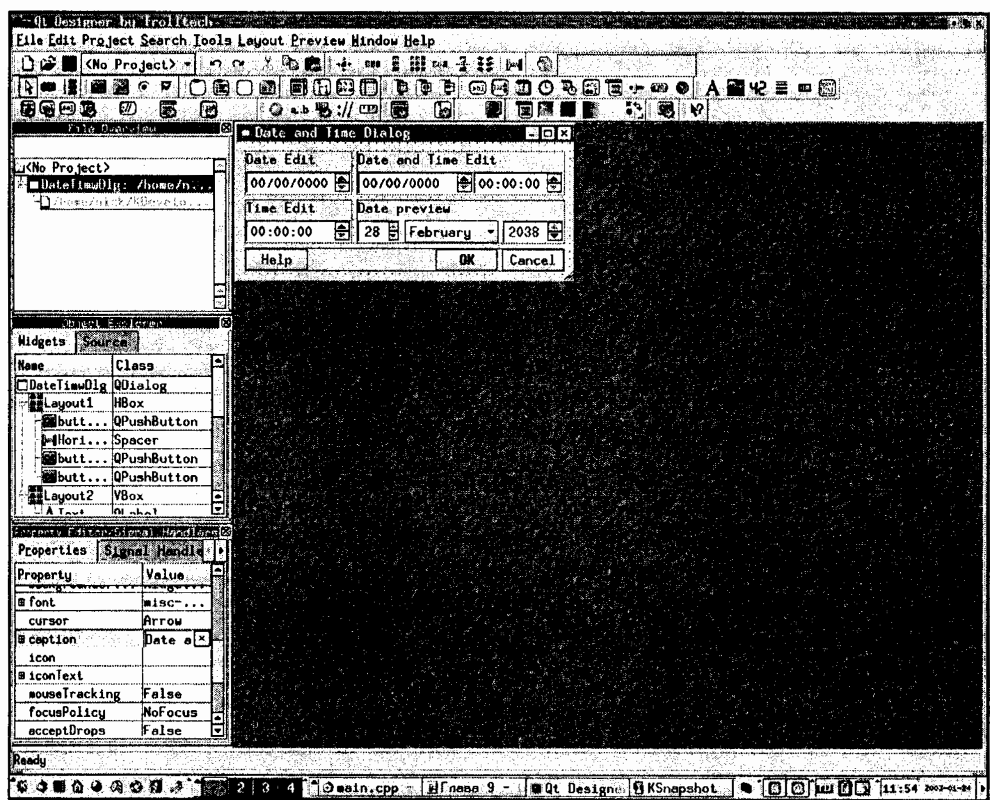


Рис. 3.7. Окончательный вид заготовки диалогового окна

25. Поместите указатель мыши в редактор даты, нажмите левую кнопку мыши и, не отпуская ее, переместите указатель мыши в свободную область заготовки диалогового окна. При отпускании кнопки мыши появится диалоговое окно **Edit Connections**.

26. Нажмите кнопку **Edit Slots**. Появится диалоговое окно **Edit Slots**.

27. Нажмите кнопку **New Slot**, введите в ставшее после этого доступным текстовое поле **Slot** сигнатуру приемника `slotDateEdit(const QDate&)` и нажмите кнопку **OK**. В классе диалогового окна появится новый приемник.

28. В окне списка **Signals** диалогового окна **Edit Connections** выделите сигнатуру сигнала `valueChanged(const QDate&)`, а в окне списка **Slots** — сигнатуру приемника `slotDateEdit(const QDate&)`. Информация о новом соединении появится в окне списка **Connections**.

29. Нажмите кнопку **OK**. Информация о новом соединении будет добавлена в заготовку диалогового окна.

30. Повторите пункты 24—29 для связывания сигнала `valueChanged(const QTime&)` редактора времени с приемником `slotTimeEdit(const QTime&)` диалогового окна.
31. Повторите пункты 24—29 для связывания сигнала `valueChanged(const QDateTime&)` редактора даты и времени с приемником `slotDateTimeEdit(const QDateTime&)` диалогового окна.
32. Сохраните изменения в файле описания ресурсов приложения.
33. Закройте приложение Qt Designer.
34. В приложении KDevelop раскройте вкладку **Classes** в окне иерархических списков, щелкните правой кнопкой мыши по каталогу **Classes** и выберите в появившемся контекстном меню команду **New class**. Появится диалоговое окно **Class Generator**.
35. В текстовое поле **Classname** введите имя класса `DateTimeDlgImpl`, в текстовое поле **Baseclass** — имя базового класса `DateTimeDlg`, в группе **Additional Options** установите флажок **generate a QWidget-Childclass**, в текстовое поле **Documentation** введите комментарий `Class of dialog implementation` и нажмите кнопку **OK**. В приложение будет добавлен новый класс и будут открыты окна редактирования его файлов заголовка и реализации.
36. В открывшемся после добавления класса файле datetimedlgimpl.h измените заголовок класса `DateTimeDlgImpl` в соответствии с текстом листинга 3.7.

Листинг 3.7. Заголовок класса `DateTimeDlgImpl`

```
class DateTimeDlgImpl : public DateTimeDlg
{
  Q_OBJECT
 public:
  DateTimeDlgImpl(QWidget *parent=0, const char *name=0,
  bool modal = true);
  ~DateTimeDlgImpl();

 public slots:
  /* Обрабатывает изменение даты */
  virtual void slotDateEdit(const QDate&);
  /* Обрабатывает изменение даты и времени */
  virtual void slotDateTimeEdit(const QDateTime&);
  /* Обрабатывает изменение времени */
  virtual void slotTimeEdit(const QTime&);
};
```

37. Щелкните правой кнопкой мыши в окне редактирования файла datetimedlgimpl.h и выберите в появившемся контекстном меню команду **Switch Header/Source**.

38. В открывшемся окне редактирования файла datetimedlgimpl.cpp измените реализацию класса `DateTimeDlgImpl` в соответствии с текстом листинга 3.8.

Листинг 3.8. Реализация класса `DateTimeDlgImpl`

```
#include <qdatetime.h>
#include <qdatetimeedit.h>
#include <kdatewidget.h>

#include "datetimedlgimpl.h"

DateTimeDlgImpl::DateTimeDlgImpl(QWidget *parent, const char *name,
bool modal)
  : DateTimeDlg(parent,name,modal)
{
}

DateTimeDlgImpl::~DateTimeDlgImpl()
{
}

/* Обрабатывает изменение даты */
void DateTimeDlgImpl::slotDateEdit(const QDate& newDate)
{
 if( newDate != DateTimeEdit-> dateEdit()-> date())
  DateTimeEdit-> dateEdit()-> setDate( newDate);

 if( newDate != ModDateWidget-> date())
  ModDateWidget-> setDate( newDate);
}

/* Обрабатывает изменение даты и времени */
void DateTimeDlgImpl::slotDateTimeEdit(const QDateTime& newDateTime)
{
 if( newDateTime.date() != DateEdit-> date())
  DateEdit-> setDate( newDateTime.date());

 if( newDateTime. date() != ModDateWidget-> date())
  ModDateWidget-> setDate( newDateTime. date());

 if( newDateTime.time() != TimeEdit-> time())
  TimeEdit-> setTime( newDateTime.time());
}
```

```
/* Обрабатывает изменение времени */
void DateTimeDlgImpl::slotTimeEdit(const QTime& newTime)
{
 if( newTime != DateTimeEdit-> timeEdit()-> time())
  DateTimeEdit-> timeEdit()-> setTime( newTime);
}
```

39. Откройте окно редактирования файла main.cpp и после строки `#include "datetime.h"` вставьте строку

```
#include "datetimedlgimpl.h"
```

40. Измените главную функцию приложения в соответствии с текстом листинга 3.9.

Листинг 3.9. Функция main

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

 KAboutData aboutData( "datetime", I18N_NOOP("DateTime"),
  VERSION, description, KAboutData::License_GPL,
  "(c) 2003, ", 0, 0, "");#include <qdatetime.h>
 aboutData.addAuthor("",0, "");
 KCmdLineArgs::init( argc, argv, &aboutData);
 KCmdLineArgs::addCmdLineOptions( options); // Добавьте свои опции

 KApplication a;
 DateTimeDlgImpl dlg;
 a.setMainWidget( &dlg);

 return dlg.exec();
}
```

41. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Приложение будет запущено на исполнение и появится его главное окно, изображенное на рис. 3.8.
42. Выделите год в редакторе даты. Главное окно приложения примет вид, изображенный на рис. 3.9.
43. Внесите изменения в различные поля приложения, за исключением модифицированного редактора даты. Все они будут отображаться в других окнах, как это показано на рис. 3.10.
44. Нажмите кнопку **OK**. Приложение закроется.

Поскольку в приемниках рассматриваемого приложения тесно переплетаются вызовы функций разных классов, обратимся сначала к самим классам, а потом перейдем к используемым в приложении функциям данных классов.

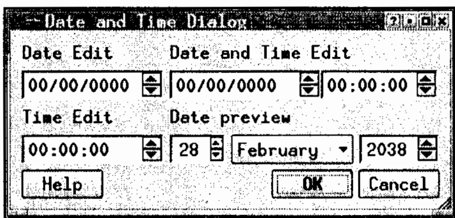


Рис. 3.8. Исходный вид главного окна приложения **DateTime**

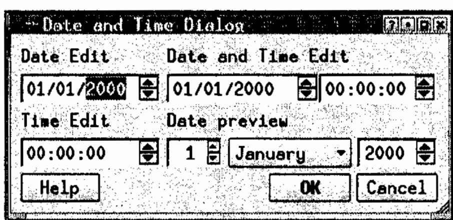


Рис. 3.9. Выделение года

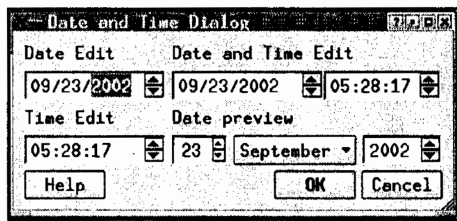


Рис. 3.10. Внесение изменений в поля приложения

Для работы с датой и временем в библиотеке Qt используются различные классы элементов управления, хранящие информацию в объектах специализированных классов. В отличие от библиотеки MFC, библиотека Qt не предлагает изысканного дизайна своих элементов управления. Да и выбор их очень ограничен.

Для редактирования даты используется объект класса `QDateEdit`, осуществляющие специальное форматирование инкрементного регулятора. Этот объект разбивает текстовое поле на три зоны, с каждой из которых инкрементный регулятор работает отдельно и устанавливает для нее свой диапазон изменения величин. Поскольку на моем компьютере установлен английский стандарт отображения даты, то он и используется при выводе информации в текстовое поле. Согласно этому формату сначала отображается месяц, затем — число, а потом — год. Поэтому значение в первом поле изменяется от 1 до 12, а во втором — в зависимости от числа дней в установленном месяце.

Внимание!

Как видно из рис. 3.8, редактор даты при запуске приложения по умолчанию инициализируется нулевыми значениями, которые, как показано на рис. 3.9, заменяются на 1 января 2000 года при первом же обращении к данному элементу управления. Такое поведение элементов управления не способствует укреплению доверия к использующему их приложению. Поэтому настоятельно рекомендуется инициализировать их вручную.

Для редактирования времени используется объект класса QTimeEdit, во многом аналогичный объекту класса QDateEdit, но оптимизированный для работы не с датой, а со временем. Поскольку дата и время понятия взаимосвязанные и пользователю часто приходится одновременно устанавливать и дату и время, то в библиотеку Qt включен класс QDateTimeEdit, позволяющий одновременно работать с объектами классов QDateEdit и QTimeEdit и по сути являющийся механическим объединением этих двух объектов.

В библиотеку KDE включены классы, позволяющие улучшить интерфейс приложений. Одним из таких классов является класс KDateWidget, позволяющий выводить дату в более красивом виде. К сожалению, этот класс предназначен только для вывода даты, а не для обеспечения возможности ее изменения пользователем, поскольку сигнал о внесении в него изменений посылается только при их программном внесении. При внесении изменений пользователем этот сигнал не посылается. Однако измененное значение может быть считано вызовом функции KDateWidget::date.

После обсуждения классов можно перейти к рассмотрению использующих их приемников приложения.

Приемник DateTimeDlgImpl::slotDateEdit вызывается при внесении пользователем изменений в редактор даты. В качестве аргумента этому приемнику передается ссылка на объект класса QDate, содержащий информацию о новой дате. Данная информация отображается в остальных элементах управления, выводящих эту информацию. Поскольку сигнал о внесении изменений в элемент управления посылается независимо от источника данного изменения, нам необходимо предотвратить бесконечные циклы, вызываемые сигналами, посылаемыми после программного изменения содержимого элемента управления. Для этого производится проверка необходимости внесения изменения, т. е. существования различия между информацией, уже хранящейся в элементе управления, и той информацией, которую в него хотят записать.

Для получения даты, содержащейся в редакторе даты и времени, сначала вызовом функции QDateTimeEdit::dateEdit получается указатель на внутренний объект класса QDateEdit, а для него уже вызывается функция QDateEdit::date. Если возвращенная дата не совпадает с датой, переданной в аргументе приемника slotDateEdit, новая дата копируется в объект QDateTimeEdit. Для этого снова получается указатель на внутренний объект класса QDateEdit и для него вызывается функция QDateEdit::setDate.

Хотя сигнал модифицированного редактора даты о внесении в него изменений не обрабатывается в данном приложении, что исключает возможность возникновения бесконечных циклов, его содержимое тоже проверяется для того, чтобы не менять его дважды. Содержимое этого элемента управления получается вызовом функции KDateWidget::date, а изменения в него вносятся вызовом функции KDateWidget::setDate.

При внесении изменений в редактор даты и времени один и тот же сигнал посылается независимо от того, что именно было изменено. Поэтому в приемнике данного сигнала помещаются как функции изменения даты, так и функции изменений времени.

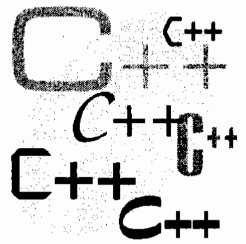
Для проверки необходимости внесения изменений в редактор даты производится проверка значения, возвращаемого функцией `QDateTime::date` объекта класса `QDateTime`, используемого редактором даты и времени для предоставления своей информации, и функцией `QDateEdit::date` объекта редактора даты. Если эти значения не совпадают, содержимое поля редактора даты изменяется вызовом функции `QDateEdit::setDate`.

Аналогичная операция производится и для объекта класса модифицированного редактора даты. При этом, как и в рассмотренном выше приемнике, содержимое этого элемента управления получается вызовом функции `KDateWidget::date`, а изменения в него вносятся вызовом функции `KDateWidget::setDate`.

Поскольку в редакторе даты и времени может быть изменена не только дата, но и время, в приемнике `slotDateTimeEdit` после внесения изменений в редакторы даты производится изменение содержимого редактора времени. При этом, прежде всего, проверяется необходимость внесения этих изменений. Для этого значение времени, возвращаемое функцией `QDateTime::time` объекта, переданного в аргументе этого приемника, сравнивается с возвращаемым значением функции `QTimeEdit::time` объекта редактора времени, и только в случае их несовпадения содержимое редактора времени изменяется вызовом функции `QTimeEdit::setTime`.

Приемник `DateTimeDlgImpl::slotTimeEdit` вызывается при внесении изменений в редактор времени. Его единственной задачей является отразить эти изменения в соответствующем поле редактора даты и времени. Прежде всего, в этом приемнике производится проверка необходимости внесения изменений. Для этого сначала с использованием функции `QDateTimeEdit::timeEdit` получается указатель на внутренний объект класса `QTimeEdit`, а затем для этого объекта вызывается функция `QTimeEdit::time` и возвращаемое ею значение сравнивается со значением аргумента приемника. Если эти значения не совпадают, в поле времени редактора даты и времени помещается новое значение. Для этого опять получается указатель на внутренний объект класса `QTimeEdit`, а затем для этого объекта вызывается функция `QTimeEdit::setTime`.

ГЛАВА 4



Классы приложений, документов и представлений

В процессе своего становления операционная система UNIX прошла несколько стадий развития пользовательского интерфейса. В начале это, конечно, были различные командные оболочки, использующие текстовый интерфейс командной строки. После этого стали появляться различные графические оболочки, существенно упрощающие работу пользователя с системой. Первое время это были оконные менеджеры, предоставляющие только средства управления элементами графического интерфейса и более или менее развитые средства запуска приложений и переключения между ними. Потом появились интегрированные среды, располагающие широким спектром написанных специально для них приложений и встроенными средствами настройки, исключающими редактирование файлов конфигурации.

В процессе развития операционных систем семейства UNIX особое внимание всегда обращалось на обеспечение надежности их функционирования, что не могло не отразиться на особенностях их пользовательского интерфейса. Долгое время основным типом графического приложения в этих операционных системах было однооконное приложение, в котором вся информация выводилась в главное окно приложения, что обеспечивало ему повышенную надежность. Однако большинство пользователей Windows привыкли работать с многооконными приложениями, предоставляющими им дополнительные возможности при работе с различными документами. Поскольку в настоящее время наблюдается определенная миграция пользователей, работавших до этого в Windwos, в среду Linux, в данной среде все более отчетливо проявляются признаки конвергенции ее пользовательского интерфейса с пользовательским интерфейсом Windows.

В основе работы с многооконными приложениями лежит концепция Документ/Представление, предполагающая, что в каждом приложении должны быть четко разделены функции манипулирования информацией и функции ее отображения. Эта же концепция может лежать и в основе однооконных

приложений, но в них она не является обязательной, поскольку для отображения одного документа в одно окно можно не разделять функции отображения и обработки информации. В случае же многооконного документа, когда для отображения содержимого одного документа может использоваться несколько объектов представления, использование данной концепции является практически безальтернативным вариантом.

Каждое приложение, построенное в соответствии с концепцией Документ/Представление, должно включать в себя описание класса, на который возлагаются задачи хранения обрабатываемой информации в оперативной памяти, предоставление этой информации по внешним запросам, запись и чтение этой информации с диска. Это же приложение должно включать в себя описание класса, производного от класса `QWidget`, на который возлагаются задачи отображения этой информации на экране. Помимо этих классов в приложение включается еще один класс, называемый классом приложения, ответственный за вывод главного окна приложения и реализацию интерфейса пользователя. При реализации данной концепции в Windows класс приложения реализует пользовательский интерфейс только в отсутствие в приложении открытых окон. При наличии окон в приложении Windows за работу с элементами пользовательского интерфейса отвечает класс представления активного окна.

Для того чтобы лучше разобраться с предлагаемой концепцией, разберем текст заготовки многооконного приложения, создаваемого мастером создания приложений среды разработки KDevelop.

Многооконное приложение Qt

Для начала разберем многооконное приложение, создаваемое средствами библиотеки Qt, поскольку оно использует классы более низкого уровня, что позволяет яснее просмотреть реализацию демонстрируемой концепции. Текст этого приложения можно найти на прилагаемом к книге CD-диске.

Чтобы создать заготовку многооконного приложения средствами Qt:

1. Выберите команду меню **Project | New** (Проект | Создать). Появится диалоговое окно **ApplicationWizard** (Мастер создания приложений), изображенное на рис. 4.1.
2. В окне иерархического списка диалогового окна раскройте папку **Qt**, выделите в ней строку **Qt MDI** (Многооконное приложение Qt) и нажмите кнопку **Next** (Далее). Появится второе окно мастера **ApplicationWizard**, изображенное на рис. 4.2.
3. В текстовое поле **Project name** (Имя проекта) введите имя проекта **QtMDI**, заполните остальные текстовые поля корректной информацией (или оставьте их без изменений, поскольку этот проект не будет

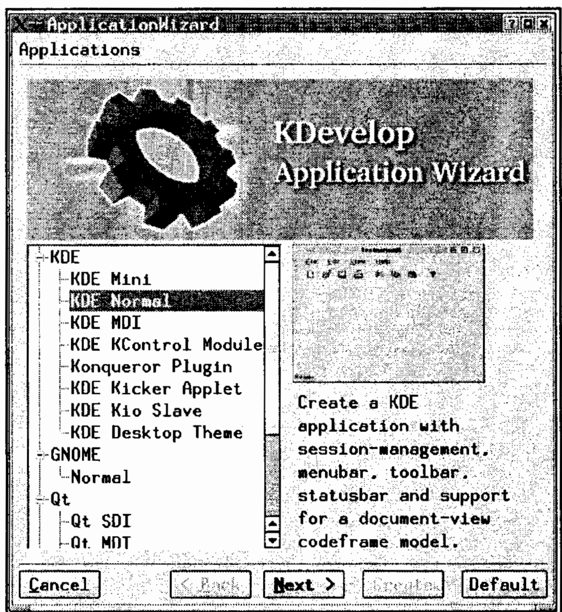


Рис. 4.1. Первое окно мастера **ApplicationWizard**

Рис. 4.2. Второе окно мастера **ApplicationWizard**

распространяться) и нажмите кнопку **Create** (Создать). Появится последнее окно мастера **ApplicationWizard**, в котором будет выводиться отчет о процессе создания приложения.

4. По завершении создания проекта, т. е. когда последнее окно мастера примет вид, изображенный на рис. 4.3, нажмите кнопку **Exit** (Выход). Окно мастера создания приложений закроется.

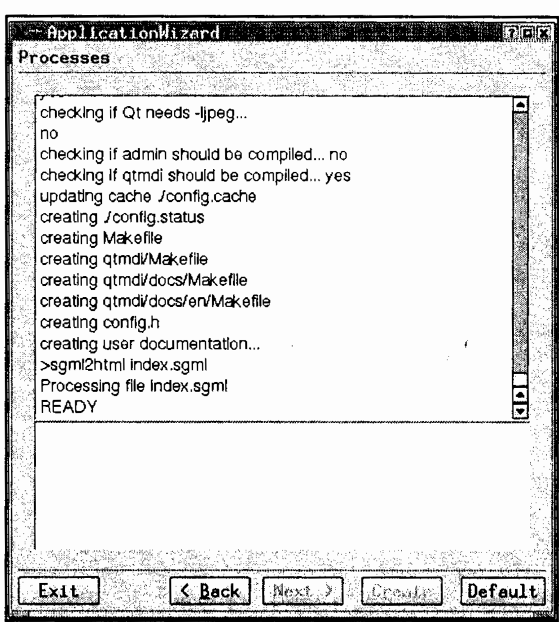


Рис. 4.3. Последнее окно мастера **ApplicationWizard**

Внимание!

Созданная заготовка многооконного приложения неработоспособна, поскольку в ней допущена ошибка: при объявлении дружественного класса или структуры компилятор требует в явной форме указать, что именно объявляется. Поскольку я не знаю, каким образом будет устранена эта ошибка, и поскольку это приложение не должно исполняться, я оставляю заготовку в ее неработоспособном виде.

Класс документа

Рассмотрим класс документа в созданном нами многооконном приложении. Для того чтобы получить доступ к файлу заголовка этого класса, в окне иерархических списков раскройте вкладку **Classes** (Классы) и щелкните левой кнопкой мыши по имени класса `QtMDIDoc`. Откроется окно редактирова-

ния файла qtmdidoc.h. Текст этого файла слишком велик, чтобы приводить его здесь полностью, поэтому в листинге 4.1 из него будут удалены шаблон и комментарий класса. Кроме того, все комментарии в нем, как и во всех последующих листингах, будут переведены на русский язык.

Листинг 4.1. Заголовок класса документа

```
#ifndef QTMDIDOC_H
#define QTMDIDOC_H

#ifdef HAVE_CONFIG_H
#include <config.h>
#endif

// включение файлов заголовков библиотеки QT
#include <qobject.h>
#include <qstring.h>
#include <qlist.h>

// предварительное объявление классов приложения QtMDI
class QtMDIView;

class QtMDIDoc : public QObject
{
 Q_OBJECT

 friend QtMDIView;

 public:
  /** Конструктор класса для работы с файлами приложения */
  QtMDIDoc();
  /** Деструктор класса для работы с файлами приложения */
  ~QtMDIDoc();

  /** добавляет в документ объект класса представления для его
    отображения. Как правило, это главный объект представления. */
  void addView(QtMDIView *view);
  /** удаляет объект класса представления из списка объектов
    представлений, с которыми связан данный документ */
  void removeView(QtMDIView *view);
  /** вызывается при удалении или добавлении объекта представления */
  void changedViewList();
  /** возвращает первый объект представления */
  QtMDIView* firstView(){ return pViewList->first(); };
```

```
 /** возвращает значение true, если запрашиваемый объект представления
   является единственным для данного документа */
 bool isLastView();
 /** Этот метод вызывается перед закрытием пользователем окна
   представления. При этом производится проверка существования других
   объектов представления, связанных с данным документом (тогда окно
   можно закрыть).
 * Если pFrame указывает на последний объект представления и
  в документ были внесены изменения,
 * пользователю посылается запрос на сохранение документа.
 */
 bool canCloseFrame(QtMDIView* pFrame);
 /** устанавливает флаг изменения после его модификации в одном
   из связанных с ним объектов представления.*/
 void setModified(bool _m=true){ modified=_m; };
 /** возвращает состояние флага изменения. Используется для определения
   необходимости сохранения документа при его закрытии.*/
 bool isModified(){ return modified; };
 /** уничтожает содержимое документа */
 void deleteContents();
 /** создает новый документ */
 bool newDocument();
 /** закрывает существующий документ */
 void closeDocument();
 /** загружает документ из файла в заданном формате и
   посылает сигнал updateViews() */
 bool openDocument(const QString &filename, const char *format=0);
 /** сохраняет документ в файле в заданном формате.*/
 bool saveDocument(const QString &filename, const char *format=0);
 /** устанавливает путь к файлу, содержащему документ */
 void setPathName(const QString &name);
 /** возвращает путь к текущему файлу документа */
 const QString& pathName() const;

 /** устанавливает имя файла документа */
 void setTitle(const QString &title);
 /** возвращает заголовок документа */
 const QString& title() const;

public slots:
 /** вызывает функции repaint() для всех представлений, связанных
   с данным объектом документа, и вызывается объектом представления,
   в котором документ был изменен.
  * Поскольку этот объект представления, как правило, перерисовывает
   себя сам, сообщение paintEvent не посылается.
  */
```

```
  void updateAllViews(QtMDIView *sender);

 private:
  /** флаг внесения изменений в текущий документ */
  bool modified;
  QString m_title;
  QString m_filename;
  /** список объектов классов представлений,
    с которыми связан данный документ */
  QList<QtMDIView> *pViewList;

};

#endif // QTMDIDOC_H
```

Данный класс является непосредственным потомком класса `QObject`, являющегося корневым классом иерархии объектов библиотеки Qt. Основное назначение этого класса — обеспечение связи объектов посредством *сигналов* (signal) и *приемников* (slot). В отличие от класса документа Windows, имеющего единственный защищенный конструктор, поскольку его объекты автоматически создаются приложением, класс документа в среде KDevelop имеет открытый конструктор, и его объекты создаются разработчиком.

Созданный нами класс имеет большое число различных методов и переменных, используемых для его работы. Однако мы рассмотрим реализацию только тех методов, которые решают те же задачи, что и методы, включаемые мастером среды разработки Microsoft Visual Studio в заготовку пользовательского класса документа — создание документа, сохранение его содержимого в файле и чтение его из файла. Кроме того, будут рассмотрены реализации конструктора и деструктора класса.

Чтобы получить доступ к файлу реализации класса документа, щелкните правой кнопкой мыши в окне редактирования файла qtmdidoc.h и выберите в появившемся контекстном меню команду **Switch Header/Source** (Переключение файлов заголовка и реализации). Откроется окно редактирования файла, фрагмент которого приведен в листинге 4.2.

Листинг 4.2. Фрагмент реализации класса документа

```
QtMDIDoc::QtMDIDoc()
{
 pViewList = new QList<QtMDIView>;
 pViewList->setAutoDelete(false);
}
```

```
QtMDIDoc::~QtMDIDoc()
{
 delete pViewList;
}

 ...

bool QtMDIDoc::newDocument()
{
 /////////////////////////////////////////////////
 // ЧТО ДЕЛАТЬ: добавить сюда инициализацию документа
 /////////////////////////////////////////////////
 modified=false;
 return true;
}

bool QtMDIDoc::openDocument(const QString &filename, const char *format
/*=0*/)
{

 QFile f( filename);
 if ( !f.open( IO_ReadOnly))
  return false;
 /////////////////////////////////////////////////
 // ЧТО ДЕЛАТЬ: добавить сюда чтение содержимого документа
 /////////////////////////////////////////////////
 f.close();

 modified=false;
 m_filename=filename;
 m_title=QFileInfo(f).fileName();
 return true;
}

bool QtMDIDoc::saveDocument(const QString &filename, const char *format
/*=0*/)
{
 QFile f( filename);
 if ( !f.open( IO_WriteOnly))
  return false;

 /////////////////////////////////////////////////
 // ЧТО ДЕЛАТЬ: добавить запись содержимого документа в файл
 /////////////////////////////////////////////////

 f.close();

 modified=false;
 m_filename=filename;
```

```
 m_title=QFileInfo(f).fileName();
 return true;
}
```

Как видно из листинга 4.2, мастер создания приложений делает предельно простые заготовки для методов класса документа. В конструкторе данного класса создается динамический объект списка объектов классов представления, связанных с данным объектом класса документа, и с помощью функции `QCollection::setAutoDelete` данному списку запрещается уничтожать удаляемые из него объекты. Созданный в конструкторе список объектов представления уничтожается в деструкторе этого класса.

Заготовка функции `newDocument`, создающей новый документ в данном объекте класса, только присваивает значение `false` переменной `modified`, указывая на то, что этот документ не содержит изменений, и возвращает значение `true`, свидетельствующее об успешном завершении операции.

Заготовка функции `openDocument`, вызываемой для чтения содержимого документа из файла, имеет более сложную структуру. В ней создается объект класса `QFile`, конструктору которого в качестве аргумента передается имя файла документа, и этот файл открывается только для чтения функцией `QFile::open`. Если в процессе открытия файла произошла ошибка, функция `openDocument` возвращает значение `false`.

Поскольку структура документа заранее неизвестна, в заготовке функции указано место, где должны производиться операции по чтению содержимого документа из файла. В отличие от Windows в данном случае информация считывается не из архива, предполагающего только потоковое чтение, а непосредственно из файла, что позволяет использовать любые методы доступа к нему. После чтения всей информации из файла он закрывается вызовом функции `QFile::close`.

После завершения операции чтения документа из файла в функции `openDocument` в переменную `modified` записывается значение `false`. В переменную `m_filename` помещается имя файла, переданное данной функции в качестве аргумента, а в переменную `m_title` записывается короткое имя файла. Для получения короткого имени файла динамически создается объект класса `QFileInfo`, его конструктору передается ссылка на объект класса `QFile`, по которому следует получить информацию, и вызывается функция `fileName` данного объекта.

Поскольку все операции по чтению содержимого документа из файла закончились успешно, в завершение своей работы функция `openDocument` возвращает значение `true`.

Текст заготовки функции `saveDocument`, сохраняющей содержимое документа в файле, в основном аналогичен тексту функции `openDocument`. Основным отличием между ними является то, что файл в данном случае открывается

только для записи, и, естественно, разработчик должен поместить в нее функции не для чтения, а для записи содержимого документа.

Рассмотренный нами класс документа является только заготовкой. Для того чтобы он превратился в полноценный класс документа, в него нужно внести следующие изменения:

- добавить в этот класс переменные, необходимые для хранения информации, содержащейся в документе;
- включить инициализацию этих переменных в функцию `newDocument`;
- организовать чтение этих переменных в функции `openDocument` и их сохранение в функции `saveDocument`.

Класс представления

Класс представления, в отличие от класса документа, является неотъемлемым атрибутом любого приложения, использующего библиотеку Qt, поскольку в данной библиотеке под ним понимается любой класс, производный от класса `QWidget`. Однако, с точки зрения концепции Документ/Представление, под классом представления понимается не любой потомок класса `QWidget`, а только тот, который может взаимодействовать с объектами классов документов.

Рассмотрим сначала файл заголовка созданного нами класса представления, доступ к которому можно получить, щелкнув левой кнопкой мыши на имени класса `QtMDIView` в каталоге **Classes** вкладки **Classes** окна иерархических списков. В листинге 4.3 приведен текст этого файла без шаблона и комментария к классу.

Листинг 4.3. Файл заголовка класса представления

```
#ifndef QTMDIVIEW_H
#define QTMDIVIEW_H

// включение файлов заголовков библиотеки Qt
#include <qwidget.h>

class QtMDIDoc;

class QtMDIView : public QWidget
{
 Q_OBJECT

 friend QtMDIDoc;

 public:
  /** Конструктор класса представления
   * @param pDoc содержит объект документа, отображаемого представлением.
```

```
    Создайте документ до вызова конструктора
   * или свяжите существующий документ с новым дочерним окном MDIt.*/
  QtMDIView(QtMDIDoc* pDoc, QWidget* parent, const char *name, int wflags);
  /** Деструктор главного объекта представления */
  ~QtMDIView();
  /** возвращает указатель на связанный с представлением документ */
  QtMDIDoc *getDocument() const;
  /** вызывается для перерисовки представления после
    внесения изменений в документ */
  void update(QtMDIView* pSender);
  /** вызывается функцией QtMDIApp::slotFilePrint() для вывода
    представления на печать*/
  void print(QPrinter *pPrinter);

 protected:
   virtual void closeEvent(QCloseEvent*);

   QtMDIDoc *doc;

 private:

};

#endif
```

Класс `QtMDIView` является непосредственным потомком класса `QWidget`, поэтому все переменные и функции, обеспечивающие его соответствие требованиям концепции Документ/Представление, включаются в сам класс, а не наследуются им от базового класса. Практически, в заготовку класса включены только эти переменные и функции.

Рассмотрим теперь реализацию класса представления, доступ к которой можно получить, щелкнув правой кнопкой мыши в окне редактирования файла qtmdiview.h и выбрав в появившемся контекстном меню команду **Switch Header/Source**. В листинге 4.4 приведен текст этого файла без шаблона.

Листинг 4.4. Файл реализации класса представления

```
// Файлы заголовков библиотеки Qt
#include <qprinter.h>
#Include <qpainter.h>

// Файлы заголовков классов приложения
#include "qtmdiview.h"
#include "qtmdidoc.h"

QtMDIView::QtMDIView(QtMDIDoc* pDoc, QWidget *parent, const char* name,
int wflags)
 : QWidget(parent, name, wflags)
```

```
{
  doc=pDoc;
}

QtMDIView::~QtMDIView()
{
}

QtMDIDoc *QtMDIView::getDocument() const
{
 return doc;
}

void QtMDIView::update(QtMDIView* pSender){
 if(pSender != this)
  repaint();
}

void QtMDIView::print(QPrinter *pPrinter)
{
 if (pPrinter->setup(this))
 {
  QPainter p;
  p.begin(pPrinter);

  ////////////////////////////////
  // ЧТО ДЕЛАТЬ: добавить функции вывода представления на печать
  ////////////////////////////////

  p.end();
 }
}

void QtMDIView::closeEvent(QCloseEvent*)
{
 // ОСТАВЬТЕ ЭТУ ФУНКЦИЮ ПУСТОЙ: ОБЪЕКТ ДАННОГО КЛАССА БУДЕТ ЗАКРЫТ
ФИЛЬТРОМ СОБЫТИЙ КЛАССА QtMDIApp
 // потому следует предотвратить вызов функции closeEvent класса QWidget.
}
```

Как видно из листинга 4.4, единственной операцией в заготовке конструктора класса представления является сохранение указателя на связанный с ним объект класса документа. Все остальные аргументы данного конструктора просто передаются конструктору его базового класса. Поскольку в конструкторе класса не создавалось никаких динамических объектов, деструктор данного класса пуст.

Реализация остальных функций данного класса предельно проста. Функция `getDocument` возвращает значение внутренней переменной данного класса

`doc`, содержащей указатель на связанный с ним объект класса документа, а функция `update` проверяет, что сообщение о перерисовке не было послано данным объектом класса представления, и, если это не так, вызывает функцию `QWidget::repaint` для перерисовки представления.

Функция `print`, используемая для вывода представления на печать, также имеет достаточно простую структуру, но она не будет здесь рассматриваться, поскольку это тема отдельного рассмотрения. Функция `QWidget::closeEvent` перегружается в данном классе для того, чтобы в нем не вызывалась соответствующая функция базового класса.

Рассмотренный нами класс представления является только заготовкой. Для того чтобы он превратился в полноценный класс представления, в него нужно внести следующие изменения:

- перегрузить виртуальные функции класса `QWidget`, обрабатывающие сигналы, поступающие от клавиатуры и мыши. В этих функциях должны реализовываться все операции по работе со связанным с данным представлением документом;
- перегрузить функции `paintEvent` и `repaint` класса `QWidget` для вывода представления на экран;
- реализовать вывод представления на печать, если в этом есть необходимость.

Класс приложения

Опытный программист, работавший в среде Windows, при рассмотрении списка изменений, которые необходимо внести в класс представления, задаст резонный вопрос: "Почему в классе представления обрабатываются только сигналы, поступающие от клавиатуры и мыши? А как же сигналы, поступающие от элементов пользовательского интерфейса?" В ответе на этот вопрос заключается основное отличие приложений, работающих в среде Windows, от приложений, созданных KDevelop.

Дело в том, что разработчики библиотеки Qt не заложили в нее поддержку концепции Документ/Представление. Такое отношение к данной концепции можно объяснить тем, что большинство приложений X Window не являются многооконными, и в них отсутствует необходимость централизованного хранения информации, отображаемой в различных окнах, и синхронизации их содержимого при внесении изменений в одно из этих окон. Первым исключением из этого правила стало приложение KOffice, представляющее собой альтернативу MS Office в интегрированной среде KDE.

Необходимость реализации данной концепции в логически завершенной библиотеке, изначально ее поддерживавшей, не могло не сказаться на качестве этой реализации. Основным ее недостатком является то, что классы

представлений должны общаться с элементами пользовательского интерфейса исключительно через объект класса приложения. В то время как в среде Windows каждому классу представления в рамках шаблона документа сопоставляется свое меню и панель инструментов, что позволяет очень просто решить вопрос работы в рамках одного приложения с различными типами документов, каждый из которых требует использования специального набора элементов пользовательского интерфейса.

Поскольку в приложениях KDevelop не создается специального файла описания ресурсов, вся информация о них включается в файл реализации класса приложения, что существенно увеличивает его размер, заполняя его большим числом однотипных структур. Поэтому здесь мы рассмотрим только структуру функций данного класса.

Доступ к файлу реализации класса приложения можно получить, раскрыв каталог **qtmdi** во вкладке **Files** окна иерархических списков и щелкнув левой кнопкой мыши на имени файла qtmdi.cpp.

В листинге 4.5 приведен текст конструктора и деструктора данного класса.

Листинг 4.5. Конструктор и деструктор класса приложения

```
QtMDIApp::QtMDIApp()
{
 setCaption(tr("QtMDI " VERSION));

 printer = new QPrinter;
 untitledCount=0;
 pDocList = new QList<QtMDIDoc>();
 pDocList->setAutoDelete(true);

 ///////////////////////////////////////////////////////////////////
 // вызов функций инициализации
 initView();
 initActions();
 initMenuBar();
 initToolBar();
 initStatusBar();
 resize( 450, 400);

 viewToolBar->setOn(true);
 viewStatusBar->setOn(true);
}

QtMDIApp::~QtMDIApp()
{
 delete printer;
}
```

Прежде всего, в конструкторе класса приложения вызывается функция QWidget::setCaption, устанавливающая заголовок главного окна приложения. Текст заголовка преобразуется функцией QObject::tr для получения его локализованной версии. Затем в конструкторе класса создается объект класса принтера, обнуляется счетчик неименованных документов untitledCount, который будет увеличиваться при каждом выборе пользователем команды меню **File | New**, создается список объектов классов документов приложения и для него, вызовом функции QCollection::setAutoDelete, устанавливается режим уничтожения исключаемых из него объектов.

После этого в конструкторе класса вызываются функции инициализации различных компонентов приложения, которые будут рассмотрены далее, и для переменных viewToolBar и viewStatusBar вызываются их функции QAction::setOn, отмечающие соответствующие им команды в меню.

Примечание

После завершения инициализации элементов пользовательского интерфейса в главном окне приложения в конструкторе класса приложения вызывается функция QWidget::resize, устанавливающая фиксированные размеры выводимого окна, что нельзя признать хорошим стилем программирования, учитывая, что создаваемое приложение должно работать на различных компьютерах, имеющих разное разрешение экрана.

В деструкторе класса уничтожается созданный в его конструкторе динамический объект класса QPrinter. Объект класса QList уничтожать не надо, поскольку он будет автоматически уничтожен, как только в приложении не останется ни одного документа.

Теперь рассмотрим подробнее функции инициализации компонентов приложения.

Первой в конструкторе класса вызывается функция initView, инициализирующая главное окно приложения. Текст этой функции приведен в листинге 4.6.

Листинг 4.6. Функция initView

```
void QtMDIApp::initView()
{
 ////////////////////////////////////////////////////////////////////////
 // инициализация главного окна приложения
 QVBox* view_back = new QVBox( this);
 view_back->setFrameStyle( QFrame::StyledPanel | QFrame::Sunken);
 pWorkspace = new QWorkspace( view_back);
 setCentralWidget(view_back);
}
```

Прежде всего, в главном окне приложения создается объект представления QVBox, располагающий свои дочерние окна одно под другим. Для вновь созданного объекта вызывается функция QFrame::setFrameStyle, устанавливающая для его дочерних окон стиль заглубленных панелей. Затем создается объект рабочей области окна, которому в качестве аргумента передается указатель на созданный нами объект представления. После этого объект представления QVBox вызовом функции QMainWindow::setCentralWidget делается центральным объектом представления приложения, вокруг которого должны располагаться панель меню, панели инструментов и строка состояния.

После инициализации главного окна приложения в конструкторе класса вызывается функция initActions, инициализирующая абстрактные объекты команд меню и панели инструментов. Текст этой функции приведен в листинге 4.7.

Листинг 4.7. Функция initActions

```
void QtMDIApp::initActions()
{
 QPixmap openIcon, saveIcon, newIcon;
 newIcon = QPixmap(filenew);
 openIcon = QPixmap(fileopen);
 saveIcon = QPixmap(filesave);


 fileNew = new QAction(tr("New File"), newIcon, tr("&New"),
QAccel::stringToKey(tr("Ctrl+N")), this);
 fileNew->setStatusTip(tr("Creates a new document"));
 fileNew->setWhatsThis(tr("New File\n\nCreates a new document"));
 connect(fileNew, SIGNAL(activated()), this, SLOT(slotFileNew()));

 fileOpen = new QAction(tr("Open File"), openIcon, tr("&Open..."), 0, this);
 fileOpen->setStatusTip(tr("Opens an existing document"));
 fileOpen->setWhatsThis(tr("Open File\n\nOpens an existing document"));
 connect(fileOpen, SIGNAL(activated()), this, SLOT(slotFileOpen()));

 fileSave = new QAction(tr("Save File"), saveIcon, tr("&Save"),
QAccel::stringToKey(tr("Ctrl+S")), this);
 fileSave->setStatusTip(tr("Saves the actual document"));
 fileSave->setWhatsThis(tr("Save File.\n\nSaves the actual document"));
 connect(fileSave, SIGNAL(activated()), this, SLOT(slotFileSave()));

 fileSaveAs = new QAction(tr("Save File As"), tr("Save &as..."), 0,
this);
 fileSaveAs->setStatusTip(tr("Saves the actual document under a new
filename"));
```

```
 fileSaveAs->setWhatsThis(tr("Save As\n\nSaves the actual document under
a new filename"));
 connect(fileSaveAs, SIGNAL(activated()), this, SLOT(slotFileSave()));

 fileClose = new QAction(tr("Close File"), tr("&Close"),
QAccel::stringToKey(tr("Ctrl+W")), this);
 fileClose->setStatusTip(tr("Closes the actual document"));
 fileClose->setWhatsThis(tr("Close File\n\nCloses the actual document"));
 connect(fileClose, SIGNAL(activated()), this, SLOT(slotFileClose()));

 filePrint = new QAction(tr("Print File"), tr("&Print"),
QAccel::stringToKey(tr("Ctrl+P")), this);
 filePrint->setStatusTip(tr("Prints out the actual document"));
 filePrint->setWhatsThis(tr("Print File\n\nPrints out the actual
document"));
 connect(filePrint, SIGNAL(activated()), this, SLOT(slotFilePrint()));

 fileQuit = new QAction(tr("Exit"), tr("E&xit"),
QAccel::stringToKey(tr("Ctrl+Q")), this);
 fileQuit->setStatusTip(tr("Quits the application"));
 fileQuit->setWhatsThis(tr("Exit\n\nQuits the application"));
 connect(fileQuit, SIGNAL(activated()), this, SLOT(slotFileQuit()));

 editCut = new QAction(tr("Cut"), tr("Cu&t"),
QAccel::stringToKey(tr("Ctrl+X")), this);
 editCut->setStatusTip(tr("Cuts the selected section and puts it to the
clipboard"));
 editCut->setWhatsThis(tr("Cut\n\nCuts the selected section and puts it
to the clipboard"));
 connect(editCut, SIGNAL(activated()), this, SLOT(slotEditCut()));

 editCopy = new QAction(tr("Copy"), tr("&Copy"),
QAccel::stringToKey(tr("Ctrl+C")), this);
 editCopy->setStatusTip(tr("Copies the selected section to the
clipboard"));
 editCopy->setWhatsThis(tr("Copy\n\nCopies the selected section to the
clipboard"));
 connect(editCopy, SIGNAL(activated()), this, SLOT(slotEditCopy()));

 editUndo = new QAction(tr("Undo"), tr("&Undo"),
QAccel::stringToKey(tr("Ctrl+Z")), this);
 editUndo->setStatusTip(tr("Reverts the last editing action"));
 editUndo->setWhatsThis(tr("Undo\n\nReverts the last editing action"));
 connect(editUndo, SIGNAL(activated()), this, SLOT(slotEditUndo()));

 editPaste = new QAction(tr("Paste"), tr("&Paste"),
QAccel::stringToKey(tr("Ctrl+V")), this);
 editPaste->setStatusTip(tr("Pastes the clipboard contents to actual
position"));
```

```
 editPaste->setWhatsThis(tr("Paste\n\nPastes the clipboard contents
to actual position"));
 connect(editPaste, SIGNAL(activated()), this, SLOT(slotEditPaste()));

 viewToolBar = new QAction(tr("Toolbar"), tr("Tool&bar"), 0, this, 0,
true);
 viewToolBar->setStatusTip(tr("Enables/disables the toolbar"));
 viewToolBar->setWhatsThis(tr("Toolbar\n\nEnables/disables the
toolbar"));
 connect(viewToolBar, SIGNAL(toggled(bool)), this,
SLOT(slotViewToolBar(bool)));

 viewStatusBar = new QAction(tr("Statusbar"), tr("&Statusbar"), 0,
 this, 0, true);
 viewStatusBar->setStatusTip(tr("Enables/disables the statusbar"));
 viewStatusBar->setWhatsThis(tr("Statusbar\n\nEnables/disables the
statusbar"));
 connect(viewStatusBar, SIGNAL(toggled(bool)), this,
SLOT(slotViewStatusBar(bool)));

 windowNewWindow = new QAction(tr("New Window"), tr("&New Window"), 0,
this);
 windowNewWindow->setStatusTip(tr("Opens a new view for the current
document"));
 windowNewWindow->setWhatsThis(tr("New Window\n\nOpens a new view for the
current document"));
 connect(windowNewWindow, SIGNAL(activated()), this,
SLOT(slotWindowNewWindow()));

 windowCascade = new QAction(tr("Cascade"), tr("&Cascade"), 0, this);
 windowCascade->setStatusTip(tr("Cascades all windows"));
 windowCascade->setWhatsThis(tr("Cascade\n\nCascades all windows"));
 connect(windowCascade, SIGNAL(activated()), pWorkspace,
SLOT(cascade()));

 windowTile = new QAction(tr("Tile"), tr("&Tile"), 0, this);
 windowTile->setStatusTip(tr("Tiles all windows"));
 windowTile->setWhatsThis(tr("Tile\n\nTiles all windows"));
 connect(windowTile, SIGNAL(activated()), pWorkspace, SLOT(tile()));

 windowAction = new QActionGroup(this, 0, false);
 windowAction->insert(windowNewWindow);
 windowAction->insert(windowCascade);
 windowAction->insert(windowTile);

 helpAboutApp = new QAction(tr("About"), tr("&About..."), 0, this);
 helpAboutApp->setStatusTip(tr("About the application"));
 helpAboutApp->setWhatsThis(tr("About\n\nAbout the application"));
 connect(helpAboutApp, SIGNAL(activated()), this, SLOT(slotHelpAbout()));

}
```

Для временного хранения битовых образов кнопок панели инструментов в данной функции создаются объекты класса QPixmap, конструкторам которых передаются имена соответствующих файлов. Эти объекты применяются в качестве второго аргумента конструктора класса QAction, используемого для работы с кнопкой панели инструментов. Если команда меню не дублируется в панели инструментов, используется другой конструктор класса QAction, не содержащий данного аргумента. В остальном эти конструкторы идентичны.

Первым аргументом конструктора класса QAction является локализованное краткое описание команды, возвращаемое функцией QObject::tr, которой в качестве аргумента передается описание этой команды на английском языке. Это описание будет выводиться при перемещении указателя мыши на соответствующую кнопку меню. После объекта значка, а при его отсутствии после описания команды, передается локализованное имя этой команды, которое будет выводиться в меню. Затем следует код комбинации клавиш, используемой для вызова этой команды. Для получения этого кода служит статическая функция QAccel::stringToKey, в качестве аргумента которой передается текстовое описание соответствующей комбинации клавиш. Если данной команде не соответствует никакая комбинация клавиш, в данном аргументе передается нулевое значение. Последним аргументом конструктора класса QAction является указатель на объект класса, которому будет принадлежать создаваемый объект.

Для созданного объекта класса QAction вызывается функция QAction::setStatusTip, устанавливающая локализованный текст, который будет выводиться в строке состояния главного окна приложения при получении данной командой фокуса ввода, и функция QAction::setWhatsThis, устанавливающая локализованный текст, который будет выводиться в специальном окне справки. После всего этого вызывается функция QObject::connect, связывающая сигнал объекта класса QAction с приемником объекта класса приложения. На этом инициализация отдельной команды заканчивается.

После инициализации команд конструктор класса приложения приступает к инициализации вызывающего эти команды объекта меню. Эта операция производится в функции initMenuBar, текст которой приведен в листинге 4.8.

Листинг 4.8. Функция initMenuBar

```
void QtMDIApp::initMenuBar()
{
  ///////////////////////////////////////////////////////////////////
  // ПАНЕЛЬ МЕНЮ

  ///////////////////////////////////////////////////////////////////
  // команды раскрывающегося меню pFileMenu
  pFileMenu=new QPopupMenu();
```

```
fileNew->addTo(pFileMenu);
fileOpen->addTo(pFileMenu);
fileClose->addTo(pFileMenu);
pFileMenu->insertSeparator();
fileSave->addTo(pFileMenu);
fileSaveAs->addTo(pFileMenu);
pFileMenu->insertSeparator();
filePrint->addTo(pFileMenu);
pFileMenu->insertSeparator();
fileQuit->addTo(pFileMenu);

///////////////////////////////////////////////////////////////////
// команды раскрывающегося меню editMenu
pEditMenu=new QPopupMenu();
editUndo->addTo(pEditMenu);
pEditMenu->insertSeparator();
editCut->addTo(pEditMenu);
editCopy->addTo(pEditMenu);
editPaste->addTo(pEditMenu);

///////////////////////////////////////////////////////////////////
// команды раскрывающегося меню viewMenu
pViewMenu=new QPopupMenu();
pViewMenu->setCheckable(true);
viewToolBar->addTo(pViewMenu);
viewStatusBar->addTo(pViewMenu);
///////////////////////////////////////////////////////////////////
// ВСТАВЬТЕ СЮДА КОМАНДЫ ПОЛЬЗОВАТЕЛЬСКИХ МЕНЮ

///////////////////////////////////////////////////////////////////
// команды раскрывающегося меню windowMenu
pWindowMenu = new QPopupMenu(this);
pWindowMenu->setCheckable(true);
connect(pWindowMenu, SIGNAL(aboutToShow()), this,
SLOT(windowMenuAboutToShow()));

///////////////////////////////////////////////////////////////////
// команды раскрывающегося меню helpMenu
pHelpMenu=new QPopupMenu();
helpAboutApp->addTo(pHelpMenu);
pHelpMenu->insertSeparator();
pHelpMenu->insertItem(tr("What's &This"), this, SLOT(whatsThis()),
SHIFT+Key_F1);

menuBar()->insertItem(tr("&File"), pFileMenu);
menuBar()->insertItem(tr("&Edit"), pEditMenu);
menuBar()->insertItem(tr("&View"), pViewMenu);
```

```
 menuBar()->insertItem(tr("&Window"), pWindowMenu);
 menuBar()->insertItem(tr("&Help"), pHelpMenu);

}
```

Инициализация команд меню производится в функции `initMenuBar`. Поскольку в приложениях Qt отсутствует файл описания ресурсов меню, данный файл одновременно является файлом описания ресурса меню и все раскрывающиеся меню и их команды будут появляться в главном окне в том же порядке, в котором они были описаны в данной функции.

Для работы с раскрывающимся меню в данной функции создается объект класса `QPopupMenu`. С целью добавления в него новой команды используется функция объекта добавляемой команды `QAction::addTo`, которой в качестве аргумента передается указатель на объект класса раскрывающегося меню, в которое она будет добавлена. Для включения разделителя между группами команд в раскрывающемся меню используется функция `QMenuData::insertSeparator`, а для того, чтобы у команд меню могли появляться флажки, применяется функция `QPopupMenu::setCheckable`.

Поскольку содержимое меню **Window** и состояние его команд зависит от наличия в приложении окон, вместо инициализации этого меню вызывается функция `QObject::connect`, связывающая сигнал `QPopupMenu::aboutToShow`, посылаемый перед тем, как раскрывающееся меню будет выведено на экран, с функцией его обработки `windowMenuAboutToShow`, текст которой приведен в листинге 4.9. Такой способ вызова меню позволяет обновлять его содержимое перед каждым раскрытием.

Листинг 4.9. Функция `windowMenuAboutToShow`

```
void QtMDIApp::windowMenuAboutToShow()
{
 pWindowMenu->clear();
 windowNewWindow->addTo(pWindowMenu);
 windowCascade->addTo(pWindowMenu);
 windowTile->addTo(pWindowMenu);

 if ( pWorkspace->windowList().isEmpty())
 {
  windowAction->setEnabled(false);
 }
 else
 {
  windowAction->setEnabled(true);
 }

 pWindowMenu->insertSeparator();
```

```
 QWidgetList windows = pWorkspace->windowList();
 for ( int i = 0; i < int(windows.count()); ++i)
 {
  int id = pWindowMenu->insertItem(QString("&%1 ").arg(i+1) +
windows.at(i)->caption(), this, SLOT( windowMenuActivated( int)));
  pWindowMenu->setItemParameter( id, i);
  pWindowMenu->setItemChecked( id, pWorkspace->activeWindow() ==
windows.at(i));
 }
}
```

Прежде всего, содержимое раскрывающегося меню очищается вызовом функции `QMenuData::clear`. После этого в него помещаются стандартные команды меню и, вызовом функции `QList::isEmpty`, производится проверка того, имеется ли в рабочей области хотя бы одно дочернее окно. В зависимости от результата проверки, вызовом функции `QActionGroup::setEnabled` разрешается или запрещается доступ к только что введенным командам меню. После этого в меню помещается разделитель и выводится список окон приложения. Для чего вызовом функции `QWorkspace::windowList` получается список дочерних окон приложения. Этот список просматривается в цикле, для каждого его элемента вызовом функции `QMenuData::insertItem` создается команда меню, используемая для активизации этого окна. Первым аргументом данной функции является текст создаваемой команды меню, вторым — класс, которому будет послано сообщение о выборе данной команды меню, а третьим — функция обработки данного сообщения.

Следующим оператором в цикле является вызов функции `QMenuData::setItemParameter`, сопоставляющий идентификатору команды меню целочисленный параметр, который будет передаваться функции обработки данной команды при ее вызове.

В завершение работы цикла в нем вызывается функция `QMenuData::setItemChecked`, устанавливающая флажок у команды активизации уже активного окна.

После создания и заполнения всех объектов раскрывающихся меню в функции `initMenuBar` для каждого из них вызывается функция `QMenuBar::insertItem`, добавляющая их в панель меню. Для получения текущего объекта класса `QMenuBar` используется функция `QMainWindow::menuBar`.

После инициализации панели меню в конструкторе класса приложения инициализируются панель инструментов и строка состояния, как это показано в листинге 4.10.

Листинг 4.10. Инициализация панели инструментов и строки состояния

```
void QtMDIApp::initToolBar()
{
```

```
 ////////////////////////////////////////////////////////////////////////
 // ПАНЕЛЬ ИНСТРУМЕНТОВ
 fileToolbar = new QToolBar(this, "file operations");
 fileNew->addTo(fileToolbar);
 fileOpen->addTo(fileToolbar);
 fileSave->addTo(fileToolbar);
 fileToolbar->addSeparator();
 QWhatsThis::whatsThisButton(fileToolbar);

}

void QtMDIApp::initStatusBar()
{
 ////////////////////////////////////////////////////////////////////////
 // СТРОКА СОСТОЯНИЯ
 statusBar()->message(tr("Ready."));
}
```

Прежде всего, в функции `initToolBar` создается пустой объект панели инструментов с именем **file operations**. Этот объект будет отображаться в верхней части главного окна приложения. Добавление команд в только что созданную панель инструментов производится функцией `QAction::addTo`, использовавшейся до этого для добавления этих команд в раскрывающиеся меню. После добавления в панель инструментов стандартных команд работы с файлом в ней, вызовом функции `QToolBar::addSeparator`, выводится разделитель и создается новая кнопка **What's This**, для чего используется статическая функция `QWhatsThis::whatsThisButton`.

Инициализация строки состояния, производимая в функции `initStatusBar`, предельно проста: в ней, с использованием функции `QStatusBar::message`, выводится локализованное сообщение о готовности приложения к работе.

Многооконное приложение KDE

Помимо приложений, применяющих библиотеку Qt, среда разработки KDevelop позволяет создавать приложения KDE, использующие библиотеку данной интегрированной среды. Эти приложения также могут задействовать концепцию Документ/Представление, но она реализуется в этих приложениях немного иначе. Чтобы разобраться в этих различиях, создадим демонстрационное приложение **KDEMDI**, текст которого можно найти на прилагаемом к книге CD-диске.

Чтобы самостоятельно создать заготовку многооконного приложения KDE:

1. Выберите команду меню **Project | New**. Появится диалоговое окно **ApplicationWizard**.

2. В окне иерархического списка диалогового окна раскройте папку **KDE**, выделите в ней строку **KDE MDI** и нажмите кнопку **Next**. Появится второе окно мастера **ApplicationWizard**.
3. В текстовое поле **Project name** введите имя проекта **KDEMDI**, заполните остальные текстовые поля корректной информацией (или оставьте их без изменений, поскольку этот проект не будет распространяться) и нажмите кнопку **Create**. Появится последнее окно мастера **ApplicationWizard**, в котором будет выводиться отчет о процессе создания приложения.
4. По завершении создания проекта нажмите кнопку **Exit**. Окно мастера создания приложений закроется.

Класс документа

Из сравнения файлов заголовков классов документов в приложении KDE и в приложении Qt можно увидеть, что различия заключаются всего в нескольких функциях и переменных. Объявим эти различия не принципиальными и сразу же перейдем к файлам реализации этих классов.

В листинге 4.2 был приведен фрагмент реализации конструктора и деструктора класса документа приложения Qt, а также реализация функций создания, открытия и сохранения документа. Поскольку реализация конструктора и деструктора класса документа, а также реализация функции создания нового документа не претерпели изменений в приложении KDE, рассмотрим только функции открытия и сохранения документа, приведенные в листинге 4.11.

Листинг 4.11. Функции openDocument и saveDocument

```
bool KDEMDIDoc::openDocument(const KURL &url, const char *format /*=0*/)
{

 QString tmpfile;
 KIO::NetAccess::download( url, tmpfile);

 /////////////////////////////////////////////////////
 QFile f( tmpfile);
 if ( !f.open( IO_ReadOnly))
  return false;
 /////////////////////////////////////////////////////
 // ЧТО ДЕЛАТЬ: Поместите сюда функции открытия вашего документа
 /////////////////////////////////////////////////////
 f.close();

 /////////////////////////////////////////////////////
 KIO::NetAccess::removeTempFile( tmpfile);
 doc_url=url;
```

```
 modified=false;
 return true;
}

bool KDEMDIDoc::saveDocument(const KURL &url, const char *format /*=0*/)
{
// QFile f( filename);
// if ( !f.open( IO_WriteOnly))
//  return false;
//
// ////////////////////////////////////////////////
// // ЧТО ДЕЛАТЬ: Поместите сюда функции сохранения вашего документа
// ////////////////////////////////////////////////
//
// f.close();
//
// modified=false;
// m_filename=filename;
// m_title=QFileInfo(f).fileName();
 return true;
}
```

В функции `openDocument`, реализующей чтение информации из файла, легко можно обнаружить блоки, взятые из заготовки одноименной функции приложения Qt. Однако приложения KDE в большой степени ориентированы на работу с Интернетом и все пути к файлам рассматриваются в них как URL. Поэтому первый аргумент данной функции имеет тип `KURL`, а не `QString`, как это имело место в приложениях Qt.

Работать с удаленным ресурсом напрямую крайне неудобно и требует больших затрат вычислительных ресурсов. Поэтому при открытии документа его копию следует поместить на локальный диск. Эта операция выполняется статической функцией `KIO::NetAccess::download`, в первом аргументе которой передается загружаемый URL, а во втором — строка, содержащая имя файла, в который будет помещен загружаемый ресурс. Если во втором аргументе передается пустая строка, то в каталоге /tmp создается временный файл, путь к которому и возвращается во втором аргументе функции. Если загружаемый URL определяет не удаленный ресурс, а файл на локальном диске, то загрузки ресурса не происходит, а во втором аргументе функции возвращается путь к этому файлу.

Теперь, когда функция имеет объект класса `QString`, содержащий путь к файлу на локальном диске, она может использовать тот же шаблон, что и одноименная функция приложения Qt: в ней создается объект класса `QFile`, он открывает файл только для чтения, разработчику предлагается считать из файла всю необходимую информацию и файл закрывается.

После того как вся информация из временного файла прочитана и необходимость в нем отпадает, он уничтожается вызовом статической функции `KIO::NetAccess::removeTempFile` (конечно, файл уничтожается только в том случае, если он был временным). URL файла сохраняется для возможного дальнейшего использования, флаг наличия изменений в документе сбрасывается и функция `openDocument` завершает свою работу.

Примечание

Предложенный метод открытия документа вызывает вполне обоснованные вопросы. Если документ так мал, что может полностью разместиться в памяти, зачем сохранять его URL? Если же он настолько велик, что к нему придется неоднократно обращаться для подкачки информации, зачем уничтожать временный файл? Вполне возможно, что разработчику приложения придется внести коррективы в эту функцию для повышения эффективности работы приложения.

Функция `saveDocument`, вызываемая для сохранения документа в файле, имеет очень странную структуру: она не выполняет никаких действий, но в ней заремаркирован полный текст одноименной функции приложения Qt, в котором использованы идентификаторы несуществующих аргументов функции и членов класса. Вероятно, разработчики среды KDevelop посчитали, что сохранение документа в Интернете настолько отличается от сохранения его на локальном диске, что здесь нельзя использовать общий шаблон.

Класс представления

Заголовок и реализация класса представления в приложениях KDE практически не отличаются от заголовка и реализации этого класса в приложениях Qt. Чтобы как-то заполнить этот раздел, отсутствие которого нарушит структуру данной главы, приведем в примечании текст комментария, автоматически помещаемый в файл заголовка класса представления мастером создания приложений.

Примечание

Класс `KDEMDIView` предоставляет окно представления связанному с ним объекту класса документа, отображаемое в дочернем окне многооконного приложения, управляемого объектом класса `KDEMDIApp`. Класс `KDEMDIApp` включает в себя метод `eventFilter()`, обрабатывающий сообщения типа `QEvent::Close`, поступающие от каждого объекта класса `KDEMDIView`.

Документ, связанный с объектом класса представления, хранит список всех объектов класса представления, в которых отображается его содержимое, поскольку один объект класса документа может быть связан с несколькими объектами класса представления. Объекты класса представления создаются в функции `KDEMDIApp::createClient()` и автоматически добавляются в соответствующий список объекта класса документа.

Объект класса представления является потомком класса `QWidget`. Помимо других специализированных классов представления, в качестве базового клас-

са объекта представления может выступать класс QMainWindow, что позволяет установить рабочую область представления путем выбора другого объекта класса представления в качестве главного окна (QMainWindow::setMainWidget()).

ВНИМАНИЕ: Не следует вносить изменения в процедуру закрытия представления (НЕ СЛЕДУЕТ ВЫЗЫВАТЬ функцию QWidget::closeEvent(e) в функции closeEvent()), поскольку установленный фильтр событий может обрабатывать событие только до его передачи объекту класса. С полученным объектом класса QCloseEvent не следует производить никаких действий, т. к. функция eventFilter уже вызвала метод accept() или ignore(). При повторном вызове функции QWidget::closeEvent() используемый по умолчанию обработчик событий уничтожит окно даже в том случае, если функция eventFilter решит игнорировать это событие.

Класс приложения

Основные отличия между приложениями Qt и KDE заключаются в их классах приложений. Различия заключаются не столько в исключении некоторых функций из класса приложения KDE, сколько в их реализации. Поэтому сразу же перейдем к рассмотрению реализации конструктора класса приложения и вызываемых в нем функций.

Текст конструктора класса приложения приведен в листинге 4.12.

Листинг 4.12. Конструктор и деструктор класса приложения

