Николай Прохоренок

PRO

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ

Qt 6

Разработка оконных приложений на C++

Управление окном приложения
Обработка сигналов и событий
Размещение компонентов внутри окна
Основные компоненты
Списки и таблицы
Работа с графикой и изображениями
Графическая сцена
Диалоговые окна
SDI- и MDI-приложения
Редактор Qt Creator

+ Материалы на www.bhv.ru

bhv®

Николай Прохоренок

Qt 6

Разработка оконных приложений на C++

Санкт-Петербург
«БХВ-Петербург»
2022

УДК 004.738.5+004.43
ББК 32.973.26-018.2
П84

Прохоренок Н. А.

П84 Qt 6. Разработка оконных приложений на C++. — СПб.: БХВ-Петербург, 2022. — 512 с.: ил. — (Профессиональное программирование)

ISBN 978-5-9775-1180-3

Описываются базовые возможности библиотеки Qt, позволяющей создавать приложения с графическим интерфейсом на языке C++. Книга ориентирована на тех, кто уже знаком с языком программирования C++ и хотел бы научиться разрабатывать оконные приложения. Рассматриваются способы обработки событий, управление свойствами окна, создание формы, а также все основные компоненты (кнопки, текстовые поля, списки, таблицы, меню и др.) и варианты их размещения внутри окна. Книга содержит большое количество практических примеров, помогающих начать разрабатывать приложения с графическим интерфейсом самостоятельно. Она будет полезна в качестве самоучителя для начинающих разработчиков. Читатели, уже имеющие опыт, могут использовать ее как удобный справочник.

Для программистов

УДК 004.738.5+004.43
ББК 32.973.26-018.2

Группа подготовки издания:

Руководитель проекта	*Евгений Рыбаков*
Зав. редакцией	*Людмила Гауль*
Компьютерная верстка	*Ольги Сергиенко*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Зои Канторович*

"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

ISBN 978-5-9775-1180-3

Оглавление

Введение

Добро пожаловать в мир Qt!

Qt — это популярная библиотека, позволяющая создавать приложения с графическим интерфейсом на языке C++. Она очень проста в использовании и идеально подходит для разработки приложений практически любой сложности. Библиотека является кросс-платформенной, поэтому мы можем создавать оконные приложения под Windows, Linux и Mac, а также мобильные приложения под Android и iOS. В этой книге мы рассмотрим процесс создания оконных приложений применительно к операционной системе Windows. Для сборки приложений будем пользоваться компилятором MinGW, а для набора кода — редактором Qt Creator, входящим в состав библиотеки Qt.

Давайте рассмотрим структуру книги.

- *Глава 1* является вводной. Мы установим необходимое программное обеспечение, настроим среду разработки, скомпилируем и запустим первую программу. Кроме того, вкратце разберемся со структурой программы.
- В *главе 2* мы научимся работать с символами и строками, а также со списками, состоящими из строк.
- В *главе 3* рассматривается жизненный цикл окна приложения: создание окна, его отображение, сокрытие и закрытие, сворачивание и разворачивание до максимального размера или во весь экран, отображение окна поверх всех окон, а также возможность блокировки других окон приложения до момента закрытия окна. Мы научимся изменять размеры окна, управлять его местоположением на экране, менять значок в заголовке окна и др.
- При взаимодействии пользователя с окном происходят *события*. В ответ на события генерируются определенные сигналы — своего рода извещения о том, что пользователь выполнил какое-либо действие или в самой системе возникло некоторое условие. События являются важнейшей составляющей приложения с графическим интерфейсом, поэтому необходимо знать, как назначить обработчик события, как удалить обработчик, а также уметь правильно обработать событие. Обработку событий мы рассмотрим в *главе 4*.
- *Глава 5* полностью посвящена менеджерам компоновки (контейнерам и панелям), позволяющим управлять размещением компонентов внутри окна.

- *В главе 6* рассматриваются основные компоненты пользовательского интерфейса, такие как надписи, кнопки, переключатели, флажки и др.
- В Qt имеется широкий выбор компонентов, позволяющих отображать как одномерный список строк (в свернутом или развернутом состояниях), так и табличные данные. Кроме того, можно отобразить данные, которые имеют очень сложную структуру, например иерархическую. Благодаря поддержке концепции «модель/представление», позволяющей отделить данные от их отображения, одни и те же данные можно отображать сразу в нескольких компонентах без их дублирования. Списки и таблицы мы рассмотрим в *главе 7*.
- В *главе 8* мы научимся работать с графикой и изображениями.
- *Глава 9* познакомит с графической сценой, позволяющей отображать объекты (например: линию, прямоугольник и др.) и производить с ними различные манипуляции (например: перемещать с помощью мыши, трансформировать и др.).
- В *главе 10* мы научимся создавать различные диалоговые окна, которые предназначены для информирования пользователя, а также для получения данных от пользователя.
- И наконец, в *главе 11* мы рассмотрим процесс создания SDI- и MDI-приложений. Научимся управлять главным меню приложения — основным компонентом пользовательского интерфейса, позволяющим компактно поместить множество команд, объединяя их в логические группы. Кроме того, мы научимся создавать панели инструментов и прикрепляемые панели.

Для полного понимания материала книги от читателя потребуется знание языка C++.

Чтобы уменьшить размер книги, основная часть кода примеров вынесена в отдельные проекты, которые расположены в электронном архиве. Электронный архив можно загрузить с сервера издательства «БХВ-Петербург» по ссылке **https://zip.bhv.ru/9785977511803.zip** или со страницы книги на сайте **www.bhv.ru** *(см. приложение)*.

Желаю приятного изучения и надеюсь, что книга поможет вам реализовать как самые простые, так и самые сложные приложения.

ГЛАВА 1

Первые шаги

Прежде всего необходимо сделать два замечания:

1. Имя пользователя компьютера должно состоять только из латинских букв. Никаких русских букв и пробелов, т. к. многие программы сохраняют различные настройки и временные файлы в каталоге C:\Users\<Имя пользователя>. Если имя пользователя содержит русские буквы, то они могут быть искажены до неузнаваемости из-за неправильного преобразования кодировок и программа не сможет сохранить настройки. Помните, что в разных кодировках русские буквы могут иметь разный код. Разработчики программ в основном работают с английским языком и ничего не знают о проблемах с кодировками, т. к. во всех однобайтовых кодировках и в кодировке UTF-8 коды латинских букв одинаковы. Так что, если хотите без проблем заниматься программированием, от использования русских букв в имени пользователя лучше отказаться.
2. Имена каталогов и файлов в пути не должны содержать русских букв и пробелов. Допустимы только латинские буквы, цифры, тире, подчеркивание и некоторые другие символы. С русскими буквами та же проблема, что описана в предыдущем пункте. При наличии пробелов в пути обычно требуется дополнительно заключать путь в кавычки. Если этого не сделать, то путь будет обрезан до первого встретившегося пробела. Такая проблема будет возникать при сборке и компиляции программ из командной строки.

Соблюдение этих двух простых правил позволит избежать множества проблем в дальнейшем при сборке и компиляции программ сторонних разработчиков.

> ***Примечание***
>
> Листинги для всех глав книги доступны отдельно. Во-первых, большие листинги в книге не смотрятся, а во-вторых, это позволило уменьшить объем всей книги, т. к. объем листингов превышает объем книги. Информация о способе получения дополнительных листингов доступна на обложке книги и в *приложении*.

1.1. Создание структуры каталогов

Перед установкой программ создадим следующую структуру каталогов:

```
book
cpp
    projectsQt
    lib
```

Каталоги book и cpp лучше разместить в корне какого-либо диска. В моем случае это будет диск C:, следовательно, пути к содержимому каталогов — C:\book и C:\cpp. Можно создать каталоги и в любом другом месте, но в пути не должно быть русских букв и пробелов — только латинские буквы, цифры, тире и подчеркивание. Остальных символов лучше избегать, если не хотите проблем с компиляцией и запуском программ.

В каталоге C:\book мы станем размещать наши тестовые программы. Внутри каталога C:\cpp у нас созданы два вложенных каталога:

- projectsQt — в этом каталоге мы станем сохранять проекты из редактора Qt Creator;
- lib — путь к этому каталогу мы добавим в системную переменную `PATH` и будем размещать в нем различные библиотеки, которые потребуются для наших программ.

1.2. Добавление пути в переменную *PATH*

Когда мы в командной строке вводим название программы без предварительного указания пути к ней, то сначала поиск программы выполняется в текущем рабочем каталоге (обычно это каталог, из которого запускается программа), а затем в путях, указанных в системной переменной `PATH`. Аналогично выполняется поиск библиотек динамической компоновки при запуске программы с помощью двойного щелчка на значке файла, но системные каталоги имеют более высокий приоритет, чем каталоги, указанные в переменной `PATH`. Пути в системной переменной `PATH` просматриваются слева направо до первого нахождения искомого объекта. Так что, если в путях расположено несколько объектов с одинаковыми именами, мы получим только первый найденный объект. Поэтому если вдруг запустилась другая программа, то следует либо удалить путь, ведущий к другой программе, либо переместить новый путь в самое начало системной переменной `PATH`.

Давайте добавим путь к каталогу C:\cpp\lib в переменную `PATH`. Чтобы изменить системную переменную в Windows, переходим в **Параметры | Панель управления | Система и безопасность | Система | Дополнительные параметры системы**. В результате откроется окно **Свойства системы** (рис. 1.1). На вкладке **Дополнительно** нажимаем кнопку **Переменные среды**. В открывшемся окне (рис. 1.2) в списке **Системные переменные** выделяем строку с переменной `Path` и нажимаем кнопку **Изменить**. Добавляем путь к каталогу C:\cpp\lib и сохраняем изменения.

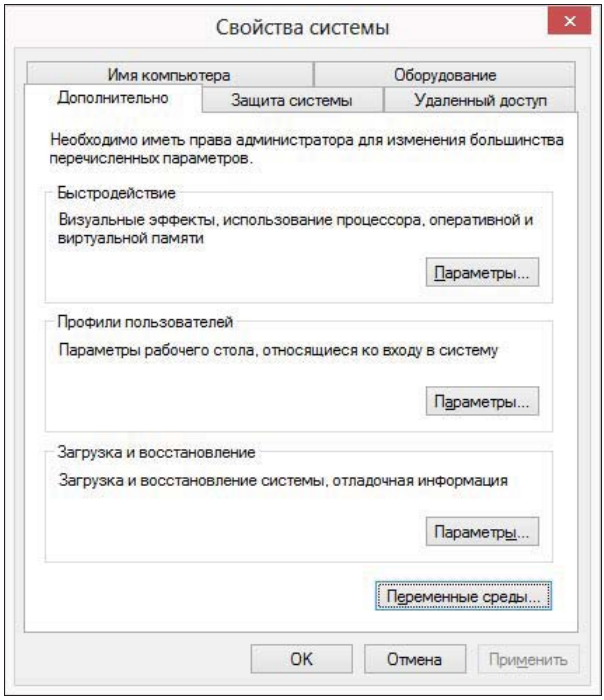

Рис. 1.1. Окно **Свойства системы**

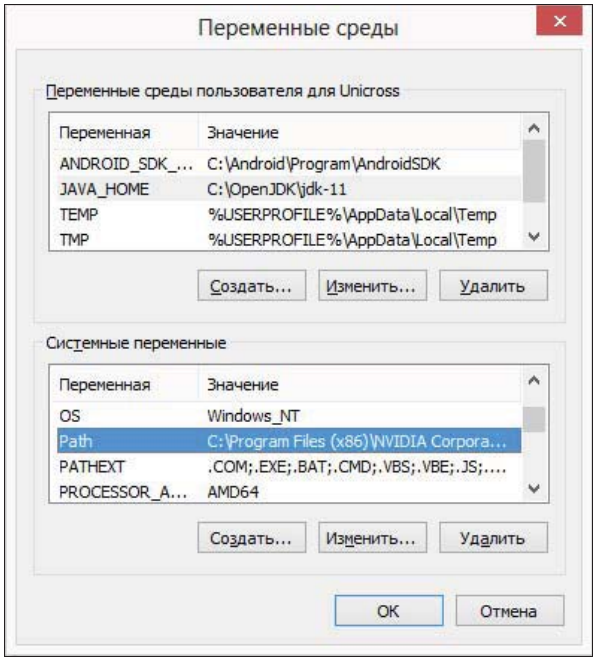

Рис. 1.2. Окно **Переменные среды**

Добавлять пути в переменную PATH мы будем несколько раз, поэтому способ изменения значения этой системной переменной нужно знать наизусть.

> ***Внимание!***
> Случайно не удалите существующее значение переменной PATH, иначе другие приложения перестанут запускаться.

1.3. Работа с командной строкой

При изучении материала мы часто будем пользоваться приложением Командная строка. Вполне возможно, что вы никогда не пользовались командной строкой и не знаете, как запустить это приложение. Давайте рассмотрим некоторые способы его запуска в Windows.

- Через поиск находим приложение Командная строка.
- Нажимаем комбинацию клавиш <Windows>+<R>. В открывшемся окне вводим cmd и нажимаем кнопку **OK**.
- Находим файл cmd.exe в каталоге C:\Windows\System32.
- В Проводнике щелкаем правой кнопкой мыши на свободном месте списка файлов, удерживая при этом нажатой клавишу <Shift>, и из контекстного меню выбираем пункт **Открыть окно команд**.
- В Проводнике в адресной строке вводим cmd и нажимаем клавишу <Enter>.

В некоторых случаях для выполнения различных команд могут потребоваться права администратора. Чтобы запустить командную строку с правами администратора, через поиск находим приложение Командная строка, щелкаем на значке правой кнопкой мыши и затем выбираем пункт **Запуск от имени администратора**.

Заучите способы запуска командной строки наизусть. В дальнейшем мы просто будем говорить «запустите командную строку» без уточнения, как это сделать.

1.4. Установка Qt 6

Для загрузки библиотеки Qt переходим на сайт **https://www.qt.io/** и нажимаем кнопку **Download**. Далее нажимаем кнопку **Go open source** из раздела **Downloads for open source users**. На следующей странице нажимаем кнопку **Download the Qt Online Installer**, а затем кнопку **Download**. Скачиваем файл qt-unified-windows-x86-4.1.0-online.exe и запускаем программу установки. Обратите внимание: для установки библиотеки потребуется активное подключение к сети Интернет.

> ***Системные требования***
> Для корректной работы Qt 6 в Windows требуется операционная система Windows 10 64-bit. Полный список поддерживаемых платформ можно посмотреть на странице **https://doc.qt.io/qt-6/supported-platforms.html**.

После запуска программы установки отобразится окно **Qt Setup** (Установка Qt), показанное на рис. 1.3. Вводим регистрационные данные или создаем новый аккаунт. Нажимаем кнопку **Next** (Далее). Если создавался новый аккаунт, то на адрес электронной почты придет письмо со ссылкой, позволяющей подтвердить адрес. Переходим по ссылке, а затем возвращаемся к программе установки и нажимаем кнопку **Next**.

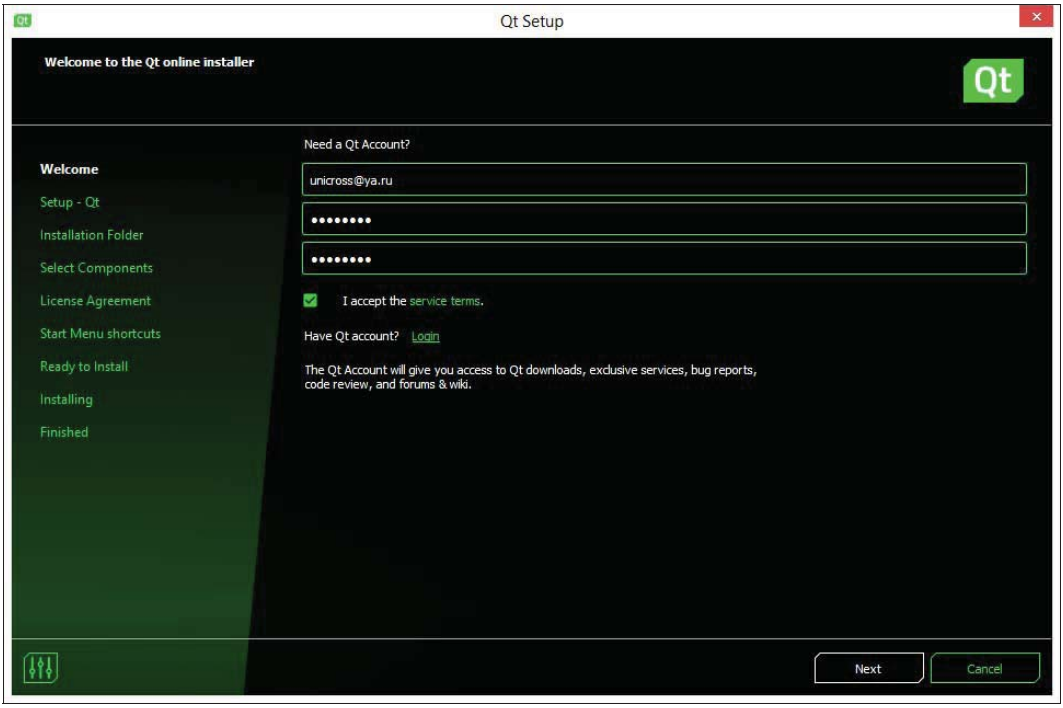

Рис. 1.3. Установка Qt. Шаг 1

На следующем шаге (рис. 1.4) устанавливаем флажки и нажимаем кнопку **Next**. В следующем окне (рис. 1.5) нажимаем кнопку **Next**. Далее нужно выбрать один из переключателей (рис. 1.6). Выбираем и нажимаем кнопку **Next**. На следующем шаге (рис. 1.7) должен быть указан путь C:\Qt и установлен флажок **Custom installation**. Нажимаем кнопку **Next**.

В следующем окне (рис. 1.8) устанавливаем флажок **Latest releases** и нажимаем кнопку **Filter** (Отфильтровать). Затем в разделе **Qt** раскрываем список **Qt 6.1.0** и устанавливаем флажок **MinGW 8.1.0 64-bit**. В разделе **Developer and Designer Tools** (рис. 1.9) устанавливаем флажки: **Qt Creator 4.15.0 CDB Debugger Support**, **Debugging Tools for Windows** и **MinGW 8.1.0 64-bit**. Нажимаем кнопку **Next**. В открывшемся окне (рис. 1.10) принимаем лицензионное соглашение и нажимаем кнопку **Next**. Затем (рис. 1.11) опять нажимаем кнопку **Next**. Нажимаем кнопку **Install** (Установить) для запуска установки (рис. 1.12). После завершения установки нажимаем кнопку **Finish** (Завершить) (рис. 1.13).

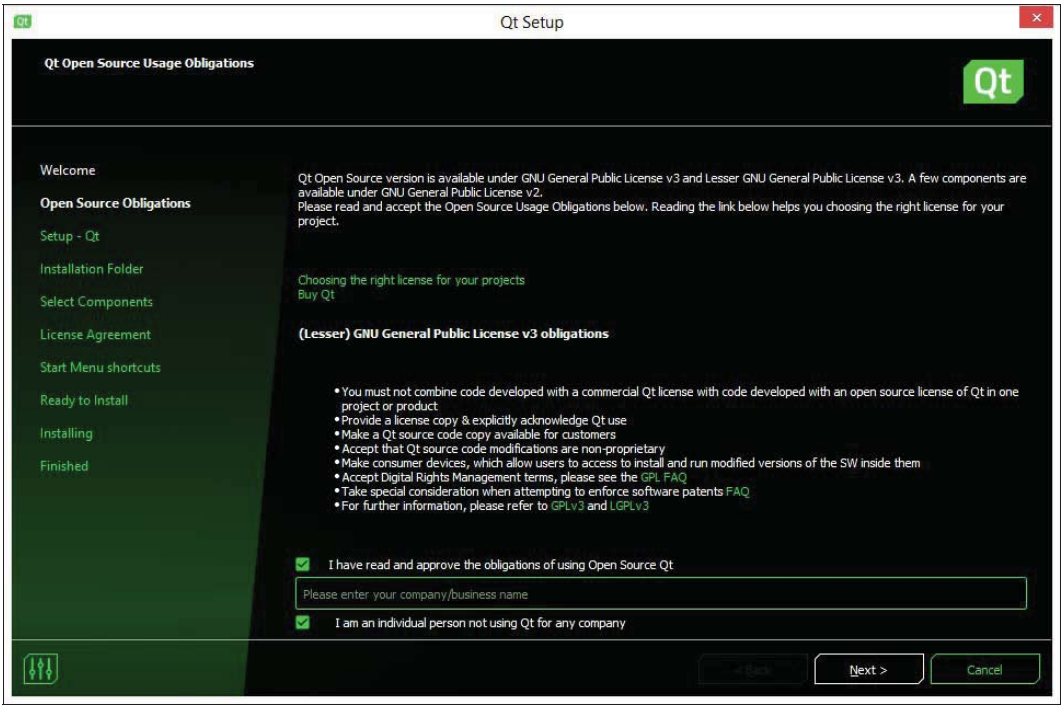

Рис. 1.4. Установка Qt. Шаг 2

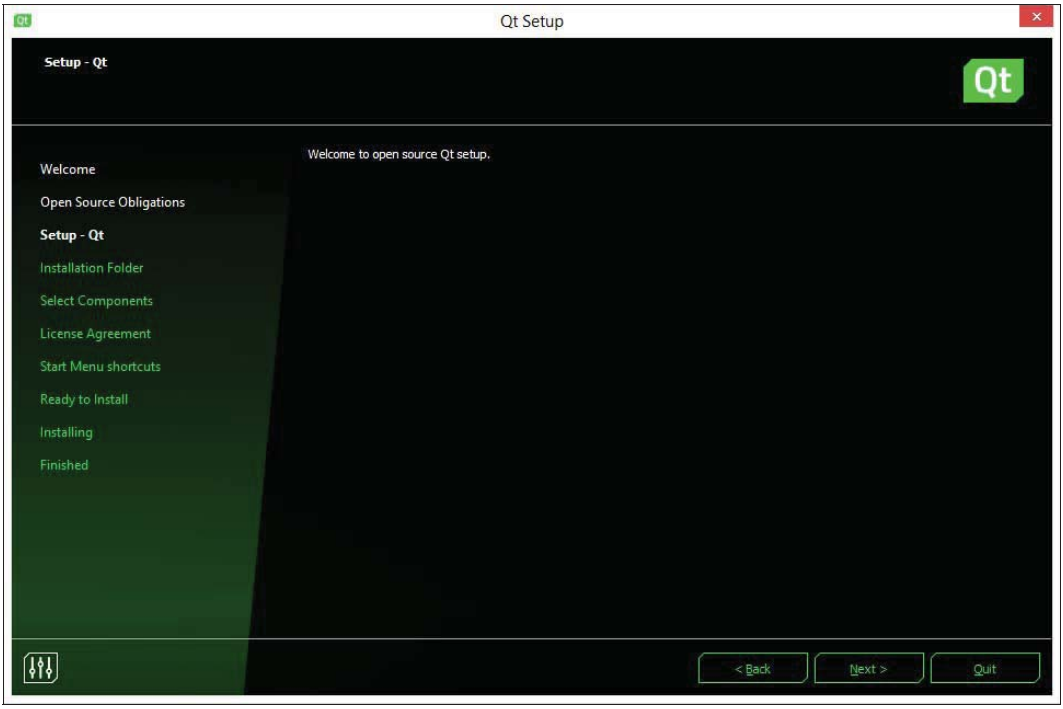

Рис. 1.5. Установка Qt. Шаг 3

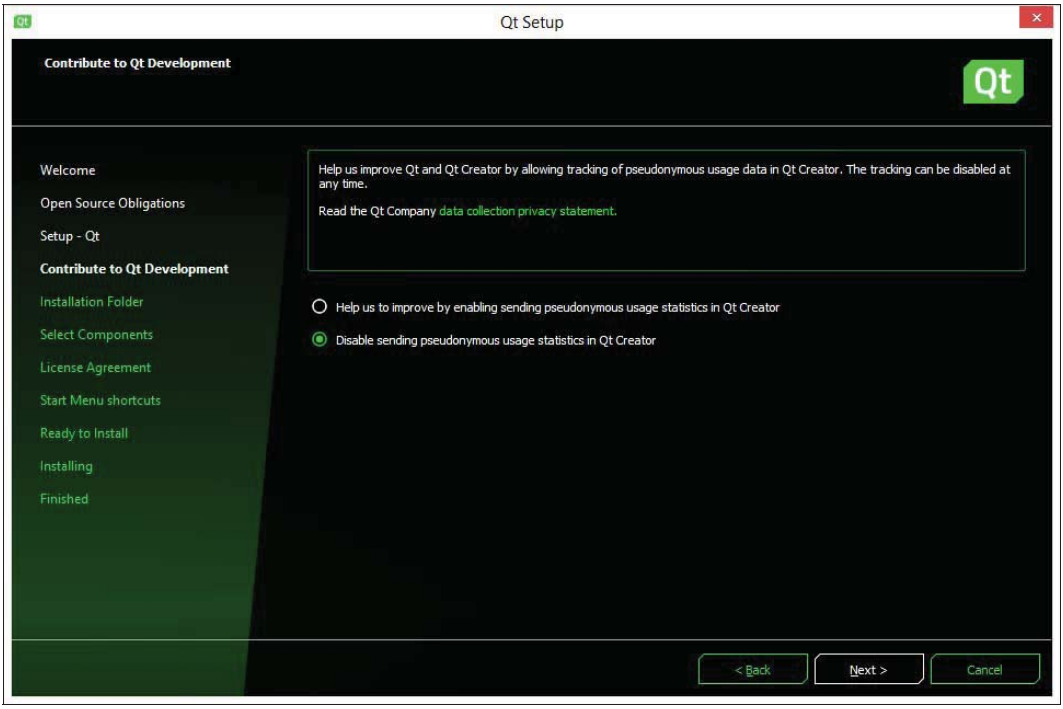

Рис. 1.6. Установка Qt. Шаг 4

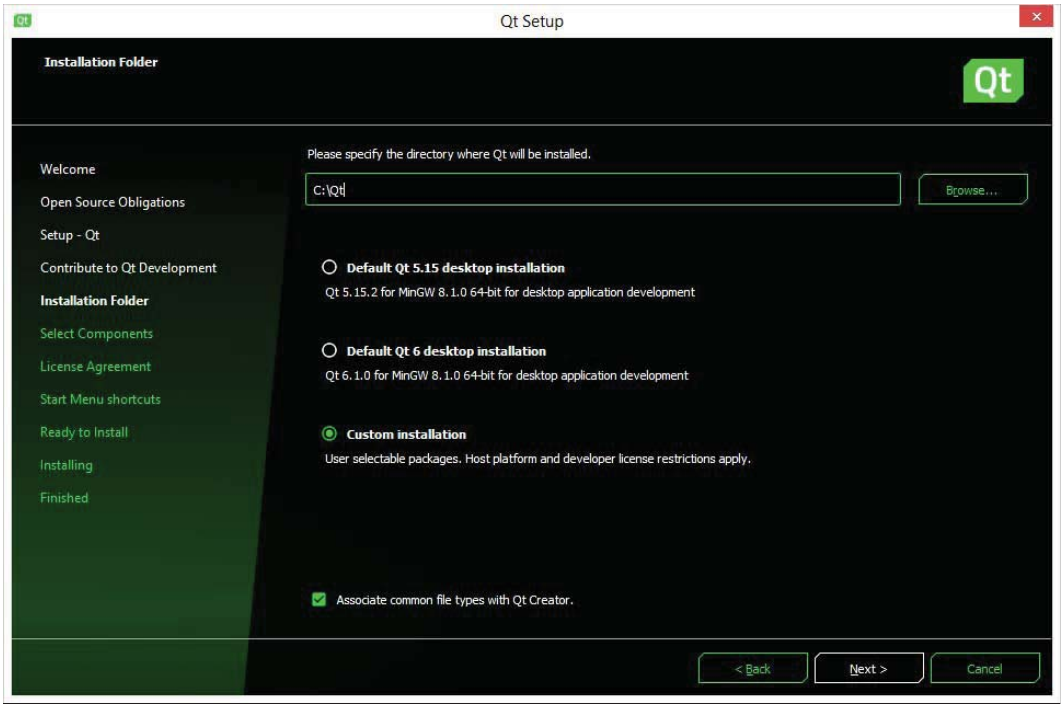

Рис. 1.7. Установка Qt. Шаг 5

Рис. 1.8. Установка Qt. Шаг 6. Раздел **Qt**

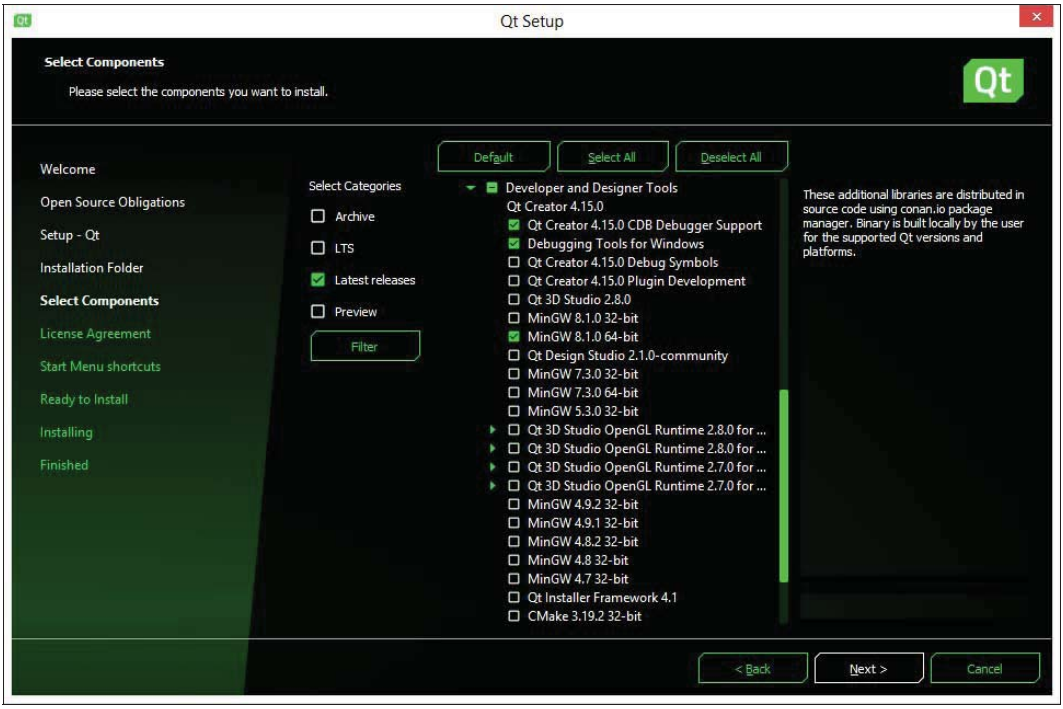

Рис. 1.9. Установка Qt. Шаг 6. Раздел **Developer and Designer Tools**

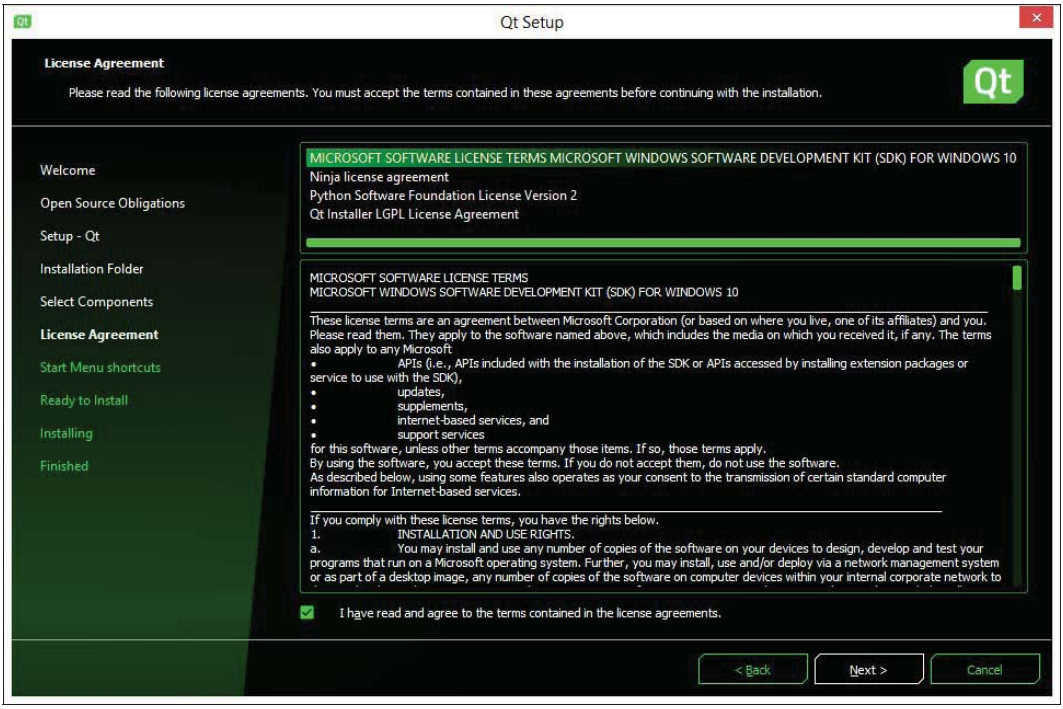

Рис. 1.10. Установка Qt. Шаг 7

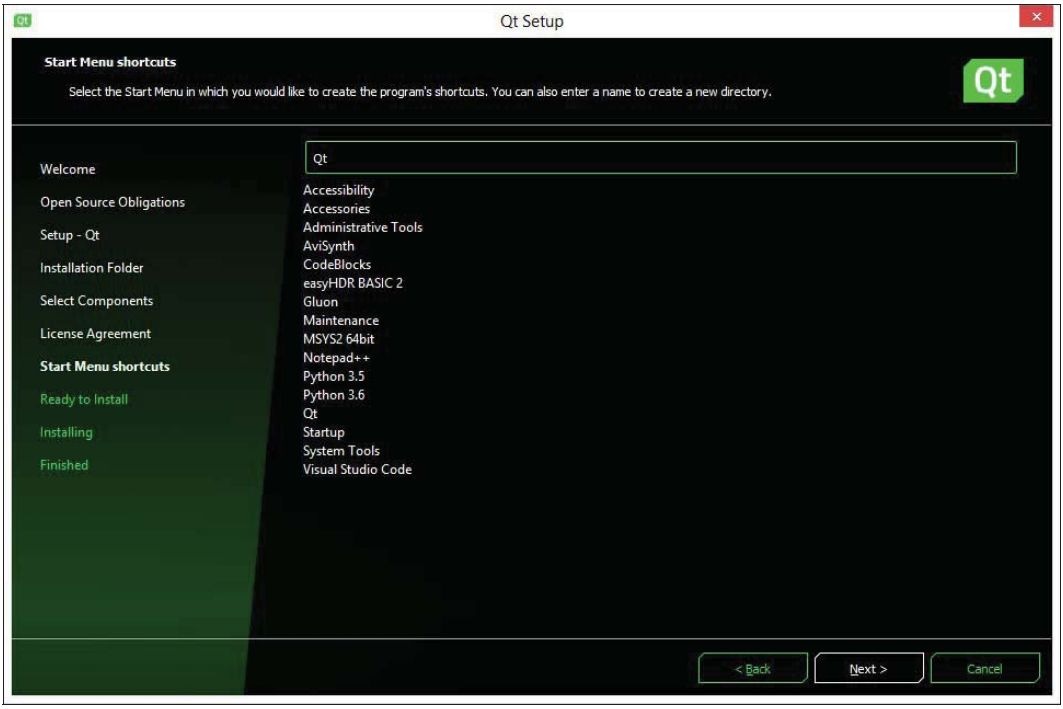

Рис. 1.11. Установка Qt. Шаг 8

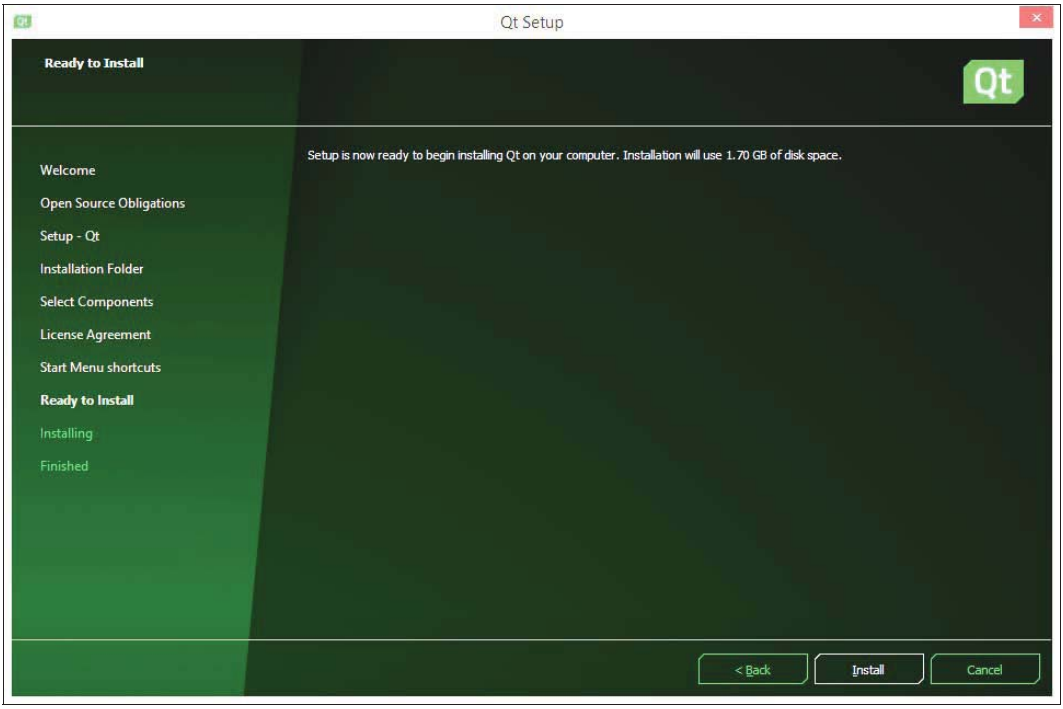

Рис. 1.12. Установка Qt. Шаг 9

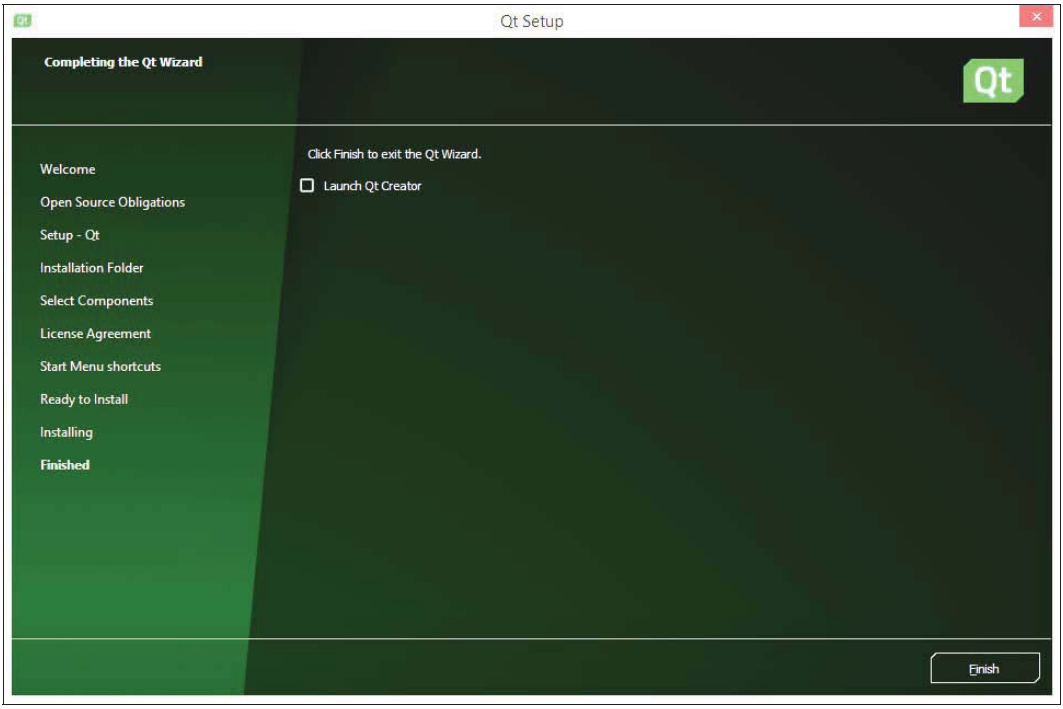

Рис. 1.13. Установка Qt. Шаг 10

В результате библиотека Qt будет установлена в каталог C:\Qt\6.1.0\mingw81_64. В каталоге C:\Qt\6.1.0\mingw81_64\bin можно найти библиотеки, необходимые для работы оконных приложений. Библиотека MinGW будет установлена в каталог C:\Qt\Tools\mingw810_64. Программу g++.exe, предназначенную для компиляции программ, написанных на языке C++, можно найти в каталоге C:\Qt\Tools\mingw810_64\bin.

