



Николай Секунов



Программирование на C++ в Linux



+ CD-ROM

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



МАСТЕР ПРОГРАММ

Николай Секунов

Программирование на C++ в Linux

Санкт-Петербург

«БХВ-Петербург»

2003

Секунов Н. Ю.

С28 Программирование на С++ в Linux. — СПб.: БХВ-Петербург, 2003. — 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 приложениях на основе взаимодействующих 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	<i>Екатерина Кондукова</i>
Зам. главного редактора	<i>Анатолий Адаменко</i>
Зав. редакцией	<i>Григорий Добин</i>
Редактор	<i>Наталья Сержантова</i>
Компьютерная верстка	<i>Ольги Сергиенко</i>
Корректор	<i>Зинаида Дмитриева</i>
Оформление серии	<i>Via Design</i>
Дизайн обложки	<i>Игоря Цырульниковца</i>
Зав. производством	<i>Николай Тверских</i>

Лицензия ИД № 02429 от 24.07.00. Подписано в печать 10.06.03.

Формат 70×100^{1/16}. Печать офсетная. Усл. печ. л. 29,67.

Тираж 3000 экз. Заказ №

"БХВ-Петербург", 198005, 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.

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 Кулих, 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 C 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 используемых для этого сигналов и приемников, приведены сведения о системе обработки событий в приложениях KDevelop, а также представлены используемые 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>+`.
- ❑ 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, оформляется следующим образом:

Внимание!

На эту информацию следует обратить особое внимание.

- ❑ Советы читателю оформляются так:

Совет

Это всего лишь совет.

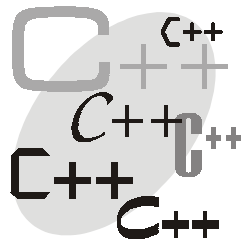
Требования к аппаратным средствам и программному обеспечению

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 `autotools` (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(QPaintEvent*)`. 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_MouseButtonDoubleClick` — 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 отпуске 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 `QPushButton::setAccel` имеет только один аргумент, в котором передается комбинация клавиш, иницилирующая 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 послышки 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-