```
KDEMDIApp::KDEMDIApp():KMainWindow(0,"KDEMDI")
{
 config=kapp->config();
 printer = new QPrinter;
 untitledCount=0;
 pDocList = new QList<KDEMDIDoc>();
 pDocList->setAutoDelete(true);

 ///////////////////////////////////////////////////////////////////////////
 // вызов функций инициализации
 initStatusBar();
 initView();
 initActions();

 readOptions();

 ///////////////////////////////////////////////////////////////////////////
 // запрет действий после запуска приложения
 fileSave->setEnabled(false);
 fileSaveAs->setEnabled(false);
 filePrint->setEnabled(false);
```

```
  editCut->setEnabled(false);
  editCopy->setEnabled(false);
  editPaste->setEnabled(false);
}
```

При сравнении конструкторов классов KDEMDIApp и QtMDIApp, прежде всего, следует обратить внимание на то, что в конструкторе класса KDEMDIApp заголовок главного окна приложения задается в аргументе конструктора базового класса, а в конструкторе класса QtMDIApp для этого используется специальная функция QWidget::setCaption. Причем, что характерно, в приложении Qt предусматривается локализация заголовка главного окна приложения, а в приложении KDE — нет.

Поскольку приложение KDE хранит информацию о своей конфигурации в системе конфигурации KDE, в первом операторе конструктора класса KDEMDIApp вызывается функция KInstance::config, возвращающая указатель на объект класса KConfig, осуществляющего взаимодействие приложения с системой конфигурации KDE, и полученный указатель сохраняется в специальной переменной.

После этого в конструкторах обоих классов создается объект класса принтера, обнуляется счетчик неименованных документов untitledCount, который будет увеличиваться при каждом выборе пользователем команды меню **File | New**, создается список объектов классов документов приложения и для него, вызовом функции setAutoDelete, устанавливается режим уничтожения исключаемых из него объектов.

Основное отличие конструкторов классов приложений Qt и KDE заключается в наборе функций инициализации компонентов приложения и порядке их вызова: в приложениях KDE не имеется специальных функций для инициализации меню и панели инструментов, а инициализация строки состояния производится до, а не после инициализации команд в функции initActions. Кроме того, в приложениях KDE не устанавливается фиксированный размер главного окна приложения, а также начальное состояние команд меню **View**, поскольку в приложениях KDE эта операция выполняется другим образом.

После инициализации компонентов приложения в конструкторе класса документа вызывается функция KDEMDIApp::readOptions, восстанавливающая состояние элементов пользовательского интерфейса на основании информации, хранящейся в системе конфигурации KDE, и устанавливается режим недоступности для действий, выполнение которых бессмысленно для пустого документа. Для установки режима доступа используется функция KAction::setEnabled, в качестве аргумента которой передается значение флага доступности функции.

Теперь рассмотрим подробнее функции инициализации компонентов приложения.

Первой в конструкторе класса вызывается функция KDEMDIApp::initStatusBar, единственным отличием которой от одноименной функции приложения Qt является то, что вместо функции QStatusBar::message для вывода сообщения в строку состояния используется функция KStatusBar::insertItem. В первом аргументе этой функции передается выводимое в строку состояния сообщение, а во втором — целочисленный идентификатор панели строки состояния, в которую это сообщение будет выводиться.

Следующей в конструкторе класса вызывается функция KDEMDIApp::initView, инициализирующая главное окно приложения. Поскольку она полностью аналогична одноименной функции приложения Qt, здесь эта функция рассматриваться не будет.

Реализация функции KDEMDIApp::initActions, приведенная в листинге 4.13, существенно отличается от реализации одноименной функции приложения Qt.

Листинг 4.13. Функция initActions

```
void KDEMDIApp::initActions()
{
 fileNew = KStdAction::openNew(this, SLOT(slotFileNew()),
actionCollection());
 fileOpen = KStdAction::open(this, SLOT(slotFileOpen()),
actionCollection());
 fileOpenRecent = KStdAction::openRecent(this,
SLOT(slotFileOpenRecent(const KURL&)), actionCollection());
 fileSave = KStdAction::save(this, SLOT(slotFileSave()),
actionCollection());
 fileSaveAs = KStdAction::saveAs(this, SLOT(slotFileSaveAs()),
actionCollection());
 fileClose = KStdAction::close(this, SLOT(slotFileClose()),
actionCollection());
 filePrint = KStdAction::print(this, SLOT(slotFilePrint()),
actionCollection());
 fileQuit = KStdAction::quit(this, SLOT(slotFileQuit()),
actionCollection());
 editCut = KStdAction::cut(this, SLOT(slotEditCut()),
actionCollection());
 editCopy = KStdAction::copy(this, SLOT(slotEditCopy()),
actionCollection());
 editPaste = KStdAction::paste(this, SLOT(slotEditPaste()),
actionCollection());
 viewToolBar = KStdAction::showToolbar(this, SLOT(slotViewToolBar()),
actionCollection());
 viewStatusBar = KStdAction::showStatusbar(this,
SLOT(slotViewStatusBar()), actionCollection());
```

```
windowNewWindow = new KAction(i18n("New &Window"), 0, this,
SLOT(slotWindowNewWindow()), actionCollection(),"window_new_window");
windowTile = new KAction(i18n("&Tile"), 0, this, SLOT(slotWindowTile()),
actionCollection(),"window_tile");
windowCascade = new KAction(i18n("&Cascade"), 0, this,
SLOT(slotWindowCascade()), actionCollection(),"window_cascade");

fileNew->setStatusText(i18n("Creates a new document"));
fileOpen->setStatusText(i18n("Opens an existing document"));
fileOpenRecent->setStatusText(i18n("Opens a recently used file"));
fileSave->setStatusText(i18n("Saves the actual document"));
fileSaveAs->setStatusText(i18n("Saves the actual document as..."));
fileClose->setStatusText(i18n("Closes the actual document"));
filePrint ->setStatusText(i18n("Prints out the actual document"));
fileQuit->setStatusText(i18n("Quits the application"));

editCut->setStatusText(i18n("Cuts the selected section and puts it to
the clipboard"));
editCopy->setStatusText(i18n("Copies the selected section to the
clipboard"));
editPaste->setStatusText(i18n("Pastes the clipboard contents to actual
position"));

viewToolBar->setStatusText(i18n("Enables/disables the toolbar"));
viewStatusBar->setStatusText(i18n("Enables/disables the statusbar"));

windowMenu = new KActionMenu(i18n("&Window"), actionCollection(),
"window_menu");
connect(windowMenu->popupMenu(), SIGNAL(aboutToShow()), this,
SLOT(windowMenuAboutToShow()));

createGUI();

}
```

Для получения указателей на объекты стандартных команд в функции `initActions` используются статические методы класса `KStdAction`. Как правило, этим функциям передаются только три аргумента: указатель на объект класса, в который включен приемник сигнала, посылаемого данной командой, символьный идентификатор приемника и указатель на родительский объект команды. Если пользователь не использует описания ресурсов в формате XML, в третьем аргументе статической функции, как правило, передается указатель на объект приложения (т. е. ключевое слово `this`), в противном случае в нем передается указатель на коллекцию команд, возвращаемый функцией `KXMLGUIClient::actionCollection`. Использование статических методов класса `KStdAction` позволяет существенно сократить объем информации, необходимый для инициализации команды, исключая, например, необходимость указания объекта значка команды и вызывающей ее комбинации клавиш.

Для создания нестандартных команд меню разработчику приходится самому создавать объекты класса `KAction`. В используемом в данном приложении конструкторе данного класса в первом аргументе передается строка, которая будет выводиться в меню, во втором аргументе — вызывающая данную команду комбинация клавиш, в третьем — указатель на объект класса, содержащий приемник сигнала этой команды, в четвертом — символьный идентификатор приемника, в пятом — указатель на родительский объект команды, а в шестом — ее символьный идентификатор.

После инициализации всех объектов команд для каждого из них вызывается функция `KAction::setStatusText`, сопоставляющая ему текст контекстной справки. Как следует из имени функции, она должна была бы выводиться в строке состояния, но она выводится в специальном окне для соответствующих кнопок панели инструментов.

Обратите внимание на то, что в приложениях KDE отсутствует функция `initMenuBar`, и меню **Window** создается в функции `initActions`. Поскольку содержимое меню **Window** и состояние его команд зависит от наличия в приложении окон, вместо инициализации этого меню вызывается функция `QObject::connect`, связывающая сигнал `QPopupMenu::aboutToShow`, посылаемый перед тем, как раскрывающееся меню будет выведено на экран, с функцией его обработки `windowMenuAboutToShow`. Те же самые действия предпринимаются и в приложении Qt, но в приложении KDE для работы с этим меню создается объект класса `KActionMenu`. Поэтому для получения указателя на объект класса `QPopupMenu` используется функция `KAction::popupMenu`.

Текст функции `KDEMDIApp::windowMenuAboutToShow` приведен в листинге 4.14.

Листинг 4.14. Функция `windowMenuAboutToShow`

```
void KDEMDIApp::windowMenuAboutToShow()
{
 windowMenu->popupMenu()->clear();
 windowMenu->insert(windowNewWindow);
 windowMenu->insert(windowCascade);
 windowMenu->insert(windowTile);

 if ( pWorkspace->windowList().isEmpty()){
  windowNewWindow->setEnabled(false);
  windowCascade->setEnabled(false);
  windowTile->setEnabled(false);
 }
 else{
  windowNewWindow->setEnabled(true);
  windowCascade->setEnabled(true);
```

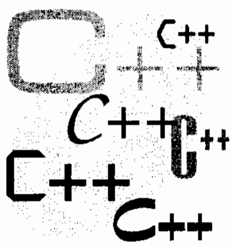
```
  windowTile->setEnabled(true);
 }
 windowMenu->popupMenu()->insertSeparator();

 QWidgetList windows = pWorkspace->windowList();
 for ( int i = 0; i < int(windows.count()); ++i)
 {
  int id = windowMenu->popupMenu()->insertItem(QString("&%1
").arg(i+1)+windows.at(i)->caption(), this, SLOT( windowMenuActivated(
int)));
  windowMenu->popupMenu()->setItemParameter( id, i);
  windowMenu->popupMenu()->setItemChecked( id, pWorkspace->activeWindow()
== windows.at(i));
 }
}
```

Как видно из сравнения листингов 4.9 и 4.14, основные отличия в реализации этих функций заключаются в первых четырех строках. Как уже говорилось выше, для работы с объектом меню в приложениях KDE используется объект класса `KActionMenu`, поэтому указатель на объект класса `QPopupMenu`, необходимый для вызова функции `QMenuData::clear`, получается при вызове функции `KAction::popupMenu`.

Поскольку класс `KAction` не является потомком класса `QAction`, вызов функции `QAction::addTo` для добавления команд в меню приложений KDE невозможен, поэтому для выполнения этой операции используется функция `KActionMenu::insert`. В остальном эти функции практически идентичны, за исключением способа получения указателя на объект класса `QPopupMenu`.

В завершение работы функции `initActions` в ней вызывается функция `KMainWindow::createGUI`, создающая пользовательский интерфейс приложения на основании созданной в функции `initActions` коллекции команд и файла XML, содержащего описание ресурсов приложения.

ГЛАВА 5



Создание элементов пользовательского интерфейса

Среда разработки KDevelop, как правило, используется для создания приложений, способных работать в графических интегрированных средах, предоставляющих пользователю интуитивно понятный графический интерфейс, существенно облегчающий его работу с приложением. Первые попытки создать такой интерфейс предпринимались еще в приложениях, запускавшихся из командной строки. Они, как правило, представляли собой многооконные приложения, пользовательский интерфейс которых состоял из меню и, в некоторых случаях, строки состояния. С появлением интегрированных сред, какой является среда KDE, в приложении появился новый элемент управления — панель инструментов.

Различные *меню* представляют собой неотъемлемую часть любой полноценной программы. Меню располагаются в специальной строке, в которую выведены заголовки раскрывающихся меню. Текст в строке меню может представлять собой и самостоятельную команду, а не заголовок меню, но такая возможность используется крайне редко. Раскрывающееся меню появляется при выборе пользователем его заголовка в строке меню. Как правило, эти меню содержат уже конкретные исполняемые команды, но ничто не мешает поместить в это меню заголовок раскрывающегося меню более низкого уровня. Помимо описанных выше типов меню существует еще контекстное меню, появляющееся при щелчке правой кнопкой мыши. Это меню выводится в текущей позиции указателя мыши.

Панели инструментов, как и меню, являются неотъемлемой частью любой полноценной программы, использующей графический интерфейс. Основное их назначение заключается в предоставлении пользователю быстрого доступа к наиболее часто используемым командам меню. В принципе можно создать панель инструментов, кнопки которой не будут дублироваться в меню, но это существенно усложнит работу с приложением без использования мыши.

Строка состояния располагается в нижней части главного окна программы и служит для отображения дополнительной информации по производимым пользователем действиям.

Все перечисленные выше элементы пользовательского интерфейса включаются в создаваемое приложение автоматически, их набор определяется мастером исходя из типа создаваемого приложения. Поэтому для адаптации заготовки приложения под свои нужды разработчику приходится вносить изменения в содержащийся в ней стандартный набор элементов управления.

Среда разработки KDevelop позволяет создавать приложения, применяющие исключительно библиотеку Qt, а также приложения, преимущественно использующие собственную библиотеку данной среды разработки. Эти приложения существенно различаются по принципам создания и модификации в них элементов пользовательского интерфейса.

Пользовательский интерфейс библиотеки Qt

Для демонстрации принципов работы с элементами пользовательского интерфейса, предоставляемого библиотекой Qt, будет использовано демонстрационное приложение **QtTools**, текст которого можно найти на прилагаемом к книге CD.

Чтобы самостоятельно создать это приложение:

1. Выберите команду меню **Project | New** (Проект | Создать).
2. В иерархическом списке типов создаваемых приложений появившегося окна **ApplicationWizard** (Мастер создания приложений) выделите строку **Qt SDI** (Однооконное приложение Qt), расположенную в папке **Qt**, и нажмите кнопку **Next** (Далее). Раскроется следующая страница мастера создания приложений.
3. В текстовое поле **Project name** (Имя проекта) введите имя проекта **QtTools** и нажмите кнопку **Create** (Создать).
4. После завершения работы мастера по созданию приложения нажмите кнопку **Exit** (Выход). Заготовка приложения будет создана.

Внесение изменений в меню

Меню можно считать основным элементом пользовательского интерфейса приложения, поскольку, как правило, только в нем реализованы все его команды и только с ним можно работать при отсутствии или отказе мыши. Поэтому рассмотрение элементов пользовательского интерфейса приложения мы начнем именно с меню.

В отличие от среды разработки Microsoft Visual Studio, используемой для создания приложений Windows, в среде разработки KDevelop не предусмотрен графический редактор меню. Поэтому все операции по внесению изменений в меню приложения разработчик должен производить вручную.

Чтобы добавить в приложение **QtTools** новое раскрывающееся меню:

1. Откройте приложение **QtTools** на редактирование.
2. В окне иерархических списков раскройте вкладку **Classes** (Классы), щелкните правой кнопкой мыши на имени класса `QtToolsApp`, расположенного в каталоге **Classes**, и выберите в появившемся контекстном меню команду **Add Slot** (Добавить приемник). Появится диалоговое окно **Class Properties** (Свойства класса), раскрытое на вкладке **Slots** (Приемники), как это изображено на рис. 5.1.

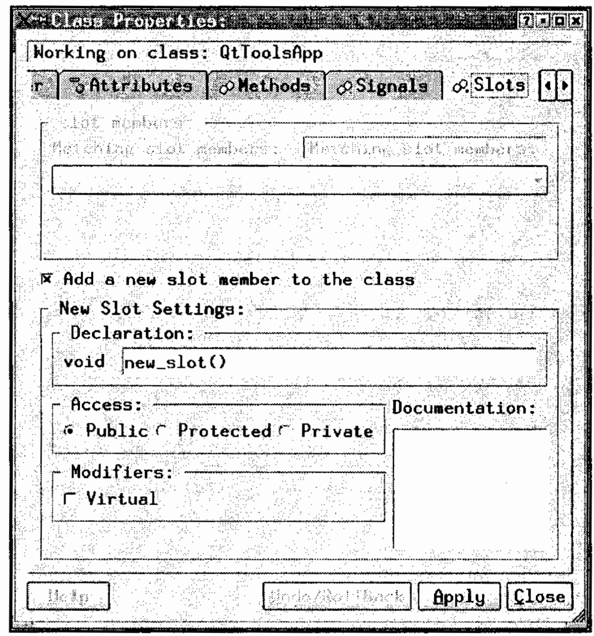


Рис. 5.1. Диалоговое окно **Class Properties**, вкладка **Slots**

3. В текстовом поле **Declaration** замените текст `new_slot()` текстом `slotUserCommand()`, введите в текстовое поле **Documentation** комментарий `Executes user command` (Выполняет пользовательскую команду) и нажмите кнопку **Apply**. В класс будет добавлен новый приемник, откроется окно редактирования файла qttools.cpp и текстовый курсор будет помещен в заготовку приемника.
4. Измените созданный приемник в соответствии с текстом листинга 5.1.

Листинг 5.1. Приемник пользовательской команды

```
/** Выполняет пользовательскую команду */
void QtToolsApp::slotUserCommand()
{
 QMessageBox::information(this, tr("User"), tr("Executing user
command..."));

 statusBar()->message(tr("Ready."));
}
```

5. В функции `QtToolsApp::initActions` после строки `connect(viewStatusBar, SIGNAL(toggled(bool)), this, SLOT(slotViewStatusBar(bool)))` вставьте следующие строки:

```
userCommand = new QAction(tr("User command"),
                          tr("&User Command"), 0, this);
userCommand->setStatusTip(tr("Executes user command"));
userCommand->setWhatsThis(tr("User\n\nExecutes user command"));
connect(userCommand, SIGNAL(activated()), this,
                           SLOT(slotUserCommand()));
```

6. В функции `QtToolsApp::initMenuBar` после строки `// EDIT YOUR APPLICATION SPECIFIC MENUENTRIES HERE` (Поместите сюда специфические команды меню приложения) вставьте следующие строки:

```
// Команды пользовательского меню
 userMenu=new QPopupMenu();
 userCommand->addTo(userMenu);
```

7. В той же функции после строки `menuBar()->insertItem(tr("&View"), pViewMenu);` вставьте строку

```
menuBar()->insertItem(tr("&User"), userMenu);
```

8. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши на имени класса `QtToolsApp` и выберите в появившемся контекстном меню команду **Add member variable** (Добавить переменную члена класса). Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes**, как это изображено на рис. 5.2.
9. В текстовое поле **Type** введите тип переменной `QPopupMenu`, в текстовое поле **Name** — идентификатор переменной `*UserMenu`, в текстовое поле **Documentation** — комментарий `User_menu contains all items of the menubar entry "User"` (Содержит команды меню "User") и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.
10. Повторите пункты 8 и 9 для добавления в класс переменной `*userCommand`, имеющей тип `QAction`, оставив ее без комментария.

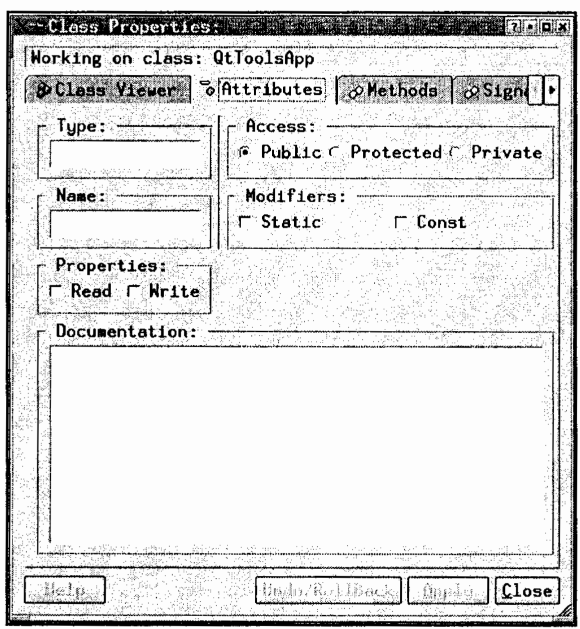


Рис. 5.2. Диалоговое окно **Class Properties**, вкладка **Attributes**

11. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Появится окно приложения, изображенное на рис. 5.3.

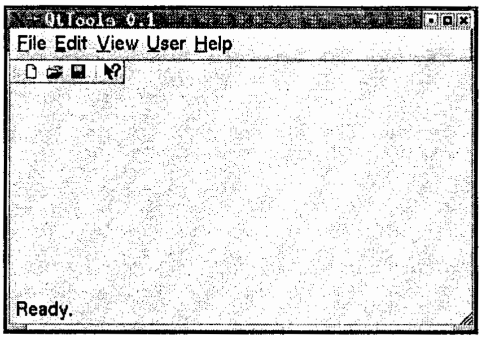


Рис. 5.3. Окно приложения Qt

12. Выберите команду меню **User | User Command** (Пользовательское, Пользовательская команда). Появится окно сообщения, изображенное на рис. 5.4.
13. Нажмите кнопку **OK**. Диалоговое окно исчезнет.
14. Закройте приложение.

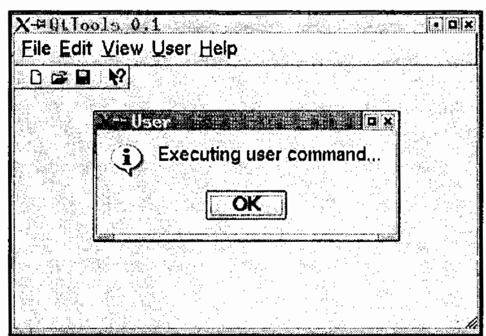


Рис. 5.4. Окно сообщения в приложении **QtTools**

Для того чтобы созданное нами приложение могло отреагировать на выбор пользователем созданной нами команды меню, в нем необходимо создать приемник, осуществляющий эту реакцию, и связать его с объектом меню. Первоочередное создание приемника в нашем примере связано с тем, что после создания приложения мастер автоматически открывает вкладку классов, из которой должна производиться данная операция. Это несколько нарушает логику построения приложения, но, я надеюсь, не настолько, чтобы полностью ее затмить.

Поскольку после создания приемника мастер обеспечивает все условия для его редактирования, в данном примере сразу же получается работоспособный приемник. Для простоты он лишь выводит окно сообщения, используя для этого вызов функции `QMessageBox::information`. После этого в нем с использованием функции `QStatusBar::message`, в строке состояния восстанавливается выводимый в ней по умолчанию текст **Ready**. Для получения указателя на объект класса `QStatusBar` задействована функция `QMainWindow::statusBar`.

В функции `QtToolsApp::initActions` создается и инициализируется динамический объект класса `QAction`, а также производится связывание сигнала меню с созданным нами приемником. В первом аргументе конструктора класса `QAction` передается текстовое имя создаваемого объекта. Это имя будет выводиться в качестве подсказки, если текст подсказки не задан в явном виде функцией `QAction::setToolTip`. Во втором аргументе конструктора класса передается выводимый пользователю текст команды меню. Третий аргумент содержит комбинацию клавиш, используемую для вызова данной команды, а четвертый аргумент — владельца данной команды.

Для созданного объекта класса `QAction` вызывается функция `QAction::setStatusTip`, определяющая текст, выводимый в строке состояния при выделении данной команды, а также функция `QAction::setWhatsThis`, содержащая текст краткой справки по этой команде. После этого вызывается функция `QObject::connect`, связывающая сигнал `activated` созданного нами

объекта класса `QAction` с созданным нами в классе `QtToolsApp` приемником `slotUserCommand`.

В результате описанных выше действий был создан и инициализирован объект класса `QAction`, а его сигнал был связан с приемником класса приложения. Теперь осталось только поместить объект команды в соответствующее раскрывающееся меню. Эта операция производится в функции `QtToolsApp::initMenuBar`. Для этого создается объект класса `QPopupMenu` и указатель на этот объект передается в качестве аргумента функции `QAction::addTo`, добавляющей данную команду в меню. Последовательность вызова функций `addTo` определяет очередность команд в меню.

Теперь осталось только объединить заголовки раскрывающихся меню в панели меню. Это производится вызовом функции `QMenuData::insertItem`. Как и в случае с командами меню, последовательность вызовов этих функций определяет порядок заголовков в панели меню. Для получения указателя на объект панели меню используется вызов функции `QMainWindow::menuBar`.

Настройка панели инструментов

Как уже говорилось выше, панель инструментов является своеобразным дополнением к меню и тесно с ним связана. При нажатии пользователем на кнопки панели инструментов, как правило, вызываются команды, уже определенные для меню. Поэтому настройка панелей инструментов приложения обычно производится после настройки его системы меню.

По умолчанию приложение создает стандартную панель инструментов, состоящую из четырех кнопок, разделенных на две группы. Эта панель инструментов изображена на рис. 5.5.

Рис. 5.5. Стандартная панель инструментов приложения

В первую группу кнопок, осуществляющих операции по созданию и сохранению документа, входят кнопки **New File** (открытие нового документа), **Open File** (открытие существующего документа) и **Save File** (сохранение документа). Во вторую группу входит всего одна кнопка **Whats this?** (позволяющая получить краткую справку по элементам управления приложения).

Среда разработки KDevelop позволяет добавлять и уничтожать кнопки в стандартной панели инструментов, но в большинстве случаев для новых кнопок целесообразнее использовать новые панели инструментов, а из стандартной панели инструментов только удалять ненужные кнопки. Поскольку все эти операции осуществляются путем редактирования текста

приложения, для понимания всех процессов нам будет достаточно создать новую панель инструментов и поместить в нее кнопку, вызывающую созданную нами команду меню.

Чтобы внести соответствующие изменения в приложение **QtTools**:

1. Откройте приложение **QtTools** на редактирование.
2. Выберите команду меню **Tools | KIconEdit** (Сервис | Приложение KIconEdit). Появится окно приложения **KIconEdit**, изображенное на рис. 5.6.

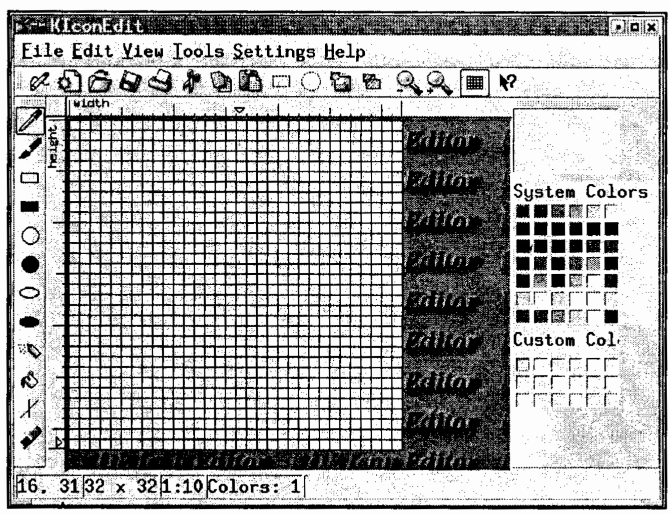


Рис. 5.6. Приложение **KIconEdit**

3. Выберите команду меню **File | New** (Файл | Создать), нажмите комбинацию клавиш <Ctrl>+<N> или кнопку **New** в панели инструментов. Появится первая панель мастера **Create New Icon** (Создание нового значка), изображенная на рис. 5.7.

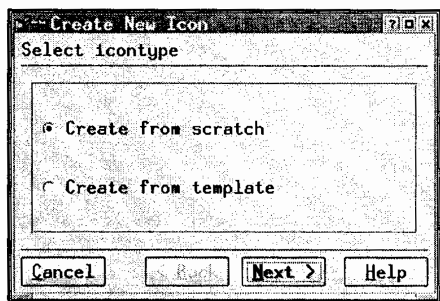


Рис. 5.7. Первая панель мастера **Create New Icon**

4. Оставьте установленным переключатель **Create from scratch** (Создать с чистого листа) и нажмите кнопку **Next**. Появится вторая панель мастера **Create New Icon**, изображенная на рис. 5.8.

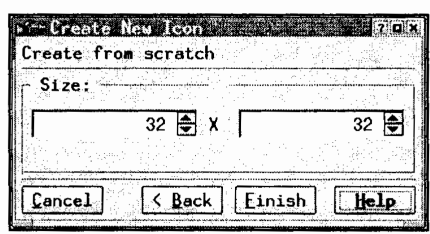


Рис. 5.8. Вторая панель мастера **Create New Icon**

5. В текстовых полях обоих инкрементных регуляторов установите значение 16 и нажмите кнопку **Finish**. В окне приложения **KIconEdit** появится пустая заготовка значка.
6. Создайте значок и выберите команду меню **File | Save** (Файл | Сохранить), нажмите комбинацию клавиш <Ctrl>+<S> или кнопку **Save** в панели инструментов. Появится диалоговое окно **Save As**, изображенное на рис. 5.9.
7. В окне списка раскройте каталог qttools, введите в текстовое поле **Location** (Путь) имя файла user.xpm и нажмите кнопку **Save**. Файл значка будет сохранен.
8. Закройте приложение **KIconEdit**.

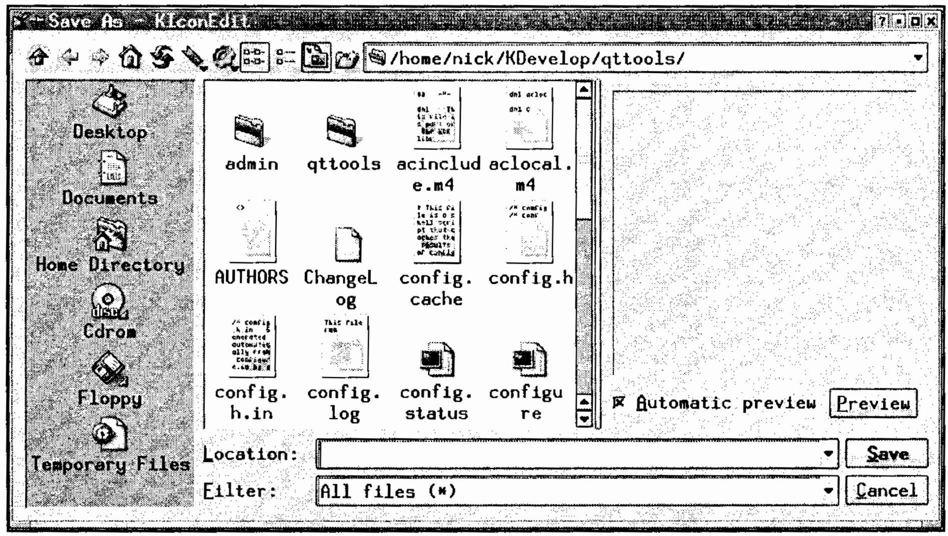


Рис. 5.9. Диалоговое окно **Save As**

Совет

Поскольку я не обладаю художественными способностями, я просто взял в каталоге **/usr/share/icons/locolor/16x16/apps** один из стандартных значков. Ничто не мешает и вам поступить так же. Однако операцию включения значка в каталог приложения лучше производить через приложение **KIconEdit**, поскольку в этом случае появляется возможность переименования файлов и содержащихся в них массивов.

9. В окне иерархических списков раскройте вкладку **Files**, раскройте в ней каталог **qttools** и щелкните левой кнопкой мыши на имени файла user.xpm. Появится диалоговое окно **Load decision**, изображенное на рис. 5.10, предлагающее загрузить файл в текстовом виде.

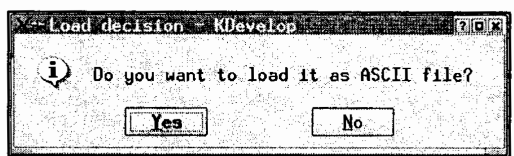


Рис. 5.10. Диалоговое окно **Load decision**

10. Нажмите кнопку **Yes**. Откроется окно редактирования файла user.xpm в текстовом виде.
11. В открывшемся окне замените строку `static char *user[]={` строкой

```
static const char *user[] = {
```

12. В окне иерархических списков раскройте вкладку **Classes**, щелкните правой кнопкой мыши по имени класса `QtToolsApp` и выберите в появившемся контекстном меню команду **Add member variable**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes**.
13. В текстовое поле **Type** введите тип создаваемой переменной `QToolBar`, в текстовое поле **Name** — имя переменной `*userToolbar`, в группе **Access** (Права доступа) установите переключатель **Private** (Закрытые), введите в текстовое поле **Documentation** текст `User toolbar` (Пользовательская панель инструментов) и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.
14. Повторите пункты 12 и 13 для добавления в класс переменной `*viewUserToolBar`, имеющей тип `QAction` и не имеющей комментария.
15. Во вкладке **Classes** щелкните правой кнопкой мыши по имени класса `QtToolsApp` и выберите в появившемся контекстном меню команду **Add slot**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Slots**.
16. В текстовое поле **Declaration** введите сигнатуру приемника `slotViewUserToolBar(bool)`, в группе **Access** установите переключатель

Private, введите в текстовое поле **Documentation** комментарий `Toggles user toolbar` (Делает пользовательскую панель инструментов видимой или невидимой) и нажмите кнопку **Apply**. В класс будет добавлен новый приемник.

17. В открывшемся после добавления нового приемника окне редактирования файла qttools.cpp измените этот приемник в соответствии с текстом листинга 5.2.

Листинг 5.2. Приемник `slotViewUserToolBar`

```
/** Делает пользовательскую панель инструментов видимой или невидимой */
void QtToolsApp::slotViewUserToolBar(bool toggle)
{
 statusBar()->message(tr("Toggle user toolbar..."));
 ///////////////////////////////////////////////////////////////////
 // Выводит пользовательскую панель инструментов на экран
 // и удаляет ее с экрана

 if (toggle == false)
 {
  userToolbar->hide();
 }
 else
 {
  userToolbar->show();
 }

 statusBar()->message(tr("Ready."));
}
```

18. В начале файла qttools.cpp после строки `#include "filesave.xpm"` вставьте строку

```
#include "user.xpm"
```

19. В конструкторе класса `QtToolsApp` после строки `viewToolBar->setOn(true);` вставьте строку

```
viewUserToolBar->setOn(true);
```

20. Измените функцию `QtToolsApp::initActions` в соответствии с текстом листинга 5.3.

Листинг 5.3. Функция `initActions`

```
/** Инициализирует все объекты класса QActions-приложения */
void QtToolsApp::initActions()
```

```
{
 QPixmap openIcon, saveIcon, newIcon, userIcon;
 newIcon = QPixmap(filenew);
 openIcon = QPixmap(fileopen);
 saveIcon = QPixmap(filesave);
 userIcon = QPixmap(user);

 fileNew = new QAction(tr("New File"), newIcon, tr("&New"),
               QAccel::stringToKey(tr("Ctrl+N")), this);
 fileNew->setStatusTip(tr("Creates a new document"));
 fileNew->setWhatsThis(tr("New File\n\nCreates a new document"));
 connect(fileNew, SIGNAL(activated()), this, SLOT(slotFileNew()));

 fileOpen = new QAction(tr("Open File"), openIcon, tr("&Open..."),
                                 0, this);
 fileOpen->setStatusTip(tr("Opens an existing document"));
 fileOpen->setWhatsThis(tr("Open File\n\nOpens an existing document"));
 connect(fileOpen, SIGNAL(activated()), this, SLOT(slotFileOpen()));

 fileSave = new QAction(tr("Save File"), saveIcon, tr("&Save"),
               QAccel::stringToKey(tr("Ctrl+S")), this);
 fileSave->setStatusTip(tr("Saves the actual document"));
 fileSave->setWhatsThis(tr("Save File.\n\nSaves the actual document"));
 connect(fileSave, SIGNAL(activated()), this, SLOT(slotFileSave()));

 fileSaveAs = new QAction(tr("Save File As"), tr("Save &as..."),
                                 0, this);
 fileSaveAs->setStatusTip(tr("Saves the actual document
                        under a new filename"));
 fileSaveAs->setWhatsThis(tr("Save As\n\nSaves the actual
                    document under a new filename"));
 connect(fileSaveAs, SIGNAL(activated()), this, SLOT(slotFileSave()));

 fileClose = new QAction(tr("Close File"), tr("&Close"),
               QAccel::stringToKey(tr("Ctrl+W")), this);
 fileClose->setStatusTip(tr("Closes the actual document"));
 fileClose->setWhatsThis(tr("Close File\n\n Closes the
                          actual document"));
 connect(fileClose, SIGNAL(activated()), this, SLOT(slotFileClose()));

 filePrint = new QAction(tr("Print File"), tr("&Print"),
               QAccel::stringToKey(tr("Ctrl+P")), this);
 filePrint->setStatusTip(tr("Prints out the actual document"));
 filePrint->setWhatsThis(tr("Print File\n\nPrints out the
                          actual document"));
 connect(filePrint, SIGNAL(activated()), this, SLOT(slotFilePrint()));
```

```
fileQuit = new aQAction(tr("Exit"), tr("E&xit"),
               QAccel::stringToKey(tr("Ctrl+Q")), this);
fileQuit->setStatusTip(tr("Quits the application"));
fileQuit->setWhatsThis(tr("Exit\n\nQuits the application"));
connect(fileQuit, SIGNAL(activated()), this, SLOT(slotFileQuit()));

editCut = new QAction(tr("Cut"), tr("Cu&t"),
               QAccel::stringToKey(tr("Ctrl+X")), this);
editCut->setStatusTip(tr("Cuts the selected section
                    and puts it to the clipboard"));
editCut->setWhatsThis(tr("Cut\n\nCuts the selected section
                    and puts it to the clipboard"));
connect(editCut, SIGNAL(activated()), this, SLOT(slotEditCut()));

editCopy = new QAction(tr("Copy"), tr("&Copy"),
               QAccel::stringToKey(tr("Ctrl+C")), this);
editCopy->setStatusTip(tr("Copies the selected section
                          to the clipboard"));
editCopy->setWhatsThis(tr("Copy\n\nCopies the selected section
                          to the clipboard"));
connect(editCopy, SIGNAL(activated()), this, SLOT(slotEditCopy()));

editPaste = new QAction(tr("Paste"), tr("&Paste"),
               QAccel::stringToKey(tr("Ctrl+V")), this);
editPaste->setStatusTip(tr("Pastes the clipboard contents
                         to actual position"));
editPaste->setWhatsThis(tr("Paste\n\nPastes the clipboard contents
                         to actual position"));
connect(editPaste, SIGNAL(activated()), this, SLOT(slotEditPaste()));

viewToolBar = new QAction(tr("Toolbar"), tr("Tool&bar"),
                           0, this, 0, true);
viewToolBar->setStatusTip(tr("Enables/disables the toolbar"));
viewToolBar->setWhatsThis(tr("Toolbar\n\nEnables/disables
                            the toolbar"));
connect(viewToolBar, SIGNAL(toggled(bool)),
                this, SLOT(slotViewToolBar(bool)));

viewUserToolBar = new QAction(tr("User toolbar"), tr("&User Toolbar"),
                           0, this, 0, true);
viewUserToolBar->setStatusTip(tr("Enables/disables user toolbar"));
viewUserToolBar->setWhatsThis(tr("User toolbar\n\nEnables/disables
                            user toolbar"));
connect(viewUserToolBar, SIGNAL(toggled(bool)),
              this, SLOT(slotViewUserToolBar(bool)));

viewStatusBar = new QAction(tr("Statusbar"), tr("&Statusbar"),
                           0, this, 0, true);
```

```
  viewStatusBar->setStatusTip(tr("Enables/disables the statusbar"));
  viewStatusBar->setWhatsThis(tr("Statusbar\n\nEnables/disables
                              the statusbar"));
  connect(viewStatusBar, SIGNAL(toggled(bool)),
                   this, SLOT(slotViewStatusBar(bool)));

  userCommand = new QAction(tr("User command"), tr("&User Command"),
                 QAccel::stringToKey(tr("Ctrl+U")), this);
  userCommand->setStatusTip(tr("Executes user command"));
  userCommand->setWhatsThis(tr("User\n\nExecutes user command"));
  connect(userCommand, SIGNAL(activated()), this, SLOT(slotUserCommand()));

  helpAboutApp = new QAction(tr("About"), tr("&About..."), 0, this);
  helpAboutApp->setStatusTip(tr("About the application"));
  helpAboutApp->setWhatsThis(tr("About\n\nAbout the application"));
  connect(helpAboutApp, SIGNAL(activated()), this, SLOT(slotHelpAbout()));
}
```

21. В функции `QtToolsApp::initMenuBar` после строки `viewToolBar->addTo(viewMenu);` вставьте строку

```
viewUserToolBar->addTo(viewMenu);
```

22. Измените функцию `QtToolsApp::initToolBar` в соответствии с текстом листинга 5.4.

Листинг 5.4. Функция initToolBar

```
void QtToolsApp::initToolBar()
{
 ///////////////////////////////////////////////////////////////////
 // ПАНЕЛЬ ИНСТРУМЕНТОВ
 fileToolbar = new QToolBar(this, "file operations");
 fileNew->addTo(fileToolbar);
 fileOpen->addTo(fileToolbar);
 fileSave->addTo(fileToolbar);
 fileToolbar->addSeparator();
 QWhatsThis::whatsThisButton(fileToolbar);

 userToolbar = new QToolBar(this, "user operations");
 userCommand->addTo(userToolbar);
}
```

23. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Появится окно приложения, изображенное на рис. 5.11.

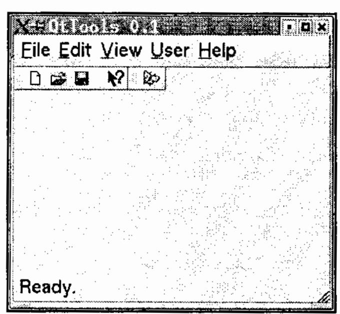


Рис. 5.11. Окно приложения **QtTools** с пользовательской панелью инструментов

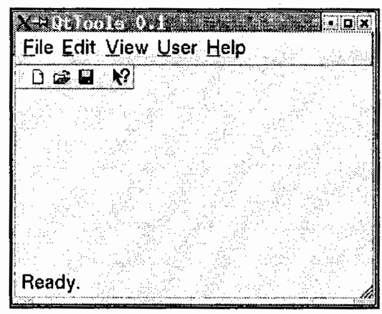


Рис. 5.12. Удаление пользовательской панели инструментов

24. Нажмите кнопку **User Command** (Пользовательская команда) в панели инструментов **User** (Пользовательская). Появится окно сообщения, свидетельствующее о выполнении пользовательской команды.
25. Закройте окно сообщения и нажмите комбинацию клавиш <Ctrl>+<U>. Появится окно сообщения, свидетельствующее о выполнении пользовательской команды.
26. Закройте окно сообщения и выберите команду меню **View | User Toolbar** (Вид | Пользовательская панель инструментов). Пользовательская панель инструментов исчезнет, как это показано на рис. 5.12.
27. Снова выберите команду меню **View | User Toolbar**. Появится пользовательская панель инструментов.
28. Закройте приложение.

Создание новой панели инструментов предполагает добавление в меню **View** команды скрытия и отображения этой панели. Это требование не является обязательным, но его невыполнение существенно снижает мнение пользователя создаваемого приложения о его разработчике. Поэтому внесенные в приложение изменения затрагивают не только панели инструментов, но и систему меню.

Прежде чем приступить к работе с панелью инструментов, необходимо создать значки для ее новых кнопок. Для создания и редактирования кнопок используется самостоятельное приложение **KIconEdit**. Полное описание работы с этим приложением не входит в задачу этой книги, а создание в нем нового значка настолько просто, что не нуждается в подробном описании.

Примечание

Справочная система среды разработки KDevelop рекомендует использовать для создания значка новой кнопки меню команду **File | New**, но при этом создается заготовка значка, полностью идентичная заготовке значка в приложении **KIconEdit**. Поэтому нет никакого смысла производить дополнительные операции.

Единственное, что нужно сделать разработчику после создания значка в приложении **KIconEdit**, — это объявить содержащийся в нем массив константным. Если этого не сделать, транслятор выдаст сообщение об ошибке.

Для непосредственной работы с панелью инструментов в классе приложения создается объект класса `QToolbar`, а для обработки команд визуализации этой панели используется объект класса `QAction` и соответствующий приемник, выводящий в строку состояния сообщение о своем вызове и устанавливающий видимость панели инструментов в зависимости от значения своего аргумента. Этот приемник полностью аналогичен приемнику, обрабатывающему команду визуализации стандартной панели инструментов.

Поскольку в функцию `QtToolsApp::initActions` было внесено много изменений, ее текст приведен полностью. Прежде всего, в эту функцию добавлена новая переменная типа `QPixmap`, используемая для работы с битовым образом значка кнопки. При инициализации данной переменной в аргументе конструктора класса передается имя данного значка, содержащееся в его файле, который необходимо добавить в файл реализации директивой `#include`.

Вторым изменением, внесенным в функцию `initActions`, является создание и инициализация объекта класса `QAction` для вывода на экран и удаления с него пользовательской панели инструментов, а также связывание его сигнала с соответствующим приемником. Поскольку связанная с данным объектом команда меню служит для переключения состояния, а не для инициализации какого-либо действия, с приемником связывается сигнал `QAction::toggled`, а не сигнал `activated`, использованный нами для команды меню.

Внимание!

Указание начального состояния команды меню в конструкторе объекта класса `QAction` влияет на выполнение данной команды, а не на ее отображение в меню. Поэтому для того, чтобы отображение команды меню соответствовало ее истинному состоянию на момент запуска приложения, в конструкторе класса `QtToolsApp` вызывается виртуальный приемник `QAction::setOn`.

Созданная нами пользовательская команда может теперь вызываться не только командой меню, но и нажатием кнопки в панели инструментов, поэтому нам необходимо внести изменения в конструктор класса, используемый для создания ее объекта. Поскольку применявшийся до этого конструктор объекта не позволял работать с панелью инструментов, нами использован другой конструктор, вторым аргументом которого является ссылка на объект класса `QIconSet`. Кроме того, чтобы не выделять это в отдельный раздел, здесь же указана комбинация клавиш, используемая для вызова этой команды. Таким образом, для данной команды реализуются все возможные способы вызова.

6 Зак. 113

Изменения, внесенные в функцию QtToolsApp::initMenuBar, минимальны: в меню **View** добавляется команда визуализации пользовательской панели инструментов.

В функцию QtToolsApp::initToolBar внесены более серьезные изменения. Прежде всего, в ней создается объект класса QToolBar и в него, с помощью функции QAction::addTo, добавляется новая кнопка. На этом все изменения, связанные с добавлением в приложение новой панели инструментов, завершаются.

Работа со строкой состояния

Строка состояния представляет собой горизонтальную строку, располагающуюся в нижней части главного окна приложения, предназначенную для вывода различной вспомогательной информации. Для работы со строкой состояния приложение использует скрытый объект класса QStatusBar, доступ к которому осуществляется вызовом функции QMainWindow::statusBar.

Мы уже использовали строку состояния для вывода различных сообщений о работе команд меню. Для вывода такого сообщения достаточно воспользоваться функцией QStatusBar::message, указав в ее аргументе текст выводимого сообщения. Чтобы удалить выведенное таким образом сообщение из строки состояния, используется функция QStatusBar::clear. Однако эта функция применяется только для очистки строки состояния, поскольку при выводе каждого нового сообщения предыдущее сообщение автоматически удаляется из строки состояния.

В приложениях Windows в строке состояния автоматически формируются панели, в которые по умолчанию выводится информация о состоянии некоторых функциональных клавиш. Хотя в большинстве случаев эта информация бесполезна, само выделение в строке состояния панелей для вывода специфической информации является плодотворной идеей, интенсивно используемой в различных приложениях. Попробуем добиться того же результата в приложениях Linux.

Чтобы внести соответствующие изменения в приложение **QtTools**:

1. Откройте приложение **QtTools** на редактирование.
2. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса QtToolsApp и выберите в появившемся контекстном меню команду **Add member variable**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes**.
3. В текстовое поле **Type** введите тип переменной QLabel, в текстовое поле **Name** — идентификатор переменной *xPane, в панели **Access** установите переключатель **Private**, в текстовое поле **Documentation** введите комментарий X pane of Status Bar (Панель строки состояния для отображения го-

ризонтальной координаты) и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.

4. Повторите пункты 2 и 3 для добавления в класс переменной `*yPane`, снабдив ее комментарием `Y pane of Status Bar` (Панель строки состояния для отображения вертикальной координаты).
5. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `QtToolsApp` и выберите в появившемся контекстном меню команду **Add slot**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Slots**.
6. В текстовое поле **Declaration** введите сигнатуру функции `onUpdateStatusBar(int, int)`, в текстовое поле **Documentation** — комментарий `Updates Status Bar panes` (Вносит изменения в панели строки состояния) и нажмите кнопку **Apply**. В класс будет добавлен новый приемник, откроется окно редактирования файла qttools.cpp и текстовый курсор будет помещен в заготовку нового приемника.
7. Измените приемник `QtToolsApp::onUpdateStatusBar` в соответствии с текстом листинга 5.5.

Листинг 5.5. Приемник `onUpdateStatusBar`

```
/** Вносит изменения в панели строки состояния */
void QtToolsApp::onUpdateStatusBar(int x, int y)
{
 xPane->setText( tr("X = %1").arg( x, -8));
 yPane->setText( tr("Y = %1").arg( y, -8));
}
```

8. В окне редактирования файла qttools.cpp измените функцию `QtToolsApp::initStatusBar` в соответствии с текстом листинга 5.6.

Листинг 5.6. Функция `initStatusBar`

```
void QtToolsApp::initStatusBar()
{
 ///////////////////////////////////////////////////////////////////////
 // СТРОКА СОСТОЯНИЯ

 xPane = new QLabel( statusBar());
 yPane = new QLabel( statusBar());

 xPane->setText( tr("X =    "));
 yPane->setText( tr("Y =    "));
```

```
 statusBar()->message(tr("Ready."), 2000);
 statusBar()->addWidget( yPane, 0, TRUE);
 statusBar()->addWidget( xPane, 0, TRUE);
}
```

9. Измените конструктор и деструктор класса `QtToolsApp` в соответствии с текстом листинга 5.7.

Листинг 5.7. Конструктор и деструктор класса `QtToolsApp`

```
QtToolsApp::QtToolsApp()
{
 xPane = NULL;
 yPane = NULL;

 setCaption(tr("QtTools " VERSION));

 ///////////////////////////////////////////////////////////////////////
 // Вызов функций инициализации компонентов приложения
 initActions();
 initMenuBar();
 initToolBar();
 initStatusBar();

 initDoc();
 initView();

 viewToolBar->setOn(true);
 userToolBar->setOn(true);
 viewStatusBar->setOn(true);
}

QtToolsApp::~QtToolsApp()
{
 if( xPane)
  delete xPane;

 if( yPane)
  delete yPane;
}
```

10. Измените функцию `QtToolsApp::initView` в соответствии с текстом листинга 5.8.

Листинг 5.8. Функция `initView`

```
void QtToolsApp::initView()
{
```

```
////////////////////////////////////////////////////////////////////
// Произведите настройку главного окна приложения
view=new QtToolsView(this, doc);
setCentralWidget(view);

view->setMouseTracking( true);

connect(view, SIGNAL(updateStatusBar(int, int)),
              this, SLOT(onUpdateStatusBar(int, int)));
}
```

11. Откройте окно редактирования файла qttools.h и после строки `#include <qpainter.h>` поместите строку

```
#include <qlabel.h>
```

12. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `QtToolsView` и выберите в появившемся контекстном меню команду **Add signal**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Signals**, как показано на рис. 5.13.

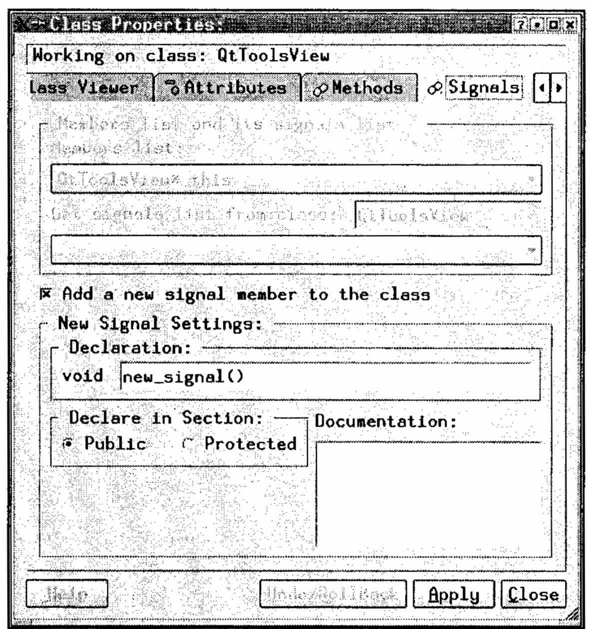


Рис. 5.13. Диалоговое окно **Class Properties**, вкладка **Signals**

13. В текстовое поле **Declaration** введите сигнатуру сигнала `updateStatusBar(int, int)`, в текстовое поле **Documentation** — комментарий `Updates Status Bar panes` (Вносит изменения в панели строки

состояния) и нажмите кнопку **Apply**. В класс будет добавлен новый сигнал.

14. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `QtToolsView` и выберите в появившемся контекстном меню команду **Add member function**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Methods**, как показано на рис. 5.14.

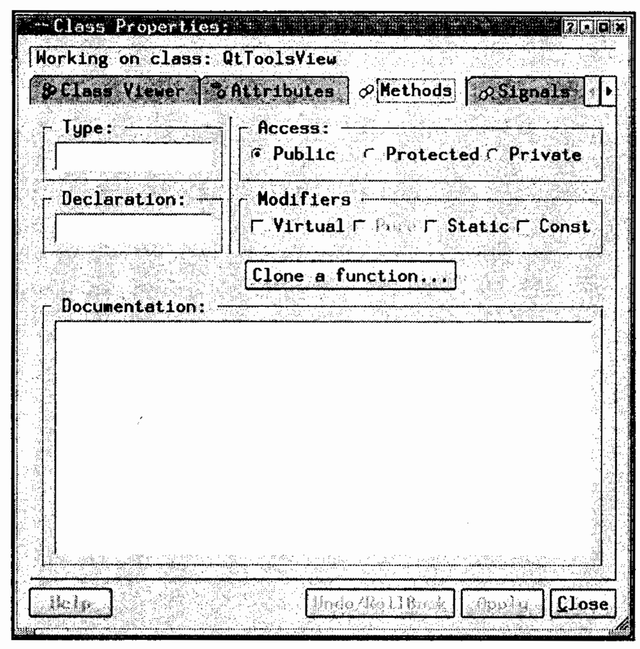


Рис. 5.14. Диалоговое окно **Class Properties**, вкладка **Methods**

15. В текстовое поле **Type** введите тип возвращаемого значения `void`, в текстовое поле **Declaration** — сигнатуру функции `mouseMoveEvent(QMouseEvent*)`, в группе **Access** установите переключатель **Protected**, в группе **Modifiers** — флажок **Virtual**, в текстовое поле **Documentation** введите комментарий `Handles the Event_MouseMove event` (Обрабатывает перемещение мыши) и нажмите кнопку **Apply**. В класс будет добавлен новый приемник, откроется окно редактирования файла qttoolsview.cpp и текстовый курсор будет помещен в заготовку новой функции.

16. В открывшемся после добавления приемника окне редактирования файла qttoolsview.cpp измените этот приемник в соответствии с текстом листинга 5.9.

Листинг 5.9. Приемник mouseMoveEvent

```
/** Обрабатывает перемещение мыши */
void QtToolsView::mouseMoveEvent(QMouseEvent* e)
{
 updateStatusBar(e->x(), e->y());
}
```

17. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Появится окно приложения.
18. Поместите указатель мыши в рабочую область окна. В строке состояния появятся его координаты, как это показано на рис. 5.15.
19. Закройте приложение.

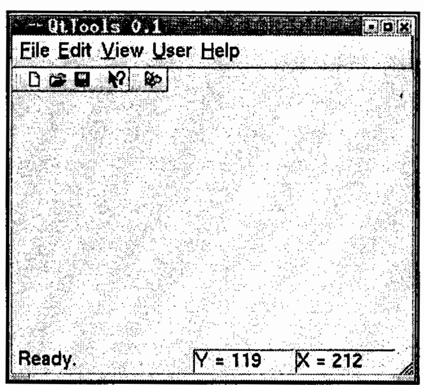


Рис. 5.15. Координаты указателя мыши в строке состояния

Для формирования панелей строки состояния нами использованы объекты класса `QLabel`, позволяющие выводить на экран различный текст. Поскольку эти объекты должны быть инициализированы конструктором класса, а сам класс не имеет перегруженного оператора присваивания, эти объекты сделаны динамическими: они создаются в конструкторе класса `QtToolsApp` и уничтожаются в его деструкторе.

Для удобства чтения файла qttools.cpp создание и инициализация панелей строки состояния производятся в функции `QtToolsApp::initStatusBar`, вызываемой в конструкторе класса. Поскольку объекты панелей принадлежат строке состояния, то указатель на ее объект передается в качестве аргумента конструкторам класса `QLabel` как указатель на объект родительского окна. Для получения этого указателя используется функция `QMainWindow::statusBar`.

После создания объектов панелей строки состояния для каждого из них вызывается функция `QLabel::setText`, помещающая в них текстовые строки.

И, наконец, полностью подготовленные объекты панелей включаются в строку состояния вызовом функции `QStatusBar::addWidget`. Первый аргумент данной функции содержит указатель на объект включаемого в строку состояния элемента управления, второй — определяет, будет ли этот элемент управления иметь фиксированные размеры или же его размеры будут определяться его содержимым, и третий аргумент определяет положение включаемого элемента управления в строке состояния. Все вышеперечисленные действия позволили нам создать в строке состояния панели со статическим текстом. Теперь перейдем к тому, как этот текст может изменяться в процессе работы приложения.

Рассматриваемое приложение выводит в панелях строки состояния координаты указателя мыши в рабочей области окна. Однако за работу в этой области отвечает объект класса представления, которому и направляются все сообщения о произошедших в этой области событиях. Поэтому для обработки сообщений о перемещении мыши нужно перегрузить функцию обработки события класса представления. Но за работу со строкой состояния отвечает класс приложения, поэтому полученную в объекте класса представления информацию необходимо передать в объект класса приложения. Для этого в объект класса представления включается сигнал, передающий два целочисленных аргумента, а в объект класса приложения — приемник для этих двух аргументов. Связывание сигнала и приемника производится в функции `QtToolsApp::initView`, поскольку именно в ней создается объект класса представления, сигнал которого тут же связывается с приемником класса приложения.

Функция обработки перемещения мыши класса представления имеет простейший вид — в ней вызывается сигнал, которому передаются текущие координаты указателя мыши, извлеченные из объекта класса события функциями `QMouseEvent::x` и `QMouseEvent::y`. Эти координаты передаются приемнику `QtToolsApp::onUpdateStatusBar`, обновляющему содержимое панелей строки состояния. При этом для каждой из панелей вызывается функция `QLabel::setText`, в качестве аргумента которой передается новая строка. Эта строка формируется функцией `QObject::tr` на основе заданного шаблона и текущего значения соответствующей координаты, преобразованного в текстовый вид функцией `QString::arg`.

Пользовательский интерфейс приложений KDE

Среда разработки KDevelop позволяет создавать не только приложения, использующие исключительно библиотеку Qt, но и приложения KDE, использующие собственную библиотеку среды разработки, являющуюся дальнейшим развитием библиотеки Qt.

Для демонстрации принципов работы с элементами пользовательского интерфейса KDE будет использовано демонстрационное приложение **KDETools**, текст которого можно найти на прилагаемом к книге CD.

Чтобы самостоятельно создать это приложение:

1. Выберите команду меню **Project | New**. Появится диалоговое окно **ApplictionWizard**.
2. В иерархическом списке типов создаваемых приложений этого окна оставьте выделенным узел **KDE Normal** (Однооконное приложение KDE) и нажмите кнопку **Next**. Раскроется следующая страница мастера.
3. В текстовое поле **Project name** введите имя проекта **KDETools** и нажмите кнопку **Create**.
4. После завершения работы мастера по созданию приложения нажмите кнопку **Exit**. Заготовка приложения будет создана.

Внесение изменений в меню

Процедура внесения изменений в меню в приложениях KDE может существенно отличаться от аналогичной процедуры, используемой в приложениях Qt. Все зависит от выбранного вами пути. Дело в том, что заготовки приложений KDE, создаваемые соответствующими мастерами, используют для описания своего пользовательского интерфейса язык XML. С другой стороны, в библиотеке KDevelop предусмотрены средства для создания команд меню и кнопок панели инструментов по методикам, использовавшимся в библиотеке Qt.

И в том и в другом случае команда меню реализуется в объекте класса `KAction`, но при использовании стандартной методики в качестве родительского объекта указывается объект главного окна приложения, а при использовании описания пользовательского интерфейса на языке XML в качестве родительского объекта указывается объект класса `KActionCollection`, в который помещаются все команды. Указатель на объект класса коллекции команд приложения возвращается функцией `KXMLGUIClient::actionCollection`.

Другим отличием заготовки приложения KDE от заготовки приложения Qt является использование статических методов класса `KStdAction` для описания стандартных команд меню. Применение этих методов существенно сокращает объем информации, которую необходимо указать при инициализации создаваемых объектов. Они даже исключают необходимость включения описания этих команд в описание пользовательского интерфейса на языке XML.

Поскольку оба метода включения команд меню в приложение считаются равноправными, в данном разделе будут созданы два пользовательских меню, для создания каждого из которых будет применена своя методика.

Чтобы добавить в приложение **KDETools** новые раскрывающиеся меню:

1. Откройте приложение **KDETools** на редактирование.
2. В раскрытой в окне иерархических списков вкладке **Classes** щелкните правой кнопкой мыши по имени класса `KDEToolsApp`, расположенного в папке **Classes**, и выберите в появившемся контекстном меню команду **Add member variable**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes**.
3. В текстовое поле **Type** введите тип переменной `KAction*`, в текстовое поле **Name** — идентификатор переменной `userFirst`, в группе **Access** установите переключатель **Private** и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.
4. Повторите пункты 2 и 3 для включения в класс переменной `userSecond`.
5. Повторите пункты 2 и 3 для включения в класс переменной `secondMenu`, имеющей тип `QPopupMenu*`.
6. Во вкладке **Classes** окна иерархических списков щелкните правой копкой мыши по имени класса `KDEToolsApp` и выберите в раскрывшемся контекстном меню команду **Add Slot**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Slots**.
7. В текстовом поле **Declaration** замените текст `new_slot()` текстом `slotFirstUserCommand()`, введите в текстовое поле **Documentation** комментарий `Executes first user command` (Выполняет первую пользовательскую команду) и нажмите кнопку **Apply**. В класс будет добавлен новый приемник, откроется окно редактирования файла **kdetools.cpp** и текстовый курсор будет помещен в заготовку приемника.
8. Повторите пункты 6 и 7 для включения в класс приемника `slotSecondUserCommand()`, снабдив его комментарием `Executes second user command` (Выполняет вторую пользовательскую команду).
9. Измените новые приемники в соответствии с текстом листинга 5.10.

Листинг 5.10. Приемники класса `KDEToolsApp`