В каталог C:\Qt\Tools\QtCreator\bin был установлен редактор Qt Creator, который мы будем использовать для набора кода и сборки проектов. Запустить редактор можно с помощью файла qtcreator.exe. Если на последнем шаге мастера был установлен флажок **Launch Qt Creator** (см. рис. 1.13), то редактор запустится автоматически.

Пути к каталогам C:\Qt\Tools\mingw810_64\bin и C:\Qt\6.1.0\mingw81_64\bin можно добавить в системную переменную `PATH`. Однако мы этого делать не станем, чтобы можно было использовать сразу несколько компиляторов. Вместо изменения переменной `PATH` на постоянной основе мы будем выполнять изменение в командной строке только для текущего сеанса. Продемонстрируем это на примере, а заодно проверим работоспособность компилятора. Запускаем командную строку и выполняем следующие команды:

```
C:\Users\Unicross>cd C:\

C:\>set Path=C:\Qt\Tools\mingw810_64\bin;C:\Qt\6.1.0\mingw81_64\bin;%Path%

C:\>gcc --version
gcc (x86_64-posix-seh-rev0, Built by MinGW-W64 project) 8.1.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.

C:\>g++ --version
g++ (x86_64-posix-seh-rev0, Built by MinGW-W64 project) 8.1.0
Copyright (C) 2018 Free Software Foundation, Inc.
This is free software; see the source for copying conditions.
There is NO warranty; not even for MERCHANTABILITY or FITNESS FOR A PARTICULAR
PURPOSE.
```

Первая команда (`cd C:\`) делает текущим корневой каталог диска C:. Вторая команда (`set Path=C:\Qt\Tools\mingw810_64\bin;C:\Qt\6.1.0\mingw81_64\bin;%Path%`) изменяет значение системной переменной `PATH` для текущего сеанса. Пути к каталогам C:\Qt\Tools\mingw810_64\bin и C:\Qt\6.1.0\mingw81_64\bin мы добавили в самое начало переменной `PATH`. Третья команда (`gcc --version`) выводит версию программы gcc.exe. Эту программу можно использовать для компиляции программ, написанных на языке C. Четвертая команда (`g++ --version`) выводит версию программы g++.exe. Эту программу мы будем использовать для компиляции программ, написанных на языке C++. Фрагменты перед командами означают приглашение для ввода команд. Текст после команд является результатом выполнения этих команд.

Вместо выполнения отдельных команд можно написать скрипт, который выполняет сразу несколько команд и отображает результат их выполнения в отдельном окне. Запускаться такой скрипт будет с помощью двойного щелчка левой кнопкой мыши на значке файла. Для создания файла потребуется текстовый редактор, позволяющий корректно работать с различными кодировками. Советую установить на компьютер редактор Notepad++. Скачать редактор можно абсолютно бесплатно со страницы **https://notepad-plus-plus.org/**. Из двух вариантов (архив и инсталлятор) советую выбрать именно инсталлятор, т. к. при установке можно будет указать язык интерфейса программы. Установка Notepad++ предельно проста и в комментариях не нуждается.

Запускаем Notepad++ и создаем новый документ. Консоль в Windows по умолчанию работает с кодировкой windows-866, поэтому мы должны и файл сохранить в этой кодировке, иначе русские буквы будут искажены. В меню **Кодировки** выбираем пункт **Кодировки | Кириллица | OEM 866**. Вводим текст скрипта (листинг 1.1) и сохраняем под названием script.bat в каталоге C:\book. Запускаем скрипт с помощью двойного щелчка левой кнопкой мыши на значке файла script.bat. Результат выполнения показан на рис. 1.14. Для закрытия окна консоли достаточно нажать любую клавишу.

Листинг 1.1. Содержимое файла script.bat

```
@echo off
title Заголовок окна
cd C:\
echo Текущий каталог: %CD%
@echo.
set Path=C:\Qt\Tools\mingw810_64\bin;%Path%
rem Вывод версии gcc.exe
echo gcc --version
@echo.
gcc --version
rem Вывод версии g++.exe
echo g++ --version
@echo.
g++ --version
pause
```

Рассмотрим инструкции из этого скрипта:

- `title` — позволяет вывести текст в заголовок окна консоли;
- `echo` — выводит текст в окно консоли;
- `@echo.` — выводит пустую строку в окно консоли;
- `%CD%` — переменная, содержащая путь к текущему рабочему каталогу;
- `%Path%` — переменная, содержащая значение системной переменной `PATH`;
- `rem` — вставляет комментарий, поясняющий фрагмент кода;

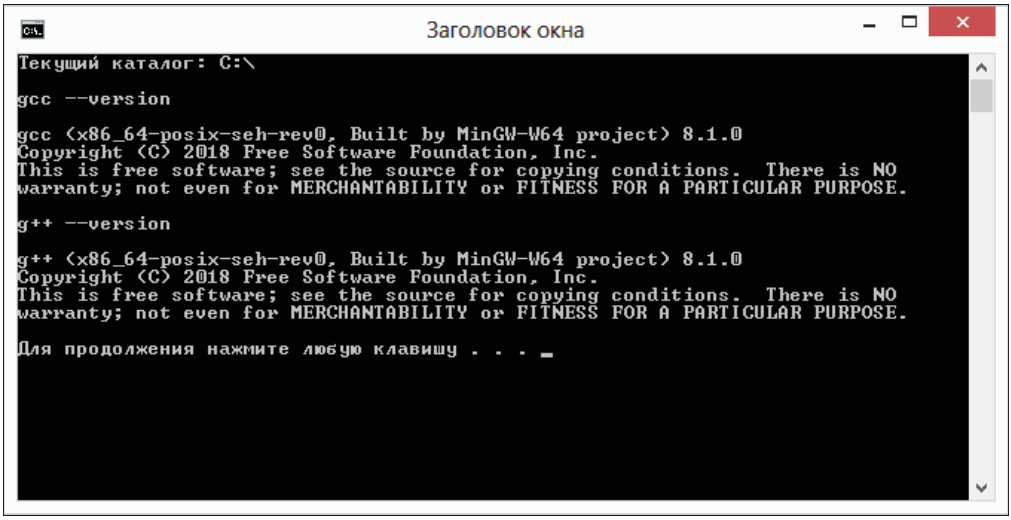

Рис. 1.14. Результат выполнения листинга 1.1

- `pause` — ожидает ввода любого символа от пользователя. Если эту инструкцию убрать из скрипта, то программа выполнится и окно консоли сразу закроется, не дав нам возможности увидеть результат.

1.5. Настройка редактора Qt Creator

В этой книге для создания и компиляции программ на языке C++ мы воспользуемся редактором Qt Creator. Редактор удобен в использовании, он позволяет автоматически закончить слово, подчеркнуть код с ошибкой, подсветить код программы, вывести список всех функций, отладить программу, а также скомпилировать все файлы проекта всего лишь нажатием одной кнопки без необходимости использования командной строки.

Для запуска редактора Qt Creator переходим в каталог C:\Qt\Tools\QtCreator\bin и щелкаем левой кнопкой мыши на значке файла qtcreator.exe. Если на последнем шаге мастера был установлен флажок (см. рис. 1.13), то редактор запустится автоматически. После запуска отобразится окно, показанное на рис. 1.15.

> ***Примечание***
>
> Если Qt Creator не запустился или работает с ошибками, то можно загрузить предыдущую версию редактора со страницы **https://github.com/qt-creator/qt-creator/releases**.

Прежде чем пользоваться редактором, нужно изменить некоторые настройки по умолчанию. Начнем с кодировки файлов, в которых мы будем вводить текст программы. Чтобы указать кодировку по умолчанию для всех файлов, в меню **Инструменты** выбираем пункт **Параметры**. В открывшемся окне переходим на вкладку **Текстовый редактор | Поведение** (рис. 1.16). В группе **Кодировки файлов** из списка **По умолчанию** выбираем пункт **UTF-8**. Сохраняем изменения. Таким образом, при создании оконных приложений мы будем пользоваться кодировкой UTF-8.

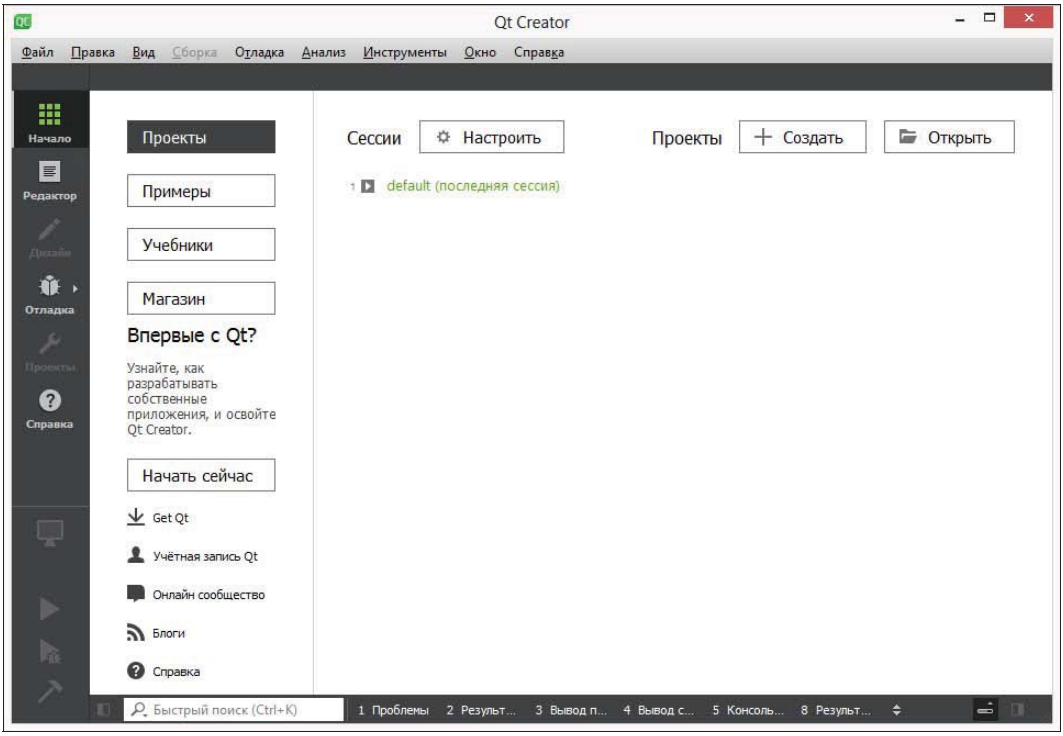

Рис. 1.15. Редактор Qt Creator

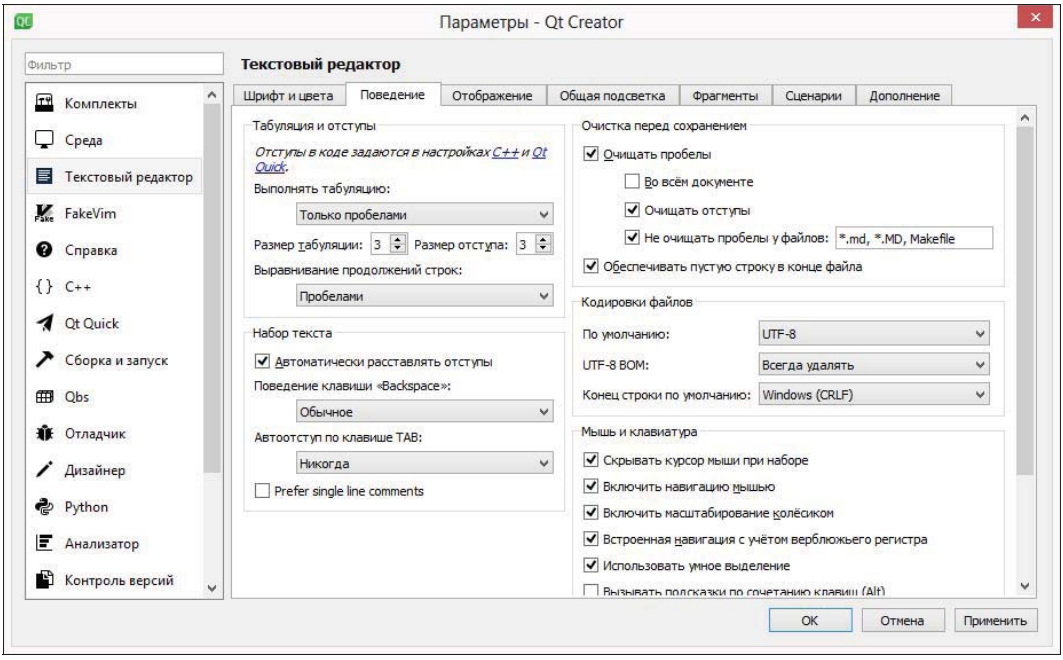

Рис. 1.16. Указание кодировки файлов по умолчанию

Если необходимо изменить кодировку уже открытого файла, в меню **Правка** выбираем пункт **Выбрать кодировку**.

> ***На заметку***
>
> Для консольных приложений кодировка файлов должна быть windows-1251.

Давайте также изменим настройку форматирования кода. Для этого в меню **Инструменты** выбираем пункт **Параметры**. В открывшемся окне переходим на вкладку **C++ | Стиль кода** (рис. 1.17). Из списка **Текущие настройки** выбираем пункт **Qt [встроенный]** и нажимаем кнопку **Копировать**. В открывшемся окне (рис. 1.18) в поле **Имя стиля кода** вводим название стиля, например `MyStyle`. Нажимаем кнопку **OK**. Далее нажимаем кнопку **Изменить**. Откроется окно (рис. 1.19), в котором можно изменить настройки нашего стиля. На вкладке **Основное** из списка **Выполнять табуляцию** выбираем пункт **Только пробелами**, а в поля **Размер табуляции** и **Размер отступа** вводим число `3`. Сохраняем все изменения.

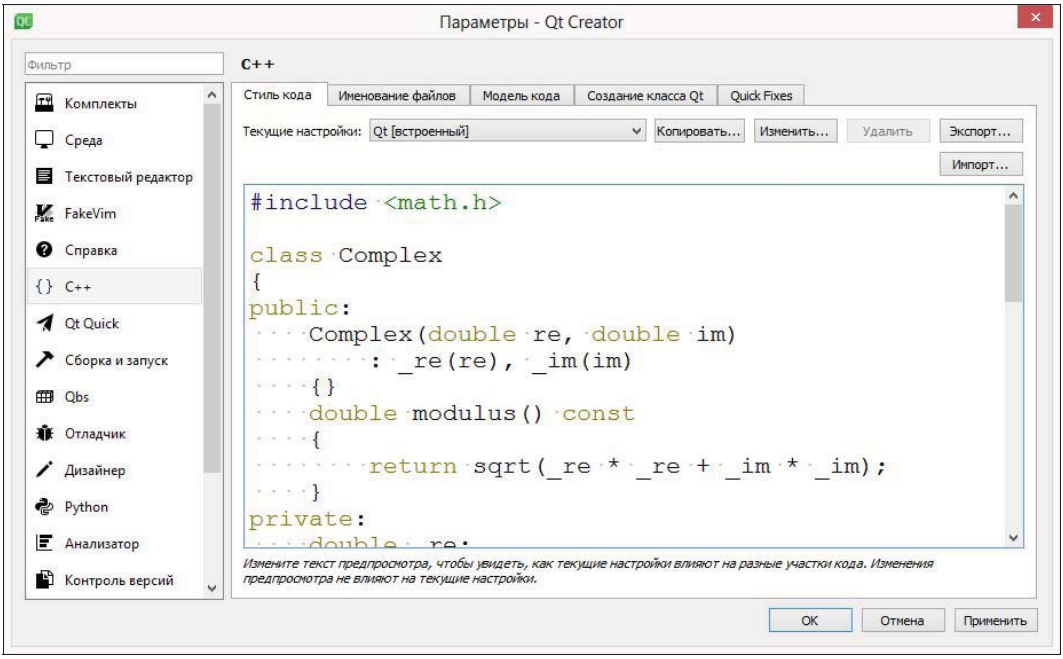

Рис. 1.17. Вкладка **C++ | Стиль кода**

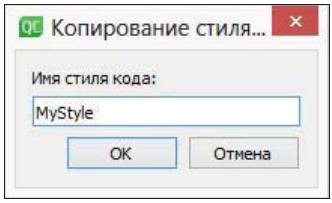

Рис. 1.18. Окно **Копирование стиля**

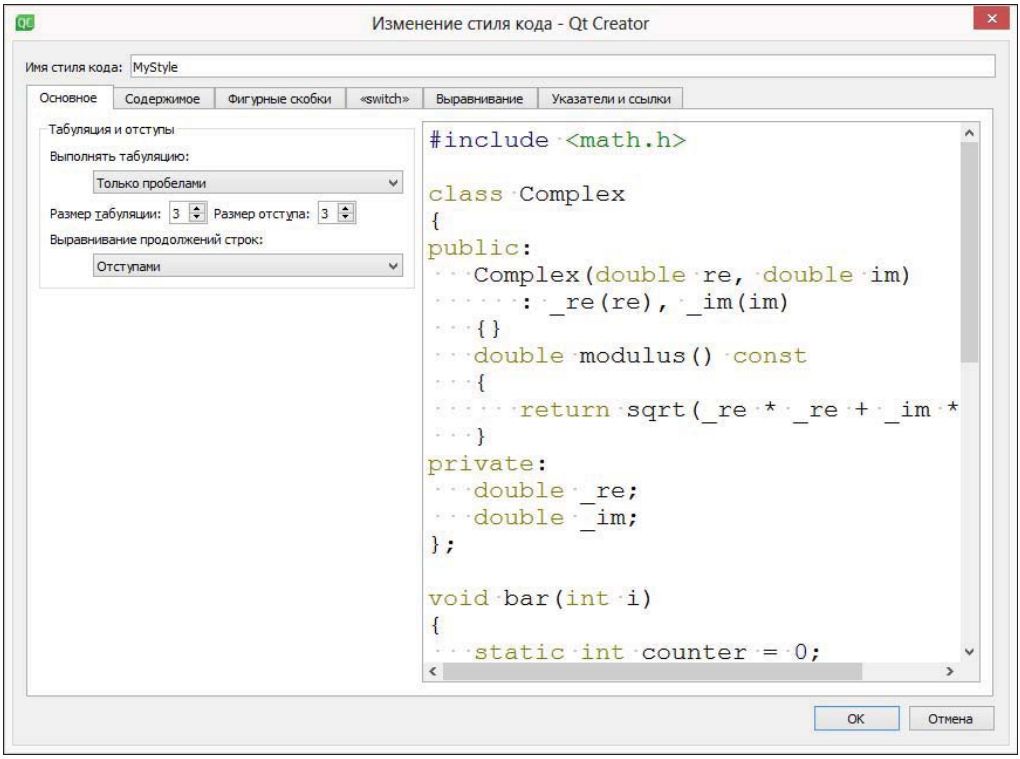

Рис. 1.19. Изменение настроек форматирования

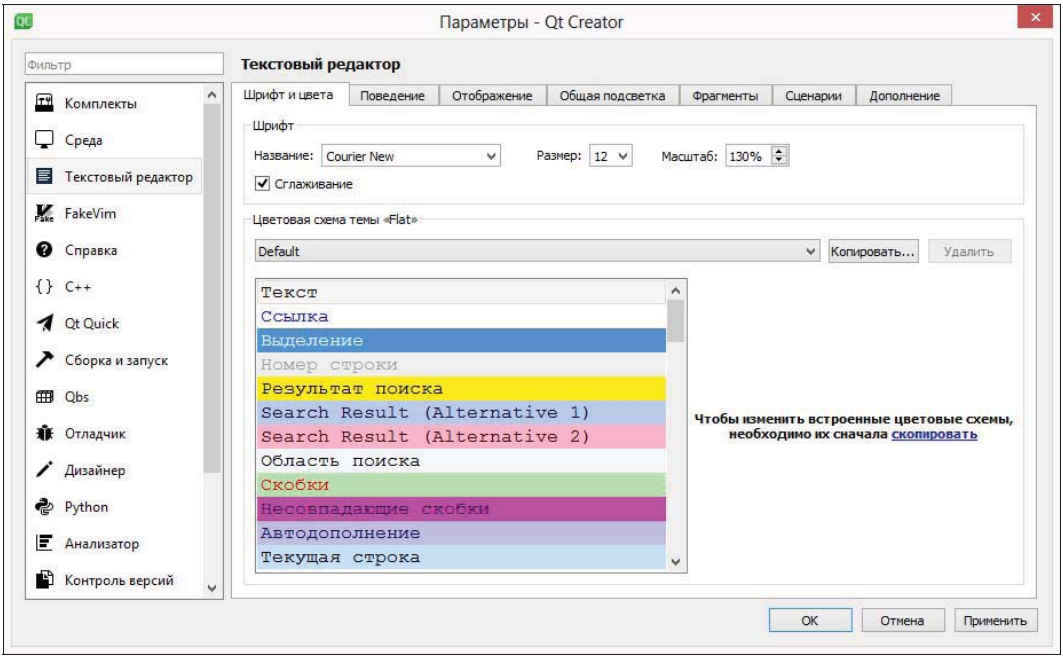

Рис. 1.20. Вкладка **Текстовый редактор | Шрифт и цвета**

Если необходимо изменить размер шрифта, то в меню **Инструменты** выбираем пункт **Параметры**. В открывшемся окне переходим на вкладку **Текстовый редактор | Шрифт и цвета** (рис. 1.20). Из списка **Название** выбираем нужный шрифт, а размер шрифта выбираем из списка **Размер**. Сохраняем изменения. Для изменения масштаба достаточно, удерживая нажатой клавишу <Ctrl>, повернуть колесико мыши.

1.6. Первая программа на Qt

Создадим простейшее оконное приложение. Для этого в меню **Файл** выбираем пункт **Создать файл или проект**. В открывшемся окне (рис. 1.21) в списке слева выбираем пункт **Приложение (Qt)**, а затем справа — пункт **Приложение Qt Widgets**. Нажимаем кнопку **Выбрать**. На следующем шаге (рис. 1.22) в поле **Название** вводим `Test`, в поле **Создать в** добавляем путь C:\cpp\projectsQt. Нажимаем кнопку **Далее**. На следующем шаге (рис. 1.23) из списка **Система сборки** выбираем пункт **qmake** и нажимаем кнопку **Далее**. На следующем шаге (рис. 1.24) задаем такие значения и нажимаем кнопку **Далее**:

- в поле **Имя класса** вводим `Widget`;
- из списка **Базовый класс** выбираем пункт **QWidget**;
- сбрасываем флажок **Создать форму**.

На следующем шаге (рис. 1.25) нажимаем кнопку **Далее**. На следующем шаге (рис. 1.26) устанавливаем флажок **Desktop Qt 6.1.0 MinGW 64-bit**. Нажимаем

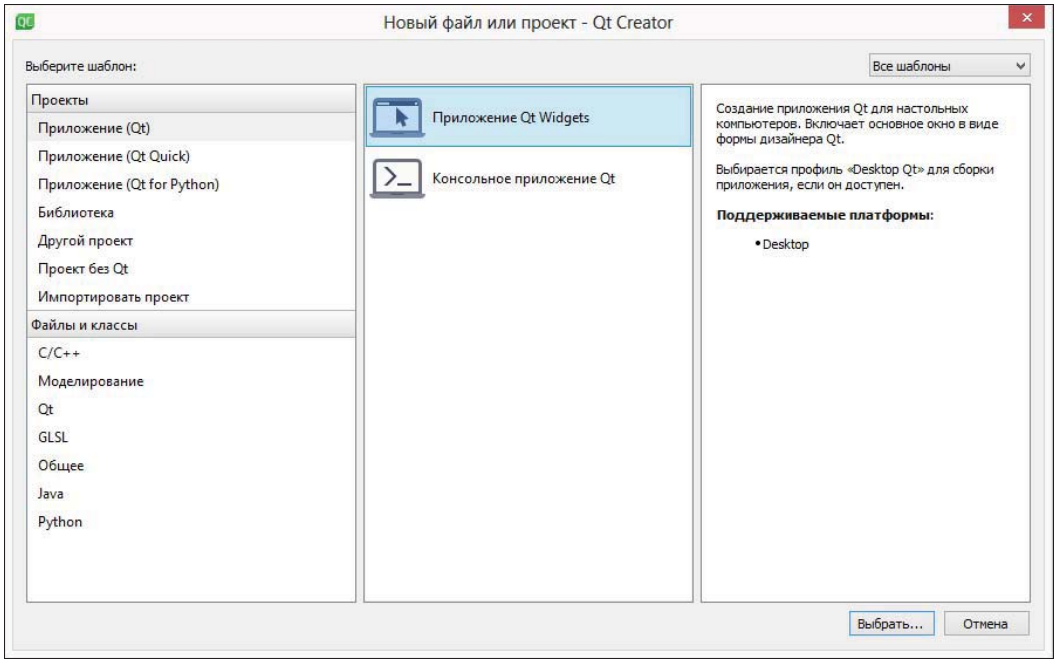

Рис. 1.21. Создание оконного приложения. Шаг 1

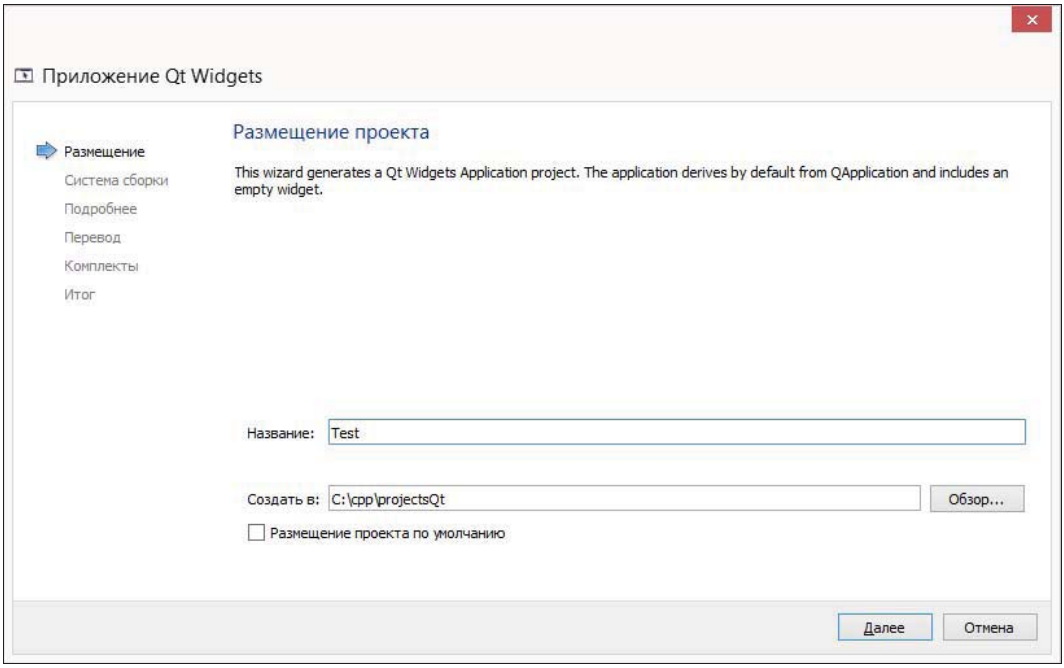

Рис. 1.22. Создание оконного приложения. Шаг 2

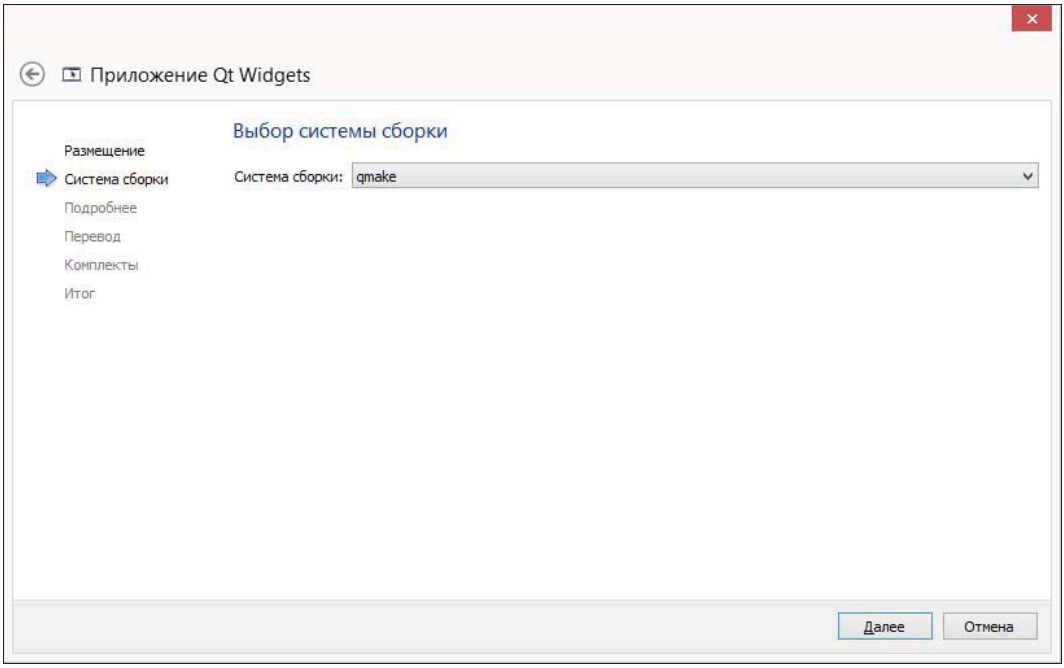

Рис. 1.23. Создание оконного приложения. Шаг 3

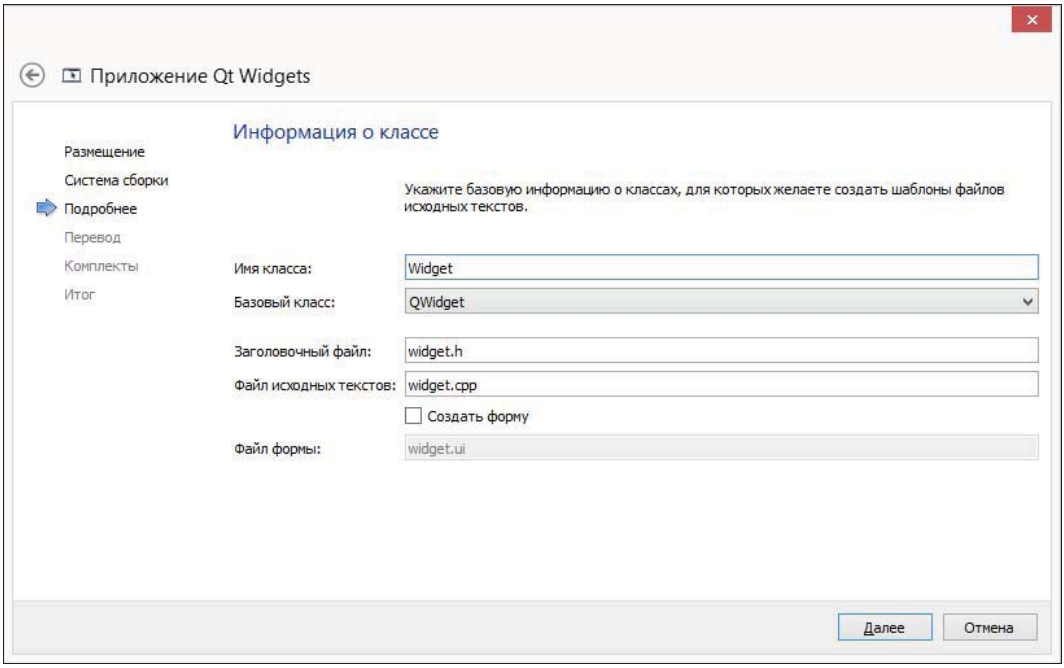

Рис. 1.24. Создание оконного приложения. Шаг 4

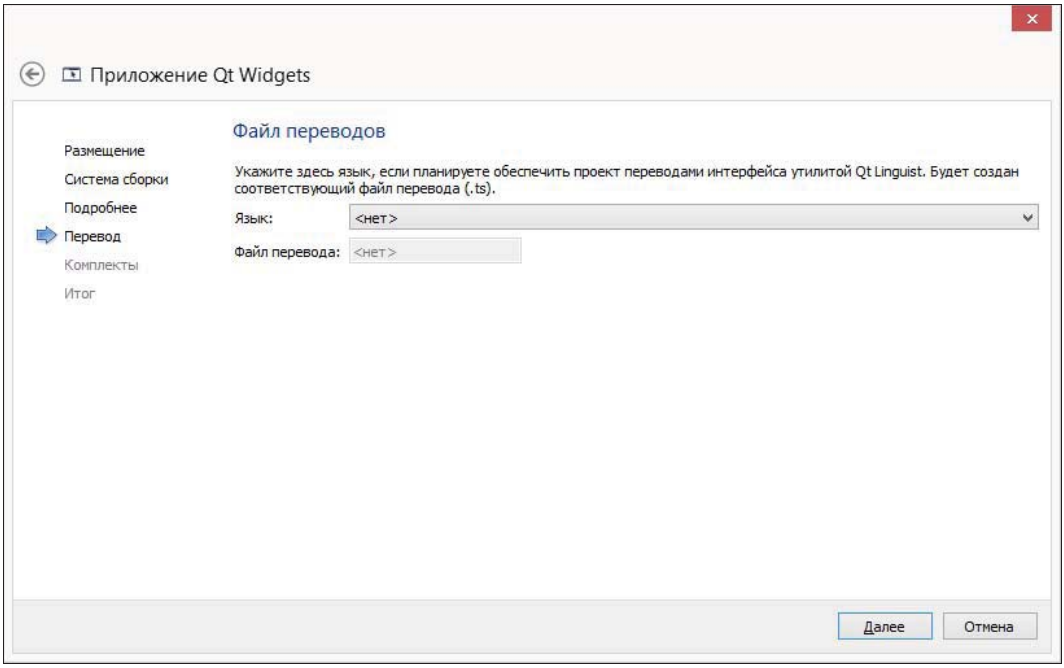

Рис. 1.25. Создание оконного приложения. Шаг 5

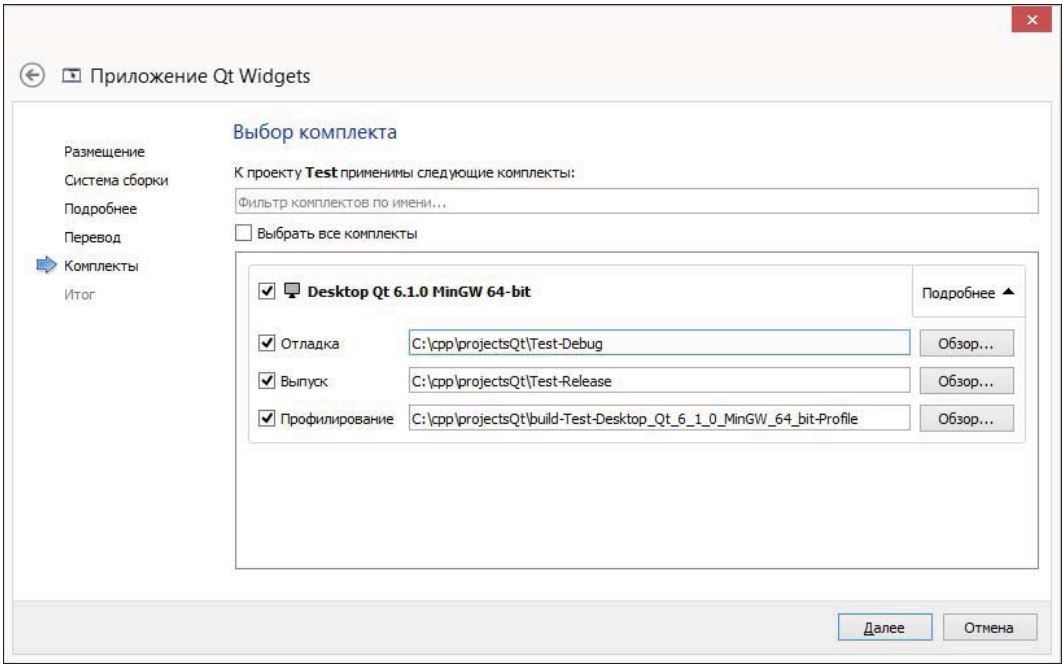

Рис. 1.26. Создание оконного приложения. Шаг 6

кнопку **Подробнее** и изменяем пути по умолчанию. В поле **Отладка** вводим значение `C:\cpp\projectsQt\Test-Debug`, а в поле **Выпуск** — значение `C:\cpp\projectsQt\Test-Release`. Нажимаем кнопку **Далее**. На следующем шаге (рис. 1.27) нажимаем кнопку **Завершить**.

В результате будет создан каталог C:\cpp\projectsQt\Test с файлами проекта. Содержимое файла Test.pro будет выглядеть так:

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

CONFIG += c++11

# You can make your code fail to compile if it uses deprecated APIs.
# In order to do so, uncomment the following line.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000    # disables all the
APIs deprecated before Qt 6.0.0

SOURCES += \
    main.cpp \
    widget.cpp

HEADERS += \
    widget.h
```

```
# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

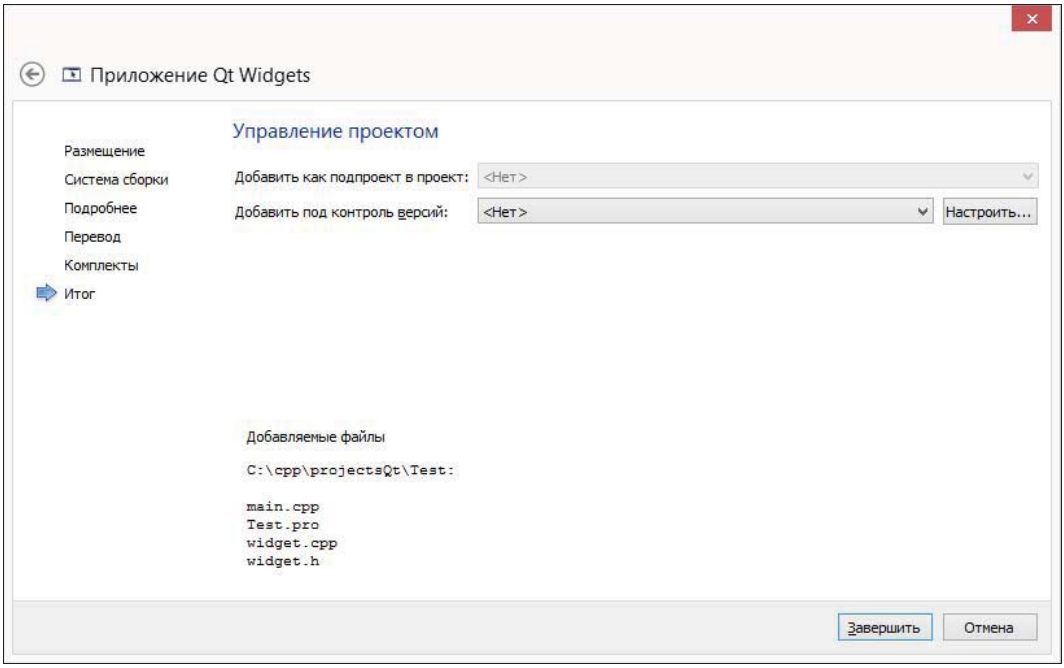

Рис. 1.27. Создание оконного приложения. Шаг 7

При изучении языков и технологий принято начинать с программы, выводящей надпись `Привет, мир!`. Не будем нарушать традицию и создадим окно с приветствием и кнопкой для закрытия окна. В файл C:\cpp\projectsQt\Test\main.cpp добавляем программу из листинга 1.2.

Листинг 1.2. Первая программа на Qt

```
#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QPushButton>
#include <QVBoxLayout>

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);

   QWidget window;
   window.setWindowTitle("Первая программа на Qt");
   window.resize(300, 70);
```

```
    QLabel *label = new QLabel("<center>Привет, мир!</center>");
    QPushButton *btnQuit = new QPushButton("&Закрыть окно");
    QVBoxLayout *vbox = new QVBoxLayout();
    vbox->addWidget(label);
    vbox->addWidget(btnQuit);
    window.setLayout(vbox);
    QObject::connect(btnQuit, SIGNAL(clicked()),
                     &app, SLOT(quit()));

    window.show();
    return app.exec();
}
```

Чтобы преобразовать текстовый файл main.cpp с программой в исполняемый EXE-файл, делаем текущей вкладку с содержимым файла main.cpp и слева сначала выбираем пункт **Отладка**, а затем нажимаем кнопку **Запустить** (содержит значок в виде зеленого треугольника) или нажимаем комбинацию клавиш <Ctrl>+<R>. Можно также в меню **Сборка** выбрать пункт **Запустить**. Если все сделано правильно, то отобразится окно, показанное на рис. 1.28.

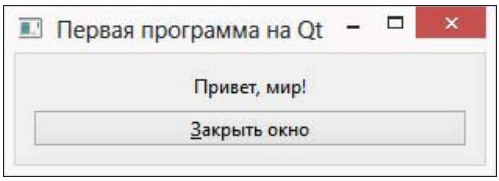

Рис. 1.28. Результат выполнения программы

1.7. Структура программы

Запускать программу мы научились, теперь рассмотрим код из листинга 1.2 построчно. В первых строках подключаются необходимые заголовочные файлы:

```
#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QPushButton>
#include <QVBoxLayout>
```

Далее определяется функция `main()`, которая является точкой входа в приложение. Через первый параметр (`argc`) доступно количество аргументов, переданных в командной строке. Следует учитывать, что первым аргументом является название исполняемого файла, поэтому значение параметра `argc` не может быть меньше единицы. Через второй параметр (`argv`) доступны все аргументы в виде строки (тип `char *`). Квадратные скобки после названия второго параметра означают, что доступен массив строк.

Инструкция

```
QApplication app(argc, argv);
```

создает объект приложения с помощью класса `QApplication`. Конструктор этого класса принимает количество и список параметров, переданных в командной строке. Следует помнить, что в программе всегда должен быть объект приложения, причем обязательно только один. Может показаться, что после создания он больше нигде не используется в программе, однако с помощью этого объекта осуществляется управление приложением незаметно для нас. Получить доступ к этому объекту из любого места в программе можно через макрос `qApp`, который преобразуется в указатель на объект приложения. Например, вывести список параметров, переданных в командной строке, можно так:

```
qDebug() << qApp->arguments();
```

Результат при запуске программы:

```
QList("C:\\cpp\\projectsQt\\Test-Debug\\debug\\Test.exe")
```

Следующая инструкция:

```
QWidget window;
```

создает объект окна с помощью класса `QWidget`. Этот класс наследуют практически все классы, реализующие компоненты графического интерфейса. Поэтому любой компонент, не имеющий родителя, обладает своим собственным окном.

Инструкция

```
window.setWindowTitle("Первая программа на Qt");
```

задает текст, который будет выводиться в заголовке окна. Следующая инструкция:

```
window.resize(300, 70);
```

задает предпочтительные размеры окна. В первом параметре метода `resize()` указывается ширина окна, а во втором параметре — высота окна. Следует учитывать, что эти размеры не включают высоту заголовка окна и ширину границ, а также являются рекомендацией, т. е. если компоненты не помещаются, размеры окна будут увеличены.

Инструкция

```
QLabel *label = new QLabel("<center>Привет, мир!</center>");
```

создает объект надписи. Текст надписи задается в качестве параметра в конструкторе класса `QLabel`. Обратите внимание на то, что внутри строки мы указали HTML-теги. В данном примере с помощью тега `<center>` произвели выравнивание текста по центру компонента. Возможность использования HTML-тегов и CSS-атрибутов является отличительной чертой библиотеки Qt. Например, внутри надписи можно вывести таблицу или изображение. Это очень удобно.

Следующая инструкция:

```
QPushButton *btnQuit = new QPushButton("&Закрыть окно");
```

создает объект кнопки. Текст, который будет отображен на кнопке, задается в качестве параметра в конструкторе класса `QPushButton`. Обратите внимание на символ `&` перед буквой `З`. Таким образом задаются клавиши быстрого доступа. Если нажать одновременно клавишу <Alt> и клавишу с буквой, перед которой в строке указан символ `&`, то кнопка будет нажата.

Инструкция

```
QVBoxLayout *vbox = new QVBoxLayout();
```

создает вертикальный контейнер. Все компоненты, добавляемые в этот контейнер, будут располагаться друг под другом в порядке добавления. Внутри контейнера автоматически производится подгонка размеров добавляемых компонентов под размеры контейнера. При изменении размеров контейнера будет произведено изменение размеров всех компонентов. В следующих двух инструкциях:

```
vbox->addWidget(label);
vbox->addWidget(btnQuit);
```

с помощью метода `addWidget()` производится добавление объектов надписи и кнопки в вертикальный контейнер. Так как объект надписи добавляется первым, он будет расположен над кнопкой. При добавлении в контейнер компоненты автоматически становятся потомками контейнера.

Следующая инструкция:

```
window.setLayout(vbox);
```

добавляет контейнер в основное окно с помощью метода `setLayout()`. Таким образом, контейнер становится потомком основного окна.

Инструкция

```
QObject::connect(btnQuit, SIGNAL(clicked()),
                 &app, SLOT(quit()));
```

назначает обработчик сигнала `clicked()`, который генерируется при нажатии кнопки. В первом параметре статического метода `connect()` передается указатель на объект, генерирующий сигнал, а во втором параметре, с помощью макроса `SIGNAL()`, — название сигнала. В третьем параметре передается указатель на объект, принимающий сигнал, а в четвертом параметре — метод этого объекта, который будет вызван при наступлении события. Метод указывается в качестве параметра макроса `SLOT()`. Этот метод принято называть *слотом*. В нашем примере получателем сигнала является объект приложения, доступный также через указатель `qApp`. При наступлении события будет вызван метод `quit()`, который завершит работу всего приложения. Пример указания макроса `qApp`:

```
QObject::connect(btnQuit, SIGNAL(clicked()),
                 qApp, SLOT(quit()));
```

Следующая инструкция:

```
window.show();
```

отображает окно и все компоненты, которые мы ранее добавили.

И наконец, инструкция

```
return app.exec();
```

запускает бесконечный цикл обработки событий. Инструкции, расположенные после вызова метода `exec()`, будут выполнены только после завершения работы приложения. Так как результат выполнения метода `exec()` мы возвращаем из функции `main()`, дальнейшее выполнение программы будет прекращено, а код возврата передан операционной системе.

1.8. Запуск приложения двойным щелчком на значке файла

Запускать приложение в редакторе Qt Creator мы научились, теперь попробуем запустить приложение из командной строки (вторую команду набираем без символа перевода строки):

```
C:\Users\Unicross>cd C:\cpp\projectsQt\Test-Debug\debug

C:\cpp\projectsQt\Test-Debug\debug>set
Path=C:\Qt\Tools\mingw810_64\bin;C:\Qt\6.1.0\mingw81_64\bin;%Path%

C:\cpp\projectsQt\Test-Debug\debug>Test.exe
```

Программа будет успешно запущена, т. к. мы предварительно добавили все необходимые пути в начало переменной окружения `PATH`.

Запустить программу двойным щелчком на значке файла можно с помощью скрипта script.bat:

```
@echo off
title Запуск программы Test.exe
set Path=C:\Qt\Tools\mingw810_64\bin;C:\Qt\6.1.0\mingw81_64\bin;%Path%
@echo.
C:\cpp\projectsQt\Test-Debug\debug\Test.exe
@echo.
pause
```

Здесь мы также добавили все необходимые пути в начало переменной окружения `PATH`, поэтому проблем с запуском быть не должно.

Однако запуск оконных приложений обычно производится с помощью двойного щелчка левой кнопкой мыши на значке файла. Если мы сейчас перейдем в каталог C:\cpp\projectsQt\Test-Debug\debug и попробуем так запустить файл Test.exe, то будет выведено сообщение о ненайденной библиотеке динамической компоновки (рис. 1.29).

Чтобы запустить программу на выполнение двойным щелчком, нужно предварительно добавить все необходимые библиотеки в каталог с программой. Выполнить такое действие автоматически позволяет программа windeployqt.exe, расположенная в каталоге C:\Qt\6.1.0\mingw81_64\bin. Прежде чем запускать программу, выполним сборку приложения в режиме **Выпуск** (рис. 1.30).

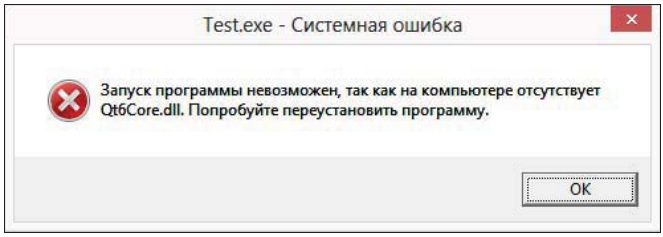

Рис. 1.29. Окно с сообщением об ошибке

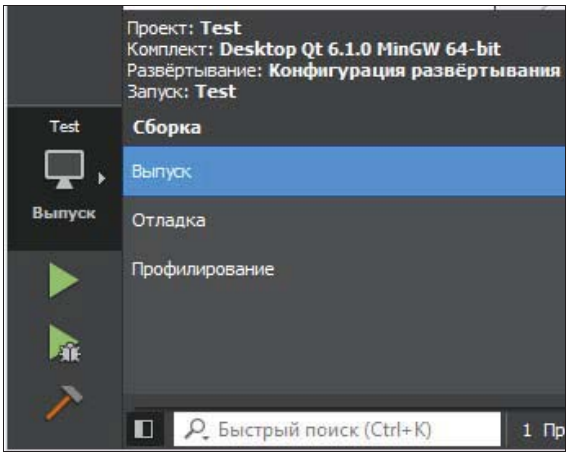

Рис. 1.30. Выбор режима сборки

После сборки копируем файл Test.exe из каталога C:\cpp\projectsQt\Test-Release\release в каталог C:\book\test. Далее в командной строке выполняем следующие инструкции (первую команду указываем на одной строке без символа перевода строки):

```
C:\Users\Unicross>set
Path=C:\Qt\Tools\mingw810_64\bin;C:\Qt\6.1.0\mingw81_64\bin;%Path%

C:\Users\Unicross>cd C:\Qt\6.1.0\mingw81_64\bin

C:\Qt\6.1.0\mingw81_64\bin>windeployqt.exe C:\book\test\Test.exe
```

В результате все необходимые библиотеки динамической компоновки будут скопированы в каталог C:\book\test и мы сможем запустить приложение двойным щелчком левой кнопкой мыши на значке файла.

1.9. ООП-стиль создания окна

Библиотека Qt написана в объектно-ориентированном стиле (ООП-стиле) и содержит более тысячи классов. Иерархия наследования всех классов имеет слишком большой размер, поэтому приводить ее в книге нет возможности. Тем не менее, чтобы показать зависимости, при описании компонентов иерархия наследования

конкретного класса будет показываться. В качестве примера выведем базовые классы для класса `QWidget`:

```
(QObject, QPaintDevice) — QWidget
```

Как видно из примера, класс `QWidget` наследует два класса — `QObject` и `QPaintDevice`. Класс `QObject` является классом верхнего уровня, его наследует большинство классов в Qt. В свою очередь, класс `QWidget` является базовым классом для всех визуальных компонентов. Таким образом, класс `QWidget` наследует все свойства и методы базовых классов. Это обстоятельство следует учитывать при изучении документации, т. к. в ней описываются свойства и методы конкретного класса, а на унаследованные свойства и методы даются только ссылки.

В своих программах вы можете наследовать стандартные классы и добавлять новую функциональность. При создании проекта мастер автоматически создал файлы widget.h и widget.cpp, давайте ими воспользуемся и создадим окно в ООП-стиле. Содержимое файла widget.h приведено в листинге 1.3, файла widget.cpp — в листинге 1.4, а файла main.cpp — в листинге 1.5.

Листинг 1.3. Содержимое файла C:\cpp\projectsQt\Test\widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QPushButton>
#include <QVBoxLayout>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent=nullptr);
   ~Widget();

private:
   QLabel *label;
   QPushButton *btnQuit;
   QVBoxLayout *vbox;
};
#endif // WIDGET_H
```

Листинг 1.4. Содержимое файла C:\cpp\projectsQt\Test\widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
```

```
{
   label = new QLabel("Привет, мир!");
   label->setAlignment(Qt::AlignCenter);
   btnQuit = new QPushButton("&Закрыть окно");
   vbox = new QVBoxLayout();
   vbox->addWidget(label);
   vbox->addWidget(btnQuit);
   setLayout(vbox);
   connect(btnQuit, SIGNAL(clicked()),
           qApp, SLOT(quit()));
}

Widget::~Widget()
{
}
```

Листинг 1.5. Содержимое файла C:\cpp\projectsQt\Test\main.cpp

```
#include "widget.h"

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   Widget window;
   window.setWindowTitle("ООП-стиль создания окна");
   window.resize(350, 70);
   window.show();
   return app.exec();
}
```

Итак, сначала в файле widget.h подключаются заголовочные файлы. Затем идет объявление класса `Widget`, который наследует класс `QWidget`:

```
class Widget : public QWidget
```

Помимо класса `QWidget` можно наследовать и другие классы, которые являются наследниками класса `QWidget`, например класс `QFrame` (окно с рамкой) или `QDialog` (диалоговое окно). При наследовании класса `QDialog` окно будет выравниваться по центру экрана (или по центру родительского окна) и иметь только одну кнопку в заголовке окна — **Закрыть**. Кроме того, можно наследовать класс `QMainWindow`, который представляет главное окно приложения с меню, панелями инструментов и строкой состояния. Наследование класса `QMainWindow` имеет свои отличия, которые мы будем рассматривать в отдельной главе.

Обратите внимание на макрос `Q_OBJECT`. Его обязательно нужно указывать для всех классов, наследующих класс `QObject`.

Объявление конструктора класса имеет следующий прототип:

```
Widget(QWidget *parent=nullptr);
```

В качестве параметра конструктор принимает указатель на родительский компонент (parent). Родительский компонент может отсутствовать, поэтому в определении конструктора параметру присваивается значение по умолчанию (nullptr). В файле widget.cpp сначала вызывается конструктор базового класса и ему передается указатель на родительский компонент:

```
Widget::Widget(QWidget *parent)
    : QWidget(parent)
```

Следующие инструкции внутри конструктора создают объекты надписи, кнопки и контейнеры, затем добавляют компоненты в контейнер, а сам контейнер — в основное окно. Следует обратить внимание на то, что указатели на объекты надписи и кнопки сохраняется в свойствах экземпляра класса. В дальнейшем из методов класса можно управлять этими объектами, например изменить текст надписи. Если объекты не сохранить, то получить к ним доступ будет не так просто.

Далее производится назначение обработчика нажатия кнопки:

```
connect(btnQuit, SIGNAL(clicked()),
        qApp, SLOT(quit()));
```

В отличие от предыдущего примера (листинг 1.2), метод connect() вызывается не через класс QObject, а через экземпляр нашего класса. Это возможно, т. к. мы наследовали класс QWidget, который, в свою очередь, является наследником класса QObject и наследует метод connect(). Еще одно отличие состоит в обращении к объекту приложения через макрос qApp.

В предыдущем примере (листинг 1.2) мы выравнивали надпись с помощью HTML-разметки. Произвести аналогичную операцию позволяет также метод setAlignment(), которому следует передать константу Qt::AlignCenter:

```
label->setAlignment(Qt::AlignCenter);
```

В некоторых случаях использование ООП-стиля является обязательным. Например, чтобы обработать нажатие клавиши клавиатуры, необходимо наследовать какой-либо класс и переопределить в нем метод с предопределенным названием. Какие методы необходимо переопределять, мы рассмотрим при изучении обработки событий.

1.10. Создание проекта с формой

Если вы ранее пользовались Visual Studio или Delphi, то вспомните, что размещение компонентов на форме производили с помощью мыши. Щелкали левой кнопкой мыши на соответствующей кнопке на панели инструментов и перетаскивали компонент на форму. Далее с помощью инспектора свойств производили настройку значений некоторых свойств, а остальные свойства получали значения по умолчанию. При этом весь код генерировался автоматически. Редактор Qt Creator также позволяет использовать формы.

Давайте по аналогии с проектом Test создадим новый проект с названием TestForm. Только на третьем шаге мастера установим флажок **Создать форму** (рис. 1.31).

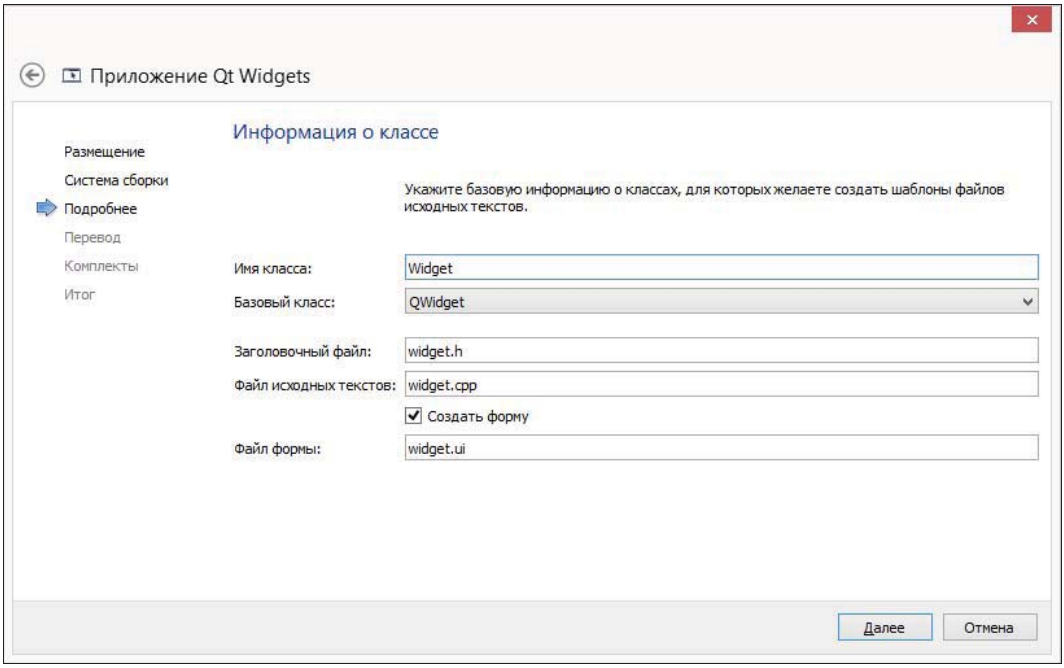

Рис. 1.31. Создание проекта. Шаг 3

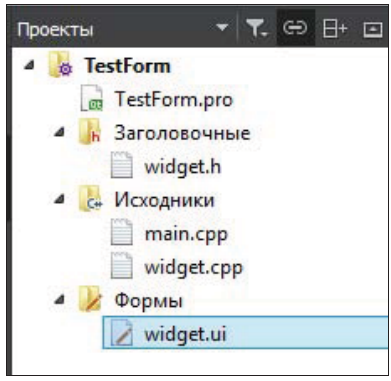

Рис. 1.32. Файлы проекта `TestForm`

Созданный проект на вкладке **Проекты** будет выглядеть так, как показано на рис. 1.32.

Чтобы отобразить форму, щелкаем левой кнопкой мыши на названии файла widget.ui на вкладке **Проекты**. Результат показан на рис. 1.33.

На панели слева находим компонент **Label**, щелкаем на нем левой кнопкой мыши и, не отпуская кнопку, перетаскиваем компонент на форму. Далее находим компонент **Push Button** и перетаскиваем его на форму ниже надписи. Выделяем форму, щелкаем правой кнопкой мыши и из контекстного меню выбираем пункт **Компоновка | Скомпоновать по вертикали**.

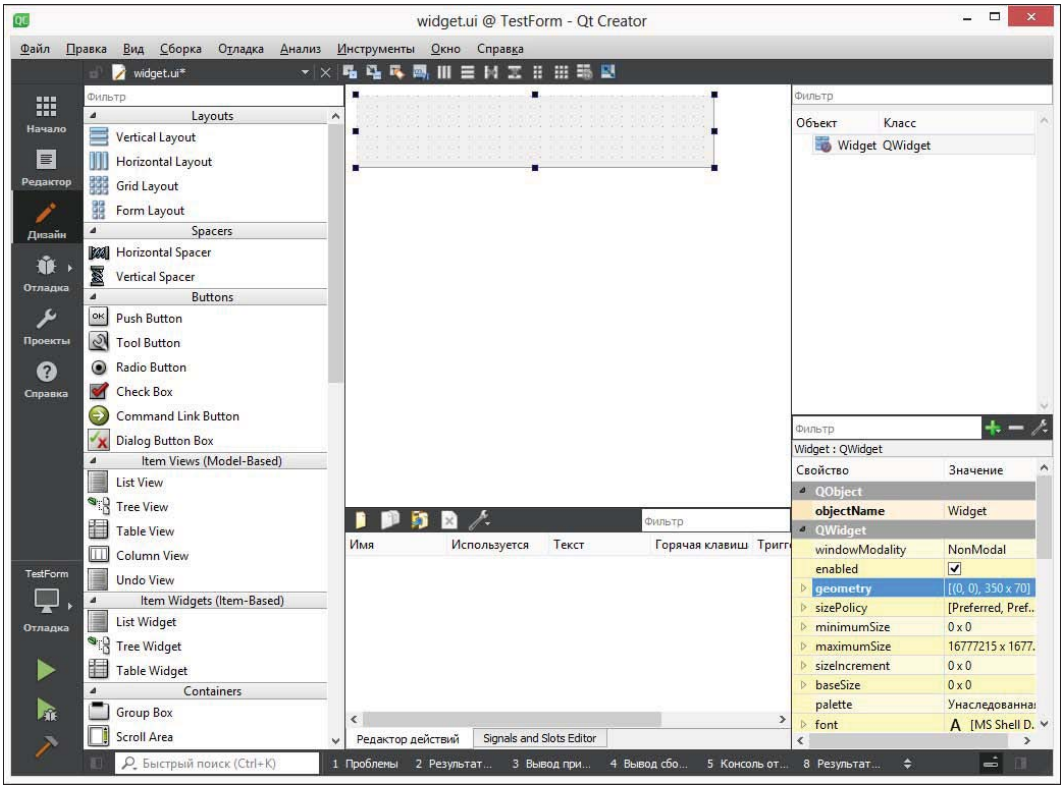

Рис. 1.33. Редактирование формы в режиме **Дизайн**

Теперь изменим свойства компонентов. Выделяем текстовую надпись и на панели справа находим свойство **objectName**. Указываем для него значение `label`. Далее находим свойство **text** и для него указываем значение `Привет, мир!`. Можно также сделать двойной щелчок мыши на надписи и ввести значение. Находим свойство **alignment**, раскрываем список и для свойства **Горизонтальное** указываем значение `AlignHCenter`.

Выделяем кнопку и для свойства **objectName** указываем значение `btn1`, а для свойства **text** задаем значение `Изменить надпись`. Можно также сделать двойной щелчок мыши на кнопке и ввести значение. Затем щелкаем правой кнопкой мыши на кнопке и из контекстного меню выбираем пункт **Перейти к слоту**. Откроется окно, показанное на рис. 1.34. Выделяем пункт **clicked()** и нажимаем кнопку **OK**. В результате в файл widget.cpp будет вставлен следующий код, описывающий обработчик нажатия кнопки:

```
void Widget::on_btn1_clicked()
{

}
```

Давайте внутри обработчика нажатия кнопки изменим текст надписи. Для этого обращаемся к надписи через объект `ui` и вызываем метод `setText()`:

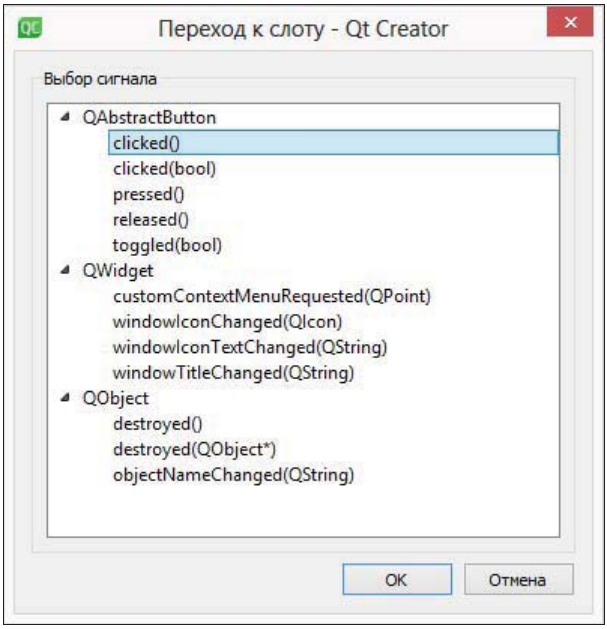

Рис. 1.34. Окно **Переход к слоту**

```
void Widget::on_btn1_clicked()
{
   ui->label->setText("Новый текст");
}
```

Содержимое файла main.cpp приведено в листинге 1.6, файла widget.h — в листинге 1.7, файла widget.cpp — в листинге 1.8, а файла widget.ui — в листинге 1.9.