```
/** Выполняет первую пользовательскую команду */
void KDEToolsApp::slotFirstUserCommand()
{
 KMessageBox::information(this, i18n("Executing first user command..."),
                            i18n("User"));

 slotStatusMsg(i18n("Ready."));
}
```

```
/** Выполняет вторую пользовательскую команду */
void KDEToolsApp::slotSecondUserCommand()
{
 KMessageBox::information(this,
        i18n("Executing second user command..."), i18n("User"));

 slotStatusMsg(i18n("Ready."));
}
```

10. В том же файле после строки `#include <kstdaction.h>` вставьте строку

```
#include <kmessagebox.h>
```

11. Измените функцию `KDEToolsApp::initActions` в соответствии с текстом листинга 5.11.

Листинг 5.11. Функция `initActions`

```
void KDEToolsApp::initActions()
{
 fileNewWindow = new KAction(i18n("New &Window"), 0, 0, this,
    SLOT(slotFileNewWindow()), actionCollection(),"file_new_window");
 fileNew = KStdAction::openNew(this,
               SLOT(slotFileNew()), actionCollection());
 fileOpen = KStdAction::open(this,
               SLOT(slotFileOpen()), actionCollection());
 fileOpenRecent = KStdAction::openRecent(this,
       SLOT(slotFileOpenRecent(const KURL&)), actionCollection());
 fileSave = KStdAction::save(this,
               SLOT(slotFileSave()), actionCollection());
 fileSaveAs = KStdAction::saveAs(this,
              SLOT(slotFileSaveAs()), actionCollection());
 fileClose = KStdAction::close(this,
              SLOT(slotFileClose()), actionCollection());
 filePrint = KStdAction::print(this,
              SLOT(slotFilePrint()), actionCollection());
 fileQuit = KStdAction::quit(this,
               SLOT(slotFileQuit()), actionCollection());
 editCut = KStdAction::cut(this,
               SLOT(slotEditCut()), actionCollection());
 editCopy = KStdAction::copy(this,
               SLOT(slotEditCopy()), actionCollection());
 editPaste = KStdAction::paste(this,
              SLOT(slotEditPaste()), actionCollection());
 userFirst = new KAction(i18n("&First user"), 0, 0,
  this, SLOT(slotFirstUserCommand()), actionCollection(), "first_user");
```

```
    userSecond = new KAction(i18n("&Second user"), 0, 0,
           this, SLOT(slotSecondUserCommand()), this, "second_user");
    viewToolBar = KStdAction::showToolbar(this,
                 SLOT(slotViewToolBar()), actionCollection());
    viewStatusBar = KStdAction::showStatusbar(this,
                 SLOT(slotViewStatusBar()), actionCollection());

    fileNewWindow->setStatusText(i18n("Opens a new application window"));
    fileNew->setStatusText(i18n("Creates a new document"));
    fileOpen->setStatusText(i18n("Opens an existing document"));
    fileOpenRecent->setStatusText(i18n("Opens a recently used file"));
    fileSave->setStatusText(i18n("Saves the actual document"));
    fileSaveAs->setStatusText(i18n("Saves the actual document as..."));
    fileClose->setStatusText(i18n("Closes the actual document"));
    filePrint ->setStatusText(i18n("Prints out the actual document"));
    fileQuit->setStatusText(i18n("Quits the application"));
    editCut->setStatusText(i18n("Cuts the selected section
                    and puts it to the clipboard"));
    editCopy->setStatusText(i18n("Copies the selected section
                          to the clipboard"));
    editPaste->setStatusText(i18n("Pastes the clipboard contents
                         to actual position"));
    userFirst->setStatusText(i18n("Executes first user command"));
    userSecond->setStatusText(i18n("Executes second user command"));
    viewToolBar->setStatusText(i18n("Enables/disables the toolbar"));
    viewStatusBar->setStatusText(i18n("Enables/disables the statusbar"));

    // Для проверки укажите абсолютный путь к файлу kdetoolsui.rc
    // при вызове функции createGUI();
    createGUI();

    QPopupMenu *pSecondMenu = new QPopupMenu;

    userSecond-> plug( pSecondMenu);

    menuBar()->insertItem(i18n("&Second"), pSecondMenu);
}
```

12. В окне иерархических списков раскройте вкладку **Files**, раскройте в ней каталог kdetools и щелкните левой кнопкой мыши по имени файла kdetoolsui.rc. Откроется окно редактирования этого файла.
13. Измените файл kdetoolsui.rc в соответствии с текстом листинга 5.12.

Листинг 5.12. Файл kdetoolsui.rc

```
<!DOCTYPE kpartgui>
<kpartgui name="kdetools" version="0.1">
```

```
<MenuBar>
 <Menu name="file"><text>&amp;File</text>
  <Action name="file_new_window"/>
 </Menu>
 <Menu name="first"><text>&amp;First</text>
  <Action name="first_user"/>
 </Menu>
</MenuBar>
</kpartgui>
```

14. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Появится окно приложения, изображенное на рис. 5.16.

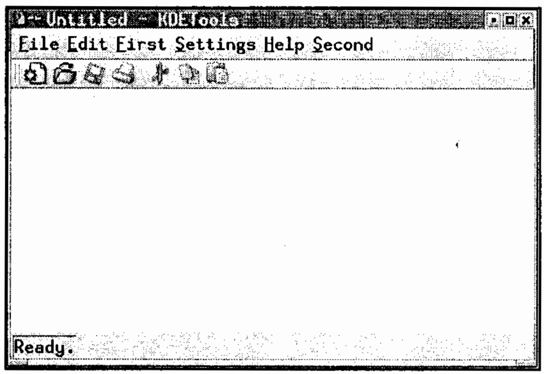


Рис. 5.16. Окно приложения KDETools

15. Выберите команду меню **First | First user** (Первое | Первая пользовательская команда). Появится окно сообщения, изображенное на рис. 5.17.

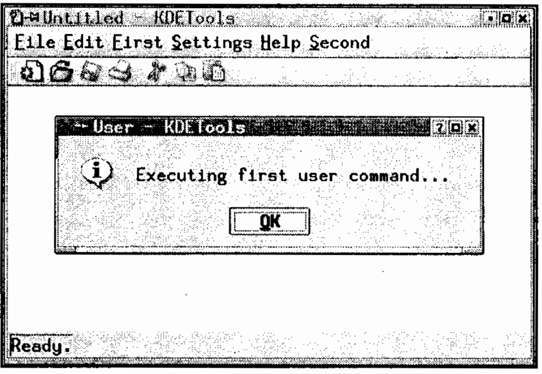


Рис. 5.17. Окно сообщения в приложении KDETools

16. Закройте окно сообщения и выберите команду меню **Second | Second user** (Второе | Вторая пользовательская команда). Появится аналогичное окно сообщения о выполнении второй пользовательской команды.
17. Закройте приложение.

Методика включения в приложение команды **Second user** и содержащего ее меню **Second** во многом аналогична уже рассмотренной методике, используемой в приложениях Qt. Основным отличием является отсутствие в классе приложения функции `initMenuBar`, в которой производилась инициализация объектов меню приложения. Поэтому эти операции были произведены в функции `KDEToolsApp::initActions`.

Для работы с раскрывающимися меню в приложениях KDE используется класс `QPopupMenu`, используемый для тех же целей приложениями Qt. Учитывая практически полную замену в приложениях KDE классов библиотеки Qt на производные от них классы библиотеки KDE, можно сделать вывод о том, что использование раскрывающихся меню не является магистральным направлением развития данного типа приложений. Однако для включения команды в меню используется уже новая виртуальная функция `KAction::plug`, поскольку класс `KAction` является прямым потомком класса `QObject` и ничего не наследует от класса `QAction`.

Набор аргументов конструктора класса `KAction` существенно отличается от конструктора класса `QAction`, поскольку позволяет уже в процессе создания объекта команды связать его с приемником. В функции `initActions`, создаваемой в приложениях KDE, сначала создаются все объекты команд, а затем для каждого из них вызывается виртуальный приемник `KAction::setStatusText`, устанавливающий подсказку, которая должна выводиться в строку состояния при выделении этой команды. Это отличает их от одноименных функций, создаваемых в приложениях Qt, где все операторы, связанные с инициализацией одной команды, были собраны в одном месте.

Последним оператором в заготовке функции `KDEToolsApp::initActions` является вызов функции `KMainWindow::createGUI`, создающей пользовательский интерфейс приложения, описанный в файле XML. Если в аргументе функции не указан путь к файлу, она производит локальный поиск файла с расширением rc и именем, образованным путем конкатенации имени приложения и суффикса ui (в нашем случае это файл kdetoolsui.rc).

Настройка панели инструментов

Процедура включения панели инструментов в приложение и кнопки в панель инструментов во многом напоминает процедуру создания нового меню и включения в него команд. Стандартная методика выполнения этих операций для меню уже была нами подробно рассмотрена и читателю не составит

труда использовать ее для работы с панелями инструментов. Следует только помнить, что в приложениях KDE кнопка добавляется в панель инструментов функцией `KAction::plug`. Поэтому в данном разделе будет рассмотрено только создание новой панели инструментов с использованием файла XML.

Чтобы внести соответствующие изменения в приложение **KDETools**:

1. Откройте приложение **KDETools** на редактирование.
2. В окне иерархических списков раскройте вкладку **Classes**, щелкните правой кнопкой мыши по имени класса `KDEToolsApp` и выберите в появившемся контекстном меню команду **Add member variable**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes**.
3. В текстовое поле **Type** введите тип создаваемой переменной `KToggleAction*`, в текстовое поле **Name** — имя переменной `viewUserToolBar`, в группе **Access** установите переключатель **Private** и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.
4. Во вкладке **Classes** щелкните правой кнопкой мыши по имени класса `KDEToolsApp` и выберите в появившемся контекстном меню команду **Add slot**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Slots**.
5. В текстовое поле **Declaration** введите имя приемника `slotViewUserToolBar()`, в текстовое поле **Documentation** — текст `Toggles user toolbar` (Выводит на экран и убирает с него пользовательскую панель инструментов) и нажмите кнопку **Apply**. В класс будет добавлен новый приемник, откроется окно редактирования файла kdetools.cpp, а текстовый курсор будет помещен в заготовку приемника.
6. Измените приемник `KDEToolsApp::slotViewUserToolBar` в соответствии с текстом листинга 5.13.

Листинг 5.13. Приемник `slotViewUserToolBar`

```
/** Выводит на экран и убирает с него пользовательскую панель
инструментов */
void KDEToolsApp::slotViewUserToolBar()
{
 slotStatusMsg(i18n("Toggling user toolbar..."));
 ///////////////////////////////////////////////////////////////////
 // Выводит на экран и убирает с него панель инструментов
 if(!viewUserToolBar->isChecked())
 {
  toolBar("userToolBar")->hide();
 }
```

```
 else
 {
  toolBar("userToolBar")->show();
  }

 slotStatusMsg(i18n("Ready."));
}
```

7. Измените функцию `KDEToolsApp::initActions` в соответствии с текстом листинга 5.14.

Листинг 5.14. Функция `initActions`

```
void KDEToolsApp::initActions()
{
 fileNewWindow = new KAction(i18n("New &Window"), 0, 0,
 this, SLOT(slotFileNewWindow()), actionCollection(),"file_new_window");
 fileNew = KStdAction::openNew(this,
               SLOT(slotFileNew()), actionCollection());
 fileOpen = KStdAction::open(this,
               SLOT(slotFileOpen()), actionCollection());
 fileOpenRecent = KStdAction::openRecent(this,
      SLOT(slotFileOpenRecent(const KURL&)), actionCollection());
 fileSave = KStdAction::save(this,
               SLOT(slotFileSave()), actionCollection());
 fileSaveAs = KStdAction::saveAs(this,
              SLOT(slotFileSaveAs()), actionCollection());
 fileClose = KStdAction::close(this,
              SLOT(slotFileClose()), actionCollection());
 filePrint = KStdAction::print(this,
              SLOT(slotFilePrint()), actionCollection());
 fileQuit = KStdAction::quit(this,
               SLOT(slotFileQuit()), actionCollection());
 editCut = KStdAction::cut(this,
               SLOT(slotEditCut()), actionCollection());
 editCopy = KStdAction::copy(this,
               SLOT(slotEditCopy()), actionCollection());
 editPaste = KStdAction::paste(this,
              SLOT(slotEditPaste()), actionCollection());
 userFirst = new KAction(i18n("&First user"), "up", 0,
  this, SLOT(slotFirstUserCommand()), actionCollection(), "first_user");
 userSecond = new KAction(i18n("&Second user"), 0, 0,
       this, SLOT(slotSecondUserCommand()), this, "second_user");
 viewToolBar = KStdAction::showToolbar(this,
             SLOT(slotViewToolBar()), actionCollection());
```

```
    viewUserToolBar = new KToggleAction(i18n("Show &User Toolbar"), 0,
           this, SLOT(slotViewUserToolBar()), actionCollection(),
                          "options_show_user_toolbar");
    viewStatusBar = KStdAction::showStatusbar(this,
                SLOT(slotViewStatusBar()), actionCollection());

    fileNewWindow->setStatusText(i18n("Opens a new application window"));
    fileNew->setStatusText(i18n("Creates a new document"));
    fileOpen->setStatusText(i18n("Opens an existing document"));
    fileOpenRecent->setStatusText(i18n("Opens a recently used file"));
    fileSave->setStatusText(i18n("Saves the actual document"));
    fileSaveAs->setStatusText(i18n("Saves the actual document as..."));
    fileClose->setStatusText(i18n("Closes the actual document"));
    filePrint ->setStatusText(i18n("Prints out the actual document"));
    fileQuit->setStatusText(i18n("Quits the application"));
    editCut->setStatusText(i18n("Cuts the selected section
                        and puts it to the clipboard"));
    editCopy->setStatusText(i18n("Copies the selected section
                              to the clipboard"));
    editPaste->setStatusText(i18n("Pastes the clipboard contents
                             to actual position"));
    userFirst->setStatusText(i18n("Executes first user command"));
    userSecond->setStatusText(i18n("Executes second user command"));
    viewToolBar->setStatusText(i18n("Enables/disables the toolbar"));
    viewUserToolBar->setStatusText(i18n("Enables/disables user toolbar"));
    viewStatusBar->setStatusText(i18n("Enables/disables the statusbar"));

    // Для проверки укажите абсолютный путь к файлу kdetoolsui.rc
    // при вызове функции createGUI();
    createGUI();

    QPopupMenu *pSecondMenu = new QPopupMenu;

    userSecond-> plug( pSecondMenu);

    menuBar()->insertItem(i18n("&Second"), pSecondMenu);

    toolBar("userToolBar")->setFullSize(false);
  }
```

8. Измените функции `KDEToolsApp::saveOptions` и `KDEToolsApp::readOptions` в соответствии с текстом листинга 5.15.

Листинг 5.15. Функции `saveOptions` и `readOptions`

```
/** Сохранение параметров приложения */
void KDEToolsApp::saveOptions()
```

```
{
 config->setGroup("General Options");
 config->writeEntry("Geometry", size());
 config->writeEntry("Show Toolbar", viewToolBar->isChecked());
 config->writeEntry("Show User Toolbar", viewUserToolBar-
>isChecked());
 config->writeEntry("Show Statusbar",viewStatusBar->isChecked());
 config->writeEntry("ToolBarPos", (int) toolBar("mainToolBar")->barPos());
 config->writeEntry("UserToolBarPos",
                    (int) toolBar("userToolBar")->barPos());
 fileOpenRecent->saveEntries(config,"Recent Files");
}

/** Чтение параметров приложения из файла конфигурации */
void KDEToolsApp::readOptions()
{
 config->setGroup("General Options");

 // Настройка состояния панелей
 bool bViewToolbar = config->readBoolEntry("Show Toolbar", true);
 viewToolBar->setChecked(bViewToolbar);
 slotViewToolBar();

 bool bViewUserToolbar = config->readBoolEntry("Show User Toolbar",
                                  true);
 viewUserToolBar->setChecked(bViewUserToolbar);
 slotViewUserToolBar();

 bool bViewStatusbar = config->readBoolEntry("Show Statusbar", true);
 viewStatusBar->setChecked(bViewStatusbar);
 slotViewStatusBar();

 // Настройка положения панелей
 KToolBar::BarPosition toolBarPos;
 toolBarPos=(KToolBar::BarPosition)
           config->readNumEntry("ToolBarPos", KToolBar::Top);
 toolBar("mainToolBar")->setBarPos(toolBarPos);

 toolBarPos=(KToolBar::BarPosition)
         config->readNumEntry("UserToolBarPos", KToolBar::Top);
 toolBar("userToolBar")->setBarPos(toolBarPos);

 // Инициализация списка последних открывавшихся файлов
 fileOpenRecent->loadEntries(config,"Recent Files");

 QSize size=config->readSizeEntry("Geometry");
 if(!size.isEmpty())
```

```
  {
   resize(size);
  }
 }
```

9. Откройте окно редактирования файла kdetoolsui.rc и измените его в соответствии с текстом листинга 5.16.

Листинг 5.16. Файл kdetoolsui.rc

```
<!DOCTYPE kpartgui>
<kpartgui name="kdetools" version="0.1">
<MenuBar>
 <Menu name="file"><text>&amp;File</text>
  <Action name="file_new_window"/>
 </Menu>
 <Menu name="first"><text>&amp;First</text>
  <Action name="first_user"/>
 </Menu>
 <Menu name="settings"><text>&amp;Settings</text>
  <Action name="options_show_user_toolbar"/>
 </Menu>
</MenuBar>
<ToolBar name="userToolBar"><text>User Toolbar</text>
 <Action name="first_user"/>
</ToolBar>
</kpartgui>
```

10. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Появится окно приложения, изображенное на рис. 5.18.

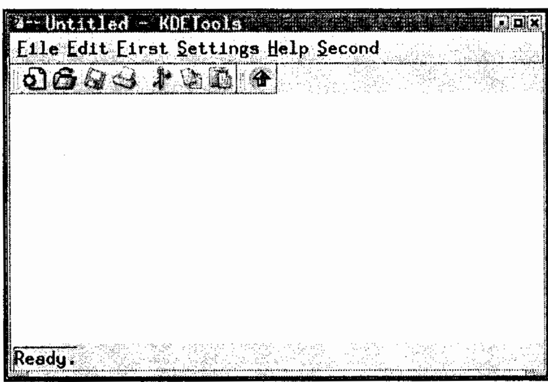


Рис. 5.18. Окно приложения **KDETools** с пользовательской панелью инструментов

11. Нажмите кнопку **Executes first user command** (Выполняет первую пользовательскую команду) в пользовательской панели инструментов. Появится окно сообщения о выполнении команды.
12. Закройте окно сообщения и выберите команду меню **Settings | Show User Toolbar** (Настройка | Показать пользовательскую панель инструментов). Пользовательская панель инструментов исчезнет.
13. Снова выберите команду меню **Settings | Show User Toolbar**. Появится пользовательская панель инструментов.
14. Измените положение пользовательской панели инструментов в окне приложения, оставив ее фиксированной, и закройте приложение.
15. Снова откройте приложение. Пользовательская панель инструментов сохранит свое новое положение, как показано на рис. 5.19.
16. Закройте приложение.

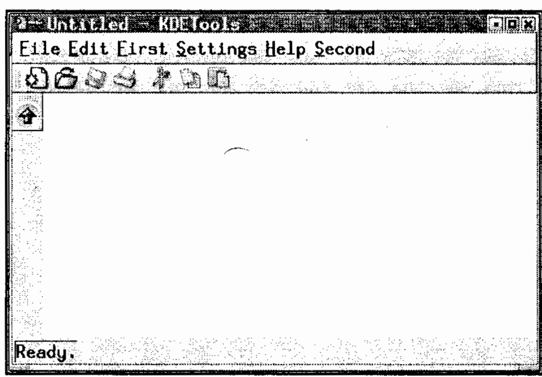


Рис. 5.19. Сохранение положения пользовательской панели инструментов

В приложениях KDE создание пользовательского значка в панели инструментов представляет собой достаточно сложную задачу. Используемые в этих приложениях значки хранятся в каталоге **/usr/share/icons/hicolor** как файлы с расширением png. Причем в этих приложениях используются два типа значков: первый из них имеет размер 16×16 точек и выводится в меню около соответствующей команды. Второй — размером 32×32 точки, выводится в панели инструментов. В некоторых случаях используются и значки размером 22×22 точки. Для хранения каждого из этих значков в данном каталоге создан свой подкаталог, имя которого совпадает с размерностью хранящихся в нем значков. Для простоты в качестве значка пользовательской команды меню был выбран файл up.png, присутствующий в каждом из этих подкаталогов.

Для того чтобы команда имела свой значок в панели инструментов, его нужно указать в конструкторе объекта класса `KAction`, соответствующего

данной команде меню. Значок может быть задан ссылкой на объект класса `QIconSet` или непосредственно именем файла значка. Причем в первом случае будет отображаться только значок данного размера. То есть он будет, например, выводиться в меню, но будет отсутствовать в панели инструментов, хотя она и без него будет работать.

Для вывода на экран и удаления с него пользовательской панели инструментов в меню **Settings** добавляется команда **Show User Toolbar**. Для работы с этой командой используется объект класса `KToggleAction`, позволяющий устанавливать и снимать флажки у команд меню. Чтобы вывести флажок у соответствующей команды меню, для создания объекта следует использовать конструктор класса, не принимающий в качестве аргумента объект класса `QIconSet` или имя файла значка. В противном случае разработчику самому придется заниматься индикацией состояния команды. В остальном конструктор класса `KToggleAction` полностью аналогичен конструктору класса `KAction`.

При создании панели инструментов она по умолчанию растягивается на все свободное место в родительском окне. Эта стратегия хороша, если в приложении имеется всего одна панель инструментов. При наличии в приложении двух панелей инструментов, расположенных по одной стороне окна, они делят свободное место пополам. Если одна из панелей инструментов содержит всего одну кнопку, как в нашем случае, это выглядит странно. Поэтому для пользовательской панели инструментов была вызвана функция `KToolBar::setFullSize` с аргументом `false`, приводящим ее размеры в соответствие с ее содержимым. Для получения указателя на объект пользовательской панели инструментов была использована функция `KMainWindow::toolBar`, которой в качестве аргумента было передано имя искомой панели инструментов. Это имя указано в файле kdetoolsui.rc.

Для сохранения текущего состояния элементов пользовательского интерфейса при закрытии приложения применяется функция `KDEToolsApp::saveOptions`. Эта функция активно использует объект класса `KConfig`, позволяющий записывать и извлекать информацию из системы конфигурации KDE. Структура системы конфигурации KDE подозрительно напоминает небезызвестный системный реестр Windows и так же, как и он, содержит записи, доступ к которым осуществляется по ключу. Записи разбиваются на группы. Если для записи в явном виде не указана группа, она помещается в специальную группу, используемую по умолчанию.

Для задания группы, в которую будет производиться запись информации или из которой она будет извлекаться, применяется функция `KConfigBase::setGroup`, которой в качестве аргумента передается имя группы. Для сохранения отдельной записи используется функция `KConfigBase::writeEntry`, в первом аргументе которой передается ключ, по которому следует сохранить запись, а во втором — ее содержимое.

Для записи в систему конфигурации KDE могут быть использованы и другие способы, например список последних открытых файлов сохраняется вы-

зовом функции `KRecentFilesAction::saveEntries`, в первом аргументе которой передается указатель на объект класса `KConfig`, используемый для доступа к системе конфигурации, а во втором — имя группы, в которой будут сохранены записи.

В рассматриваемом приложении для пользовательской панели инструментов сохраняется ее состояние, возвращаемое функцией `KToggleAction::isChecked`, а также ее положение, возвращаемое функцией `KToolBar::barPos`.

Восстановление состояния элементов пользовательского интерфейса при повторном запуске приложения производится в функции `KDEToolsApp::readOptions`. Прежде всего, в ней вызывается функция `KConfigBase::setGroup`, выбирающая для чтения информации ту же группу, в которую производилась ее запись.

Для чтения состояния пользовательской панели инструментов вызывается функция `KConfigBase::readBoolEntry`, в первом аргументе которой указывается ключ, по которому хранится информация, а во втором — значение, которое должна возвратить функция в том случае, если указанный ключ отсутствует. После получения нового состояния команды меню оно устанавливается вызовом функции `KToggleAction::setChecked` и вызывается приемник, обрабатывающий произошедшее изменение состояния команды.

Для получения нового положения пользовательской панели инструментов вызывается функция `KConfigBase::readNumEntry`, отличающаяся от рассмотренной ранее функции `readBoolEntry` только типом возвращаемого значения. Возвращаемое функцией `readNumEntry` значение преобразуется к перечислимому типу `KToolBar::BarPosition` и используется в качестве аргумента функции `KToolBar::setBarPos`, производящей все необходимые действия по позиционированию панели инструментов.

Приемник `KDEToolsApp::slotViewUserToolBar`, обрабатывающий команду меню **Settings | Show User Toolbar**, полностью аналогичен приемнику `KDEToolsApp::slotViewToolBar`, обрабатывающему команду меню **Settings | Show Toolbar** (Настройка | Показать панель инструментов). И в том и в другом приемнике в строку состояния вызовом приемника `KDEToolsApp::slotStatusMsg` выводится текст сообщения о выполняемой операции и вызывается функция `KToggleAction::isChecked`, возвращающая текущее состояние команды меню, в зависимости от которого вызывается функция `KToolBar::show`, выводящая панель инструментов на экран, или функция `KToolBar::hide`, убирающая ее с экрана. В заключение работы приемников в них вызывается приемник `slotViewToolBar`, восстанавливающий в строке состояния выводимый в ней по умолчанию текст.

Теперь нам осталось только рассмотреть изменения, внесенные в файл kdetoolsui.rc. В отличие от процедуры добавления нового пользовательского меню, которую можно было произвести, пользуясь уже имеющейся в файле информацией, для процедуры добавления пользовательской панели инструментов этой информации недостаточно. Это связано с тем, что здесь требу-

ется добавить команду в уже имеющееся меню, никак не описанное в этом файле, и создать новую панель инструментов при отсутствии подходящего шаблона. Всю необходимую информацию можно получить в файле ui_standards.rc, расположенном в каталоге /usr/share/config/ui/ и содержащем описание используемых по умолчанию ресурсов приложения.

Работа со строкой состояния

Работа со строкой состояния в приложениях Qt уже была нами рассмотрена ранее, и поскольку класс `KStatusBar` является потомком класса `QStatusBar`, все рассмотренные методики могут быть использованы и в приложениях KDE, однако в класс `KStatusBar` были добавлены функции, существенно облегчающие работу со строкой состояния. Продемонстрируем их использование для решения той же задачи, что и в приложении Qt.

Чтобы внести соответствующие изменения в приложение **KDETools**:

1. Откройте приложение **KDETools** на редактирование.
2. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `KDEToolsApp` и выберите в появившемся контекстном меню команду **Add slot**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Slots**.
3. В текстовое поле **Declaration** введите сигнатуру функции `onUpdateStatusBar(int, int)`, в текстовое поле **Documentation** — комментарий `Updates Status Bar panes` (Вносит изменения в панели строки состояния) и нажмите кнопку **Apply**. В класс будет добавлен новый приемник, откроется окно редактирования файла kdetools.cpp и текстовый курсор будет помещен в заготовку нового приемника.
4. Измените приемник `QtToolsApp::onUpdateStatusBar` в соответствии с текстом листинга 5.17.

Листинг 5.17. Приемник `onUpdateStatusBar`

```
/** Вносит изменения в панели строки состояния */
void KDEToolsApp::onUpdateStatusBar(int x, int y)
{
 statusBar()->changeItem( i18n("X = %1").arg( x), ID_STATUS_X);
 statusBar()->changeItem( i18n("Y = %1").arg( y), ID_STATUS_Y);
}
```

5. В файле kdetools.cpp после строки `#define ID_STATUS_MSG 1` вставьте строки

```
#define ID_STATUS_X  2
#define ID_STATUS_Y  3
```

6. Измените функцию `KDEToolsApp::initStatusBar` в соответствии с текстом листинга 5.18.

Листинг 5.18. Функция `initStatusBar`

```
void KDEToolsApp::initStatusBar()
{
 ///////////////////////////////////////////////////////////////////
 // СТРОКА СОСТОЯНИЯ
 // ЧТО ДЕЛАТЬ: добавьте свои собственные элементы для отображения
 // текущего состояния приложения.
 statusBar()->insertItem(i18n("Ready."), ID_STATUS_MSG);
 statusBar()->insertFixedItem(i18n(" X =    "), ID_STATUS_X, true);
 statusBar()->insertFixedItem(i18n(" Y =    "), ID_STATUS_Y, true);
}
```

7. Измените функцию `KDEToolsApp::initView` в соответствии с текстом листинга 5.19.

Листинг 5.19. Функция `initView`

```
void KDEToolsApp::initView()
{
 ////////////////////////////////////////////////////////////////////
 // Создайте здесь главное окно приложения, управляемое объектом
 // класса KTMainWindow, и свяжите окно с документом для отображения
 // его содержимого

 view = new KDEToolsView(this);
 doc->addView(view);
 setCentralWidget(view);
 setCaption(doc->URL().fileName(),false);

 view->setMouseTracking( true);

 connect(view, SIGNAL(updateStatusBar(int, int)),
               this, SLOT(onUpdateStatusBar(int, int)));
}
```

8. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `KDEToolsView` и выберите в появившемся контекстном меню команду **Add signal**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Signals**.
9. В текстовое поле **Declaration** введите сигнатуру функции `updateStatusBar(int, int)`, в текстовое поле **Documentation** — коммен-

тарий `Updates Status Bar panes` и нажмите кнопку **Apply**. В класс будет добавлен новый сигнал.

10. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `KDEToolsView` и выберите в появившемся контекстном меню команду **Add member function**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Methods**.
11. В текстовое поле **Type** введите тип возвращаемого значения `void`, в текстовое поле **Declaration** — сигнатуру функции `mouseMoveEvent(QMouseEvent*)`, в группе **Access** установите переключатель **Protected**, в группе **Modifiers** — флажок **Virtual**, в текстовое поле **Documentation** введите комментарий `Handles the Event_MouseMove event` (Обрабатывает перемещение мыши) и нажмите кнопку **Apply**. В класс будет добавлен новый приемник, откроется окно редактирования файла kdetoolsview.cpp и текстовый курсор будет помещен в заготовку новой функции.
12. Измените функцию `KDEToolsView::mouseMoveEvent` в соответствии с текстом листинга 5.20.

Листинг 5.20. Функция mouseMoveEvent

```
/** Обрабатывает перемещение мыши */
void KDEToolsView::mouseMoveEvent(QMouseEvent* e)
{
 updateStatusBar(e->x(), e->y());
}
```

13. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Появится окно приложения.
14. Поместите указатель мыши в рабочую область окна. В строке состояния появятся его координаты, как показано на рис. 5.20.
15. Закройте приложение.

Как видно из сравнения двух методик, работа со строкой состояния в приложениях KDE действительно облегчена, поскольку теперь основная работа связана не с ней, а с подготовкой информации для нее.

Прежде всего, теперь отпала необходимость в создании специальных оконных объектов для реализации панелей строки состояния. Эту работу взял на себя объект класса `KStatusBar`. Для создания панели строки состояния теперь достаточно вызвать функцию `KStatusBar::insertItem` или `KStatusBar::insertFixedItem`, передав в первом ее аргументе текст, который изначально будет выводиться в этой панели, а во втором — целочисленный идентификатор панели. Функция `insertFixedItem` отличается от функции `insertItem`

тем, что размер создаваемой панели, определяемый размером изначально выводимого в нее текста, остается неизменным при изменении текста в панели. Значение `true` третьего аргумента функции `insertFixedItem` указывает на то, что создается постоянная панель строки состояния, располагающаяся в ее правой части.

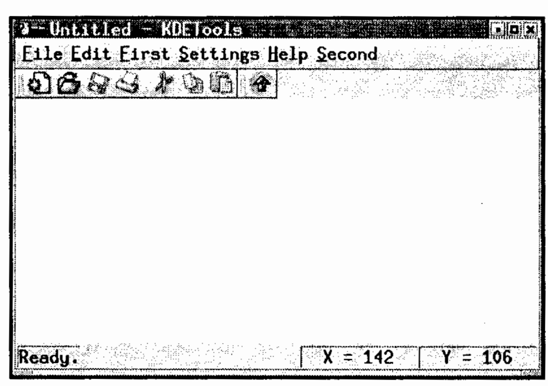


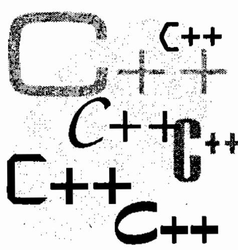
Рис. 5.20. Координаты указателя мыши в строке состояния

Для замены текста в панели строки состояния используется функция `KStatusBar::changeItem`, в первом аргументе которой передается новый текст, а во втором — целочисленный идентификатор панели строки состояния.

Для вывода информации во временную панель строки состояния используется созданный мастером приемник `KDEToolsApp::slotStatusMsg`, в котором перед вызовом функции `changeItem` вызывается функция `KStatusBar::clear`. Необходимость создания такого приемника вызвана, по всей видимости, недостаточной надежностью работы функции `changeItem`.

В демонстрационном приложении **QtTools** связывание приемника класса приложения и сигнала класса представления производилось в функции `createClient`. Поскольку в данном приложении такая функция отсутствует, эта операция была произведена в функции `KDEToolsApp::initView` сразу же после создания и инициализации объекта класса представления.

ГЛАВА 6



Вывод информации на экран

При запуске на исполнение заготовок многооконных и однооконных приложений, создаваемых мастером **ApplicationWizard** среды разработки KDevelop, на экран выводится пустое окно документа. Данная глава посвящена вопросу о том, каким образом это окно может быть заполнено полезной информацией.

Существует четыре принципиально разные возможности заполнить окно документа в приложении:

1. Использовать линии для рисования фигур.
2. Использовать кисти для заполнения замкнутых геометрических фигур.
3. Вывести текстовую информацию.
4. Вывести битовые образы.

Рисование фигур

Рисование мышью линий в окне является, наверное, простейшей задачей, использующей графический интерфейс. Поэтому именно с ее решения мы и начнем ознакомление с реализацией графического интерфейса в приложениях KDevelop. Для этого нами будет создано демонстрационное приложение **Line**, текст которого можно найти на прилагаемом к книге CD.

Чтобы самостоятельно создать это приложение:

1. Выберите команду меню **Project | New** (Проект | Создать). Появится диалоговое окно **ApplicationWizard** (Мастер создания приложений).
2. В окне иерархического списка выделите тип создаваемого приложения **KDE Normal** (Однооконное приложение KDE) и нажмите кнопку **Next** (Далее). Раскроется вторая панель мастера.

3. В текстовое поле **Project name** (Имя проекта) введите имя создаваемого приложения **Line** и нажмите кнопку **Create** (Создать). Раскроется последняя панель мастера и начнется процесс создания заготовки приложения.
4. По завершении процесса создания заготовки приложения нажмите кнопку **Exit** (Выход). Окно мастера создания приложений закроется.
5. В окне иерархических списков раскройте вкладку **Classes** (Классы), щелкните правой кнопкой мыши по имени класса `LineDoc` и выберите в появившемся контекстном меню команду **Add member variable** (Добавить переменную члена класса). Появится диалоговое окно **Class Properties** (Свойства класса), раскрытое на вкладке **Attributes** (Атрибуты).
6. В текстовое поле **Type** (Тип) введите тип переменной `QPen`, в текстовое поле **Name** (Имя) — идентификатор переменной `myPen`, в текстовое поле **Documentation** (Документация) — комментарий `User pen` (Пользовательское перо) и нажмите кнопку **Apply** (Применить). В класс будет добавлена новая переменная.
7. В открывшемся после добавления новой переменной окне редактирования файла linedoc.h после строки `#include <qlist.h>` вставьте строку

   ```
   #include <qpen.h>
   ```

8. Щелкните правой кнопкой мыши в окне редактирования файла linedoc.h и выберите в появившемся контекстном меню команду **Switch Header/Source** (Переключение файлов заголовка и реализации). Откроется окно редактирования файла linedoc.cpp.
9. Измените функцию `LineDoc::newDocument` в соответствии с текстом листинга 6.1.

Листинг 6.1. Функция newDocument

```
bool LineDoc::newDocument()
{
 /////////////////////////////////////////////////
 // ЧТО ДЕЛАТЬ: Произведите здесь инициализацию документа
 /////////////////////////////////////////////////
 myPen=QPen( Qt::black, 3);
 modified=false;
 doc_url.setFileName(i18n("Untitled"));

 return true;
}
```

10. Во вкладке **Classes** окна иерархического списка щелкните правой кнопкой мыши по имени класса `LineView` и выберите в появившемся контек-

стном меню команду **Add member function** (Добавить функцию члена класса). Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Methods** (Методы).

11. В текстовое поле **Type** (Тип возвращаемого значения) введите тип возвращаемого значения `void`, в текстовое поле **Declaration** (Сигнатура) — сигнатуру функции `mousePressEvent(QMouseEvent *)`, в группе **Access** установите переключатель **Protected**, в группе **Modifiers** — флажок **Virtual**, в текстовое поле **Documentation** введите комментарий `Handles mouse press event` (Обрабатывает нажатие кнопки мыши) и нажмите кнопку **Apply**. В класс будет добавлена новая функция.
12. Повторите пункты 10 и 11 для добавления в класс функции `mouseReleaseEvent(QMouseEvent *)`, снабдив ее комментарием `Handles mouse release event` (Обрабатывает отпускание кнопки мыши).
13. Повторите пункты 10 и 11 для добавления в класс функции `mouseMoveEvent(QMouseEvent *)`, снабдив ее комментарием `Handles mouse move event` (Обрабатывает перемещение мыши).
14. Повторите пункты 10 и 11 для добавления в класс функции `mouseDoubleClickEvent(QMouseEvent *)`, снабдив ее комментарием `Handles mouse double click event` (Обрабатывает двойной щелчок кнопкой мыши).
15. В открывшемся после добавления функций окне редактирования файла lineview.cpp измените эти функции в соответствии с текстом листинга 6.2.

Листинг 6.2. Функции обработки событий класса `LineView`

```
/** Обрабатывает нажатие кнопки мыши */
void LineView::mousePressEvent( QMouseEvent * e)
{
 if( e->button() & LeftButton)
 {
  isPressed = true;

  painter.begin( this);
  painter.setPen( getDocument()->myPen);
  painter.moveTo( e->pos());
 }
 else
  getDocument()->myPen.setWidth( getDocument()->myPen.width() + 5);

}

/** Обрабатывает отпускание кнопки мыши */
void LineView::mouseReleaseEvent( QMouseEvent *)
```

```
{
 painter.end();
 isPressed = false;
}

/** Обрабатывает перемещение мыши */
void LineView:: mouseMoveEvent( QMouseEvent * e)
{
 if ( isPressed)
  painter.lineTo( e->pos());
}

/** Обрабатывает двойной щелчок кнопкой мыши */
void LineView::mouseDoubleClickEvent(QMouseEvent * e)
{
 if( e->button() & RightButton)
 {
  int nWidth = getDocument()->myPen.width();

  if( nWidth > 10)
   nWidth -= 10;
  else
   nWidth = 1;

   getDocument()->myPen.setWidth( nWidth);
 }
}
```

16. Измените конструктор класса `LineView` в соответствии с текстом листинга 6.3.

Листинг 6.3. Конструктор класса `LineView`

```
LineView::LineView(QWidget *parent, const char *name) :
QWidget(parent, name)
{
 setBackgroundMode(PaletteBase);
 setCursor( Qt::crossCursor);
 isPressed=false;
}
```

17. Во вкладке **Classes** окна иерархического списка щелкните правой кнопкой мыши по имени класса `LineView` и выберите в появившемся контекстном меню команду **Add member variable**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes**.
18. В текстовое поле **Type** (Тип) введите тип переменной `bool`, в текстовое поле **Name** (Идентификатор) — идентификатор переменной `isPressed`,

в группе **Access** (Модификатор права доступа) установите переключатель **Private**, в текстовое поле **Documentation** введите комментарий `Indicates the state of mouse button` (Отражает состояние кнопки мыши) и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.

19. Повторите пункты 17 и 18 для включения в класс переменной `painter`, имеющей тип `QPainter` и комментарий `Paint object` (Объект класса рисования).

20. В открывшемся после добавления переменных окне редактирования файла lineview.h после строки `#include <qwidget.h>` вставьте строку

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

21. Выберите команду меню **Debug | Start** (Отладка | Пуск) или нажмите кнопку **Debug** в панели инструментов. Приложение будет откомпилировано и появится его главное окно.

22. Поместите указатель мыши в рабочую область окна, нажмите левую кнопку мыши и переместите ее указатель. На экране появится линия.

23. Щелкните правой кнопкой мыши и повторите пункт 22. Рисуемая линия станет толще.

24. Дважды щелкните правой кнопкой мыши и повторите пункт 22. Рисуемая линия станет прежней толщины. В результате произведенных действий окно приложения примет вид, изображенный на рис. 6.1.

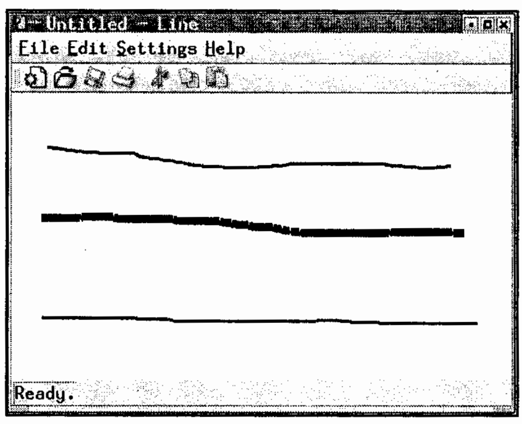


Рис. 6.1. Вывод линий в окне

25. Измените размеры окна. Его рабочая область очистится.

26. Закройте приложение.

Для связи приложения с устройствами отображения информации в библиотеке Qt используется объект класса `QPainter`, в котором объединены основ-

ные функции, применяемые для вывода информации на экран или на другое устройство вывода графической информации, а также функции, служащие для задания параметров вывода этой информации.

Для использования объекта класса `QPainter` необходимо предварительно связать его с объектом класса `QPaintDevice` или одного из его дочерних классов, описывающих используемое графическое устройство. Эта операция может производиться непосредственно в процессе создания объекта класса `QPainter`, для чего конструктору данного класса передается указатель на объект класса `QPaintDevice`, или же вызовом функции `QPainter::begin`, как это сделано в рассматриваемом приложении.

В указанном приложении связывание объекта класса `QPainter` с объектом класса `QPaintDevice` производится в функции обработки события `LineView::mousePressEvent`, вызываемой при нажатии пользователем любой кнопки мыши. Поскольку в приложении задействованы как левая, так и правая кнопки мыши, в этой функции сначала определяется нажатая кнопка. Если нажата левая кнопка мыши, в данной функции устанавливается флаг нажатой кнопки, производится связывание объекта класса `QPainter` с объектом класса `QPaintDevice`, вызовом функции `QPainter::setPen` в контексте устройства устанавливается *перо*, объект которого хранится в документе, и текушая позиция пера вызовом функции `QPainter::moveTo` устанавливается в текущую позицию указателя мыши, получаемую вызовом функции `QMouseEvent::pos`.

Объект текущего пера удобнее было бы создать в классе представления, но правила хорошего тона требуют, чтобы вся изменяемая и сохраняемая информация приложения хранилась в объекте класса документа. Не будем отступать от этих правил, хотя мы и не собираемся сохранять никакой информации. Для доступа к объекту класса документа из связанного с ним класса приложения используется создаваемая мастером функция `LineView::getDocument`.

Если вызов функции `mousePressEvent` не был вызван нажатием левой кнопки мыши, то, для простоты, считается, что была нажата правая кнопка мыши. В этом случае производится увеличение ширины используемого пера. Для этого, с помощью функции `QPen::width` фиксируется текущая ширина используемого пера, полученное значение увеличивается и результат передается функции `QPen::setWidth`, устанавливающей новую ширину пера.

Текст функции `LineView::mouseReleaseEvent`, обрабатывающей сообщение об отпускании пользователем кнопки мыши, предельно прост: в нем вызывается функция `QPainter::end`, завершающая процесс рисования и освобождающая используемые при этом ресурсы, а затем сбрасывается флаг нажатой кнопки. Следует обратить внимание на то, что в данной функции не проверяется, какая кнопка мыши была отпущена.

Функция `LineView::mouseMoveEvent`, обрабатывающая сообщения о перемещении мыши, также не отличается особой сложностью: в ней проверяется состояние флага рисования и, если он установлен, вызывается функция `QPainter::lineTo` для проведения линии между текущей позицией указателя мыши и текущей позицией пера. После завершения работы этой функции текущая позиция пера совпадает с текущей позицией указателя мыши.

Функция `LineView::mouseDoubleClickEvent`, обрабатывающая двойной щелчок кнопкой мыши, используется для уменьшения ширины текущего пера. В этой функции, прежде всего, проверяется, что двойной щелчок был произведен правой кнопкой мыши. Если это так, то текущая ширина пера сохраняется во временной переменной. Необходимость этой операции обусловлена тем, что двойной щелчок кнопкой состоит из двух одиночных щелчков, обработка первого из которых уже привела к увеличению ширины пера, и тем, что ширина пера не может принимать отрицательных значений. Поэтому в данной функции производится проверка текущего значения ширины пера и, в зависимости от результата, используются различные процедуры его корректировки. Новое значение ширины пера передается в качестве аргумента функции `QPen::setWidth`.

Как видно из рис. 6.1, представляющего результат работы приложения, качество вывода линий функциями библиотеки Qt оставляет желать лучшего, но можно надеяться на то, что это будет исправлено в следующих версиях.

Работа с кистью

Если перья используются для рисования линий и фигур, то кисти предназначены для заполнения замкнутых пространств определенными цветами или текстурами. *Кисть* (Brush) может заполнять область равномерным фоном любого цвета, использовать стандартные трафареты (pattern) или задаваемые пользователем битовые поля.

Текст приложения **Brush**, демонстрирующего принципы работы с кистью, можно найти на прилагаемом к книге CD.

Чтобы самостоятельно создать это приложение:

1. Выберите команду меню **Project | New**. Появится диалоговое окно **ApplicationWizard**.
2. В окне иерархического списка выделите тип создаваемого приложения **KDE MDI** и нажмите кнопку **Next**. Раскроется следующая панель мастера.
3. В текстовое поле **Project name** введите имя создаваемого приложения **Brush** и нажмите кнопку **Create**. Раскроется последняя панель мастера и начнется процесс создания заготовки приложения.

7 Зак. 113

4. По завершении процесса создания заготовки приложения нажмите кнопку **Exit**. Окно мастера создания приложений закроется.
5. В окне иерархических списков раскройте вкладку **Classes**, щелкните правой кнопкой мыши по имени класса `BrushDoc` и выберите в появившемся контекстном меню команду **Add member variable**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes**.
6. В текстовое поле **Type** введите тип переменной `int`, в текстовое поле **Name** — идентификатор переменной `brushStyle`, в текстовое поле **Documentation** — комментарий `Style of brush` (Стиль кисти) и нажмите кнопку **Apply**. В класс будет добавлена новая переменная и откроется окно редактирования файла brushdoc.h.
7. В начале заголовка класса `BrushDoc` замените строку `friend BrushView;` строкой

   ```
   friend class BrushView;
   ```

8. Щелкните правой кнопкой мыши в окне редактирования файла brushdoc.h и выберите в появившемся контекстном меню команду **Switch Header/Source**. Откроется окно редактирования файла brushdoc.cpp.
9. Измените в нем функцию `BrushDoc::newDocument` в соответствии с текстом листинга 6.4

Листинг 6.4. Функция newDocument

```
bool BrushDoc::newDocument()
{
 ///////////////////////////////////////////////////
 // ЧТО ДЕЛАТЬ: Поместите сюда инициализацию документа
 ///////////////////////////////////////////////////
 brushStyle=0;
 modified=false;
 return true;
}
```

10. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `BrushView` и выберите в появившемся контекстном меню команду **Add member function**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Methods**.
11. В текстовое поле **Type** введите тип возвращаемого значения `void`, в текстовое поле **Declaration** — объявление функции `mousePressEvent(QMouseEvent *)`, в группе **Access** установите переключатель **Protected**, в группе **Modifiers** установите флажок **Virtual**, в текстовое поле **Documentation** введите комментарий `Handles mouse press event`

(Обрабатывает нажатие кнопки мыши) и нажмите кнопку **Apply**. В класс будет добавлена новая функция.

12. В открывшемся после добавления функции окне редактирования файла brushview.cpp измените эту функцию в соответствии с текстом листинга 6.5.

Листинг 6.5. Функция mousePressEvent

```
/** Обрабатывает нажатие кнопки мыши */
void BrushView::mousePressEvent( QMouseEvent * e)
{
 if( e->button() & LeftButton)
 {
  QPainter pnt( this);
  QBrush  myBrush( (BrushStyle) getDocument()->brushStyle);
  QPen   myPen(Qt::blue, 1);

  pnt.setPen( myPen);

  pnt.drawRect( e->x(), e->y(), 20, 20);
  pnt.fillRect( e->x(), e->y(), 20, 20, myBrush);
 }
 else
 {
  getDocument()->brushStyle++;
  getDocument()->brushStyle %= 15;
 }
}
```

13. Щелкните правой кнопкой мыши в окне редактирования файла brushview.cpp и выберите в появившемся контекстном меню команду **Switch Header/Source**. Откроется окно редактирования файла brushview.h.

14. В начале заголовка класса `BrushView` замените строку `friend BrushDoc;` строкой

    ```
    friend class BrushDoc;
    ```

15. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Приложение будет откомпилировано и появится его главное окно.

16. Поместите указатель мыши в рабочую область окна и щелкните левой кнопкой мыши. На экране появится квадрат в синей рамке.

17. Щелкните правой кнопкой мыши и повторите пункт 16. Будет выведен черный квадрат.

18. Несколько раз повторите пункт 17. Заполнение квадратов будет изменяться, как это показано на рис. 6.2.

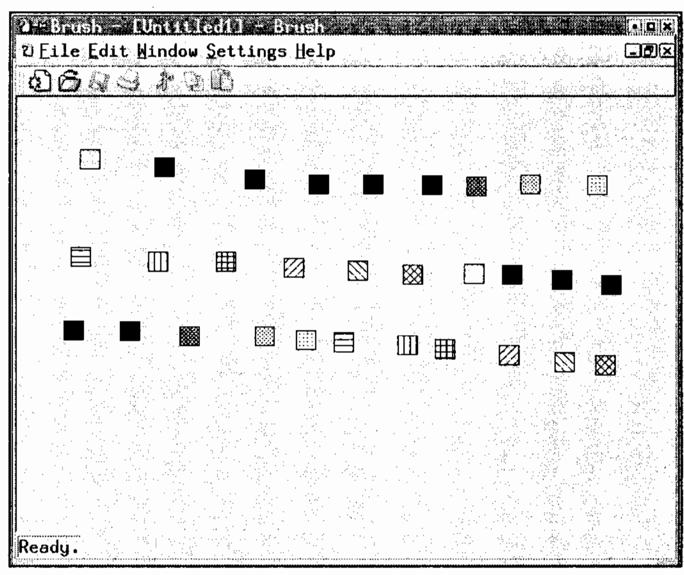


Рис. 6.2. Результат работы приложения **Brush**

19. Измените размер окна. Его рабочая область очистится.
20. Закройте приложение.

Для работы с кистью в библиотеке Qt используется объект класса `QBrush`. В данном приложении для создания этого объекта использовался самый простой конструктор, создающий черную кисть с заданным стандартным трафаретом. Другой часто применяемый конструктор данного класса позволяет создавать кисти любого цвета, по умолчанию не использующие трафаретов, а равномерно заливающие указанную область. Но, поскольку трафарет задается одним целым числом, а цвет — тремя, и поскольку в данной книге не предполагается использовать цветные иллюстрации, в приложении изменяется трафарет, а не цвет.

Как и в предыдущем случае, информация об изменяемых параметрах приложения хранится в объекте класса документа. В данном случае — это индекс трафарета. Этот индекс может принимать 16 различных значений, но, поскольку шестнадцатое значение используется для задания пользовательского трафарета, в этом приложении применяются только 15 значений индекса.

Все основные операции приложения производятся в функции `BrushView::mousePressEvent`, обрабатывающей нажатия пользователем кнопок мыши.

Так как эта функция обрабатывает нажатия всех кнопок, в ней, прежде всего, устанавливается, какая из кнопок мыши была нажата.

Если была нажата левая кнопка мыши, в рабочую область окна выводится квадрат в синей рамке, заполненный кистью текущего стиля. Для этого создаются объекты классов `QPainter`, `QBrush` и `QPen`. Для установки текущего пера в контексте устройства применяется описанная выше функция `QPainter::setPen`, после чего это перо используется для отображения внешней рамки заполняемой области вызовом функции `QPainter::drawRect`. Для заполнения области нам нет необходимости выбирать кисть в контекст устройства, поскольку объект класса кисти передается в качестве аргумента функции `QPainter::fillRect`, выполняющей эту задачу.

Для изменения текущего стиля кисти нами используется щелчок правой кнопки мыши (для простоты, эта же операция выполняется и щелчком средней кнопки мыши). Поэтому, если была нажата не левая кнопка мыши, в функции `BrushView::mousePressEvent` производится увеличение текущего значения стиля кисти и, чтобы это значение не вышло за границы допустимого диапазона, полученное значение делится по модулю на используемое нами число стилей.

Перерисовка окна

Данное приложение так же, как и предыдущее, не сохраняет содержимого своей рабочей области при изменении ее размеров. Это связано с тем, что при изменении размеров окна приложение генерирует события, функции обработки которых, прежде всего, очищают рабочую область окна. Для восстановления информации в окне нам необходимо перегрузить эти функции. Кроме того, окно приложения по непонятной причине имеет серый фон, хотя окно однооконного приложения имеет белый фон.

Чтобы обеспечить белый фон и перерисовку окна:

1. Откройте приложение **Brush**.
2. Раскройте вкладку **Classes** окна иерархических списков, щелкните правой кнопкой мыши по имени класса `BrushDoc` и выберите в появившемся контекстном меню команду **Add member variable**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes**.
3. В текстовое поле **Type** введите тип переменной `int`, в текстовое поле **Name** — идентификатор двумерного массива `buf[128][3]`, в текстовое поле **Documentation** — комментарий `Data array` (Массив данных) и нажмите кнопку **Apply**. В класс будет добавлен новый массив.
4. Повторите пункты 2 и 3 для добавления в класс переменной `count`, имеющей тип `int`, сопроводив ее комментарием `Array counter` (Счетчик массива).

5. Откройте окно редактирования файла brushdoc.cpp и измените в нем функцию `BrushDoc::newDocument` в соответствии с текстом листинга 6.6.

Листинг 6.6. Функция `newDocument`

```
bool BrushDoc::newDocument()
{
 /////////////////////////////////////////////////
 // ЧТО ДЕЛАТЬ: Поместите сюда инициализацию документа
 /////////////////////////////////////////////////
 brushStyle=0;
 count = 0;
 modified=false;
 return true;
}
```

6. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `BrushView` и выберите в появившемся контекстном меню команду **Add member function**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Methods**.
7. В текстовое поле **Type** введите тип возвращаемого значения `void`, в текстовое поле **Declaration** — сигнатуру функции `paintEvent(QPaintEvent *)`, в группе **Access** установите переключатель **Protected**, в группе **Modifiers** — флажок **Virtual**, в текстовое поле **Documentation** введите комментарий `Handles paint event` (Производит перерисовку окна) и нажмите кнопку **Apply**. В класс будет добавлен новая виртуальная функция.
8. В открывшемся после добавления функции окне редактирования файла brushview.cpp измените функции обработки событий класса `BrushView` в соответствии с текстом листинга 6.7.

Листинг 6.7. Функции обработки событий класса `BrushView`

```
/** Обрабатывает нажатие кнопки мыши */
void BrushView::mousePressEvent( QMouseEvent * e)
{
 BrushDoc* doc = getDocument();

 if( e->button() & LeftButton)
 {
  QPainter pnt( this);
  QBrush  myBrush( (BrushStyle) doc->brushStyle);
  QPen   myPen(Qt::blue, 1);
```

```
    pnt.setPen( myPen);

    pnt.drawRect( e->x(), e->y(), 20, 20);
    pnt.fillRect( e->x(), e->y(), 20, 20, myBrush);

    if( doc->count < 128)
    {
     doc->buf[doc->count][0] = doc->brushStyle;
     doc->buf[doc->count][1] = e->x();
     doc->buf[doc->count][2] = e->y();
     doc->count++;
    }
   }
   else
   {
    doc->brushStyle++;
    doc->brushStyle %= 15;
   }
  }

  /** Производит перерисовку окна */
  void BrushView::paintEvent( QPaintEvent *)
  {
   int    i;
   QPainter pnt( this);
   QPen   myPen(Qt::blue, 1);
   BrushDoc* doc = getDocument();

   pnt.fillRect( rect(), QBrush( white));

   pnt.setPen( myPen);

   for(i=0; i < doc->count; i++)
   {
    QBrush  myBrush( (BrushStyle) doc->buf[i][0]);

    pnt.drawRect( doc->buf[i][1], doc->buf[i][2], 20, 20);
    pnt.fillRect( doc->buf[i][1], doc->buf[i][2], 20, 20, myBrush);
   }
  }
```

9. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Приложение будет откомпилировано и появится его главное окно.
10. По описанной выше методике выведите в рабочую область окна приложения несколько квадратов и измените размер окна. Содержимое рабочей области окна сохранится.

11. Выберите команду меню **Window | New Window** (Окно | Новое). Появится новое окно, содержимое которого будет идентично старому. (Поскольку это окно будет развернуто на весь экран, внешне ничего не изменится.)
12. Выберите команду меню **Window | Tile** (Окно | Упорядочить). Дочерние окна многооконного приложения будут распределены в родительском окне по горизонтали, как это показано на рис. 6.3.

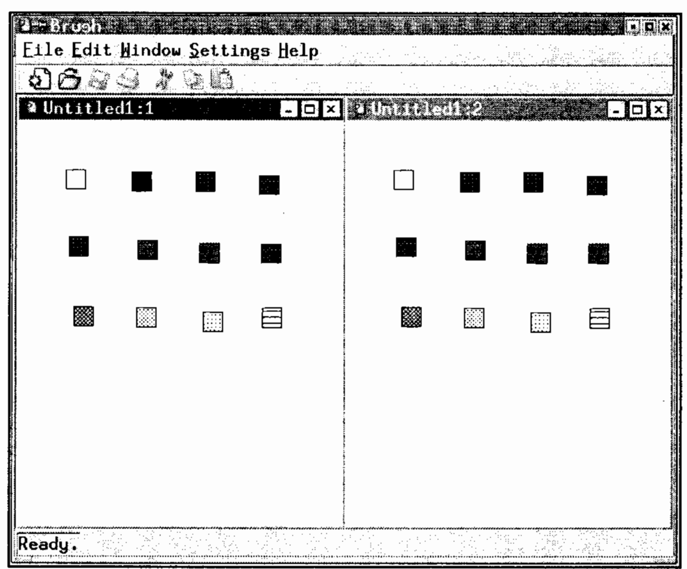


Рис. 6.3. Упорядоченные окна

13. Внесите изменения в одно из окон. Они не будут отображаться в другом окне.
14. Закройте приложение.

С целью обеспечения возможности сохранения информации в окне после его перерисовки нам пришлось добавить в приложение массив для сохранения в нем всей выведенной на экран информации. В качестве указателя на последнюю свободную строку данного массива нами используется целочисленная переменная. В соответствии с правилами хорошего тона, а также для того, чтобы изменения в одном окне документа могли бы отображаться в другом окне того же документа, все эти переменные добавлены в объект класса документа приложения.

Поскольку при щелчке левой кнопкой мыши нам теперь нужно не только выводить информацию на экран, но и дублировать ее в массиве, нам следует внести изменения в функцию `BrushView::mousePressEvent`. Исходя из существенного увеличения в данной функции обращений к объекту класса документа приложения, было признано целесообразным сохранить указатель на

этот объект в специальной переменной, а не получать его каждый раз при вызове функции `BrushView::getDocument`.

Трудолюбие пользователей безгранично, и они могут, если постараются, вывести в рабочую область любое число квадратов, для хранения информации о которых не хватит никаких массивов. Поэтому в функции `mousePressEvent` производится проверка текущего числа элементов, хранящихся в массиве, и после его заполнения сохранение в нем информации прекращается. Для восстановления информации на экране необходимо знать стиль кисти и координаты квадрата. Поэтому каждая строка массива содержит три элемента.

Процедура восстановления информации в окне при его перерисовке во многом аналогична процедуре вывода информации на экран при щелчке левой кнопкой мыши, только информация о квадратах берется из массива. Поскольку объект контекста устройства и выбранное в него перо не изменяются в процессе обновления информации, конструкторы объектов класса контекста устройства и пера, а также процедура выбора пера в контекст устройства, помещены вне цикла. С другой стороны, объект кисти необходимо создавать заново для каждого выводимого квадрата, т. к. значение его стиля хранится в массиве и является одним из изменяемых параметров.

Поскольку при внесении изменений в окно его фон остается неизменным, обновлять фон окна нужно только при его полном обновлении в функции `BrushView::paintEvent`. Для этого в ней вызывается уже рассмотренная нами функция `QPainter::fillRect`. Чтобы получить координаты рабочей области окна, которую следует заполнить цветом фона, используется функция `QWidget::rect`, а объект класса белой кисти создается непосредственно во втором аргументе данной функции.

Синхронизация объектов представления

В предыдущем разделе было показано, что рассматриваемое приложение не обеспечивает синхронизации объектов представления. То есть изменения, внесенные в документ в одном из связанных с ним объектов класса представления, не отображаются сразу же во всех остальных связанных с ним объектах.

Чтобы обеспечить синхронизацию объектов представления:

1. Откройте приложение **Brush**.
2. Откройте окно редактирования файла brushview.cpp и измените в нем функцию `BrushView::mousePressEvent` в соответствии с текстом листинга 6.8.

Листинг 6.8. Функция `mousePressEvent`

```
/** Обрабатывает нажатие кнопки мыши */
void BrushView::mousePressEvent( QMouseEvent * e)
```