Листинг 1.6. Содержимое файла C:\cpp\projectsQt\TestForm\main.cpp

```
#include "widget.h"

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   Widget window;
   window.setWindowTitle("Заголовок окна");
   window.show();
   return app.exec();
}
```

Листинг 1.7. Содержимое файла C:\cpp\projectsQt\TestForm\widget.h

```
#ifndef WIDGET_H
#define WIDGET_H
```

```
#include <QWidget>
#include <QApplication>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent = nullptr);
   ~Widget();

private slots:
   void on_btn1_clicked();

private:
   Ui::Widget *ui;
};
#endif // WIDGET_H
```

Листинг 1.8. Содержимое файла C:\cpp\projectsQt\TestForm\widget.cpp

```
#include "widget.h"
#include "ui_widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
   , ui(new Ui::Widget)
{
   ui->setupUi(this);
}

Widget::~Widget()
{
   delete ui;
}

void Widget::on_btn1_clicked()
{
   ui->label->setText("Новый текст");
}
```

Листинг 1.9. Содержимое файла C:\cpp\projectsQt\TestForm\widget.ui

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
```

```
 <class>Widget</class>
 <widget class="QWidget" name="Widget">
  <property name="geometry">
   <rect>
    <x>0</x>
    <y>0</y>
    <width>350</width>
    <height>70</height>
   </rect>
  </property>
  <property name="windowTitle">
   <string>Widget</string>
  </property>
  <layout class="QVBoxLayout" name="verticalLayout">
   <item>
    <widget class="QLabel" name="label">
     <property name="text">
      <string>Привет, мир!</string>
     </property>
     <property name="alignment">
      <set>Qt::AlignCenter</set>
     </property>
    </widget>
   </item>
   <item>
    <widget class="QPushButton" name="btn1">
     <property name="text">
      <string>Изменить надпись</string>
     </property>
    </widget>
   </item>
  </layout>
 </widget>
 <resources/>
 <connections/>
</ui>
```

Как видите, и в Qt можно создавать формы и размещать компоненты с помощью мыши. Все это очень удобно. Тем не менее, чтобы полностью владеть технологией, необходимо уметь создавать код вручную. Поэтому в оставшейся части книги мы больше не будем рассматривать создание формы в режиме **Дизайн**. Научиться «мышкотворчеству» вы сможете и самостоятельно без моей помощи.

1.11. Доступ к документации

Библиотека Qt содержит свыше тысячи классов. Если описывать один класс на одной странице, то объем книги по Qt будет более тысячи страниц. Однако уложиться в одну страницу практически невозможно, следовательно, объем вырастет в два, а то и три раза. Издать книгу такого большого объема не представляется возможным, поэтому в этой книге мы рассмотрим только наиболее часто используемые возможности библиотеки Qt, те возможности, которые вы будете использовать каждый день в своей практике. Чтобы получить полную информацию, следует обращаться к документации. Где ее найти, мы сейчас и рассмотрим.

Самая последняя версия документации в формате HTML доступна на сайте **https://doc.qt.io/**. Документация входит также в состав дистрибутива и доступна в редакторе Qt Creator. Чтобы отобразить документацию, на левой боковой панели выбираем пункт **Справка**. В результате окно будет выглядеть так, как показано на рис. 1.35. Чтобы получить справку по какому-либо классу, нужно выделить название класса в коде и нажать клавишу <F1>. Помимо документации в состав дистрибутива входят примеры, которые можно увидеть, выбрав на левой боковой панели пункт **Начало**, а затем нажав кнопку **Примеры**. Обязательно изучите эти примеры.

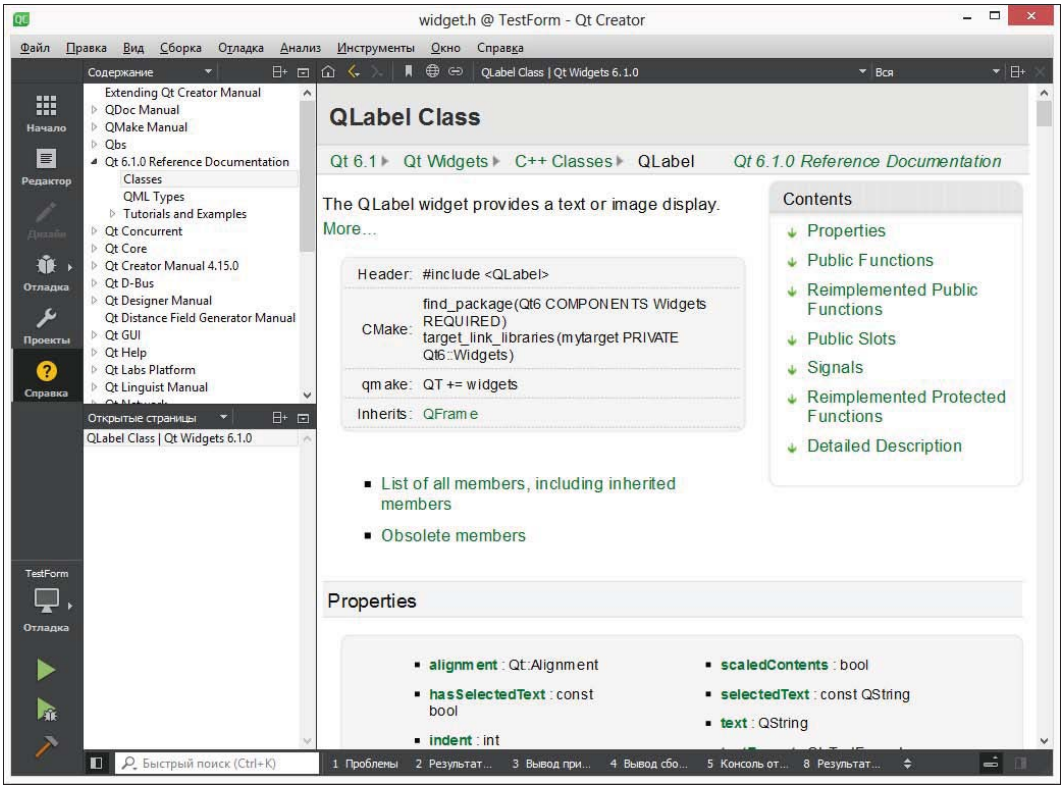

Рис. 1.35. Отображение документации в редакторе Qt Creator

ГЛАВА 2

Работа с символами и строками

Прежде чем начинать рассматривать создание оконных приложений, необходимо изучить работу с символами и строками. Ведь файлы с программой у нас сохраняются в кодировке UTF-8, а русские символы в этой кодировке кодируются двумя байтами. По этой причине использовать функции, предназначенные для работы с С-строками, нельзя. Выполнять операции с символами и строками в кодировке Unicode позволяют классы `QChar` и `QString`. Давайте их рассмотрим подробно.

2.1. Псевдонимы для элементарных типов

Помимо стандартных типов данных из языка C++ в Qt-программе можно использовать дополнительные типы (минимальный размер типа гарантируется для поддерживаемых библиотекой Qt платформ):

- `qint8` — псевдоним для типа `signed char`:

```
typedef signed char qint8;
```

- `quint8` и `uchar` — псевдонимы для типа `unsigned char`:

```
typedef unsigned char quint8;
typedef unsigned char uchar;
```

- `qint16` — псевдоним для типа `short`:

```
typedef short qint16;
```

- `quint16` и `ushort` — псевдонимы для типа `unsigned short`:

```
typedef unsigned short quint16;
typedef unsigned short ushort;
```

- `qint32` — псевдоним для типа `int`:

```
typedef int qint32;
```

- `quint32` и `uint` — псевдонимы для типа `unsigned int`:

```
typedef unsigned int quint32;
typedef unsigned int uint;
```

- `ulong` — псевдоним для типа `unsigned long`:

```
typedef unsigned long ulong;
```

- `qint64` и `qlonglong` — псевдонимы для типа `long long`:

```
typedef long long qint64;
typedef qint64 qlonglong;
```

  При указании значения после числа необходимо добавить буквы `LL` или использовать макрос `Q_INT64_C()`:

```
qint64 x = 9223372036854775807LL;
qint64 y = Q_INT64_C(9223372036854775807);
```

- `quint64` и `qulonglong` — псевдонимы для типа `unsigned long long`:

```
typedef unsigned long long quint64;
typedef quint64 qulonglong;
```

  При указании значения после числа необходимо добавить буквы `ULL` или использовать макрос `Q_UINT64_C()`:

```
quint64 x = 18446744073709551615ULL;
quint64 y = Q_UINT64_C(18446744073709551615);
```

- `qreal` — вещественное число. Соответствует типу `double` (в большинстве случаев) или `float` (при указании дополнительно флага):

```
#if defined(QT_COORD_TYPE)
typedef QT_COORD_TYPE qreal;
#else
typedef double qreal;
#endif
```

Дополнительные типы:

```
typedef QIntegerForSizeof<void *>::Unsigned quintptr;
typedef QIntegerForSizeof<void *>::Signed qptrdiff;
typedef qptrdiff qintptr;
using qsizetype = QIntegerForSizeof<std::size_t>::Signed;
```

2.2. Класс *QChar:* символ в кодировке Unicode

Класс `QChar` описывает символ в кодировке UTF-16 (один символ занимает два байта). Подключение заголовочного файла:

```
#include <QChar>
```

2.2.1. Создание объекта

Создать объект позволяют следующие конструкторы:

```
QChar()
QChar(char ch)
```

```
QChar(uchar ch)
QChar(wchar_t ch)
QChar(char16_t ch)
QChar(QLatin1Char ch)
QChar(QChar::SpecialCharacter ch)
QChar(int code)
QChar(uint code)
QChar(short code)
QChar(uchar cell, uchar row)
QChar(ushort code)
```

Пример:

```
QChar ch1(1082);
qDebug() << ch1;     // '\u043a'
QChar ch2(L'к');
qDebug() << ch2;     // '\u043a'
QChar ch3(u'к');
qDebug() << ch3;     // '\u043a'
QChar ch4;
qDebug() << ch4;     // '\x0'
```

Можно также воспользоваться статическими методами `fromLatin1()` и `fromUcs2()`. Прототипы методов:

```
static QChar fromLatin1(char c)
static QChar fromUcs2(char16_t c)
```

Пример:

```
QChar ch1 = QChar::fromLatin1('d');
qDebug() << ch1; // 'd'
QChar ch2 = QChar::fromUcs2(u'к');
qDebug() << ch2; // '\u043a'
```

Над двумя объектами определены операции ==, !=, <, >, <= и >=:

```
QChar ch1(u'd');
QChar ch2(u'd');
QChar ch3(u's');
qDebug() << (ch1 == ch2); // true
qDebug() << (ch1 != ch2); // false
qDebug() << (ch1 < ch3);  // true
qDebug() << (ch1 > ch3);  // false
qDebug() << (ch1 <= ch3); // true
qDebug() << (ch1 >= ch3); // false
```

Преобразовать объект в другой тип данных позволяют методы `unicode()` и `toLatin1()`. Прототипы методов:

```
char16_t unicode() const
char16_t &unicode()
char toLatin1() const
```

Пример:

```
QChar ch1 = L'к';
QChar ch2 = L'd';
char16_t ch3 = ch1.unicode();
qDebug() << ch3;            // '\u043a'
qDebug() << (ushort)ch3;    // 1082
qDebug() << ch2.toLatin1(); // d
```

2.2.2. Изменение регистра символа

Изменить регистр символа позволяют следующие методы:

- `toLower()` — возвращает символ в нижнем регистре. Прототип метода:

  ```
  QChar toLower() const
  ```

  Пример:

  ```
  QChar ch1 = L'K';
  QChar ch2 = L'D';
  qDebug() << ch1.toLower(); // '\u043a'
  qDebug() << ch2.toLower(); // 'd'
  ```

- `toUpper()` — возвращает символ в верхнем регистре. Прототип метода:

  ```
  QChar toUpper() const
  ```

  Пример:

  ```
  QChar ch1 = L'к';
  QChar ch2 = L'd';
  qDebug() << ch1.toUpper(); // '\u041a'
  qDebug() << ch2.toUpper(); // 'D'
  ```

- `isLower()` — возвращает значение `true`, если объект содержит символ в нижнем регистре, и `false` — в противном случае. Прототип метода:

  ```
  bool isLower() const
  ```

  Пример:

  ```
  QChar ch1 = L'D';
  QChar ch2 = L'd';
  qDebug() << ch1.isLower(); // false
  qDebug() << ch2.isLower(); // true
  ```

- `isUpper()` — возвращает значение `true`, если объект содержит символ в верхнем регистре, и `false` — в противном случае. Прототип метода:

  ```
  bool isUpper() const
  ```

  Пример:

  ```
  QChar ch1 = L'D';
  QChar ch2 = L'd';
  ```

```
qDebug() << ch1.isUpper(); // true
qDebug() << ch2.isUpper(); // false
```

2.2.3. Проверка типа содержимого символа

Проверить тип содержимого символа позволяют следующие методы:

- `isDigit()` — возвращает значение `true`, если символ является десятичной цифрой, в противном случае — `false`. Прототип метода:

  ```
  bool isDigit() const
  ```

  Пример:

  ```
  QChar ch1 = L'1';
  QChar ch2 = L'd';
  qDebug() << ch1.isDigit(); // true
  qDebug() << ch2.isDigit(); // false
  ```

- `isNumber()` — возвращает значение `true`, если символ является числом, в противном случае — `false`. Прототип метода:

  ```
  bool isNumber() const
  ```

  Пример:

  ```
  QChar ch1 = L'1';
  QChar ch2 = L'd';
  qDebug() << ch1.isNumber(); // true
  qDebug() << ch2.isNumber(); // false
  ```

- `digitValue()` — возвращает числовое значение цифры или значение `-1`, если символ не является цифрой. Прототип метода:

  ```
  int digitValue() const
  ```

  Пример:

  ```
  QChar ch1 = L'2';
  QChar ch2 = L'd';
  qDebug() << ch1.digitValue(); // 2
  qDebug() << ch2.digitValue(); // -1
  ```

- `isLetter()` — возвращает значение `true`, если символ является буквой, в противном случае — `false`. Прототип метода:

  ```
  bool isLetter() const
  ```

  Пример:

  ```
  QChar ch1 = L'1';
  QChar ch2 = L'd';
  qDebug() << ch1.isLetter(); // false
  qDebug() << ch2.isLetter(); // true
  ```

- `isSpace()` — возвращает значение `true`, если символ является пробельным символом (пробелом, табуляцией, переводом строки или возвратом каретки), в противном случае — `false`. Прототип метода:

```
bool isSpace() const
```

Пример:

```
QChar ch1 = L'd';
QChar ch2 = L' ';
qDebug() << ch1.isSpace(); // false
qDebug() << ch2.isSpace(); // true
```

- `isLetterOrNumber()` — возвращает значение `true`, если символ является буквой или цифрой, в противном случае — `false`. Прототип метода:

```
bool isLetterOrNumber() const
```

Пример:

```
QChar ch1 = L'd';
QChar ch2 = L'\r';
qDebug() << ch1.isLetterOrNumber(); // true
qDebug() << ch2.isLetterOrNumber(); // false
```

- `isPunct()` — возвращает значение `true`, если символ является символом пунктуации, в противном случае — `false`. Прототип метода:

```
bool isPunct() const
```

Пример:

```
QChar ch1 = L'.';
QChar ch2 = L',';
qDebug() << ch1.isPunct(); // true
qDebug() << ch2.isPunct(); // true
```

- `isPrint()` — возвращает значение `true`, если символ является печатаемым (включая пробел), в противном случае — `false`. Прототип метода:

```
bool isPrint() const
```

Пример:

```
QChar ch1 = L'8';
QChar ch2 = L'\r';
qDebug() << ch1.isPrint(); // true
qDebug() << ch2.isPrint(); // false
```

- `isNull()` — возвращает значение `true`, если символ является нулевым символом, в противном случае — `false`. Прототип метода:

```
bool isNull() const
```

Пример:

```
QChar ch1;
QChar ch2 = L'd';
```

```
qDebug() << ch1.isNull(); // true
qDebug() << ch2.isNull(); // false
```

Прочие методы:

```
bool isMark() const
bool isNonCharacter() const
bool isSymbol() const
```

2.3. Класс *QString:* строка в кодировке Unicode

Класс `QString` описывает строку в кодировке UTF-16 (каждый символ занимает два байта и имеет тип `QChar`). Подключение заголовочного файла:

```
#include <QString>
```

2.3.1. Создание объекта

Создать объект позволяют следующие конструкторы класса `QString`:

```
QString()
QString(QChar ch)
QString(qsizetype size, QChar ch)
QString(const QChar *unicode, qsizetype size = -1)
QString(const char *str)
QString(const QByteArray &ba)
QString(const char8_t *str)
QString(QLatin1String str)
QString(const QString &other)
QString(QString &&other)
```

Примеры:

```
QChar ch = L'd';
QString str1 = ch;
qDebug() << str1; // "d"
QString str2(5, ch);
qDebug() << str2; // "ddddd"
```

Пятый конструктор принимает С-строку в кодировке UTF-8. Учитывая, что файл у нас сохранен в кодировке UTF-8, никаких дополнительных действий выполнять не нужно. Везде, где ожидается объект `QString`, мы можем передать С-строку:

```
QString str = "строка";
qDebug() << str; // "строка"
```

Создать объект позволяют также следующие статические методы:

```
static QString fromUtf8(const char *str, qsizetype size)
static QString fromUtf8(QByteArrayView str)
static QString fromUtf8(const QByteArray &str)
```

```
static QString fromUtf8(const char8_t *str)
static QString fromUtf8(const char8_t *str, qsizetype size)

static QString fromWCharArray(const wchar_t *string, qsizetype size = -1)
static QString fromRawData(const QChar *unicode, qsizetype size)
static QString fromStdString(const std::string &str)
static QString fromStdWString(const std::wstring &str)

static QString fromLatin1(const char *str, qsizetype size)
static QString fromLatin1(QByteArrayView str)
static QString fromLatin1(const QByteArray &str)
static QString fromLocal8Bit(const char *str, qsizetype size)
static QString fromLocal8Bit(QByteArrayView str)
static QString fromLocal8Bit(const QByteArray &str)

static QString fromUtf16(const char16_t *unicode, qsizetype size = -1)
static QString fromUcs4(const char32_t *unicode, qsizetype size = -1)
static QString fromStdU16String(const std::u16string &str)
static QString fromStdU32String(const std::u32string &str)
static QString fromCFString(CFStringRef string)
static QString fromNSString(const NSString *string)
```

Пример:

```
QString str1 = QString::fromUtf8("строка");
qDebug() << str1; // "строка"
std::string s("строка");
QString str2 = QString::fromStdString(s);
qDebug() << str2; // "строка"
std::wstring w(L"строка");
QString str3 = QString::fromStdWString(w);
qDebug() << str3; // "строка"
```

Присвоить значение после создания объекта позволяет оператор =:

```
QString str;
str = "строка";
qDebug() << str;  // "строка"
```

Можно также воспользоваться следующими методами:

```
QString &setRawData(const QChar *unicode, qsizetype size)
QString &setUnicode(const QChar *unicode, qsizetype size)
QString &setUtf16(const ushort *unicode, qsizetype size)
```

2.3.2. Преобразование объекта в другой тип данных

Преобразовать объект `QString` в другой тип данных позволяют следующие методы:

```
QChar *data()
const QChar *data() const
```

```
const QChar *unicode() const
const QChar *constData() const
const ushort *utf16() const
QByteArray toUtf8() const
std::string toStdString() const
std::wstring toStdWString() const
qsizetype toWCharArray(wchar_t *array) const
std::u16string toStdU16String() const
std::u32string toStdU32String() const
QList<uint> toUcs4() const
QByteArray toLatin1() const
QByteArray toLocal8Bit() const
CFStringRef toCFString() const
NSString *toNSString() const
```

Пример:

```
QString str1 = "строка";
std::string s = str1.toStdString();
QString str2 = QString::fromStdString(s);
qDebug() << str2; // "строка"
std::wstring w = str1.toStdWString();
qDebug() << w;    // "строка"
```

2.3.3. Получение и изменение размера строки

Для получения и изменения размера строки предназначены следующие методы:

- `size()`, `length()` и `count()` — возвращают текущее количество символов в строке. Прототипы методов:

  ```
  qsizetype size() const
  qsizetype length() const
  qsizetype count() const
  ```

  Пример:

  ```
  QString str("строка");
  qDebug() << str.size();    // 6
  qDebug() << str.length();  // 6
  qDebug() << str.count();   // 6
  ```

- `capacity()` — возвращает количество символов, которое можно записать в строку без перераспределения памяти. Прототип метода:

  ```
  qsizetype capacity() const
  ```

  Пример:

  ```
  QString str("строка");
  qDebug() << str.size();     // 6
  qDebug() << str.capacity(); // 12
  ```

```
str += " строка2 строка3";
qDebug() << str.size();     // 22
qDebug() << str.capacity(); // 24
```

- `reserve()` — позволяет задать минимальное количество символов, которое можно записать в строку без перераспределения памяти. Как видно из предыдущего примера, выделение дополнительной памяти производится автоматически с некоторым запасом. Если дозапись в строку производится часто, то это может снизить эффективность программы, т. к. перераспределение памяти будет выполнено несколько раз. Поэтому, если количество символов заранее известно, следует указать его с помощью метода `reserve()`. Прототип метода:

  ```
  void reserve(qsizetype size)
  ```

  Пример указания минимального размера строки:

  ```
  QString str("строка");
  str.reserve(50);
  qDebug() << str.size();     // 6
  qDebug() << str.capacity(); // 50
  str += " строка2 строка3";
  qDebug() << str.size();     // 22
  qDebug() << str.capacity(); // 50
  ```

- `shrink_to_fit()` и `squeeze()` — уменьшают размер строки до минимального значения. Прототипы методов:

  ```
  void shrink_to_fit()
  void squeeze()
  ```

  Пример:

  ```
  QString str("строка");
  str.reserve(50);
  qDebug() << str.capacity(); // 50
  str.squeeze();
  qDebug() << str.capacity(); // 6
  ```

- `resize()` — задает количество символов в строке, равное числу `n`. Если указанное количество символов меньше текущего количества, то лишние символы будут удалены. Если количество символов необходимо увеличить, то в параметре `ch` можно указать символ, который заполнит новое пространство. Прототипы метода:

  ```
  void resize(qsizetype n)
  void resize(qsizetype n, QChar ch)
  ```

  Пример:

  ```
  QString str("строка");
  str.resize(4);
  qDebug() << str; // "стро"
  str.resize(8, L'*');
  qDebug() << str; // "стро****"
  ```

- `truncate()` — обрезает строку до указанного количества символов. Если размер строки меньше указанного количества, то ничего не происходит. Прототип метода:

  ```
  void truncate(qsizetype position)
  ```

  Пример:

  ```
  QString str("строка");
  str.truncate(4);
  qDebug() << str; // "стро"
  ```

- `clear()` — удаляет все символы. Прототип метода:

  ```
  void clear()
  ```

  Пример:

  ```
  QString str("строка");
  str.clear();
  qDebug() << str.size(); // 0
  ```

- `isEmpty()` — возвращает значение `true`, если строка не содержит символов, и `false` — в противном случае. Прототип метода:

  ```
  bool isEmpty() const
  ```

  Пример:

  ```
  QString str("строка");
  qDebug() << str.isEmpty(); // false
  str.clear();
  qDebug() << str.isEmpty(); // true
  ```

- `isNull()` — возвращает значение `true`, если объект не содержит строки, даже пустой, и `false` — в противном случае. Прототип метода:

  ```
  bool isNull() const
  ```

  Пример:

  ```
  QString str1;
  qDebug() << str1.isNull();  // true
  QString str2("");
  qDebug() << str2.isNull();  // false
  ```

- `fill()` — заменяет все символы в строке указанным символом. Во втором параметре можно передать новый размер строки. Прототип метода:

  ```
  QString &fill(QChar ch, qsizetype size = -1)
  ```

  Пример:

  ```
  QString str = "ABC";
  str.fill(L'*');
  qDebug() << str;               // "***"
  str.fill(L'+', 2);
  qDebug() << str;               // "++"
  ```

- `leftJustified()` — задает новый размер строки. Метод возвращает измененную строку. Прототип метода:

```
QString leftJustified(qsizetype width,
   QChar fill = QLatin1Char(' '), bool truncate = false) const
```

Если число `width` больше размера строки, то после строки будут добавлены символы `fill`:

```
QString str = "строка";
QString str2 = str.leftJustified(10, L'*');
qDebug() << str2; // "строка****"
```

Если число `width` меньше размера строки, то поведение зависит от параметра `truncate`. Если параметр `truncate` имеет значение `false`, то возвращается вся строка, в противном случае строка обрезается:

```
QString str = "строка";
QString str2 = str.leftJustified(3, L'*');
qDebug() << str2; // "строка"
str2 = str.leftJustified(3, L'*', true);
qDebug() << str2; // "стр"
```

- `rightJustified()` — задает новый размер строки. Метод возвращает измененную строку. Прототип метода:

```
QString rightJustified(qsizetype width,
   QChar fill = QLatin1Char(' '), bool truncate = false) const
```

Если число `width` больше размера строки, то перед строкой будут добавлены символы `fill`:

```
QString str = "строка";
QString str2 = str.rightJustified(10, L'*');
qDebug() << str2; // "****строка"
```

Если число `width` меньше размера строки, то поведение зависит от параметра `truncate`. Если параметр `truncate` имеет значение `false`, то возвращается вся строка, в противном случае строка обрезается:

```
QString str = "строка";
QString str2 = str.rightJustified(3, L'*');
qDebug() << str2; // "строка"
str2 = str.rightJustified(3, L'*', true);
qDebug() << str2; // "стр"
```

2.3.4. Доступ к отдельным символам

К любому символу в строке можно обратиться как к элементу массива. Достаточно указать его индекс в квадратных скобках. Нумерация начинается с нуля. Можно как получить символ, так и изменить его. Если индекс выходит за границы диапазона, то возвращаемое значение не определено. Каждый символ в строке является объектом `QChar`.

Пример доступа к символу по индексу:

```
QString str = "string";
qDebug() << str[0];   // 's'
str[0] = QChar(L'S');
qDebug() << str;      // "String"
```

Для доступа к символам можно также воспользоваться следующими методами:

- `at()` — возвращает символ, расположенный по индексу `pos`. Метод позволяет как получить символ, так и изменить его. Если индекс выходит за границы диапазона, то возникнет ошибка. Прототип метода:

  ```
  const QChar at(qsizetype pos) const
  ```

  Пример:

  ```
  QString str = "string";
  qDebug() << str.at(0);   // 's'
  ```

- `front()` — возвращает ссылку на первый символ в строке или сам символ. Метод позволяет как получить символ, так и изменить его. Прототипы метода:

  ```
  QChar front() const
  QChar &front()
  ```

  Пример:

  ```
  QString str = "String";
  str.front() = L's';
  qDebug() << str.front();   // 's'
  qDebug() << str;           // "string"
  ```

- `back()` — возвращает ссылку на последний символ в строке или сам символ. Метод позволяет как получить символ, так и изменить его. Прототипы метода:

  ```
  QChar back() const
  QChar &back()
  ```

  Пример:

  ```
  QString str = "String";
  str.back() = L'G';
  qDebug() << str.back();    // 'G'
  qDebug() << str;           // "StrinG"
  ```

2.3.5. Перебор символов строки

Перебрать символы внутри строки позволяет цикл `for`. В качестве примера выведем все символы по одному на строке:

```
QString str = "string";
for (qsizetype i = 0, len = str.size(); i < len; ++i) {
   qDebug() << str[i];
}
```

Для перебора всех символов строки удобно использовать цикл `for each`:

```
QString str = "string";
for (QChar &ch : str) {
   qDebug() << ch;
}
```

Примечание

Пример перебора символов строки с помощью итераторов приведен в листинге 2.1.

2.3.6. Итераторы

Итератор — это объект, выполняющий в контейнере роль указателя. С помощью итератора можно перемещаться внутри контейнера и получать доступ к отдельным элементам. В классе `QString` определены следующие типы итераторов:

- `iterator` — итератор. При увеличении значения итератор перемещается к концу строки. Пример объявления переменной:

  ```
  QString::iterator it;
  ```

- `const_iterator` — константный итератор. Изменить значение, на которое ссылается итератор, нельзя. Пример объявления переменной:

  ```
  QString::const_iterator it;
  ```

- `reverse_iterator` — обратный итератор. При увеличении значения итератор перемещается к началу строки. Пример объявления переменной:

  ```
  QString::reverse_iterator it;
  ```

- `const_reverse_iterator` — константный обратный итератор. Изменить значение, на которое ссылается итератор, нельзя. Пример объявления переменной:

  ```
  QString::const_reverse_iterator it;
  ```

Присвоить значения переменным позволяют следующие методы:

- `begin()` — возвращает итератор, установленный на первый символ строки. Прототипы метода:

  ```
  QString::iterator begin()
  QString::const_iterator begin() const
  ```

  Выведем первый символ строки:

  ```
  QString str = "String";
  QString::iterator it = str.begin();
  qDebug() << *it; // 'S'
  ```

- `end()` — возвращает итератор, установленный на позицию после последнего символа строки. Прототипы метода:

  ```
  QString::iterator end()
  QString::const_iterator end() const
  ```

Выведем последний символ строки:

```
QString str = "String";
QString::iterator it = str.end();
qDebug() << *(--it); // 'g'
```

- `cbegin()` и `constBegin()` — возвращают константный итератор, установленный на первый символ строки. Прототипы методов:

```
QString::const_iterator cbegin() const
QString::const_iterator constBegin() const
```

- `cend()` и `constEnd()` — возвращают константный итератор, установленный на позицию после последнего символа строки. Прототипы методов:

```
QString::const_iterator cend() const
QString::const_iterator constEnd() const
```

- `rbegin()` — возвращает обратный итератор, установленный на последний символ строки. Прототипы метода:

```
QString::reverse_iterator rbegin()
QString::const_reverse_iterator rbegin() const
```

Выведем последний символ строки:

```
QString str = "String";
QString::reverse_iterator it = str.rbegin();
qDebug() << *it; // 'g'
```

- `rend()` — возвращает обратный итератор, установленный на позицию перед первым символом строки. Прототипы метода:

```
QString::reverse_iterator rend()
QString::const_reverse_iterator rend() const
```

Выведем первый символ строки:

```
QString str = "String";
QString::reverse_iterator it = str.rend();
qDebug() << *(--it); // 'S'
```

- `crbegin()` — возвращает константный обратный итератор, установленный на последний символ строки. Прототип метода:

```
QString::const_reverse_iterator crbegin() const
```

- `crend()` — возвращает константный обратный итератор, установленный на позицию перед первым символом строки. Прототип метода:

```
QString::const_reverse_iterator crend() const
```

С итераторами можно производить такие же операции, как и с указателями. Чтобы получить или изменить значение, на которое ссылается итератор, перед названием переменной указывается оператор `*` (`*it`). Перемещение итератора осуществляется с помощью операторов `+`, `-`, `++` и `--`. Кроме того, итераторы можно сравнивать с по-

мощью операторов сравнения. В качестве примера изменим значение первого символа, а затем выведем все символы строки в прямом и обратном порядке с помощью цикла `for` (листинг 2.1).

Листинг 2.1. Перебор символов в строке с помощью итераторов

```
QString str = "String";
QString::iterator it1, it2;
QString::reverse_iterator it3, it4;
it1 = str.begin();
*it1 = L's'; // Изменение значения
// Перебор символов в прямом порядке
for (it1 = str.begin(), it2 = str.end(); it1 != it2; ++it1) {
   qDebug() << *it1;
}
qDebug() << "------------------";
// Перебор символов в обратном порядке
for (it3 = str.rbegin(), it4 = str.rend(); it3 != it4; ++it3) {
   qDebug() << *it3;
}
```

2.3.7. Конкатенация строк

Для объектов класса `QString` определена операция *конкатенации* (объединения строк). Оператор `+` позволяет объединить:

- два объекта класса `QString`:

```
QString str1 = "A", str2 = "B";
QString str3 = str1 + str2;
qDebug() << str3; // "AB"
```

- объект класса `QString` с С-строкой в кодировке UTF-8:

```
QString str1 = "A", str2, str3;
str2 = str1 + "B";
str3 = "B" + str1;
qDebug() << str2; // "AB"
qDebug() << str3; // "BA"
```

Помимо оператора `+` доступен оператор `+=`, который производит конкатенацию с присваиванием:

```
QString str1 = "A", str2 = "B";
str1 += "C";
qDebug() << str1; // "AC"
str2 += str1;
qDebug() << str2; // "BAC"
str2 += QChar(L'D');
qDebug() << str2; // "BACD"
```

Примечание

Вместо операторов + и += можно воспользоваться методами `push_back()` и `append()` *(см. разд. 2.3.8)*.

2.3.8. Добавление и вставка символов

Для добавления и вставки символов предназначены следующие методы:

- `push_back()` — добавляет символ или строку в конец исходной строки. Прототип метода:

```
void push_back(QChar ch)
void push_back(const QString &other)
```

Пример:

```
QString str = "String";
str.push_back(L'1');
qDebug() << str; // "String1"
str.push_back(" String2");
qDebug() << str; // "String1 String2"
```

- `append()` — добавляет символ или строку в конец исходной строки. Прототипы метода:

```
QString &append(QChar ch)
QString &append(const char *str)
QString &append(const QString &str)
QString &append(const QChar *str, qsizetype len)
QString &append(QLatin1String str)
QString &append(const QByteArray &ba)
```

Пример:

```
QString str = "String";
str.append(L'1');
qDebug() << str; // "String1"
str.append(" String2").append(" String3");
qDebug() << str; // "String1 String2 String3"
```

- `insert()` — вставляет символы или строки в позицию, указанную индексом. Остальные символы сдвигаются к концу строки. Прототипы метода:

```
QString &insert(qsizetype position, QChar ch)
QString &insert(qsizetype position, const char *str)
QString &insert(qsizetype position, const QString &str)
QString &insert(qsizetype position, const QChar *unicode, qsizetype size)
QString &insert(qsizetype position, QStringView str)
QString &insert(qsizetype position, QLatin1String str)
QString &insert(qsizetype position, const QByteArray &str)
```

Пример:

```
QString str = "A";
str.insert(0, L'*');           // В начало строки
```

```
qDebug() << str;                // "*A"
str.insert(str.size(), "+++"); // В конец строки
qDebug() << str;                // "*A+++"
```

- `push_front()` — добавляет символ или строку в начало исходной строки. Прототипы метода:

```
void push_front(QChar ch)
void push_front(const QString &other)
```

Пример:

```
QString str = "A";
str.push_front(L'*');
qDebug() << str;                // "*A"
str.push_front("+++");
qDebug() << str;                // "+++*A"
```

- `prepend()` — добавляет символ или строку в начало исходной строки. Прототипы метода:

```
QString &prepend(QChar ch)
QString &prepend(const char *str)
QString &prepend(const QString &str)
QString &prepend(const QChar *str, qsizetype len)
QString &prepend(QStringView str)
QString &prepend(QLatin1String str)
QString &prepend(const QByteArray &ba)
```

Пример:

```
QString str = "A";
str.prepend(L'*');
qDebug() << str;                // "*A"
str.prepend("+++");
qDebug() << str;                // "+++*A"
```

- `repeated()` — возвращает строку, являющуюся повтором исходной строки `n` раз. Прототип метода:

```
QString repeated(qsizetype n) const
```

Пример:

```
QString str = "ABC";
qDebug() << str.repeated(3); // "ABCABCABC"
```

2.3.9. Удаление символов

Для удаления символов предназначены следующие методы:

- `truncate()` — обрезает строку до указанного количества символов. Если размер строки меньше указанного количества, то ничего не происходит. Прототип метода:

```
void truncate(qsizetype position)
```

Пример:

```
QString str("строка");
str.truncate(4);
qDebug() << str; // "стро"
```

- `clear()` — удаляет все символы. Прототип метода:

```
void clear()
```

Пример:

```
QString str("строка");
str.clear();
qDebug() << str.size(); // 0
```

- `erase()` — удаляет символы внутри диапазона, ограниченного двумя итераторами. Прототип метода:

```
QString::iterator erase(QString::const_iterator first,
                        QString::const_iterator last)
```

Пример:

```
QString str = "ABCDE";
QString::const_iterator first = str.cbegin();
QString::const_iterator last = str.cend();
++first;
--last;
QString::iterator it = str.erase(first, last);
qDebug() << str;               // "AE"
qDebug() << *it;               // 'E'
```

Для удаления символов можно также воспользоваться функцией `erase()`. Прототип функции:

```
qsizetype erase(QString &s, const T &t)
```

Пример:

```
QString str = "nN123n";
qsizetype n = erase(str, QChar(L'n'));
qDebug() << str;  // "N123"
qDebug() << n;    // 2
```

- `erase_if()` — удаляет символы, для которых функция `pred` возвращает `true`. Прототип функции:

```
qsizetype erase_if(QString &s, Predicate pred)
```

Удалим букву `n` вне зависимости от регистра символов:

```
QString str = "nN123n";
qsizetype n = erase_if(str, [](const QChar &ch){
   return (ch == QChar(L'n')) || (ch == QChar(L'N'));
});
```

```
qDebug() << str;  // "123"
qDebug() << n;    // 3
```

- `remove()` — удаляет символы. Прототипы метода:

```
QString &remove(qsizetype position, qsizetype n)
QString &remove(QChar ch,
                Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &remove(const QString &str,
                Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &remove(QLatin1String str,
                Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &remove(const QRegularExpression &re)
```

Пример:

```
QString str = "123456";
str.remove(0, 3);
qDebug() << str;  // "456"
str.remove(L'6');
qDebug() << str;  // "45"
str.remove("5");
qDebug() << str;  // "4"
```

Если в параметре `cs` указана константа `Qt::CaseSensitive`, то поиск зависит от регистра символов. Чтобы поиск не зависел от регистра символов, нужно указать константу `Qt::CaseInsensitive`:

```
QString str = "nN123";
str.remove("n", Qt::CaseSensitive);
qDebug() << str;  // "N123"
str = "nN123";
str.remove("n", Qt::CaseInsensitive);
qDebug() << str;  // "123"
```

- `removeIf()` — удаляет символы, для которых функция `pred` возвращает `true`. Прототип метода:

```
QString &removeIf(Predicate pred)
```

Удалим букву `n` вне зависимости от регистра символов:

```
QString str = "nN123";
str.removeIf([](const QChar &ch){
   return (ch == QChar(L'n')) || (ch == QChar(L'N'));
});
qDebug() << str;  // "123"
```

- `trimmed()` — удаляет пробельные символы в начале и конце строки и возвращает измененную строку. Пробельными символами считаются: пробел, символ перевода строки (`\n`), символ возврата каретки (`\r`), символы горизонтальной (`\t`) и вертикальной (`\v`) табуляции, перевод формата (`\f`). Прототип метода:

```
QString trimmed() const
```

Пример:

```
QString str = "     str\n\r\v\t\f";
qDebug() << str.trimmed();  // "str"
```

- `simplified()` — удаляет пробельные символы в начале и конце строки и возвращает измененную строку. Дополнительно заменяет пробельные символы внутри строки одним пробелом. Пробельными символами считаются: пробел, символ перевода строки (`\n`), символ возврата каретки (`\r`), символы горизонтальной (`\t`) и вертикальной (`\v`) табуляции, перевод формата (`\f`). Прототип метода:

```
QString simplified() const
```

Пример:

```
QString str = "     str \n\v str\n\r\v\t\f";
qDebug() << str.simplified();  // "str str"
```

- `chop()` — удаляет указанное количество символов из конца строки. Прототип метода:

```
void chop(qsizetype n)
```

Пример:

```
QString str = "123456";
str.chop(3);
qDebug() << str;  // "123"
```

- `chopped()` — удаляет указанное количество символов из конца строки и возвращает измененную строку. Прототип метода:

```
QString chopped(qsizetype len) const
```

Пример:

```
QString str = "123456";
QString str2 = str.chopped(3);
qDebug() << str2;  // "123"
```

- `first()` и `left()` — возвращают указанное количество символов из начала строки. Прототипы методов:

```
QString first(qsizetype n) const
QString left(qsizetype n) const
```

Пример:

```
QString str = "123456";
QString str2 = str.first(3);
qDebug() << str2; // "123"
```

При использовании метода `left()`, если количество больше размера строки или меньше нуля, возвращается вся строка:

```
QString str = "123456";
QString str2 = str.left(3);
```

```
qDebug() << str2; // "123"
str2 = str.left(8);
qDebug() << str2; // "123456"
```

- `last()` и `right()` — возвращают указанное количество символов из конца строки. Прототипы методов:

```
QString last(qsizetype n) const
QString right(qsizetype n) const
```

Пример:

```
QString str = "123456";
QString str2 = str.last(3);
qDebug() << str2; // "456"
```

При использовании метода `right()`, если количество больше размера строки или меньше нуля, возвращается вся строка:

```
QString str = "123456";
QString str2 = str.right(3);
qDebug() << str2; // "456"
str2 = str.right(8);
qDebug() << str2; // "123456"
```

2.3.10. Изменение регистра символов

Изменить регистр символов строки позволяют следующие методы:

- `toLower()` — переводит все символы в нижний регистр и возвращает новую строку. Прототип метода:

```
QString toLower() const
```

Пример:

```
QString str1 = "СТРОКА";
QString str2 = str1.toLower();
qDebug() << str2;              // "строка"
```

- `toUpper()` — переводит все символы в верхний регистр и возвращает новую строку. Прототип метода:

```
QString toUpper() const
```

Пример:

```
QString str1 = "строка";
QString str2 = str1.toUpper();
qDebug() << str2;              // "СТРОКА"
```

- `isLower()` — возвращает значение `true`, если все символы в строке в нижнем регистре, и `false` — в противном случае. Прототип метода:

```
bool isLower() const
```

Пример:

```
QString str1 = "строка";
QString str2 = "СТРОКА";
qDebug() << str1.isLower(); // true
qDebug() << str2.isLower(); // false
```

- `isUpper()` — возвращает значение `true`, если все символы в строке в верхнем регистре, и `false` — в противном случае. Прототип метода:

```
bool isUpper() const
```

Пример:

```
QString str1 = "строка";
QString str2 = "СТРОКА";
qDebug() << str1.isUpper(); // false
qDebug() << str2.isUpper(); // true
```

2.3.11. Получение фрагмента строки

Метод `sliced()` возвращает фрагмент строки, состоящий из `n` символов, начиная с индекса `pos`. Если длина не задана, то возвращается фрагмент, начиная с индекса `pos` до конца строки. Прототипы метода:

```
QString sliced(qsizetype pos) const
QString sliced(qsizetype pos, qsizetype n) const
```

Пример:

```
QString str = "строка";
qDebug() << str.sliced(0, 3); // "стр"
qDebug() << str.sliced(3);    // "ока"
```

Можно также воспользоваться аналогичным методом `mid()`. Прототип метода:

```
QString mid(qsizetype position, qsizetype n = -1) const
```

Пример:

```
QString str = "строка";
qDebug() << str.mid(0, 3);    // "стр"
qDebug() << str.mid(3);       // "ока"
```

Если параметр `position` имеет значение больше размера строки, то метод вернет пустую строку. Если параметр `n` имеет значение больше оставшихся символов в строке, то метод вернет оставшиеся символы:

```
QString str = "строка";
qDebug() << str.mid(10, 3); // ""
qDebug() << str.mid(3, 8);  // "ока"
```

2.3.12. Поиск в строке

Перечислим методы, предназначенные для поиска в строке:

- `indexOf()` — производит поиск символа или фрагмента с начала строки или с индекса `from` до конца строки. Возвращает индекс первого совпадения, если фрагмент найден, или значение `-1` — в противном случае. Прототипы метода:

```
qsizetype indexOf(QChar ch, qsizetype from = 0,
          Qt::CaseSensitivity cs = Qt::CaseSensitive) const
qsizetype indexOf(const QString &str, qsizetype from = 0,
          Qt::CaseSensitivity cs = Qt::CaseSensitive) const
qsizetype indexOf(QStringView str, qsizetype from = 0,
          Qt::CaseSensitivity cs = Qt::CaseSensitive) const
qsizetype indexOf(QLatin1String str, qsizetype from = 0,
          Qt::CaseSensitivity cs = Qt::CaseSensitive) const
qsizetype indexOf(const QRegularExpression &re,
          qsizetype from = 0,
          QRegularExpressionMatch *rmatch = nullptr) const
```

  Пример:

```
QString str = "123454321";
qDebug() << str.indexOf(L'2');    // 1
qDebug() << str.indexOf(L'6');    // -1
qDebug() << str.indexOf(L'2', 3); // 7
qDebug() << str.indexOf("2");     // 1
qDebug() << str.indexOf("6");     // -1
qDebug() << str.indexOf("2", 3);  // 7
```

  Если в параметре `cs` указана константа `Qt::CaseSensitive`, то поиск зависит от регистра символов. Чтобы поиск не зависел от регистра символов, нужно указать константу `Qt::CaseInsensitive`:

```
QString str = "строка";
qDebug() << str.indexOf(L'т', 0, Qt::CaseSensitive);   // 1
qDebug() << str.indexOf(L'Т', 0, Qt::CaseSensitive);   // -1
qDebug() << str.indexOf(L'Т', 0, Qt::CaseInsensitive); // 1
```

- `lastIndexOf()` — производит поиск символа или фрагмента с конца строки или с индекса `from` до начала строки. Возвращает индекс первого совпадения, если фрагмент найден, или значение `-1` — в противном случае. Прототипы метода:

```
qsizetype lastIndexOf(QChar ch, qsizetype from = -1,
          Qt::CaseSensitivity cs = Qt::CaseSensitive) const
qsizetype lastIndexOf(const QString &str, qsizetype from = -1,
          Qt::CaseSensitivity cs = Qt::CaseSensitive) const
qsizetype lastIndexOf(QStringView str, qsizetype from = -1,
          Qt::CaseSensitivity cs = Qt::CaseSensitive) const
qsizetype lastIndexOf(QLatin1String str, qsizetype from = -1,
          Qt::CaseSensitivity cs = Qt::CaseSensitive) const
```

```
qsizetype lastIndexOf(const QRegularExpression &re,
        qsizetype from = -1,
        QRegularExpressionMatch *rmatch = nullptr) const
```

Пример:

```
QString str = "123454321";
qDebug() << str.lastIndexOf(L'2');    // 7
qDebug() << str.lastIndexOf(L'6');    // -1
qDebug() << str.lastIndexOf(L'2', 3); // 1
qDebug() << str.lastIndexOf("2");     // 7
qDebug() << str.lastIndexOf("6");     // -1
qDebug() << str.lastIndexOf("2", 3);  // 1
```

Если в параметре `cs` указана константа `Qt::CaseSensitive`, то поиск зависит от регистра символов. Чтобы поиск не зависел от регистра символов, нужно указать константу `Qt::CaseInsensitive`:

```
QString str = "строка";
qDebug() << str.lastIndexOf(L'т', -1, Qt::CaseSensitive);   // 1
qDebug() << str.lastIndexOf(L'Т', -1, Qt::CaseSensitive);   // -1
qDebug() << str.lastIndexOf(L'Т', -1, Qt::CaseInsensitive); // 1
```

- `contains()` — проверяет, содержит ли строка указанную подстроку. Если содержит, то возвращается значение `true`, в противном случае — `false`. Прототипы метода:

```
bool contains(QChar ch,
              Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool contains(const QString &str,
              Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool contains(QStringView str,
              Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool contains(QLatin1String str,
              Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool contains(const QRegularExpression &re,
              QRegularExpressionMatch *rmatch = nullptr) const
```

Пример:

```
QString str = "123454321";
qDebug() << str.contains(L'2');    // true
qDebug() << str.contains(L'6');    // false
qDebug() << str.contains("2");     // true
qDebug() << str.contains("6");     // false
```

Если в параметре `cs` указана константа `Qt::CaseSensitive`, то поиск зависит от регистра символов. Чтобы поиск не зависел от регистра символов, нужно указать константу `Qt::CaseInsensitive`:

```
QString str = "строка";
qDebug() << str.contains(L'т', Qt::CaseSensitive);   // true
```

```
qDebug() << str.contains(L'T', Qt::CaseSensitive);   // false
qDebug() << str.contains(L'T', Qt::CaseInsensitive); // true
```

- `startsWith()` — проверяет, начинается ли строка с указанной подстроки. Если начинается, то возвращается значение `true`, в противном случае — `false`. Прототипы метода:

```
bool startsWith(QChar c,
                Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool startsWith(const QString &s,
                Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool startsWith(QStringView str,
                Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool startsWith(QLatin1String s,
                Qt::CaseSensitivity cs = Qt::CaseSensitive) const
```

Пример:

```
QString str = "строка";
qDebug() << str.startsWith(L'c');   // true
qDebug() << str.startsWith(L'C');   // false
qDebug() << str.startsWith("стр");  // true
qDebug() << str.startsWith("Стр");  // false
```

Если в параметре `cs` указана константа `Qt::CaseSensitive`, то поиск зависит от регистра символов. Чтобы поиск не зависел от регистра символов, нужно указать константу `Qt::CaseInsensitive`:

```
QString str = "строка";
qDebug() << str.startsWith(L'c', Qt::CaseSensitive);   // true
qDebug() << str.startsWith(L'C', Qt::CaseSensitive);   // false
qDebug() << str.startsWith(L'C', Qt::CaseInsensitive); // true
```

- `endsWith()` — проверяет, заканчивается ли строка указанной подстрокой. Если заканчивается, то возвращается значение `true`, в противном случае — `false`. Прототипы метода:

```
bool endsWith(QChar c,
              Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool endsWith(const QString &s,
              Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool endsWith(QStringView str,
              Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool endsWith(QLatin1String s,
              Qt::CaseSensitivity cs = Qt::CaseSensitive) const
```

Пример:

```
QString str = "строка";
qDebug() << str.endsWith(L'a');   // true
qDebug() << str.endsWith(L'A');   // false
qDebug() << str.endsWith("ока");  // true
qDebug() << str.endsWith("Ока");  // false
```

Если в параметре cs указана константа Qt::CaseSensitive, то поиск зависит от регистра символов. Чтобы поиск не зависел от регистра символов, нужно указать константу Qt::CaseInsensitive:

```
QString str = "строка";
qDebug() << str.endsWith(L'a', Qt::CaseSensitive);   // true
qDebug() << str.endsWith(L'A', Qt::CaseSensitive);   // false
qDebug() << str.endsWith(L'A', Qt::CaseInsensitive); // true
```

- count() — возвращает число вхождений подстроки в строку. Если подстрока в строку не входит, то возвращается значение 0. Прототипы метода:

```
qsizetype count(QChar ch,
                Qt::CaseSensitivity cs = Qt::CaseSensitive) const
qsizetype count(const QString &str,
                Qt::CaseSensitivity cs = Qt::CaseSensitive) const
qsizetype count(QStringView str,
                Qt::CaseSensitivity cs = Qt::CaseSensitive) const
qsizetype count(const QRegularExpression &re) const
```

Пример:

```
QString str = "пример пример";
qDebug() << str.count(L'п');   // 2
qDebug() << str.count(L'т');   // 0
qDebug() << str.count("при");  // 2
qDebug() << str.count("При");  // 0
```

Если в параметре cs указана константа Qt::CaseSensitive, то поиск зависит от регистра символов. Чтобы поиск не зависел от регистра символов, нужно указать константу Qt::CaseInsensitive:

```
QString str = "пример пример";
qDebug() << str.count(L'П', Qt::CaseSensitive);   // 0
qDebug() << str.count(L'П', Qt::CaseInsensitive); // 2
```

2.3.13. Замена в строке

Произвести замену в строке позволяют следующие методы:

- fill() — заменяет все символы в строке указанным символом. Во втором параметре можно передать новый размер строки. Прототип метода:

```
QString &fill(QChar ch, qsizetype size = -1)
```

Пример:

```
QString str = "ABC";
str.fill(L'*');
qDebug() << str;               // "***"
str.fill(L'+', 2);
qDebug() << str;               // "++"
```

- `swap()` — меняет содержимое двух строк местами. Прототип метода:

```
void swap(QString &other)
```

Пример:

```
QString str1("12345"), str2("67890");
str1.swap(str2);
qDebug() << str1;   // "67890"
qDebug() << str2;   // "12345"
```

- `replace()` — заменяет фрагмент строки отдельным символом или подстрокой. Если вставляемая подстрока меньше фрагмента, то остальные символы сдвигаются к началу строки, а если больше, то раздвигаются таким образом, чтобы вместить всю вставляемую подстроку. Прототипы метода:

```
QString &replace(qsizetype position, qsizetype n, QChar after)
QString &replace(qsizetype position, qsizetype n,
                 const QString &after)
QString &replace(qsizetype position, qsizetype n,
                 const QChar *unicode, qsizetype size)
QString &replace(QChar before, QChar after,
                 Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &replace(QChar ch, const QString &after,
                 Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &replace(QChar ch, QLatin1String after,
                 Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &replace(const QChar *before, qsizetype blen,
                 const QChar *after, qsizetype alen,
                 Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &replace(const QString &before, const QString &after,
                 Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &replace(const QString &before, QLatin1String after,
                 Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &replace(QLatin1String before, QLatin1String after,
                 Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &replace(QLatin1String before, const QString &after,
                 Qt::CaseSensitivity cs = Qt::CaseSensitive)
QString &replace(const QRegularExpression &re,
                 const QString &after)
```

Пример:

```
QString str = "12345";
str.replace(0, 3, L'*');
qDebug() << str;   // "*45"
str = "12345";
str.replace(0, 3, "++++");
qDebug() << str;   // "++++45"
str = "12345";
str.replace(L'2', L'8');
```

```
qDebug() << str;   // "18345"
str = "12345";
str.replace("234", "+++");
qDebug() << str;   // "1+++5"
str = "12 34 532";
// #include <QRegularExpression>
str.replace(QRegularExpression("[0-9]+"), "+");
qDebug() << str;   // "+ + +"
```

Если в параметре `cs` указана константа `Qt::CaseSensitive`, то поиск зависит от регистра символов. Чтобы поиск не зависел от регистра символов, нужно указать константу `Qt::CaseInsensitive`:

```
QString str = "тест";
str.replace("Е", "о", Qt::CaseSensitive);
qDebug() << str;  // "тест"
str.replace("Е", "о", Qt::CaseInsensitive);
qDebug() << str;  // "тост"
```

- `toHtmlEscaped()` — заменяет специальные символы из HTML соответствующими HTML-эквивалентами и возвращает измененную строку. Прототип метода:

```
QString toHtmlEscaped() const
```

Пример:

```
QString str = "<b>\"&";
QString str2 = str.toHtmlEscaped();
qDebug() << str2; // "&lt;b&gt;&quot;&amp;"
```

2.3.14. Сравнение строк

Над двумя объектами или объектом класса `QString` и С-строкой определены операции ==, !=, <, >, <= и >=:

```
QString str1 = "abc", str2 = "abcd";
if (str1 == str2) {
   qDebug() << "str1 == str2";
}
else if (str1 < str2) {
   qDebug() << "str1 < str2";
}
else if (str1 > str2) {
   qDebug() << "str1 > str2";
} // Результат: str1 < str2
```

Обратите внимание на то, что сравнивать с помощью этих операторов С-строки нельзя, т. к. будут сравниваться адреса, а не символы в строках. Для сравнения С-строк необходимо использовать специальные функции, например `strcmp()`. Для корректной работы операторов сравнения необходимо, чтобы справа или слева от оператора находился объект класса `QString`. Пример:

```
char a[] = "abc", b[] = "abc";
QString c = "abc";
qDebug() << (a == b); // Сравниваются адреса!!!
qDebug() << (a == c); // OK (равны)
qDebug() << (c == b); // OK (равны)
```

Для сравнения объекта класса `QString` с другим объектом класса `QString` можно использовать метод `compare()`. В качестве значения метод возвращает:

- `отрицательное число` — если объект класса `QString` меньше строки, переданной в качестве параметра;
- `0` — если строки равны;
- `положительное число` — если объект класса `QString` больше строки, переданной в качестве параметра.

Прототипы метода:

```
int compare(QChar ch, Qt::CaseSensitivity cs = Qt::CaseSensitive) const
int compare(const QString &other,
            Qt::CaseSensitivity cs = Qt::CaseSensitive) const
int compare(QStringView s,
            Qt::CaseSensitivity cs = Qt::CaseSensitive) const
int compare(QLatin1String other,
            Qt::CaseSensitivity cs = Qt::CaseSensitive) const
```

Пример:

```
QString str1 = "abc", str2 = "abd";
qDebug() << str1.compare("abd");      // -1
qDebug() << str2.compare("abd");      // 0
qDebug() << str1.compare("abb");      // 1
```

Если в параметре `cs` указана константа `Qt::CaseSensitive`, то сравнение зависит от регистра символов. Чтобы сравнение не зависело от регистра символов, нужно указать константу `Qt::CaseInsensitive`:

```
QString str1 = "abc", str2 = "ABC";
qDebug() << str1.compare(str2, Qt::CaseSensitive);   // 32
qDebug() << str1.compare(str2, Qt::CaseInsensitive); // 0
```

Можно также воспользоваться статическим методом `compare()`. Прототипы метода:

```
static int compare(const QString &s1, const QString &s2,
                   Qt::CaseSensitivity cs = Qt::CaseSensitive)
static int compare(const QString &s1, QLatin1String s2,
                   Qt::CaseSensitivity cs = Qt::CaseSensitive)
static int compare(QLatin1String s1, const QString &s2,
                   Qt::CaseSensitivity cs = Qt::CaseSensitive)
static int compare(const QString &s1, QStringView s2,
                   Qt::CaseSensitivity cs = Qt::CaseSensitive)
static int compare(QStringView s1, const QString &s2,
                   Qt::CaseSensitivity cs = Qt::CaseSensitive)
```

Пример:

```
QString str1 = "abc", str2 = "abd";
qDebug() << QString::compare(str1, "abd");      // -1
qDebug() << QString::compare(str2, "abd");      // 0
qDebug() << QString::compare(str1, "abb");      // 1
```

Если в параметре `cs` указана константа `Qt::CaseSensitive`, то сравнение зависит от регистра символов. Чтобы сравнение не зависело от регистра символов, нужно указать константу `Qt::CaseInsensitive`:

```
QString str1 = "abc", str2 = "ABC";
qDebug() << QString::compare(str1, str2, Qt::CaseSensitive);   // 32
qDebug() << QString::compare(str1, str2, Qt::CaseInsensitive); // 0
```

Можно также воспользоваться статическим методом `localeAwareCompare()`, который учитывает настройки локали. Прототипы метода:

```
static int localeAwareCompare(const QString &s1, const QString &s2)
static int localeAwareCompare(QStringView s1, QStringView s2)
```

Пример:

```
QString str1 = "abc", str2 = "abd";
qDebug() << QString::localeAwareCompare(str1, "abd"); // -1
qDebug() << QString::localeAwareCompare(str2, "abd"); // 0
qDebug() << QString::localeAwareCompare(str1, "abb"); // 1
```

2.3.15. Преобразование строки в число

Для преобразования строки в число используются следующие методы:

```
short toShort(bool *ok = nullptr, int base = 10) const
ushort toUShort(bool *ok = nullptr, int base = 10) const
int toInt(bool *ok = nullptr, int base = 10) const
uint toUInt(bool *ok = nullptr, int base = 10) const
long toLong(bool *ok = nullptr, int base = 10) const
ulong toULong(bool *ok = nullptr, int base = 10) const
qlonglong toLongLong(bool *ok = nullptr, int base = 10) const
qulonglong toULongLong(bool *ok = nullptr, int base = 10) const
float toFloat(bool *ok = nullptr) const
double toDouble(bool *ok = nullptr) const
```

Пробельные символы в начале строки игнорируются. Результат выполнения операции доступен через первый параметр (значение `true`, если операция выполнена успешно). Метод возвращает преобразованное число или значение `0`, если преобразовать не удалось. Пример преобразования строки в целое число:

```
QString str = " \n\r25";
bool ok;
int n = str.toInt(&ok);
qDebug() << n;  // 25
qDebug() << ok; // true
```

```
str = "ff";
n = str.toInt(&ok);
qDebug() << n;  // 0
qDebug() << ok; // false
```

В параметре `base` можно указать систему счисления или значение `0`, при котором система счисления определяется автоматически:

```
QString str = " \n\r25";
bool ok;
int n = str.toInt(&ok, 10);
qDebug() << n;  // 25
qDebug() << ok; // true
str = "ff";
n = str.toInt(&ok, 16);
qDebug() << n;  // 255
qDebug() << ok; // true
```

При преобразовании всегда используется локаль `C`. Если нужно учитывать настройки произвольной локали, то следует воспользоваться следующими методами из класса `QLocale`:

```
#include <QLocale>
short toShort(const QString &s, bool *ok = nullptr) const
short toShort(QStringView s, bool *ok = nullptr) const
ushort toUShort(const QString &s, bool *ok = nullptr) const
ushort toUShort(QStringView s, bool *ok = nullptr) const
int toInt(const QString &s, bool *ok = nullptr) const
int toInt(QStringView s, bool *ok = nullptr) const
uint toUInt(const QString &s, bool *ok = nullptr) const
uint toUInt(QStringView s, bool *ok = nullptr) const
long toLong(const QString &s, bool *ok = nullptr) const
long toLong(QStringView s, bool *ok = nullptr) const
ulong toULong(const QString &s, bool *ok = nullptr) const
ulong toULong(QStringView s, bool *ok = nullptr) const
qlonglong toLongLong(const QString &s, bool *ok = nullptr) const
qlonglong toLongLong(QStringView s, bool *ok = nullptr) const
qulonglong toULongLong(const QString &s, bool *ok = nullptr) const
qulonglong toULongLong(QStringView s, bool *ok = nullptr) const
float toFloat(const QString &s, bool *ok = nullptr) const
float toFloat(QStringView s, bool *ok = nullptr) const
double toDouble(const QString &s, bool *ok = nullptr) const
double toDouble(QStringView s, bool *ok = nullptr) const
```

Пример:

```
QString str = "25.56";
bool ok;
double n = str.toDouble(&ok);
qDebug() << n;  // 25.56
qDebug() << ok; // true
```

```
str = "25,56";
n = str.toDouble(&ok);
qDebug() << n;  // 0
qDebug() << ok; // false
QLocale ru(QLocale::Russian);
n = ru.toDouble(str, &ok);
qDebug() << n;  // 25.56
qDebug() << ok; // true
```

2.3.16. Преобразование числа в строку

Преобразовать число в строку позволяют следующие методы:

```
QString &setNum(short n, int base = 10)
QString &setNum(ushort n, int base = 10)
QString &setNum(int n, int base = 10)
QString &setNum(uint n, int base = 10)
QString &setNum(long n, int base = 10)
QString &setNum(ulong n, int base = 10)
QString &setNum(qlonglong n, int base = 10)
QString &setNum(qulonglong n, int base = 10)
QString &setNum(float n, char format = 'g', int precision = 6)
QString &setNum(double n, char format = 'g', int precision = 6)
```

А также статические методы:

```
static QString number(int n, int base = 10)
static QString number(uint n, int base = 10)
static QString number(long n, int base = 10)
static QString number(ulong n, int base = 10)
static QString number(qlonglong n, int base = 10)
static QString number(qulonglong n, int base = 10)
static QString number(double n, char format = 'g', int precision = 6)
```

Пример:

```
QString str;
str.setNum(25);
qDebug() << str;  // "25"
str.setNum(25.56);
qDebug() << str;  // "25.56"
str = QString::number(25);
qDebug() << str;  // "25"
str = QString::number(25.56);
qDebug() << str;  // "25.56"
```

В параметре `base` можно указать систему счисления:

```
QString str;
str.setNum(255, 16);
qDebug() << str;  // "ff"
```

```
str = QString::number(255, 16);
qDebug() << str;  // "ff"
```

При преобразовании всегда используется локаль C. Если нужно учитывать настройки произвольной локали, то следует воспользоваться методом `toString()` из класса `QLocale`:

```
#include <QLocale>
QString toString(short i) const
QString toString(ushort i) const
QString toString(int i) const
QString toString(uint i) const
QString toString(long i) const
QString toString(ulong i) const
QString toString(qlonglong i) const
QString toString(qulonglong i) const
QString toString(float i, char f = 'g', int prec = 6) const
QString toString(double i, char f = 'g', int prec = 6) const
```

Пример:

```
QLocale ru(QLocale::Russian);
QString str = ru.toString(25.56);
qDebug() << str;  // "25,56"
```

2.3.17. Форматирование строки

Выполнить подстановку значения в строку позволяет метод `arg()`. Прототипы метода:

```
QString arg(short a, int fieldWidth = 0, int base = 10,
            QChar fillChar = QLatin1Char(' ')) const
QString arg(ushort a, int fieldWidth = 0, int base = 10,
            QChar fillChar = QLatin1Char(' ')) const
QString arg(int a, int fieldWidth = 0, int base = 10,
            QChar fillChar = QLatin1Char(' ')) const
QString arg(uint a, int fieldWidth = 0, int base = 10,
            QChar fillChar = QLatin1Char(' ')) const
QString arg(long a, int fieldWidth = 0, int base = 10,
            QChar fillChar = QLatin1Char(' ')) const
QString arg(ulong a, int fieldWidth = 0, int base = 10,
            QChar fillChar = QLatin1Char(' ')) const
QString arg(qlonglong a, int fieldWidth = 0, int base = 10,
            QChar fillChar = QLatin1Char(' ')) const
QString arg(qulonglong a, int fieldWidth = 0, int base = 10,
            QChar fillChar = QLatin1Char(' ')) const
QString arg(double a, int fieldWidth = 0, char format = 'g',
            int precision = -1, QChar fillChar = QLatin1Char(' ')) const
QString arg(char a, int fieldWidth = 0,
            QChar fillChar = QLatin1Char(' ')) const
```

```
QString arg(QChar a, int fieldWidth = 0,
            QChar fillChar = QLatin1Char(' ')) const
QString arg(const QString &a, int fieldWidth = 0,
            QChar fillChar = QLatin1Char(' ')) const
QString arg(QStringView a, int fieldWidth = 0,
            QChar fillChar = QLatin1Char(' ')) const
QString arg(QLatin1String a, int fieldWidth = 0,
            QChar fillChar = QLatin1Char(' ')) const
QString arg(Args &&... args) const
```

Место вставки в строке помечается комбинацией `%N`, где *N* — число от `1` до `99`:

```
QString str = QString("%1").arg(25.56);
qDebug() << str;  // "25.56"
QString str2 = QString("%1 %2").arg(2).arg("июня");
qDebug() << str2;  // "2 июня"
```

При преобразовании всегда используется локаль `C`. Если нужно учитывать настройки произвольной локали, то следует перед цифрой указать букву `L`:

```
// Настройка локали
QLocale::setDefault(QLocale(QLocale::Russian));
QString str = QString("%1 %L2").arg(25.56).arg(25.56);
qDebug() << str;  // "25.56 25,56"
```

В параметре `fieldWidth` можно указать ширину поля. Если число положительное, то выравнивание производится по правой стороне поля, а если отрицательное — по левой стороне поля. Пустое пространство заполняется символом `fillChar`:

```
QString str = QString("%1").arg(25, 7, 10, QChar(L'*'));
qDebug() << str;  // "*****25"
QString str2 = QString("%1").arg(25, -7, 10, QChar(L'*'));
qDebug() << str2;  // "25*****"
```

Для форматирования строки можно также воспользоваться следующими статическими методами:

```
static QString asprintf(const char *cformat, ...)
static QString vasprintf(const char *cformat, va_list ap)
```

Пример:

```
QString str = QString::asprintf("%d %.2f", 2, 2.5);
qDebug() << str; // "2 2.50"
```

2.3.18. Разделение строки на подстроки по разделителю

Метод `split()` разбивает строку по разделителю или шаблону и возвращает список подстрок. Прототипы метода:

```
QStringList split(QChar sep,
                  Qt::SplitBehavior behavior = Qt::KeepEmptyParts,
                  Qt::CaseSensitivity cs = Qt::CaseSensitive) const
```

```
QStringList split(const QString &sep,
                  Qt::SplitBehavior behavior = Qt::KeepEmptyParts,
                  Qt::CaseSensitivity cs = Qt::CaseSensitive) const
QStringList split(const QRegularExpression &re,
                  Qt::SplitBehavior behavior = Qt::KeepEmptyParts) const
```

Если в параметре `behavior` указано значение `Qt::SkipEmptyParts`, то пустые подстроки не будут добавляться в список. Пример:

```
// #include <QStringList>
QString str = "12,,34,56";
QStringList list = str.split(L',', Qt::KeepEmptyParts);
qDebug() << list;   // QList("12", "", "34", "56")
QStringList list2 = str.split(L',', Qt::SkipEmptyParts);
qDebug() << list2;  // QList("12", "34", "56")
```

Если в параметре `cs` указана константа `Qt::CaseSensitive`, то сравнение зависит от регистра символов. Чтобы сравнение не зависело от регистра символов, нужно указать константу `Qt::CaseInsensitive`:

```
// #include <QStringList>
QString str = "12aA34a56";
QStringList list = str.split("a", Qt::SkipEmptyParts,
                             Qt::CaseSensitive);
qDebug() << list;   // QList("12", "A34", "56")
QStringList list2 = str.split("a", Qt::SkipEmptyParts,
                              Qt::CaseInsensitive);
qDebug() << list2;  // QList("12", "34", "56")
```

В первом параметре можно указать шаблон регулярного выражения:

```
// #include <QStringList>
// #include <QRegularExpression>
QString str = "word1, word2\nword3\r\nword4.word5";
QStringList list = str.split(QRegularExpression("[\\s,.]+"),
                             Qt::SkipEmptyParts);
qDebug() << list;
// QList("word1", "word2", "word3", "word4", "word5")
```

Пример добавления всех букв из строки в список:

```
QString str = "word";
QStringList list = str.split(QString(), Qt::SkipEmptyParts);
qDebug() << list; // QList("w", "o", "r", "d")
```

Выполнить обратную операцию позволяет метод `join()` из класса `QStringList` *(см. разд. 2.4.13)*:

```
// #include <QStringList>
QStringList list = { "A", "B", "C", "D" };
QString str = list.join(L',');
qDebug() << str;   // "A,B,C,D"
```

Метод `section()` разделяет строку на подстроки по разделителю и возвращает строку, содержащую подстроки от индекса `start` до индекса `end`. Прототипы метода:

```
QString section(QChar sep, qsizetype start, qsizetype end = -1,
                QString::SectionFlags flags = SectionDefault) const
QString section(const QString &sep, qsizetype start, qsizetype end = -1,
                QString::SectionFlags flags = SectionDefault) const
QString section(const QRegularExpression &re, qsizetype start,
                qsizetype end = -1,
                QString::SectionFlags flags = SectionDefault) const
```

Пример:

```
QString str = "1,2,3,4,5,6";
QString str2 = str.section(L',', 2, 2);
qDebug() << str2; // "3"
str2 = str.section(L',', 2, 3);
qDebug() << str2; // "3,4"
str2 = str.section(L',', 2, 4);
qDebug() << str2; // "3,4,5"
```

2.4. Класс *QStringList:* список строк

Класс `QStringList` описывает список строк (специализация `QList<QString>`). Подключение заголовочного файла:

```
#include <QStringList>
```

2.4.1. Создание объекта

Класс `QStringList` содержит следующие конструкторы:

```
QStringList(const QString &str)
QStringList(const QList<QString> &other)
QStringList(QList<QString> &&other)
```

Пример:

```
QStringList list(QString("A"));
qDebug() << list; // QList("A")
QStringList list2(list);
qDebug() << list2; // QList("A")
```

Класс `QStringList` наследует такие конструкторы из класса `QList`:

```
QList()
QList(qsizetype size)
QList(qsizetype size, QList::parameter_type value)
QList(std::initializer_list<T> args)
QList(InputIterator first, InputIterator last)
```

Пример:

```
QStringList list;
list << "A" << "B" << "C";
qDebug() << list; // QList("A", "B", "C")
QStringList list2 = {QString("A"), QString("B")};
qDebug() << list2; // QList("A", "B")
QStringList list3(list.begin(), list.end());
qDebug() << list3; // QList("A", "B", "C")
```

Над двумя объектами определены операции ==, !=, <, <=, > и >=. Пример сравнения двух объектов:

```
QStringList list;
list << "A" << "B" << "C";
QStringList list2(list.begin(), list.end());
if (list == list2) {
   qDebug() << "list == list2";
}
```

Кроме того, один объект можно присвоить другому объекту. В этом случае выполняется поэлементное копирование (оператор копирования) или перемещение элементов (оператор перемещения). Пример:

```
QStringList list, list2;
list << "A" << "B" << "C";
// Создание копии
list2 = list;
list2.front() = "D";
qDebug() << list;     // QList("A", "B", "C")
qDebug() << list2;    // QList("D", "B", "C")
```

Доступно также присваивание элементов из списка инициализации:

```
QStringList list;
list = {"A", "B", "C"};
qDebug() << list;     // QList("A", "B", "C")
```

2.4.2. Вставка элементов

Вставить элементы в конец списка можно с помощью оператора <<:

```
QStringList list, list2;
list << "A" << "B";
list << "C";
qDebug() << list;     // QList("A", "B", "C")
list2 << list;
qDebug() << list2;    // QList("A", "B", "C")
```

Добавить элементы в конец списка позволяет также оператор +=:

```
QStringList list, list2;
list += "A";
list += "B";
```

```
list2 += "C";
list += list2;
qDebug() << list;    // QList("A", "B", "C")
```

Для объединения двух списков можно воспользоваться оператором +:

```
QStringList list, list2, list3;
list << "A" << "B";
list2 << "C" << "D";
list3 = list + list2;
qDebug() << list3;    // QList("A", "B", "C", "D")
```

Вставить элементы позволяют также следующие методы:

- `push_front()` — добавляет элемент в начало списка. Прототипы метода:

```
void push_front(QList::parameter_type value)
void push_front(QList::rvalue_ref value)
```

- `push_back()` — добавляет элемент в конец списка. Прототипы метода:

```
void push_back(QList::parameter_type value)
void push_back(QList::rvalue_ref value)
```

  Пример:

```
QStringList list = { "B", "C" };
list.push_front("A");  // В начало
list.push_back("D");   // В конец
qDebug() << list;      // QList("A", "B", "C", "D")
```

- `prepend()` — добавляет элемент в начало списка. Прототипы метода:

```
void prepend(QList::parameter_type value)
void prepend(QList::rvalue_ref value)
```

- `append()` — добавляет элемент в конец списка. Прототипы метода:

```
void append(QList::parameter_type value)
void append(QList::rvalue_ref value)
void append(const QList<T> &value)
void append(QList<T> &&value)
```

  Пример:

```
QStringList list = { "B" };
list.prepend("A");     // В начало
list.append("C");      // В конец
qDebug() << list;      // QList("A", "B", "C")
QStringList list2 = { "D", "E" };
list.append(list2);    // В конец
qDebug() << list;      // QList("A", "B", "C", "D", "E")
```

- `insert()` — вставляет элемент перед указанной позицией. Прототипы метода:

```
QList::iterator insert(qsizetype i, QList::parameter_type value)
QList::iterator insert(qsizetype i, qsizetype count,
                       QList::parameter_type value)
```

```
QList::iterator insert(qsizetype i, QList::rvalue_ref value)
QList::iterator insert(QList::const_iterator before,
                       QList::parameter_type value)
QList::iterator insert(QList::const_iterator before,
       qsizetype count, QList::parameter_type value)
QList::iterator insert(QList::const_iterator before,
                       QList::rvalue_ref value)
```

Пример:

```
QStringList list = { "B" };
list.insert(0, "A");               // В начало
list.insert(list.size(), "C");     // В конец
qDebug() << list;      // QList("A", "B", "C")
QStringList list2 = { "B" };
list2.insert(list2.cbegin(), "A"); // В начало
list2.insert(list2.cend(), "C");   // В конец
qDebug() << list2;     // QList("A", "B", "C")
```

- `emplace()` — вставляет элемент в заданную первым параметром позицию. Значения, указанные через запятую, передаются конструктору объекта. Прототипы метода:

```
QList::iterator emplace(qsizetype i, Args &&... args)
QList::iterator emplace(QList::const_iterator before,
                        Args &&... args)
```

Пример:

```
QStringList list = { "B" };
list.emplace(0, "A");  // В начало
qDebug() << list;      // QList("A", "B")
```

- `emplace_back()` и `emplaceBack()` — создают объект, передавая конструктору указанные через запятую значения, а затем добавляют объект в конец списка. Прототипы методов:

```
QList::reference emplace_back(Args &&... args)
QList::reference emplaceBack(Args &&... args)
```

Пример:

```
QStringList list = { "A" };
list.emplace_back("B");
list.emplaceBack("C");
qDebug() << list;      // QList("A", "B", "C")
```

- `swap()` — меняет элементы двух списков местами:

```
QStringList list = { "A" }, list2 = { "B" };
list.swap(list2);
qDebug() << list;      // QList("B")
qDebug() << list2;     // QList("A")
```

- `swapItemsAt()` — меняет два элемента местами. Прототип метода:

```
void swapItemsAt(qsizetype i, qsizetype j)
```

Пример:

```
QStringList list = { "A", "B" };
list.swapItemsAt(0, 1);
qDebug() << list;      // QList("B", "A")
```

2.4.3. Определение и изменение количества элементов

Для определения и изменения количества элементов списка предназначены следующие методы:

- `size()`, `length()` и `count()` — возвращают количество элементов списка. Прототипы методов:

```
qsizetype size() const
qsizetype length() const
qsizetype count() const
```

Пример:

```
QStringList list = { "A", "B", "C" };
qDebug() << list.size();    // 3
qDebug() << list.length();  // 3
qDebug() << list.count();   // 3
```

- `empty()` и `isEmpty()` — возвращают значение `true`, если список не содержит элементов, и `false` — в противном случае. Прототипы методов:

```
bool empty() const
bool isEmpty() const
```

Пример:

```
QStringList list;
qDebug() << list.empty();    // true
qDebug() << list.isEmpty();  // true
list = { "A", "B", "C" };
qDebug() << list.empty();    // false
qDebug() << list.isEmpty();  // false
```

- `resize()` — задает количество элементов, равное числу `size`. Если указанное количество элементов меньше текущего количества, то лишние элементы будут удалены, в противном случае в конец добавляются пустые строки. Прототип метода:

```
void resize(qsizetype size)
```

Пример:

```
QStringList list = { "A", "B", "C" };
list.resize(2);
```

```
qDebug() << list;    // QList("A", "B")
list.resize(4);
qDebug() << list;    // QList("A", "B", "", "")
```

- `capacity()` — возвращает количество элементов, которое может содержать список без перераспределения памяти. Прототип метода:

```
qsizetype capacity() const
```

- `reserve()` — позволяет задать минимальное количество элементов, которое может содержать список без перераспределения памяти. Прототип метода:

```
void reserve(qsizetype size)
```

Пример указания минимального размера списка:

```
QStringList list;
list.reserve(50);
qDebug() << list.size();      // 0
qDebug() << list.capacity();  // 50
```

- `shrink_to_fit()` и `squeeze()` — уменьшают размер списка до минимального значения. Прототипы методов:

```
void shrink_to_fit()
void squeeze()
```

Пример:

```
QStringList list;
list.reserve(50);
qDebug() << list.capacity();  // 50
list.squeeze();
qDebug() << list.capacity();  // 0
```

2.4.4. Удаление элементов

Для удаления элементов предназначены следующие методы:

- `pop_front()` и `removeFirst()` — удаляют первый элемент. Прототипы методов:

```
void pop_front()
void removeFirst()
```

Пример:

```
QStringList list = { "A", "B", "C" };
list.pop_front();
qDebug() << list;  // QList("B", "C")
list.removeFirst();
qDebug() << list;  // QList("C")
```

- `pop_back()` и `removeLast()` — удаляют последний элемент. Прототипы методов:

```
void pop_back()
void removeLast()
```

Пример:

```
QStringList list = { "A", "B", "C" };
list.pop_back();
qDebug() << list;  // QList("A", "B")
list.removeLast();
qDebug() << list;  // QList("A")
```

- `erase()` — удаляет один элемент или элементы из диапазона. Прототипы метода:

```
QList::iterator erase(QList::const_iterator pos)
QList::iterator erase(QList::const_iterator begin,
                      QList::const_iterator end)
```

Пример:

```
QStringList list = { "A", "B", "C", "D" };
list.erase(list.cbegin());
qDebug() << list;  // QList("B", "C", "D")
list.erase(list.cbegin(), list.cend());
qDebug() << list;  // QList()
```

Для удаления элементов можно также воспользоваться функцией `erase()`. Прототип функции:

```
qsizetype erase(QList<T> &list, const AT &t)
```

- `erase_if()` — удаляет элементы, для которых функция `pred` возвращает `true`. Прототип функции:

```
qsizetype erase_if(QList<T> &list, Predicate pred)
```

Пример:

```
QStringList list = { "A", "B", "C", "B" };
qsizetype n = erase_if(list, [](const QString &s){
   return s == QString("B");
});
qDebug() << list;  // QList("A", "C")
qDebug() << n;     // 2
```

- `remove()` — удаляет `n` элементов, начиная с индекса `i`. Прототип метода:

```
void remove(qsizetype i, qsizetype n = 1)
```

Пример:

```
QStringList list = { "A", "B", "C", "D" };
list.remove(0, 3);
qDebug() << list;  // QList("D")
list = { "A", "B", "C", "D" };
list.remove(1);
qDebug() << list;  // QList("A", "C", "D")
```

- `removeAt()` — удаляет элемент, расположенный по индексу `i`. Прототип метода:

```
void removeAt(qsizetype i)
```

Пример:

```
QStringList list = { "A", "B", "C", "D" };
list.removeAt(0);
qDebug() << list;  // QList("B", "C", "D")
```

- `removeOne()` — удаляет первый элемент с указанным значением. Прототип метода:

```
bool removeOne(const AT &t)
```

Пример:

```
QStringList list = { "A", "B", "C", "B" };
qDebug() << list.removeOne("B"); // true
qDebug() << list;                // QList("A", "C", "B")
```

- `removeAll()` — удаляет все элементы с указанным значением и возвращает количество удаленных элементов. Прототип метода:

```
qsizetype removeAll(const AT &t)
```

Пример:

```
QStringList list = { "A", "B", "C", "B" };
qDebug() << list.removeAll("B"); // 2
qDebug() << list;                // QList("A", "C")
```

- `removeIf()` — удаляет элементы, для которых функция `pred` возвращает `true`. Прототип метода:

```
qsizetype removeIf(Predicate pred)
```

Пример:

```
QStringList list = { "A", "B", "C", "B" };
qsizetype n = list.removeIf([](const QString &s){
   return s == QString("B");
});
qDebug() << n;    // 2
qDebug() << list; // QList("A", "C")
```

- `takeFirst()` — удаляет первый элемент и возвращает его. Прототип метода:

```
QList::value_type takeFirst()
```

Пример:

```
QStringList list = { "A", "B", "C" };
QString s = list.takeFirst();
qDebug() << s;    // "A"
qDebug() << list; // QList("B", "C")
```

- `takeLast()` — удаляет последний элемент и возвращает его. Прототип метода:

```
QList::value_type takeLast()
```

Пример:

```
QStringList list = { "A", "B", "C" };
QString s = list.takeLast();
qDebug() << s;    // "C"
qDebug() << list; // QList("A", "B")
```

- `takeAt()` — удаляет элемент с указанным индексом и возвращает его. Прототип метода:

```
T takeAt(qsizetype i)
```

Пример:

```
QStringList list = { "A", "B", "C" };
QString s = list.takeAt(1);
qDebug() << s;    // "B"
qDebug() << list; // QList("A", "C")
```

- `clear()` — удаляет все элементы. Прототип метода:

```
void clear()
```

Пример:

```
QStringList list = { "A", "B", "C" };
list.clear();
qDebug() << list; // QList()
```

- `removeDuplicates()` — удаляет все повторы элементов. Метод возвращает количество удаленных элементов. Прототип метода:

```
qsizetype removeDuplicates()
```

Пример:

```
QStringList list = { "A", "B", "B", "A" };
qsizetype n = list.removeDuplicates();
qDebug() << list;   // QList("A", "B")
qDebug() << n;      // 2
```

2.4.5. Доступ к элементам

Обратиться к элементам списка можно, указав индекс внутри квадратных скобок. Можно как получить значение, так и изменить его:

```
QStringList list = { "A", "B", "C" };
qDebug() << list[0]; // "A"
list[0] = "D";
qDebug() << list; // QList("D", "B", "C")
```

Для доступа к элементам списка предназначены следующие методы:

- `front()` — возвращает ссылку на первый элемент. Метод позволяет как получить значение, так и изменить его. Прототипы метода:

```
QList::reference front()
QList::const_reference front() const
```

- `back()` — возвращает ссылку на последний элемент. Метод позволяет как получить значение, так и изменить его. Прототипы метода:

```
QList::reference back()
QList::const_reference back() const
```

Пример:

```
QStringList list = { "A", "B", "C" };
list.front() = "D";
list.back() = "E";
qDebug() << list.front(); // "D"
qDebug() << list.back();  // "E"
qDebug() << list;         // QList("D", "B", "E")
```

- `at()` — возвращает константную ссылку на элемент, расположенный по указанному индексу. Прототип метода:

```
QList::const_reference at(qsizetype i) const
```

Пример:

```
QStringList list = { "A", "B", "C" };
qDebug() << list.at(0); // "A"
```

- `first()` — возвращает ссылку на первый элемент. Метод позволяет как получить значение, так и изменить его. Прототипы метода:

```
T &first()
const T &first() const
```

- `last()` — возвращает ссылку на последний элемент. Метод позволяет как получить значение, так и изменить его. Прототипы метода:

```
T &last()
const T &last() const
```

Пример:

```
QStringList list = { "A", "B", "C" };
list.first() = "D";
list.last() = "E";
qDebug() << list.first(); // "D"
qDebug() << list.last();  // "E"
qDebug() << list;         // QList("D", "B", "E")
```

- `value()` — возвращает значение элемента, расположенного по указанному индексу, или значение `defaultValue`, если такого индекса нет. Прототипы метода:

```
T value(qsizetype i) const
T value(qsizetype i, QList::parameter_type defaultValue) const
```

Пример:

```
QStringList list = { "A", "B", "C" };
qDebug() << list.value(0); // "A"
qDebug() << list.value(1); // "B"
```

Можно также воспользоваться следующими методами:

```
QList::pointer data()
QList::const_pointer data() const
QList::const_pointer constData() const
const T &constFirst() const
const T &constLast() const
```

2.4.6. Итераторы

Итератор — это объект, выполняющий в контейнере роль указателя. С помощью итератора можно перемещаться внутри контейнера и получать доступ к отдельным элементам. В классе `QList` определены следующие типы итераторов:

- `iterator` — итератор. При увеличении значения итератор перемещается к концу списка. Пример объявления переменной:

  ```
  QStringList::iterator it;
  ```

- `const_iterator` — константный итератор. Изменить значение, на которое ссылается итератор, нельзя. Пример объявления переменной:

  ```
  QStringList::const_iterator it;
  ```

- `reverse_iterator` — обратный итератор. При увеличении значения итератор перемещается к началу списка. Пример объявления переменной:

  ```
  QStringList::reverse_iterator it;
  ```

- `const_reverse_iterator` — константный обратный итератор. Изменить значение, на которое ссылается итератор, нельзя. Пример объявления переменной:

  ```
  QStringList::const_reverse_iterator it;
  ```

Присвоить значения переменным позволяют следующие методы:

- `begin()` — возвращает итератор, установленный на первый элемент. Прототипы метода:

  ```
  QList::iterator begin()
  QList::const_iterator begin() const
  ```

  Выведем первый элемент:

  ```
  QStringList list = { "A", "B", "C" };
  QStringList::iterator it = list.begin();
  qDebug() << *it; // "A"
  ```

- `end()` — возвращает итератор, установленный на позицию после последнего элемента. Прототипы метода:

```
QList::iterator end()
QList::const_iterator end() const
```

Выведем последний символ строки:

```
QStringList list = { "A", "B", "C" };
QStringList::iterator it = list.end();
qDebug() << *(--it); // "C"
```

- `cbegin()` и `constBegin()` — возвращают константный итератор, установленный на первый символ строки. Прототипы методов:

```
QList::const_iterator cbegin() const
QList::const_iterator constBegin() const
```

- `cend()` и `constEnd()` — возвращают константный итератор, установленный на позицию после последнего символа строки. Прототипы методов:

```
QList::const_iterator cend() const
QList::const_iterator constEnd() const
```

- `rbegin()` — возвращает обратный итератор, установленный на последний элемент. Прототипы метода:

```
QList::reverse_iterator rbegin()
QList::const_reverse_iterator rbegin() const
```

Выведем последний элемент:

```
QStringList list = { "A", "B", "C" };
QStringList::reverse_iterator it = list.rbegin();
qDebug() << *it; // "C"
```

- `rend()` — возвращает обратный итератор, установленный на позицию перед первым элементом. Прототипы метода:

```
QList::reverse_iterator rend()
QList::const_reverse_iterator rend() const
```

Выведем первый элемент:

```
QStringList list = { "A", "B", "C" };
QStringList::reverse_iterator it = list.rend();
qDebug() << *(--it); // "A"
```

- `crbegin()` — возвращает константный обратный итератор, установленный на последний элемент. Прототип метода:

```
QList::const_reverse_iterator crbegin() const
```

- `crend()` — возвращает константный обратный итератор, установленный на позицию перед первым элементом. Прототип метода:

```
QList::const_reverse_iterator crend() const
```

С итераторами можно производить такие же операции, как и с указателями. Чтобы получить или изменить значение, на которое ссылается итератор, перед названием

переменной указывается оператор * (*it). Перемещение итератора осуществляется с помощью операторов +, -, ++ и --. Кроме того, итераторы можно сравнивать с помощью операторов сравнения.

2.4.7. Перебор элементов

Перебрать все элементы можно с помощью циклов for, for each и итераторов. Пример использования цикла for each:

```
QStringList list = { "A", "B", "C" };
for (QString &el : list) {
   qDebug() << el;
}
```

Пример перебора элементов с помощью итераторов и цикла for:

```
QStringList list = { "A", "B", "C" };
QStringList::iterator it1, it2;
QStringList::reverse_iterator it3, it4;
// Перебор в прямом порядке
for (it1 = list.begin(), it2 = list.end(); it1 != it2; ++it1) {
   qDebug() << *it1;
}
qDebug() << "------------------";
// Перебор в обратном порядке
for (it3 = list.rbegin(), it4 = list.rend(); it3 != it4; ++it3) {
   qDebug() << *it3;
}
```

2.4.8. Сортировка списка

Для сортировки списка предназначен метод sort(). Прототип метода:

```
void sort(Qt::CaseSensitivity cs = Qt::CaseSensitive)
```

Если в параметре cs указана константа Qt::CaseSensitive, то сравнение зависит от регистра символов. Чтобы сравнение не зависело от регистра символов, нужно указать константу Qt::CaseInsensitive:

```
QStringList list = { "C", "A", "B" };
list.sort(Qt::CaseSensitive);
qDebug() << list; // QList("A", "B", "C")
```

2.4.9. Получение фрагмента списка

Получить фрагмент списка позволяют следующие методы:

- sliced() — возвращает фрагмент длиной n, начиная с индекса pos. Если второй параметр не задан, то возвращаются все элементы до конца списка. Прототипы метода:

```
QList<T> sliced(qsizetype pos) const
QList<T> sliced(qsizetype pos, qsizetype n) const
```

Пример:

```
QStringList list = { "A", "B", "C", "D" };
QStringList list2 = list.sliced(1);
qDebug() << list2; // QList("B", "C", "D")
list2 = list.sliced(1, 2);
qDebug() << list2; // QList("B", "C")
```

- `mid()` — возвращает фрагмент длиной `length`, начиная с индекса `pos`. Если второй параметр не задан, то возвращаются все элементы до конца списка. Прототип метода:

```
QList<T> mid(qsizetype pos, qsizetype length = -1) const
```

Пример:

```
QStringList list = { "A", "B", "C", "D" };
QStringList list2 = list.mid(1);
qDebug() << list2; // QList("B", "C", "D")
list2 = list.mid(1, 2);
qDebug() << list2; // QList("B", "C")
```

- `first()` — возвращает список с первыми элементами в количестве `n`. Прототип метода:

```
QList<T> first(qsizetype n) const
```

Пример:

```
QStringList list = { "A", "B", "C", "D" };
QStringList list2 = list.first(2);
qDebug() << list2; // QList("A", "B")
```

- `last()` — возвращает список с последними элементами в количестве `n`. Прототип метода:

```
QList<T> last(qsizetype n) const
```

Пример:

```
QStringList list = { "A", "B", "C", "D" };
QStringList list2 = list.last(2);
qDebug() << list2; // QList("C", "D")
```

2.4.10. Поиск элементов

Выполнить поиск элемента внутри списка позволяют следующие методы:

- `indexOf()` — производит поиск элемента с начала списка или с индекса `from` до конца списка. Возвращает индекс первого совпадения, если элемент найден, или значение `-1` — в противном случае. Прототипы метода:

```
qsizetype indexOf(const AT &value, qsizetype from = 0) const
qsizetype indexOf(const QRegularExpression &re,
                  qsizetype from = 0) const
```

Пример:

```
QStringList list = { "A", "B", "C", "B" };
qDebug() << list.indexOf("B");    // 1
qDebug() << list.indexOf("D");    // -1
qDebug() << list.indexOf("B", 2); // 3
```

- `lastIndexOf()` — производит поиск элемента с конца списка или с индекса `from` до начала списка. Возвращает индекс первого совпадения, если элемент найден, или значение `-1` — в противном случае. Прототипы метода:

```
qsizetype lastIndexOf(const AT &value, qsizetype from = -1) const
qsizetype lastIndexOf(const QRegularExpression &re,
                      qsizetype from = -1) const
```

Пример:

```
QStringList list = { "A", "B", "C", "B" };
qDebug() << list.lastIndexOf("B");    // 3
qDebug() << list.lastIndexOf("D");    // -1
qDebug() << list.lastIndexOf("B", 2); // 1
```

- `contains()` — проверяет, содержит ли список указанный элемент. Если содержит, то возвращается значение `true`, в противном случае — `false`. Прототипы метода:

```
bool contains(const QString &str,
              Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool contains(QLatin1String str,
              Qt::CaseSensitivity cs = Qt::CaseSensitive) const
bool contains(QStringView str,
              Qt::CaseSensitivity cs = Qt::CaseSensitive) const
```

Если в параметре `cs` указана константа `Qt::CaseSensitive`, то сравнение зависит от регистра символов. Чтобы сравнение не зависело от регистра символов, нужно указать константу `Qt::CaseInsensitive`:

```
QStringList list = { "A", "B", "C" };
qDebug() << list.contains("B");                     // true
qDebug() << list.contains("D");                     // false
qDebug() << list.contains("b");                     // false
qDebug() << list.contains("b", Qt::CaseInsensitive); // true
```

- `startsWith()` — проверяет, начинается ли список с указанного элемента. Если начинается, то возвращается значение `true`, в противном случае — `false`. Прототип метода:

```
bool startsWith(QList::parameter_type value) const
```

Пример:

```
QStringList list = { "A", "B", "C" };
qDebug() << list.startsWith("A");      // true
qDebug() << list.startsWith("B");      // false
```

- `endsWith()` — проверяет, заканчивается ли список указанным элементом. Если заканчивается, то возвращается значение `true`, в противном случае — `false`. Прототип метода:

```
bool endsWith(QList::parameter_type value) const
```

Пример:

```
QStringList list = { "A", "B", "C" };
qDebug() << list.endsWith("C");      // true
qDebug() << list.endsWith("B");      // false
```

- `count()` — возвращает число вхождений элемента в список. Если элемент в списке отсутствует, то возвращается значение `0`. Прототип метода:

```
qsizetype count(const AT &value) const
```

Пример:

```
QStringList list = { "A", "B", "C", "B" };
qDebug() << list.count("B");      // 2
qDebug() << list.count("D");      // 0
```

2.4.11. Замена элементов

Произвести замену в списке позволяют следующие методы:

- `fill()` — заменяет все элементы указанным элементом. Во втором параметре можно передать новый размер списка. Прототип метода:

```
QList<T> &fill(QList::parameter_type value, qsizetype size = -1)
```

Пример:

```
QStringList list = { "A", "B", "C" };
qDebug() << list.fill("*");      // QList("*", "*", "*")
qDebug() << list.fill("+", 2);   // QList("+", "+")
```

- `move()` — перемещает элемент. Прототип метода:

```
void move(qsizetype from, qsizetype to)
```

Пример:

```
QStringList list = { "A", "B", "C" };
list.move(1, 2);
qDebug() << list;   // QList("A", "C", "B")
```

- `replace()` — заменяет элемент. Прототипы метода:

```
void replace(qsizetype i, QList::parameter_type value)
void replace(qsizetype i, QList::rvalue_ref value)
```

Пример:

```
QStringList list = { "A", "B", "C" };
list.replace(1, "D");
qDebug() << list;   // QList("A", "D", "C")
```

- `replaceInStrings()` — заменяет фрагмент `before` фрагментом `after` во всех элементах списка. Прототипы метода:

```
QStringList &replaceInStrings(const QString &before,
             const QString &after,
             Qt::CaseSensitivity cs = Qt::CaseSensitive)
QStringList &replaceInStrings(QStringView before,
             QStringView after,
             Qt::CaseSensitivity cs = Qt::CaseSensitive)
QStringList &replaceInStrings(const QString &before,
             QStringView after,
             Qt::CaseSensitivity cs = Qt::CaseSensitive)
QStringList &replaceInStrings(QStringView before,
             const QString &after,
             Qt::CaseSensitivity cs = Qt::CaseSensitive)
QStringList &replaceInStrings(const QRegularExpression &re,
                              const QString &after)
```

Заменим все числа символом + с помощью регулярного выражения:

```
// #include <QRegularExpression>
QStringList list = { "A12", "B23", "C45" };
list.replaceInStrings(QRegularExpression("[0-9]+"), "+");
qDebug() << list;   // QList("A+", "B+", "C+")
```

Если в параметре `cs` указана константа `Qt::CaseSensitive`, то поиск зависит от регистра символов. Чтобы поиск не зависел от регистра символов, нужно указать константу `Qt::CaseInsensitive`:

```
QStringList list = { "A1", "a2", "B3" };
list.replaceInStrings("a", "+");
qDebug() << list;   // QList("A1", "+2", "B3")
list.replaceInStrings("a", "+", Qt::CaseInsensitive);
qDebug() << list;   // QList("+1", "+2", "B3")
```

2.4.12. Фильтрация списка

Метод `filter()` возвращает список строк, содержащих подстроку `str`. Прототипы метода:

```
QStringList filter(const QString &str,
                   Qt::CaseSensitivity cs = Qt::CaseSensitive) const
QStringList filter(QStringView str,
                   Qt::CaseSensitivity cs = Qt::CaseSensitive) const
QStringList filter(const QRegularExpression &re) const
```

Пример получения элементов, содержащих числа:

```
// #include <QRegularExpression>
QStringList list = { "A12", "B23", "C", "D" };
QStringList list2 = list.filter(QRegularExpression("[0-9]+"));
qDebug() << list2;   // QList("A12", "B23")
```

Если в параметре `cs` указана константа `Qt::CaseSensitive`, то поиск зависит от регистра символов. Чтобы поиск не зависел от регистра символов, нужно указать константу `Qt::CaseInsensitive`:

```
QStringList list = { "A", "a", "C", "D" };
QStringList list2 = list.filter("a");
qDebug() << list2;   // QList("a")
list2 = list.filter("a", Qt::CaseInsensitive);
qDebug() << list2;   // QList("A", "a")
```

2.4.13. Преобразование списка в строку

Преобразовать список в строку позволяет метод `join()`. Элементы добавляются через указанный разделитель. Прототипы метода:

```
QString join(QChar separator) const
QString join(const QString &separator) const
QString join(QStringView separator) const
QString join(QLatin1String separator) const
```

Пример:

```
QStringList list = { "A", "B", "C", "D" };
QString str = list.join(L',');
qDebug() << str;   // "A,B,C,D"
str = list.join("+");
qDebug() << str;   // "A+B+C+D"
```

Выполнить обратную операцию позволяет метод `split()` из класса `QString` *(см. разд. 2.3.18)*:

```
QString str = "A,B,C,D";
QStringList list = str.split(",");
qDebug() << list;   // QList("A", "B", "C", "D")
```

ГЛАВА 3

Управление окном приложения

Создать окно и управлять им позволяет класс `QWidget`. Класс `QWidget` наследует два класса — `QObject` и `QPaintDevice`. В свою очередь, класс `QWidget` является базовым классом для всех визуальных компонентов, поэтому любой компонент, не имеющий родителя, обладает своим собственным окном. В этой главе мы рассмотрим методы класса `QWidget` применительно к окну верхнего уровня, однако следует помнить, что те же самые методы можно применять и к любым компонентам. Например, метод, позволяющий управлять размерами окна, можно использовать и для изменения размеров компонента, имеющего родителя. Тем не менее некоторые методы имеет смысл использовать только для окон верхнего уровня. Например, метод, позволяющий изменить текст в заголовке окна, не имеет смысла использовать в обычных компонентах.

Для создания окна верхнего уровня помимо класса `QWidget` можно использовать и другие классы, которые являются наследниками класса `QWidget`, например класс `QFrame` (окно с рамкой) или `QDialog` (диалоговое окно). При использовании класса `QDialog` окно будет выравниваться по центру экрана (или по центру родительского окна) и иметь только кнопку **Закрыть** в заголовке окна. Кроме того, можно использовать класс `QMainWindow`, который представляет главное окно приложения с меню, панелями инструментов и строкой состояния. Использование классов `QDialog` и `QMainWindow` имеет свои отличия, которые мы будем рассматривать в отдельных главах.

3.1. Создание и отображение окна

Самый простой способ создать пустое окно приведен в листинге 3.1.

Листинг 3.1. Создание и отображение окна

```
#include <QApplication>
#include <QWidget>

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

```
{
   QApplication app(argc, argv);
   QWidget window;
   window.setWindowTitle("Заголовок окна");
   window.resize(350, 70);
   window.show();
   return app.exec();
}
```

Конструктор класса `QWidget` имеет следующий формат:

```
QWidget(QWidget *parent = nullptr, Qt::WindowFlags f = Qt::WindowFlags())
```

В параметре `parent` передается указатель на родительский компонент. Если параметр не указан или имеет значение `nullptr`, то компонент будет обладать своим собственным окном. Если в параметре `f` указан тип окна, то компонент, имея родителя, будет обладать своим собственным окном, но будет привязан к родителю. Это позволяет, например, создать модальное окно, которое будет блокировать только окно родителя, а не все окна приложения. Какие именно значения можно указать в параметре `f`, мы рассмотрим в следующем разделе.

Передать указатель на родительский компонент уже после создания объекта позволяет метод `setParent()`. Прототипы метода:

```
void setParent(QWidget *parent)
void setParent(QWidget *parent, Qt::WindowFlags f)
```

Получить указатель на родительской компонент можно с помощью метода `parentWidget()`. Если компонент не имеет родителя, то метод возвращает нулевой указатель. Прототип метода:

```
QWidget *parentWidget() const
```

Для изменения текста в заголовке окна предназначен метод `setWindowTitle()`. Метод является слотом. Прототип метода:

```
void setWindowTitle(const QString &)
```

Пример:

```
window.setWindowTitle("Текст, отображаемый в заголовке");
```

После создания окна необходимо вызвать метод `show()`, чтобы окно и все дочерние компоненты отобразились на экране. Для сокрытия окна предназначен метод `hide()`. Методы `show()` и `hide()` являются слотами. Для отображения и сокрытия компонентов можно также воспользоваться методом `setVisible()`. Метод является слотом. Прототип метода:

```
virtual void setVisible(bool visible)
```

Если в параметре указано значение `true`, то компонент будет отображен, а если значение `false`, то компонент будет скрыт. Пример отображения окна и всех дочерних компонентов:

```
window.setVisible(true);
```

Проверить, отображен компонент в настоящее время или нет, позволяет метод `isVisible()`. Метод возвращает `true`, если компонент отображен, и `false` — в противном случае. Кроме того, можно воспользоваться методом `isHidden()`. Метод возвращает `true`, если компонент скрыт, и `false` — в противном случае. Прототипы методов:

```
bool isVisible() const
bool isHidden() const
```

3.2. Указание типа окна

При использовании класса `QWidget` по умолчанию окно создается с заголовком, в котором расположены: значок, текст заголовка и кнопки **Свернуть**, **Развернуть** и **Закрыть**. Указать другой тип создаваемого окна позволяет метод `setWindowFlags()` или параметр `f` в конструкторе класса `QWidget`. Обратите внимание на то, что метод `setWindowFlags()` должен вызываться перед отображением окна. Прототип метода:

```
void setWindowFlags(Qt::WindowFlags type)
```

В параметре `type` можно указать следующие константы:

- `Qt::Widget` — тип по умолчанию для класса `QWidget`;
- `Qt::Window` — указывает, что компонент является окном независимо от того, имеет он родителя или нет. Окно выводится с рамкой и заголовком, в котором расположены кнопки **Свернуть**, **Развернуть** и **Закрыть**. По умолчанию размеры окна можно изменять с помощью мыши;
- `Qt::Dialog` — диалоговое окно. Окно выводится с рамкой и заголовком, в котором расположена кнопка **Закрыть**. Размеры окна можно изменять с помощью мыши. Пример указания типа для диалогового окна:

  ```
  window.setWindowFlags(Qt::Dialog);
  ```

- `Qt::Sheet`;
- `Qt::Drawer`;
- `Qt::Popup` — указывает, что окно является всплывающим меню. Окно выводится без рамки и заголовка. Кроме того, окно может отбрасывать тень. Изменить размеры окна с помощью мыши нельзя;
- `Qt::Tool` — сообщает, что окно является панелью инструментов. Окно выводится с рамкой и заголовком (уменьшенным по высоте по сравнению с заголовком обычного окна), в котором расположена кнопка **Закрыть**. Размеры окна можно изменять с помощью мыши;
- `Qt::ToolTip` — указывает, что окно является всплывающей подсказкой. Окно выводится без рамки и заголовка. Изменить размеры окна с помощью мыши нельзя;
- `Qt::SplashScreen` — сообщает, что окно является заставкой. Окно выводится без рамки и заголовка. Изменить размеры окна с помощью мыши нельзя. Значение по умолчанию для класса `QSplashScreen`;

- `Qt::SubWindow` — сообщает, что окно является дочерним компонентом независимо от того, имеет он родителя или нет. Окно выводится с рамкой и заголовком (уменьшенным по высоте по сравнению с заголовком обычного окна) без кнопок. Изменить размеры окна с помощью мыши нельзя;
- `Qt::ForeignWindow`;
- `Qt::CoverWindow`.

Определить тип окна из программы позволяет метод `windowType()`. Прототип метода:

```
Qt::WindowType windowType() const
```

Для окон верхнего уровня можно дополнительно указать следующие константы через оператор `|` (перечислены только наиболее часто используемые константы; полный список смотрите в документации):

- `Qt::MSWindowsFixedSizeDialogHint` — запрещает изменение размеров окна. Изменить размеры с помощью мыши нельзя. Кнопка **Развернуть** в заголовке окна становится неактивной;
- `Qt::FramelessWindowHint` — убирает рамку и заголовок окна. Изменять размеры окна и перемещать его нельзя;
- `Qt::CustomizeWindowHint` — убирает заголовок окна. Размеры окна можно изменять с помощью мыши;
- `Qt::WindowTitleHint` — добавляет заголовок окна. Выведем окно с заголовком, в котором находится только текст и неактивная кнопка **Закрыть**:

```
window.setWindowFlags(Qt::Window |
                      Qt::WindowTitleHint);
```

- `Qt::WindowSystemMenuHint` — добавляет оконное меню;
- `Qt::WindowMinimizeButtonHint` — кнопка **Свернуть** в заголовке окна делается активной;
- `Qt::WindowMaximizeButtonHint` — кнопка **Развернуть** в заголовке окна делается активной;
- `Qt::WindowMinMaxButtonsHint` — кнопки **Свернуть** и **Развернуть** в заголовке окна делаются активными;
- `Qt::WindowCloseButtonHint` — добавляет кнопку **Закрыть**;
- `Qt::WindowContextHelpButtonHint` — добавляет кнопку **Справка**. Кнопки **Свернуть** и **Развернуть** в заголовке окна не отображаются. Добавим кнопки **Справка** и **Закрыть**:

```
window.setWindowFlags(Qt::Window |
                      Qt::WindowCloseButtonHint |
                      Qt::WindowContextHelpButtonHint);
```

- `Qt::WindowStaysOnTopHint` — сообщает системе, что окно всегда должно отображаться поверх всех других окон;

- `Qt::WindowStaysOnBottomHint` — сообщает системе, что окно всегда должно быть расположено позади всех других окон.