```
{
 BrushDoc* doc = getDocument();

 if( e->button() & LeftButton)
 {
  QPainter pnt( this);
  QBrush  myBrush( (BrushStyle) doc->brushStyle);
  QPen   myPen(Qt::blue, 1);

  pnt.setPen( myPen);

  pnt.drawRect( e->x(), e->y(), 20, 20);
  pnt.fillRect( e->x(), e->y(), 20, 20, myBrush);

  if( doc->count < 128)
  {
   doc->buf[doc->count][0] = doc->brushStyle;
   doc->buf[doc->count][1] = e->x();
   doc->buf[doc->count][2] = e->y();
   doc->count++;

   doc->updateAllViews( this);
  }
 }
 else
 {
  doc->brushStyle++;
  doc->brushStyle %= 15;
 }
}
```

3. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Приложение будет откомпилировано и появится его главное окно.
4. По описанной выше методике заполните окно приложения квадратами, откройте второе окно и обеспечьте одновременную видимость двух окон.
5. Внесите изменения в одно из окон. Они будут отображаться в другом окне.
6. Закройте приложение.

Как следует из анализа внесенных изменений, для обеспечения в приложении синхронизации объектов представления достаточно после каждого внесения изменений в объект документа вызывать его приемник `BrushDoc::updateAllViews`, реализация которого автоматически включается в класс документа мастером создания приложений.

Поскольку все изменения в объект документа, как правило, производятся через объект класса представления, то в объекте, через который производи-

лась модификация документа, никаких изменений вносить не нужно, т. к. они уже были произведены пользователем. Поэтому в аргументе приемника `updateAllViews` передается указатель на объект класса представления, в котором вызывается данная функция, т. е. ключевое слово `this`.

Вывод текста

Вывод текстовой информации в графическом режиме не намного сложнее, чем рисование фигур и заполнение их текстурами. В Linux эта задача решается намного проще, чем в Windows, поскольку здесь не требуется инициализировать объект структуры `LOGFONT`, содержащий полную информацию об используемом шрифте.

Для демонстрации принципов вывода текстовой информации в графическое окно нами будет создано демонстрационное приложение **Text**, текст которого можно найти на прилагаемом к книге CD.

Чтобы самостоятельно создать это приложение:

1. По описанной в первом разделе методике создайте однооконное приложение с именем **Text**.
2. В окне иерархических списков раскройте вкладку **Classes**, щелкните правой кнопкой мыши по имени класса `TextDoc` и выберите в появившемся контекстном меню команду **Add member variable**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes**.
3. В текстовое поле **Type** введите тип переменной `QPoint`, в текстовое поле **Name** — идентификатор одномерного массива `pointArray[128]`, в текстовое поле **Documentation** — комментарий `Array of points` (Массив точек) и нажмите кнопку **Apply**. В класс будет добавлен новый массив.
4. Повторите пункты 2 и 3 для добавления в класс целочисленной переменной `count`, сопроводив ее комментарием **Array counter** (Счетчик массива).
5. Щелкните левой кнопкой мыши в окне редактирования файла заголовка textdoc.h, открывшегося при добавлении в класс новых переменных, и выберите в появившемся контекстном меню команду **Switch Header/Source**. Откроется окно редактирования файла textdoc.cpp.
6. Измените в нем функцию `TextDoc::newDocument` в соответствии с текстом листинга 6.9.

Листинг 6.9. Функция `newDocument`

```
bool TextDoc::newDocument()
{
```

```
  /////////////////////////////////////////////////
  // ЧТО ДЕЛАТЬ: Поместите сюда инициализацию документа
  /////////////////////////////////////////////////
  count = 0;
  modified=false;
  doc_url.setFileName(i18n("Untitled"));

  return true;
}
```

7. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `TextView` и выберите в появившемся контекстном меню команду **Add member function**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Methods**.
8. В текстовое поле **Type** введите тип возвращаемого значения `void`, в текстовое поле **Declaration** — сигнатуру функции `mousePressEvent(QMouseEvent *)`, в группе **Access** установите переключатель **Protected**, в группе **Modifiers** — флажок **Virtual**, в текстовое поле **Documentation** введите комментарий `Handles mouse press event` и нажмите кнопку **Apply**. В класс будет добавлена новая виртуальная функция, откроется окно редактирования файла textview.cpp и текстовый курсор будет помещен в заготовку новой функции.
9. Повторите пункты 7 и 8 для добавления в класс функции `paintEvent(QPaintEvent *)`, снабдив ее комментарием `Handles paint event`.
10. В открывшемся после добавления функций окне редактирования файла textview.cpp измените их в соответствии с текстом листинга 6.10.

Листинг 6.10. Функции обработки событий класса TextView

```
/** Обрабатывает нажатие кнопки мыши */
void TextView::mousePressEvent( QMouseEvent * e)
{
 TextDoc* doc = getDocument();

 if( e->button() & LeftButton)
 {
  QPainter pnt( this);
  QString  strTemp;

  strTemp = tr("[%1,%2]").arg(e->x()).arg(e->y());

  pnt.drawText( e->pos(), strTemp);

  if( doc->count < 128)
  {
   doc->pointArray[doc->count] = e->pos();
```

```
      doc->count++;
     }
    }
   }

   /** Производит перерисовку окна */
   void TextView::paintEvent( QPaintEvent * e)
   {
    int    i;
    QPainter pnt( this);
    QString  strTemp;
    TextDoc* doc = getDocument();

    QWidget::paintEvent( e);

    for(i=0; i < doc->count; i++)
    {
     strTemp =
     tr("[%1,%2]").arg(doc->pointArray[i].x()).arg(doc-
   >pointArray[i].y());
     pnt.drawText( doc->pointArray[i], strTemp);
     strTemp = "";
    }
   }
```

11. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Приложение будет откомпилировано и появится его главное окно.
12. Щелкните левой кнопкой мыши в рабочей области окна. На месте щелчка будут выведены текущие координаты указателя мыши.
13. Повторите пункт 12 несколько раз. Главное окно приложения примет вид, изображенный на рис. 6.4.
14. Измените размеры окна. Вся информация в окне сохранится.
15. Закройте приложение

Для вывода текстовой информации в графическом режиме библиотека Qt использует функцию `QPainter::drawText`. В рассматриваемом примере в первом аргументе этой функции передается ссылка на объект класса `QPoint`, содержащий координаты выводимого текста, а во втором аргументе — ссылка на объект класса `QString`, содержащий выводимый текст.

Для формирования выводимого текста используется перегруженная в классе `QString` операция сложения, позволяющая добавлять к объекту данного класса отдельные символы, и функция `QString::arg`, преобразующая целочисленную величину в строку. С целью создания временного объекта класса `QString`, необходимого для вызова этой функции, нами использован вызов

функции `QObject::tr`. Поскольку в выводимом тексте все символы интернациональны, другой причины вызова этой функции нет.

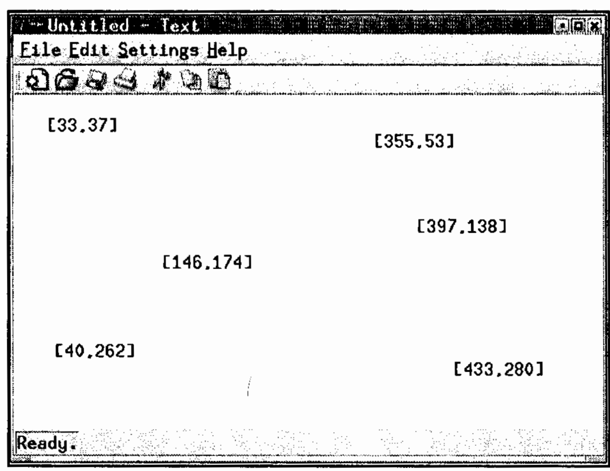


Рис. 6.4. Результат работы приложения **Text**

Работа с битовыми образами

Работа с *битовыми образами* (bitmap) является, наверное, самой сложной задачей, стоящей перед разработчиком графических приложений, но как раз использование битовых образов предоставляет ему наиболее широкие возможности в этой области. Различают *аппаратно-зависимые* (device-dependent bitmap или DDB) и *аппаратно-независимые* (device-independent bitmap или DIB). Первая разновидность битовых образов существует только в памяти компьютера и представляет собой битовый образ изображения, подготовленный приложением для вывода на экран. Вторая разновидность битовых образов может храниться в файлах (как правило, имеющих расширение bmp) и может выводиться на любом компьютере, имеющем соответствующее программное и аппаратное обеспечение, поскольку содержит информацию о цвете, достаточную для их вывода на любое графическое устройство.

Аппаратно-зависимые битовые образы

Как уже говорилось выше, аппаратно-зависимые битовые образы создаются в памяти компьютера непосредственно перед выводом изображения на экран и удаляются из памяти вместе с создавшим их приложением. Отсутствие в них избыточной информации позволяет эффективно решать с их помощью широкий круг чисто утилитарных задач. Эта разновидность битовых образов используется, например, для хранения текстур, заполняющих опре-

деленные области экрана, или для быстрого перемещения части изображения по экрану.

Для демонстрации принципов работы с аппаратно-зависимыми битовыми образами рассмотрим приложение **DDB**, текст которого можно найти на прилагаемом к книге CD.

Чтобы самостоятельно создать это приложение:

1. По описанной в первом разделе методике создайте однооконное приложение с именем **DDB**.
2. В окне иерархических списков раскройте вкладку **Classes**, щелкните правой кнопкой мыши по имени класса `DDBView` и выберите в появившемся контекстном меню команду **Add member variable**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes**.
3. В текстовое поле **Type** введите тип переменной `QRect`, в текстовое поле **Name** — идентификатор переменной `patternRect`, в группе **Access** установите переключатель **Private**, в текстовое поле **Documentation** введите комментарий `Rectangle of patterns` (Область образцов) и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.
4. Повторите пункты 2 и 3 для добавления в класс переменной `wasCopied`, имеющей тип `bool` и комментарий `Copy flag` (Флаг копирования).
5. Повторите пункты 2 и 3 для добавления в класс переменной `pictBuffer`, имеющей тип `QPixmap` и комментарий `Picture buffer` (Буфер изображений).
6. В открывшемся после добавления в класс переменных окне редактирования файла ddbview.h после строки `#include <qwidget.h>` вставьте строку

   ```
   #include <qpixmap.h>
   ```
7. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `DDBView` и выберите в появившемся контекстном меню команду **Add member function**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Methods**.
8. В текстовое поле **Type** введите тип возвращаемого значения `void`, в текстовое поле **Declaration** — сигнатуру функции `mousePressEvent(QMouseEvent *)`, в группе **Access** установите переключатель **Protected**, в группе **Modifiers** — флажок **Virtual**, в текстовое поле **Documentation** введите комментарий `Handles mouse press event` и нажмите кнопку **Apply**. В класс будет добавлена новая виртуальная функция, откроется окно редактирования файла ddbview.cpp и текстовый курсор будет помещен в заготовку новой функции.
9. Повторите пункты 7 и 8 для добавления в класс функции `paintEvent(QPaintEvent *)`, снабдив ее комментарием `Handles paint event`.

10. Измените функции обработки событий класса DDBView в соответствии с текстом листинга 6.11.

Листинг 6.11. Функции обработки событий класса DDBView

```
/** Обрабатывает нажатие кнопки мыши */
void DDBView::mousePressEvent( QMouseEvent * e)
{
 if( e->button() & LeftButton)
 {
  int wd = patternRect.width() >> 2;

  if( patternRect.contains(e->pos()))
  {
   int nRect = (e->x() - patternRect.left())/wd;

   pictBuffer = QPixmap(wd, 40);

   bitBlt( &pictBuffer, 0, 0, this, patternRect.left()
                  + nRect*wd, 10,wd, 40, CopyROP);

   wasCopied = true;
  }
  else
   if( wasCopied && (e->x() < (rect().right() - wd)) &&
                  (e->y() < (rect().bottom() - 40)))
   bitBlt( this, e->x(), e->y(), &pictBuffer, 0, 0, wd, 40, CopyROP);
 }
}

/** Производит перерисовку окна */
void DDBView::paintEvent( QPaintEvent *)
{
 if((width() > 24) && (height() > 80))
 {
  QPainter pnt( this);
  QRect   rc;

  patternRect = QRect( 10, 10, width() - 20, 40);

  int wd = patternRect.width() >> 2;

  rc = patternRect;
  rc.setRight(rc.left() + wd);

  pnt.fillRect(rc, QBrush(QColor(0, 0, 0)));

  rc.setLeft(rc.right());
  rc.setRight(rc.right() + wd);
```

```
    pnt.fillRect(rc, QBrush(QColor(255, 0, 0)));

    rc.setLeft(rc.right());
    rc.setRight(rc.right() + wd);

    pnt.fillRect(rc, QBrush(QColor(0, 255, 0)));

    rc.setLeft(rc.right());
    rc.setRight(rc.right() + wd);

    pnt.fillRect(rc, QBrush(QColor(0, 0, 255)));
   }

  wasCopied = false;
 }
```

11. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Приложение будет откомпилировано и появится его главное окно, изображенное на рис. 6.5.

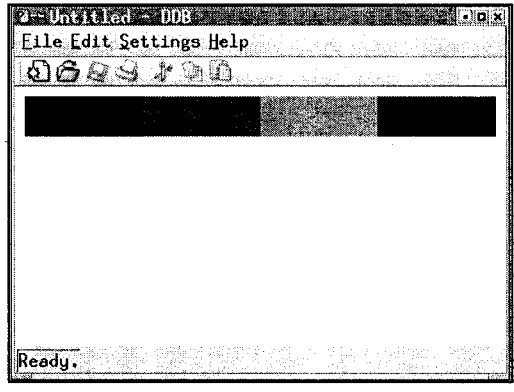


Рис. 6.5. Окно приложения DDB

12. Щелкните левой кнопкой мыши по одному из цветных прямоугольников, расположенных в верхней части главного окна приложения.
13. Щелкните левой кнопкой мыши на свободном пространстве в рабочей области окна. На месте щелчка появится прямоугольник, цвет которого будет совпадать с цветом эталонного прямоугольника, на котором был произведен последний щелчок левой кнопкой мыши.
14. Повторите пункты 12 и 13 несколько раз. Результат работы приложения приведен на рис. 6.6.
15. Измените размеры окна. В нем останутся только прямоугольники эталонных цветов.
16. Закройте приложение.

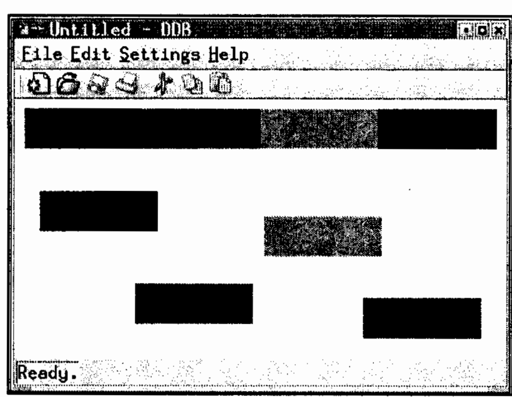


Рис. 6.6. Результат работы приложения **DDB**

Одной из основных областей применения аппаратно-зависимых битовых образов является копирование какого-либо фрагмента экрана и перенесение его в другое место. Еще одной областью их применения является подготовка сложных изображений к выводу их на экран, что позволяет избежать мерцаний при их отображении. В приложении **DDB** рассмотрена первая из этих областей применения.

В качестве копируемых изображений используются четыре прямоугольника, каждый из которых окрашен в один из основных цветов (чистый тон каждого из основных цветов и отсутствие цветов или черный цвет). Вывод этих прямоугольников на экран осуществляется в функции `DDBView::paintEvent`.

Прежде всего, в функции `paintEvent` производится проверка того, что новый размер окна достаточен для вывода прямоугольников. Для этого с помощью функций `QWidget::width` и `QWidget::height` определяются соответственно ширина и высота рабочей области окна и полученные значения сравниваются с порогами. Если размер окна недостаточен, прямоугольники не выводятся.

Для вывода прямоугольников создается объект класса `QRect`, определяющий область вывода, эта область разбивается на четыре подобласти и каждая из них заполняется вызовом функции `QPainter::fillRect`. Для перемещения левой и правой границ прямоугольника используются функции `QRect::setLeft` и `QRect::setRight`, а для создания кисти нужного цвета — конструктор класса `QBrush`, принимающий в качестве аргумента ссылку на объект класса `QColor`.

Поскольку перерисовка окна может быть связана с изменением его размеров, последним оператором функции `paintEvent` является сброс флага наличия битового образа.

Копирование битовых образов производится в функции `DDBView::mousePressEvent`. Для порядка в ней проверяется, что была нажата именно левая кнопка мыши, хотя ничто не мешает производить копирование при

нажатии любой кнопки мыши. После этого, исходя из текущего размера области эталонов, рассчитывается ширина эталона и вызовом функции `QRect::contains` определяется, расположен ли указатель мыши в области эталонов.

Если это так, определяется номер эталона, на котором расположен указатель мыши, и создается объект класса `QPixmap` соответствующего размера. После этого вызывается функция `QPaintDevice::bitBlt`, копирующая указанный эталон в объект класса `QPixmap`, и устанавливается флаг наличия битового образа.

Если указатель мыши расположен на свободном пространстве рабочей области окна, производится проверка наличия битового образа и свободного места для его вывода. Если все нормально, вызывается функция `bitBlt`, копирующая битовый образ на экран.

Аппаратно-независимые битовые образы

Как следует из их названия, аппаратно-зависимые битовые образы жестко привязаны к текущей конфигурации компьютера и обычно не могут переноситься с одного компьютера на другой. Что же делать в том случае, если вам необходимо вывести в своем приложении некоторое растровое изображение? Вам придется или жестко задать графический режим работы приложения (что делается во многих играх), или воспользоваться аппаратно-независимыми битовыми образами, содержащими всю необходимую информацию для своего вывода на любое графическое устройство.

Для демонстрации принципов работы с аппаратно-независимыми битовыми образами рассмотрим приложение **DIB**, текст которого можно найти на прилагаемом к книге CD.

Чтобы самостоятельно создать это приложение:

1. По описанной в первом разделе методике создайте однооконное приложение с именем **DIB**.
2. В окне иерархических списков раскройте вкладку **Classes**, щелкните правой кнопкой мыши по имени класса `DIBView` и выберите в появившемся контекстном меню команду **Add member function**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Methods**.
3. В текстовое поле **Type** введите тип возвращаемого значения `void`, в текстовое поле **Declaration** — сигнатуру функции `paintEvent(QPaintEvent *)`, в группе **Access** установите переключатель **Protected**, в группе **Modifiers** установите флажок **Virtual**, в текстовое поле **Documentation** введите комментарий `Handles paint event` и нажмите кнопку **Apply**. В класс будет добавлена новая виртуальная функция, откроется окно редактирования файла dibview.cpp и текстовый курсор будет помещен в заготовку новой функции.

4. Измените заготовку новой функции в соответствии с текстом листинга 6.12.

Листинг 6.12. Функция paintEvent

```
/** Производит перерисовку окна */
void DIBView::paintEvent( QPaintEvent *)
{
 int    i, j, k;

 QPainter pnt( this);
 QImage  greyPal = QImage( size(), 32);

 for(i=0; i < height(); i++)
 {
  QRgb* line = (QRgb*) greyPal.scanLine(i);

  for(j=0, k=i; j < width(); j++, k++)
   line[j] = qRgb(k, k, k);
 }

 pnt.drawImage(0, 0, greyPal);
}
```

5. В начале файла dibview.cpp после строки `#include <qpainter.h>` вставьте строку

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

6. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Приложение будет откомпилировано и появится его главное окно, изображенное на рис. 6.7.
7. Измените размер окна. Изображение в нем сохранится.
8. Закройте приложение.

Для работы с аппаратно-независимыми битовыми образами в библиотеке Qt служат объекты класса `QImage`. Основным параметром аппаратно-независимого битового образа является используемая в нем глубина цвета, т. е. число бит, применяемое для кодирования цвета одного элемента изображения. Этот параметр, задаваемый в аргументе `depth` конструкторов класса `QImage`, может принимать значения 1, 8, 16 и 32. В рассматриваемом примере этот параметр имеет значение 32, т. е. используется полная 24-разрядная палитра, в которой каждому из трех основных цветов соответствует свое байтовое значение интенсивности.

Операционная система Linux используется на различных компьютерах, отличающихся способом хранения в них информации. Так, компьютеры, построенные на процессорах SPARC и Motorola, первыми располагают старшие байты, а компьютеры, построенные на процессорах Intel и Alpha, —

младшие байты. Для задания того, как должны интерпретироваться данные в буфере, используются значения перечислимого типа `QImage::Endian`, также указываемые в большинстве конструкторов класса `QImage`. 16-разрядное кодирование цвета является для этого класса не совсем законным и для него отсутствует автоматическая обработка порядка следования байтов.

Поскольку создаваемый нами аппаратно-независимый битовый образ должен заполнять всю рабочую область окна, он создается в функции `DIBView::paintEvent` и для определения его размера вызывается функция `QWidget::size`. После своего создания объект класса `QImage` заполняется информацией. Эта операция производится по строкам изображения.

Для получения указателя на массив, содержащий требуемую строку изображения, используется функция `QImage::scanLine`, возвращающая указатель на переменную типа `uchar`. Поскольку этот тип значений нас не интересует, он сохраняется как указатель на переменную типа `QRgb`, как указатель на целое число без знака. Так как проверка типов в Linux намного слабее, чем в Windows (что, впрочем, и к лучшему), полученный нами указатель рассматривается как массив, для доступа к которому используется соответствующий синтаксис. Элементами массива являются значения цвета, все составляющие которого имеют одинаковую интенсивность, равную сумме координат точки вывода. Для формирования значения цвета в точке применяется функция `QColor::qRgb`. Хотя аргументы этой функции имеют тип `int`, у них используется только младший байт, что позволило исключить излишние преобразования типов ее аргументов.

После заполнения буфера изображения в объекте класса `QImage`, он выводится на экран вызовом функции `QPainter::drawImage`.

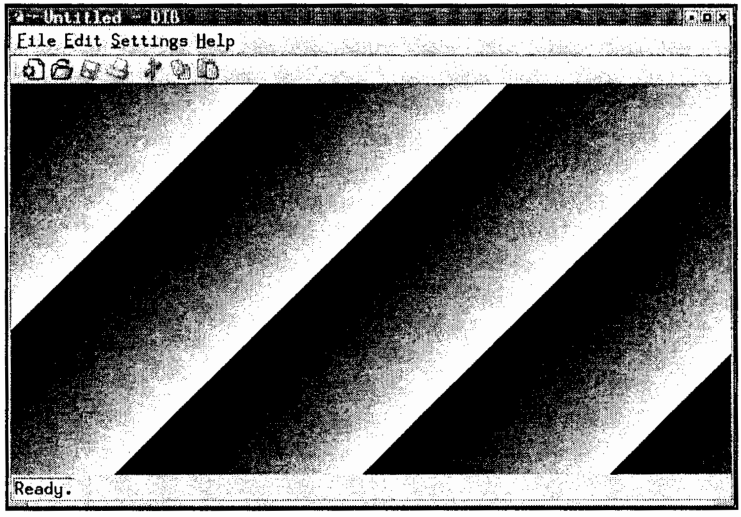
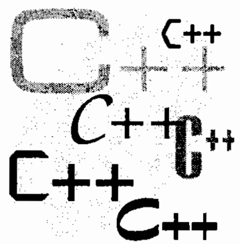


Рис. 6.7. Окно приложения DIB

ГЛАВА 7



Работа с файлами документов

В предыдущей главе уже рассматривался вопрос о том, как сохранить изображение в окне при изменении его размера. Однако предложенный в этой главе метод сохранения информации действует только до тех пор, пока приложение не закрыто. Что же делать в том случае, если нужно восстановить изображение после повторного запуска приложения? Казалось бы, странный вопрос: конечно же использовать файл. Все хорошо, но мы пока еще не знаем, как это сделать. Ответу на поставленный вопрос и посвящена данная глава.

В отличие от реализации концепции Документ/Представление в Windows, когда для чтения и сохранения документа выделяется специальная функция `Serialize`, которой в качестве аргумента передается ссылка на объект специального класса `CArchive`, в библиотеке Qt те же действия выполняются различными функциями, расположенными в разных классах.

Сохранение и восстановление информации в приложении

Для демонстрации того, какие изменения нужно внести в приложение для реализации в нем возможности сохранения документов в файлах и восстановления их из файлов, реализуем эту возможность в созданном в предыдущей главе приложении **Text**.

Чтобы внести в приложение необходимые изменения:

1. Откройте приложение **Text**.
2. В окне иерархических списков раскройте вкладку **Classes** (Классы), раскройте подкаталог **TextDoc** каталога **Classes** и щелкните левой кнопкой мыши по имени функции `newDocument()`. Откроется окно редактирования файла textdoc.cpp и текстовый курсор будет помещен в текст выбранной функции.

3. Измените функции `TextDoc::openDocument` и `TextDoc::saveDocument` в соответствии с текстом листинга 7.1.

Листинг 7.1. Создание, открытие и сохранение документа в файле

```
bool TextDoc::newDocument()
{
 /////////////////////////////////////////////////
 // ЧТО ДЕЛАТЬ: Поместите сюда инициализацию документа
 /////////////////////////////////////////////////
 count = 0;
 modified=false;
 doc_url.setFileName(i18n("Untitled"));

 slotUpdateAllViews( NULL);

 return true;
}

bool TextDoc::openDocument(const KURL& url, const char *format /*=0*/)
{
 QString tmpfile;
 KIO::NetAccess::download( url, tmpfile);
 /////////////////////////////////////////////////
 // ЧТО ДЕЛАТЬ: Поместите сюда функции открытия документа
 /////////////////////////////////////////////////

 QFile fileTemp( tmpfile);

 fileTemp.open( IO_ReadOnly);

 fileTemp.readBlock( (char*)(&count), sizeof( int));

 for(int i=0; i < count; i++)
  fileTemp.readBlock( (char*)(&pointArray[i]), sizeof( QPoint));

 KIO::NetAccess::removeTempFile( tmpfile);

 slotUpdateAllViews( NULL);

 modified=false;
 return true;
}

bool TextDoc::saveDocument(const KURL& url, const char *format /*=0*/)
{
 /////////////////////////////////////////////////
 // ЧТО ДЕЛАТЬ: Поместите сюда функции сохранения документа
 /////////////////////////////////////////////////
```

```
 QFile fileTemp( url.path());

 fileTemp.open( IO_WriteOnly);

 fileTemp.writeBlock( (char*)(&count), sizeof( int));

 for(int i=0; i < count; i++)
  fileTemp.writeBlock( (char*)(&pointArray[i]), sizeof( QPoint));

 modified=false;
 return true;
}
```

4. Откройте окно редактирования файла textview.cpp и измените в нем функцию `TextView::mousePressEvent` в соответствии с текстом листинга 7.2.

Листинг 7.2. Функция mousePressEvent

```
/** Обрабатывает нажатие кнопки мыши */
void TextView::mousePressEvent( QMouseEvent * e)
{
 TextDoc* doc = getDocument();

 if( e->button() & LeftButton)
 {
  QPainter pnt( this);
  QString  strTemp;

  strTemp = tr("[%1,%2]").arg(e->x()).arg(e->y());

  pnt.drawText( e->pos(), strTemp);

  if( doc->count < 128)
  {
   doc->pointArray[doc->count] = e->pos();
   doc->count++;
   doc->setModified( true);
  }
 }
}
```

5. Выберите команду меню **Debug | Start** (Отладка | Пуск) или нажмите кнопку **Debug** в панели инструментов. Приложение будет откомпилировано и появится его главное окно.
6. Щелкните левой кнопкой мыши в рабочей области окна. На месте щелчка будут выведены текущие координаты указателя мыши.

7. Повторите пункт 6 несколько раз и выберите команду меню **File | Quit** (Файл | Закрыть), нажмите комбинацию клавиш <Ctrl>+<Q> или кнопку **Закрыть** главного окна приложения. Появится диалоговое окно **Warning** (Предупреждение), изображенное на рис. 7.1, сообщающее о том, что в текущий документ были внесены изменения и запрашивающее согласие пользователя на их сохранение.

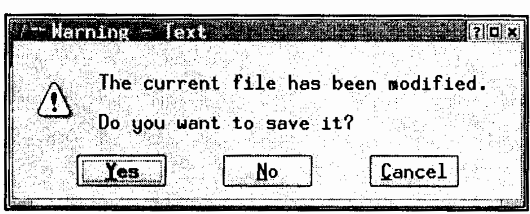


Рис. 7.1. Диалоговое окно **Warning**

8. Нажмите кнопку **Yes** (Да). (Поскольку по умолчанию эта кнопка является активной, достаточно просто нажать клавишу <Enter>.) Появится диалоговое окно **Save as** (Сохранить как), изображенное на рис. 7.2.

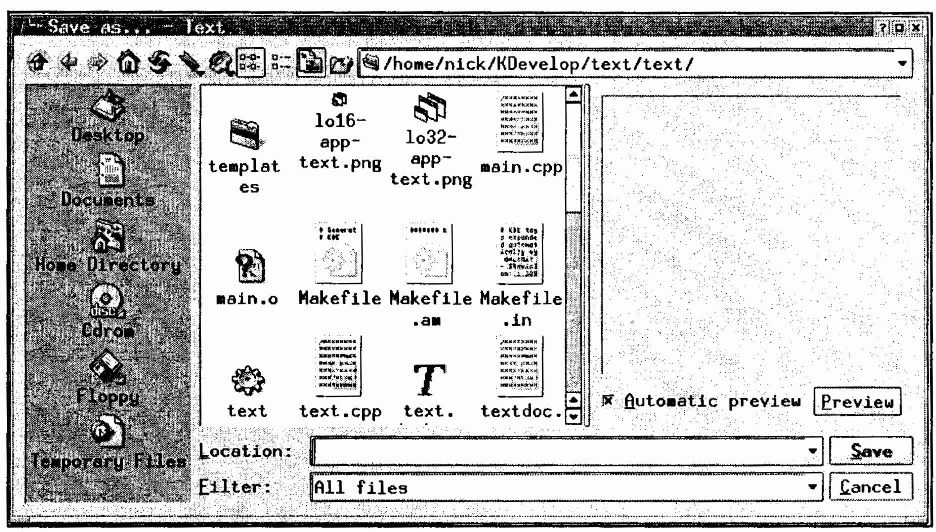


Рис. 7.2. Диалоговое окно **Save as**

9. В текстовое поле **Location** (Путь) введите имя файла first.pct и нажмите кнопку **OK**. Диалоговое окно **Save as** и приложение закроются.
10. Снова запустите приложение **Text** на исполнение.
11. Выберите команду меню **File | Open** (Файл | Открыть), нажмите комбинацию клавиш <Ctrl>+<O> или кнопку **Open an existing document** (От-

крытие существующего документа) в панели инструментов. Появится диалоговое окно **Open File** (Открытие файла), изображенное на рис. 7.3.

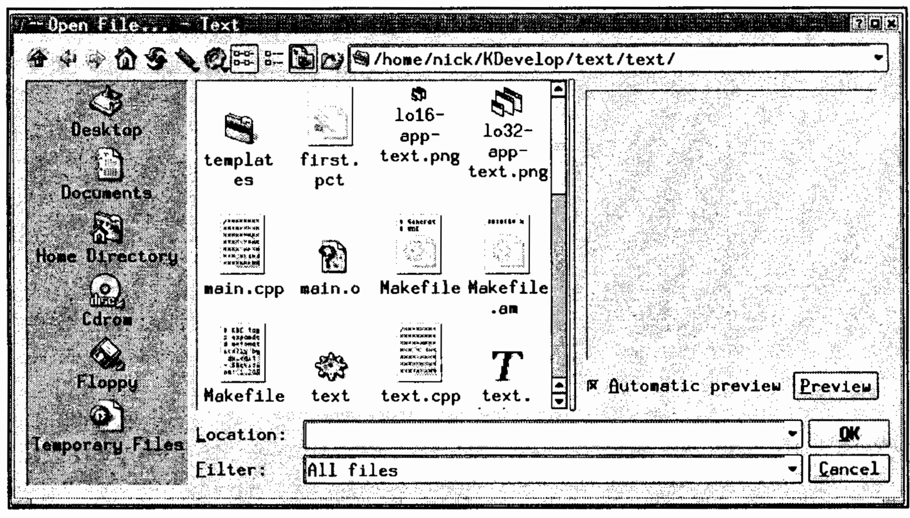


Рис. 7.3. Диалоговое окно **Open File**

12. В окне списка выделите имя файла first.pct, в котором было сохранено изображение в окне, и нажмите кнопку **OK**. Информация в окне восстановится, как это показано на рис. 7.4.

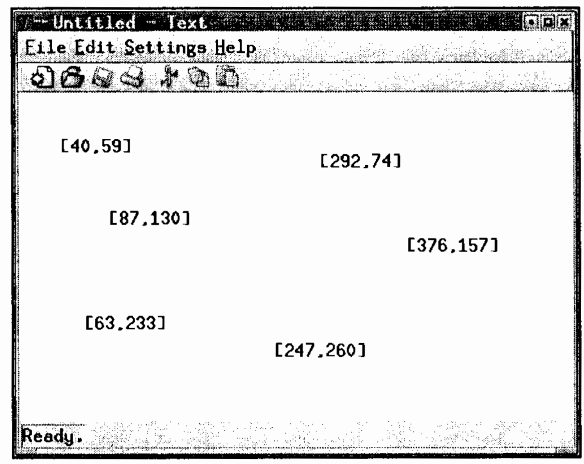


Рис. 7.4. Восстановленная информация в окне приложения

Описание функций мы начнем не в порядке их появления в тексте, а в порядке их использования в приложении, поскольку прежде, чем считать ин-

формацию из файла, нужно эту информацию в него записать. Кроме того, функция сохранения информации в файле несколько проще функции записи в него информации.

В качестве аргумента, определяющего файл, в который следует поместить информацию, функции `TextDoc::saveDocument` передается ссылка на объект класса `KURL`. Уже из названия этого класса можно сделать вывод, что приложения, создаваемые в среде KDevelop, изначально предназначены для работы в Интернете и работа с файлами, расположенными на локальном компьютере, рассматривается в них только как частный случай, для работы с которым не требуется создавать специальных средств.

Для работы с файлами в библиотеке Qt создан специальный класс `QFile`, конструктору его в качестве аргумента передается путь к файлу, с которым этот класс будет работать. Если файл задается каким-либо другим способом, для создания объекта класса должен использоваться конструктор без аргументов. Этот аргумент имеет тип ссылки на объект класса `QString`, а аргумент функции `saveDocument`, в котором передается информация об открываемом файле, как уже говорилось выше, имеет тип ссылки на объект класса `KURL`. Для преобразования типов используется функция `KURL::path`.

Для работы с файлом созданный объект класса `QFile` должен открыть его функцией `QFile::open`. Если файл уже был определен своим путем в конструкторе класса, то в функции `open` определяется только режим открытия файла (в данном случае файл открывается только для записи и в аргументе функции передается значение `IO_WriteOnly`). Если же файл задается каким-либо другим способом, и для создания объекта был использован пустой конструктор, помимо режима открытия файла, в функции `open` необходимо определить и сам файл (например, своим дескриптором).

Прежде всего, в открытый файл записывается размер помещаемого в него массива точек. Для этого используется функция `QFile::writeBlock`, первый аргумент которой содержит указатель на записываемый в файл текстовый буфер, а второй — размер этого буфера в байтах. После этого в цикле в файл последовательно записываются все элементы этого массива.

Примечание

Операционная система Linux, как, впрочем, и Windows, в основном ориентирована на работу с текстовой информацией. Поэтому в классе `QFile` определено очень много функций для чтения и записи текста и единственная функция, ориентированная на работу с двоичной информацией, все равно в качестве аргумента имеет указатель на текстовый буфер, что приводит к необходимости использования при работе с ней преобразования типов.

Внимание!

По утверждениям разработчиков библиотеки Qt, в том случае, если файл открыт одновременно для чтения и записи, после завершения операции записи

информации в файл с использованием функции `QFile::writeBlock` необходимо вызвать функцию `QFile::flush`, переписывающую всю информацию из буфера на диск. В противном случае при последующем чтении не только может быть считана неверная информация, но, что еще хуже, та же самая информация будет записана в файл.

Поскольку вся содержащаяся в окне информация сохранена, флаг наличия в документе изменений сбрасывается. После этого функция возвращает значение `true`, свидетельствующее об успешном завершении операции.

Примечание

В рассматриваемой функции мы явным образом открыли файл функцией `QFile::open`, но не вызывали функцию `QFile::close`, закрывающую этот файл. Дело в том, что эта функция по умолчанию вызывается деструктором класса `QFile`.

Функция `TextDoc::openDocument` в еще большей степени, чем функция `TextDoc::saveDocument` испытывает на себе влияние ориентации приложений KDevelop на работу в Интернете, поскольку в ней предполагается работа с временным файлом, в который помещается извлеченная из Интернета информация. Для этого в данной функции создается объект класса `QString`, в который функция `KIO::NetAccess::download` помещает путь к созданному ею временному файлу. Если в качестве первого аргумента данной функции был передан не URL, а путь к файлу, то никакой загрузки информации не производится, а во втором аргументе просто возвращается этот путь.

Полученный путь к файлу передается в качестве аргумента конструктору объекта класса `QFile`. Созданный объект открывает этот файл только для чтения и с использованием функции `QFile::readBlock`, первый аргумент которой содержит указатель на текстовый буфер, а второй — число байт, которые нужно считать в этот буфер, считывает число точек, которые нужно вывести на экран.

После этого в цикле заполняется массив точек и вызывается функция `KIO::NetAccess::removeTempFile`, удаляющая временный файл, созданный функцией `download` для хранения в нем информации из Интернета. Что происходит в том случае, если файл не является временным, разработчики KDevelop не поясняют, но, по крайней мере, он не уничтожается.

Затем вызывается функция `TextDoc::slotUpdateAllViews` с нулевым аргументом, перерисовывающая рабочую область окна. Необходимость в вызове данной функции связана с тем, что в противном случае информация в окне будет восстановлена только частично.

Поскольку информация в окно только что загружена, флаг наличия в документе изменений сбрасывается. После этого функция возвращает значение `true`, свидетельствующее об успешном завершении операции.

В функцию `TextDoc::newDocument` было внесено только одно изменение — в нее был добавлен вызов функции `slotUpdateAllViews`, очищающей рабочую область окна.

Настройка диалоговых окон

В рассмотренном нами приложении в окнах списка диалоговых окон **Save as** и **Open File** выводятся все файлы, содержащиеся в текущем каталоге. Во многих случаях пользователя интересует только один конкретный тип файла и ему не нужно выводить в окне списка файлы, имеющие другое расширение. Исправим этот недостаток нашего приложения.

Чтобы внести необходимые коррективы в приложение:

1. Откройте приложение **Text**.
2. В окне иерархических списков раскройте вкладку **Classes**, раскройте подкаталог **TextApp** каталога **Classes** и щелкните левой кнопкой мыши по имени функции `slotFileOpen()`. Откроется окно редактирования файла text.cpp и текстовый курсор будет помещен в текст выбранной функции.
3. Измените функцию `TextApp::slotFileOpen` в соответствии с текстом листинга 7.3.

Листинг 7.3. Функция slotFileOpen

```
void TextApp::slotFileOpen()
{
 slotStatusMsg(i18n("Opening file..."));

 if(!doc->saveModified())
 {
   // В процессе сохранения документа возникли ошибки

 }
 else
 {
  KURL url=KFileDialog::getOpenURL(QString::null,
    i18n("*.pct |Picture files \n * |All files"), this,
                        i18n("Open File..."));

  if(!url.isEmpty())
  {
   doc->openDocument(url);
   setCaption(url.fileName(), false);
   fileOpenRecent->addURL( url);
  }
 }
```

```
 slotStatusMsg(i18n("Ready."));
}
```

4. В подкаталоге **TextApp** каталога **Classes** щелкните левой кнопкой мыши по имени функции `slotFileSaveAs()`. Текстовый курсор переместится в текст выбранной функции.
5. Измените функцию `TextApp::slotFileSaveAs` в соответствии с текстом листинга 7.4.

Листинг 7.4. Функция `slotFileSaveAs`

```
void TextApp::slotFileSaveAs()
{
 slotStatusMsg(i18n("Saving file with a new filename..."));

 KURL url=KFileDialog::getSaveURL(QDir::currentDirPath(),
    i18n("*.pct |Picture files \n * |All files"), this,
    i18n("Save as..."));

 if(!url.isEmpty())
 {
  doc->saveDocument(url);
  fileOpenRecent->addURL(url);
  setCaption(url.fileName(),doc->isModified());
 }

 slotStatusMsg(i18n("Ready."));
}
```

6. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Приложение будет откомпилировано и появится его главное окно.
7. Выберите команду меню **File | Open**, нажмите комбинацию клавиш <Ctrl>+<O> или кнопку **Open an existing document** в панели инструментов. Появится диалоговое окно **Open File**, изображенное на рис. 7.5.
8. В окне списка выделите имя файла first.pct, в котором было сохранено содержимое окна, и нажмите кнопку **OK**. Информация в окне восстановится.
9. Щелкните левой кнопкой мыши в рабочей области окна. На месте щелчка будут выведены текущие координаты указателя мыши.
10. Выберите команду меню **File | New** (Файл | Создать), нажмите комбинацию клавиш <Ctrl>+<N> или кнопку **Create a new document** в панели инструментов. Появится диалоговое окно **Warning**, сообщающее о том, что в текущий документ были внесены изменения.

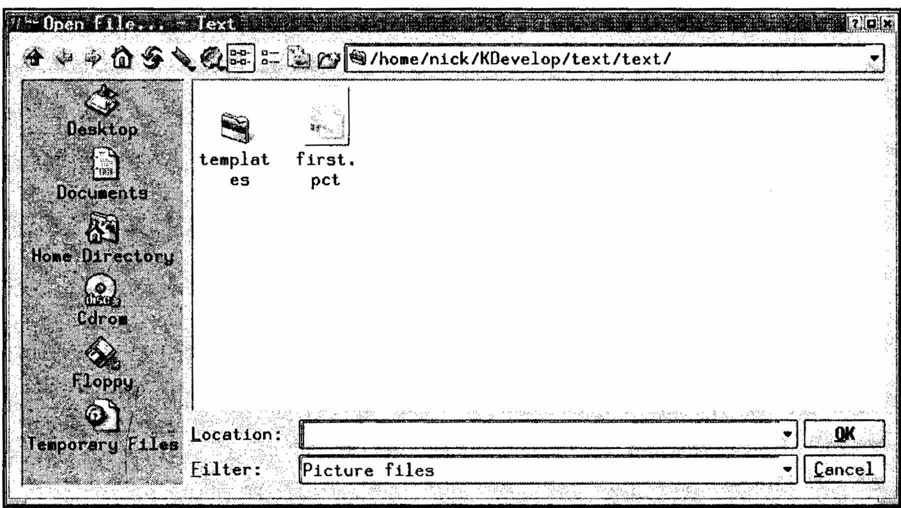


Рис. 7.5. Диалоговое окно **Open File**

11. Нажмите кнопку **Yes**. Появится диалоговое окно **Save as**, изображенное на рис. 7.6.
12. Нажмите кнопку **Cancel**. Диалоговое окно закроется.

Из сравнения рис. 7.3 и 7.5 и рис. 7.2 и 7.6 видно, что поставленная нами задача выполнена. Разберем, как это было достигнуто, и попутно рассмотрим функции класса приложения, ответственные за чтение и сохранение файла документа.

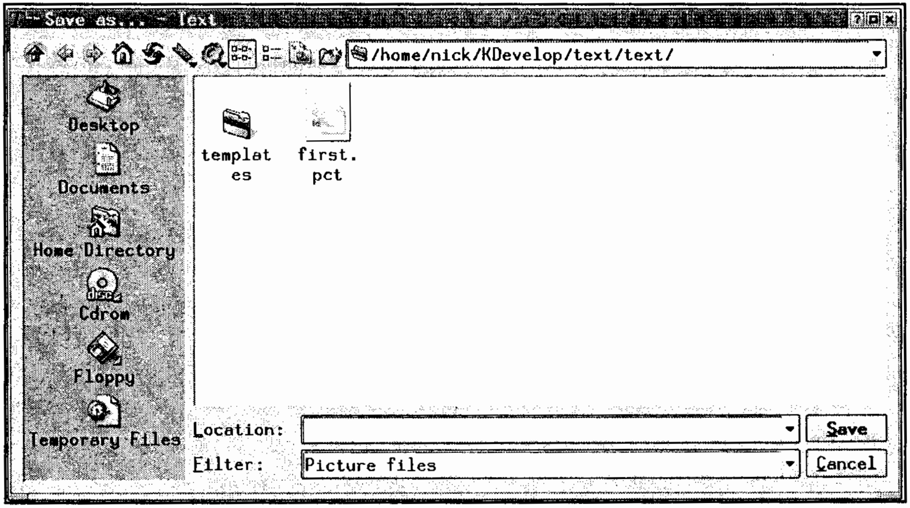


Рис. 7.6. Диалоговое окно **Save as**

Функция TextApp::slotFileOpen вызывается приложением при выборе пользователем команды меню **File | Open** или выполнении им других действий, приводящих к тому же результату.

Прежде всего, в данной функции вызывается функция TextApp::slotStatusMsg, выводящая в строку состояния сообщение о производимой операции. После этого вызывается перегруженная функция класса документа saveModified, сохраняющая текущий документ в его файле. Если эта функция завершилась с ошибкой, разработчику предлагается предпринять необходимые действия по устранению последствий ошибки. По умолчанию никакие действия не предпринимаются.

Если же сохранение текущего документа прошло успешно, вызовом функции KFileDialog::getOpenURL на экран выводится диалоговое окно **Open File**. Рассмотрим эту функцию подробнее, поскольку именно в нее и были внесены изменения. Первый аргумент данной функции содержит имя каталога, содержимое которого будет отображено в окне списка. В нашем примере в этом аргументе передается значение QString::null, означающее, что поиск файла будет начинаться с текущего каталога или же с последнего каталога, в котором в данном приложении были прочитаны или записаны файлы. Второй аргумент данной функции содержит строку фильтров, определяющую тип файлов, выводимых в окно списка создаваемого диалогового окна. Одиночный фильтр состоит из собственно фильтра и его описания, разделенных вертикальной чертой. Отдельные фильтры в строке разделяются символом перевода строки (\n). Описания фильтров будут располагаться в раскрывающемся списке **Filter** создаваемого диалогового окна. Третий аргумент функции getOpenURL содержит указатель на объект родительского окна создаваемого диалогового окна, а четвертый — его заголовок.

Функция getOpenURL возвращает выбранный пользователем URL. Если пользователь сделал свой выбор и строка url не пуста, она передается в качестве аргумента рассмотренной выше перегруженной функции класса документа openDocument. Имя файла открываемого документа, возвращаемое функцией KURL::fileName, передается в качестве аргумента функции QWidget::setCaption для вывода в качестве заголовка главного окна приложения. Кроме того, полученный url вызовом функции KRecentFilesAction::addURL помещается в список последних открытых файлов меню **File** приложения.

В завершение работы функции slotFileOpen вызовом функции slotStatusMsg восстанавливается выводимое по умолчанию сообщение.

Функция TextApp::slotFileSaveAs вызывается приложением при выборе пользователем команды меню **File | Save As** или выполнении им других действий, приводящих к тому же результату.

Прежде всего, в данной функции вызывается функция slotStatusMsg, выводящая в строку состояния сообщение о производимой операции. После этого вызовом функции KFileDialog::getSaveURL на экран выводится диалого-

вое окно **Save as**. Аргументы этой функции полностью аналогичны аргументам функции `getOpenURL` того же класса. Однако в данном случае текущий каталог указан в явном виде, а для его получения использована функция `QDir::currentDirPath`.

Возвращаемый функцией `getSaveURL` выбранный пользователем url так же, как и в случае с открытием файла, проверяется на корректность. Если пользователь сделал свой выбор и строка url не пуста, она передается в качестве аргумента рассмотренной выше перегруженной функции класса документа `saveDocument`. Поскольку при сохранении файла пользователь мог его переименовать, url сохраняемого документа помещается в список последних открытых файлов меню **File** приложения, а имя файла выводится в заголовок окна функцией `QWidget::setCaption`.

В завершение работы функции `slotFileSaveAs` вызовом функции `slotStatusMsg` восстанавливается выводимое по умолчанию сообщение.

Внесение изменений в меню

До сих пор для сохранения информации в окне мы закрывали окно или создавали новый документ. Почему же мы не использовали для этого соответствующие команды меню? Для ответа на этот вопрос достаточно открыть меню **File**: команды сохранения файла в нем недоступны.

Чтобы исправить этот недостаток:

1. Откройте приложение **Text**.
2. В окне иерархических списков раскройте вкладку **Classes**, щелкните правой кнопкой мыши по имени класса `TexApp` и выберите в появившемся контекстном меню команду **Add member function** (Добавить переменную члена класса). Появится диалоговое окно **Class Properties** (Свойства класса), раскрытое на вкладке **Methods** (Функции).
3. В текстовое поле **Type** (Тип) введите тип возвращаемого значения `void`, в текстовое поле **Declaration** (Сигнатура) — сигнатуру функции `saveUpdate(bool)`, в текстовое поле **Documentation** (Документация) — комментарий `Updates Save command` (Обновляет состояние команды сохранения документа) и нажмите кнопку **Apply**. В класс будет добавлена новая функция.
4. Повторите пункты 2 и 3 для добавления в класс функции `saveAsUpdate(bool)`, снабдив ее комментарием `Update Save As command` (Обновляет состояние команды сохранения документа под другим именем).
5. В открывшемся после добавления в класс новых функций окне редактирования файла text.cpp измените эти функции в соответствии с текстом листинга 7.5.

8 Зак. 113

Листинг 7.5. Функции обновления меню

```
/** Обновляет состояние команды сохранения документа */
void TextApp::saveUpdate( bool b)
{
 fileSave->setEnabled(b);
}

/** Обновляет состояние команды сохранения документа под другим именем */
void TextApp::saveAsUpdate( bool b)
{
 fileSaveAs->setEnabled(b);
}
```

6. В том же файле измените функции `TextApp::slotFileNew` и `TextApp::slotFileOpen` в соответствии с текстом листинга 7.6.

Листинг 7.6. Функции slotFileNew и slotFileOpen

```
void TextApp::slotFileNew()
{
 slotStatusMsg(i18n("Creating new document..."));

 if(!doc->saveModified())
 {
   // В процессе сохранения документа возникли ошибки

 }
 else
 {
  doc->newDocument();
  setCaption(doc->URL().fileName(), false);
  saveUpdate( false);
  saveAsUpdate( false);
 }

 slotStatusMsg(i18n("Ready."));
}

void TextApp::slotFileOpen()
{
 slotStatusMsg(i18n("Opening file..."));

 if(!doc->saveModified())
 {
   // В процессе сохранения документа возникли ошибки

 }
```

```
  else
  {
   KURL url=KFileDialog::getOpenURL(QString::null,
     i18n("*.pct |Picture files \n * |All files"), this,
     i18n("Open File..."));

   if(!url.isEmpty())
   {
    doc->openDocument(url);
    setCaption(url.fileName(), false);
    fileOpenRecent->addURL( url);
    saveUpdate( false);
    saveAsUpdate( true);
   }
  }

  slotStatusMsg(i18n("Ready."));
}
```

7. Откройте окно редактирования файла textview.cpp и измените в нем функцию `TextView::mousePressEvent` в соответствии с текстом листинга 7.7.

Листинг 7.7. Функция mousePressEvent

```
/** Обрабатывает нажатие кнопки мыши */
void TextView::mousePressEvent( QMouseEvent * e)
{
 TextDoc* doc = getDocument();

 if( e->button() & LeftButton)
 {
  QPainter pnt( this);
  QString  strTemp;
  TextApp* theApp = (TextApp *) parentWidget();

  strTemp = tr("[%1,%2]").arg(e->x()).arg(e->y());

  pnt.drawText( e->pos(), strTemp);

  if( doc->count < 128)
  {
   doc->pointArray[doc->count] = e->pos();
   doc->count++;
   doc->setModified( true);
```

```
      theApp->saveUpdate( true);
      theApp->saveAsUpdate( true);
    }
  }
}
```

8. В начале файла textview.cpp после строки `#include <qpainter.h>` вставьте строку

```
#include <klocale.h>
```

9. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Приложение будет откомпилировано и появится его главное окно.
10. Откройте меню **File**. Его команды **Save** и **Save As** будут недоступны.
11. Выберите команду меню **File | Open** и откройте новый документ.
12. Снова откройте меню **File**. Его команда **Save As** стала доступной, а команда **Save** осталась недоступной.
13. Щелкните левой кнопкой мыши в рабочей области окна и снова откройте меню **File**. Команда **Save** стала доступной.
14. Выберите команду меню **File | New** и откажитесь от сохранения изменений.
15. Откройте меню **File**. Его команды **Save** и **Save As** снова будут недоступны.

Для управления видимостью команд меню нам пришлось добавить в класс приложения две открытые функции, обеспечивающие доступ к закрытым переменным класса. В каждой из этих функций для объекта класса `KAction`, используемого для связи приложения с соответствующими элементами пользовательского интерфейса (командой меню, кнопкой панели инструментов и комбинацией клавиш, используемых для инициализации одной и той же операции), вызывается функция `KAction::setEnabled`, делающая доступной или недоступной ту или иную операцию.

После создания функций управления состоянием элементов пользовательского интерфейса нужно расположить их вызовы в соответствии с логикой приложения. Команда меню **File | Save As** должна становиться доступной, как только в документе появляется какая-либо информация, которую можно сохранить в файле, а команда меню **File | Save** должна становиться доступной только в том случае, если в документ была внесена новая информация.

Все операции, которые могут привести к изменению доступности команд меню, выполняются в функциях `TextApp::slotFileNew`, `TextApp::slotFileOpen` и `TextView::mousePressEvent`. При создании нового документа обе команды меню делаются недоступными, поскольку этот документ пуст. При откры-

тии нового документа делается доступной только команда меню **File | Save As**, поскольку в документе появляется информация, но она уже сохранена в своем файле. При нажатии пользователем левой кнопки мыши команды меню **File | Save** и **File | Save As** становятся доступными, поскольку в документ записывается новая информация. Однако если выведенная в окне информация не была до этого сохранена в файле, то при выборе команды меню **File | Save** все равно будет вызываться функция обработки команды **File | Save As**.

Установка рабочего каталога

По умолчанию при запуске приложения из среды разработки KDevelop его рабочим каталогом считается рабочий каталог проекта, а при его самостоятельном запуске — домашний каталог пользователя, что, в большинстве случаев, очень неудобно. Изменим наше приложение таким образом, чтобы рабочим каталогом становился последний каталог, в котором было произведено чтение или запись файла, независимо от того, закрывалось ли после этого приложение или нет.

Чтобы внести в приложение соответствующие изменения:

1. Откройте приложение **Text**.
2. В окне иерархических списков раскройте вкладку **Classes**, раскройте подкаталог **TextApp** каталога **Classes** и щелкните левой кнопкой мыши по имени функции `slotFileOpen()`. Откроется окно редактирования файла text.cpp и текстовый курсор будет помещен в текст выбранной функции.
3. Измените выбранную функцию в соответствии с текстом листинга 7.8.

Листинг 7.8. Функция `slotFileOpen`

```
void TextApp::slotFileOpen()
{
 slotStatusMsg(i18n("Opening file..."));

 if(!doc->saveModified())
 {
   // В процессе сохранения документа возникли ошибки

 }
 else
 {
  config->setGroup("General Options");

  QString szPath = config->readPathEntry("Current Path",
                                         QString::null);
```

```
  KURL url=KFileDialog::getOpenURL( szPath, i18n("*.pct |Picture files
            \n * |All files"), this, i18n("Open File..."));

  if(!url.isEmpty())
  {
   doc->openDocument(url);
   setCaption(url.fileName(), false);
   fileOpenRecent->addURL( url);
   saveUpdate( false);
   saveAsUpdate( true);

   config->writeEntry("Current Path", url.path());
  }
 }

 slotStatusMsg(i18n("Ready."));
}
```

4. В том же файле измените функцию `TextApp::slotFileSaveAs` в соответствии с текстом листинга 7.9.

Листинг 7.9. Функция `slotFileSaveAs`

```
void TextApp::slotFileSaveAs()
{
 slotStatusMsg(i18n("Saving file with a new filename..."));

 config->setGroup("General Options");

 QString szPath = config->readPathEntry("Current Path",
                        QDir::currentDirPath());

 KURL url=KFileDialog::getOpenURL( szPath,
  i18n("*.pct |Picture files \n * |All files"), this,
  i18n("Save as..."));

 if(!url.isEmpty())
 {
  doc->saveDocument(url);
  fileOpenRecent->addURL(url);
  setCaption(url.fileName(),doc->isModified());

  config->writeEntry("Current Path", url.path());
 }

 slotStatusMsg(i18n("Ready."));
}
```

5. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Приложение будет откомпилировано и появится его главное окно.
6. Убедитесь, что рабочий каталог приложения сохраняется даже после его закрытия.

Для сохранения своей текущей конфигурации приложения KDevelop используют объект класса `KConfig`, автоматически включаемый мастером создания приложения в класс приложения. По умолчанию этот объект применяется для сохранения информации о геометрии окон приложения, о наличии в его окне стандартной панели инструментов и строки состояния, а также о положении стандартной панели инструментов в окне. Однако этот объект может служить и для сохранения других параметров приложения.

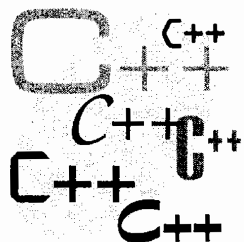
Поскольку все параметры приложения по умолчанию сохранялись в группе **General Options** файла конфигурации KDE, то нам нет никакого смысла помещать сохраняемую нами информацию в другую группу. Поэтому перед чтением информации из объекта вызывается функция `KConfigBase::setGroup`, устанавливающая группу, в которой будет производиться поиск указанного ключа. После этого вызывается функция `KConfigBase::readPathEntry`, возвращающая текстовое значение, содержащееся по указанному в ее первом аргументе ключу, и интерпретирующая его как путь к файлу. Так как по указанному ключу при вызове данной функции еще может быть не записано никакого значения, во втором аргументе этой функции помещается строка, которая должна быть возвращена в этом случае.

Если при закрытии диалогового окна пользователь указал путь к файлу, этот путь сохраняется в объекте класса `KConfig` вызовом функции `KConfigBase::writeEntry`.

Примечание

Поскольку для получения пути к файлу используется вызов функции `KURL::path`, то в объекте класса `KConfig` сохраняется не только путь к файлу, но и его имя, которое будет выводиться в текстовое поле `Location` диалогового окна. Это помогает указать правильное расширение нового файла при сохранении документа.

ГЛАВА 8



Работа с текстовыми документами

Текстовые документы всегда считались одним из главных, если не основным типом документов, обрабатываемых на компьютере. Эта традиция берет свое начало еще с тех времен, когда компьютеры практически не имели графического интерфейса и не могли обрабатывать другие типы документов.

Хотя в настоящее время большинство приложений является графическими, внимание к текстовым документам сохранилось, и для их обработки создаются специализированные классы, которые и будут рассмотрены в данной главе.

Создание простейшего текстового редактора

Для работы с текстовыми документами в библиотеке Qt используется объект класса `QMultiLineEdit`, обладающий достаточно широкими функциональными возможностями.

Чтобы создать простейший текстовый редактор:

1. Выберите команду меню **Project | New** (Проект | Создать).
2. В иерархическом списке типов создаваемых приложений появившегося окна **ApplicationWizard** (Мастер создания приложений) выделите строку **Qt SDI** (однооконное приложение Qt), расположенную в каталоге Qt, и нажмите кнопку **Next** (Далее).
3. В текстовое поле **Project name** (Имя проекта) введите имя проекта **QtEdit** и нажмите кнопку **Create** (Создать).
4. После завершения работы мастера по созданию приложения нажмите кнопку **Exit** (Выход). Заготовка приложения будет создана.

5. В окне иерархических списков раскройте вкладку **Classes** (Классы) и щелкните левой кнопкой мыши по имени класса `QtEditView`. Откроется окно редактирования файла qteditview.h.
6. В начале этого файла замените строку `#include <qwidget.h>` строкой

   `#include <qmultilineedit.h>`
7. В заголовке класса `QtEditView` замените строку `class QtEditView : public QWidget` строкой

   `class QtEditView : public QMultiLineEdit`
8. Щелкните правой кнопкой мыши в окне редактирования файла qteditview.h и выберите в появившемся контекстном меню команду **Switch Header/Source** (Переключить файлы заголовка и реализации). Откроется окно редактирования файла qteditview.cpp.
9. Измените конструктор класса `QtEditView` в соответствии с текстом листинга 8.1.

Листинг 8.1. Конструктор класса `QtEditView`

```
QtEditView::QtEditView(QWidget *parent, QtEditDoc *doc)
  : QMultiLineEdit(parent)
{
 /** связывает документ с представлением */
 connect(doc, SIGNAL(documentChanged()),
                 this, SLOT(slotDocumentChanged()));
}
```

10. Откройте окно редактирования файла qtedit.cpp и измените в нем приемники команд меню **Edit** в соответствии с текстом листинга 8.2.

Листинг 8.2. Приемники команд меню Edit

```
/* Вырезает выделенный текст в буфер обмена */
void QtEditApp::slotEditCut()
{
 statusBar()->message(tr("Cutting selection..."));

 view-> cut();

 statusBar()->message(tr("Ready."));
}

/* Копирует выделенный текст в буфер обмена */
void QtEditApp::slotEditCopy()
{
 statusBar()->message(tr("Copying selection to clipboard..."));
```

```
 view-> copy();

 statusBar()->message(tr("Ready."));
}

/** Вставляет текст из буфера обмена */
void QtEditApp::slotEditPaste()
{
 statusBar()->message(tr("Inserting clipboard contents..."));

 view-> paste();

 statusBar()->message(tr("Ready."));
}
```

11. Выберите команду меню **Debug | Start** (Отладка | Пуск) или нажмите кнопку **Debug** в панели инструментов. Приложение будет запущено на исполнение.
12. Введите в рабочую область окна какой-либо текст. Приложение примет вид, изображенный на рис. 8.1.

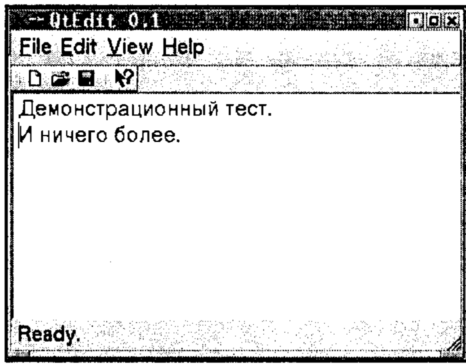


Рис. 8.1. Окно редактора

13. Выделите фрагмент введенного текста и выберите команду меню **Edit | Cut** (Правка | Вырезать) или нажмите комбинацию клавиш <Ctrl>+<X>. Выделенный фрагмент исчезнет.
14. Выберите команду меню **Edit | Paste** (Правка | Вставить) или нажмите комбинацию клавиш <Ctrl>+<V>. Введенный текст появится снова.
15. Закройте приложение.

Как видно из приведенного примера, создание простейшего текстового редактора действительно не представляет особых сложностей. Однако следует заметить, что создание простейшего текстового редактора в Windows еще проще: там уже в заготовке приложения устанавливается нужный базовый

класс, а в приложении заранее осуществлена связь команд меню **Edit** и соответствующих им кнопок панели инструментов с обрабатывающими их методами класса представления. В библиотеке Qt эти операции приходится выполнять разработчику.

Таким образом, чтобы преобразовать заготовку приложения в текстовый редактор, необходимо:

- заменить в заголовке и в конструкторе класса представления его базовый класс `QWidget` классом `QMultiLineEdit`;
- включить в приемники команд меню **Edit**, расположенные в файле реализации класса приложения, вызовы соответствующих приемников класса `QMultiLineEdit`.

Создание более сложного редактора

Для более подробного ознакомления с возможностями класса `QMultiLineEdit` внесем изменения в созданное нами приложение, позволяющие отменять и восстанавливать отмененные действия, выравнивать текст, находить в нем слова и автоматически прокручивать текст для вывода на экран найденного слова, если оно находится за пределами отображаемого фрагмента текста.

Чтобы внести изменения в приложение **QtEdit**:

1. Выберите команду меню **File | New** (Файл | Новый), нажмите комбинацию клавиш <Ctrl>+<N> или кнопку **New** в панели инструментов. Появится диалоговое окно **New File** (Новый файл).
2. В окне списка выделите строку **Qt Designer File (*.ui)**, в текстовое поле **Filename** введите имя файла finddlg (расширение к нему будет добавлено автоматически) и нажмите кнопку **OK**. Появится диалоговое окно **Load decision**, предлагающее загрузить файл в текстовом виде.
3. Нажмите кнопку **No**. Откроется окно приложения Qt Designer.
4. В окне списка диалогового окна **New File**, автоматически открывшегося при запуске приложения, выделите значок **Dialog with Buttons (Bottom)** (Диалоговое окно с кнопками, расположенными снизу) и нажмите кнопку **OK**. В окне приложения **Qt Designer** появится заготовка диалогового окна.
5. В панели **Property Editor/Signal Handlers** (Редактор свойств/Обработчики сигналов) выделите строку **name** (если она уже не была выделена при создании заготовки) и введите в текстовое поле связанного с ней раскрывающегося списка имя диалогового окна **FindDlg**.
6. В той же панели выделите строку **caption** и введите в связанное с ней текстовое поле заголовок диалогового окна **Find**.

7. Выберите команду меню **Tools | Display | TextLabel** (Сервис | Изображения | Статический текст) или нажмите кнопку **Text Label** в панели инструментов **Display** (при этом кнопка должна "утопиться") и щелкните левой кнопкой мыши в левом верхнем углу заготовки диалогового окна. Появится рамка статического текста.
8. В панели **Property Editor/Signal Handlers** выделите строку **text** и введите в связанное с ней текстовое поле строку **Find:** (Найти).
9. Выберите команду меню **Tools | Input | LineEdit** (Сервис | Ввод | Текстовое поле) или нажмите кнопку **Line Edit** в панели инструментов **Input** и щелкните левой кнопкой мыши под только что введенным статическим текстом. На месте щелчка появится текстовое поле.
10. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя текстового поля на **FindLineEdit**.
11. Выделите статический текст и текстовое поле, выберите команду меню **Layout | Lay Out Vertically** (Расположить | Расположить по вертикали), нажмите комбинацию клавиш <Ctrl>+<L> или кнопку **Lay Out Vertically** в панели инструментов **Layout**. Статический текст и текстовое поле будут выровнены по вертикали.
12. Выберите команду меню **Tools | Buttons | CheckBox** (Сервис | Кнопки | Флажок) или нажмите кнопку **Check Box** в панели инструментов **Buttons** и щелкните левой кнопкой мыши под только что введенным текстовым полем. На месте щелчка появится флажок.
13. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя кнопки на **BackwardCheck**.
14. В той же панели выделите строку **text** и введите в связанное с ней текстовое поле строку **Backwards** (Назад).
15. Повторите пункты 12—14, поместив справа от только что введенного флажка новый флажок с именем **CaseCheck** и текстом **Case sensitive** (С учетом регистра).
16. Выделите оба флажка и выберите команду меню **Layout | Lay Out Horizontally** (Расположить | Расположить по горизонтали), нажмите комбинацию клавиш <Ctrl>+<H> или кнопку **Lay Out Horizontally** в панели инструментов **Layout**. Флажки будут выровнены по горизонтали.
17. Выберите команду меню **Layout | Add Spacer** (Расположить | Добавить разделитель) или нажмите кнопку **Spacer** в панели инструментов **Layout**. Кнопка **Spacer** в панели инструментов **Layout** "утопится".
18. Щелкните левой кнопкой мыши между текстовым полем и группой флажков и выберите в появившемся контекстном меню команду **Vertical** (Вертикальный разделитель). На месте щелчка появится вертикальный разделитель.

19. Повторите пункты 17 и 18 для добавления вертикального разделителя между группой флажков и группой кнопок.
20. Щелкните левой кнопкой мыши в свободной области заготовки диалогового окна и выберите команду меню **Layout | Lay Out Vertically**, нажмите комбинацию клавиш <Ctrl>+<L> или кнопку **Lay Out Vertically** в панели инструментов **Layout**. Элементы управления диалогового окна будут выровнены по вертикали.
21. Выберите команду меню **Layout | Adjust Size** (Расположить | Настроить размер), нажмите комбинацию клавиш <Ctrl>+<J> или кнопку **Adjust Size** в панели инструментов **Layout**. Размер диалогового окна будет приведен в соответствие с его содержимым. В результате произведенных действий заготовка диалогового окна примет вид, изображенный на рис. 8.2.

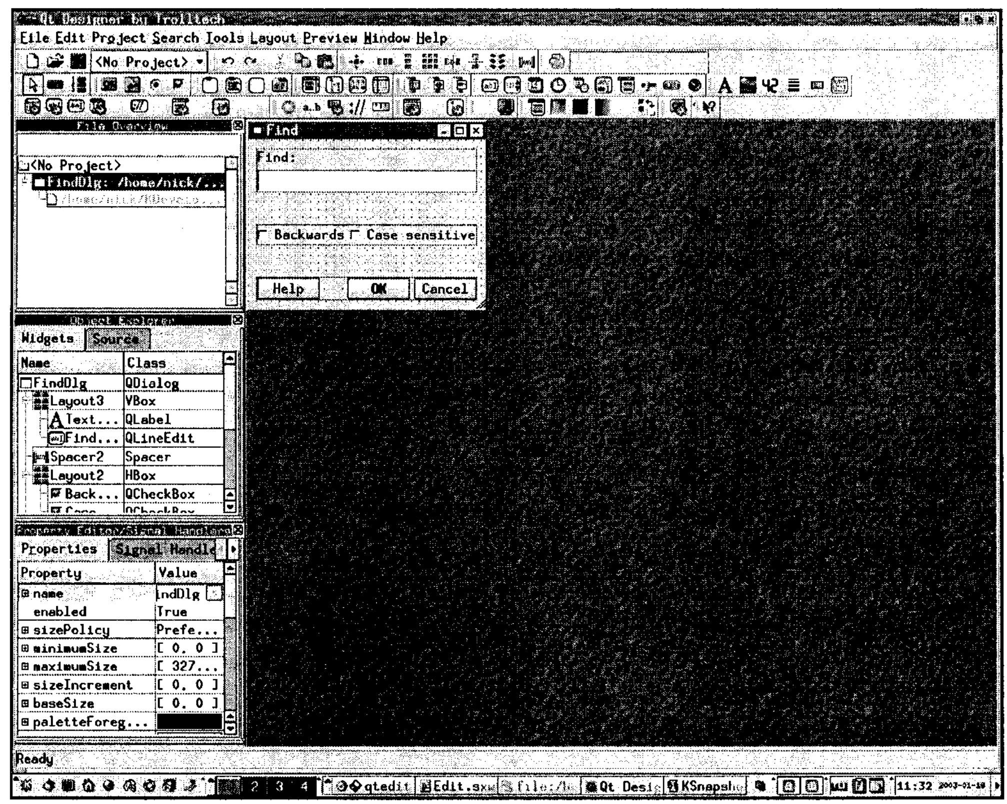


Рис. 8.2. Заготовка диалогового окна

22. Выберите команду меню **Tools | Connect Signal/Slots** (Сервис | Связывание сигналов и приемников), нажмите клавишу <F3> или кнопку

Connect Signal/Slots в панели инструментов **Tools**. Кнопка **Connect Signal/Slots** в панели инструментов **Tools** "утопится".

23. Поместите указатель мыши на кнопку **OK**, нажмите левую кнопку мыши и, не отпуская ее, переместите указатель мыши на свободное пространство заготовки диалогового окна. Отпустите левую кнопку мыши. Появится диалоговое окно **Edit Connections** (Редактирование соединений).
24. Нажмите кнопку **Edit Slots** (Редактирование приемников). Появится диалоговое окно **Edit Slots**.
25. Нажмите кнопку **New Slot** (Новый приемник) и введите в текстовое поле **Slot** (Приемник), расположенное в группе **Slot Properties** (Свойства приемников), сигнатуру приемника `onOK()`.
26. Нажмите кнопку **OK**. В списке приемников класса диалогового окна появится новый приемник.
27. В окне списка **Signals** (Сигналы) выделите сигнатуру сигнала `clicked()`, в окне списка **Slots** (Приемники) — сигнатуру приемника `onOK()` и нажмите кнопку **OK**. В класс диалогового окна будет добавлена информация о новом соединении.
28. Выберите команду меню **File | Save** (Файл | Сохранить), нажмите комбинацию клавиш <Ctrl>+<S> или кнопку **Save** в панели инструментов. Появится диалоговое окно **Save Form** (Сохранить форму).
29. Перейдите в каталог, в котором хранится файл описания ресурсов вашего приложения, выделите его в окне списка и нажмите кнопку **Save**. Появится окно сообщения о том, что этот файл существует и предложением переписать его.
30. Нажмите кнопку **Yes**. Информация о заготовке диалогового окна сохранится в файле описания ресурсов.
31. Закройте приложение Qt Designer.
32. В среде разработки KDevelop выберите команду меню **Build | Make** (Трансляция | Компилировать), нажмите клавишу <F8> или кнопку **Make** в панели инструментов. Приложение будет откомпилировано.
33. Раскройте вкладку **Classes** в окне иерархических списков, щелкните правой кнопкой мыши по каталогу **Classes** и выберите в появившемся контекстном меню команду **New class** (Новый класс). Появится диалоговое окно **Class Generator** (Создание класса).
34. В текстовое поле **Classname** (Имя класса) введите имя класса `FindDlgImpl`, в текстовое поле **Baseclass** (Имя базового класса) — имя базового класса `FindDlg`, в группе **Additional Options** (Дополнительные параметры) установите флажок **generate a QWidget-Childclass** (Создать дочерний класс класса `QWidget`), в текстовое поле **Documentation** (Докумен-

тация) введите комментарий `Implementation of dialog class` (Реализация класса диалогового окна) и нажмите кнопку **OK**. В приложение будет добавлен новый класс и будут открыты окна редактирования его файлов заголовка и реализации.

35. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `FindDlgImpl` и выберите в появившемся контекстном меню команду **Add slot** (Добавить приемник). Появится диалоговое окно **Class Properties** (Свойства класса), раскрытое на вкладке **Slots** (Приемники).

36. В текстовое поле **Declaration** (Сигнатура) введите сигнатуру приемника `onOK()`, в группе **Modifiers** (Модификаторы) установите флажок **Virtual**, в текстовое поле **Documentation** (Документация) введите комментарий `User pressed OK` (Пользователь нажал кнопку OK) и нажмите кнопку **Apply** (Применить). В класс будет добавлен новый приемник.

37. В открывшемся после добавления приемника окне редактирования файла finddlgimpl.cpp измените реализацию класса `FindDlgImpl` в соответствии с текстом листинга 8.3.

Листинг 8.3. Реализация класса `FindDlgImpl`

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

FindDlgImpl::FindDlgImpl(QWidget *parent, const char *name, bool modal)
  : FindDlg(parent,name,modal)
{
}

FindDlgImpl::~FindDlgImpl()
{
}

/** Пользователь нажал кнопку OK */
void FindDlgImpl::onOK()
{
 findText        = FindLineEdit-> text();
 searchBackwards = BackwardCheck-> isChecked();
 caseSensitive    = CaseCheck-> isChecked();
}
```

38. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `FindDlgImpl` и выберите в появившемся контекстном меню команду **Add member variable** (Добавить переменную члена класса). Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes** (Атрибуты).

39. В текстовое поле **Type** введите тип переменной QString, в текстовое поле **Name** — имя переменной findText, в текстовое поле **Documentation** — комментарий Text, that must be found (Искомый текст) и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.
40. Повторите пункты 38 и 39 для добавления в класс FindDlgImpl переменной searchBackwards, имеющей тип bool, снабдив ее комментарием Search direction flag (Флаг направления поиска).
41. Повторите пункты 38 и 39 для добавления в класс FindDlgImpl переменной caseSensitive, имеющей тип bool, снабдив ее комментарием Case sensitivity flag (Флаг учета регистра символов).
42. В открывшемся после добавления переменных окне редактирования файла finddlgimpl.h замените строку FindDlgImpl(QWidget *parent=0, const char *name=0); строкой

    ```
    FindDlgImpl(QWidget *parent=0, const char *name=0, bool modal = TRUE);
    ```

43. Во вкладке **Classes** окна иерархических списков, щелкните правой кнопкой мыши по имени класса QtEditApp и выберите в появившемся контекстном меню команду **Add member variable**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes**.
44. В текстовое поле **Type** введите тип переменной QAction, в текстовое поле **Name** — имя переменной *editUndo, в группе **Access** (Право доступа) установите переключатель **Private** (Закрытый) и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.
45. Повторите пункты 43 и 44 для включения в класс переменной *editRedo.
46. Повторите пункты 43 и 44 для включения в класс переменной *editFind.
47. Повторите пункты 43 и 44 для включения в класс переменной *editFindNext.
48. Повторите пункты 43 и 44 для включения в класс переменной *formatMenu, имеющей тип QPopupMenu, снабдив ее комментарием Format_menu contains all items of the menubar entry "Format" (Содержит все команды меню Формат).
49. Повторите пункты 43 и 44 для включения в класс переменной *formatLeft.
50. Повторите пункты 43 и 44 для включения в класс переменной *formatCenter.
51. Повторите пункты 43 и 44 для включения в класс переменной *formatRight.
52. Повторите пункты 43 и 44 для включения в класс переменной findText, имеющей тип QString.

53. Повторите пункты 43 и 44 для включения в класс переменной `searchBackwards`, имеющей тип `bool`.
54. Повторите пункты 43 и 44 для включения в класс переменной `caseSensitive`, имеющей тип `bool`.
55. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `QtEditApp` и выберите в появившемся контекстном меню команду **Add slot** (Добавить приемник). Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Slots**.
56. В текстовое поле **Declaration** введите сигнатуру приемника `slotEditUndo()`, в текстовое поле **Documentation** — комментарий `Reverts the last editing step` (Отменяет последнюю операцию редактирования) и нажмите кнопку **Apply**. В класс будет добавлен новый приемник.
57. Повторите пункты 55 и 56 для включения в класс приемника `slotEditRedo()`, снабдив его комментарием `Re-executes the last undone step` (Восстанавливает последнюю отмененную операцию).
58. Повторите пункты 55 и 56 для включения в класс приемника `slotEditFind()`, снабдив его комментарием `Searches the text in the document` (Производит поиск текста в документе).
59. Повторите пункты 55 и 56 для включения в класс приемника `slotEditFindNext()`, снабдив его комментарием `Repeats the last search` (Производит поиск следующего вхождения текста).
60. Повторите пункты 55 и 56 для включения в класс приемника `slotFormatLeft()`, снабдив его комментарием `Aligns left` (Выравнивает по левому краю).
61. Повторите пункты 55 и 56 для включения в класс приемника `slotFormatCenter()`, снабдив его комментарием `Aligns center` (Выравнивает по центру).
62. Повторите пункты 55 и 56 для включения в класс приемника `slotFormatRight()`, снабдив его комментарием `Aligns right` (Выравнивает по правому краю).
63. В открывшемся после добавления приемников окне редактирования файла qtedit.cpp после строки `#include "qtedit.h"` вставьте строку

    ```
    #include "finddlgimpl.h"
    ```

64. В том же файле измените реализацию функций `QtEditApp::initActions` и `QtEditApp::initMenuBar` в соответствии с текстом листинга 8.4.

Листинг 8.4. Функции `initActions` и `initMenuBar`

```
/** Инициализирует все объекты класса QActions-приложения */
void QtEditApp::initActions()
```

```
{
 QPixmap openIcon, saveIcon, newIcon;
 newIcon = QPixmap(filenew);
 openIcon = QPixmap(fileopen);
 saveIcon = QPixmap(filesave);

 fileNew = new QAction(tr("New File"), newIcon, tr("&New"),
              QAccel::stringToKey(tr("Ctrl+N")), this);
 fileNew->setStatusTip(tr("Creates a new document"));
 fileNew->setWhatsThis(tr("New File\n\nCreates a new document"));
 connect(fileNew, SIGNAL(activated()), this, SLOT(slotFileNew()));

 fileOpen = new QAction(tr("Open File"), openIcon, tr("&Open..."),
                             0, this);
 fileOpen->setStatusTip(tr("Opens an existing document"));
 fileOpen->setWhatsThis(tr("Open File\n\nOpens an existing document"));
 connect(fileOpen, SIGNAL(activated()), this, SLOT(slotFileOpen()));

 fileSave = new QAction(tr("Save File"), saveIcon, tr("&Save"),
              QAccel::stringToKey(tr("Ctrl+S")), this);
 fileSave->setStatusTip(tr("Saves the actual document"));
 fileSave->setWhatsThis(tr("Save File.\n\nSaves the actual document"));
 connect(fileSave, SIGNAL(activated()), this, SLOT(slotFileSave()));

 fileSaveAs = new QAction(tr("Save File As"), tr("Save &as..."),
                             0, this);
 fileSaveAs->setStatusTip(tr("Saves the actual document
                        under a new filename"));
 fileSaveAs->setWhatsThis(tr("Save As\n\nSaves the actual document
                        under a new filename"));
 connect(fileSaveAs, SIGNAL(activated()), this,
SLOT(slotFileSave()));

 fileClose = new QAction(tr("Close File"), tr("&Close"),
              QAccel::stringToKey(tr("Ctrl+W")), this);
 fileClose->setStatusTip(tr("Closes the actual document"));
 fileClose->setWhatsThis(tr("Close File\n\nCloses
                        the actual document"));
 connect(fileClose, SIGNAL(activated()), this, SLOT(slotFileClose()));

 filePrint = new QAction(tr("Print File"), tr("&Print"),
              QAccel::stringToKey(tr("Ctrl+P")), this);
 filePrint->setStatusTip(tr("Prints out the actual document"));
 filePrint->setWhatsThis(tr("Print File\n\nPrints out
                        the actual document"));
 connect(filePrint, SIGNAL(activated()), this, SLOT(slotFilePrint()));
```

```
fileQuit = new QAction(tr("Exit"), tr("E&xit"),
               QAccel::stringToKey(tr("Ctrl+Q")), this);
fileQuit->setStatusTip(tr("Quits the application"));
fileQuit->setWhatsThis(tr("Exit\n\nQuits the application"));
connect(fileQuit, SIGNAL(activated()), this, SLOT(slotFileQuit()));

editUndo = new QAction(tr("Undo"), tr("U&ndo"),
               QAccel::stringToKey(tr("Ctrl+Z")), this);
editUndo->setStatusTip(tr("Reverts the last editing step"));
editUndo->setWhatsThis(tr("Undo\n\nReverts the last editing step"));
connect(editUndo, SIGNAL(activated()), this, SLOT(slotEditUndo()));

editRedo = new QAction(tr("Redo"), tr("R&edo"),
               QAccel::stringToKey(tr("Ctrl+Y")), this);
editRedo->setStatusTip(tr("Re-execute the last undone step"));
editRedo->setWhatsThis(tr("Redo\n\nRe-execute the last undone step"));
connect(editRedo, SIGNAL(activated()), this, SLOT(slotEditRedo()));

editCut = new QAction(tr("Cut"), tr("Cu&t"),
               QAccel::stringToKey(tr("Ctrl+X")), this);
editCut->setStatusTip(tr("Cuts the selected section
                    and puts it to the clipboard"));
editCut->setWhatsThis(tr("Cut\n\nCuts the selected section
                    and puts it to the clipboard"));
connect(editCut, SIGNAL(activated()), this, SLOT(slotEditCut()));

editCopy = new QAction(tr("Copy"), tr("&Copy"),
               QAccel::stringToKey(tr("Ctrl+C")), this);
editCopy->setStatusTip(tr("Copies the selected section
                          to the clipboard"));
editCopy->setWhatsThis(tr("Copy\n\nCopies the selected section
                          to the clipboard"));
connect(editCopy, SIGNAL(activated()), this, SLOT(slotEditCopy()));

editPaste = new QAction(tr("Paste"), tr("&Paste"),
               QAccel::stringToKey(tr("Ctrl+V")), this);
editPaste->setStatusTip(tr("Pastes the clipboard contents
                          to actual position"));
editPaste->setWhatsThis(tr("Paste\n\nPastes the clipboard contents
                          to actual position"));
connect(editPaste, SIGNAL(activated()), this, SLOT(slotEditPaste()));

editFind = new QAction(tr("Find"), tr("&Find"),
               QAccel::stringToKey(tr("Ctrl+F")), this);
editFind->setStatusTip(tr("Searches the text in the document"));
editFind->setWhatsThis(tr("Search\n\nSearches the text
                          in the document"));
connect(editFind, SIGNAL(activated()), this, SLOT(slotEditFind()));
```

```
editFindNext = new QAction(tr("Find Next"), tr("Find &Next"),
                    QAccel::stringToKey(tr("F3")), this);
editFindNext->setStatusTip(tr("Repeats the last search"));
editFindNext->setWhatsThis(tr("Repeat search\n\nRepeats
                          the last search"));
connect(editFindNext, SIGNAL(activated()),
                   this, SLOT(slotEditFindNext()));

viewToolBar = new QAction(tr("Toolbar"), tr("Tool&bar"),
                          0, this, 0, true);
viewToolBar->setStatusTip(tr("Enables/disables the toolbar"));
viewToolBar->setWhatsThis(tr("Toolbar\n\nEnables/disables
                           the toolbar"));
connect(viewToolBar, SIGNAL(toggled(bool)),
                  this, SLOT(slotViewToolBar(bool)));

viewStatusBar = new QAction(tr("Statusbar"), tr("&Statusbar"),
                          0, this, 0, true);
viewStatusBar->setStatusTip(tr("Enables/disables the statusbar"));
viewStatusBar->setWhatsThis(tr("Statusbar\n\nEnables/disables
                          the statusbar"));
connect(viewStatusBar, SIGNAL(toggled(bool)),
                 this, SLOT(slotViewStatusBar(bool)));

formatLeft = new QAction(tr("Left"), tr("&Left"), 0, this);
formatLeft->setStatusTip(tr("Aligns left"));
formatLeft->setWhatsThis(tr("Left\n\nAligns left"));
connect(formatLeft, SIGNAL(activated()), this,
SLOT(slotFormatLeft()));

formatCenter = new QAction(tr("Center"), tr("&Center"), 0, this);
formatCenter->setStatusTip(tr("Aligns center"));
formatCenter->setWhatsThis(tr("Center\n\nAligns center"));
connect(formatCenter, SIGNAL(activated()),
                   this, SLOT(slotFormatCenter()));

formatRight = new QAction(tr("Right"), tr("&Right"), 0, this);
formatRight->setStatusTip(tr("Aligns right"));
formatRight->setWhatsThis(tr("Right\n\nAligns right"));
connect(formatRight, SIGNAL(activated()),
                    this, SLOT(slotFormatRight()));

helpAboutApp = new QAction(tr("About"), tr("&About..."), 0, this);
helpAboutApp->setStatusTip(tr("About the application"));
helpAboutApp->setWhatsThis(tr("About\n\nAbout the application"));
connect(helpAboutApp, SIGNAL(activated()),
                     this, SLOT(slotHelpAbout()));
}
```

```
void QtEditApp::initMenuBar()
{
 ///////////////////////////////////////////////////////////////////
 // ПАНЕЛЬ МЕНЮ

 ///////////////////////////////////////////////////////////////////
 // Команды меню fileMenu
 fileMenu=new QPopupMenu();
 fileNew->addTo(fileMenu);
 fileOpen->addTo(fileMenu);
 fileClose->addTo(fileMenu);
 fileMenu->insertSeparator();
 fileSave->addTo(fileMenu);
 fileSaveAs->addTo(fileMenu);
 fileMenu->insertSeparator();
 filePrint->addTo(fileMenu);
 fileMenu->insertSeparator();
 fileQuit->addTo(fileMenu);

 ///////////////////////////////////////////////////////////////////
 // Команды меню editMenu
 editMenu=new QPopupMenu();
 editUndo->addTo(editMenu);
 editRedo->addTo(editMenu);
 editCut->addTo(editMenu);
 editCopy->addTo(editMenu);
 editPaste->addTo(editMenu);
 editFind->addTo(editMenu);
 editFindNext->addTo(editMenu);

 ///////////////////////////////////////////////////////////////////
 // Команды меню viewMenu
 viewMenu=new QPopupMenu();
 viewMenu->setCheckable(true);
 viewToolBar->addTo(viewMenu);
 viewStatusBar->addTo(viewMenu);
 ///////////////////////////////////////////////////////////////////
 // ПОМЕСТИТЕ СЮДА СПЕЦИФИЧЕСКИЕ КОМАНДЫ МЕНЮ ПРИЛОЖЕНИЯ

 ///////////////////////////////////////////////////////////////////
 // Команды меню formatMenu
 formatMenu=new QPopupMenu();
 formatLeft->addTo(formatMenu);
 formatCenter->addTo(formatMenu);
 formatRight->addTo(formatMenu);
```

```
 ////////////////////////////////////////////////////////////////////
 // Команды меню helpMenu
 helpMenu=new QPopupMenu();
 helpAboutApp->addTo(helpMenu);

 ////////////////////////////////////////////////////////////////////
 // КОНФИГУРАЦИЯ ПАНЕЛИ МЕНЮ
 menuBar()->insertItem(tr("&File"), fileMenu);
 menuBar()->insertItem(tr("&Edit"), editMenu);
 menuBar()->insertItem(tr("&View"), viewMenu);
 menuBar()->insertItem(tr("&Format"), formatMenu);
 menuBar()->insertSeparator();
 menuBar()->insertItem(tr("&Help"), helpMenu);
}
```

65. В том же файле измените приемники класса `QtEditApp` в соответствии с текстом листинга 8.5.

Листинг 8.5. Приемники класса QtEditApp

```
/** Отменяет последнюю операцию редактирования */
void QtEditApp::slotEditUndo()
{
 statusBar()->message(tr("Reverting the last editing operation..."));

 view-> undo();

 statusBar()->message(tr("Ready."));
}

/** Восстанавливает последнюю отмененную операцию */
void QtEditApp::slotEditRedo()
{
 statusBar()->message(tr("Re-executing the last undone step..."));

 view-> redo();

 statusBar()->message(tr("Ready."));
}

/** Производит поиск текста в документе */
void QtEditApp::slotEditFind()
{
 FindDlgImpl dlg;

 statusBar()->message(tr("Searching the text..."));

 dlg.exec();
```

```
  findText     = dlg.findText;
  searchBackwards = dlg.searchBackwards;
  caseSensitive  = dlg.caseSensitive;

  slotEditFindNext();
}

/** Производит поиск следующего вхождения текста */
void QtEditApp::slotEditFindNext()
{
  statusBar()->message(tr("Repeating search..."));

  view-> find(findText, caseSensitive, false, !searchBackwards);

  statusBar()->message(tr("Ready."));
}

/** Выравнивает по левому краю */
void QtEditApp::slotFormatLeft()
{
  statusBar()->message(tr("Aligning the text..."));

  view-> setAlignment( AlignLeft);

  statusBar()->message(tr("Ready."));
}

/** Выравнивает по центру */
void QtEditApp::slotFormatCenter()
{
  statusBar()->message(tr("Aligning the text..."));

  view-> setAlignment( AlignCenter);

  statusBar()->message(tr("Ready."));
}

/** Выравнивает по правому краю */
void QtEditApp::slotFormatRight()
{
  statusBar()->message(tr("Aligning the text..."));

  view-> setAlignment( AlignRight);

  statusBar()->message(tr("Ready."));
}
```

66. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Приложение будет запущено на исполнение.

67. Введите в рабочую область окна какой-либо текст, разбив его на строки, и выберите команду меню **Format | Center** (Формат | Выравнивание по центру). Текст в окне будет отцентрирован по горизонтали, как это показано на рис. 8.3.

Примечание

В имеющейся у меня библиотеке Qt версии 3 выравнивание текста объектом класса `QMultiLineEdit` производится не совсем корректно: после выравнивания на экране остаются остатки нового и старого текстов. Чтобы привести окно в нормальное состояние, его нужно перерисовать (но не функцией `QWidget::repaint`, вызов которой в данном случае не помогает). Надеюсь, что такая грубая ошибка будет исправлена разработчиками библиотеки до выхода этой книги из печати.

Кстати, в версии 2 данной библиотеки выравнивание производилось относительно самой длинной строки, но работало корректно.

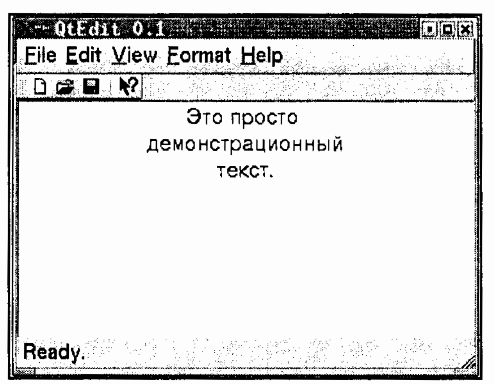


Рис. 8.3. Центрированный текст

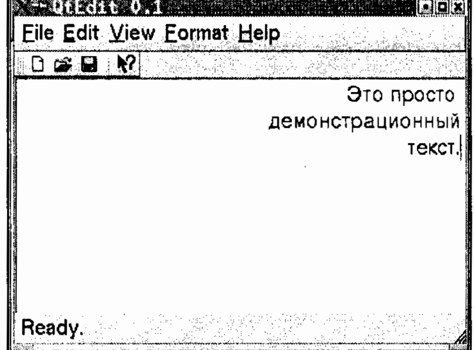


Рис. 8.4. Выравнивание по правому краю

68. Выберите команду меню **Format | Right** (Формат | Выравнивание по правому краю). Текст в окне будет выровнен по правому краю, как это показано на рис. 8.4.

69. Выберите команду меню **Format | Left** (Формат | Выравнивание по левому краю). Текст в окне будет выровнен по левому краю.

70. Введите новый текст и выберите команду меню **Edit | Undo** (Правка | Отменить) или нажмите комбинацию клавиш <Ctrl>+<Z>. Последний введенный текст будет удален.

71. Выберите команду меню **Edit | Redo** (Правка | Повторить) или нажмите комбинацию клавиш <Ctrl>+<Y>. Удаленный текст будет восстановлен.

72. Установите текстовый курсор в начало текста и выберите команду меню **Edit | Find** (Правка | Найти) или нажмите комбинацию клавиш <Ctrl>+<F>. Появится диалоговое окно **Find**, изображенное на рис. 8.5.

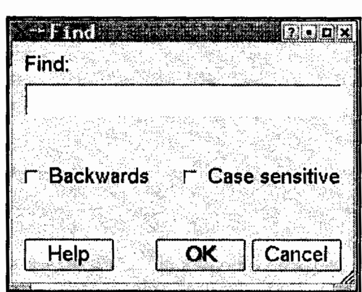


Рис. 8.5. Диалоговое окно **Find**

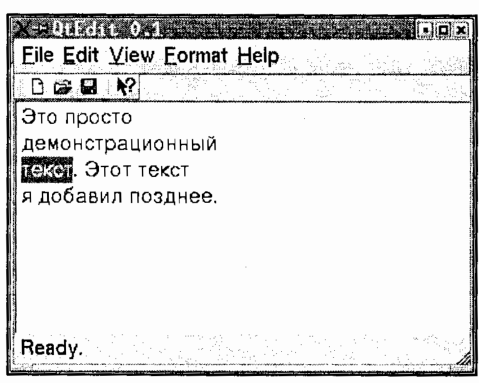


Рис. 8.6. Вывод результатов поиска

73. В текстовое поле **Find** введите фрагмент текста, содержащийся в его конце, и нажмите кнопку **OK**. Текст будет найден и выделен, как это показано на рис. 8.6.
74. Выберите команду меню **Edit | Find Next** (Правка | Найти далее) или нажмите клавишу <F3>. В текстовом окне будет найдено и выделено следующее вхождение искомого фрагмента.
75. Закройте приложение.

Внимание!

Несмотря на то, что приложение включает в себя объект класса, производного от класса `QMultiLineEdit`, доступ к приемникам данного класса из меню можно осуществлять только через приемники класса приложения. Это связано с тем, что функция `initActions`, в которой производится связывание команд меню с приемниками, вызывается в конструкторе класса до функции `initView`, в которой создается объект класса представления.

Как следует из рассматриваемого приложения, класс `QMultiLineEdit` включает в себя средства для создания достаточно сложного редактора.

Создание элементов пользовательского интерфейса было подробно рассмотрено в *главе 5*, а создание диалоговых окон — в *главе 2*. Поэтому мы не будем здесь подробно останавливаться на включении в приложение новых команд меню и диалогового окна для ввода искомого текста, а сразу сосредоточимся на реализации приемников.

Отмена операций редактирования выполняется приемником `QtEditApp::slotEditUndo`, в котором вызывается приемник `QMultiLineEdit::undo`. Восстановление отмененной операции производится приемником `QtEditApp::slotEditRedo`, вызывающим приемник `QMultiLineEdit::redo`.

Поиск нового фрагмента текста выполняется приемником `QtEditApp::slotEditFind`, в котором создается и выводится на экран объект пользова-

тельского диалогового окна. Поскольку элементы управления диалогового окна не сохраняют своего состояния после его закрытия, в классе этого диалогового окна создан приемник, сохраняющий при закрытии диалогового окна текст, содержащийся в текстовом поле, в строковой переменной, а также состояние флажков диалогового окна в логических переменных. После закрытия диалогового окна этот текст и значения этих переменных копируются из переменных класса диалогового окна в переменные класса приложения. После этого вызывается приемник `QtEditApp::slotEditFindNext`, осуществляющий непосредственный поиск текста.

Приемник `QtEditApp::slotEditFindNext` осуществляет поиск текста по заданному образцу. Эта функция вызывается как при поиске первого, так и при поиске последующих вхождений заданного текста. Для поиска заданного фрагмента текста используется функция `QTextEdit::find`, в первом аргументе которой передается искомый фрагмент текста, во втором — логическая величина, определяющая, будет ли при поиске учитываться регистр символов, в третьем — логическая величина, определяющая, является ли искомый фрагмент отдельным словом или фрагментом другого слова, а в четвертом — логическая величина, определяющая направление поиска. Для того чтобы не перегружать созданное нами диалоговое окно поиска флажками, при поиске жестко установлено, что заданный текстовый фрагмент не является отдельным словом.

Выравнивание текста осуществляется приемниками `QtEditApp::slotFormatLeft`, `QtEditApp::slotFormatCenter` и `QtEditApp::slotFormatRight`, имеющими сходную структуру. В каждом из этих приемников вызывается функция `QMultiLineEdit::setAlignment`, которой в качестве аргумента передается соответствующий флаг.

Создание редактора KDE

В библиотеке среды разработки KDevelop для создания текстового редактора предусмотрен специальный класс `KEdit`, являющийся потомком класса `QMultiLineEdit`. Для демонстрации возможностей этого класса создадим демонстрационное приложение **KDEEdit**, текст которого можно найти на прилагаемом к книге CD.

Чтобы создать простейший текстовый редактор:

1. Выберите команду меню **Project | New**.
2. В иерархическом списке типов создаваемых приложений появившегося окна **ApplicationWizard** оставьте выделенным узел **KDE Normal** (Однооконное приложение KDE) и нажмите кнопку **Next**.
3. Введите в текстовое поле **Project name** имя проекта **KDEEdit** и нажмите кнопку **Create**.

4. После завершения работы мастера по созданию приложения нажмите кнопку **Exit**. Заготовка приложения будет создана.
5. В окне иерархических списков раскройте вкладку **Classes** и щелкните левой кнопкой мыши по имени класса `KDEEditView`. Откроется окно редактирования файла kdeeditview.h.
6. В начале этого файла замените строку `#include <qwidget.h>` строкой

   ```
   #include <keditcl.h>
   ```

7. В заголовке класса `KDEEditView` замените строку `class KDEEditView : public QWidget` строкой

   ```
   class KDEEditView : public KEdit
   ```

8. Щелкните правой кнопкой мыши в окне редактирования файла kdeeditview.h и выберите в появившемся контекстном меню команду **Switch Header/Source**. Откроется окно редактирования файла kdeeditview.cpp.
9. Измените конструктор класса `KDEEditView` в соответствии с текстом листинга 8.6.

Листинг 8.6. Конструктор класса `KDEEditView`

```
KDEEditView::KDEEditView(QWidget *parent, const char *name)
  : KEdit(parent, name)
{
 setBackgroundMode(PaletteBase);
}
```

10. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `KDEEditApp` и выберите в появившемся контекстном меню команду **Add member variable**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes**.
11. В текстовое поле **Type** введите тип переменной `KAction*`, в текстовое поле **Name** — имя переменной `editUndo`, в группе **Acceess** установите переключатель **Private** и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.
12. Повторите пункты 10 и 11 для включения в класс переменной `editRedo`.
13. Повторите пункты 10 и 11 для включения в класс переменной `editFind`.
14. Повторите пункты 10 и 11 для включения в класс переменной `editReplace`.
15. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `KDEEditApp` и выберите в появившемся кон-

текстном меню команду **Add slot**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Slots**.

16. В текстовое поле **Declaration** введите сигнатуру приемника `slotEditUndo()`, в текстовое поле **Documentation** — комментарий `Reverts the last user action` и нажмите кнопку **Apply**. В класс будет добавлена новый приемник.
17. Повторите пункты 15 и 16 для включения в класс приемника `slotEditRedo()`, снабдив его комментарием `Re-executes the last undone step`.
18. Повторите пункты 15 и 16 для включения в класс приемника `slotEditFind()`, снабдив его комментарием `Searches the text in the document`.
19. Повторите пункты 15 и 16 для включения в класс приемника `slotEditReplace()`, снабдив его комментарием `Searches and replaces expression` (Производит поиск и замену выражения).
20. Повторите пункты 15 и 16 для включения в класс приемника `slotUpdateUndo(bool)`, снабдив его комментарием `Updates Undo command` (Изменяет состояние команды Undo).
21. Повторите пункты 15 и 16 для включения в класс приемника `slotUpdateRedo(bool)`, снабдив его комментарием `Updates Redo command` (Изменяет состояние команды Redo).
22. Повторите пункты 15 и 16 для включения в класс приемника `slotUpdateCopy(bool)`, снабдив его комментарием `Updates Cut and Copy commands` (Изменяет состояние команд Cut и Copy).
23. Повторите пункты 15 и 16 для включения в класс приемника `slotUpdateSave()`, снабдив его комментарием `Updates Save command` (Изменяет состояние команды Save).
24. Измените приемники класса `KDEEditApp` в соответствии с текстом листинга 8.7.

Листинг 8.7. Приемники класса `KDEEditApp`

```
/** Создает новый документ */
void KDEEditApp::slotFileNew()
{
 slotStatusMsg(i18n("Creating new document..."));

 if(!doc->saveModified())
 {
   // в процессе сохранения документа возникла ошибка
 }
```

```
  else
  {
   doc->newDocument();
   doc->deleteContents();
   setCaption(doc->URL().fileName(), false);
   fileSave->setEnabled( false);
   fileSaveAs->setEnabled( false);
  }

  slotStatusMsg(i18n("Ready."));
 }

/** Открывает существующий документ */
void KDEEditApp::slotFileOpen()
{
  slotStatusMsg(i18n("Opening file..."));

  if(!doc->saveModified())
  {
    // в процессе сохранения документа возникла ошибка
  }
  else
  {
   config->setGroup("General Options");

   QString szPath = config->readPathEntry("Current Path",
                                QString::null);

   KURL url=KFileDialog::getOpenURL( szPath,
   i18n("*.txt |Text files \n * |All files"), this,
                                          i18n("Open File..."));

   if(!url.isEmpty())
   {
    doc->openDocument(url);
    setCaption(url.fileName(), false);
    fileOpenRecent->addURL( url);
    fileSave->setEnabled( false);
    fileSaveAs->setEnabled( true);
   }
  }

  slotStatusMsg(i18n("Ready."));
}

/** Открывает документ из списка последних открытых документов */
void KDEEditApp::slotFileOpenRecent(const KURL& url)
{
  slotStatusMsg(i18n("Opening file..."));
```

```
 if(!doc->saveModified())
 {
   // в процессе сохранения документа возникла ошибка
 }
 else
 {
  doc->openDocument(url);
  setCaption(url.fileName(), false);
  fileSave->setEnabled( false);
  fileSaveAs->setEnabled( true);
 }

 slotStatusMsg(i18n("Ready."));
}

/** Сохраняет существующий документ */
void KDEEditApp::slotFileSave()
{
 slotStatusMsg(i18n("Saving file..."));

 if (doc->URL().fileName() == i18n("Untitled"))
 {
  slotFileSaveAs();
 }
 else
 {
  doc->saveDocument(doc->URL());
  fileSave->setEnabled( false);
 };

 slotStatusMsg(i18n("Ready."));
}

/** Сохраняет документ под другим именем */
void KDEEditApp::slotFileSaveAs()
{
 slotStatusMsg(i18n("Saving file with a new filename..."));

 config->setGroup("General Options");

 QString szPath = config->readPathEntry("Current Path",
                         QDir::currentDirPath());

 KURL url=KFileDialog::getSaveURL(szPath,
   i18n("*.txt |Text files \n * |All files"), this,
                                          i18n("Save as..."));

 if(!url.isEmpty())
 {
  doc->saveDocument(url);
```

```
    fileOpenRecent->addURL(url);
    setCaption(url.fileName(),doc->isModified());
    fileSave->setEnabled( false);
  }

  slotStatusMsg(i18n("Ready."));
}

/** Прекращает работу с файлом */
void KDEEditApp::slotFileClose()
{
  slotStatusMsg(i18n("Closing file..."));

  if(!doc->saveModified())
  {
     // В процессе сохранения документа возникла ошибка

  }
  else
  {
    doc->newDocument();
    setCaption(doc->URL().fileName(), false);
    fileSave->setEnabled( false);
    fileSaveAs->setEnabled( false);
  }

  slotStatusMsg(i18n("Ready."));
}

/** Отменяет последнюю операцию редактирования */
void KDEEditApp::slotEditUndo()
{
  slotStatusMsg(i18n("Reverting last action..."));

  view->undo();

  slotStatusMsg(i18n("Ready."));
}

/** Восстанавливает последнюю отмененную операцию */
void KDEEditApp::slotEditRedo()
{
  slotStatusMsg(i18n("Re-executing last action..."));

  view->redo();

  slotStatusMsg(i18n("Ready."));
}
```

```
/* Вырезает выделенный текст в буфер обмена */
void KDEEditApp::slotEditCut()
{
 slotStatusMsg(i18n("Cutting selection..."));

 view->cut();

 editPaste->setEnabled(true);

 slotStatusMsg(i18n("Ready."));
}

/** Копирует выделенный текст в буфер обмена */
void KDEEditApp::slotEditCopy()
{
 slotStatusMsg(i18n("Copying selection to clipboard..."));

 view->copy();

 editPaste->setEnabled(true);

 slotStatusMsg(i18n("Ready."));
}

/** Вставляет текст из буфера обмена */
void KDEEditApp::slotEditPaste()
{
 slotStatusMsg(i18n("Inserting clipboard contents..."));

 view->paste();

 slotStatusMsg(i18n("Ready."));
}

/** Производит поиск текста в документе */
void KDEEditApp::slotEditFind()
{
 slotStatusMsg(i18n("Searching the text in the document..."));

 view->search();

 slotStatusMsg(i18n("Ready."));
}

/** Производит поиск и замену выражения */
void KDEEditApp::slotEditReplace()
{
 slotStatusMsg(i18n("Searching and replacing expression..."));
```

```
  view->replace();

  slotStatusMsg(i18n("Ready."));
}

/** Изменяет состояние команды Undo */
void KDEEditApp::slotUpdateUndo( bool b)
{
  editUndo->setEnabled( b);
}

/** Изменяет состояние команды Redo */
void KDEEditApp::slotUpdateRedo( bool b)
{
  editRedo->setEnabled( b);
}

/** Изменяет состояние команд Cut и Copy */
void KDEEditApp::slotUpdateCopy( bool b)
{
  editCut->setEnabled( b);
  editCopy->setEnabled( b);
}

/** Изменяет состояние команды Save */
void KDEEditApp::slotUpdateSave()
{
  fileSave->setEnabled( true);
  fileSaveAs->setEnabled( true);
  doc->setModified( true);
}
```

25. Измените конструктор класса `KDEEditApp` в соответствии с текстом листинга 8.8.

Листинг 8.8. Конструктор класса `KDEEditApp`

```
KDEEditApp::KDEEditApp(QWidget*, const char* name):KMainWindow(0, name)
{
  config=kapp->config();

  ///////////////////////////////////////////////////////////////////
  // Производит инициализацию всех компонентов приложения
  initStatusBar();
  initActions();
  initDocument();
  initView();
```

```
 readOptions();

 ///////////////////////////////////////////////////////////////////
 // Делает команды недоступными при запуске приложения
 fileSave->setEnabled(false);
 fileSaveAs->setEnabled(false);
 filePrint->setEnabled(false);
 editUndo->setEnabled(false);
 editRedo->setEnabled(false);
 editCut->setEnabled(false);
 editCopy->setEnabled(false);
 editPaste->setEnabled(false);
}
```

26. Измените функцию `KDEEditApp::initActions` в соответствии с текстом листинга 8.9.

Листинг 8.9. Функция initActions

```
void KDEEditApp::initActions()
{
 fileNewWindow = new KAction(i18n("New &Window"), 0, 0, this,
    SLOT(slotFileNewWindow()), actionCollection(),"file_new_window");
 fileNew = KStdAction::openNew(this, SLOT(slotFileNew()),
                          actionCollection());
 fileOpen = KStdAction::open(this, SLOT(slotFileOpen()),
                          actionCollection());
 fileOpenRecent = KStdAction::openRecent(this,
       SLOT(slotFileOpenRecent(const KURL&)), actionCollection());
 fileSave = KStdAction::save(this, SLOT(slotFileSave()),
                          actionCollection());
 fileSaveAs = KStdAction::saveAs(this, SLOT(slotFileSaveAs()),
                          actionCollection());
 fileClose = KStdAction::close(this, SLOT(slotFileClose()),
                          actionCollection());
 filePrint = KStdAction::print(this, SLOT(slotFilePrint()),
                          actionCollection());
 fileQuit = KStdAction::quit(this, SLOT(slotFileQuit()),
                          actionCollection());
 editUndo = KStdAction::undo(this, SLOT(slotEditUndo()),
                          actionCollection());
 editRedo = KStdAction::redo(this, SLOT(slotEditRedo()),
                          actionCollection());
 editCut = KStdAction::cut(this, SLOT(slotEditCut()),
                          actionCollection());
```

```
    editCopy = KStdAction::copy(this, SLOT(slotEditCopy()),
                              actionCollection());
    editPaste = KStdAction::paste(this, SLOT(slotEditPaste()),
                              actionCollection());
    editFind = KStdAction::find(this, SLOT(slotEditFind()),
                              actionCollection());
    editReplace = KStdAction::replace(this, SLOT(slotEditReplace()),
                              actionCollection());
    viewToolBar = KStdAction::showToolbar(this, SLOT(slotViewToolBar()),
                              actionCollection());
    viewStatusBar = KStdAction::showStatusbar(this,
                SLOT(slotViewStatusBar()), actionCollection());

    fileNewWindow->setStatusText(i18n("Opens a new application window"));
    fileNew->setStatusText(i18n("Creates a new document"));
    fileOpen->setStatusText(i18n("Opens an existing document"));
    fileOpenRecent->setStatusText(i18n("Opens a recently used file"));
    fileSave->setStatusText(i18n("Saves the actual document"));
    fileSaveAs->setStatusText(i18n("Saves the actual document as..."));
    fileClose->setStatusText(i18n("Closes the actual document"));
    filePrint ->setStatusText(i18n("Prints out the actual document"));
    fileQuit->setStatusText(i18n("Quits the application"));

    editUndo->setStatusText(i18n("Reverts the last editing step"));
    editRedo->setStatusText(i18n("Re-execute the last undone step"));
    editCut->setStatusText(i18n("Cuts the selected section and puts
                            it to the clipboard"));
    editCopy->setStatusText(i18n("Copies the selected section
                              to the clipboard"));
    editPaste->setStatusText(i18n("Pastes the clipboard contents
                             to actual position"));
    editFind->setStatusText(i18n("Searches the text in the document"));
    editReplace->setStatusText(i18n("Searches and replaces expression"));

    viewToolBar->setStatusText(i18n("Enables/disables the toolbar"));
    viewStatusBar->setStatusText(i18n("Enables/disables the statusbar"));

    // Для проверки используйте абсолютный путь к файлу kdeeditui.rc
    // при вызове функции createGUI();
    createGUI();

}
```

27. Измените функцию `KDEEditApp::initView` в соответствии с текстом листинга 8.10.

Листинг 8.10. Функция initView

```
void KDEEditApp::initView()
{
 ////////////////////////////////////////////////////////////////////
 // создайте здесь главное окно приложения, которое будет управляться
 // объектом класса KTMainWindow, и свяжите это окно с объектом класса
 // документа для вывода его содержимого на экран.

 view = new KDEEditView(this);
 doc->addView(view);
 setCentralWidget(view);
 setCaption(doc->URL().fileName(),false);

 connect(view, SIGNAL(undoAvailable(bool)), this,
                      SLOT(slotUpdateUndo(bool)));
 connect(view, SIGNAL(redoAvailable(bool)), this,
                      SLOT(slotUpdateRedo(bool)));
 connect(view, SIGNAL(copyAvailable(bool)), this,
                      SLOT(slotUpdateCopy(bool)));
 connect(view, SIGNAL(textChanged()), this, SLOT(slotUpdateSave()));
}
```

28. Откройте окно редактирования файла kdeeditdoc.cpp и после строки `#include <kio/netaccess.h>` вставьте строку

```
#include <qtextstream.h>
```

29. В том же файле измените функции `KDEEditDoc::openDocument`, `KDEEditDoc::saveDocument` и `KDEEditDoc::deleteContents` в соответствии с текстом листинга 8.11.

Листинг 8.11. Функции класса KDEEditDoc

```
/** Открытие документа */
bool KDEEditDoc::openDocument(const KURL& url, const char *format
                                                          /*=0*/)
{
 QString tmpfile;
 KDEEditView *w;

 if( url.isValid())
 {
  KIO::NetAccess::download( url, tmpfile);

  QFile fileTemp( tmpfile);

  fileTemp.open( IO_ReadOnly);
```

```
  if(pViewList)
  {
   w=pViewList->first();

   QTextStream ts( &fileTemp);

   w-> insertText( &ts);
  }

  KIO::NetAccess::removeTempFile( tmpfile);

  modified=false;
 }
 else
  newDocument();

 return true;
}

/** Сохранение документа */
bool KDEEditDoc::saveDocument(const KURL& url, const char *format
                                                              /*=0*/)
{
 KDEEditView *w;

 QFile fileTemp( url.path());

 fileTemp.open( IO_WriteOnly);

 if(pViewList)
 {
  w=pViewList->first();

  QTextStream ts( &fileTemp);

  w-> saveText( &ts);
 }

 modified=false;
 return true;
}

/** Очистка объектов класса представления */
void KDEEditDoc::deleteContents()
{
 KDEEditView *w;

 if(pViewList)
 {
  w=pViewList->first();
```

```
    w-> clear();
   }

  setModified(false);
 }
```

30. Выберите команду меню **Debug | Start** или нажмите кнопку **Debug** в панели инструментов. Приложение будет запущено на исполнение.
31. Введите в рабочую область окна какой-либо текст. В панели инструментов станут доступными кнопки **Undo** и **Save**, как это показано на рис. 8.7.

Примечание

Кнопка **Undo** становится доступной только после нажатия клавиши <Return>.

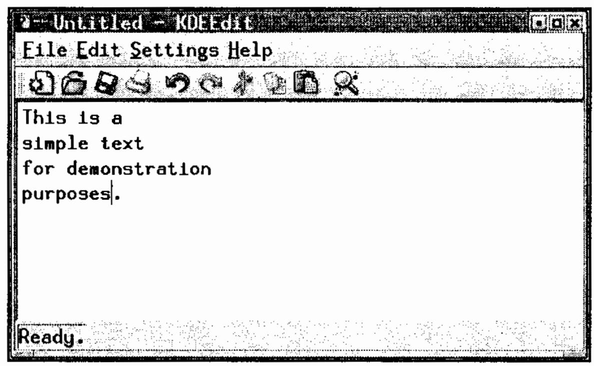


Рис. 8.7. Окно редактора KDE

32. Выделите фрагмент введенного текста. В панели инструментов станут доступными кнопки **Cut** и **Copy**.
33. Выберите команду **Edit | Cut**, нажмите кнопку **Cut** в панели инструментов или комбинацию клавиш <Ctrl>+<X>. Выделенный текст исчезнет и станет доступной кнопка **Paste** в панели инструментов.
34. Выберите команду меню **Edit | Paste**, нажмите кнопку **Paste** в панели инструментов или комбинацию клавиш <Ctrl>+<V>. Удаленный текст восстановится.
35. Выберите команду меню **Edit | Undo**, нажмите кнопку **Undo** в панели инструментов или комбинацию клавиш <Ctrl>+<Z>. Восстановленный текст снова исчезнет и станет доступной кнопка **Redo**.
36. Выберите команду меню **Edit | Find**, нажмите кнопку **Find** в панели инструментов или комбинацию клавиш <Ctrl>+<F>. Появится диалоговое окно **Find**, изображенное на рис. 8.8.

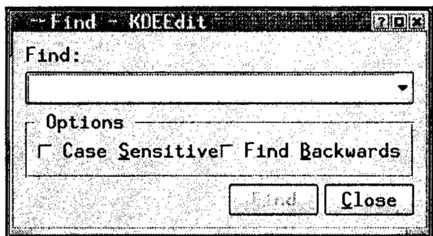


Рис. 8.8. Диалоговое окно **Find**

37. В текстовое поле **Find** введите искомый текст, флажками **Case Sensitive** (С учетом регистра символов) и **Find Backwards** (Поиск в обратном направлении) установите режимы поиска и нажмите кнопку **Find**. Будет найдено первое вхождение искомого текста.
38. Снова нажмите кнопку **Find**. Будет найдено следующее вхождение искомого текста.
39. Закройте диалоговое окно **Find**.
40. Выберите команду меню **Edit | Replace** или нажмите комбинацию клавиш <Ctrl>+<R>. Появится диалоговое окно **Replace**, изображенное на рис. 8.9.

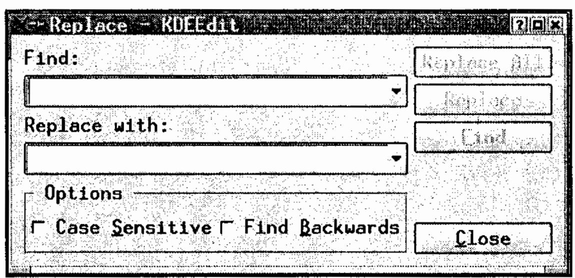


Рис. 8.9. Диалоговое окно **Replace**

41. В текстовое поле **Find** введите заменяемый текст, в текстовое поле **Replace with** (заменить на) — заменяющий текст и нажмите кнопку **Find**. Будет найдено первое вхождение заменяемого текста.
42. Нажмите кнопку **Replace** (Заменить), текст будет заменен и останется выделенным.
43. Повторите пункты 41 и 42. Будет заменено следующее вхождение искомого текста.
44. Закройте диалоговое окно **Replace**.
45. Выберите команду меню **File | Save**, нажмите комбинацию клавиш <Ctrl>+<S> или кнопку **Save** в панели инструментов. Появится диалоговое окно **Save as**, изображенное на рис. 8.10.

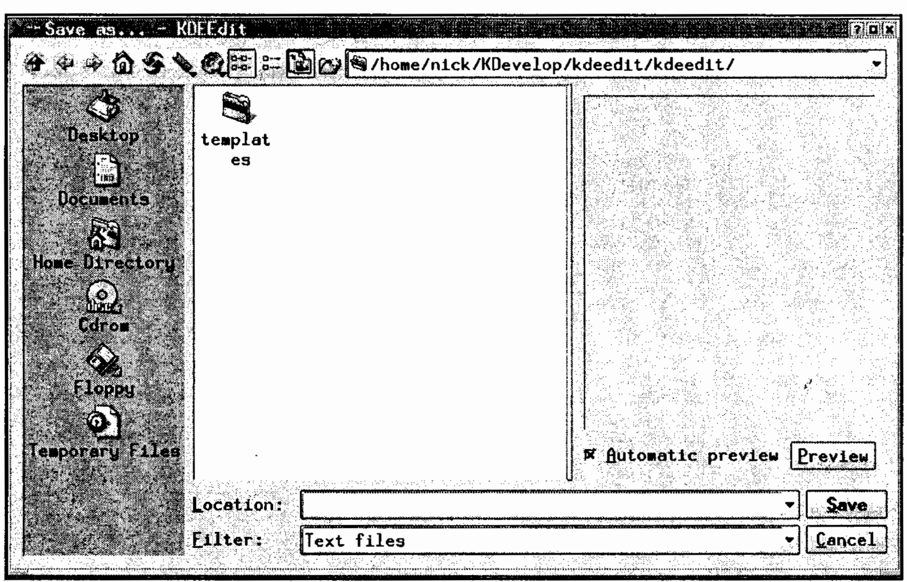


Рис. 8.10. Диалоговое окно **Save as**

46. Сохраните текст в файле, не забыв указать для него расширение txt.
47. Выберите команду меню **File | New**, нажмите комбинацию клавиш <Ctrl>+<N> или кнопку **New** в панели инструментов. Окно редактора очистится.
48. Выберите команду меню **File | Open**, нажмите комбинацию клавиш <Ctrl>+<O> или кнопку **Open** в панели инструментов. Появится диалоговое окно **Open File**, изображенное на рис. 8.11.
49. В окне списка диалогового окна выделите имя файла, в котором был сохранен текст, и нажмите кнопку **OK**. Текст в окне восстановится.

Примечание

При демонстрации свойств данного приложения намеренно был использован английский текст. Это связано с тем, что русский текст некорректно читается из файла.

50. Внесите изменения в текст и выберите команду меню **File | Quit**, нажмите комбинацию клавиш <Ctrl>+<Q> или кнопку **Закрыть** в заголовке окна. Появится окно сообщения, изображенное на рис. 8.12 и предлагающее сохранить внесенные в документ изменения.
51. Нажмите кнопку **No**. Приложение будет закрыто.

Основной целью данного приложения было продемонстрировать стандартные возможности, предоставляемые приложениями KDE и классом `KEdit`

для быстрого создания приложений. Поэтому в приложение нами были включены только стандартные команды, т. е. те команды, указатели на объекты которых возвращаются статическими функциями класса `KStdAction`.

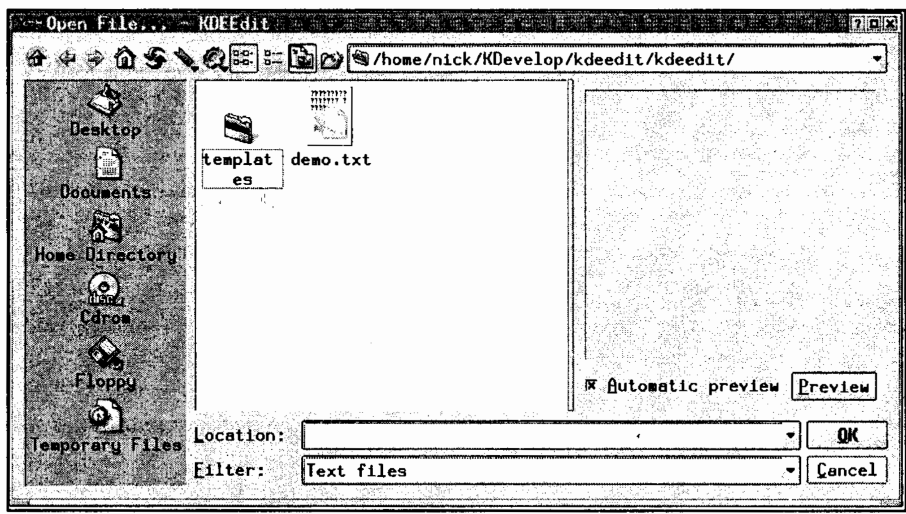


Рис. 8.11. Диалоговое окно **Open File**

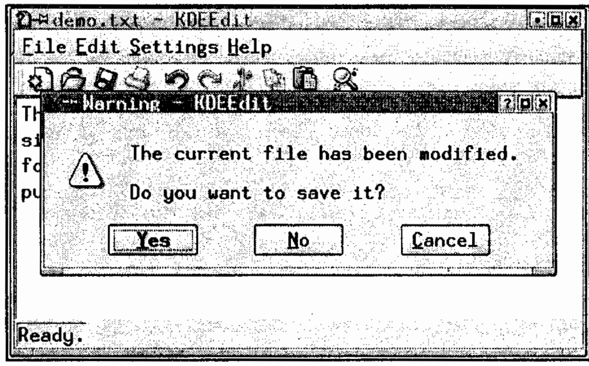


Рис. 8.12. Окно сообщения

В редакторе KDE так же, как и в редакторе Qt, были реализованы команды **Undo** и **Redo**, однако, в отличие от редактора Qt, эти команды становились доступными только в том случае, если они могли выполнять возложенную на них задачу. Для этого в класс приложения были включены приемники `KDEEditApp::slotUpdateUndo` и `KDEEditApp::slotUpdateRedo`. В каждом из них вызывалась функция `KAction::setEnabled`, в качестве аргумента которой передавался аргумент приемника.

Эти приемники были связаны с сигналами класса QMultiLineEdit: приемник slotUpdateUndo был связан с сигналом QMultiLineEdit::undoAvailable, а приемник slotUpdateRedo — с сигналом QMultiLineEdit::redoAvailable. Так что управление доступностью команд меню можно было бы осуществить и в редакторе Qt, однако там это было не принято. Связывание сигнала с приемником производилось в функции KDEEditApp::initView сразу же после создания и инициализации объекта класса представления.

Для установки начального состояния команд **Undo** и **Redo** нами были внесены изменения в конструктор класса KDEEditApp. Поскольку сразу после запуска приложения отсутствуют действия, которые можно отменить или восстановить, для обеих команд вызываются функции KAction::setEnabled с аргументом false.

Помимо команд **Undo** и **Redo**, нами было реализовано управление доступностью команд **Cut**, **Copy**, **Paste**, **Save** и **Save As**. Управление доступностью команд **Cut** и **Copy** осуществлялось аналогично управлению доступности команд **Undo** и **Redo**, однако, т. к. эти команды могут быть или одновременно доступны, или одновременно недоступны, управление доступом к ним выполнялось в одном приемнике KDEEditApp::slotUpdateCopy, связанном с сигналом QMultiLineEdit::copyAvailable. Поскольку все необходимые действия уже были произведены мастером создания приложения, никаких изменений в конструктор класса при реализации управления доступностью команд **Cut** и **Copy** не вносилось.

Управление доступностью команд **Save** и **Save As** осуществлялось несколько иным образом: приемник KDEEditApp::slotUpdateSave, связанный с сигналом QMultiLineEdit::textChanged, только делает эти команды доступными. Недоступными они делаются в конструкторе класса приложения и соответствующих приемниках команд меню. Помимо установки доступности команд в приемнике slotUpdateSave вызовом функции KDEEditDoc::setModified осуществляется также установка флага наличия изменений в объекте класса документа.