Получить все установленные флаги из программы позволяет метод `windowFlags()`. Прототип метода:

```
Qt::WindowFlags windowFlags() const
```

3.3. Изменение и получение размеров окна

Для изменения размеров окна предназначены следующие методы:

- `resize()` — изменяет текущий размер окна. Если содержимое окна не помещается в установленный размер, то размер будет выбран так, чтобы компоненты поместились без искажения, при условии, что используются менеджеры геометрии. Следовательно, заданный размер может не соответствовать реальному размеру окна. Если используется абсолютное позиционирование, то компоненты могут оказаться наполовину или полностью за пределами видимой части окна. Прототипы метода:

  ```
  void resize(int w, int h)
  void resize(const QSize &)
  ```

  Пример:

  ```
  window.resize(350, 70);
  ```

- `setGeometry()` — изменяет одновременно положение компонента и его размер. Первые два параметра задают координаты левого верхнего угла (относительно родительского компонента или экрана), а третий и четвертый параметры — ширину и высоту. Прототипы метода:

  ```
  void setGeometry(int x, int y, int w, int h)
  void setGeometry(const QRect &)
  ```

  Пример:

  ```
  window.setGeometry(100, 100, 350, 70);
  ```

- `setFixedSize()` — задает фиксированный размер. Изменить размеры окна с помощью мыши нельзя. Кнопка **Развернуть** в заголовке окна становится неактивной. Прототипы метода:

  ```
  void setFixedSize(int w, int h)
  void setFixedSize(const QSize &s)
  ```

  Пример:

  ```
  window.setFixedSize(350, 70);
  ```

- `setFixedWidth()` — задает фиксированный размер только по ширине. Изменить ширину окна с помощью мыши нельзя. Прототип метода:

  ```
  void setFixedWidth(int w)
  ```

- `setFixedHeight()` — задает фиксированный размер только по высоте. Изменить высоту окна с помощью мыши нельзя. Прототип метода:

```
void setFixedHeight(int h)
```

- `setMinimumSize()` — задает минимальный размер. Прототипы метода:

```
void setMinimumSize(int minw, int minh)
void setMinimumSize(const QSize &)
```

  Пример:

```
window.setMinimumSize(350, 70);
```

- `setMinimumWidth()` и `setMinimumHeight()` — задают минимальный размер только по ширине и высоте соответственно. Прототипы методов:

```
void setMinimumWidth(int minw)
void setMinimumHeight(int minh)
```

- `setMaximumSize()` — задает максимальный размер. Прототипы метода:

```
void setMaximumSize(int maxw, int maxh)
void setMaximumSize(const QSize &)
```

- `setMaximumWidth()` и `setMaximumHeight()` — задают максимальный размер только по ширине и высоте соответственно. Прототипы методов:

```
void setMaximumWidth(int maxw)
void setMaximumHeight(int maxh)
```

- `setBaseSize()` — задает базовые размеры. Прототипы метода:

```
void setBaseSize(int basew, int baseh)
void setBaseSize(const QSize &)
```

- `adjustSize()` — подгоняет размеры компонента под содержимое. При этом учитываются рекомендуемые размеры, возвращаемые методом `sizeHint()`. Прототип метода:

```
void adjustSize()
```

Получить размеры позволяют следующие методы:

- `width()` и `height()` — возвращают ширину и высоту соответственно. Прототипы методов:

```
int width() const
int height() const
```

  Пример:

```
window.resize(350, 70);
qDebug() << window.width() << window.height(); // 350 70
```

- `size()` — возвращает экземпляр класса `QSize`, содержащий размеры. Прототип метода:

```
QSize size() const
```

Пример:

```
window.resize(350, 70);
qDebug() << window.size().width()
         << window.size().height(); // 350 70
```

- `minimumSize()` — возвращает экземпляр класса `QSize`, содержащий минимальные размеры. Прототип метода:

```
QSize minimumSize() const
```

- `minimumWidth()` и `minimumHeight()` — возвращают минимальную ширину и высоту соответственно. Прототипы методов:

```
int minimumWidth() const
int minimumHeight() const
```

- `maximumSize()` — возвращает экземпляр класса `QSize`, содержащий максимальные размеры. Прототип метода:

```
QSize maximumSize() const
```

- `maximumWidth()` и `maximumHeight()` — возвращают максимальную ширину и высоту соответственно. Прототипы методов:

```
int maximumWidth() const
int maximumHeight() const
```

- `baseSize()` — возвращает экземпляр класса `QSize`, содержащий базовые размеры. Прототип метода:

```
QSize baseSize() const
```

- `sizeHint()` — возвращает экземпляр класса `QSize`, содержащий рекомендуемый размер компонента. Если возвращаемые размеры являются отрицательными, то считается, что нет рекомендуемого размера. Прототип метода:

```
virtual QSize sizeHint() const
```

- `minimumSizeHint()` — возвращает экземпляр класса `QSize`, содержащий рекомендуемый минимальный размер компонента. Если возвращаемые размеры являются отрицательными, то считается, что нет рекомендуемого минимального размера. Прототип метода:

```
virtual QSize minimumSizeHint() const
```

- `rect()` — возвращает экземпляр класса `QRect`, содержащий координаты и размеры прямоугольника, в который вписан компонент. Прототип метода:

```
QRect rect() const
```

Пример:

```
window.setGeometry(100, 100, 350, 70);
QRect rect = window.rect();
qDebug() << rect.left() << rect.top();     // 0 0
qDebug() << rect.width() << rect.height(); // 350 70
```

- `geometry()` — возвращает ссылку на экземпляр класса `QRect`, содержащий координаты относительно родительского компонента. Прототип метода:

```
const QRect &geometry() const
```

Пример:

```
window.setGeometry(100, 100, 350, 70);
QRect rect = window.geometry();
qDebug() << rect.left() << rect.top();     // 100 100
qDebug() << rect.width() << rect.height(); // 350 70
```

При изменении и получении размеров окна следует учитывать, что:

- размеры не включают высоту заголовка окна и ширину границ;
- размер компонентов может изменяться в зависимости от настроек стиля. Например, на разных компьютерах может быть указан шрифт разного наименования и размера. Поэтому от указания фиксированных размеров лучше отказаться;
- размер окна может изменяться в промежутке между получением значения и действиями, выполняющими обработку этих значений в программе. Например, сразу после получения размера пользователь изменил размеры окна с помощью мыши.

Чтобы получить размеры окна, включая высоту заголовка и ширину границ, следует воспользоваться методом `frameSize()`. Метод возвращает экземпляр класса `QSize`. Прототип метода:

```
QSize frameSize() const
```

Обратите внимание на то, что полные размеры окна доступны только после отображения окна. До этого момента размеры совпадают с размерами окна без учета высоты заголовка и ширины границ. Пример получения полного размера окна:

```
window.resize(350, 70);                  // Задаем размеры
// ...
window.show();                           // Отображаем окно
qDebug() << window.width()
         << window.height();             // 350 70
qDebug() << window.frameSize().width()
         << window.frameSize().height(); // 366 109
```

Чтобы получить координаты окна с учетом высоты заголовка и ширины границ, следует воспользоваться методом `frameGeometry()`. Прототип метода:

```
QRect frameGeometry() const
```

Обратите внимание на то, что полные размеры окна доступны только после отображения окна. Пример:

```
window.setGeometry(100, 100, 350, 70);
// ...
window.show();                               // Отображаем окно
```

```
QRect rect = window.geometry();
qDebug() << rect.left() << rect.top();     // 100 100
qDebug() << rect.width() << rect.height(); // 350 70
rect = window.frameGeometry();
qDebug() << rect.left() << rect.top();     // 92 69
qDebug() << rect.width() << rect.height(); // 366 109
```

3.4. Местоположение окна на экране

Задать местоположение окна на экране монитора позволяют следующие методы:

- `move()` — изменяет положение компонента относительно родителя. Метод учитывает высоту заголовка и ширину границ. Прототипы метода:

  ```
  void move(int x, int y)
  void move(const QPoint &)
  ```

  Пример вывода окна в левом верхнем углу экрана:

  ```
  window.move(0, 0);
  ```

- `setGeometry()` — изменяет одновременно положение компонента и его размер. Первые два параметра задают координаты левого верхнего угла (относительно родительского компонента), а третий и четвертый параметры — ширину и высоту. Обратите внимание на то, что метод не учитывает высоту заголовка и ширину границ. Поэтому, если указать координаты `(0, 0)`, заголовок окна и левая граница будут за пределами экрана. Прототипы метода:

  ```
  void setGeometry(int x, int y, int w, int h)
  void setGeometry(const QRect &)
  ```

  Пример:

  ```
  window.setGeometry(100, 100, 350, 70);
  ```

  > ***ОБРАТИТЕ ВНИМАНИЕ***
  >
  > Начало координат расположено в левом верхнем углу. Положительная ось `X` направлена вправо, а положительная ось `Y` — вниз.

Получить позицию окна позволяют следующие методы:

- `x()` и `y()` — возвращают координаты левого верхнего угла окна относительно родителя по осям `X` и `Y` соответственно. Методы учитывают высоту заголовка и ширину границ. Прототипы методов:

  ```
  int x() const
  int y() const
  ```

  Пример:

  ```
  window.move(10, 10);
  qDebug() << window.x() << window.y(); // 10 10
  ```

- `pos()` — возвращает экземпляр класса `QPoint`, содержащий координаты левого верхнего угла окна относительно родителя. Метод учитывает высоту заголовка и ширину границ. Прототип метода:

  ```
  QPoint pos() const
  ```

  Пример:

  ```
  window.move(10, 10);
  qDebug() << window.pos().x() << window.pos().y(); // 10 10
  ```

- `geometry()` — возвращает ссылку на экземпляр класса `QRect`, содержащий координаты относительно родительского компонента. Обратите внимание на то, что метод не учитывает высоту заголовка и ширину границ. Прототип метода:

  ```
  const QRect &geometry() const
  ```

  Пример:

  ```
  window.resize(350, 70);
  window.move(10, 10);
  // ...
  window.show();
  QRect rect = window.geometry();
  qDebug() << rect.left() << rect.top();     // 18 41
  qDebug() << rect.width() << rect.height(); // 350 70
  ```

- `frameGeometry()` — возвращает экземпляр класса `QRect`, содержащий координаты с учетом высоты заголовка и ширины границ. Обратите внимание на то, что полные размеры окна доступны только после отображения окна. Прототип метода:

  ```
  QRect frameGeometry() const
  ```

  Пример:

  ```
  window.resize(350, 70);
  window.move(10, 10);
  // ...
  window.show();
  QRect rect = window.frameGeometry();
  qDebug() << rect.left() << rect.top();     // 10 10
  qDebug() << rect.width() << rect.height(); // 366 109
  ```

3.4.1. Получение информации о размере экрана

Для отображения окна по центру экрана или у правой или нижней границы необходимо знать размеры экрана. Для получения размеров экрана сначала следует вызвать статический метод `QApplication::primaryScreen()`, который возвращает указатель на объект основного экрана. Прототип метода:

```
#include <QScreen>
static QScreen *primaryScreen()
```

Получить размеры экрана позволяют следующие методы из класса `QScreen`:

- `geometry()` — возвращает экземпляр класса `QRect`, содержащий координаты и размеры всего экрана. Прототип метода:

  ```
  QRect geometry() const
  ```

  Пример:

  ```
  // #include <QScreen>
  QScreen *screen = QApplication::primaryScreen();
  QRect rect = screen->geometry();
  qDebug() << rect.left() << rect.top();     // 0 0
  qDebug() << rect.width() << rect.height(); // 1920 1080
  ```

- `availableGeometry()` — возвращает экземпляр класса `QRect`, содержащий координаты и размеры только доступной части экрана (без размера **Панели задач**). Прототип метода:

  ```
  QRect availableGeometry() const
  ```

  Пример:

  ```
  QScreen *screen = QApplication::primaryScreen();
  QRect rect = screen->availableGeometry();
  qDebug() << rect.left() << rect.top();     // 0 0
  qDebug() << rect.width() << rect.height(); // 1920 1040
  ```

Получить список всех экранов позволяет статический метод `screens()` объекта приложения. Прототип метода:

```
static QList<QScreen *> screens()
```

Пример отображения окна примерно по центру экрана показан в листинге 3.2.

Листинг 3.2. Вывод окна примерно по центру экрана

```
#include <QApplication>
#include <QWidget>
#include <QScreen>

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   QWidget window;
   window.setWindowTitle("Вывод окна по центру экрана");
   window.resize(350, 70);

   QScreen *screen = QApplication::primaryScreen();
   QRect rect = screen->geometry();
   int x = (rect.width() - window.width()) / 2;
   int y = (rect.height() - window.height()) / 2;
   window.move(x, y);
```

```
    window.show();
    return app.exec();
}
```

В этом примере мы воспользовались методами `width()` и `height()`, которые не учитывают высоту заголовка и ширину границ. В большинстве случаев этого способа достаточно. Если необходима точность при выравнивании, то для получения размеров окна можно воспользоваться методом `frameSize()`. Однако этот метод возвращает корректные значения только после отображения окна. Если код выравнивания по центру расположить после вызова метода `show()`, то окно отобразится сначала в одном месте экрана, а затем переместится в центр, что вызовет неприятное мелькание. Чтобы исключить мелькание, следует сначала отобразить окно за пределами экрана, а затем переместить окно в центр экрана (листинг 3.3).

Листинг 3.3. Вывод окна по центру экрана

```
#include <QApplication>
#include <QWidget>
#include <QScreen>

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    QWidget window;
    window.setWindowTitle("Вывод окна по центру экрана");
    window.resize(350, 70);
    window.move(window.width() * -2, 0);
    window.show();

    QScreen *screen = QApplication::primaryScreen();
    QRect rect = screen->geometry();
    int x = (rect.width() - window.frameSize().width()) / 2;
    int y = (rect.height() - window.frameSize().height()) / 2;
    window.move(x, y);

    return app.exec();
}
```

Этот способ можно также использовать для выравнивания окна по правому краю экрана. Например, чтобы расположить окно в правом верхнем углу экрана, следует заменить код, выравнивающий окно по центру, из предыдущего примера следующим кодом:

```
QScreen *screen = QApplication::primaryScreen();
QRect rect = screen->availableGeometry();
int x = rect.x() + rect.width() - window.frameSize().width();
window.move(x, rect.y());
```

Если попробовать вывести окно в правом нижнем углу, то может возникнуть проблема, т. к. в операционной системе Windows в нижней части экрана располагается **Панель задач**. В итоге окно будет частично расположено под **Панелью задач**. Чтобы при размещении окна учитывать местоположение **Панели задач**, необходимо использовать метод `availableGeometry()`. Получить высоту **Панели задач**, расположенной в нижней части экрана, можно, например, так:

```
QScreen *screen = QApplication::primaryScreen();
QRect rect = screen->geometry();
int taskBarHeight = rect.height() -
                    screen->availableGeometry().height();
```

Следует также заметить, что в некоторых операционных системах **Панель задач** может быть прикреплена к любой стороне экрана. Кроме того, экран может быть разделен на несколько рабочих столов. Все это необходимо учитывать при размещении окна. За подробной информацией обращайтесь к документации по классу `QScreen`.

3.5. Указание координат и размеров

В двух предыдущих разделах были упомянуты классы `QPoint`, `QSize` и `QRect`. Класс `QPoint` описывает координаты точки, класс `QSize` — размеры, а класс `QRect` — координаты и размеры прямоугольной области. Рассмотрим эти классы более подробно.

> ***Примечание***
>
> Классы `QPoint`, `QSize` и `QRect` предназначены для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать классы `QPointF`, `QSizeF` и `QRectF` соответственно.

3.5.1. Класс *QPoint:* координаты точки

Класс `QPoint` описывает координаты точки. Для создания экземпляра класса предназначены следующие форматы конструкторов:

```
#include <QPoint>
QPoint()
QPoint(int xpos, int ypos)
```

Первый конструктор создает экземпляр класса с нулевыми координатами:

```
QPoint p;
qDebug() << p.x() << p.y(); // 0 0
```

Второй конструктор позволяет явно указать координаты точки:

```
QPoint p(10, 88);
qDebug() << p.x() << p.y(); // 10 88
```

Пример создания копии объекта:

```
QPoint p(10, 88);
QPoint p2(p);
qDebug() << p2.x() << p2.y(); // 10 88
```

Через экземпляр класса доступны следующие методы:

- `x()` и `y()` — возвращают координаты по осям `X` и `Y` соответственно. Прототипы методов:

```
int x() const
int y() const
```

- `rx()` и `ry()` — возвращают ссылки на координаты по осям `X` и `Y` соответственно. Прототипы методов:

```
int &rx()
int &ry()
```

- `setX()` и `setY()` — задают координаты по осям `X` и `Y` соответственно. Прототипы методов:

```
void setX(int x)
void setY(int y)
```

- `isNull()` — возвращает `true`, если координаты равны нулю, и `false` — в противном случае. Прототип метода:

```
bool isNull() const
```

  Пример:

```
QPoint p;
qDebug() << p.isNull();     // true
p.setX(10);
p.setY(88);
qDebug() << p.x() << p.y(); // 10 88
qDebug() << p.isNull();     // false
```

- `manhattanLength()` — возвращает сумму абсолютных значений координат:

```
int manhattanLength() const
```

  Пример:

```
QPoint p(10, 88);
qDebug() << p.manhattanLength();     // 98
```

Над двумя экземплярами класса `QPoint` определены операции `+`, `+=`, `-` (минус), `-=`, `==` и `!=`. Для смены знака координат можно воспользоваться унарным оператором `-`. Кроме того, экземпляр класса `QPoint` можно умножить или разделить на вещественное число (операторы `*`, `*=`, `/` и `/=`). Пример:

```
QPoint p1(10, 20), p2(5, 9);
qDebug() << p1 + p2;     // QPoint(15,29)
qDebug() << p1 - p2;     // QPoint(5,11)
qDebug() << p1 * 2.5;    // QPoint(25,50)
qDebug() << p1 / 2.0;    // QPoint(5,10)
qDebug() << (p1 == p2);  // false
qDebug() << (p1 != p2);  // true
qDebug() << (-p1);       // QPoint(-10,-20)
```

3.5.2. Класс *QSize:* размеры прямоугольной области

Класс `QSize` описывает размеры прямоугольной области. Для создания экземпляра класса предназначены следующие форматы конструкторов:

```
#include <QSize>
QSize()
QSize(int width, int height)
```

Первый конструктор создает экземпляр класса с отрицательной шириной и высотой. Второй конструктор позволяет явно указать ширину и высоту. Пример:

```
QSize s1;
qDebug() << s1;     // QSize(-1, -1)
QSize s2(10, 55);
qDebug() << s2;     // QSize(10, 55)
```

Пример создания копии объекта:

```
QSize s1(10, 55);
QSize s2(s1);
qDebug() << s2;     // QSize(10, 55)
```

Через экземпляр класса доступны следующие методы:

- `width()` и `height()` — возвращают ширину и высоту соответственно. Прототипы методов:

  ```
  int width() const
  int height() const
  ```

- `rwidth()` и `rheight()` — возвращают ссылки на ширину и высоту соответственно. Прототипы методов:

  ```
  int &rwidth()
  int &rheight()
  ```

- `setWidth()` и `setHeight()` — задают ширину и высоту соответственно. Прототипы методов:

  ```
  void setWidth(int width)
  void setHeight(int height)
  ```

  Пример:

  ```
  QSize s;
  s.setWidth(10);
  s.setHeight(55);
  qDebug() << s.width();     // 10
  qDebug() << s.height();    // 55
  ```

- `isNull()` — возвращает `true`, если ширина и высота равны нулю, и `false` — в противном случае. Прототип метода:

  ```
  bool isNull() const
  ```

- `isValid()` — возвращает `true`, если ширина и высота больше или равны нулю, и `false` — в противном случае. Прототип метода:

```
bool isValid() const
```

- `isEmpty()` — возвращает `true`, если один параметр (ширина или высота) меньше или равен нулю, и `false` — в противном случае. Прототип метода:

```
bool isEmpty() const
```

- `scale()` — производит изменение размеров области в соответствии со значением параметра `mode`. Метод изменяет текущий объект и ничего не возвращает. Прототипы метода:

```
void scale(int width, int height, Qt::AspectRatioMode mode)
void scale(const QSize &size, Qt::AspectRatioMode mode)
```

  В параметре `mode` могут быть указаны следующие константы:

  - `Qt::IgnoreAspectRatio` — непосредственно изменяет размеры без сохранения пропорций сторон;
  - `Qt::KeepAspectRatio` — производится попытка масштабирования старой области внутри новой области без нарушения пропорций;
  - `Qt::KeepAspectRatioByExpanding` — производится попытка полностью заполнить новую область без нарушения пропорций старой области.

  Если новая ширина или высота имеет значение `0`, то размеры изменяются непосредственно без сохранения пропорций вне зависимости от значения параметра `mode`.

  Пример:

```
QSize s(50, 20);
s.scale(70, 60, Qt::IgnoreAspectRatio);
qDebug() << s;              // QSize(70, 60)
s = QSize(50, 20);
s.scale(70, 60, Qt::KeepAspectRatio);
qDebug() << s;              // QSize(70, 28)
s = QSize(50, 20);
s.scale(70, 60, Qt::KeepAspectRatioByExpanding);
qDebug() << s;              // QSize(150, 60)
```

  Можно также воспользоваться методом `scaled()`. Прототипы метода:

```
QSize scaled(int width, int height,
             Qt::AspectRatioMode mode) const
QSize scaled(const QSize &s, Qt::AspectRatioMode mode) const
```

  Пример:

```
QSize s(50, 20), s2;
s2 = s.scaled(70, 60, Qt::IgnoreAspectRatio);
qDebug() << s2;              // QSize(70, 60)
```

```
s2 = s.scaled(70, 60, Qt::KeepAspectRatio);
qDebug() << s2;             // QSize(70, 28)
s2 = s.scaled(70, 60, Qt::KeepAspectRatioByExpanding);
qDebug() << s2;             // QSize(150, 60)
```

- `boundedTo()` — возвращает экземпляр класса `QSize`, который содержит минимальную ширину и высоту из текущих размеров и размеров, указанных в параметре. Прототип метода:

  ```
  QSize boundedTo(const QSize &otherSize) const
  ```

  Пример:

  ```
  QSize s(50, 20);
  qDebug() << s.boundedTo(QSize(400, 5)); // QSize(50, 5)
  qDebug() << s.boundedTo(QSize(40, 50)); // QSize(40, 20)
  ```

- `expandedTo()` — возвращает экземпляр класса `QSize`, который содержит максимальную ширину и высоту из текущих размеров и размеров, указанных в параметре. Прототип метода:

  ```
  QSize expandedTo(const QSize &otherSize) const
  ```

  Пример:

  ```
  QSize s(50, 20);
  qDebug() << s.expandedTo(QSize(400, 5)); // QSize(400, 20)
  qDebug() << s.expandedTo(QSize(40, 50)); // QSize(50, 50)
  ```

- `transpose()` — меняет значения местами. Метод изменяет текущий объект и ничего не возвращает. Прототип метода:

  ```
  void transpose()
  ```

  Пример:

  ```
  QSize s(50, 20);
  s.transpose();
  qDebug() << s; // QSize(20, 50)
  ```

  Можно также воспользоваться методом `transposed()`. Прототип метода:

  ```
  QSize transposed() const
  ```

  Пример:

  ```
  QSize s(50, 20);
  qDebug() << s.transposed(); // QSize(20, 50)
  ```

Над двумя экземплярами класса `QSize` определены операции `+`, `+=`, `-` (минус), `-=`, `==` и `!=`. Кроме того, экземпляр класса `QSize` можно умножить или разделить на вещественное число (операторы `*`, `*=`, `/` и `/=`). Пример:

```
QSize s1(50, 20), s2(10, 5);
qDebug() << s1 + s2;    // QSize(60, 25)
qDebug() << s1 - s2;    // QSize(40, 15)
```

```
qDebug() << s1 * 2.5;   // QSize(125, 50)
qDebug() << s1 / 2.0;   // QSize(25, 10)
qDebug() << (s1 == s2); // false
qDebug() << (s1 != s2); // true
```

3.5.3. Класс *QRect:* координаты и размеры прямоугольной области

Класс `QRect` описывает координаты и размеры прямоугольной области. Для создания экземпляра класса предназначены следующие форматы конструктора:

```
#include <QRect>
QRect()
QRect(int x, int y, int width, int height)
QRect(const QPoint &topLeft, const QSize &size)
QRect(const QPoint &topLeft, const QPoint &bottomRight)
```

Первый конструктор создает экземпляр класса со значениями по умолчанию:

```
QRect r;
qDebug() << r.left() << r.top() << r.right()
         << r.bottom() << r.width() << r.height(); // 0 0 -1 -1 0 0
```

Второй и третий конструкторы позволяют указать координаты левого верхнего угла и размеры области. Во втором конструкторе значения указываются отдельно:

```
QRect r(10, 15, 400, 300);
qDebug() << r.left() << r.top() << r.right()
         << r.bottom() << r.width() << r.height();
// 10 15 409 314 400 300
```

В третьем конструкторе координаты задаются с помощью класса `QPoint`, а размеры — с помощью класса `QSize`:

```
QRect r(QPoint(10, 15), QSize(400, 300));
qDebug() << r.left() << r.top() << r.right()
         << r.bottom() << r.width() << r.height();
// 10 15 409 314 400 300
```

Четвертый конструктор позволяет указать координаты левого верхнего угла и правого нижнего угла. В качестве значений указываются экземпляры класса `QPoint`:

```
QRect r(QPoint(10, 15), QPoint(409, 314));
qDebug() << r.left() << r.top() << r.right()
         << r.bottom() << r.width() << r.height();
// 10 15 409 314 400 300
```

Пример создания копии объекта:

```
QRect r(10, 15, 400, 300);
QRect r2(r);
qDebug() << r2; // QRect(10,15 400x300)
```

Изменить значения уже после создания экземпляра позволяют следующие методы:

- `setLeft()`, `setX()`, `setTop()` и `setY()` — задают координаты левого верхнего угла по осям `X` и `Y`. Прототипы методов:

```
void setLeft(int x)
void setX(int x)
void setTop(int y)
void setY(int y)
```

  Пример:

```
QRect r;
r.setLeft(10);
r.setTop(55);
qDebug() << r; // QRect(10,55 -10x-55)
r.setX(12);
r.setY(81);
qDebug() << r; // QRect(12,81 -12x-81)
```

- `setRight()` и `setBottom()` — задают координаты правого нижнего угла по осям `X` и `Y`. Прототипы методов:

```
void setRight(int x)
void setBottom(int y)
```

  Пример:

```
QRect r;
r.setRight(12);
r.setBottom(81);
qDebug() << r; // QRect(0,0 13x82)
```

- `setTopLeft()` — задает координаты левого верхнего угла. Прототип метода:

```
void setTopLeft(const QPoint &position)
```

- `setTopRight()` — задает координаты правого верхнего угла. Прототип метода:

```
void setTopRight(const QPoint &position)
```

- `setBottomLeft()` — задает координаты левого нижнего угла. Прототип метода:

```
void setBottomLeft(const QPoint &position)
```

- `setBottomRight()` — задает координаты правого нижнего угла. Прототип метода:

```
void setBottomRight(const QPoint &position)
```

  Пример:

```
QRect r;
r.setTopLeft(QPoint(10, 5));
r.setBottomRight(QPoint(39, 19));
qDebug() << r; // QRect(10,5 30x15)
r.setTopRight(QPoint(39, 5));
r.setBottomLeft(QPoint(10, 19));
qDebug() << r; // QRect(10,5 30x15)
```

- `setWidth()`, `setHeight()` и `setSize()` — задают ширину и высоту области. Прототипы методов:

```
void setWidth(int width)
void setHeight(int height)
void setSize(const QSize &size)
```

- `setRect()` — задает координаты левого верхнего угла и размеры области. Прототип метода:

```
void setRect(int x, int y, int width, int height)
```

- `setCoords()` — задает координаты левого верхнего угла и правого нижнего угла. Прототип метода:

```
void setCoords(int x1, int y1, int x2, int y2)
```

Пример:

```
QRect r;
r.setRect(10, 10, 100, 500);
qDebug() << r; // QRect(10,10 100x500)
r.setCoords(10, 10, 109, 509);
qDebug() << r; // QRect(10,10 100x500)
```

Переместить область при изменении координат позволяют следующие методы:

- `moveTo()`, `moveLeft()` и `moveTop()` — перемещают координаты левого верхнего угла. Прототипы методов:

```
void moveTo(int x, int y)
void moveTo(const QPoint &position)
void moveLeft(int x)
void moveTop(int y)
```

Пример:

```
QRect r(10, 15, 400, 300);
r.moveTo(0, 0);
qDebug() << r; // QRect(0,0 400x300)
r.moveTo(QPoint(10, 10));
qDebug() << r; // QRect(10,10 400x300)
r.moveLeft(5);
r.moveTop(0);
qDebug() << r; // QRect(5,0 400x300)
```

- `moveRight()` и `moveBottom()` — перемещают координаты правого нижнего угла. Прототипы методов:

```
void moveRight(int x)
void moveBottom(int y)
```

- `moveTopLeft()` — перемещает координаты левого верхнего угла. Прототип метода:

```
void moveTopLeft(const QPoint &position)
```

- `moveTopRight()` — перемещает координаты правого верхнего угла. Прототип метода:

```
void moveTopRight(const QPoint &position)
```

- `moveBottomLeft()` — перемещает координаты левого нижнего угла. Прототип метода:

```
void moveBottomLeft(const QPoint &position)
```

- `moveBottomRight()` — перемещает координаты правого нижнего угла. Прототип метода:

```
void moveBottomRight(const QPoint &position)
```

Пример:

```
QRect r(10, 15, 400, 300);
r.moveTopLeft(QPoint(0, 0));
qDebug() << r; // QRect(0,0 400x300)
r.moveBottomRight(QPoint(599, 499));
qDebug() << r; // QRect(200,200 400x300)
```

- `moveCenter()` — перемещает координаты центра. Прототип метода:

```
void moveCenter(const QPoint &position)
```

- `translate()` — перемещает координаты левого верхнего угла относительно текущего значения координат. Прототипы метода:

```
void translate(int dx, int dy)
void translate(const QPoint &offset)
```

Пример:

```
QRect r(0, 0, 400, 300);
r.translate(20, 15);
qDebug() << r; // QRect(20,15 400x300)
r.translate(QPoint(10, 5));
qDebug() << r; // QRect(30,20 400x300)
```

- `translated()` — метод аналогичен методу `translate()`, но возвращает новый экземпляр класса `QRect`, а не изменяет текущий. Прототипы метода:

```
QRect translated(int dx, int dy) const
QRect translated(const QPoint &offset) const
```

- `adjust()` — сдвигает координаты левого верхнего угла и правого нижнего угла относительно текущих значений координат. Прототип метода:

```
void adjust(int dx1, int dy1, int dx2, int dy2)
```

Пример:

```
QRect r(0, 0, 400, 300);
r.adjust(10, 5, 10, 5);
qDebug() << r; // QRect(10,5 400x300)
```

- `adjusted()` — метод аналогичен методу `adjust()`, но возвращает новый экземпляр класса `QRect`, а не изменяет текущий. Прототип метода:

```
QRect adjusted(int dx1, int dy1, int dx2, int dy2) const
```

Для получения значений предназначены следующие методы:

- `left()` и `x()` — возвращают координату левого верхнего угла по оси `X`. Прототипы методов:

```
int left() const
int x() const
```

- `top()` и `y()` — возвращают координату левого верхнего угла по оси `Y`. Прототипы методов:

```
int top() const
int y() const
```

- `right()` и `bottom()` — возвращают координаты правого нижнего угла по осям `X` и `Y` соответственно. Прототипы методов:

```
int right() const
int bottom() const
```

- `width()` и `height()` — возвращают ширину и высоту соответственно. Прототипы методов:

```
int width() const
int height() const
```

- `size()` — возвращает размеры в виде экземпляра класса `QSize`. Прототип метода:

```
QSize size() const
```

Пример:

```
QRect r(10, 15, 400, 300);
qDebug() << r.left() << r.top() << r.x() << r.y()
         << r.right() << r.bottom();
// 10 15 10 15 409 314
qDebug() << r.width() << r.height() << r.size();
// 400 300 QSize(400, 300)
```

- `topLeft()` — возвращает координаты левого верхнего угла. Прототип метода:

```
QPoint topLeft() const
```

- `topRight()` — возвращает координаты правого верхнего угла. Прототип метода:

```
QPoint topRight() const
```

- `bottomLeft()` — возвращает координаты левого нижнего угла. Прототип метода:

```
QPoint bottomLeft() const
```

- `bottomRight()` — возвращает координаты правого нижнего угла. Прототип метода:

```
QPoint bottomRight() const
```

Пример:

```
QRect r(10, 15, 400, 300);
qDebug() << r.topLeft() << r.topRight();
// QPoint(10,15) QPoint(409,15)
qDebug() << r.bottomLeft() << r.bottomRight();
// QPoint(10,314) QPoint(409,314)
```

- `center()` — возвращает координаты центра области. Прототип метода:

  ```
  QPoint center() const
  ```

  Например, вывести окно по центру доступной области экрана можно так:

  ```
  QScreen *screen = QApplication::primaryScreen();
  QRect rect = screen->availableGeometry();
  window.move(rect.center() - window.rect().center());
  ```

- `getRect()` — позволяет получить координаты левого верхнего угла и размеры области. Прототип метода:

  ```
  void getRect(int *x, int *y, int *width, int *height) const
  ```

- `getCoords()` — позволяет получить координаты левого верхнего угла и правого нижнего угла. Прототип метода:

  ```
  void getCoords(int *x1, int *y1, int *x2, int *y2) const
  ```

Прочие методы:

- `isNull()` — возвращает `true`, если ширина и высота равны нулю, и `false` — в противном случае. Прототип метода:

  ```
  bool isNull() const
  ```

- `isValid()` — возвращает `true`, если `left() <= right()` и `top() <= bottom()`, и `false` — в противном случае. Прототип метода:

  ```
  bool isValid() const
  ```

- `isEmpty()` — возвращает `true`, если `left() > right()` или `top() > bottom()`, и `false` — в противном случае. Прототип метода:

  ```
  bool isEmpty() const
  ```

- `normalized()` — исправляет ситуацию, при которой `left() > right()` или `top() > bottom()`, и возвращает новый экземпляр класса `QRect`. Прототип метода:

  ```
  QRect normalized() const
  ```

  Пример:

  ```
  QRect r(QPoint(409, 314), QPoint(10, 15));
  qDebug() << r;              // QRect(409,314 -398x-298)
  qDebug() << r.normalized(); // QRect(11,16 398x298)
  ```

- `contains()` — возвращает `true`, если точка с указанными координатами расположена внутри области или на ее границе, и `false` — в противном случае. Если

в параметре `proper` указано значение `true`, то точка должна быть расположена только внутри области, а не на ее границе. Значение параметра по умолчанию — `false`. Прототипы метода:

```
bool contains(int x, int y) const
bool contains(int x, int y, bool proper) const
bool contains(const QPoint &point, bool proper = false) const
```

Пример:

```
QRect r(0, 0, 400, 300);
qDebug() << r.contains(0, 10);       // true
qDebug() << r.contains(0, 10, true); // false
```

- `contains()` — возвращает `true`, если указанная область расположена внутри текущей области или на ее краю, и `false` — в противном случае. Если в параметре `proper` указано значение `true`, то указанная область должна быть расположена только внутри текущей области, а не на ее краю. Значение параметра по умолчанию — `false`. Прототип метода:

```
bool contains(const QRect &rectangle, bool proper = false) const
```

Пример:

```
QRect r(0, 0, 400, 300);
qDebug() << r.contains(QRect(0, 0, 20, 5));       // true
qDebug() << r.contains(QRect(0, 0, 20, 5), true); // false
```

- `intersects()` — возвращает `true`, если указанная область пересекается с текущей областью, и `false` — в противном случае. Прототип метода:

```
bool intersects(const QRect &rectangle) const
```

- `intersected()` — возвращает область, которая расположена на пересечении текущей и указанной областей. Прототип метода:

```
QRect intersected(const QRect &rectangle) const
```

Пример:

```
QRect r(0, 0, 20, 20);
qDebug() << r.intersects(QRect(10, 10, 20, 20));
// true
qDebug() << r.intersected(QRect(10, 10, 20, 20));
// QRect(10,10 10x10)
```

- `united()` — возвращает область, которая охватывает текущую и указанную области. Прототип метода:

```
QRect united(const QRect &rectangle) const
```

Пример:

```
QRect r(0, 0, 20, 20);
qDebug() << r.united(QRect(30, 30, 20, 20)); // QRect(0,0 50x50)
```

Над двумя экземплярами класса QRect определены операции & и &= (пересечение), | и |= (объединение), == и !=. Пример:

```
QRect r1(0, 0, 20, 20), r2(10, 10, 20, 20);
qDebug() << (r1 & r2);  // QRect(10,10 10x10)
qDebug() << (r1 | r2);  // QRect(0,0 30x30)
qDebug() << (r1 == r2); // false
qDebug() << (r1 != r2); // true
```

3.6. Разворачивание и сворачивание окна

В заголовке окна расположены кнопки **Свернуть** и **Развернуть**, с помощью которых можно свернуть окно в значок на **Панели задач** или максимально развернуть окно. Выполнить подобные действия из программы позволяют следующие методы класса QWidget:

- showMinimized() — сворачивает окно на **Панель задач**. Эквивалентно нажатию кнопки **Свернуть** в заголовке окна. Метод является слотом. Прототип метода:

  ```
  void showMinimized()
  ```

- showMaximized() — разворачивает окно до максимального размера. Эквивалентно нажатию кнопки **Развернуть** в заголовке окна. Метод является слотом. Прототип метода:

  ```
  void showMaximized()
  ```

- showFullScreen() — включает полноэкранный режим отображения окна. Окно отображается без заголовка и границ. Метод является слотом. Прототип метода:

  ```
  void showFullScreen()
  ```

- showNormal() — отменяет сворачивание, максимальный размер и полноэкранный режим. Метод является слотом. Прототип метода:

  ```
  void showNormal()
  ```

- activateWindow() — делает окно активным (т. е. имеющим фокус ввода). В Windows, если окно было ранее свернуто в значок на **Панель задач**, то оно автоматически не будет отображено на экране. В этом случае станет активным только значок на **Панели задач**. Прототип метода:

  ```
  void activateWindow()
  ```

- setWindowState() — изменяет статус окна в зависимости от переданных флагов. Прототип метода:

  ```
  void setWindowState(Qt::WindowStates windowState)
  ```

  В качестве параметра указывается комбинация следующих констант через побитовые операторы:

  - Qt::WindowNoState — нормальное состояние окна;
  - Qt::WindowMinimized — окно свернуто;

- `Qt::WindowMaximized` — окно максимально развернуто;
- `Qt::WindowFullScreen` — полноэкранный режим;
- `Qt::WindowActive` — окно имеет фокус ввода, т. е. является активным.