Изменения, внесенные в приемник KDEEditApp::slotFileNew, связаны с реализацией управления доступностью команд **Save** и **Save As** и необходимостью очистки окна при вызове данной команды в случае отсутствия изменений в документе. Если сохранение текущего содержимого окна редактора произошло без осложнений, в нем создается новый документ и, в связи с этим, команды **Save** и **Save As** делаются недоступными. Для гарантированной очистки экрана даже в том случае, когда она не была произведена в функции KDEEditDoc::saveModified, в данной функции используется функция KDEEditDoc::deleteContents, которая будет описана далее.

Изменения, внесенные в приемник KDEEditApp::slotFileOpen, более существенны, и это связано с тем, что в данном приложении реализовано сохранение рабочего каталога приложения и выбор конкретного типа документов

(в данном случае текстовых). Подробно связанные с этим изменения описаны в *главе 7*. После загрузки в редактор нового текста из файла команда **Save** делается недоступной, т. к. в документе отсутствуют изменения, а команда **Save As** делается доступной, поскольку файл может быть сохранен под другим именем.

Приемник `KDEEditApp::slotFileOpenRecent` полностью аналогичен приемнику `slotFileOpen` за тем исключением, что в нем не нужно выводить диалоговое окно для получения пути к файлу, поскольку этот путь уже передан в аргументе приемника. Поэтому состояние команд **Save** и **Save As**, устанавливаемое в данной функции, полностью соответствует состоянию команд, устанавливаемому в приемнике `slotFileOpen`.

При сохранении файла, независимо от того, производится ли оно в приемнике `KDEEditApp::slotFileSave` или `KDEEditApp::slotFileSaveAs`, недоступной становится только команда **Save**, поскольку теперь в документе отсутствуют изменения. Даже в том случае, если документ был сохранен под другим именем, то это имя стало новым именем документа и использовать команду **Save** для сохранения файла под старым именем невозможно, т. к. вся информация о нем утеряна.

Изменения в приемнике `KDEEditApp::slotFileClose` связаны с тем, что, помоему, при закрытии файла его содержимое должно удаляться из окна редактирования. Предложенная мастером создания приложений реализация не удовлетворяла этому требованию. Поэтому для данного приемника была выбрана реализация, идентичная реализации приемника `slotFileNew`.

Реализации приемников `KDEEditApp::slotEditUndo` и `KDEEditApp::slotEditRedo` полностью аналогичны рассмотренным ранее реализациям одноименных приемников в приложении **QtEdit**, поэтому мы не будем здесь их снова рассматривать.

Приемники `KDEEditApp::slotEditCut` и `KDEEditApp::slotEditCopy` имеют очень простую структуру. Сначала в них выводится сообщение в строку состояния о выполняемой операции, затем вызывается соответствующий приемник класса `QMultiLineEdit` (`QMultiLineEdit::cut` для вырезания или `QMultiLineEdit::copy` для копирования текста), вызовом функции `KAction::setEnabled` делается доступной команда меню **Paste** и в строку состояния выводится сообщение о готовности выполнения следующих операций.

Приемник `KDEEditApp::slotEditPaste` имеет аналогичную структуру, только в нем вызывается приемник `QMultiLineEdit::paste` и не производится изменения режима доступности команды.

Основные изменения по сравнению с приложением **QtEdit** претерпел приемник `KDEEditApp::slotEditFind`, имеющий теперь тот же формат, что и приемник `KDEEditApp::slotEditPaste`. Все операции по выводу стандартного диалогового окна, поиску и повторному поиску фрагментов текста выполня-

ет теперь функция `KEdit::search`. Точно так же все операции по поиску и замене текста в приемнике `KDEEditApp::slotEditReplace` выполняет функция `KEdit::replace`. Следует только отметить несколько своеобразный алгоритм замены фрагментов текста данной функцией, которая, в отличие от общепринятого стандарта, не производит автоматического поиска следующего фрагмента текста после его замены. Таким образом, пользователь класса `KEdit` теперь избавлен от необходимости самому создавать диалоговые окна для поиска текста.

Для того чтобы сделать рассматриваемый редактор полноценным, в нем реализовано сохранение текста в файле и чтение его оттуда. Это привело к необходимости внесения изменений в класс документа приложения. Особые трудности были связаны с тем, что класс `KEdit` не является, собственно говоря, классом приложения и вместо взаимодействия с классом документа самостоятельно реализует все операции по сохранению в файле и восстановлению из файла своего содержимого. В однооконном приложении это обстоятельство не так существенно, но в многооконном приложении встает вопрос о синхронизации объектов представлений, связанных с одним документом. Здесь у разработчика могут возникнуть серьезные проблемы.

При запуске приложения функция `main` вызывает функцию `KDEEditApp::openDocumentFile`, передавая ей в качестве аргумента первый аргумент командной строки приложения, если он присутствует. При отсутствии в командной строке приложения аргументов функция `openDocumentFile` вызывается без аргументов, т. е. с используемым по умолчанию нулевым аргументом. Функция `openDocumentFile` вызывает функцию `KDEEditDoc::openDocument`, передавая ей свой аргумент. Таким образом, при отсутствии в командной строке приложения аргументов (т. е. в подавляющем большинстве случаев) при запуске приложения функции `openDocument` передается нулевая ссылка на объект класса `KURL`.

При отсутствии проверки корректности передаваемой ссылки на объект класса `KURL` приложение работает не совсем корректно, например в окне редактирования отсутствует текстовый курсор. Поэтому в функции `openDocument`, прежде всего, вызывается функция `KURL::isValid` и в случае некорректности переданного объекта просто создается новый пустой документ.

Если функции был передан корректный объект класса `KURL`, этот объект передается в качестве первого аргумента статической функции `KIO::NetAccess::download`, загружающей указанный ресурс на локальный диск и возвращающей в своем втором аргументе имя файла, в котором был сохранен этот ресурс. Для работы с этим файлом создается объект класса `QFile` и вызывается его функция `QFile::open`, открывающая этот файл только для чтения.

Как уже говорилось ранее, объект класса `KEdit` сам способен работать с файлами, поэтому для чтения в него информации нужно получить указатель

на его объект. В однооконном приложении может существовать только один объект класса приложения, указатель на который должен быть включен в список pViewList. Если этот список отсутствует, то отсутствует и объект класса представления. Поэтому в функции openDocument сначала проверяется корректность указателя на объект списка, а затем, в случае успешного завершения этой проверки, вызовом функции QList::first из списка извлекается первый (и единственный) указатель на объект класса представления.

Объект класса KEdit способен работать только с объектом класса QTextStream, который и создается нами на базе объекта класса QFile. После того, как мы имеем всю необходимую для этого информацию, мы можем вызвать функцию KEdit::insertText, считывающую информацию из файла в объект класса KEdit.

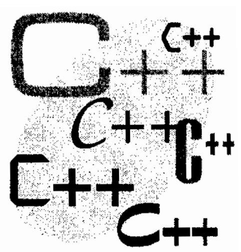
После завершения операции чтения из файла временный файл удаляется с диска вызовом статической функции KIO::NetAccess::removeTempFile и сбрасывается флажок наличия изменений в объекте класса документа.

Структура функции KDEEditDoc::saveDocument, осуществляющей сохранение файла, во многом аналогична структуре функции KDEEditDoc::openDocument, однако есть и существенные различия. При создании объекта класса QFile конструктору класса передается не имя временного файла, а непосредственный путь к файлу, возвращаемый функцией KURL::path. Файл открывается, конечно же, только для записи. Проверки корректности передаваемого пути в этом случае не производится, поскольку подобная проверка должна быть произведена до вызова данной функции.

После получения указателя на объект класса представления и создания объекта класса QTextStream, используемого для работы с файлом, вызывается функция KEdit::saveText, сохраняющая содержимое объекта класса KEdit в файле. После этого флажок наличия изменений в объекте класса документа сбрасывается, поскольку все имевшиеся в нем изменения были сохранены.

Функция KDEEditDoc::deleteContents вызывается в различных частях приложения для очистки объектов классов представления, связанных с удаляемым документом. В этой функции, помимо сброса флажка наличия в документе изменений, для объекта класса представления вызывается приемник QMultiLineEdit::clear, удаляющий из него все содержимое.

ГЛАВА 9



Шаблоны и классы коллекций

При написании программ часто приходится создавать функции или классы для выполнения некоторых действий над объектами разных типов, причем во многих случаях над объектами различных типов необходимо производить одни и те же операции. Например, нужно разработать класс, выводящий на экран содержимое массива в виде графика. Без использования шаблонов пришлось бы создавать специальный класс для каждого возможного типа переменных, хранящихся в данном массиве, и практически полностью повторять тексты функций в каждом из классов. Использование шаблонов позволяет написать один класс, который сможет работать с массивами всех возможных типов. Эта идея была заложена в концепцию шаблонов, описанную в рабочих документах ISO WG21/ANSI X3J16.

Одним из наиболее распространенных способов использования шаблонов являются классы коллекций, позволяющие создавать различные структуры для хранения объектов произвольного типа. Однако, прежде чем перейти к их рассмотрению, нам необходимо познакомиться с самими шаблонами.

Шаблоны

Для задания шаблона используется ключевое слово `template`. Оператор `template`, формат которого приведен ниже, позволяет задавать абстрактные типы данных при объявлении и описании функций и классов.

```
template < [typelist] [, [ arglist ]] > declaration
```

Аргументом оператора `template` является список абстрактных типов (имеющих формат `class identifier` или `typename identifier`) или других объектов, используемых в теле шаблона. В поле `declaration` размещается объявление функции или класса.

Объявление класса с использованием шаблона ничем не отличается от объявления класса без использования шаблона за исключением того, что в дан-

ном классе при объявлении типов его переменных, аргументов функций и возвращаемых значений может применяться идентификатор абстрактного класса, объявленный в аргументе оператора `template`. Пример объявления класса с использованием шаблона приведен в листинге 9.1.

Листинг 9.1. Объявление класса с помощью шаблона

```
// Заголовок класса, использующего шаблон

template <class Type, int len>
class tempClass
{
 private:
 Type buffer[len];
 public:
 void setAt(Type, int);
};

// Реализация класса, использующего шаблон

template <class Type, int len>
void tempClass<Type,len>::setAt(Type x, int i)
{
 buf[i] = x;
};
```

Чтобы создать объект данного класса в программе, необходимо использовать объявление переменной, подобное приведенному ниже.

```
tempClass<double, 256> someObjt;
```

Понятие шаблона

Шаблоны представляют собой механизм создания функций и классов с использованием абстрактных типов переменных, дающих возможность применять вместо них любые допустимые типы переменных. Использование шаблона позволяет создать один класс, способный работать с переменными различных типов, вместо того, чтобы создавать множество классов, по одному на каждый из возможных типов переменных.

Например, для создания функции, сохраняющей тип возвращаемой величины при определении минимальной из двух величин, можно написать ряд перегруженных функций, подобных приведенным в листинге 9.2.

Листинг 9.2. Определение минимальной из двух величин

```
// Выбор минимального из двух целых чисел

int min( int x int y)
```

```
{
 return ( x < y) ? x : y;
}

// Выбор минимального из двух коротких целых чисел

short min( short x, short y)
{
 return ( x < y) ? x : y;
}

// Выбор минимального из двух действительных чисел

float min( float x, float y)
{
 return ( x < y) ? x : y;
}

// и т. д...
```

С использованием шаблона все эти функции могут быть заменены одной функцией, аналогичной приведенной ниже.

```
template <class Type> Type min( Type x, Type y)
{
 return ( x < y) ? x : y;
}
```

Это позволяет уменьшить размер программы и облегчить программирование приложений при сохранении проверки корректности используемых типов данных.

Практически во всех случаях использования шаблонов можно обойтись и без них, однако применение шаблонов дает следующие преимущества:

- облегчает процесс написания программ. Вместо того чтобы писать одинаковые функции и классы для различных типов, достаточно написать одну обобщенную функцию или один обобщенный класс;
- повышает читабельность программы, поскольку обеспечивает возможность использования абстрактных типов данных;
- обеспечивает проверку типов используемых данных. Поскольку типы данных, с которыми работает данный шаблон, становятся известными при компиляции программы, компилятор может производить проверку на совместимость используемых в функции типов данных.

Во многих отношениях шаблоны аналогичны макросам условной трансляции, подставляющим вместо себя выражение, в котором аргументы макроса заменяются указанными в них переменными. Например, вместо приведенного выше шаблона функции мог бы быть использован следующий шаблон:

```
#define min(x, y) (((x) < (y)) ? (x) : (y))
```

Однако при использовании этого макроса могут возникнуть определенные проблемы, связанные с тем, что:

- транслятор не может проверить совместимость типов аргументов макроса, поэтому производимые в нем действия будут осуществляться без специальной проверки типов аргументов;
- переменные x и y дважды включаются в макрос, поэтому если, например, любой из этих аргументов инкрементируется, то он будет инкрементирован дважды;
- поскольку макросы раскрываются препроцессором, то ошибки трансляции будут указаны для операторов раскрытого макроса, а не для самого макроса. При отладке макрос будет представляться в раскрытом виде, что затрудняет сопоставление исходного текста программы с ее отладочным текстом.

До появления шаблонов многие решаемые с их помощью задачи реализовывались с использованием функций, в качестве аргументов которым передавались указатели на переменные типа void, поскольку к этому типу может быть преобразован любой другой указатель. Таким образом, появлялась возможность производить простейшие действия над объектами, не обращая внимание на их тип. Однако это удобство имеет и оборотную сторону: игнорирование истинных типов аргументов функции не позволяло производить проверку на их совместимость, отсутствие которой могло привести к появлению трудноуловимых ошибок.

Использование шаблонов позволяет создавать функции и классы, работающие с типизированными данными. При написании функций они имеют абстрактные типы, которые преобразуются в конкретные типы в процессе компиляции. При этом для каждого типа аргумента, используемого при вызове данной функции или класса, в файле, содержащем описание данной функции или класса, создается версия данной функции или класса, использующая аргументы данного типа.

Такой подход позволяет транслятору рассматривать функции и классы, использующие абстрактные типы, как прототипы функций или классов для создания их версий под конкретные типы данных, задействованные в создаваемом приложении. Использование шаблонов повышает читабельность функций, поскольку позволяет не создавать специальных версий данных функций для сложных типов, таких как объекты классов или структур.

Шаблоны функций

Шаблоны функций используются для замены набора функций, имеющих один и тот же текст, но работающих с аргументами, имеющими различный тип. Пример шаблона функции приведен в листинге 9.3.

Листинг 9.3. Пример шаблона функции

```
template <class Type>
void swapIt( Type& x, Type& y)
{
 Type z;
 z = x;
 x = y;
 y = z;
}
```

Приведенный шаблон функции производит обмен значениями своих аргументов, в качестве которых могут выступать не только переменные простых типов, таких как `int` и `long`, но и объекты любых созданных пользователем типов. Этой функции можно передавать в качестве аргументов даже объекты классов, в которых определен конструктор, не имеющий аргументов, и переопределена операция присваивания.

При трансляции шаблона функции производится проверка типов его аргументов, поэтому он не позволит производить обмен значениями между объектами, имеющими несовместимые типы, поскольку компилятор имеет информацию о типе переменных `x` и `y` на момент трансляции. Несовместимыми в данном случае считаются два типа, если хотя бы для одного из них отсутствует неявное преобразование другому типу.

Вызов функции, использующей шаблон, практически ничем не отличается от вызова обычной функции. В листинге 9.4 приведен пример использования описанной выше функции.

Листинг 9.4. Пример использования функции

```
double r = 10.0;
double s = 20.0;
int i = 5;
swapIt( r, s); // OK
swapIt( i, r); // Ошибка. Использованы различные типы аргументов.
```

Для обеспечения совместимости типов можно произвести явное преобразование типов аргументов или явным образом указать тип аргументов в шаблоне функции. В листинге 9.5 приведен пример такого преобразования.

Листинг 9.5. Пример преобразования типов аргументов

```
template <class Type>
Type someFunc(Type x)
```

```
{
 return ++x;
}

void otherFunc(double& r)
{
 r = someFunc<int>(r); // Вызывает функцию someFunc(int)
}
```

При явном задании типа аргумента для него при необходимости производится соответствующее преобразование типов. В приведенном выше примере переменная `r`, имеющая тип `double`, будет преобразована к типу `int`.

При первом вызове функции, использующей шаблон с новым типом аргумента, транслятор создает версию данной функции, в которой абстрактный тип заменяется использованным при вызове функции типом. После этого все последующие вызовы данной функции с этим типом аргумента будут переадресовываться этой версии функции. Если в различных модулях создается несколько идентичных версий функции с одинаковыми типами аргументов, в исполняемом файле останется только одна версия из них.

Таким образом, чтобы в программе появилась версия функции, использующей данный тип аргумента, надо, чтобы транслятор встретил объявление этой функции с указанным типом аргумента. Поскольку трансляция приложения производится по файлам, необходимо, чтобы реализация функции и его вызов располагались в одном файле. В противном случае нужная версия функции не будет создана, а при вызове ее из другого файла компоновщик выдаст соответствующее сообщение об ошибке.

Для исключения подобных ситуаций в файле реализации функции необходимо указать все типы аргументов, с которыми она будет использоваться. Особое внимание этому объявлению следует уделить в том случае, если функция помещена в библиотеку. Например, для того чтобы функция `swapIt` могла бы вызываться с аргументами типа `short`, в файл реализации этой функции нужно поместить следующую строку:

```
template void swapIt<short>( short&, short&);
```

Поскольку типы аргументов однозначно задаются в списке формальных параметров, эта строка может быть записана и в таком виде:

```
template void swapIt( short&, short&);
```

Шаблоны классов

Шаблоны классов используются для замены набора классов, имеющих один и тот же набор переменных и функций, но в котором эти переменные, аргументы и возвращаемые значения функций имеют различный тип. Пример шаблона класса приведен в листинге 9.6.

Листинг 9.6. Пример шаблона класса

```
template <class Type, int n>
class someClass
{
 public:
  someClass( void);
  ~someClass( void);
  void someFunc( Type*);
 private:
  Type array[n];
};
```

В этом примере шаблон класса имеет два параметра, первый из которых имеет тип `Type`, а второй является целочисленной переменной `n`. Параметру `Type` может быть передано значение любого типа, включая структуру или класс. Параметру `n` может быть передано значение любой целочисленной константы. Поскольку переменная `n` представляет собой константу, определяемую в процессе компиляции, пользователь может создать массив с размером `n`, используя стандартное объявление массива.

Объявление членов класса, использующего шаблон, несколько отличается от объявления членов класса не использующих шаблон. Так, текст функции `someFunc`, являющейся членом описанного выше класса, может иметь вид, приведенный в листинге 9.7.

Листинг 9.7. Текст функции someFunc

```
template <class Type, int n>
void someClass< Type, n >::someFunc( Type* a)
{
 for(int i=0; i < n; i++)
  array[i] = a[i];
}
```

Хотя в конструкторах и деструкторах имя шаблона класса упоминается дважды, параметры шаблона должны полностью указываться только один раз.

```
template <class Type, int n>
someClass< Type, n >::someClass( void)
{
 // Конструктор класса
}

template <class Type, int n>
someClass< Type, n >::~someClass( void)
```

```
{
 // Деструктор класса
}
```

В отличие от шаблонов функций, при создании конкретных версий классов пользователь должен в явном виде указать передаваемые аргументы шаблона класса. В листинге 9.8 приведены примеры правильного и неправильного объявления объекта класса `someClass` в приложении.

Листинг 9.8. Объявление объекта класса someClass

```
someClass< char, 6 > First;     // OK
someClass< double, c++ > Second; // Ошибка, второй аргумент шаблона
                                 // должен быть константой
```

Для шаблона класса (или функции) не создается никакого исполнительного кода, пока транслятор не встретит объявления данного класса с конкретными параметрами. Более того, исполнительный код для функций-членов класса создается только в том случае, если эта функция вызывается в данной версии класса. Шаблон класса сначала определяется, а затем для него создается необходимая версия класса. Компилятор не создает версии класса для данного типа аргументов, пока этот класс не будет вызван с этим типом аргументов.

При работе с шаблонами классов следует особое внимание обращать на положение угловых и круглых скобок (<>). Необходимо таким образом разместить круглые скобки в выражениях, чтобы за угловые скобки не могли быть ошибочно приняты такие операции, как >> и ->. Например, выражение

```
someClass< int, x < y ? x : y > Z;
```

должно быть заменено выражением

```
someClass< int, (x < y ? x : y) > Z;
```

Кроме того, необходимо обратить внимание на макросы, использующие угловые скобки в качестве аргументов шаблонов.

При рассмотрении шаблонов функций уже отмечалась проблема, связанная с тем, что без предварительного объявления транслятор создает только те версии функции, которые присутствуют в ее файле реализации. Та же самая проблема возникает и при создании шаблонов классов. Способ решения этой проблемы для шаблонов классов аналогичен способу ее решения для шаблонов функций. Например, подобное объявление класса `someClass` в его файле реализации создаст версию класса, работающую с действительными переменными и способного хранить шестнадцать переменных

```
template class someClass<float, 16>;
```

Это выражение создает соответствующую версию класса, не создавая самого объекта данного класса. При этом программный код генерируется для всех функций класса.

Если необходимо создать программный код только для некоторых функций класса, все они могут быть объявлены по отдельности как шаблоны функций.

Примечание

При объявлении отдельных функций класса необходимо помнить о том, что для работы с ними надо сначала создать объект класса, а потом его уничтожить. Поэтому, помимо самих функций, нужно объявить еще конструктор и деструктор класса. Таким образом, для обеспечения вызова единственной функции класса `someClass` в файл его реализации придется вставить следующие строки:

```
template someClass <int, 10>::someClass();
template someClass <int, 10>::~someClass();
template void someClass <int, 10>::someFunc( int*);
```

Поэтому объявление функций класса по отдельности оправдано только в том случае, если реально в нем будет использоваться ограниченный набор его функций, и предполагаются жесткие ограничения на объем исполняемого кода программы.

Классы коллекций

Библиотека Qt предлагает широкий выбор классов для работы с группами объектов. Эти классы могут быть условно разделены на две группы:

- классы, созданные с использованием шаблонов;
- классы, созданные без использования шаблонов.

В отличие от применяемой в приложениях Visual Studio библиотеки MFC, где каждому классу коллекции, содержащей объекты любого типа, соответствовал класс коллекции, содержащий указатели на объекты любого типа, а классы коллекций, не использующие шаблонов, считались устаревшими, в библиотеке Qt реализован достаточно утилитарный набор классов коллекций.

Виды классов коллекций

Классы коллекций различаются между собой по видам и по типу своих элементов. Вид класса определяет способ организации и хранения элементов в классе. В библиотеке Qt содержатся классы трех видов: списки, массивы и карты отображений (иногда называемые словарями). Это позволяет выбрать способ хранения информации максимально соответствующей поставленной задаче.

Краткое описание каждого вида классов приведено ниже.

- *Списки* — представляют собой упорядоченный, неиндексированный набор элементов, организованных в связанный список. В списке возможен непосредственный доступ к его первому, последнему и текущему элементу. Для поиска всех остальных элементов необходимо произвести последовательный поиск, основываясь на хранящихся в каждом элементе списка указателях на предыдущий и последующий элементы. Эта структура позволяет сравнительно просто добавлять элементы в любую позицию списка.
- *Массивы* — представляют собой упорядоченный набор объектов, каждому из которых соответствует целочисленный индекс. Массив может расширяться в процессе выполнения программы.
- *Карты отображений* (иногда называемые словарями) — представляют собой набор объектов, для доступа к которым используется некоторый ключевой объект.

Простейшим способом создания класса коллекции, хранящего объекты любого типа и проверяющие соответствие типа хранящихся объектов, является использование содержащихся в библиотеке Qt шаблонов стандартных классов коллекций.

Как и в библиотеке MFC, в библиотеке Qt различают классы коллекций, хранящие объекты классов или значения простых типов, и классы коллекций, хранящие указатели на объекты классов. Ниже приведен список этих классов и дано их краткое описание.

- Классы коллекций, хранящие объекты:
  - `QArray` — шаблон класса коллекции, используемый для создания массивов;
  - `QValueList` — шаблон класса коллекции, используемый для создания связных списков;
  - `QValueStack` — шаблон класса коллекции, используемый для создания стеков. Является потомком класса `QValueList`;
  - `QMap` — шаблон класса коллекции, используемый для создания карт отображений.
- Классы коллекций, хранящие указатели на объекты:
  - `QCache` и `QIntCache` — шаблоны карты отображений, применяемой для доступа к постоянно используемой информации. Различаются типом ключей, служащих для доступа к информации;
  - `QAsciiDict`, `QDict`, `QIntDict` и `QPtrDict` — шаблоны карты отображений, используемой для хранения пользовательской информации. Различаются типом ключей, служащих для доступа к информации;

- QList — шаблон класса коллекции, используемый для создания связных списков;
- QQueue — шаблон класса коллекции, используемый для создания очередей (первым пришел — первым ушел);
- QStack — шаблон класса коллекции, используемый для создания стека (последним пришел — первым ушел);
- QVector — шаблон класса коллекции, используемый для создания массивов.

Классы коллекций, хранящие указатели на объекты, имеют один общий базовый абстрактный класс QCollection. Этот класс предназначен для хранения элементов, имеющих тип Item (обобщенный тип элемента коллекции), указатель на который имеет тип void*. Класс QCollection хранит информацию о числе своих элементов и о стратегии их уничтожения (о необходимости уничтожения объекта при исключении его из коллекции).

Объекты потомков класса QCollection хранят только указатели на объекты, что, с одной стороны, обеспечивает экономию памяти и повышает быстродействие приложения, но, с другой стороны, не обеспечивает уникальность данных, которые могут быть изменены с использованием другого указателя на элемент класса коллекции. Проблему уникальности данных может решить применение классов, хранящих объекты, но набор этих классов намного скромнее. Поэтому в класс QCollection включена защищенная виртуальная функция QCollection::newItem, позволяющая создавать копию включаемого объекта. По умолчанию эта функция не включает новый объект, а возвращает указатель на старый.

По умолчанию элементы коллекции не уничтожаются при исключении из нее. Для того чтобы изменить эту стратегию уничтожения элементов коллекции, следует вызвать функцию QCollection::setAutoDelete с аргументом true. В этом случае при вызове виртуальной функции QCollection::deleteItem по умолчанию удаляемый объект будет уничтожаться, хотя эту функцию можно и перегрузить, чтобы вызов функции QCollection::setAutoDelete не оказывал никакого влияния на стратегию уничтожения.

Непосредственными потомками класса QCollection являются классы QGCache, QGDict, QGList и QGVector. Эти классы не являются абстрактными, но их самостоятельное использование запрещено, поскольку они, как и их базовый класс, оперируют объектами типа Item, т. е. указателями типа void*. Использоваться могут только перечисленные выше потомки этих классов.

Как уже говорилось, в библиотеке MFC классы коллекций, не использующие шаблонов, считаются устаревшими. В библиотеке Qt эти классы являются потомками классов, использующих шаблоны. В данных классах жестко фиксируется тип хранящихся в них объектов и в них добавлены специальные функции, ориентированные на работу с этим типом объектов. Напри-

мер, класс `QByteArray` является потомком класса `QArray`, оптимизированным для хранения байтовых массивов.

При выборе класса коллекции для своего приложения необходимо сопоставить характеристики различных видов классов и другие особенности их реализации. Эти характеристики включают в себя:

- особенности различных видов классов, такие как порядок расположения объектов, индексирование и производительность. Эти параметры приведены в табл. 9.1;
- возможность выбора класса, не использующего шаблоны и оптимизированного для работы с данным типом объектов;
- возможность проведения диагностики хранимых объектов;
- проверка типов при работе с данным классом.

В табл. 9.1 приведены характеристики имеющихся видов классов коллекций.

Столбцы 2 и 3 описывают упорядоченность и возможность доступа по индексу к хранящемуся объекту. Под упорядоченностью понимается возможность введения понятия предыдущего и последующего объекта в наборе.

Столбцы 4 и 5 описывают скорость выполнения определенных операций с объектами данных классов. Важность каждого из этих параметров зависит от конкретного приложения.

Таблица 9.1. *Характеристики классов коллекций*

Вид класса	**Упорядочен?**	**Индексируется?**	**Скорость вставки элемента**	**Скорость поиска элемента**
Список	Да	Нет	Быстрая	Медленная
Массив	Да	Целыми числами	Медленная	Быстрая
Карта отображений	Да	По ключу	Быстрая	Быстрая

Массивы

Массивы являются, наверное, самым распространенным типом классов коллекции, они во многом аналогичны обычным массивам, представляющим собой фиксированные области памяти для хранения упорядоченных объектов, но их размер может определяться и изменяться в процессе выполнения программы. Поэтому объекты этих классов используются в тех случаях, когда нельзя или сложно предсказать заранее размер используемого массива.

Поскольку массивы являются простейшим по реализации типом классов коллекций, при их использовании особенно остро встает вопрос об эффективности доступа к элементам, что, в свою очередь, возлагает на разработчика особую ответственность за правильный выбор класса коллекции. Выбор должен быть произведен между классом `QArray`, хранящим сами объекты, и классом `QVector`, хранящим указатели на объекты.

Рассмотрим сначала класс `QArray`. Объект этого класса может хранить значения простых типов или объекты структур и классов, не имеющих конструкторов, деструкторов и виртуальных функций (в Visual Studio требовалось только, чтобы класс имел конструктор без аргументов, копирующий конструктор и переопределенную операцию присваивания). Для сравнения и копирования значений своих элементов класс `QArray` использует побитовые операции.

При работе с классом `QArray` и производными от него классами необходимо помнить о том, что их объекты используют явное разделение памяти. Использование явного разделения памяти в этих классах позволяет не только минимизировать объем применяемой приложением оперативной памяти, но и сократить затраты на ненужное копирование объектов.

При использовании классом `QArray` разделяемого блока данных в его объекте содержится только указатель на этот блок, а сам блок, помимо данных, содержит счетчик ссылок на себя, указывающий на то, сколько объектов классов в настоящее время применяют этот блок. При создании нового объекта класса, использующего этот блок, или при передаче указателя на этот блок объекту уже существующего класса счетчик ссылок увеличивается на единицу. При уничтожении объекта класса, ссылающегося на блок, счетчик ссылок уменьшается на единицу. Как только счетчик ссылок примет нулевое значение, разделяемый блок данных уничтожается.

Естественно, что при внесении изменений в блок разделяемых данных из любого объекта эти изменения сразу же отразятся на всех остальных объектах, использующих этот блок. Такое поведение не всегда желательно. Поэтому в библиотеке Qt предусмотрено создание индивидуальных блоков данных для конкретного объекта класса (который впоследствии снова может стать разделяемым). Поэтому, чтобы потом долго не искать источник необъяснимой ошибки, разработчику следует четко понимать, когда он обеспечивает доступ объекта к разделяемым данным, а когда передает их в монопольное пользование.

В библиотеке Qt различают *поверхностное* (shallow) и *глубокое* (deep) копирование данных. При поверхностном копировании передается только указатель на объект данных, т. е. этот объект становится разделяемым ресурсом. При глубоком копировании данные переписываются в другую область памяти и указатель на нее передается объекту. Таким образом, данные передаются объекту класса в монопольное владение.

Внимание!

При работе с классом QArray следует помнить о том, что операция присваивания QArray::operator= производит поверхностное копирование информации, которая поступает при этом в совместную собственность объектов. Для глубокого копирования информации необходимо использовать функцию QArray::copy, осуществляющую глубокое копирование разделяемого блока данных в другой объект, или функцию QArray::detach, выделяющую данные из разделяемого блока в монопольное владение объекта, как это показано в листинге 9.9.

Листинг 9.9. Выделение данных в монопольное владение объекта

```
QByteArray x(2), y(5)  // Создано два массива разной длины
y[0] = 16; y[1] = 32;  // Инициализация массива
x = y;          // Массив x изменяет размер и разделяет
                // содержимое массива y
QByteArray z = x;     // Все массивы имеют одинаковый размер
                      // и разделяют один и тот же блок данных
x.detach(); // Массив x имеет свой блок данных, содержимое
            // которого совпадает с содержимым разделяемого
            // блока
x[0] = 8; a[1] = 64;   // Изменение содержимого массива x
y = a.copy();          // Глубокое копирование массива x в массив y
```

Практически все остальные классы библиотеки Qt используют неявное разделение памяти, когда любое внесение изменений в блок данных, совместно используемый несколькими объектами, приводит к созданию копии этого блока, внесению в него изменений и передаче этого блока в монопольное владение изменившим его объектом.

Как видно из приведенного выше примера, синтаксис обращения к элементам объекта класса QArray и производных от него классов ничем не отличается от обращения к обычным массивам. Главное отличие заключается в объявлении объектов классов, использующих для задания аргументов не квадратные, а круглые скобки, в которые можно помещать любое выражение, возвращающее целочисленное значение. Следует отметить, что класс QArray позволяет создавать только одномерные массивы.

Класс QVector, являющийся потомком класса QGVector, используется для хранения массива указателей на объекты. Например, его можно использовать для создания многомерных массивов изменяемой размерности. В качестве аргумента шаблона класса передается тип его объектов, а не тип указателей на объекты, которые, собственно, и хранятся в этом массиве. Для сравнения элементов массива вызывается виртуальная функция QGVector::compareItems, по умолчанию сравнивающая значения указателей на объекты, хранящиеся в массиве.

Внимание!

Для помещения указателей на объекты в массив объекта класса `QVector` следует применять функцию `QVector::insert`, в первом аргументе которой передается индекс изменяемого элемента, а во втором — указатель на добавляемый объект. Использовать для этой цели операцию индексирования `QVector::operator[]` бессмысленно, поскольку она только возвращает значение указателя, хранящегося по указанному индексу.

В листинге 9.10 приведен пример использования класса `QVector` для создания треугольного массива.

Листинг 9.10. Использование класса QVector

```
QVector< QArray<int> > qv(3);

qv.insert(0, new QArray<int>(1));
qv.insert(1, new QArray<int>(2));
qv.insert(2, new QArray<int>(3));

qv[0][0] = 10;
```

Внимание!

Среди функций класса `QVector` имеется функция `QVector::remove`, удаляющая объект из списка. Пользоваться этой функцией нужно с особой осторожностью, поскольку она не производит уплотнение массива, а только записывает в соответствующий его элемент нулевой указатель. Кроме того, после вызова этой функции функция `QVector::count` уменьшает возвращаемое ей значение на единицу, указывая на число ненулевых элементов в массиве. Таким образом, возвращаемое этой функцией значение теперь не может быть использовано для оценки размеров массива.

Связные списки

Классы связных списков используются для создания различных списков, для каждого элемента которых, кроме первого и последнего, однозначно определены предшествующий и последующий элементы. Первый элемент связного списка не имеет предшествующего ему элемента, а последний — последующего. Для доступа к связным спискам в их классы включаются функция для доступа к их первому и последнему элементам, а также функции перехода к предыдущему и последующему элементу списка. Для поиска требуемого элемента списка нужно сначала выбрать направление поиска, затем, в зависимости от выбранного направления поиска, получить доступ к первому или к последнему элементу списка и произвести последовательный их перебор, пока среди элементов списка не будет обнаружен искомый.

Так же, как и в случае массивов, существуют классы списков для хранения объектов и для хранения указателей на объекты. Для создания списков объектов применяются класс `QValueList` и производный от него класс `QValueStack`, используемый для хранения объектов в стеке. Эти классы могут использоваться для хранения значений простых типов или объектов классов, имеющих копирующий конструктор, переопределенную операцию присваивания и конструктор без аргументов. Поскольку объекты класса `QObject` и всех производных от него классов не удовлетворяют поставленным требованиям, объекты большинства классов библиотеки `Qt`, в том числе объекты всех оконных классов, не могут храниться в объекте класса `QValueList`.

Так как классический поиск в связном списке производится очень медленно, к объекту класса `QValueList` одновременно предоставляется доступ как к массиву, что существенно ускоряет обращение к нему. В листинге 9.11 приведен пример использования различных методов поиска информации в объекте класса `QValueList`.

Листинг 9.11. Поиск информации в объекте

```
/** создание объекта списка */
 QValueList<int> qvl;

/** заполнение списка */
 qvl.append( 10);
 qvl.append( 20);
 qvl += 30;
 qvl << 40;
 qvl << 50;

/** поиск в списке как в массиве */
int k = 30;

 for(uint i=0; i < qvl.count(); i++)
 {
  if(qvl[i] == k)
   cout << "Found in array" << endl;
 }

/** использование итератора */
 QValueList<int>::Iterator it;

 for( it = qvl.begin(); it != qvl.end(); it++)
 {
  if((*it) == k)
   cout << "Found in list" << endl;
 }
```

```
/** использование специальных функций поиска */
 it = qvl.begin();

 it = qvl.find( it, k);

 if( it != qvl.end())
  cout << "Found with function" << endl;

 cout << "Contains " << qvl.contains( k) << "times" << endl;
```

Для заполнения списка в рассматриваемом примере использована функция QValueList::append, добавляющая элемент в конец списка и возвращающая итератор, указывающий на добавленный объект (что такое итератор будет объяснено позже), и эквивалентные ей перегруженные операции += и <<. Помимо этой функции для добавления элементов в список могут использоваться функция QValueList::prepend, добавляющая новый элемент в начало списка, и функция QValueList::insert, добавляющая его перед элементом, на который указывает итератор, передаваемый в первом ее аргументе.

После заполнения списка в нем производится поиск как в обычном массиве. Для определения размеров списка вызывается функция QValueList::count, а для доступа к элементу списка используется перегруженная операция индексации.

Для доступа к элемента списка как к таковым используется специальный шаблон класса QValueListIterator. Для простоты создания объектов этого класса в заголовке шаблона класса QValueList производится переопределение шаблона класса QValueListIterator к типу Iterator. Поэтому создание объекта класса итератора не представляет никакой сложности.

Для установки итератора на первый элемент связного списка используется функция QValueList::begin, а для перевода итератора на следующий элемент списка — постфиксная операция инкрементирования QValueListIterator::operator++(int). Условием завершения поиска является равенство полученного итератора значению, возвращаемому функцией QValueList::end.

Примечание

Вопреки утверждению Microsoft о невозможности раздельной перегрузки постфиксной и префиксной операции инкрементирования и декрементирования, в библиотеке Qt производится такая перегрузка, и в классе QValueListIterator помимо постфиксной операции существует и префиксная операция инкрементирования QValueListIterator::operator++().

Поиск в списке мог бы быть проведен и с его конца. Для этого итератор устанавливается на последний элемент списка вызовом функции QValueList::fromLast, а перевод итератора на предыдущий элемент списка

осуществляется префиксной и постфиксной операцией декрементирования QValueListIterator::operator--() и QValueListIterator::operator--(int).

Поиск с конца списка лучше всего производить в том случае, если есть полная уверенность в том, что искомый элемент не содержится в первом элементе списка. Дело в том, что при поиске с начала файла для прекращения поиска полученный итератор сравнивается со значением, возвращаемым функцией QValueList::end. Это значение соответствует положению итератора за последним элементом списка. При поиске с конца файла поиск приходится прекращать при совпадении полученного итератора с возвращаемым значением функции QValueList::begin, соответствующим положению итератора на первом элементе списка. Таким образом, этот элемент не будет просмотрен.

Несколько искусственно эту проблему можно разрешить следующим образом: использовать для начальной установки итератора функцию QValueList::end и производить декремент итератора в первом же операторе цикла поиска, но при этом следует помнить о том, что такой алгоритм некорректно работает с пустыми списками, когда функции QValueList::begin и QValueList::end возвращают одно и то же значение.

Для поиска объекта может быть использована и специальная функция QValueList::find, производящая поиск с текущей позиции итератора и возвращающая значение итератора, указывающего на найденный объект, или значение, равное возвращаемому значению функции QValueList::end, если искомый объект не найден.

Для поиска первого вхождения объекта в список может быть использована и функция QValueList::findIndex, возвращающая его индекс или значение −1, если искомый объект не найден. Для подсчета числа вхождений объекта в список используется функция QValueList::contains.

Работа с объектом класса коллекции QList, используемым для создания списков ссылок на объекты, во многом аналогична работе с объектом класса QValueList, непосредственно хранящим сами объекты. Основное различие заключается в том, что в классе QList имеются два набора функций: один для работы с указателями на объекты, а другой — для работы с самими объектами.

Для просмотра объектов класса QList используется объект класса итератора QListIterator, однако при работе с ним отсутствуют удобства, предоставляемые в классе QValueList, и пользователю приходится самому следить за корректностью передаваемых аргументов шаблонов. Кроме того, в этом классе отсутствуют постфиксные операции инкрементирования и декрементирования, что не совсем удобно для разработчиков, отвыкших от использования префиксных операций.

В листинге 9.12 приведен пример использования классов QList и QListIterator.

Листинг 9.12. Пример использования классов QList и QListIterator

```
/** Демонстрационный класс */
class someClass
{
 public:

  someClass( int i) { n = i;}

  int Get()      {return n;}

 private:

  int n;
};

void main()
{
 /** Создание объекта списка */
 QList<someClass> ql;

/** Установка режима удаления элементов списка */
 ql.setAutoDelete( TRUE);

/** Заполнение списка */
 ql.append( new someClass( 10));
 ql.append( new someClass( 20));
 ql.append( new someClass( 30));
 ql.append( new someClass( 40));
 ql.append( new someClass( 50));

/** Поиск в списке по индексу */
 int k = 30;

 for(uint i=0; i < ql.count(); i++)
 {
  if( ql.at(i)-> Get() == k)
   cout << "Found in array" << endl;
 }

/** Поиск по итератору */
 QListIterator<someClass> it(ql);

 for (; it.current(); ++it)
 {
  if( it.current()-> Get() == k)
   cout << "Found in list" << endl;
 }
}
```

После создания объекта класса `QList` для него сразу же вызывается функция `QCollection::setAutoDelete`, устанавливающая режим автоматического уничтожения объектов, на которые указывают элементы списка, при уничтожении объекта самого списка. Затем с использованием функции `QList::append` производится заполнение списка.

Хотя в классе `QList` нельзя использовать синтаксис обращения к массиву, поскольку в нем не перегружена операция индексирования, каждому элементу списка сопоставлен целочисленный индекс, который может служить для доступа к объекту посредством функции `QList::at`, возвращающей указатель на объект. Одноименная функция существует и в классе `QValueList`, однако она возвращает итератор, установленный на указанный элемент списка.

Для поиска в списке создается объект класса итератора `QListIterator`. Поскольку при инициализации данного объекта он автоматически устанавливается на начало списка, при инициализации цикла поиска нет необходимости дополнительно вызывать функцию `QListIterator::toFirst` для выполнения этой операции. Переход к следующему элементу списка, как и в рассмотренном ранее примере, производится операцией инкрементирования, но за неимением постфиксной операции используется префиксная операция инкрементирования. Для определения условия завершения цикла вызывается функция `QList::current`, возвращающая указатель на текущий объект или нулевой указатель, если поиск вышел за пределы списка.

Внимание!

При работе с объектами класса `QList` следует помнить о том, что нулевой указатель разрывает список. То есть вместо одного списка появляются два списка, рассматриваемые приложением как один.

Использование разнообразных функций поиска в классе `QList` несколько затруднено тем, что они производят поиск указателя на объект, а пользователя в большинстве случаев интересует поиск по содержимому объекта.

При рассмотрении класса `QVector` были отмечены серьезные проблемы, возникающие при вызове функции `QVector::remove`. В классе `QList` имеется одноименная функция, позволяющая уничтожать текущий объект, объект, расположенный по указанному индексу, или первое вхождение указанного объекта. Эти функции работают корректно, поскольку для исключения элемента из связного списка достаточно замкнуть друг на друга предшествующий и последующий элементы. Помимо функции `QList::remove` в этот класс включены специальные функции `QList::removeFirst` и `QList::removeLast`, уничтожающие первый и последний элементы списка соответственно.

10 Зак. 113

Карты отображений

Классы карт отображений, называемых еще словарями, используются для хранения различных объектов, доступ к которым осуществляется по ключам. В качестве ключа может быть использован объект практически любого класса, для которого определены операции сравнения на равенство (operator==) и операции упорядочения (operator<, operator<=, operator> и operator>=). Определение в объекте ключа операторов упорядочения позволяет существенно повысить скорость поиска в объекте карты отображений, что позволяет использовать ее даже для кэширования информации.

Карты отображений можно рассматривать как связные списки, имеющие индексы произвольного типа. Поэтому многие способы доступа к информации в картах отображения аналогичны способам доступа в связных списках.

Рассмотрим сначала класс QMap, используемый для непосредственного хранения значений простых типов и объектов, к которым предъявляются те же требования, что и к объектам, хранящимся в объектах классов QArray и QValueList.

Обратимся к реализации различных способов поиска информации в объекте класса QMap, как это показано в листинге 9.13.

Листинг 9.13. Поиск информации в объекте класса QMap

```
QMap<QString,QString> qm;

qm.insert( "Allen", "Paul");
qm.insert( "Ballmer", "Steve");
qm.insert( "Gates", "Bill");
qm.insert( "Sommer", "Ron");

QString sz = "Ballmer";

QMap<QString,QString>::Iterator it;

for (it = qm.begin(); it != qm.end(); it++)
{
 if( it.data() == "Paul")
  cout << "Paul found. He is " << it.key() << endl;

 if( it.key() == sz)
  cout << sz << " is " << qm[it.key()] << endl;
}

it = qm.begin();

it = qm.find( sz);
```

```
if( it != qm.end())
  cout << "Found with function" << endl;

if( qm.contains(sz));
  cout << "Contains " << sz << endl;
```

Использование в качестве ключей объектов классов существенно ограничивает возможные методы поиска информации в картах отображений. Хотя в них можно использовать синтаксис доступа к массиву, употребляя в качестве индекса значение ключа, в большинстве случаев достаточно сложно организовать полный перебор значений ключей для проведения поиска. Кроме того, при применении данного синтаксиса нужно быть абсолютно уверенным в том, что указанный ключ определен в данной карте отображения. Иначе будет создана пустая запись с этим ключом. Поэтому, как уже говорилось выше, методы просмотра карт отображений аналогичны методам просмотра связных списков.

Для просмотра объекта класса `QMap` используется объект класса `QMapIterator`, во многом аналогичный объекту класса `QValueListIterator`, применяемого для просмотра объекта класса `QValueList`. Шаблон этого класса переопределен в классе `QMap` к типу `Iterator` с указанием типа хранящихся в карте отображений объектов.

Так же, как и при работе с объектом класса `QValueList`, для установки итератора на первый объект карты отображений используется функция `QMap::begin`, для проверки завершения операции поиска — функция `QMap::end`, а для перехода к следующему элементу карты — перегруженная операция инкрементирования. В классе `QMap` отсутствует возможность непосредственного перехода к последнему элементу карты отображения, но в классе `QMapIterator` сохранен полный набор префиксных и постфиксных операций инкрементирования и декрементирования.

Поскольку в рассматриваемом примере используется простейший класс хранящегося объекта, поиск в нем производится не только по ключу, но и по содержимому хранимой записи. Для доступа к данным, хранящимся в элементе, на который указывает итератор, применяется функция `QMapIterator::data`, а для доступа к ключу этого элемента — функция `QMapIterator::key`. Для получения содержимого записи по ключу используется перегруженная операция индексирования `QMap::operator[]`.

Поскольку поиск в карте сообщений производится по ключам, их значения должны быть уникальными, поэтому функция `QMap::find` возвращает итератор единственной записи, имеющей указанный ключ, или значение, возвращаемое функцией `QMap::end`, если указанный ключ не определен в карте отображений. По той же причине функция `QMap::contains` возвращает не число вхождений указанного ключа, а логическое значение, указывающее на его наличие в данной карте отображений.

Классы карт отображения, применяемые для хранения указателей на объекты, являются потомками класса QGDict и отличаются друг от друга типом используемых в них объектов ключей. К ним относятся следующие классы:

- QAsciiDict — использует ключи типа char*;
- QDict — использует в качестве ключей объекты типа QString;
- QIntDict — использует ключи типа long;
- QPtrDict — использует ключи типа void*.

В остальном эти классы и используемые ими функции практически эквивалентны. Для демонстрации принципов работы с этими классами преобразуем рассмотренный выше пример для использования в нем объекта класса QDict, как это показано в листинге 9.14.

Листинг 9.14. Использование объекта класса QDict

```
QDict<QString> qd(7, false);

qd.setAutoDelete( true);

qd.insert( "Allen", new QString("Paul"));
qd.insert( "Ballmer", new QString("Steve"));
qd.insert( "Gates", new QString("Bill"));
qd.insert( "Sommer", new QString("Ron"));

QString sz = "Ballmer";

QDictIterator<QString> it(qd);

for (; it.current(); ++it)
{
 if( *it.current() == "Paul")
  cout << "Paul found. He is " << it.currentKey() << endl;

 if( it.currentKey() == sz)
  cout << sz << " is " << qd[it.currentKey()] << endl;
}

if( qd.find( sz))
 cout << "Found with function" << endl;
```

В отличие от конструктора класса QMap, не имеющего аргументов, конструктор класса QDict имеет два аргумента, правда значения этих аргументов могут подставляться по умолчанию. В первом аргументе конструктора класса QDict передается размер создаваемой карты отображений. По умолчанию новая карта отображений содержит 17 элементов. Во втором аргументе конструктора передается флаг, определяющий необходимость учета регистра

символов при сравнении ключей. По умолчанию эта проверка производится, мы же от нее отказались.

Для того чтобы впоследствии не заботиться о созданных нами объектах, включенных в карту отображений, сразу же после создания объекта класса `QDict` нами вызывается функция `QCollection::setAutoDelete` с аргументом `true`. И только после этого производится заполнение карты отображений функциями `QDict::insert`. Если в этой функции указан уже существующий ключ, то уже существующие записи с этим ключом становятся недоступными. Эти записи помещаются в стек, из которого они последовательно извлекаются при вызове функции `QDict::remove`.

Внимание!

Работая с классами коллекций, хранящих указатели на объекты, следует быть особо осторожными при помещении в них указателей на простые типы. Например, при замене в рассматриваемом примере класса хранящегося объекта `QString` на тип `char` вызов функции `QDict::remove` приведет к аварийному завершению приложения.

Для поиска записи по ключу создается объект класса итератора `QDictIterator`. Его конструктору в качестве аргумента передается объект класса `QDict`, в котором будет производиться поиск. Процедура перебора элементов карты отображений в объекте класса `QDict` организована аналогично рассмотренной выше процедуре перебора элементов связного списка в объекте класса `QList`. Так же, как и в случае связного списка, при своем создании объект итератора устанавливается на первую запись карты отображений, поэтому для него нет необходимости вызывать функцию `QDictIterator::toFirst`, выполняющую эту операцию.

Примечание

В классе `QDictIterator` реализован очень ограниченный набор функций. Поскольку в нем нет возможности получить доступ к последнему элементу карты отображений, в нем не реализованы операции декрементирования. Поэтому единственной возможностью перейти к другому элементу карты отображений является операция префиксного инкрементирования.

Для получения указателя на объект данных, хранящийся в элементе, на который указывает итератор, используется функция `QDictIterator::current`, а для доступа к ключу этого элемента — функция `QDictIterator::currentKey`. Для получения содержимого записи по ключу используется перегруженная операция индексирования `QDict::operator[]`. Необходимо помнить о том, что функция `current` возвращает указатель на объект, и использовать операцию `*` для получения доступа к самому объекту.

Поскольку в карте сообщений доступен только один элемент с указанным значением ключа, то в классе `QDict` отсутствует функция `contains`. В него

включена только функция QDict::find. Для повышения эффективности работы этой функции при добавлении объекта по ключу значение ключа преобразуется (хэшируется) в целочисленный индекс массива. При поиске объекта значение ключа опять преобразуется в индекс массива и уже по нему производится поиск. Для повышения эффективности поиска размер карты отображений должен быть простым числом.

Другие классы коллекций

Каждый из описанных выше классов коллекций имеет свою область применения. Некоторые задачи из этих областей встречаются так часто, что для их решения в библиотеке Qt были созданы специализированные классы коллекций. Формально эти классы являются независимыми, но при их рассмотрении легко обнаружить, какой из базовых классов коллекции использован в качестве их основы.

Классы QQueue и QStack базируются на связных списках. Класс QQueue реализует очередь указателей на объекты (первым пришел — первым ушел), а класс QStack — стек указателей на объекты (последним пришел — первым ушел). Эти классы позволяют производить очень ограниченный набор операций с формируемой ими очередью, что дает возможность существенно повысить эффективность реализации этих функций. Рассмотрим использование основных функций этих классов на примере, приведенном в листинге 9.15.

Листинг 9.15. Классы QQueue и QStack

```
QQueue<QString> qq;
QStack<QString> qs;

qq.setAutoDelete( true);
qs.setAutoDelete( true);

qq.enqueue(new QString("First"));
qq.enqueue(new QString("Second"));
qq.enqueue(new QString("Third"));

qs.push(new QString("First"));
qs.push(new QString("Second"));
qs.push(new QString("Third"));

if( *qq.head() == "First")
 cout << "QQueue::head is OK" << endl;

if( *qq.dequeue() == "First")
 cout << "QQueue::dequeue is OK" << endl;
```

```
if( *qq.current() == "Second")
 cout << "QQueue::current is OK" << endl;

qq.clear();

if( qq.isEmpty())
 cout << "QQueue::clear is OK" << endl;

if( !qq.count())
 cout << "QQueue::count is OK" << endl;

if( *qs.top() == "Third")
 cout << "QStack::top is OK" << endl;

if( *qs.pop() == "Third")
 cout << "QStack::pop is OK" << endl;

if( *qs.current() == "Second")
 cout << "QStack::current is OK" << endl;

qs.clear();

if( qs.isEmpty())
 cout << "QStack::clear is OK" << endl;

if( !qs.count())
 cout << "QStack::count is OK" << endl;
```

После создания объектов классов QQueue и QStack для каждого из них вызывается функция QCollection::setAutoDelete для автоматического уничтожения извлекаемых из них элементов. Чтобы поместить элемент в объект класса QQueue, используется функция QQueue::enqueue, а для выполнения той же операции в объекте класса QStack применяется функция QStack::push. Для наглядности в оба объекта помещаются одинаковые элементы в одном и том же порядке.

С целью получения указателя на объект, находящийся в голове списка, в классе QQueue используется функция QQueue::head. Тот же самый результат можно получить и вызвав функцию QQueue::current. При вызове любой из этих функций содержимое очереди не меняется. Функция QQueue::dequeue также возвращает указатель на объект, находящийся первым в очереди, но при этом удаляет его из очереди. Виртуальная функция QQueue::clear удаляет все элементы из очереди и уничтожает все связанные с ними объекты, если для объекта класса QQueue установлен режим автоматического удаления объектов. В том, что очередь действительно пуста, можно убедиться, вызвав функцию QQueue::isEmpty или проверив размер очереди, возвращаемый функцией QQueue::count.

Для получения указателя на объект, находящийся в вершине стека, в классе QStack используется функция QStack::top. Тот же самый результат можно получить и вызвав функцию QStack::current. При вызове любой из этих функций содержимое стека не меняется. Функция QStack::pop также возвращает указатель на объект, находящийся в вершине стека, но при этом удаляет его из очереди. Виртуальная функция QStack::clear удаляет все элементы из стека и уничтожает все связанные с ними объекты, если для объекта класса QStack установлен режим автоматического удаления объектов. В том, что стек действительно пуст, можно убедиться, вызвав функцию QStack::isEmpty или проверив размер стека, возвращаемый функцией QStack::count.

Если классы QQueue и QStack являются упрощенными классами связных списков, то классы QAsciiCache, QCache и QIntCache — усложненными классами карт отображений, поскольку в них, в отличие от обычных классов карт отображений, реализована стратегия автоматического удаления объектов. С этой целью при добавлении в коллекцию каждого нового элемента для него устанавливается вес или цена (cost). Как только сумма весов всех элементов коллекции превысит заданный предел, из нее удаляется элемент, к которому дольше всего не было обращений.

Классы QAsciiCache, QCache и QIntCache отличаются друг от друга только типом используемых в них ключей: класс QAsciiCache использует ключи типа char*, класс QCache — ключи типа QString, а класс QIntCache — типа long. В остальном эти классы практически эквивалентны. Для простоты рассмотрим работу с этими классами на примере класса QIntCache, приведенного в листинге 9.16.

Листинг 9.16. Пример класса QIntCache

```
int i;
QIntCache<QString> qic;

qic.setAutoDelete( true);

qic.setMaxCost( 20);

qic.insert( 0, new QString("First"), 5);
qic.insert( 1, new QString("Second"), 5);
qic.insert( 2, new QString("Third"), 5);
qic.insert( 3, new QString("Fourth"), 5);

cout << "Before insertion" << endl << endl;

if( qic.find( 0))
  cout << "Item 0 exists" << endl;
```

```
qic.insert( 4, new QString("Fifth"), 7);

cout << endl << "After insertion" << endl << endl;

for(i=0; i < 5; i++)
{
  if( qic.find( i))
   cout << "Item " << i << " exists" << endl;
}
```

В результате выполнения приведенной программы на экран будет выведен текст:

```
Before insertion

Item 0 exists

After insertion

Item 0 exists
Item 3 exists
Item 4 exists
```

Поскольку работа с объектами классов карт отображений была подробно рассмотрена ранее, в данном примере основное внимание сконцентрировано на демонстрации стратегии автоматического удаления объектов.

Конструктор класса `QIntCache` имеет два аргумента, в первом из которых передается максимальный суммарный вес элементов создаваемого объекта класса, а во втором — число этих элементов в объекте. В рассматриваемом примере использованы значения данных аргументов по умолчанию: суммарный вес 17 элементов объекта класса не может превышать 100.

Для автоматического удаления объектов в приложении вызывается функция `QCollection::setAutoDelete` и производится снижение максимального суммарного веса вызовом функции `QIntCache::setMaxCost`. Максимальный вес можно было бы и не снижать, а просто использовать большие веса, но так интереснее.

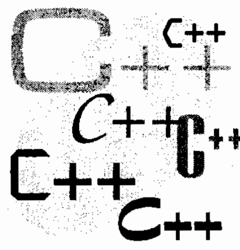
После этого в объект класса `QIntCache` с использованием функции `QIntCache::insert` добавляется четыре элемента и их веса выбираются таким образом, чтобы их сумма в точности равнялась бы значению порога. Произведенная после этого проверка показывает, что все добавленные элементы хранятся в объекте.

После проверки наличия нулевого элемента в объекте в него добавляется новый элемент, имеющий больший вес, чем предыдущие. Проведенная после этого проверка показывает, что при этом из объекта удаляются элементы с индексами 1 и 2, а элемент с индексом 0 остается. Это объясняется тем,

что из объекта класса QIntCache удаляются не дольше всего хранящиеся в нем элементы, а дольше всего не использующиеся. Причем удаление элементов продолжается до тех пор, пока суммарный вес оставшихся элементов не будет меньшим либо равным порогу.

Использовавшаяся для поиска указателя на объект по его индексу функция QIntCache::find имеет и второй аргумент, по умолчанию принимающий значение true. Если бы этот аргумент имел при первом вызове этой функции значение false, то при добавлении нового элемента из объекта класса QIntCache были бы удалены элементы с индексами 0 и 1, поскольку в этом случае поиск элемента не считался бы его использованием.

ГЛАВА 10



Реализация многозадачности в приложении

Особой гордостью приверженцев операционной системы UNIX (а значит и построенной на ее основе операционной системы Linux) является истинная многозадачность данной операционной системой, поскольку она допускает одновременное исполнение нескольких процессов. Под *процессом* (process) можно понимать исполняемый на компьютере код программы или отдельного фрагмента программы.

Процессы являюся практически полностью независимыми друг от друга и ошибки, вызвавшие крах одного процесса, практически никак не влияют на исполнение других процессов, если между этими процессами не было организовано взаимодействие. Однако даже и в этом случае крах взаимодействующего процесса приведет только к потере источника информации для другого процесса. Такая структура выполнения вычислений обеспечивает очень высокую надежность операционной системы.

Исполняемые на компьютере процессы могут быть независимыми или взаимосвязанными. Для обеспечения связи процессов операционная система Linux предоставляет средства порождения одного процесса другим, завершения процесса из другого процесса, синхронизации работы процессов и реакции в одном процессе на событие, произошедшее в другом процессе.

Взаимодействие процессов осуществляется в стиле пользовательского интерфейса, т. е. порожденный процесс не может определить, инициирован он пользователем или другим процессом, а просто получает информацию через стандартный поток ввода `stdin`, возвращает обработанную информацию в стандартный поток вывода `stdout` и помещает сообщения об ошибках в стандартный поток `stderr`. Такая организация взаимодействия потоков существенно облегчает использование в новых программах возможностей, реализованных в уже существующих консольных приложениях.

В настоящее время практически все создаваемые приложения имеют графический интерфейс, и, казалось бы, про консольные приложения можно уже

и забыть. Но, если подумать, наличие графического интерфейса в дочернем процессе только мешает его использованию, поскольку графический интерфейс приложения должен быть выполнен в едином стиле, а этого трудно добиться при объединении в нем компонентов, полученных из различных источников.

Исходя из вышеизложенного, при создании приложений, компоненты которых предполагается использовать в других приложениях, целесообразно применять многозадачность. То есть создать отдельный процесс, являющийся главным процессом приложения, в котором будет реализован весь его графический интерфейс, а всю обработку производить в дочерних процессах, не имеющих графического интерфейса, т. е. реализованных как консольные приложения.

Помимо облегчения повторного использования компонентов приложения, применение многозадачности позволяет существенно повысить быстродействие приложения, выделяя длительные операции обработок или работы с каким-либо ресурсом в отдельный процесс, позволяя тем самым основному процессу заняться решением других задач.

Взаимодействие процессов

Для демонстрации совместной работы процессов в приложении рассмотрим демонстрационные приложения, текст которых можно найти на прилагаемом к книге CD.

Приложение или процесс, запрашивающий услугу у некоторого другого процесса или компонента, принято называть *клиентом* (client), а процесс или компонент, оказывающий эту услугу, — *сервером* (server). Естественно, что для демонстрации взаимодействия клиента и сервера необходимо создать оба приложения. Для начала нами будет создан приемник для простейшего сервера, в качестве которого будет выступать заготовка консольного приложения.

Создание клиента для простейшего сервера

Хотя создание приложения сервера не должно вызвать никаких проблем, для полноты оно также помещено на прилагаемый к книге CD.

Чтобы создать приложение сервера:

1. Выберите команду меню **Project | New** (Проект | Новый). Появится диалоговое окно **ApplicationWizard** (Мастер создания приложений).
2. В окне иерархического списка раскройте каталог **Terminal** (Консольные приложения), выделите в нем строку **C++** и нажмите кнопку **Next** (Далее). Раскроется вторая вкладка мастера создания приложений.

3. В текстовое поле **Project name** (Имя проекта) введите имя приложения **SimpleServer** и нажмите кнопку **Create** (Создать).
4. По завершении процесса создания приложения нажмите кнопку **Exit** (Выход). Будет создана заготовка консольного приложения.
5. Выберите команду меню **Build | Make** (Создать | Компилировать), нажмите клавишу <F8> или кнопку **Make** в панели инструментов. Приложение сервера будет откомпилировано.
6. Переместите созданный исполняемый файл в каталог \usr\bin.

Внимание!

Для перемещения файла в папку \usr\bin вам необходимо иметь права системного администратора. Поскольку отлаживать программы, имея такие широкие права, решаются только самые отважные разработчики, для получения этих прав вам, скорее всего, придется закрыть текущий сеанс и перейти в сеанс системного администратора (с учетной записью root).

Поскольку мастер создания приложений уже помещает в создаваемую им заготовку консольного приложения операцию вывода текста в стандартный поток `stdout`, нам нет никакой необходимости вносить изменения в заготовку сервера. Поэтому мы сразу можем перейти к созданию заготовки приложения клиента, текст которого помещен на прилагаемом к книге CD.

Чтобы создать приложение клиента для созданного нами сервера:

1. Выберите команду меню **Project | New**. Появится диалоговое окно **ApplicationWizard**.
2. В окне иерархического списка оставьте выделенной строку **KDE Normal** (Однооконное приложение KDE) и нажмите кнопку **Next**. Раскроется вторая вкладка мастера создания приложений.
3. В текстовое поле **Project name** введите имя приложения **SimpleClient** и нажмите кнопку **Create**.
4. По завершении процесса создания приложения нажмите кнопку **Exit**. Будет создана заготовка однооконного приложения KDE.
5. В окне иерархических списков раскройте вкладку **Classes** (Классы), щелкните правой кнопкой мыши по имени класса `SimpleClientApp` и выберите в появившемся контекстном меню команду **Add slot** (Добавить приемник). Появится диалоговое окно **Class Properties** (Свойства класса), раскрытое на вкладке **Slots** (Приемники).
6. В текстовое поле **Declaration** (Сигнатура) введите сигнатуру приемника `slotDataReceived(KProcess*,char*,int)`, в текстовое поле **Documentation** (Документация) — комментарий `Receives data from process` (Получает данные из процесса) и нажмите кнопку **Apply** (Применить). В класс будет добавлен новый приемник.

7. Повторите пункты 5 и 6 для добавления в класс `SimpleClientApp` приемника `slotProcessExited(KProcess*)`, снабдив его комментарием `Receives process terminate signal` (Получает сигнал о завершении процесса).
8. В открывшемся после добавления приемников файле simpleclient.cpp измените эти приемники в соответствии с текстом листинга 10.1.

Листинг 10.1. Приемники класса SimpleClientApp

```
/** Получает данные из процесса */
void SimpleClientApp::slotDataReceived(KProcess* proc,char* buf,int len)
{
  QString sz(buf);

  sz.truncate( len);

  view-> insertLine( sz);
}

/** Получает сигнал о завершении процесса */
void SimpleClientApp::slotProcessExited(KProcess* proc)
{
 QString sz;

 sz.sprintf("Process %d exited!", (int)proc->getPid());

 view-> insertLine( sz);
}
```

9. Измените функцию `SimpleClientApp::initView` в соответствии с текстом листинга 10.2.

Листинг 10.2. Функция initView

```
void SimpleClientApp::initView()
{
 ////////////////////////////////////////////////////////////////////
 // Создайте здесь главное окно приложения, которое будет управляться
 // объектом класса KTMainWindow, и свяжите это окно с объектом класса
 // документа для вывода его содержимого на экран.

 view = new SimpleClientView(this);
 doc->addView(view);
 setCentralWidget(view);
 setCaption(doc->URL().fileName(),false);

 proc << "simpleserver";
```

```
    connect(&proc, SIGNAL(processExited(KProcess *)),
                   this, SLOT(slotProcessExited(KProcess *)));

    connect(&proc, SIGNAL(receivedStdout(KProcess *, char *, int)),
            this, SLOT(slotDataReceived(KProcess *, char *, int)));

    connect(&proc, SIGNAL(receivedStderr(KProcess *, char *, int)),
            this, SLOT(slotDataReceived(KProcess *, char *, int)));

    proc.start(KProcess::NotifyOnExit, KProcess::AllOutput);
}
```

10. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши на имени класса `SimpleClientApp` и выберите в появившемся контекстном меню команду **Add member variable** (Добавить переменную члена класса). Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes** (Атрибуты).
11. В текстовое поле **Type** (Тип) введите тип переменной `KProcess`, в текстовое поле **Name** (Идентификатор) — идентификатор переменной `proc`, в группе **Access** (Модификатор прав доступа) установите переключатель **Private** (Закрытая), в текстовое поле **Documentation** введите комментарий `Process object` (Объект класса процесса) и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.
12. В открывшемся после добавления переменной окне редактирования файла simpleclient.h после строки `#include <kurl.h>` поместите строку

    ```
    #include <kprocess.h>
    ```

13. Во вкладке **Classes** окна иерархических списков щелкните левой кнопкой мыши на имени класса `SimpleClientView`. Откроется окно редактирования файла simpleclientview.h.
14. В начале этого файла замените строку `#include <qwidget.h>` строкой

    ```
    #include <keditcl.h>
    ```

15. В заголовке класса `SimpleClientView` замените строку `class SimpleClientView : public QWidget` строкой

    ```
    class SimpleClientView : public KEdit
    ```

16. Щелкните правой кнопкой мыши в окне редактирования файла simpleclientview.h и выберите в появившемся контекстном меню команду **Switch Header/Source** (Переключить файлы заголовка и реализации). Откроется окно редактирования файла simpleclientview.cpp.
17. Измените конструктор класса `SimpleClientView` в соответствии с текстом листинга 10.3.

Листинг 10.3. Конструктор класса SimpleClientView

```
SimpleClientView::SimpleClientView(QWidget *parent, const char *name)
  : KEdit(parent, name)
{
 setBackgroundMode(PaletteBase);
}
```

18. Выберите команду меню **Build | Execute**, нажмите клавишу <F9> или кнопку **Run** в панели инструментов. Появится окно приложения, изображенное на рис. 10.1.

Внимание!

При работе в отладочном режиме клиент не может общаться с приемником, используя стандартные потоки ввода/вывода. Поэтому в случае применения для запуска рассматриваемого приложения команды меню **Debug | Start** или кнопки **Debug** в панели инструментов клиент получит только сообщение о закрытии серверного процесса.

19. Закройте приложение.

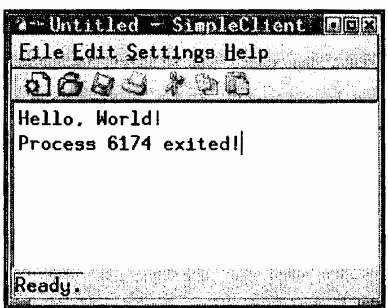


Рис. 10.1. Окно приложения клиента

Как уже говорилось выше, в качестве серверного приложения нами использовалась заготовка консольного приложения, помещающая в стандартный поток вывода сакраментальный текст "Hello, World!". Текст, конечно, изрядно приелся, но менять его в демонстрационном приложении нет никакого смысла. Для того чтобы приложение клиента могло без труда обнаружить свой сервер, он помещен в каталог \usr\bin, куда помещаются все стандартные серверы.

В качестве клиента созданного нами сервера можно было бы также использовать консольное приложение, но среда разработки KDevelop не включает в заготовку консольного приложения поддержку библиотеки KDE, поэтому, чтобы не включать эту поддержку самостоятельно, в качестве заготовки приложения клиента нами выбрано однооконное приложение KDE.

Поскольку сервер помещает в поток текстовую информацию и вывод текстовых сообщений является простой и понятной формой организации пользовательского интерфейса создаваемого приложения клиента (цель которого убедить пользователя в том, что все работает), это приложение реализовано как простейший текстовый редактор KDE, подробно описанный в *главе 8*. Для того чтобы превратить заготовку приложения в текстовый редактор, нам было достаточно изменить базовый класс для класса представления, что мы и сделали.

Так как приемники сигналов сервера работают с объектом класса представления, вызывать сервер можно только после того, как в приложении будет создан объект класса представления. Этот объект создается в функции `SimpleClientApp::initView`, поэтому в эту же функцию помещаются операции инициализации объекта класса `KProcess` и вызова сервера. Объект класса `KProcess` представляет собой специальный объект, используемый приложением для вызова сервера и взаимодействия с ним. Для этого в данный класс включены специальные функции, сигналы и приемники, использование которых позволяет предельно упростить процедуру взаимодействия клиента и сервера.

Для вызова сервера необходимо указать его имя и передать ему значения аргументов командной строки, если таковые имеются. С этой целью рекомендуется использовать перегруженный оператор << (`KProcess::operator<<`). Этот оператор может использоваться неоднократно, причем каждый его аргумент заносится в отдельную запись списка аргументов сервера. В голове этого списка (в его первой записи) находится имя исполняемого файла, содержащего реализацию вызываемого сервера, а в остальных записях — его аргументы. Поскольку наш сервер не анализирует переданные ему при запуске аргументы командной строки, то нет никакой необходимости их указывать. Поэтому в рассматриваемом приложении в список аргументов помещается только имя его исполняемого файла.

Используемый нами сервер помещает текст в стандартный поток вывода, поэтому в приложении приемника должны быть предусмотрены средства для извлечения этой информации. Для извещения о том, что сервер поместил информацию в стандартный поток вывода и для передачи этой информации в классе `KProcess` предусмотрен сигнал `receivedStdout`. Для обработки этого сигнала нами создан приемник `SimpleClientApp::slotDataReceived`. Учитывая, что никто не застрахован от ошибок, о возникновении которых тоже хотелось бы знать, этот же приемник используется и для обработки сигнала `KProcess::receivedStderr`, имеющего тот же набор аргументов, что и сигнал `KProcess::receivedStdout`, и отличающийся от него только тем, что он работает со стандартным потоком ошибок, а не со стандартным потоком вывода.

Приемник `SimpleClientApp::slotDataReceived` имеет достаточно простую структуру: в первом его аргументе передается указатель на объект класса

11 Зак. 113

KProcess, сигнал которого обрабатывается данным приемником, во втором аргументе передается указатель на байтовый буфер, содержащий переданную сервером информацию, а в третьем аргументе — объем переданной информации в байтах. Для вывода информации на экран используется функция QMultiLineEdit::insertLine, в качестве аргумента которой передается объект класса QString. Преобразование указателя на текстовый буфер может быть произведено неявным образом непосредственно при вызове функции insertLine, но нет никакой уверенности в том, что передаваемая строка будет завершена нулевым символом. Поэтому преобразование производится в конструкторе класса QString, а затем вызывается функция QString::truncate, устанавливающая истинный размер выводимого текста.

Для того чтобы пользователь мог убедиться в том, что работа сервера завершилась с выводом сообщения, в рассматриваемом приложении обрабатывается сигнал KProcess::processExited, посылаемый объектом класса KProcess при завершении процессом сервера своей работы. Этот сигнал обрабатывается приемником SimpleClientApp::slotProcessExited. В качестве аргумента ему передается указатель на объект класса KProcess, сигнал которого обрабатывается данным приемником. Чтобы идентифицировать закрывающийся процесс, для указанного объекта вызывается функция KProcess::getPid, возвращающая его идентификатор, который сразу же приводится к целочисленному типу. Для формирования сообщения, содержащего значение этого идентификатора, используется функция QString::sprintf. Вывод сообщения на экран производится все той же функцией QMultiLineEdit::insertLine.

Для запуска процесса на исполнение в функции SimpleClientApp::initView используется функция KProcess::start, имеющая два аргумента. В первом аргументе данной функции передается значение перечислимого типа KProcess::RunMode. Рассмотрим подробнее влияние передаваемых в этом аргументе значений на процесс активизации сервера.

- DontCare — порождаемый процесс выполняется одновременно с порождающим процессом. Порождающий процесс не получает извещения о завершении порожденного им процесса.
- NotifyOnExit— порождаемый процесс выполняется одновременно с порождающим процессом. Порожденный процесс посылает породившему его процессу сигнал processExited. Поскольку порождающий процесс должен обработать сигнал о завершении порожденного им процесса, он должен существовать на этот момент.
- Block — выполнение порождающего процесса приостанавливается до тех пор, пока порожденный им процесс не завершит свою работу. Не рекомендуется блокировать вызывающий процесс в приложениях, использующих графический интерфейс, поскольку это может привести к задержке реакции приложения на действия пользователя.

Во втором аргументе функции KProcess::start передается значение перечислимого типа KProcess::Communication, определяющего доступные способы взаимодействия порождающего и порожденного процессов. Значения этого типа представляют собой флаги и могут объединяться операцией ЛОГИЧЕСКОГО ИЛИ. Определены следующие флаги:

- NoCommunication — процессы не взаимодействуют;
- Stdin — производится только передача информации порожденному процессу;
- Stdout — производится только чтение информации из стандартного потока вывода;
- Stderr — производится только чтение информации из стандартного потока ошибок;
- AllOutput — производится только чтение информации;
- All — производится как передача, так и чтение информации из сервера;
- NoRead — запрещает считывание информации из потоков вывода. Требует одновременной установки флага Stdout или Stderr (явно или неявно).

Процесс может быть запущен только один раз. Поэтому при попытке повторного запуска уже исполняющегося процесса функция start возвратит значение false. Однако, если не проверять возвращаемое этой функцией значение, для разработчика ничего не изменится от повторного вызова данной функции: он получит исполняющийся процесс, но следует помнить о том, что, независимо от числа подобных запусков, остановить процесс он может только один раз. Для определения текущего состояния процесса используется функция KProcess::isRunning, возвращающая значение true, если указанный процесс исполняется.

В рассматриваемом приложении порожденный процесс завершается самостоятельно, как только он выполнит всю свою работу. В некоторых случаях требуется, чтобы порожденный процесс постоянно находился в режиме ожидания данных от породившего его процесса для их обработки. В этом случае порожденный процесс не может сам определить, когда ему следует завершить свою работу, и должен полагаться в этом вопросе на породивший его процесс.

Для завершения работы порожденного процесса в породившем его процессе достаточно вызвать функцию KProcess::kill. Эта функция возвращает логическое значение, указывающее на успешность доставки сигнала процессу. Если процедура завершения работы закрываемого процесса может занять продолжительное время, о завершении этой процедуры можно узнать, вызвав функцию KProcess::isRunning, которая в данном случае должна возвратить значение false.

Создание более сложного сервера

Одним из типичных примеров задач, выделяемых в отдельный процесс, является работа с удаленными ресурсами, например чтение файла по сети. Чтение и запись сетевого файла выполняются с малой скоростью и, если выполнение этой задачи не выделить в отдельный процесс, процессор большую часть времени будет простаивать или выполнять задачи конкурентов.

Поскольку библиотека KDE не делает принципиального различия между локальными и удаленными файлами, создаваемый нами сервер, текст которого можно найти на прилагаемом к книге CD, будет записывать и считывать информацию из локального файла, имя которого будет передаваться в первом аргументе его командной строки.

Чтобы создать приложение сервера:

1. Выберите команду меню **Project | New**. Появится диалоговое окно **ApplicationWizard**.
2. В окне иерархического списка раскройте каталог **KDE**, выделите строку **KDE Mini** (Приложение KDE с минимальными возможностями) и нажмите кнопку **Next**. Раскроется вторая вкладка мастера создания приложений.
3. В текстовое поле **Project name** введите имя приложения **FileServer** и нажмите кнопку **Create**.
4. По завершении процесса создания приложения нажмите кнопку **Exit**. Будет создана заготовка приложения.
5. Выберите команду меню **Project | Options** или нажмите клавишу <F7>. Появится диалоговое окно **Project Options** (Свойства проекта).
6. Раскройте вкладку **Linker Options** (Свойства компоновщика). Диалоговое окно **Project Options** примет вид, изображенный на рис. 10.2.
7. Установите флажок **kfile** и нажмите кнопку **OK**. Диалоговое окно **Project Options** закроется.
8. В окне раскрывающихся списков раскройте вкладку **Classes**, щелкните правой кнопкой мыши по имени класса `FileServer` и выберите в появившемся контекстном меню команду **Add member function** (Добавить функцию члена класса). Появится диалоговое окно **Class Properties**, открытое на вкладке **Methods** (Функции).
9. В текстовое поле **Type** введите тип возвращаемого значения `int`, в текстовое поле **Declaration** — сигнатуру функции `fileRead(char*, int)`, в текстовое поле **Documentation** — комментарий `Reads the file` (Производит чтение файла) и нажмите кнопку **Apply**. В класс будет добавлена новая функция.

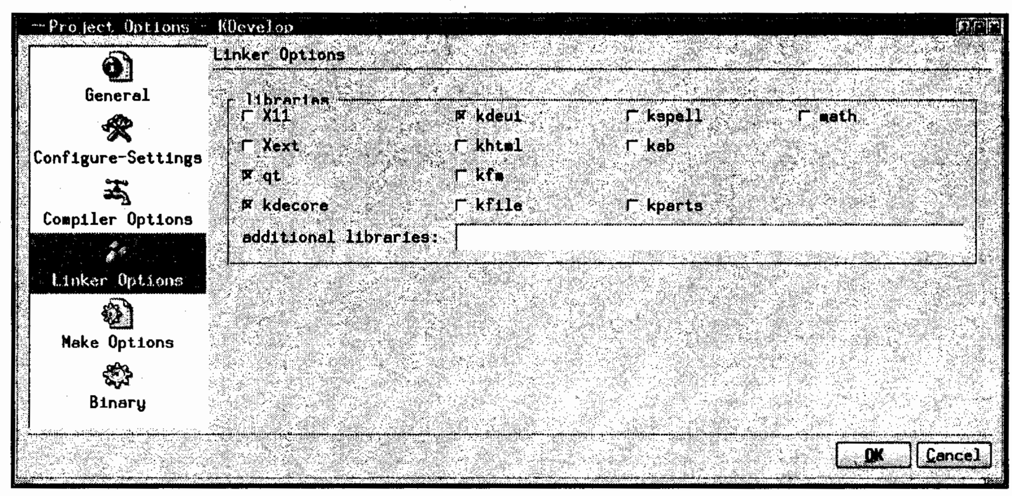


Рис. 10.2. Диалоговое окно **Project Options**

10. Повторите пункты 8 и 9 для добавления в класс функции `fileWrite(char*)`, снабдив ее комментарием **Writes the file** (Производит запись в файл).
11. В открывшемся после добавления функций окне редактирования файла fileserver.cpp измените реализацию класса `FileServer` в соответствии с текстом листинга 10.4.

Листинг 10.4. Реализация класса FileServer

```
/** Конструктор */
FileServer::FileServer(char* fileName, bool ind)
{
 isReading = ind;

 fileTemp = new QFile( fileName);

 if( ind)
  fileTemp-> open( IO_ReadOnly);
 else
  fileTemp-> open( IO_WriteOnly);
}

/** Деструктор */
FileServer::~FileServer()
{
 delete fileTemp;
}
```

```
/** Производит чтение файла */
int FileServer::fileRead( char* buf, int len)
{
 return fileTemp-> readBlock( buf, len);
}

/** Производит запись в файл */
int FileServer::fileWrite( char* buf)
{
 int res;

 if( buf[0])
  res = fileTemp-> writeBlock( buf, strlen( buf));
 else
  res = fileTemp-> writeBlock( buf, 1);

 fileTemp-> flush();

 return res;
}
```

12. В окне раскрывающихся списков раскройте вкладку **Classes**, щелкните правой кнопкой мыши по имени класса `FileServer` и выберите в появившемся контекстном меню команду **Add member variable**. Появится диалоговое окно **Class Properties**, открытое на вкладке **Attributes**.
13. В текстовое поле **Type** введите тип переменной `bool`, в текстовое поле **Name** — идентификатор переменной `isReading`, в группе **Access** установите переключатель **Private**, в текстовое поле **Documentation** введите комментарий `File operation flag` (Флаг выполняемой с файлом операции) и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.
14. Повторите пункты 12 и 13, добавив в класс `FileServer` переменную `fileTemp`, имеющую тип `QFile*`, снабдив ее комментарием `Pointer to temporary file object` (Указатель на временный файловый объект).
15. В открывшемся после добавления новых переменных окне редактирования файла fileserver.h измените заголовок класса `FileServer` в соответствии с текстом листинга 10.5.

Листинг 10.5. Заголовок класса `FileServer`

```
#include <kapp.h>
#include <qfile.h>

/** Класс FileServer представляет собой базовый класс приложения */
class FileServer
{
 public:
```

```
  /** Конструктор */
  FileServer( char*, bool);
  /** Деструктор */
  ~FileServer();
  /** Производит запись в файл */
  int fileWrite( char*);
  /** Производит чтение файла */
  int fileRead( char*, int);
 private: // Закрытые переменные
  /** Флаг выполняемой с файлом операции */
  bool isReading;
  /** Указатель на временный файловый объект */
  QFile* fileTemp;
};
```

16. Откройте окно редактирования файла main.cpp и измените функцию `main` в соответствии с текстом листинга 10.6.

Листинг 10.6. Функция `main`

```
#include <klocale.h>
#include <iostream.h>
#include <stdlib.h>

#include "fileserver.h"

int main(int argc, char *argv[])
{
 if( argc < 3)
 {
  cerr << argc <<
      i18n(" arguments is not enough. Must be 3 at least.") << endl;
  return EXIT_FAILURE;
 }

 bool isReading = !strcasecmp("read", argv[2]);
 char buf[256];

 FileServer fs( argv[1], isReading);

 if( isReading)
 {
  while( fs.fileRead( buf, sizeof(buf)) == sizeof(buf))
  {
   cout.write(buf, sizeof(buf));
  }
```

```
  cout.write(buf, sizeof(buf));
 }
 else
 {
  do
  {
   cin.read(buf, sizeof(buf));

   fs.fileWrite( buf);

  } while( buf[0]);
 }

 return EXIT_SUCCESS;
}
```

17. Выберите команду меню **Build | Make**, нажмите клавишу <F8> или кнопку **Make** в панели инструментов. Приложение сервера будет откомпилировано.
18. Переместите созданный исполняемый файл в каталог \usr\bin.

Созданный нами сервер предназначен для работы с файлами, поэтому при его создании использовались возможности класса `QFile`, содержащегося в библиотеке Qt. В создаваемых средой разработки KDevelop заготовках консольного приложения не предусмотрена поддержка этой библиотеки, а все приложения Qt имеют слишком сложную для данного приложения структуру. Поэтому в качестве основы создаваемого приложения была выбрана заготовка приложения KDE с минимальными возможностями.

Поскольку заготовка приложения уже содержала описание класса, то этот класс и был использован нами для выполнения возлагаемых на приложение задач. Для этого пришлось отказаться от его базового класса и внести изменения в его конструктор и деструктор.

Стандартные потоки ввода/вывода, используемые для обмена информацией между клиентом и сервером в операционной системе UNIX, предназначены для работы с текстовой информацией и передача по ним двоичной информации сопряжена с необходимостью решения некоторых проблем, прежде всего связанных с определением объема переданной информации. Чтобы не связываться с решением этих вопросов, создаваемый нами сервер будет работать исключительно с текстовой информацией.

При создании класса `FileServer` нами использовались те же способы обращения к файлу, что и в стандартных классах документов, рассмотренных в *главе 7*. Однако, чтобы не производить лишних преобразований типов, данное приложение ориентировано исключительно на работу с локальными файлами и в нем не используются объекты класса `KURL` и связанных с ним классов.

Создаваемый нами класс предназначен как для чтения, так и для записи информации в файл, однако, поскольку одновременное чтение и запись информации в файл могут привести к порче его содержимого, объект класса может осуществлять или только чтение, или только запись информации, в зависимости от значения второго аргумента командной строки приложения. В первом аргументе командной строки, как уже говорилось ранее, передается имя файла, в котором следует сохранить или из которого требуется извлечь информацию.

Поскольку в первом элементе массива аргументов командной строки приложения передается путь к его исполняемому файлу, этот массив для данного приложения должен содержать не менее трех элементов. Размерность массива аргументов командной строки передается в первом аргументе функции `main`, поэтому первый же оператор данной функции проверяет значение этого аргумента и посылает сообщение об ошибке, если размерность массива недостаточна.

После этого в функции `main` на основании значения второго аргумента командной строки формируется флаг выполняемого действия. Поскольку все аргументы командной строки сами являются строками, то для определения текущего состояния флага используется функция `strcasecmp`, производящая сравнение двух строк без учета регистра символов и возвращающая нулевое значение в случае их равенства.

Для работы с файлом создается объект класса `FileServer`, в котором, в зависимости от состояния флага выполняемого действия, производится чтение информации из файла или запись информации в него. Реализация конструктора данного класса достаточно проста. После сохранения флага выполняемой операции в закрытой переменной класса в его конструкторе создается динамический объект класса `QFile` для работы с файлом и, в зависимости от производимой с ним операции, открывается для чтения или для записи.

Для чтения информации из файла используется функция `FileServer::fileRead`, осуществляющая блочное чтение из файла. Эта функция вызывается в цикле в функции `main` до тех пор, пока она в состоянии полностью заполнить буфер информацией. Если буфер заполнен не полностью, это означает, что достигнут конец файла и вся информация из него считана. По мере поступления информации она передается в стандартный поток вывода вызовом функции `ostream::write`. В отличие от перегруженной операции << (`ostream::operator<<`) этого класса данная функция позволяет указать объем передаваемой информации, что позволяет сохранить целостность строк, поскольку исключает анализ передаваемой информации.

Учитывая, что сервер предназначен для работы с текстовой информацией, размер посылаемого буфера в нем остается неизменным, поскольку для оп-

ределения границ полезной информации здесь используются специальные символы.

Для непосредственного чтения информации из файла используется функция `QFile::readBlock`, возвращающая число считанных ею байт информации, которое является также возвращаемым значением функции `FileServer::fileRead`. В первом аргументе функции `readBlock` передается указатель на буфер, в который следует считать информацию, а во втором — размер этого буфера.

Для записи информации в файл используется функция `FileServer::fileWrite`. Она также вызывается в цикле, но условие завершения работы этого цикла определяется не возвращаемым значением этой функции, а полученной из стандартного потока ввода информацией. Для завершения операции записи информации в файл в этот поток следует поместить пустую строку, т. е. первый элемент в текстовом буфере должен иметь нулевое значение. Для извлечения информации из стандартного потока ввода используется функция `istream::read`, которая так же, как и функция `ostream::write`, позволяет указывать объем передаваемой информации. Это обстоятельство очень важно, поскольку перегруженная операция >> (`istream::operator>>`) не делает различий между символами пробела и перевода строки, что очень нежелательно при работе с текстами.

Непосредственная запись информации в файл осуществляется функцией `QFile::writeBlock`, в первом аргументе которой передается указатель на блок записываемой информации, а во втором — ее объем. Для определения объема записываемой информации используется функция `strlen`, возвращающая размер строки без завершающего ее нулевого символа. Поскольку перед закрытием файла этот символ следует записать, то запись пустой строки производится специальным образом. При закрытии файла содержимое его промежуточного буфера будет автоматически сохранено на диске, однако, для большей надежности, после вызова функции `writeBlock` сразу же вызывается функция `QFile::flush`, принудительно записывающая содержимое этого буфера на диск.

При уничтожении объекта класса `FileServer` в нем необходимо уничтожить динамический объект класса `QFile`, что и производится в деструкторе класса `FileServer`.

Создание клиента

Теперь, после создания сервера, можно приступить к созданию использующего его клиента. Текст этого приложения можно найти на прилагаемом к книге CD.

Чтобы создать приложение клиента для созданного нами сервера:

1. Выберите команду меню **Project | New**. Появится диалоговое окно **ApplicationWizard**.

2. В окне иерархического списка оставьте выделенной строку **KDE Normal** и нажмите кнопку **Next**. Раскроется вторая вкладка мастера создания приложений.
3. В текстовое поле **Project name** введите имя приложения **FileClient** и нажмите кнопку **Create**.
4. По завершении процесса создания приложения нажмите кнопку **Exit**. Будет создана заготовка однооконного приложения KDE.
5. В окне иерархических списков раскройте вкладку **Classes**, щелкните правой кнопкой мыши по имени класса `FileClientApp` и выберите в появившемся контекстном меню команду **Add slot**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Slots**.
6. В текстовое поле **Declaration** введите имя приемника `slotDataReceived(KProcess*, char*, int)`, в текстовое поле **Documentation** — комментарий `Receives data from process` (Получает данные из процесса) и нажмите кнопку **Apply**. В класс будет добавлен новый приемник.
7. Повторите пункты 5 и 6 для добавления в класс `FileClientApp` приемника `slotLineSent(KProcess*)`, снабдив его комментарием `Sends an another line` (Посылает следующую строку).
8. Повторите пункты 5 и 6 для добавления в класс `FileClientApp` приемника `slotErrorReceived(KProcess*, char*, int)`, снабдив его комментарием `Receives error information` (Получает информацию об ошибке).
9. Повторите пункты 5 и 6 для добавления в класс `FileClientApp` приемника `slotProcessExited(KProcess*)`, снабдив его комментарием `Receives process terminate signal` (Получает сигнал о завершении процесса).
10. В открывшемся после добавления приемников файле fileclient.cpp измените эти приемники в соответствии с текстом листинга 10.7.

Листинг 10.7. Приемники класса `FileClientApp`

```
/** Получает данные из процесса */
void FileClientApp::slotDataReceived(KProcess* prc, char* buf, int len)
{
 QString sz(buf);

 sz.truncate( len);

 view-> insertLine( sz);
}

/** Посылает следующую строку */
void FileClientApp::slotLineSent(KProcess* prc)
```

```
{
 if( count < view-> numLines())
  prc-> writeStdin( view-> textLine(count) + "\n", 256);
 else
  if( count == view-> numLines())
   prc-> writeStdin( "", 256);

 count++;
}

/** Получает информацию об ошибке */
void FileClientApp::slotErrorReceived(KProcess* prc, char* buf, int len)
{
 QString sz(buf);

 sz.truncate( len);

 err += sz;
}

/** Получает сигнал о завершении процесса */
void FileClientApp::slotProcessExited(KProcess* prc)
{
 QString sz;

 if( !err.isEmpty())
  KMessageBox::error( this, err);

 err = "";

 sz.sprintf("Process %d exited!", (int)prc-> getPid());

 KMessageBox::information( this, sz);
}
```

11. В конструкторе класса `FileClientApp` удалите строку

```
fileSaveAs->setEnabled(false);
```

12. Измените функцию `FileClientApp::initView` в соответствии с текстом листинга 10.8.

Листинг 10.8. Функция initView

```
void FileClientApp::initView()
{
 ////////////////////////////////////////////////////////////////////
 // Создайте здесь главное окно приложения, которое будет управляться
 // объектом класса KTMainWindow, и свяжите это окно с объектом класса
 // документа для вывода его содержимого на экран.
```

```
  view = new FileClientView(this);
  doc->addView(view);
  setCentralWidget(view);
  setCaption(doc->URL().fileName(),false);

  connect(&proc, SIGNAL(processExited(KProcess *)),
              this, SLOT(slotProcessExited(KProcess *)));

  connect(&proc, SIGNAL(wroteStdin(KProcess *)),
                this, SLOT(slotLineSent(KProcess *)));

  connect(&proc, SIGNAL(receivedStdout(KProcess *, char *, int)),
          this, SLOT(slotDataReceived(KProcess *, char *, int)));

  connect(&proc, SIGNAL(receivedStderr(KProcess *, char *, int)),
          this, SLOT(slotErrorReceived(KProcess *, char *, int)));
}
```

13. Измените приемники `FileClientApp::slotFileOpen` и `FileClientApp::slotFileSaveAs` в соответствии с текстом листинга 10.9.

Листинг 10.9. Приемники slotFileOpen и slotFileSaveAs

```
/** Обрабатывает команду открытия файла */
void FileClientApp::slotFileOpen()
{
 slotStatusMsg(i18n("Opening file..."));

 if(!doc->saveModified())
 {
   // here saving wasn't successful

 }
 else
 {
  KURL url=KFileDialog::getOpenURL(QString::null,
    i18n("* |All files"), this, i18n("Open File..."));

   if(!url.isEmpty() && !proc.isRunning())
   {
    proc.clearArguments();

    proc << "fileserver" << url.path() << "read";

    proc.start(KProcess::NotifyOnExit, KProcess::All);

    setCaption(url.fileName(), false);
   }
 }
```

```
 slotStatusMsg(i18n("Ready."));
}

/** Обрабатывает команду сохранения файла под другим именем */
void FileClientApp::slotFileSaveAs()
{
 slotStatusMsg(i18n("Saving file with a new filename..."));

 KURL url=KFileDialog::getSaveURL(QDir::currentDirPath(),
    i18n("* |All files"), this, i18n("Save as..."));

 if(!url.isEmpty() && !proc.isRunning())
 {
  proc.clearArguments();

  proc << "fileserver" << url.path() << "write";

  count = 1;

  proc.start(KProcess::NotifyOnExit, KProcess::All);
  proc.writeStdin( view-> textLine(0) + "\n", 256);

  setCaption(url.fileName(), false);
 }

 slotStatusMsg(i18n("Ready."));
}
```

14. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `FileClientApp` и выберите в появившемся контекстном меню команду **Add member variable**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes**.
15. В текстовое поле **Type** введите тип переменной `KProcess`, в текстовое поле **Name** — идентификатор переменной `proc`, в группе **Access** установите переключатель **Private**, в текстовое поле **Documentation** введите комментарий `Process object` (Объект класса процесса) и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.
16. Повторите пункты 14 и 15 для добавления в класс `FileClientApp` переменной `count`, имеющей тип `int`, снабдив ее комментарием `Lines counter` (Счетчик строк).
17. Повторите пункты 14 и 15 для добавления в класс `FileClientApp` переменной `err`, имеющей тип `QString`, снабдив ее комментарием `Error message` (Сообщение об ошибке).
18. В начале открывшегося после добавления переменных окна редактирования файла simpleclient.h после строки `#include <kurl.h>` поместите строку

```
#include <kprocess.h>
```

19. Во вкладке **Classes** окна иерархических списков щелкните левой кнопкой мыши на имени класса FileClientView. Откроется окно редактирования файла fileclientview.h.
20. В начале этого файла замените строку #include <qwidget.h> строкой

```
#include <keditcl.h>
```

21. В заголовке класса SimpleClientView замените строку class SimpleClientView : public QWidget строкой

```
class SimpleClientView : public Kedit
```

22. Щелкните правой кнопкой мыши в окне редактирования файла simpleclientview.h и выберите в появившемся контекстном меню команду **Switch Header/Source**. Откроется окно редактирования файла fileclientview.cpp.
23. Измените конструктор класса FileClientView в соответствии с текстом листинга 10.10.

Листинг 10.10. Конструктор класса FileClientView

```
FileClientView::FileClientView(QWidget *parent, const char *name)
  : KEdit(parent, name)
{
 setBackgroundMode(PaletteBase);
}
```

24. Выберите команду меню **Build | Execute**, нажмите клавишу <F9> или кнопку **Run** в панели инструментов. Появится окно приложения, изображенное на рис. 10.3.

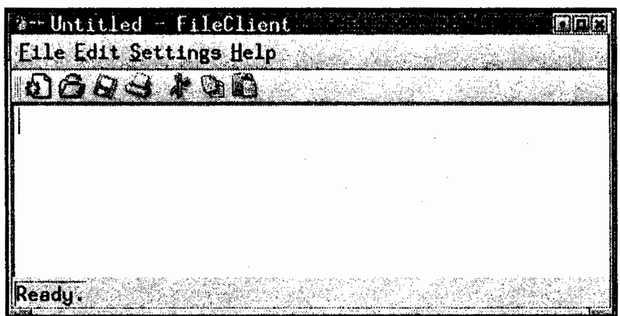


Рис. 10.3. Окно приложения сервера

25. Введите в окно какой-нибудь текст и выберите команду меню **File | Save As** (Файл | Сохранить как). Появится диалоговое окно **Save as**.

26. В текстовое поле **Location** введите путь к файлу, в котором будет сохранен текст, и нажмите кнопку **Save**. Появится окно сообщения о завершении процесса, изображенное на рис. 10.4.

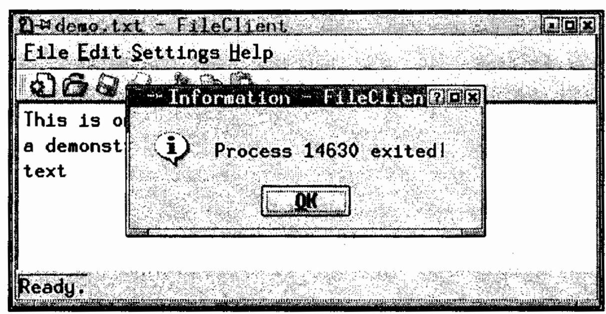


Рис. 10.4. Окно сообщения

27. Выберите команду меню **File | Open** (Файл | Открыть), нажмите комбинацию клавиш <Ctrl>+<O> или кнопку **Open** в панели инструментов. Появится диалоговое окно **Open File**.
28. В окне списка выделите имя файла, в котором был сохранен текст, и нажмите кнопку **OK**. Сохраненный текст появится на экране, как это показано на рис. 10.5.
29. Закройте приложение.

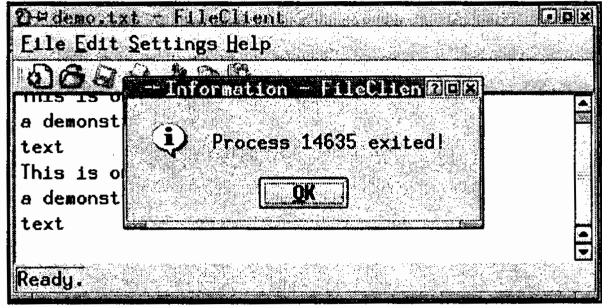


Рис. 10.5. Вывод сохраненного текста

Приложение клиента создавалось исключительно для демонстрации принципов работы с сервером, поэтому в нем были реализованы только необходимые для этого функции. Поскольку исследуемый сервер работает с текстовой информацией, созданное нами приложение клиента по методике, описанной в *главе 8*, было превращено в простейший текстовый редактор. Так как нас не интересовали функциональные возможности этого редактора, в пользовательский интерфейс приложения не было внесено никаких

изменений за исключением того, что в нем была сделана доступной использовавшаяся в приложении команда меню **File | Save As**.

Связывание сигналов объекта класса KProcess с соответствующими приемниками класса приложения, как и в рассмотренном ранее приемнике простейшего сервера, было произведено в функции FileClientApp::initView, однако инициализация этого объекта производилась в использующих его приемниках команд меню. Поскольку данное приложение не только считывает посылаемую сервером информацию, но и само посылает ему данные, в функцию initView добавлен вызов еще одной функции QObject::connect. Данная функция связывает сигнал KProcess::wroteStdin, посылаемый после завершения процедуры записи очередной порции информации в стандартный поток ввода, с обрабатывающим его приемником. Кроме того, в данном случае сигналы KProcess::receivedStdout и KProcess::receivedStderr обрабатываются разными приемниками, чтобы не смешивать полезную информацию и информацию об ошибках.

Процесс чтения информации из файла инициируется в приемнике FileClientApp::slotFileOpen, обрабатывающем команду меню **File | Open**. Поскольку объект класса KEdit является самодостаточным и не нуждается в поддержке класса документа, из данного приемника исключено большинство операций, вносящих изменения в документ. Для сохранения структуры функции в ней оставлен вызов функции FileClientDoc::saveModified, однако, поскольку мы нигде не устанавливали флага наличия в документе изменений, этот вызов является пустым.

Для получения объекта класса KURL, хранящего информацию о выбранном пользователем файле, используется статическая функция KFileDialog::getOpenURL, выводящая на экран диалоговое окно **Open File** и возвращающая информацию о выборе пользователя. Если пользователь сделал свой выбор и запущенный ранее процесс взаимодействия с файлом завершен, вызывается функция KProcess::clearArguments, очищающая массив аргументов запускаемого процесса. Если бы эта функция не вызывалась, следующий оператор просто добавил бы дополнительные аргументы в конец этого массива и при вызове функции KProcess::start процесс был бы запущен с параметрами, установленными при первом его вызове после запуска приложения или после последнего вызова функции KProcess::clearArguments.

Очищенный массив аргументов запускаемого процесса заполняется с использованием перегруженной операции << (KProcess::operator<<). Заполнение массива может производиться в одном операторе, как это сделано в рассматриваемом примере, или может осуществляться в нескольких операторах, которые не обязательно должны вызываться друг за другом. Главное, чтобы для добавления каждого элемента массива аргументов использовалась своя операция <<. В классе KProcess отсутствуют функции для замены в запус-

каемом процессе только аргументов командной строки, зато имеется функция KProcess::setExecutable, позволяющая заменить имя исполняемого файла процесса без внесения изменений в аргументы его командной строки.

Для запуска процесса на исполнение вызывается функция KProcess::start, первый аргумент которой указывает на то, что запускаемый процесс должен послать сигнал после завершения своей работы, а второй ее аргумент указывает на то, что для обмена информацией между порождающим и порожденным процессами могут использоваться все три стандартных потока ввода/вывода.

После инициализации процесса чтения информации из файла его имя вызовом функции QWidget::setCaption выводится в заголовок окна приложения, а в строку состояния вызовом приемника FileClientApp::slotStatusMsg выводится стандартное сообщение о завершении обработки команды.

После своего запуска в приемнике FileClientApp::slotFileOpen сервер считывает информацию из файла и помещает ее в стандартный поток вывода. Получив сведения о наличии в этом потоке предназначенной ему информации, объект класса KProcess извлекает ее оттуда, помещает в буфер и посылает сигнал KProcess::receivedStdout. Этот сигнал обрабатывается приемником FileClientApp::slotDataReceived. Структура этого приемника во многом напоминает структуру одноименного приемника рассмотренного ранее простейшего сервера. В этом приемнике содержимое текстового буфера копируется в объект класса QString, в указанном объекте устанавливается истинная длина полученной строки, и эта строка выводится на экран вызовом функции QMultiLineEdit::insertLine.

Процесс записи информации в файл инициируется в приемнике FileClientApp::slotFileSaveAs, обрабатывающем команду меню **File | Save As**. Для получения объекта класса KURL, хранящего информацию о выбранном пользователем файле, используется статическая функция KFileDialog::getSaveURL, выводящая на экран диалоговое окно **Save as** и возвращающая информацию о выборе пользователя. Если пользователь сделал свой выбор и запущенный ранее процесс взаимодействия с файлом завершен, вызывается функция KProcess::clearArguments, очищающая массив аргументов запускаемого процесса. Причины вызова этой функции такие же, как и при ее вызове в приемнике slotFileOpen.

Очищенный массив аргументов запускаемого процесса заполняется с использованием перегруженной операции << (KProcess::operator<<). После этого счетчику посланных строк присваивается единичное значение. Хотя процессы Linux выполняются в одном потоке и вызов одного приемника до завершения работы другого должен быть исключен, тем не менее для надежности подготовка к вызову приемника FileClientApp::slotLineSent производится до запуска обращающегося к ней процесса.

Для запуска процесса на исполнение вызывается функция KProcess::start с теми же значениями аргументов, что и в приемнике slotFileOpen, и вызывается функция KProcess::writeStdin, помещающая информацию в стандартный поток ввода. Первый аргумент данной функции содержит указатель на текстовый буфер, содержащий посылаемую информацию, а второй — объем посылаемой информации в байтах. Поскольку в сервере эта информация будет извлекаться вызовом функции istream::read, объем посылаемой клиентом информации, указываемый во втором аргументе функции writeStdin, должен быть равен объему принимаемой сервером информации, указываемому во втором аргументе функции istream::read.

Текст посылается по строкам. Для получения строки с указанным номером используется функция QMultiLineEdit::textLine. Поскольку данная функция не включает в возвращаемый ею объект класса QString символ перевода строки, этот символ приходится вручную добавлять при ее пересылке серверу.

После инициализации процесса записи информации в файл его новое имя вызовом функции QWidget::setCaption выводится в заголовок окна приложения, а в строку состояния вызовом приемника FileClientApp::slotStatusMsg выводится стандартное сообщение о завершении обработки команды.

Поскольку новую информацию в стандартный поток ввода нельзя помещать до того, как из него будет извлечена старая информация, эта процедура производится в приемнике FileClientApp::slotLineSent, обрабатывающем сигнал о том, что считывание старой информации завершено. Если значение счетчика строк на момент вызова данного приемника не превышает числа строк сохраняемого текста, возвращаемого функцией QMultiLineEdit::numLines, в нем вызывается функция KProcess::writeStdin, помещающая в поток новую строку, и значение счетчика увеличивается на единицу. Если значение счетчика строк равно числу строк хранимого текста, то серверу посылается пустая строка, побуждающая его закрыть файл, и снова значение счетчика строк увеличивается на единицу. Таким образом, при следующем вызове данного приемника он не пошлет серверу никакой информации, и это будет его последний вызов.

Для того чтобы проверить работу приемника FileClientApp::slotErrorReceived, достаточно при инициализации процесса не передать ему какой-либо один или оба аргумента командной строки. Этот приемник имеет, на первый взгляд, довольно странную структуру, поскольку вместо того, чтобы вывести полученное сообщение об ошибке на экран, он накапливает его в объекте класса QString. Это связано с тем, что стандартный поток вывода ошибок разбивает помещенный в него текст на произвольные фрагменты, поэтому при непосредственном выводе полученной информации в окно сообщения мы получим множество окон сообщения, содержащих фрагменты текста. Кроме того, вывод всех сообщений в одной функции позволяет избежать наложения окон сообщений друг на друга.

Вывод информации о завершении работы порожденного процесса производится в приемнике `FileClientApp::slotProcessExited`. Помимо операторов, содержавшихся в одноименном приемнике клиента простейшего сервера и выводящих идентификатор завершенного процесса, в этот приемник включены операторы для вывода окна сообщения о возникших ошибках. Для определения того, следует ли выводить это окно, используется функция `QString::isEmpty`, возвращающая значение `true`, если данная строка пуста. Поскольку информация об ошибке после своего вывода утратила актуальность, содержащая ее строка очищается.

Некоторые замечания

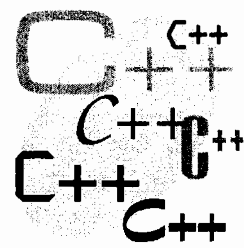
Как уже говорилось выше, использование процессов позволяет обеспечить высокую надежность операционной системы. Однако не следует забывать о том, что за 30 лет существования операционной системы UNIX и за 20 лет ее активного использования надежность вычислительной техники существенно повысилась, и такие меры обеспечения безопасности во многих случаях являются излишними. Прежде чем восторгаться надежностью операционной системы, позволяющей разработчикам не слишком заботиться о качестве создаваемых ими приложений, следует помнить, за счет чего она достигается.

Для своего исполнения каждый процесс запрашивает у компьютера ресурсы, которые он освобождает после того, как надобность в них отпадает. Только после этого ресурс может быть выделен другому процессу. Ресурсы не могут разделяться процессами, что, во-первых, приводит к необходимости использовать для передачи объемных ресурсов от одного процесса другому узких каналов стандартных потоков ввода/вывода, и, во-вторых, приводит к необходимости создания копий ресурсов в каждом из использующих их процессов.

Один и тот же процесс не может разделяться между несколькими процессами для выполнения сходных задач. Это следует, хотя бы, из синтаксиса запуска процесса, определяемого исключительно именем своего исполняемого файла. Идентификатор процесса получается исключительно для справки и не используется при обращении к его функциям. Поэтому, если приложению потребуется выполнить несколько одинаковых обработок различных объектов, ему придется запустить для каждой из этих обработок отдельный процесс. Конечно, в настоящее время оперативная память стремительно дешевеет, однако и возлагаемые на компьютер задачи становятся все более сложными, поэтому не стоит ей разбрасываться.

Операционная система Windows, конечно, не является истинно многозадачной, поскольку допускает распараллеливание операций в пределах одного процесса. Однако еще остается вопросом, является ли это ее недостатком.

По мере своего развития Windows получила достаточно мощные средства для создания процессов и организации их взаимодействия, а возможность разделения ресурсов между потоками и использование всеми объектами данного потока одного и того же программного кода существенно снижает требования приложений Windows к оперативной памяти. В последнее время разработчики библиотеки Qt предпринимают попытки использования потоков и в приложениях Linux, однако их успех существенно затруднен тем обстоятельством, что основные классы этой библиотеки не обеспечивают безопасной работы приложений в многопотоковой среде.



ГЛАВА 11

Справка в приложении

Одной из основных отличительных особенностей профессионального приложения является наличие в нем хорошо организованной справочной системы. Отсутствие справочной системы допустимо только в приложениях, создаваемых разработчиком для решения своих внутренних задач, и то только в том случае, если это приложение предельно просто или функционально закончено и его последующая доработка не предполагается. В противном случае разработчику придется через некоторое время вспоминать, какие функции были заложены в данное приложение и как ими воспользоваться.

Особой формой справочной системы являются комментарии в тексте программы. Эта справочная информация предназначена для разработчика программы, память которого имеет ограниченную емкость, и для других разработчиков, которые захотят внести изменения в программу. Этой формой справки не стоит пренебрегать даже в том случае, если программу не предполагается никому передавать, иначе потом придется судорожно вспоминать, какая идея была заложена в данную функцию, а также почему все действия в ней производятся "через левое ухо".

Несмотря на особую роль справочной системы в оценке приложения потребителем, многие программисты откладывают ее создание до полной отладки проекта. Однако многочисленные исследования показали, что создание справочной системы до начала этапа программирования позволяет существенно повысить качество создаваемого проекта и уменьшить время, затрачиваемое на его разработку, за счет более тщательной проработки пользовательского интерфейса и взаимодействия составных частей приложения.

Как ни странно, разработчики среды KDevelop придерживаются других взглядов. Только этим или боязнью создать себе конкурентов может быть объяснено их пренебрежение справочной системой создаваемых в этой среде приложений. И если в приложениях Qt речь идет об отсутствии в меню

Help каких-либо команд, за исключением команды вывода никому не нужного диалогового окна **About**, то в приложениях KDE речь идет о большем.

Формы представления справочной информации

Различные виды справочной информации предполагают разные способы доступа к ней. Например, для получения информации о кнопке панели инструментов достаточно краткой контекстной справки, появляющейся при помещении на нее указателя мыши, а для получения развернутой информации по какому-либо вопросу необходимо вывести специальное окно с гипертекстовыми ссылками. Поэтому, прежде чем приступить к созданию справочной системы, следует для каждого ее элемента определить:

- Каким образом он должен вызываться?
- В какое окно будет выводиться справочная информация?
- В какой форме она будет представляться?
- Каким образом справочная система будет включена в текст программы?

Ниже будут даны ответы на каждый из поставленных вопросов.

Способы доступа к справочной системе

В Linux так же, как и в Windows, используются различные способы доступа к справочной информации. Ниже перечислены основные из них.

- *Вызов команды меню.* Многие приложения позволяют пользователю применять для доступа к справочной системе приложения специальные команды меню, расположенного в панели меню главного окна приложения. Как правило, это меню создается автоматически мастером создания приложения, и в него помещаются соответствующие команды.
- *Вызов с клавиатуры.* При нажатии пользователем клавиши <F1> окну, имеющему в настоящее время фокус ввода, посылается сигнал. Если это окно является дочерним (например, элемент управления диалогового окна), оно пересылает этот сигнал своему родительскому окну. Если в момент нажатия клавиши <F1> фокус ввода принадлежал команде меню, то сигнал передается окну, в котором расположено данное меню.
- *Вызов с помощью мыши.* Доступ к справочной системе приложения, осуществляемый с помощью мыши, отличается большим разнообразием. Как уже говорилось, при помещении курсора на некоторые элементы управления рядом с ними появляется маленькое окно с контекстной справкой. Для получения более подробной информации по некоторым элементам управления может быть использована кнопка (**What's**

this?), превращающая указатель мыши в наклонную стрелку со знаком вопроса. При помещении такого указателя на элемент управления и щелчке левой кнопкой мыши на месте щелчка появляется окно с более подробной справкой.

Описанные выше способы доступа к справочной системе приложения не являются взаимоисключающими, а дополняют и дублируют друг друга. Так например, клавиша <F1> и кнопка **What's this?** (Что это?) в панели инструментов приложения часто дублируются в его меню.

Способы представления справочной информации

После того как приложение получило запрос на вывод справочной информации, оно должно определить форму вывода ее на экран. Linux не имеет такой строгой системы стандартных элементов пользовательского интерфейса, как Windows, но, тем не менее, от разработчиков все-таки требуется придерживаться определенных правил при выборе дизайна окон, в которые будет выводиться справочная информация. К счастью, об этом уже позаботились разработчики библиотеки Qt и среды разработки KDevelop.

Справочная система может выводиться на экран одним из перечисленных ниже способов.

- *В отдельном окне.* В отличие от Windows, где формат окна справочной системы един для всех приложений и зависит только от формы представления информации (обычная справка или справка в формате HTML), в приложениях Linux окно справочной системы имеет произвольный формат. Это окно может быть использовано для поиска информации по ключевому слову, просмотра оглавлений, переходов по гипертекстовым ссылкам и других операций, однако минимальный набор функциональных возможностей этих окон, в отличие от окон справочной системы Windows, не определен. Наиболее полно предъявляемым к этому типу окон требованиям отвечает окно справочной системы KDE, изображенное на рис. 11.1.
- *Во всплывающем окне.* Это окно, изображенное на рис. 11.2, используется для вывода справочной информации небольшого объема или кратких пояснений. Оно не имеет элементов управления и закрывается при щелчке кнопкой мыши за его пределами. Это окно нельзя минимизировать или переместить.
- *В строке состояния.* В строке состояния, как правило, выводится информация по выделенным в настоящее время командам меню и другая вспомогательная информация, как это показано на рис. 11.3.

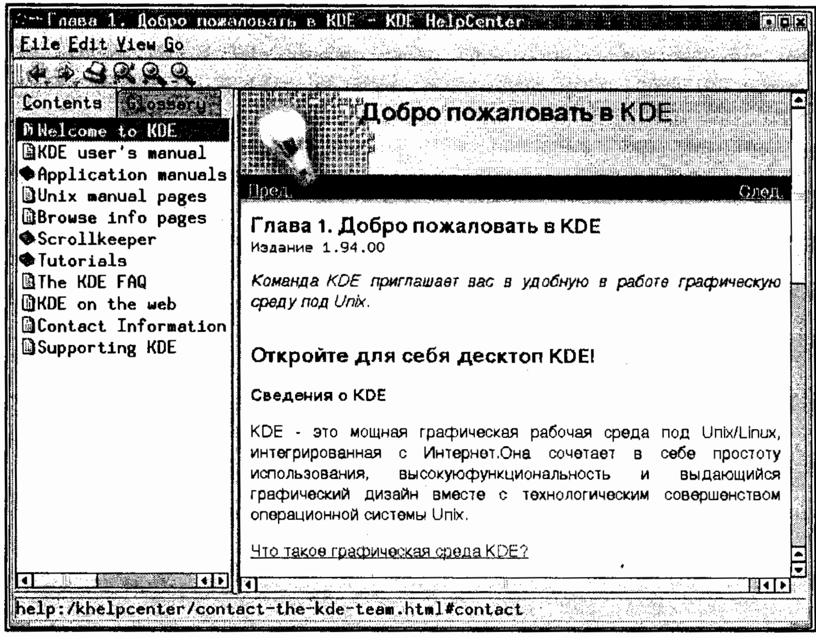


Рис. 11.1. Окно справочной системы KDE

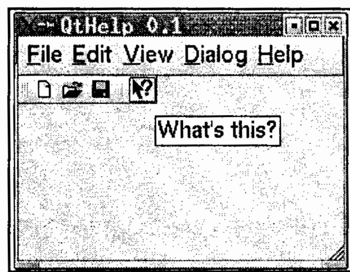


Рис. 11.2. Всплывающее окно справки

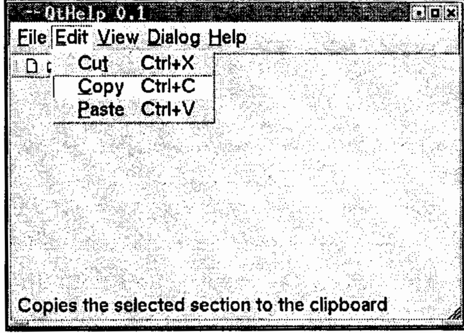


Рис. 11.3. Справочная информация в строке состояния

Формы представления информации

Форма представления справочной информации в основном определяется причинами, по которым она вызывается. Справочная информация может быть представлена в одной из перечисленных ниже форм.

- *Контекстная справка.* Данная форма служит для ответа на вопросы типа "Для чего предназначена данная кнопка?" или "Что нужно ввести в это текстовое поле?".
- *Справка по методике решения задачи.* Эта форма служит для получения описания алгоритма действий при решении конкретной задачи данным приложением. Например, о том, как распечатать документ.
- *Справка по ключевому слову.* Данная форма служит для предоставления развернутой информации по конкретному вопросу.

В отличие от приложений Windows, приложения Linux не имеют единой стратегии доступа к контекстной справке, поскольку этот доступ реализуется разработчиком приложения на основании своих личных предпочтений. Однако было бы желательно, чтобы в приложениях Linux повторялась привычная многим схема вызова контекстной справки, отработанная в приложениях Windows, когда при нажатии клавиши <F1> вызывается контекстная справка по активному окну (элемент управления так же является окном), а при нажатии комбинации клавиш <Ctrl>+<F1> активизировалась кнопка **What's this?**

Справка по методике решения задачи и справка по ключевому слову иногда объединяются под общим наименованием *командной справки*, поскольку в большинстве случаев для их вызова используются команды меню.

Программирование контекстной справки

Вопрос о программировании справочной системы приложения является наиболее важным для разработчиков, поскольку эта задача полностью ложится на их плечи. Некоторые аспекты программирования контекстной справки уже освещались нами, однако они оказались разбросанными по разным главам, где им не было уделено должного внимания. Поэтому здесь они будут объединены воедино и подробно рассмотрены.

Как уже говорилось выше, контекстная справка, предназначенная для оперативного информирования пользователя о выделенных им элементах пользовательского интерфейса, может выступать в различных формах, и именно этот тип справки широко использовался нами в создаваемых приложениях.

Программирование контекстной справки в приложениях Qt и в приложениях KDE несколько различается, но эти отличия не являются настолько существенными, чтобы выделять рассмотрение каждого из типов приложений

в отдельный раздел. Поэтому описание принципов программирования контекстной справки будет разбито по типам предоставляемой справочной информации.

Вывод подсказок

Простейшим видом контекстной справки является подсказка, появляющаяся в маленьком окне при помещении пользователем указателя мыши на тот или иной элемент пользовательского интерфейса и исчезающая после перемещения указателя мыши за пределы этого элемента или по истечении определенного времени. Как правило, подсказки выводятся для кнопок панели инструментов.

При создании приложений Qt для команд меню и связанных с ними кнопок панели инструментов подсказки могут задаваться в первом аргументе конструктора класса `QAction`. Например, оператор

```
fileNew = new QAction(tr("New File"), newIcon, tr("&New"),
                   QAccel::stringToKey(tr("Ctrl+N")), this);
```

создает объект класса `QAction` с подсказкой **New File** (Создание файла). Текст подсказки может быть в любое время изменен вызовом виртуальной функции `QAction::setToolTip`.

Подсказка кнопки панели инструментов формируется из указанного в конструкторе класса `QAction` текста подсказки и помещенной в круглые скобки комбинации клавиш, используемой для вызова соответствующей команды, как это показано на рис. 11.4.

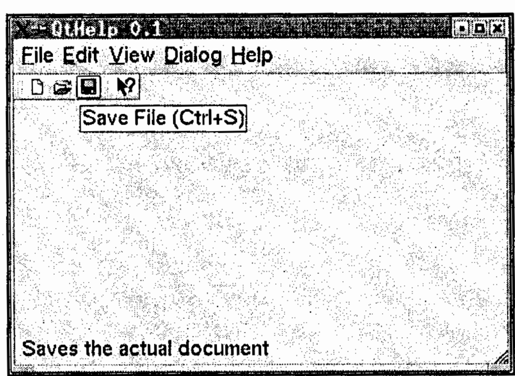


Рис. 11.4. Подсказка к кнопке панели инструментов в приложении Qt

Если возникает необходимость выводить подсказку для какого-либо иного элемента пользовательского интерфейса, то разработчику следует воспользоваться статическими методами класса `QToolTip`. Метод `QToolTip::add` вклю-

чает подсказку в объект класса `QWidget`, а метод `QToolTip::remove` удаляет эту подсказку. Подсказка может быть установлена для всего окна или для любой его статической или динамической прямоугольной области.

В том случае, когда подсказка должна сопровождаться расширенным пояснением, выводимым, как правило, в строку состояния приложения, разработчик должен использовать объект класса `QToolTipGroup`.

В приложениях KDE, использующих библиотеку данной интегрированной среды, текст подсказки для кнопки панели инструментов задается, как это ни странно, вызовом функции `KAction::setStatusText` или `KToggleAction::setStatusText`, в зависимости от связанной с этой кнопкой команды. Хотя, судя по своему названию и практике использования одноименной функции класса `QAction`, которая будет описана ниже, эта функция должна выводить сообщение в строку состояния. Пример вывода подсказки к кнопке панели инструментов приложения KDE приведен на рис. 11.5.

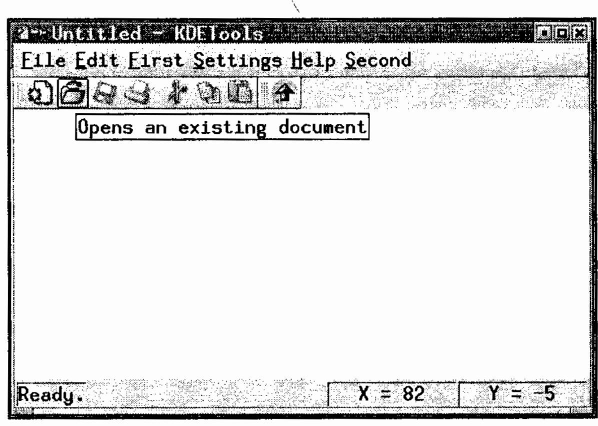


Рис. 11.5. Подсказка к кнопке панели инструментов в приложении KDE

Вывод справочной информации в строку состояния

Другой разновидностью контекстной справки являются сообщения, выводимые в строку состояния. Эти сообщения могут появляться при выделении пользователем команд меню или при выполнении им каких-либо действий. Сообщения, выводимые при выделении команд меню, связываются с командой меню вызовом виртуальной функции `QAction::setStatusTip`. Вызов этой функции является непременным атрибутом инициализации объекта класса `QAction` в функции `initActions` объекта приложения. Поскольку справочная информация в строку состояния выводится одновременно с выделением кнопки панели инструментов, то пример вывода этой информации можно найти на рис. 11.4.

В рассматриваемых ранее приложениях мы часто использовали вывод информации о выполняемой операции в строку состояния, однако эти операции протекали так быстро, что мы не успевали прочитать эту информацию. Поэтому вывод этой информации обоснован только в том случае, если выполняемая операция занимает продолжительное время и пользователю нужно сообщить, что она успешно началась и пока что продолжается без сбоев. Строка состояния может быть также использована и для вывода в нее сообщений об ошибках. Все эти операции реализуются вызовом приемника `QStatusBar::message`, в качестве аргумента которому передается выводимое сообщение. Если строка состояния уже содержала до этого какое-либо сообщение, то оно удаляется из нее.

Как уже говорилось выше, в приложениях KDE функции `KAction::setStatusText` используются с целью вывода подсказок для кнопок панели инструментов. Поэтому для вывода информации в строку состояния разработчику приходится самому вызывать соответствующие функции классов `KStatusBar` и его базового класса `QStatusBar`.

Получение информации по конкретному элементу пользовательского интерфейса

Последним способом вызова контекстной справки является использование кнопки **What's this?**, иногда дублируемой комбинацией клавиш <Ctrl>+<F1> и соответствующей командой меню **Help**. Как уже говорилось, при нажатии на эту кнопку указатель мыши принимает форму наклонной стрелки со знаком вопроса. При щелчке левой кнопкой мыши на элементе пользовательского интерфейса, для которого определен этот вид контекстной справки, на экране появляется всплывающее окно, содержащее его краткое описание, как это показано на рис. 11.6.

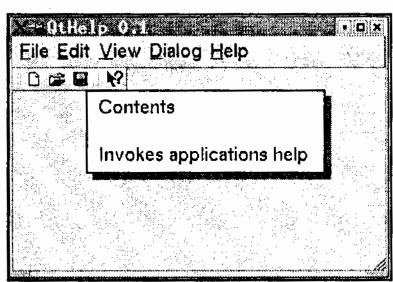


Рис. 11.6. Контекстная справка по команде меню приложения Qt

В приложениях Qt включение подобной контекстной справки в объект команды осуществляется вызовом функции `QAction::setWhatsThis`, аргумен-

том которой является текст выводимой справки. Как правило, этот текст состоит из заголовка, отделенного от текста справки двумя символами перевода строки (\n).

В тех случаях, когда контекстную справку нужно получить по элементам управления диалогового окна или по другим элементам пользовательского интерфейса, не являющимся командами меню или кнопками панели инструментов, разработчику следует использовать методы класса QWhatsThis. Так, для добавления контекстной информации в объект класса QWidget разработчику достаточно вызвать статическую функцию QWhatsThis::add, передав в первом ее аргументе указатель на объект класса QWidget, а во втором аргументе — текст выводимой справки. Для удаления контекстной справки из объекта окна применяется статическая функция QWhatsThis::remove.

Класс KAction, используемый приложениями KDE, имеет виртуальный приемник setWhatsThis, позволяющий задать текст контекстной справки по данному объекту, однако мастер создания приложений KDE не вызывает эти приемники для создаваемых им команд меню. Это тем более странно, что, в отличие от приложений Qt, в приложения KDE включена команда меню **Help | What's this?** (Справка | Что это?) и соответствующая ей комбинация клавиш <Ctrl>+<F1>.

Совет

Рассмотренные выше типы контекстной справки являются непременными атрибутами стандартных команд меню приложения. Поэтому при создании новых команд меню или кнопок панели инструментов, не имеющих соответствия среди команд меню, следует включать в создаваемый объект класса QAction, KAction или KToggleAction все типы контекстной справочной информации, включенные в стандартные команды мастером создания приложений, чтобы новые команды меню и кнопки панели инструментов ничем не отличались от стандартных.

Программирование командной справки

Прежде чем приступить к рассмотрению вопроса о программировании командной справки, следует сначала рассмотреть вопрос о включаемых в заготовку приложения файлах системы командной справки и используемом в них формате. При этом необходимо учесть, что форматы файлов командной справки приложений Qt и KDE различаются.

В приложениях KDE меню **Help** включается в недрах класса KMainWindow в его далеко не виртуальной функции createGUI. Это меню содержит практически полный набор команд, но под видом справки по приложению показывает справку по интегрированной среде KDE, а команда меню **Help | What's This?** в создаваемой заготовке не работает, поскольку ни для одного из элементов управления главного окна приложения не установлен выводи-

мый по этой команде текст. Создание подобного меню в заготовке можно расценивать только как тонкое издевательство над пользователем. Поэтому здесь мы рассмотрим только командную справку приложений Qt.

Формат файлов командной справки приложений Qt

Файлы командной справки, включаемые мастером в заготовку создаваемого им приложения Qt, написаны в соответствии с требованиями языка SGML (Standard Generalized Markup Language, Стандартный обобщенный язык разметки), позволяющего составлять спецификации языков разметки. Эти спецификации, носящие название DTD (Document Type Definition, Определение типа документа), определяют структуру документа и содержащиеся в нем ярлыки (tags).

Готовые спецификации DTD транслируются SGML для получения файлов в выходном формате. В большинстве случаев в качестве формата выходных файлов используется формат HTML, позволяющий получать требуемую справочную информацию по Интернету. Кроме того, этот формат поддерживается большинством персональных компьютеров. Недаром этот формат стал стандартом де-факто для справочных систем последних разработок Microsoft. Этот же формат используется KDE в своей справочной системе KDEHelp.

Для работы с файлами документации KDevelop использует пакет SGML-Tools, ранее называвшийся LinuxDoc. (Изменение названия пакета связано с тем, что он может работать в любой системе UNIX, а не только под Linux.) Первая версия пакета использует спецификацию, за которой оставлено имя LinuxDoc, и содержит набор утилит и файлов преобразования ярлыков LinuxDoc. LinuxDoc DTD является дальнейшим развитием Qwertz DTD, созданной для замены ярлыков в текстовой системе LaTeX и других подобных системах.

Во второй версии пакета SGML-Tools осуществлен переход на спецификацию DocBook DTD, имеющую расширенный набор ярлыков и лучше работающую с таблицами. Для обеспечения совместимости во вторую версию включен преобразователь формата LinuxDoc в формат DocBook.

Однако для создания справочной системы приложения Qt продолжают использовать первую версию пакета SGML-Tools, что объясняется следующими причинами:

- простотой написания документации в формате LinuxDoc;
- простотой установки первой версии пакета SGML-Tools;
- наличием в KDE утилиты ksgml2html, преобразующей файлы HTML, создаваемые утилитой sgml2html первой версии пакета SGML-Tools к стилю документации KDE.

Внимание!

Не все разработчики Linux разделяют любовь библиотеки Qt к первой версии пакета SGML-Tools и могут просто не включить ее в дистрибутив своей операционной системы. Поэтому, несмотря на описанные выше преимущества, вам, как и мне, придется пользоваться второй версией данного продукта.

При создании проекта Qt в среде разработки KDevelop в подкаталог docs/en помещается файл index.sgml, содержащий документацию на английском языке, и полученные из него файлы HTML. Файлы HTML уже включены в проект и для них определено, что при установке приложения они будут размещаться в каталоге KDE HTML. В документацию уже включена информация об имени проекта, номере версии и сведения о разработчике. Помимо этого в файл index.sgml включены содержание справочной системы, сведения по установке и описание лицензионных прав на приложение, основанное на лицензии GPL. Поэтому от разработчика требуется только внести информацию, касающуюся конкретного проекта.

Наличие в подкаталоге docs/en наряду с файлом index.sgml оттранслированных файлов HTML вызывает у некоторых разработчиков соблазн использовать эти файлы для создания справочной системы приложения вместо того, чтобы вносить все изменения в файл index.sgml и транслировать его для получения исправленных файлов HTML. Во многих случаях этот соблазн вызван незнанием языка SGML и нежеланием его учить. Такой подход приводит к возникновению проблем при локализации приложения, поскольку требует внесения изменений в набор файлов HTML, содержащих справку на каждом из языков, вместо внесения изменений в один файл SGML. Помимо повышенных трудозатрат такой подход не гарантирует синхронности справочной системы на различных языках.

Примечание

К сожалению, разработчики KDevelop постарались выбить основание из-под приведенных доводов, решив, что локализация приложений никому не нужна, и оставив только упоминание о ней в своей справочной системе. (По крайней мере, так обстоит дело в используемом мной Mandrake Linux 9.0.)

Для тех, кто согласен с приведенными доводами и готов изучить основы языка SGML, ниже будет приведено краткое описание его основных конструкций.

Поскольку файл SGML может задействовать различные DTD, он всегда должен начинаться с объявления используемого в нем DTD. Это объявление применяет ярлык `doctype`, поэтому первая строка файла index.sgml всегда имеет вид

```
<!doctype linuxdoc system>
```

т. е. сообщает транслятору, что данный файл использует LinuxDoc DTD.

После этого в документе определяется его стиль. В зависимости от назначения документа и его размера для задания стиля могут использоваться следующие ярлыки:

- <notes> — используется для коротких пояснений;
- <article> — документ представляет собой законченную статью, содержащую от 10 до 20 страниц. Этот стиль используется средой разработки KDevelop для большинства приложений;
- <report> — аналогичен стилю <article>, но документ имеет больший размер;
- <book> — используется для написания объемных документов, таких как руководства пользователя среды KDevelop;
- <slides> — задает последовательность вывода изображений на экран. Используется для создания презентаций. В большинстве случаев эти документы транслируются в документы LaTeX;
- <letter> — используется для написания писем;
- <telefax> — предназначен для создания факсимильных сообщений;
- <manpage> — применяется для создания страницы справочной системы.

Стиль документа определяет только его структуру, а не вид получаемого документа. Как уже говорилось выше, заготовки справочной системы приложений Qt имеют стиль <article>. Поскольку весь создаваемый документ должен иметь один стиль, в конце его располагается ярлык </article>.

После задания стиля документа в нем, как правило, помещается описание его титульной страницы. Титульная страница заготовки приложения Qt включает в себя ярлыки <title>, <author> и <date>, содержимое которых можно оставить без изменения, чтобы не нарушать привычного дизайна справочной системы приложения.

Между титульной страницей и содержимым документа могут помещаться ярлыки используемых в документе индексаторов. В LinuxDoc DTD определены следующие ярлыки индексаторов:

- <toc> — содержание;
- <lof> — список рисунков;
- <lot> — список таблиц.

Наличие любого из этих ярлыков не требует включения в документ соответствующего завершающего ярлыка.

При использовании стиля <article> содержимое документа разбивается на разделы разных уровней ярлыками <sect>, <sect1>, <sect2> и т. д. до последнего допустимого уровня разбиения. При использовании стиля документа <book> верхний уровень разбиения задается ярлыком <chapt>. За каж-

дым из ярлыков главы, раздела или подраздела следует его заголовок, завершаемый ярлыком <p>, за которым следует содержимое данного раздела, для форматирования которого могут быть использованы различные ярлыки языка SGML. Для выделения заголовка главы могут быть также использованы ярлыки <title> и </title>.

Пример использования описанных выше ярлыков можно найти в заготовке файла index.sgml, созданного мастером для приложения **Help**, которое будет рассматриваться позднее. Текст этого файла приведен в листинге 11.1.

Листинг 11.1. Файл index.sgml

```
<!doctype linuxdoc system>
<article>
<title>The QtHelp Handbook
<author> <tt></tt>
<date>Version 0.1, Sat Jan 11 13:08:27 MSK 2003
<abstract>
This Handbook describes QtHelp Version 0.1
</abstract>

<toc>

<sect>Introduction
<p>
<sect1>Changes
<p>

<sect>Installation
<p>
<sect1>How to obtain QtHelp
<p>

<sect1>Requirements
<p>

<sect1>Compilation and installation
<p>

In order to compile and install QtHelp on your system, type the following
in the base directory of the QtHelp distribution:

<tscreen><verb>
% ./configure
% make
% make install
</verb></tscreen>
```

```
<p>
Since QtHelp uses <verb>autoconf</verb> you should have not trouble
compiling it.
Should you run into problems please report them to the the author at
<htmlurl url="mailto:" name = "">


<p>
<sect> Usage <p>
<sect1> General Usage
<p>
<sect> Another Section
<p>

<sect>Questions and Answers<p>

<sect>Copyright<p>

QtHelp Copyright 2003 ,


This program is free software; you can redistribute it and/or modify
it under the terms of the GNU General Public License as published by
the Free Software Foundation; either version 2 of the License, or
(at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.

<p>
</article>
```

Этот файл автоматически транслируется при компиляции приложения и его можно просмотреть непосредственно в среде разработки KDevelop, если раскрыть каталог **Current Project** (Текущий проект), расположенный во вкладке **Books** (Книги) окна иерархических списков и выделить в нем строку **User Manual** (Руководство пользователя). При этом среда разработки KDevelop примет вид, изображенный на рис. 11.7.

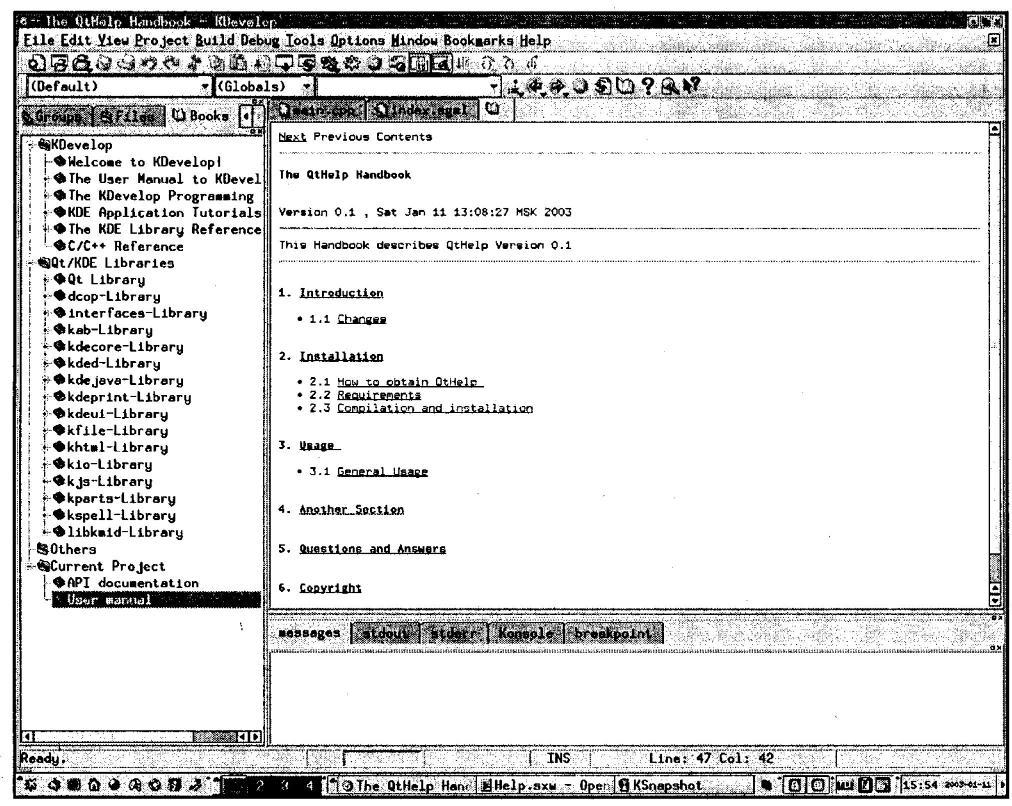


Рис. 11.7. Просмотр командной справки в среде KDevelop

Создание демонстрационного приложения Qt

Для демонстрации принципов создания различных видов контекстной справки в приложениях Qt рассмотрим приложение **QtHelp**, текст которого можно найти на прилагаемом к книге CD.

Чтобы самостоятельно создать это приложение:

1. Выберите команду меню **Project | New** (Проект | Создать). Появится диалоговое окно **ApplicationWizard** (Мастер создания приложений).
2. В окне иерархического списка диалогового окна в папке **Qt** выделите строку **Qt SDI** (Однооконное приложение **Qt**) и нажмите кнопку **Next** (Далее). Появится второе окно мастера **ApplicationWizard**.
3. В текстовое поле **Project name** (Имя проекта) введите имя проекта **QtHelp**, заполните остальные текстовые поля корректной информацией (или оставьте их без изменений, поскольку этот проект не будет распространяться) и нажмите кнопку **Create** (Создать). Появится последнее

окно мастера **ApplicationWizard** и начнется процесс создания заготовки приложения.

4. Как только станет доступной кнопка **Exit** (Выход), нажмите ее и выйдите из мастера создания приложений.
5. Выберите команду меню **File | New** (Файл | Создать), нажмите комбинацию клавиш <Ctrl>+<N> или кнопку **New** в панели инструментов. Появится диалоговое окно **New File** (Новый файл).
6. В окне списка раскрытой вкладки **General** (Общие свойства) выделите строку **Qt Designer File (*.ui)** (Файл приложения Qt Designer), в текстовое поле **Filename** введите имя файла helpdlg (расширение ui будет добавлено к нему автоматически) и нажмите кнопку **OK**. Появится диалоговое окно **Load decision** (Тип загрузки), предлагающее загрузить файл в текстовом виде.
7. Нажмите кнопку **No** (Нет). Откроется окно приложения Qt Designer by Trolltech.
8. В окне списка появившегося при старте приложения диалогового окна **New File** выделите значок **Dialog with Buttons (Bottom)** (Диалоговое окно с кнопками, расположенными снизу) и нажмите кнопку **OK**. В приложении появится пустая заготовка диалогового окна, а в панели **Property Editor/Signal Handlers** (Редактор свойств/Обработчики сигналов) — список свойств созданного окна.
9. В панели **Property Editor/Signal Handlers** выделите строку **name** и замените содержащееся в нем имя диалогового окна **MyDialog** именем **HelpDlg**.
10. В той же панели выделите строку **caption** и замените содержащийся в нем заголовок диалогового окна **MyDialog** заголовком **Dialog**.
11. Выберите команду меню **Tools | Display | TextLabel** (Сервис | Изображения | Статический текст) или нажмите кнопку **Text Label** в панели инструментов **Display** и щелкните левой кнопкой мыши в левом верхнем углу заготовки диалогового окна. Появится рамка статического текста.
12. В панели **Property Editor/Signal Handlers** выделите строку **text** и введите в связанное с ней текстовое поле строку **Enter some text** (Введите какой-нибудь текст).
13. Выберите команду меню **Tools | Input | LineEdit** (Сервис | Ввод | Текстовое поле) или нажмите кнопку **Line Edit** в панели инструментов **Input** и щелкните левой кнопкой мыши под только что введенным статическим текстом. На месте щелчка появится текстовое поле.
14. В панели **Property Editor/Signal Handlers** выделите строку **name** и измените имя текстового поля на **LineEdit**.

15. В той же панели выделите строку **toolTip** (подсказка) и введите в нее подсказку **This is an absolutely useless line edit** (Это абсолютно бесполезное текстовое поле).
16. Там же выделите строку **whatsThis** и введите в нее комментарий **This line edit was created for demonstration purposed only** (Это текстовое поле было создано только для примера).
17. Выделите текстовое поле с его заголовком и выберите команду меню **Layout | Lay Out Vertically** (Расположение | Расположить по вертикали), нажмите комбинацию клавиш <Ctrl>+<L> или кнопку **Lay Out Vertically** в панели инструментов **Layout**. Текстовое поле и его заголовок будут автоматически выровнены по вертикали.
18. Поместите вертикальные разделители над статическим текстом и под текстовым полем.
19. Поместите горизонтальные разделители слева и справа от текстового поля.
20. Щелкните левой кнопкой мыши на свободном месте диалогового окна и выберите команду меню **Layout | Lay Out in a Grid** (Расположение | Расположить в сетке), нажмите комбинацию клавиш <Ctrl>+<G> или кнопку **Lay Out Horizontally** в панели инструментов **Layout**. Элементы управления диалогового окна будут выровнены, как это показано на рис. 11.8.
21. Выберите команду меню **Tools | Connect Signal/Slots** (Сервис | Связать Сигналы/Приемники), нажмите клавишу <F3> или кнопку **Connect Signal/Slots** в панели инструментов **Tools** (кнопка должна "утопиться").
22. Поместите указатель мыши на кнопку **Help**, нажмите левую кнопку мыши и, не отпуская ее, переместите указатель мыши на свободное место в заготовке диалогового окна (за пределы рамок выравнивания). Появится диалоговое окно **Edit Connections** (Редактирование связей).
23. Нажмите кнопку **Edit Slots** (Редактировать приемники). Появится диалоговое окно **Edit Slots**.
24. Нажмите кнопку **New Slot** (Новый приемник), введите в текстовое поле **Slot** (Приемник) сигнатуру нового приемника `onHelp()` и нажмите кнопку **OK**. Диалоговое окно **Edit Slots** закроется и в списке **Slots** диалогового окна **Edit Connections** появится новый приемник.
25. В окне списка **Signals** (Сигналы) выделите сигнал `pressed()`, в окне списка **Slots** — приемник `onHelp()`. В окне списка **Connections** (Соединения) появится информация о новом соединении.
26. Нажмите кнопку **OK**. Диалоговое окно **Edit Connections** закроется.
27. Сохраните информацию о диалоговом окне в файле описания ресурсов и закройте приложение Qt Designer by Trolltech.

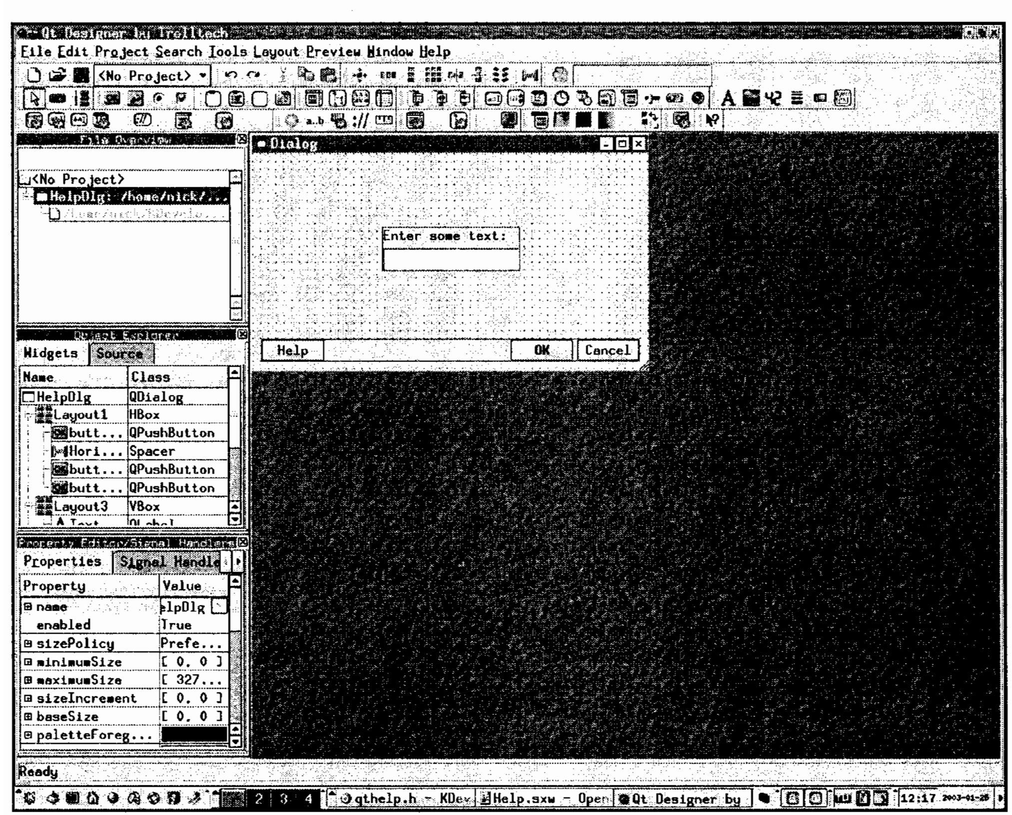


Рис. 11.8. Заготовка диалогового окна

28. В среде разработки KDevelop в окне иерархических списков раскройте вкладку **Classes** (Классы), щелкните правой кнопкой мыши на каталоге **Classes** и выберите в появившемся контекстном меню команду **New class** (Новый класс). Появится диалоговое окно **Class Generator**.

29. В текстовое поле **Classname** введите имя класса `HelpDlgImpl`, в текстовое поле **Baseclass** — имя базового класса `HelpDlg`, в текстовое поле **Documentation** — комментарий `User dialog implementation` (Реализация пользовательского диалогового окна), в группе **Additional Options** установите флажок **generate a QWidget-Childclass** и нажмите кнопку **OK**. В проект будут добавлены файлы заголовка и реализации, содержащие заготовку нового класса, и будут открыты окна редактирования этих файлов.

30. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса `HelpDlgImpl` и выберите в появившемся контекстном меню команду **Add slot** (Добавить приемник). Появится диалоговое окно **Class Properties** (Свойства класса), раскрытое на вкладке **Slots** (Приемники).

31. В текстовое поле **Declaration** введите сигнатуру приемника `onHelp()`, в группе **Modifiers** установите флажок **Virtual**, в текстовое поле **Documentation** введите комментарий `Invokes help` (Выводит справку) и нажмите кнопку **Apply**. В класс будет добавлен новый приемник.
32. В открывшемся после добавления приемника окне редактирования файла helpdlgimpl.cpp измените реализацию класса `HelpDlgImpl` в соответствии с текстом листинга 11.2.

Листинг 11.2. Реализация класса `HelpDlgImpl`

```
#include <qmessagebox.h>
#include <qprocess.h>
#include "helpdlgimpl.h"

HelpDlgImpl::HelpDlgImpl(QWidget *parent, const char *name, bool modal)
  : HelpDlg(parent,name, modal)
{
}
HelpDlgImpl::~HelpDlgImpl()
{
}

/** Выводит справку */
void HelpDlgImpl::onHelp()
{
 QProcess proc( QString("khelpcenter"), this);

 proc.addArgument( QDir::currentDirPath() + "/docs/en/index-5.html");
 proc.addArgument( "s5");

 if( !proc.start())
  QMessageBox::critical( this, tr("Error"),
   tr("Could not launch help"), QMessageBox::Ok, QMessageBox::NoButton);
}
```

33. Щелкните левой кнопкой мыши в окне редактирования файла helpdlgimpl.cpp и выберите в появившемся контекстном меню команду **Switch Header/Source** (Переключение файлов заголовка и реализации). Откроется окно редактирования файла helpdlgimpl.h.
34. В файле helpdlgimpl.h замените строку `HelpDlgImpl(QWidget *parent=0, const char *name=0);` строкой

    ```
    HelpDlgImpl(QWidget *parent=0, const char *name=0, bool modal = false);
    ```

35. В окне иерархических списков раскройте вкладку **Files**, раскройте в ней подкаталог qthelp/docs/en и щелкните левой кнопкой мыши на имени файла index.sgml. Откроется окно редактирования этого файла.

36. Перед строкой <sect>Questions and Answers<p> вставьте следующие строки:

```
<sect> Demo Dialog <p>
This is only a help for demo dialog.
```

37. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса QtHelpApp и выберите в появившемся контекстном меню команду **Add member variable** (Добавить переменную члена класса). Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Attributes** (Атрибуты).
38. В текстовое поле **Type** введите тип переменной QPopupMenu, в текстовое поле **Name** — идентификатор переменной ***dialogMenu**, в группе **Access** (Модификатор прав доступа) установите переключатель **Private** (Закрытая), в текстовое поле **Documentation** введите комментарий Dialog_menu contains command to invoke a demo dialog (Меню dialog_menu содержит команду для вывода на экран демонстрационного диалогового окна) и нажмите кнопку **Apply**. В класс будет добавлена новая переменная.
39. Повторите пункты 37 и 38 для добавления в класс переменной *dialogDialog, имеющей тип QAction и не имеющей комментария.
40. Повторите пункт 39 для добавления в класс переменной *helpContents.
41. Повторите пункт 39 для добавления в класс переменной *helpWhatsThis.
42. Во вкладке **Classes** окна иерархических списков щелкните правой кнопкой мыши по имени класса QtHelpApp и выберите в появившемся контекстном меню команду **Add slot**. Появится диалоговое окно **Class Properties**, раскрытое на вкладке **Slots**.
43. В текстовое поле **Declaration** введите сигнатуру приемника slotHelpContents(), в группе **Modifiers** установите флажок **Virtual**, в текстовое поле **Documentation** введите комментарий Invokes help (Выводит справку) и нажмите кнопку **Apply**. В класс будет добавлен новый приемник.
44. Повторите пункты 35 и 36 для добавления в класс QtHelpApp приемника slotHelpWhatsThis(), снабдив его комментарием Enters "What's this?" **mode** (Переводит в режим контекстной справки).
45. Повторите пункты 35 и 36 для добавления в класс QtHelpApp приемника slotDialogDialog(), снабдив его комментарием Invokes demo dialog (Выводит демонстрационное диалоговое окно). Флажок **Virtual** в группе **Modifiers** при этом устанавливать не нужно.
46. В открывшемся после добавления приемников окне редактирования файла qthelp.cpp измените эти приемники в соответствии с текстом листинга 11.3.

Листинг 11.3. Новые приемники класса QtHelpApp

```
/** Выводит демонстрационное диалоговое окно */
void QtHelpApp::slotDialogDialog()
{
 HelpDlgImpl dlg;

 dlg.exec();
}

/** Выводит справку */
void QtHelpApp::slotHelpContents()
{
 QProcess proc( QString("khelpcenter"), this);

 proc.addArgument( QDir::currentDirPath() + "/docs/en/index.html");

 if( !proc.start())
  QMessageBox::critical( this, tr("Error"), tr("Could not launch
  help"), QMessageBox::Ok, QMessageBox::NoButton);
}

/** Переводит в режим контекстной справки */
void QtHelpApp::slotHelpWhatsThis()
{
 QWhatsThis::enterWhatsThisMode();
}
```

47. В начале файла qthelp.cpp перед строкой `#include "qthelp.h"` вставьте строки

```
#include <qprocess.h>
#include "helpdlgimpl.h"
```

48. Измените функции `QtHelpApp::initActions` и `QtHelpApp::initMenuBar` в соответствии с текстом листинга 11.4.

Листинг 11.4. Функции initActions и initMenuBar

```
/** Инициализирует все объекты класса QActions приложения */
void QtHelpApp::initActions()
{
 QPixmap openIcon, saveIcon, newIcon;
 newIcon = QPixmap(filenew);
 openIcon = QPixmap(fileopen);
 saveIcon = QPixmap(filesave);
```

```
fileNew = new QAction(tr("New File"), newIcon, tr("&New"),
              QAccel::stringToKey(tr("Ctrl+N")), this);
fileNew->setStatusTip(tr("Creates a new document"));
fileNew->setWhatsThis(tr("New File\n\nCreates a new document"));
connect(fileNew, SIGNAL(activated()), this, SLOT(slotFileNew()));

fileOpen = new QAction(tr("Open File"), openIcon, tr("&Open..."),
                              0, this);
fileOpen->setStatusTip(tr("Opens an existing document"));
fileOpen->setWhatsThis(tr("Open File\n\nOpens an existing document"));
connect(fileOpen, SIGNAL(activated()), this, SLOT(slotFileOpen()));

fileSave = new QAction(tr("Save File"), saveIcon, tr("&Save"),
              QAccel::stringToKey(tr("Ctrl+S")), this);
fileSave->setStatusTip(tr("Saves the actual document"));
fileSave->setWhatsThis(tr("Save File.\n\nSaves the actual document"));
connect(fileSave, SIGNAL(activated()), this, SLOT(slotFileSave()));

fileSaveAs = new QAction(tr("Save File As"), tr("Save &as..."),
                              0, this);
fileSaveAs->setStatusTip(tr("Saves the actual document
                        under a new filename"));
fileSaveAs->setWhatsThis(tr("Save As\n\nSaves the actual document
                        under a new filename"));
connect(fileSaveAs, SIGNAL(activated()), this, SLOT(slotFileSave()));

fileClose = new QAction(tr("Close File"), tr("&Close"),
              QAccel::stringToKey(tr("Ctrl+W")), this);
fileClose->setStatusTip(tr("Closes the actual document"));
fileClose->setWhatsThis(tr("Close File\n\nCloses
                        the actual document"));
connect(fileClose, SIGNAL(activated()), this, SLOT(slotFileClose()));

filePrint = new QAction(tr("Print File"), tr("&Print"),
              QAccel::stringToKey(tr("Ctrl+P")), this);
filePrint->setStatusTip(tr("Prints out the actual document"));
filePrint->setWhatsThis(tr("Print File\n\nPrints out
                        the actual document"));
connect(filePrint, SIGNAL(activated()), this, SLOT(slotFilePrint()));

fileQuit = new QAction(tr("Exit"), tr("E&xit"),
              QAccel::stringToKey(tr("Ctrl+Q")), this);
fileQuit->setStatusTip(tr("Quits the application"));
fileQuit->setWhatsThis(tr("Exit\n\nQuits the application"));
connect(fileQuit, SIGNAL(activated()), this, SLOT(slotFileQuit()));
```

```
editCut = new QAction(tr("Cut"), tr("Cu&t"),
               QAccel::stringToKey(tr("Ctrl+X")), this);
editCut->setStatusTip(tr("Cuts the selected section
                    and puts it to the clipboard"));
editCut->setWhatsThis(tr("Cut\n\nCuts the selected section
                    and puts it to the clipboard"));
connect(editCut, SIGNAL(activated()), this, SLOT(slotEditCut()));

editCopy = new QAction(tr("Copy"), tr("&Copy"),
               QAccel::stringToKey(tr("Ctrl+C")), this);
editCopy->setStatusTip(tr("Copies the selected section
                          to the clipboard"));
editCopy->setWhatsThis(tr("Copy\n\nCopies the selected section
                          to the clipboard"));
connect(editCopy, SIGNAL(activated()), this, SLOT(slotEditCopy()));

editPaste = new QAction(tr("Paste"), tr("&Paste"),
               QAccel::stringToKey(tr("Ctrl+V")), this);
editPaste->setStatusTip(tr("Pastes the clipboard contents
                         to actual position"));
editPaste->setWhatsThis(tr("Paste\n\nPastes the clipboard contents
                         to actual position"));
connect(editPaste, SIGNAL(activated()), this, SLOT(slotEditPaste()));

viewToolBar = new QAction(tr("Toolbar"), tr("Tool&bar"),
                          0, this, 0, true);
viewToolBar->setStatusTip(tr("Enables/disables the toolbar"));
viewToolBar->setWhatsThis(tr("Toolbar\n\nEnables/disables
                            the toolbar"));
connect(viewToolBar, SIGNAL(toggled(bool)),
                  this, SLOT(slotViewToolBar(bool)));

viewStatusBar = new QAction(tr("Statusbar"), tr("&Statusbar"),
                          0, this, 0, true);
viewStatusBar->setStatusTip(tr("Enables/disables the statusbar"));
viewStatusBar->setWhatsThis(tr("Statusbar\n\nEnables/disables
                          the statusbar"));
connect(viewStatusBar, SIGNAL(toggled(bool)),
                 this, SLOT(slotViewStatusBar(bool)));

dialogDialog = new QAction(tr("Dialog"), tr("&Dialog..."), 0, this);
dialogDialog->setStatusTip(tr("Invokes demo dialog"));
dialogDialog->setWhatsThis(tr("Dialog\n\nInvokes demo dialog"));
connect(dialogDialog, SIGNAL(activated()),
                    this, SLOT(slotDialogDialog()));
```

```
  helpContents = new QAction(tr("Contents"), tr("&Contents..."),
                              0, this);
  helpContents->setStatusTip(tr("Invokes applications help"));
  helpContents->setWhatsThis(tr("Contents\n\nInvokes
                         applications help"));
  connect(helpContents, SIGNAL(activated()),
                    this, SLOT(slotHelpContents()));

  helpWhatsThis = new QAction(tr("What's This?"), tr("&What's This?"),
              QAccel::stringToKey(tr("Shift+F1")), this);
  helpWhatsThis->setStatusTip(tr("Invokes applications help"));
  helpWhatsThis->setWhatsThis(tr("Contents\n\nInvokes
                         applications help"));
  connect(helpWhatsThis, SIGNAL(activated()),
                   this, SLOT(slotHelpWhatsThis()));

  helpAboutApp = new QAction(tr("About"), tr("&About..."), 0, this);
  helpAboutApp->setStatusTip(tr("About the application"));
  helpAboutApp->setWhatsThis(tr("About\n\nAbout the application"));
  connect(helpAboutApp, SIGNAL(activated()),
                     this, SLOT(slotHelpAbout()));
}

/** Инициализирует меню приложения */
void QtHelpApp::initMenuBar()
{
  ///////////////////////////////////////////////////////////////////
  // ПАНЕЛЬ МЕНЮ

  ///////////////////////////////////////////////////////////////////
  // Команды меню File
  fileMenu=new QPopupMenu();
  fileNew->addTo(fileMenu);
  fileOpen->addTo(fileMenu);
  fileClose->addTo(fileMenu);
  fileMenu->insertSeparator();
  fileSave->addTo(fileMenu);
  fileSaveAs->addTo(fileMenu);
  fileMenu->insertSeparator();
  filePrint->addTo(fileMenu);
  fileMenu->insertSeparator();
  fileQuit->addTo(fileMenu);

  ///////////////////////////////////////////////////////////////////
  // Команды меню Edit
  editMenu=new QPopupMenu();
  editCut->addTo(editMenu);
```

```
    editCopy->addTo(editMenu);
    editPaste->addTo(editMenu);

    ///////////////////////////////////////////////////////////////////
    // Команды меню View
    viewMenu=new QPopupMenu();
    viewMenu->setCheckable(true);
    viewToolBar->addTo(viewMenu);
    viewStatusBar->addTo(viewMenu);
    ///////////////////////////////////////////////////////////////////
    // ВВЕДИТЕ СЮДА СПЕЦИФИЧЕСКИЕ КОМАНДЫ МЕНЮ ВАШЕГО ПРИЛОЖЕНИЯ

    ///////////////////////////////////////////////////////////////////
    // Команды меню Dialog
    dialogMenu=new QPopupMenu();
    dialogDialog->addTo(dialogMenu);

    ///////////////////////////////////////////////////////////////////
    // Команды меню Help
    helpMenu=new QPopupMenu();
    helpContents->addTo(helpMenu);
    helpWhatsThis->addTo(helpMenu);
    helpAboutApp->addTo(helpMenu);

    ///////////////////////////////////////////////////////////////////
    // КОНФИГУРАЦИЯ ПАНЕЛИ МЕНЮ
    menuBar()->insertItem(tr("&File"), fileMenu);
    menuBar()->insertItem(tr("&Edit"), editMenu);
    menuBar()->insertItem(tr("&View"), viewMenu);
    menuBar()->insertItem(tr("&Dialog"), dialogMenu);
    menuBar()->insertSeparator();
    menuBar()->insertItem(tr("&Help"), helpMenu);
}
```

49. Выберите команду меню **Debug | Start** (Отладка | Пуск) или нажмите кнопку **Debug** в панели инструментов. Появится окно приложения.
50. Выберите команду меню **Dialog | Dialog** (Диалоговое окно | Диалоговое окно). Появится демонстрационное диалоговое окно.
51. Поместите указатель мыши на текстовое поле **Enter some text**. Появится подсказка, изображенная на рис. 11.9.
52. Нажмите кнопку **Help**, расположенную в заголовке диалогового окна. Указатель мыши примет форму наклонной стрелки со знаком вопроса.
53. Щелкните левой кнопкой мыши на текстовом поле **Enter some text**. Появится контекстная справка, изображенная на рис. 11.10.

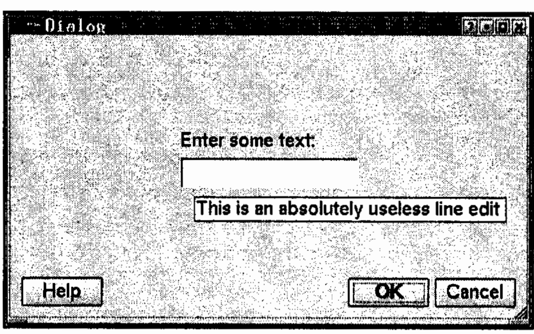


Рис. 11.9. Вывод подсказки

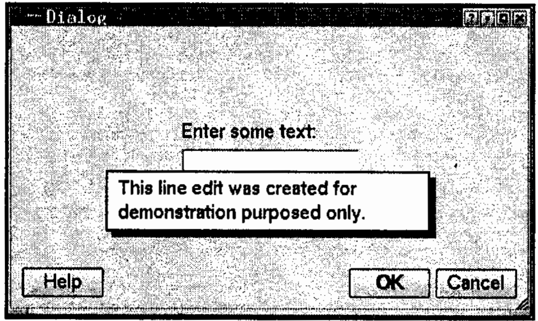


Рис. 11.10. Контекстная справка

54. Нажмите кнопку **Help**, расположенную в нижней части диалогового окна. Появится окно **KDE HelpCenter**, изображенное на рис. 11.11, и содержащее информацию по данному диалоговому окну.
55. Закройте окно **KDE HelpCenter**.
56. Закройте диалоговое окно и выберите команду меню **Help | Contents** (Справка | Вызов справки). Появится окно **KDE HelpCenter**, изображенное на рис. 11.12 и содержащее информацию по всему приложению.
57. Выберите команду меню **Help | What's This?** (Справка | Что это?) или нажмите комбинацию клавиш <Shift>+<F1>. Указатель мыши примет форму наклонной стрелки со знаком вопроса.
58. Щелкните левой кнопкой мыши на одной из кнопок панели инструментов. Появится контекстная информация по этой кнопке.
59. Закройте приложение.

Меню **Help** заготовки приложения Qt практически пусто, поэтому почти вся работа по созданию его справочной системы ложится на плечи разработчика. Он избавлен только от написания подсказок и контекстной справки для стандартных команд приложения.

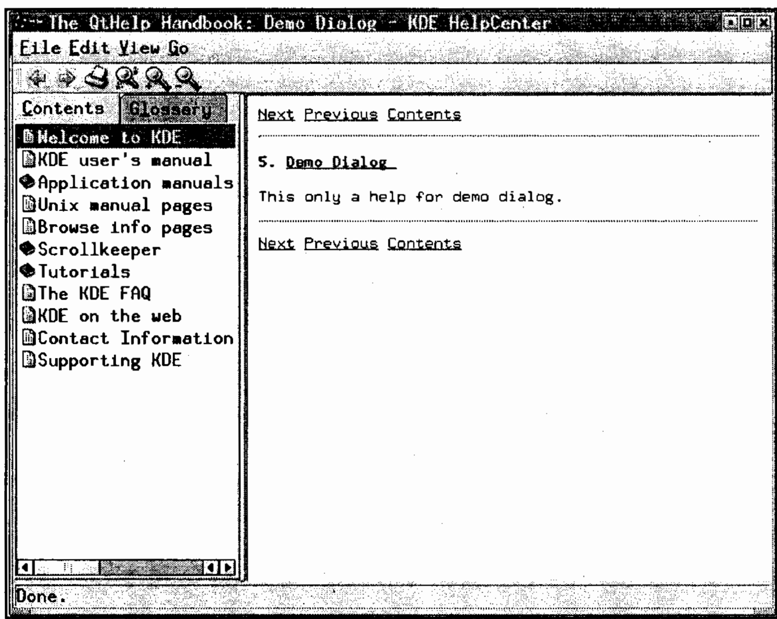


Рис. 11.11. Справка по диалоговому окну

The QtHelp Handbook - KDE HelpCenter
File Edit View Go
Contents
Glossary
Welcome to KDE
KDE user's manual
Application manuals
Unix manual pages
Browse info pages
Scrollkeeper
Tutorials
The KDE FAQ
KDE on the web
Contact Information
Supporting KDE

Next Previous Contents

The QtHelp Handbook

Version 0.1 , Thu Jan 16 12:02:05 MSK 2003

This Handbook describes QtHelp Version 0.1

1. Introduction
- 1.1 Changes

2. Installation
- 2.1 How to obtain QtHelp
- 2.2 Requirements
- 2.3 Compilation and installation

3. Usage
- 3.1 General Usage

4. Another Section

Рис. 11.12. Справка по приложению

Проще всего включить в приложение вызов командной справки из меню и сопоставить ей стандартную комбинацию клавиш. Для этого достаточно создать эту команду, указав в ней нужную комбинацию клавиш, и вызвать в приемнике этой команды (в данном случае в функции `QtHelpApp::slotHelpWhatsThis`) статическую функцию `QWhatsThis::enterWhatsThisMode`, выполняющую все необходимые действия. При рассмотрении этой команды следует обратить внимание на то, что она никак не связана с соответствующей ей кнопкой панели инструментов, добавленной туда вызовом статической функции `QWhatsThis::whatsThisButton`.

Вывод командной справки приложения осуществляется в приемнике `QtHelpApp::slotHelpContents`, обрабатывающем выбор пользователем соответствующей команды меню. Мне не удалось обнаружить в библиотеке Qt никаких стандартных средств для выполнения этой задачи, поэтому я воспользовался стандартными средствами запуска дочернего процесса. Поскольку мне не хотелось формировать списки строк, создавая для этого специальный объект, я выбрал более простую форму запуска процесса, позволяющую работать с отдельными строками.

Для работы с процессом, осуществляющим вывод командной справки приложения, использовался объект класса `QProcess`, конструктору которого в качестве аргументов были переданы имя приложения, осуществляющего вывод командной справки в приложениях KDE, и указатель на родительский объект. Поскольку для вывода командной справки в аргументах командной строки запускаемого приложения необходимо передать путь к корневому файлу этой справки, для созданного объекта вызывается функция `QProcess::addArgument`. Так как приложение будет распространяться в исходном виде без преобразования его в пакет, в нем использован относительный путь к файлу справки. В качестве базы взят каталог исполняемого файла, путь к которому возвращается статической функцией `QDir::currentDirPath`.

Запуск процесса осуществляется вызовом функции `QProcess::start`. Если в процессе запуска приложения возникла ошибка, сообщение о ней пользователь получает в окне, выводимом статической функцией `QMessageBox::critical`.

Для демонстрации способов вывода справочной информации в диалоговом окне нами было создано простейшее диалоговое окно и в приложение была добавлена команда меню для его вызова. Основные операции по включению справочной информации в диалоговое окно производятся в приложении Qt Designer by Trolltech при создании элементов управления диалогового окна. Однако сигналы кнопки **Help**, включаемой мастером в заготовку диалогового окна наравне с кнопками **OK** и **Cancel**, в отличие от этих кнопок, не связаны ни с какими приемниками. Поэтому, чтобы не выглядеть глупо, разработчик должен или удалить эту кнопку из своего диалогового окна, или создать приемник и связать его с сигналом этой кнопки.

Программирование вывода справки по диалоговому окну мало чем отличается от программирования вывода справки по всему приложению. Поскольку диалоговое окно является частью приложения, справка по нему должна быть частью справки по всему приложению. Поэтому при вызове этой справки нужно указать не только файл, в котором она располагается, но и имя раздела, который нужно вывести. Для этого в приемник `HelpDlgImpl::onHelp` включен второй вызов функции `QProcess::addArgument`, добавляющей имя раздела в список аргументов командной строки запускаемого приложения.

ПРИЛОЖЕНИЕ 1

Что на CD

На прилагаемом к данной книге CD помещены исходные тексты описанных в ней примеров. Каждый пример хранится в отдельном каталоге и представляет собой полноценный проект, который можно загружать в среду разработки и компилировать.

Имя каждого каталога совпадает с именем содержащегося в нем проекта. Ниже приведен список проектов с указанием глав, к которым они относятся.

- Brush — приложение, демонстрирующее принципы работы с кистью и заполнение областей в окне *(см. главу 6)*.
- DateTime — приложение, демонстрирующее принципы работы с элементами управления, позволяющими вводить и выводить дату и время *(см. главу 3)*.
- DDB — приложение, демонстрирующее принципы работы с аппаратно-зависимыми битовыми образами *(см. главу 6)*.
- Dialog — приложение, демонстрирующее принципы создания диалоговых приложений и принципы работы с диалоговыми окнами *(см. главу 2)*.
- DIB — приложение, демонстрирующее принципы работы с аппаратно-независимыми битовыми образами *(см. главу 6)*.
- FileClien — приложение клиента, использующего сервер FileServer *(см. главу 10)*.
- FileServer — приложение сервера, сохраняющего данные в файле и читающего их из него *(см. главу 10)*.
- KDEEdit — приложение текстового редактора, использующего классы библиотеки KDE *(см. главу 8)*.

- KDEMDI — многооконное приложение, использующее классы библиотеки KDE *(см. главу 4)*.
- KDETools — приложение, демонстрирующее принципы работы с панелью инструментов и строкой состояния с использованием классов библиотеки KDE *(см. главу 5)*.
- Line — приложение, демонстрирующее принципы работы с пером и рисование линий в окне *(см. главу 6)*.
- ListBox — приложение, демонстрирующее принципы работы с окном списка *(см. главу 3)*.
- Progress — приложение, демонстрирующее принципы работы с линейным индикатором и линейным регулятором *(см. главу 3)*.
- QtEdit — приложение текстового редактора, использующего классы библиотеки Qt *(см. главу 8)*.
- QtHelp — приложение, демонстрирующее принципы создания справочной системы в приложениях Qt *(см. главу 11)*.
- QtMDI — многооконное приложение, использующее классы библиотеки Qt *(см. главу 4)*.
- QtTools — приложение, демонстрирующее принципы работы с панелью инструментов и строкой состояния с использованием классов библиотеки Qt *(см. главу 5)*.
- SimpleClient — приложение клиента, использующего сервер SimpleServer *(см. главу 10)*.
- SimpleServer — простейшее приложение сервера *(см. главу 10)*.
- TabDialog — диалоговое приложение, демонстрирующее принципы работы с диалоговыми окнами с вкладками *(см. главу 2)*.
- Text — приложение, демонстрирующее принципы вывода текста в окне, принципы сохранения информации в файле и чтения ее из него *(см. главы 6 и 7)*.
- Wizard — диалоговое приложение, демонстрирующее принципы работы с окнами мастеров *(см. главу 2)*.

ПРИЛОЖЕНИЕ 2

Ресурсы Интернета

Для получения дополнительной информации по рассматриваемым в данной книге вопросам вы можете обратиться на следующие сайты:

- **www.kdevelop.org** — сайт компании KDevelop. Содержит полную информацию о данной компании и выпускаемых ею продуктах, включая их исходные тексты;
- **www.trolltech.com** — сайт компании TrollTech. Содержит полную информацию о данной компании и выпускаемых ею продуктах, включая их исходные тексты;
- **www.kde.org** — сайт интегрированной среды KDE. Содержит полную информацию о данной интегрированной среде и связанными с ней вопросами.

Предметный указатель

M

Meta Object Compiler 18

Б

Битовые образы 206
◊ аппаратно-зависимые 206
◊ аппаратно-независимые 211

Д

Демонстрационное приложение:
◊ Brush 193
◊ DateTime 105
◊ DDB 207
◊ Dialog 39
◊ DIB 211
◊ FileClient 315
◊ FileServer 308
◊ KDEEdit 250
◊ KDEMDI 136
◊ KDETools 169
◊ Line 187
◊ ListBox 88
◊ Progress 98
◊ QtEdit 232
◊ QtHelp 340
◊ QtMDI 115
◊ QtTools 147
◊ SimpleClient 301
◊ SimpleServer 301
◊ TabDialog 67
◊ Text 203, 214
◊ Wizard 77

Добавление:
◊ переменных 149
◊ приемников 148
◊ сигналов 165
◊ функций 166

И

Изменение:
◊ меню 147, 169
◊ панели инструментов 152, 175
◊ строки состояния 162, 183

К

Кисть 193
Класс:
◊ KAccel 31
◊ KAction 144, 174, 228
◊ KActionCollection 169
◊ KActionMenu 144
◊ KConfig 141, 181, 231
◊ KDateWidget 112
◊ KEdit 264
◊ KMainWindow 334
◊ KProcess 305
◊ KStatusBar 183, 333
◊ KStdAction 143, 169, 265
◊ KToggleAction 181
◊ KURL 138, 219, 268, 321
◊ QAccel 31
◊ QAction 132, 151, 331
◊ QApplication 24
◊ QArray 282
◊ QAsciiCache 296

◊ QBrush 196
◊ QByteArray 281
◊ QCache 296
◊ QCollection 280
◊ QColor 210
◊ QDateEdit 111
◊ QDateTime 113
◊ QDateTimeEdit 112
◊ QDict 292
◊ QDictIterator 293
◊ QDropSite 32
◊ QEvent 23
◊ QFile 122, 138, 219, 268, 312
◊ QFileInfo 122
◊ QFocusEvent 27
◊ QIconSet 161
◊ QImage 212
◊ QIntCache 296
◊ QKeyEvent 31
◊ QLabel 167
◊ QList 128, 287
◊ QListBox 96
◊ QListBoxItem 97
◊ QListIterator 287
◊ QMap 290
◊ QMapIterator 291
◊ QMouseEvent 29
◊ QMultiLineEdit 232
◊ QObject 120
◊ QPaintDevice 192
◊ QPainter 191
◊ QPixmap 132, 161, 211
◊ QPoint 205
◊ QPopupMenu 134, 152
◊ QProcess 353
◊ QProgressBar 104
◊ QQueue 294
◊ QRect 210
◊ QSignal 19
◊ QSlider 104
◊ QStack 294
◊ QStatusBar 162, 333
◊ QString 205
◊ QTextStream 269
◊ QTimeEdit 112
◊ QToolbar 161
◊ QToolTip 331
◊ QToolTipGroup 332
◊ QValueList 285
◊ QValueListIterator 286
◊ QValueStack 285
◊ QVector 283
◊ QWhatsThis 334

Класс коллекций 278
◊ карта отображений 290
◊ массив 281
◊ связный список 284

Клиент 300

Ключевое слово:
◊ signals 18
◊ slots 20

Концепция Документ/Представление 114

Копирование данных:
◊ глубокое 282
◊ поверхностное 282

М

Макрос:
◊ Q_OBJECT 18
◊ SIGNAL 21
◊ SLOT 21

Мастер:
◊ ApplicationWizard 39
◊ Create New Icon 153

Меню 146

О

Окно:
◊ вывод битовых образов 206
◊ вывод линий 187
◊ вывод текста 203
◊ закраска фигур 193
◊ перерисовка 197
◊ синхронизация 201

П

Пакет SGML-Tools 335

Панель инструментов 146

Перечислимый тип:
◊ KProcess:
  ▫ Communication 307
  ▫ RunMode 306
◊ KToolBar:
  ▫ BarPosition 182
◊ QImage:
  ▫ Endian 213

Продолжение рубрики см. на с. 360

Перечислимый тип *(прод.)*:
- Qt:
  - ButtonState 29
  - Key 30
- QWidget:
  - FocusPolicy 27

Перо 192

Приемник 17
- KAction:
  - setStatusText 174
  - setWhatsThis 334
- QAction:
  - setOn 161
- QMultiLineEdit:
  - clear 269
  - copy 267
  - cut 267
  - paste 267
  - redo 249
  - undo 249
- QStatusBar:
  - message 333
- QWizard:
  - back 87
  - next 86

Приложение:
- KIconEdit 153
- Qt Designer 40

Процесс 299

С

Сервер 300

Сигнал 17
- KProcess:
  - processExited 306
  - receivedStderr 305
  - receivedStdout 305
  - wroteStdin 321
- QAction:
  - activated 151
  - toggled 161
- QMultiLineEdit:
  - copyAvailable 266
  - redoAvailable 266
  - textChanged 266
  - undoAvailable 266
- QPopupMenu:
  - aboutToShow 134

Событие:
- обработка 22
- синтетическое 25

Создание:
- командной справки 334
- контекстной справки 330

Сохранение конфигурации приложения 181

Спецификация LinuxDoc 335

Справка:
- командная 330
- контекстная 330
- по ключевому слову 330
- по методике решения задачи 330
- формат файлов 335

Строка состояния 147

Ф

Функция:
- istream:
  - operator>> 314
  - read 314
- KAction:
  - plug 174
  - popupMenu 144
  - setEnabled 141, 228, 265
  - setStatusText 144, 332
- KActionMenu:
  - insert 145
- KConfigBase:
  - readBoolEntry 182
  - readNumEntry 182
  - readPathEntry 231
  - setGroup 181, 231
  - writeEntry 181, 231
- KDateWidget:
  - date 112
  - setDate 112
- KDEEditDoc:
  - setModified 266
- KEdit:
  - insertText 269
  - replace 268
  - saveText 269
  - search 268
- KFileDialog:
  - getOpenURL 224, 321
  - getSaveURL 224, 322
- KInstance:
  - config 141
- KIO::NetAccess:
  - download 138, 220, 268
  - removeTempFile 139, 220, 269

◊ KMainWindow:
 ▫ createGUI 145, 174, 334
 ▫ toolBar 181
◊ KProcess:
 ▫ clearArguments 321
 ▫ getPid 306
 ▫ isRunning 307
 ▫ kill 307
 ▫ operator<< 305
 ▫ setExecutable 322
 ▫ start 306
 ▫ writeStdin 323
◊ KRecentFilesAction:
 ▫ addURL 224
 ▫ saveEntries 182
◊ KStatusBar:
 ▫ changeItem 186
 ▫ insertFixedItem 185
 ▫ insertItem 142, 185
◊ KToggleAction:
 ▫ isChecked 182
 ▫ setChecked 182
 ▫ setStatusText 332
◊ KToolBar:
 ▫ barPos 182
 ▫ hide 182
 ▫ setBarPos 182
 ▫ setFullSize 181
 ▫ show 182
◊ KURL:
 ▫ fileName 224
 ▫ isValid 268
 ▫ path 219, 269
◊ KXMLGUIClient:
 ▫ actionCollection 143, 169
◊ ostream:
 ▫ operator<< 313
 ▫ write 313
◊ QAccel:
 ▫ activated 31
 ▫ connectItem 31
 ▫ insertItem 31
 ▫ stringToKey 132
◊ QAction:
 ▫ addTo 134, 152
 ▫ setOn 128
 ▫ setStatusTip 132, 151, 332
 ▫ setToolTip 151, 331
 ▫ setWhatsThis 132, 151, 333
◊ QActionGroup:
 ▫ setEnabled 135
◊ QApplication:
 ▫ notify 23
 ▫ postEvent 36
 ▫ processEvents 36
 ▫ sendEvent 35
 ▫ setMainWidget 66
◊ QArray:
 ▫ copy 283
 ▫ detach 283
 ▫ operator= 283
◊ QButton:
 ▫ setAccel 31
◊ QCheckBox:
 ▫ isChecked 77
◊ QCollection:
 ▫ deleteItem 280
 ▫ newItem 280
 ▫ setAutoDelete 122, 280
◊ QColor:
 ▫ qRgb 213
◊ QComboBox:
 ▫ insertItem 65
◊ QDateEdit:
 ▫ date 112
 ▫ setDate 112
◊ QDateTime:
 ▫ date 113
 ▫ time 113
◊ QDateTimeEdit:
 ▫ dateEdit 112
 ▫ timeEdit 113
◊ QDialog:
 ▫ exec 66
◊ QDict:
 ▫ find 294
 ▫ insert 293
 ▫ operator[] 293
 ▫ remove 293
◊ QDictIterator:
 ▫ current 293
 ▫ currentKey 293
 ▫ toFirst 293
◊ QDir:
 ▫ currentDirPath 225, 353
◊ QEvent:
 ▫ type 23
◊ QFile:
 ▫ close 122
 ▫ flush 220, 314
 ▫ open 122, 219, 268
 ▫ readBlock 220, 314
 ▫ writeBlock 219, 314

Продолжение рубрики см. на с. 362

Функция *(прод.)*:
- ◊ QFrameset:
  - ▫ FrameStyle 129
- ◊ QGVector:
  - ▫ compareItems 283
- ◊ QImage:
  - ▫ scanLine 213
- ◊ QIntCache:
  - ▫ find 298
  - ▫ insert 297
  - ▫ setMaxCost 297
- ◊ QKeyEvent:
  - ▫ ascii 30
  - ▫ ignore 30
  - ▫ key 30
  - ▫ state 31
- ◊ QLabel:
  - ▫ setText 167
- ◊ QLineEdit:
  - ▫ clear 97
  - ▫ setText 65, 97
  - ▫ text 97
- ◊ QList:
  - ▫ append 289
  - ▫ at 289
  - ▫ current 289
  - ▫ first 269
  - ▫ isEmpty 135
  - ▫ remove 289
  - ▫ removeFirst 289
  - ▫ removeLast 289
- ◊ QListBox:
  - ▫ clear 97
  - ▫ currentItem 97
  - ▫ insertItem 97
  - ▫ removeItem 97
- ◊ QListBoxItem:
  - ▫ text 97
- ◊ QListIterator:
  - ▫ toFirst 289
- ◊ QMainWindow:
  - ▫ menuBar 135, 152
  - ▫ setCentralWidget 129
  - ▫ statusBar 167
- ◊ QMap:
  - ▫ begin 291
  - ▫ contains 291
  - ▫ end 291
  - ▫ find 291
  - ▫ operator[] 291
- ◊ QMapIterator:
  - ▫ data 291
  - ▫ key 291
- ◊ QMenuBar:
  - ▫ insertItem 135
- ◊ QMenuData:
  - ▫ clear 135
  - ▫ insertItem 135, 152
  - ▫ insertSeparator 134
  - ▫ setAccel 31
  - ▫ setItemChecked 135
  - ▫ setItemParameter 135
- ◊ QMessageBox:
  - ▫ critical 353
  - ▫ information 151
- ◊ QMouseEvent:
  - ▫ button 29
  - ▫ pos 192
  - ▫ state 29
  - ▫ x 168
  - ▫ y 168
- ◊ QMultiLineEdit:
  - ▫ insertLine 306
  - ▫ numLines 323
  - ▫ setAlignment 250
  - ▫ textLine 323
- ◊ QObject:
  - ▫ connect 21, 132, 151
  - ▫ disconnect 22
  - ▫ event 23
  - ▫ eventFilter 33
  - ▫ installEventFilter 33
  - ▫ removeEventFilter 33
  - ▫ tr 128, 168, 206
- ◊ QPaintDevice:
  - ▫ bitBlt 211
- ◊ QPainter:
  - ▫ begin 192
  - ▫ drawImage 213
  - ▫ drawRect 197
  - ▫ drawText 205
  - ▫ end 192
  - ▫ fillRect 197
  - ▫ lineTo 193
  - ▫ moveTo 192
  - ▫ setPen 192
- ◊ QPen:
  - ▫ setWidth 192
  - ▫ width 192
- ◊ QPopupMenu:
  - ▫ setCheckable 134
- ◊ QProcess:
  - ▫ addArgument 353
  - ▫ start 353

◊ QProgressBar:
- progress 104
- setProgress 104

◊ QQueue:
- clear 295
- count 295
- current 295
- dequeue 295
- enqueue 295
- head 295
- isEmpty 295

◊ QRect:
- contains 211
- setLeft 210
- setRight 210

◊ QSpinBox:
- setValue 65

◊ QStack:
- clear 296
- count 296
- current 296
- isEmpty 296
- pop 296
- push 295
- top 296

◊ QStatusBar:
- addWidget 168
- clear 162
- message 136, 151

◊ QString:
- arg 168, 205
- isEmpty 324
- sprintf 306
- toInt 65
- truncate 306

◊ QTextEdit:
- find 250

◊ QTimeEdit:
- setTime 113
- time 113

◊ QToolBar:
- addSeparator 136

◊ QToolTip:
- add 331
- remove 332

◊ QValueList:
- append 286
- begin 286
- contains 287
- count 286
- end 286
- find 287
- findIndex 287
- fromLast 286
- insert 286
- prepend 286

◊ QValueListIterator:
- operator-- 287
- operator++ 286

◊ QVector:
- count 284
- insert 284
- operator[] 284
- remove 284

◊ QWhatsThis:
- add 334
- enterWhatsThisMode 353
- remove 334
- whatsThisButton 136, 353

◊ QWidget:
- closeEvent 126
- event 24
- focusInEvent 27
- focusOutEvent 27
- height 210
- pos 35
- rect 201
- repaint 126
- setAcceptDrops 32
- setCaption 128, 224, 225, 322
- setFocusPolicy 27
- setMouseTracking 30
- size 35
- width 210

◊ QWizard:
- currentPage 87
- setFinishEnabled 87
- setNextEnabled 87
- showPage 87

◊ QWorkspace:
- windowList 135

Ш

Шаблон 270
◊ класса 275
◊ функции 273

Я

Язык SGML 335

Секунов Николай Юрьевич, автор книг "Самоучитель Visual C++ 6", "Обработка звука на PC", "Самоучитель C#", "Самоучитель Visual C++.NET", а также переводчик и технический редактор ряда книг по программированию на C/C++ и C#.

Освойте универсальную кроссплатформенную систему разработки приложений

Программирование на C++ в Linux

Вниманию читателей представлена мощная система разработки приложений на языке C++, которая может использоваться в средах Linux и Windows. Она предоставляет пользователю удобный интерфейс, а также широкий набор встроенных утилит, существенно упрощающий процесс разработки. Подробные комментарии к примерам помогут в изучении основных приемов использования среды программирования KDevelop и позволят в короткий срок научиться создавать работоспособные приложения. Книга будет полезна как специалистам, имеющим большой опыт работы на языке C++ в среде Windows и желающим перейти на программирование в среде Linux, так и начинающим программистам.

Компакт-диск с примерами программ

ИНТЕРНЕТ-МАГАЗИН
www.computerbook.ru

Уровень пользователя	Средний/высокий
Категория	Программирование

ISBN 5-94157-355-3

БХВ-Петербург 198005, Санкт-Петербург, Измайловский пр., 29
E-mail: mail@bhv.ru Internet: www.bhv.ru Тел.: (812) 251-4244 Факс: (812) 251-1295