Например, включить полноэкранный режим можно так:

```
this->setWindowState((this->windowState() &
    ~(Qt::WindowMinimized | Qt::WindowMaximized))
    | Qt::WindowFullScreen);
```

Проверить статус окна позволяют следующие методы:

- `isMinimized()` — возвращает `true`, если окно свернуто, и `false` — в противном случае. Прототип метода:

```
bool isMinimized() const
```

- `isMaximized()` — возвращает `true`, если окно раскрыто до максимальных размеров, и `false` — в противном случае. Прототип метода:

```
bool isMaximized() const
```

- `isFullScreen()` — возвращает `true`, если включен полноэкранный режим, и `false` — в противном случае. Прототип метода:

```
bool isFullScreen() const
```

- `isActiveWindow()` — возвращает `true`, если окно имеет фокус ввода, и `false` — в противном случае. Прототип метода:

```
bool isActiveWindow() const
```

- `windowState()` — возвращает комбинацию флагов, обозначающих статус окна. Прототип метода:

```
Qt::WindowStates windowState() const
```

Пример проверки использования полноэкранного режима:

```
if (this->windowState() & Qt::WindowFullScreen) {
   qDebug() << "Полноэкранный режим";
}
```

Пример разворачивания и сворачивания окна приведен в листинге 3.4.

Листинг 3.4. Разворачивание и сворачивание окна

```
#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QVBoxLayout>

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   QWidget window;
```

```
    window.setWindowTitle("Разворачивание и сворачивание окна");
    window.resize(350, 200);

    QPushButton *btnMin = new QPushButton("Свернуть");
    QPushButton *btnMax = new QPushButton("Развернуть");
    QPushButton *btnFull = new QPushButton("Полный экран");
    QPushButton *btnNormal = new QPushButton("Нормальный размер");

    QVBoxLayout *vbox = new QVBoxLayout();
    vbox->addWidget(btnMin);
    vbox->addWidget(btnMax);
    vbox->addWidget(btnFull);
    vbox->addWidget(btnNormal);
    window.setLayout(vbox);

    QObject::connect(btnMin, SIGNAL(clicked()),
                     &window, SLOT(showMinimized()));
    QObject::connect(btnMax, SIGNAL(clicked()),
                     &window, SLOT(showMaximized()));
    QObject::connect(btnFull, SIGNAL(clicked()),
                     &window, SLOT(showFullScreen()));
    QObject::connect(btnNormal, SIGNAL(clicked()),
                     &window, SLOT(showNormal()));

    window.show();
    return app.exec();
}
```

3.7. Управление прозрачностью окна

Сделать окно полупрозрачным позволяет метод `setWindowOpacity()` из класса `QWidget`. Прототип метода:

```
void setWindowOpacity(qreal level)
```

В качестве параметра указывается вещественное число от `0.0` до `1.0`. Число `0.0` соответствует полностью прозрачному окну, а число `1.0` — отсутствию прозрачности. Для получения степени прозрачности окна из программы предназначен метод `windowOpacity()`, который возвращает вещественное число от `0.0` до `1.0`. Прототип метода:

```
qreal windowOpacity() const
```

Выведем окно со степенью прозрачности `0.5` (листинг 3.5).

Листинг 3.5. Полупрозрачное окно

```
#include <QApplication>
#include <QWidget>
```

```
int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   QWidget window;
   window.setWindowTitle("Полупрозрачное окно");
   window.resize(350, 100);
   window.setWindowOpacity(0.5);
   window.show();
   qDebug() << window.windowOpacity(); // 0.498039
   return app.exec();
}
```

3.8. Модальные окна

Модальным называется окно, которое не позволяет взаимодействовать с другими окнами в том же приложении. Пока модальное окно не будет закрыто, сделать активным другое окно нельзя. Например, если в программе Microsoft Word выбрать пункт меню **Файл | Сохранить как**, то откроется модальное диалоговое окно, позволяющее выбрать путь и название файла. Пока это окно не будет закрыто, вы не сможете взаимодействовать с главным окном приложения.

Указать, что окно является модальным, позволяет метод `setWindowModality()` из класса `QWidget`. Прототип метода:

```
void setWindowModality(Qt::WindowModality windowModality)
```

В качестве параметра могут быть указаны следующие константы:

- `Qt::NonModal` — окно не является модальным;
- `Qt::WindowModal` — окно блокирует только родительские окна в пределах иерархии;
- `Qt::ApplicationModal` — окно блокирует все окна в приложении.

Окна, открытые из модального окна, не блокируются. Следует также учитывать, что метод `setWindowModality()` должен быть вызван до отображения окна.

Получить текущее значение позволяет метод `windowModality()`. Проверить, является ли окно модальным, можно с помощью метода `isModal()`. Метод возвращает `true`, если окно является модальным, и `false` — в противном случае. Прототипы методов:

```
Qt::WindowModality windowModality() const
bool isModal() const
```

Создадим два независимых окна. В первом окне разместим кнопку, при нажатии которой откроем модальное окно. Это модальное окно будет блокировать только первое окно, но не второе. При открытии модального окна отобразим его примерно по центру родительского окна. Содержимое файла widget.h приведено в листинге 3.6, файла widget.cpp — в листинге 3.7, а файла main.cpp — в листинге 3.8.

Листинг 3.6. Содержимое файла widget.h

```
#include <QPushButton>
#include <QVBoxLayout>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent=nullptr);
   ~Widget();
private slots:
   void on_btn1_clicked();
private:
   QPushButton *btn1;
   QVBoxLayout *vbox;
};
#endif // WIDGET_H
```

Листинг 3.7. Содержимое файла widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   btn1 = new QPushButton("Открыть модальное окно");
   vbox = new QVBoxLayout();
   vbox->addWidget(btn1);
   setLayout(vbox);
   connect(btn1, SIGNAL(clicked()),
           this, SLOT(on_btn1_clicked()));
}

void Widget::on_btn1_clicked()
{
   QWidget *w = new QWidget(this, Qt::Window);
   w->setWindowTitle("Модальное окно");
   w->resize(300, 50);
   w->setWindowModality(Qt::WindowModal);
   w->setAttribute(Qt::WA_DeleteOnClose, true);
   w->move(frameGeometry().center() - w->rect().center());
   w->show();
}

Widget::~Widget() {}
```

Листинг 3.8. Содержимое файла main.cpp

```
#include "widget.h"

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);

   Widget window;
   window.setWindowTitle("Обычное окно");
   window.resize(350, 100);
   window.show();

   QWidget window2;
   window2.setWindowTitle(
            "Это окно не будет блокировано при WindowModal");
   window2.resize(500, 100);
   window2.show();
   return app.exec();
}
```

Если запустить приложение и нажать кнопку **Открыть модальное окно**, то откроется окно, выровненное примерно (произвести точное выравнивание вы сможете самостоятельно) по центру родительского окна. При этом получить доступ к родительскому окну можно только при закрытии модального окна, а второе окно блокировано не будет. Если заменить константу `Qt::WindowModal` константой `Qt::ApplicationModal`, то оба окна будут блокированы.

Обратите внимание на то, что в конструктор модального окна мы передали указатель на первое окно и константу `Qt::Window`. Если указатель не передать, то окно блокировано не будет, а если константу не указать, то окно вообще не откроется. Чтобы объект окна автоматически удалялся при закрытии окна, константе `Qt::WA_DeleteOnClose` в методе `setAttribute()` было присвоено значение `true`.

Модальные окна в большинстве случаев являются диалоговыми. Для работы с диалоговыми окнами в Qt предназначен класс `QDialog`, который автоматически выравнивает окно по центру экрана или по центру родительского окна. Кроме того, этот класс предоставляет множество специальных методов, позволяющих дождаться закрытия окна, определить статус завершения и многое другое. Подробно класс `QDialog` мы будем изучать в отдельной главе.

3.9. Смена значка в заголовке окна

По умолчанию в левом верхнем углу окна отображается стандартный значок. Отобразить другой значок в заголовке окна позволяет метод `setWindowIcon()` из класса `QWidget`. В качестве параметра метод принимает экземпляр класса `QIcon`. Прототип метода:

```
void setWindowIcon(const QIcon &icon)
```

Чтобы загрузить значок из файла, следует передать путь к файлу конструктору класса. Если указан относительный путь, то поиск файла будет производиться относительно текущего рабочего каталога. Получить список поддерживаемых форматов файлов можно с помощью статического метода `supportedImageFormats()` из класса `QImageReader`. Метод возвращает список с экземплярами класса `QByteArray`. Прототип метода:

```
#include <QImageReader>
static QList<QByteArray> supportedImageFormats()
```

Получим список поддерживаемых форматов:

```
qDebug() << QImageReader::supportedImageFormats();
```

Результат выполнения на моем компьютере:

```
QList("bmp", "cur", "gif", "ico", "jpeg", "jpg", "pbm", "pgm", "png",
"ppm", "svg", "svgz", "xbm", "xpm")
```

Если для окна не задан значок, то будет использоваться значок приложения, установленный с помощью метода `setWindowIcon()` из класса `QApplication`. В качестве параметра метод принимает экземпляр класса `QIcon`. Прототип метода:

```
void setWindowIcon(const QIcon &icon)
```

Вместо загрузки значка из файла можно воспользоваться одним из встроенных значков. Загрузить стандартный значок позволяет следующий код:

```
// #include <QIcon>
// #include <QStyle>
QIcon ico = window.style()->standardIcon(
               QStyle::SP_MessageBoxCritical);
window.setWindowIcon(ico);
```

Посмотреть список всех встроенных значков можно в документации к классу `QStyle`.

В качестве примера создаем значок в формате ICO и сохраняем его в одной папке с программой, а далее устанавливаем этот значок для окна и для всего приложения (листинг 3.9).

Листинг 3.9. Смена значка в заголовке окна

```
#include <QApplication>
#include <QWidget>
#include <QIcon>

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   QWidget window;
   window.setWindowTitle("Смена значка в заголовке окна");
```

```
    window.resize(350, 70);
    QIcon ico("C:\\cpp\\projectsQt\\Test\\test.ico");
    window.setWindowIcon(ico); // Значок для окна
    app.setWindowIcon(ico);    // Значок для приложения
    window.show();
    return app.exec();
}
```

3.10. Изменение цвета фона окна

Чтобы изменить цвет фона окна (или компонента), следует установить палитру с настроенной ролью `Window`. Цветовая палитра содержит цвета для каждой роли и состояния компонента. Указать состояние компонента позволяют следующие константы из класса `QPalette`:

- `Active` и `Normal` — компонент активен (окно находится в фокусе ввода);
- `Disabled` — компонент недоступен;
- `Inactive` — компонент неактивен (окно находится вне фокуса ввода).

Получить текущую палитру компонента позволяет метод `palette()`. Прототип метода:

```
const QPalette &palette() const
```

Чтобы изменить цвет для какой-либо роли и состояния, следует воспользоваться методом `setColor()` из класса `QPalette`. Прототипы метода:

```
#include <QPalette>
void setColor(QPalette::ColorRole role, const QColor &color)
void setColor(QPalette::ColorGroup group, QPalette::ColorRole role,
              const QColor &color)
```

В параметре `role` указывается, для какого элемента изменяется цвет. Например, константа `Window` изменяет цвет фона, а `WindowText` — цвет текста. Полный список констант смотрите в документации по классу `QPalette`.

В параметре `color` указывается цвет элемента. В качестве значения можно указать константы (например: `Qt::black`, `Qt::red` и т. д.) или экземпляр класса `QColor` (например: `QColor("red")`, `QColor(255, 0, 0)` и др.).

После настройки палитры необходимо вызвать метод `setPalette()` и передать ему измененный объект палитры. Прототип метода:

```
void setPalette(const QPalette &)
```

Следует помнить, что компоненты-потомки по умолчанию имеют прозрачный фон и не перерисовываются автоматически. Чтобы включить перерисовку, необходимо передать значение `true` методу `setAutoFillBackground()`. Прототип метода:

```
void setAutoFillBackground(bool enabled)
```

Изменить цвет фона можно также с помощью CSS-атрибута `background-color`. Для этого следует передать таблицу стилей в метод `setStyleSheet()`. Прототип метода:

```
void setStyleSheet(const QString &styleSheet)
```

Таблицы стилей могут быть внешними (подключение через командную строку), установленными на уровне приложения (с помощью метода `setStyleSheet()` из класса `QApplication`) или установленными на уровне компонента (с помощью метода `setStyleSheet()` из класса `QWidget`). Атрибуты, установленные последними, обычно перекрывают значения аналогичных атрибутов, указанных ранее. Если вы занимались веб-программированием, то CSS вам уже знаком, а если нет, то придется дополнительно изучить HTML и CSS.

Создадим окно с надписью. Для активного окна установим зеленый цвет, а для неактивного — красный. Цвет фона надписи сделаем белым. Для изменения фона окна будем устанавливать палитру, а для изменения цвета фона надписи — CSS-атрибут `background-color` (листинг 3.10).

Листинг 3.10. Изменение цвета фона окна

```
#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QVBoxLayout>
#include <QPalette>
#include <QColor>

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   QWidget window;
   window.setWindowTitle("Изменение цвета фона окна");
   window.resize(350, 70);
   QPalette pal = window.palette();
   pal.setColor(QPalette::Normal, QPalette::Window,
                QColor(0, 128, 0));
   pal.setColor(QPalette::Inactive, QPalette::Window,
                QColor(255, 0, 0));
   window.setPalette(pal);

   QLabel *label = new QLabel("Текст надписи");
   label->setAlignment(Qt::AlignCenter);
   label->setStyleSheet("background-color: #ffffff;");
   label->setAutoFillBackground(true);
   QVBoxLayout *vbox = new QVBoxLayout;
   vbox->addWidget(label);
   window.setLayout(vbox);
```

```
    window.show();
    return app.exec();
}
```

3.11. Использование изображения в качестве фона

В качестве фона окна (или компонента) можно использовать изображение. Для этого необходимо получить текущую палитру компонента с помощью метода `palette()`, а затем вызвать метод `setBrush()` из класса `QPalette`. Прототипы метода:

```
void setBrush(QPalette::ColorRole role, const QBrush &brush)
void setBrush(QPalette::ColorGroup group, QPalette::ColorRole role,
              const QBrush &brush)
```

Параметры `group` и `role` аналогичны соответствующим параметрам в методе `setColor()`, который мы рассматривали в предыдущем разделе. В параметре `brush` указывается экземпляр класса `QBrush`. Прототипы конструктора класса:

```
#include <QBrush>
QBrush()
QBrush(Qt::BrushStyle style)
QBrush(Qt::GlobalColor color, Qt::BrushStyle style = Qt::SolidPattern)
QBrush(const QColor &color, Qt::BrushStyle style = Qt::SolidPattern)
QBrush(Qt::GlobalColor color, const QPixmap &pixmap)
QBrush(const QColor &color, const QPixmap &pixmap)
QBrush(const QPixmap &pixmap)
QBrush(const QImage &image)
QBrush(const QGradient &gradient)
QBrush(const QBrush &other)
```

В параметре `style` указываются константы, задающие стиль кисти, например: `Qt::NoBrush`, `Qt::SolidPattern`, `Qt::Dense1Pattern`, `Qt::Dense2Pattern`, `Qt::Dense3Pattern`, `Qt::Dense4Pattern`, `Qt::Dense5Pattern`, `Qt::Dense6Pattern`, `Qt::Dense7Pattern`, `Qt::CrossPattern` и др. С помощью этого параметра можно сделать цвет сплошным (`Qt::SolidPattern`) или имеющим текстуру (например, константа `Qt::CrossPattern` задает текстуру в виде сетки).

В параметре `color` указывается цвет кисти. В качестве значения можно указать константы (например: `Qt::black`, `Qt::red` и т. д.) или экземпляр класса `QColor` (например: `QColor("red")`, `QColor(255, 0, 0)` и др.). Например, установка сплошного цвета фона окна выглядит так:

```
QPalette pal = window.palette();
pal.setBrush(QPalette::Normal, QPalette::Window,
             QBrush(QColor(0, 128, 0), Qt::SolidPattern));
window.setPalette(pal);
```

Параметры `pixmap` и `image` позволяют передать объекты изображений. Конструкторы этих классов принимают путь к файлу, который может быть как абсолютным, так и относительным. Основные прототипы конструкторов:

```
QPixmap(const QString &fileName, const char *format = nullptr,
        Qt::ImageConversionFlags flags = Qt::AutoColor)
QImage(const QString &fileName, const char *format = nullptr)
```

После настройки палитры необходимо вызвать метод `setPalette()` и передать ему измененный объект палитры. Следует помнить, что компоненты-потомки по умолчанию имеют прозрачный фон и не перерисовываются автоматически. Чтобы включить перерисовку, необходимо передать значение `true` в метод `setAutoFillBackground()`.

Указать, что изображение используется в качестве фона, можно также с помощью CSS-атрибутов `background` и `background-image`. С помощью CSS-атрибута `background-repeat` можно дополнительно указать режим повтора фонового рисунка. Он может принимать значения `repeat`, `repeat-x` (повтор только по горизонтали), `repeat-y` (повтор только по вертикали) и `no-repeat` (не повторяется).

Создадим окно с надписью. Для активного окна установим одно изображение (с помощью изменения палитры), а для надписи — другое изображение (с помощью CSS-атрибута `background-image`) (листинг 3.11).

Листинг 3.11. Использование изображения в качестве фона

```
#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QVBoxLayout>
#include <QPalette>
#include <QPixmap>
#include <QBrush>

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   QWidget window;
   window.setWindowTitle("Изображение в качестве фона");
   window.resize(350, 70);

   QPalette pal = window.palette();
   pal.setBrush(QPalette::Normal, QPalette::Window,
       QBrush(QPixmap("C:\\cpp\\projectsQt\\Test\\texture2.jpg")));
   window.setPalette(pal);

   QLabel *label = new QLabel("Текст надписи");
   label->setAlignment(Qt::AlignCenter);
```

```
    label->setStyleSheet(
     "background-image: url(C:/cpp/projectsQt/Test/texture.jpg);");
    label->setAutoFillBackground(true);
    QVBoxLayout *vbox = new QVBoxLayout;
    vbox->addWidget(label);
    window.setLayout(vbox);

    window.show();
    return app.exec();
}
```

3.12. Создание окна произвольной формы

Чтобы создать окно произвольной формы, необходимо выполнить следующие шаги:

- создать изображение нужной формы с прозрачным фоном и сохранить его, например, в формате PNG;
- создать экземпляр класса `QPixmap`, передав конструктору класса абсолютный или относительный путь к изображению;
- установить изображение в качестве фона окна с помощью палитры;
- отделить альфа-канал с помощью метода `mask()` из класса `QPixmap`. Прототип метода:

  ```
  QBitmap mask() const
  ```

- передать маску в метод `setMask()` объекта окна. Прототип метода:

  ```
  void setMask(const QBitmap &bitmap)
  ```

- убрать рамку окна, например передав комбинацию следующих флагов:

  ```
  Qt::Window | Qt::FramelessWindowHint
  ```

Если для создания окна используется класс `QLabel`, то вместо установки палитры можно передать экземпляр класса `QPixmap` в метод `setPixmap()`, а маску — в метод `setMask()`. Прототипы методов:

```
void setPixmap(const QPixmap &)
void setMask(const QBitmap &bitmap)
```

В качестве примера создадим круглое окно с кнопкой, с помощью которой можно закрыть окно. Окно выведем без заголовка и границ (листинг 3.12).

Листинг 3.12. Создание окна произвольной формы

```
#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QPalette>
```

```
#include <QPixmap>
#include <QBrush>

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   QWidget window;
   window.setWindowTitle("Создание окна произвольной формы");
   window.setWindowFlags(Qt::Window | Qt::FramelessWindowHint);
   window.resize(300, 300);

   QPixmap pixmap("C:\\cpp\\projectsQt\\Test\\fon.png");
   QPalette pal = window.palette();
   pal.setBrush(QPalette::Normal, QPalette::Window,
                QBrush(pixmap));
   pal.setBrush(QPalette::Inactive, QPalette::Window,
                QBrush(pixmap));
   window.setPalette(pal);
   window.setMask(pixmap.mask());

   QPushButton *button = new QPushButton("Закрыть окно", &window);
   button->setFixedSize(150, 30);
   button->move(75, 135);
   QObject::connect(button, SIGNAL(clicked()),
                    qApp, SLOT(quit()));
   window.show();
   return app.exec();
}
```

3.13. Всплывающие подсказки

При работе с программой у пользователя могут возникать вопросы: для чего предназначен тот или иной компонент? Обычно для информирования пользователя используются надписи, расположенные над компонентом или перед ним. Но часто место в окне либо ограничено, либо вывод таких надписей испортит весь дизайн окна. В таких случаях принято выводить текст подсказки в отдельном окне без рамки при наведении указателя мыши на компонент. После выведения указателя окно должно автоматически закрываться.

В Qt нет необходимости создавать окно с подсказкой самому и следить за перемещениями указателя мыши. Весь процесс автоматизирован и максимально упрощен. Чтобы создать всплывающие подсказки для окна или любого другого компонента и управлять ими, нужно воспользоваться следующими методами из класса `QWidget`:

- `setToolTip()` — задает текст вплывающей подсказки. В качестве параметра можно указать простой текст или текст в формате HTML. Чтобы отключить

вывод подсказки, достаточно передать в этот метод пустую строку. Прототип метода:

```
void setToolTip(const QString &)
```

- `toolTip()` — возвращает текст всплывающей подсказки. Прототип метода:

```
QString toolTip() const
```

- `setWhatsThis()` — задает текст справки. Обычно этот метод используется для вывода информации большего объема, чем во всплывающей подсказке. Чтобы отобразить текст справки, необходимо сделать компонент активным и нажать комбинацию клавиш <Shift>+<F1>. Кроме того, в заголовке окна может быть кнопка **Справка**. После нажатия этой кнопки вид курсора изменится на стрелку со знаком вопроса. Чтобы в этом случае отобразить текст справки, следует щелкнуть мышью на компоненте. В качестве параметра можно указать простой текст или текст в формате HTML. Чтобы отключить вывод подсказки, достаточно передать в этот метод пустую строку. Прототип метода:

```
void setWhatsThis(const QString &)
```

- `whatsThis()` — возвращает текст справки. Прототип метода:

```
QString whatsThis() const
```

Создадим окно с кнопкой и зададим для них текст всплывающих подсказок и текст справки (листинг 3.13).

Листинг 3.13. Всплывающие подсказки

```
#include <QApplication>
#include <QWidget>
#include <QPushButton>

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   QWidget window;
   window.setWindowTitle("Всплывающие подсказки");
   window.setWindowFlags(Qt::Window |
                         Qt::WindowCloseButtonHint |
                         Qt::WindowContextHelpButtonHint);
   window.resize(300, 70);

   QPushButton *button = new QPushButton("Закрыть окно", &window);
   button->setFixedSize(150, 30);
   button->move(75, 20);

   button->setToolTip("Это всплывающая подсказка для кнопки");
   window.setToolTip("Это всплывающая подсказка для окна");
   button->setWhatsThis("Это справка для кнопки");
   window.setWhatsThis("Это справка для окна");
```

```
    QObject::connect(button, SIGNAL(clicked()), qApp, SLOT(quit()));
    window.show();
    return app.exec();
}
```

3.14. Закрытие окна из программы

В предыдущих разделах для закрытия окна мы использовали слот `quit()` объекта приложения. Прототип метода:

```
void quit()
```

Однако этот метод не только закрывает текущее окно, но и завершает выполнение всего приложения. Чтобы закрыть только текущее окно, следует воспользоваться слотом `close()` из класса `QWidget`. Прототип метода:

```
bool close()
```

Метод возвращает значение `true`, если окно успешно закрыто, и `false` — в противном случае. Закрыть сразу все окна приложения позволяет слот `closeAllWindows()` из класса `QApplication`. Прототип метода:

```
void closeAllWindows()
```

Если для окна константа `Qt::WA_DeleteOnClose` установлена в истинное значение, то после закрытия окна объект окна будет автоматически удален. Если константа имеет ложное значение, то окно просто скрывается. Значение можно изменить с помощью метода `setAttribute()`:

```
w->setAttribute(Qt::WA_DeleteOnClose, true);
```

После вызова метода `close()` или нажатия кнопки **Закрыть** в заголовке окна генерируется событие `QEvent::Close`. Если внутри класса определить метод с предопределенным названием `closeEvent()`, то это событие можно перехватить и обработать. Прототип метода:

```
virtual void closeEvent(QCloseEvent *event)
```

В качестве параметра метод принимает указатель на объект класса `QCloseEvent`, который содержит методы `accept()` (позволяет закрыть окно) и `ignore()` (запрещает закрытие окна). Вызывая эти методы, можно контролировать процесс закрытия окна. Прототипы методов:

```
void accept()
void ignore()
```

В качестве примера закроем окно при нажатии кнопки (листинг 3.14).

Листинг 3.14. Закрытие окна из программы

```
#include <QApplication>
#include <QWidget>
#include <QPushButton>
```

```
int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   QWidget window;
   window.setWindowTitle("Закрытие окна из программы");
   window.resize(350, 70);

   QPushButton *button = new QPushButton("Закрыть окно", &window);
   button->setFixedSize(150, 30);
   button->move(75, 20);
   QObject::connect(button, SIGNAL(clicked()),
                    &window, SLOT(close()));
   window.show();
   return app.exec();
}
```

ГЛАВА 4

Обработка сигналов и событий

При взаимодействии пользователя с окном происходят *события*. В ответ на события система генерирует определенные сигналы. *Сигналы* — это своего рода извещения системы о том, что пользователь выполнил какое-либо действие или в самой системе возникло некоторое условие. Сигналы являются важнейшей составляющей приложения с графическим интерфейсом, поэтому необходимо знать, как назначить обработчик сигнала, как удалить обработчик, а также уметь правильно обработать событие. Какие сигналы генерирует тот или иной компонент, мы будем рассматривать при изучении конкретного компонента.

4.1. Назначение обработчиков сигналов

Чтобы обработать какой-либо сигнал, необходимо сопоставить ему метод класса, который будет вызван при наступлении события. Назначить обработчик позволяет статический метод `connect()` из класса `QObject`. Прототипы метода:

```
static QMetaObject::Connection connect(const QObject *sender,
   const char *signal, const QObject *receiver, const char *method,
   Qt::ConnectionType type = Qt::AutoConnection)
static QMetaObject::Connection connect(const QObject *sender,
   const QMetaMethod &signal, const QObject *receiver,
   const QMetaMethod &method,
   Qt::ConnectionType type = Qt::AutoConnection)
static QMetaObject::Connection connect(const QObject *sender,
   PointerToMemberFunction signal, const QObject *receiver,
   PointerToMemberFunction method,
   Qt::ConnectionType type = Qt::AutoConnection)
static QMetaObject::Connection connect(const QObject *sender,
   PointerToMemberFunction signal, Functor functor)
static QMetaObject::Connection connect(const QObject *sender,
   PointerToMemberFunction signal, const QObject *context,
   Functor functor, Qt::ConnectionType type = Qt::AutoConnection)
```

Кроме того, существует обычный (нестатический) метод `connect()`:

```
QMetaObject::Connection connect(const QObject *sender,
   const char *signal, const char *method,
   Qt::ConnectionType type = Qt::AutoConnection) const
```

Наиболее часто используются следующие форматы:

```
connect(<Объект1>, <Сигнал>, <Объект2>, <Слот>)
connect(<Объект1>, <Сигнал>, <Слот>)
```

При нажатии кнопки закроем окно:

```
QPushButton *button = new QPushButton("Закрыть окно", &window);
QObject::connect(button, SIGNAL(clicked()),
                 &window, SLOT(close()));
```

Метод позволяет назначить обработчик сигнала `<Сигнал>`, возникшего при изменении статуса объекта `<Объект1>`. В первом параметре передается указатель на объект `<Объект1>`. В параметре `<Сигнал>` указывается результат выполнения макроса `SIGNAL()`. Формат макроса:

```
SIGNAL(<Название сигнала>([Тип параметров]))
```

Каждый компонент имеет определенный набор сигналов, например при щелчке по кнопке генерируется сигнал, имеющий следующий формат:

```
void clicked(bool checked = false)
```

Внутри круглых скобок могут быть указаны типы параметров, которые передаются в обработчик. Если параметров нет, то указываются только круглые скобки. Пример указания сигнала без параметров:

```
SIGNAL(clicked())
```

В этом случае обработчик не принимает никаких параметров. Указание сигнала с параметром выглядит следующим образом:

```
SIGNAL(clicked(bool))
```

В этом случае обработчик должен принимать один параметр, значение которого всегда будет равно `false`, т. к. это значение по умолчанию для сигнала `clicked()`.

В параметре `<Объект2>` передается указатель на объект, метод которого должен быть вызван. В параметре `<Сигнал>` указывается результат выполнения макроса `SLOT()`. Формат макроса:

```
SLOT(<Название слота>([Тип параметров]))
```

Пример слота без параметров:

```
SLOT(close())
```

Пример слота с одним параметром:

```
SLOT(on_btn1_clicked(bool))
```

Пользовательские слоты должны объявляться внутри объявления класса в секции:

```
<Модификатор доступа> slots:
```

Пример объявления закрытого слота внутри класса:

```
private slots:
   void on_btn1_clicked();
```

Создадим окно с двумя кнопками. Для кнопок назначим обработчики нажатия разными способами. Содержимое файла widget.h приведено в листинге 4.1, файла widget.cpp — в листинге 4.2, а файла main.cpp — в листинге 4.3.

Листинг 4.1. Содержимое файла widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QVBoxLayout>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent=nullptr);
   ~Widget();
private slots:
   void on_btn1_clicked();
   void on_btn2_clicked(bool);
private:
   QPushButton *btn1;
   QPushButton *btn2;
   QVBoxLayout *vbox;
};
#endif // WIDGET_H
```

Листинг 4.2. Содержимое файла widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   btn1 = new QPushButton("Кнопка 1");
   btn2 = new QPushButton("Кнопка 2");
   vbox = new QVBoxLayout();
   vbox->addWidget(btn1);
   vbox->addWidget(btn2);
```

```
   setLayout(vbox);
   connect(btn1, SIGNAL(clicked()),
           this, SLOT(on_btn1_clicked()));
   connect(btn2, SIGNAL(clicked(bool)),
           SLOT(on_btn2_clicked(bool)));
}

void Widget::on_btn1_clicked()
{
   qDebug() << "Нажата кнопка 1";
}

void Widget::on_btn2_clicked(bool a)
{
   qDebug() << "Нажата кнопка 2" << a;
}

Widget::~Widget() {}
```

Листинг 4.3. Содержимое файла main.cpp

```
#include "widget.h"

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   Widget window;
   window.setWindowTitle("Назначение обработчиков сигналов");
   window.resize(350, 120);
   window.show();
   return app.exec();
}
```

Назначение обработчика для первой кнопки выглядит так:

```
connect(btn1, SIGNAL(clicked()),
        this, SLOT(on_btn1_clicked()));
```

Здесь мы воспользовались статическим методом `connect()`, хотя и получили доступ к нему, как к обычному методу класса. Класс `QWidget` наследует класс `QObject`, поэтому такая запись возможна. Иначе нужно написать так:

```
QObject::connect(btn1, SIGNAL(clicked()),
                 this, SLOT(on_btn1_clicked()));
```

Можно также назначить обработчик следующим образом:

```
QObject::connect(btn1, &QPushButton::clicked,
                 this, &Widget::on_btn1_clicked);
```

При назначении обработчика нажатия второй кнопки мы опустили передачу указателя `this`:

```
connect(btn2, SIGNAL(clicked(bool)),
        SLOT(on_btn2_clicked(bool)));
```

В качестве обработчика может выступать обычная функция или лямбда-выражение. Пример приведен в листинге 4.4.

Листинг 4.4. Содержимое файла main.cpp

```
#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QVBoxLayout>

void on_clicked() {
   qDebug() << "Нажата кнопка 1. on_clicked()";
}
void on_clicked2() {
   qDebug() << "Нажата кнопка 1. on_clicked2()";
}

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   QWidget window;
   window.setWindowTitle("Назначение обработчиков сигналов");
   window.resize(350, 70);

   QPushButton *btn1 = new QPushButton("Кнопка 1");
   QPushButton *btn2 = new QPushButton("Кнопка 2");
   QVBoxLayout *vbox = new QVBoxLayout();
   vbox->addWidget(btn1);
   vbox->addWidget(btn2);
   window.setLayout(vbox);
   QObject::connect(btn1, &QPushButton::clicked,
                    on_clicked);
   QObject::connect(btn1, &QPushButton::clicked,
                    on_clicked2);
   QObject::connect(btn2, &QPushButton::clicked,
      [=]() {
         qDebug() << "Нажата кнопка 2";
      });
   window.show();
   return app.exec();
}
```

Обратите внимание: для одного сигнала можно назначить несколько обработчиков, которые будут вызываться в порядке назначения в программе.

Можно также связать один сигнал с другим сигналом. При нажатии первой кнопки сгенерируем сигнал нажатия второй кнопки и обработаем его (листинг 4.5).

Листинг 4.5. Содержимое файла main.cpp

```
#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QVBoxLayout>

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   QWidget window;
   window.setWindowTitle("Назначение обработчиков сигналов");
   window.resize(350, 100);

   QPushButton *btn1 = new QPushButton("Кнопка 1");
   QPushButton *btn2 = new QPushButton("Кнопка 2");
   QVBoxLayout *vbox = new QVBoxLayout();
   vbox->addWidget(btn1);
   vbox->addWidget(btn2);
   window.setLayout(vbox);
   QObject::connect(btn1, SIGNAL(clicked()),
                    btn2, SIGNAL(clicked()));
   QObject::connect(btn2, &QPushButton::clicked,
      [=]() {
         qDebug() << "Нажата кнопка 2";
      });
   window.show();
   return app.exec();
}
```

Необязательный параметр `type` определяет тип соединения между сигналом и обработчиком. На этот параметр следует обратить особое внимание при использовании нескольких потоков в приложении, т. к. изменять GUI-поток из другого потока нельзя. В параметре можно указать одну из следующих констант:

- `Qt::AutoConnection` — значение по умолчанию. Если источник сигнала и обработчик находятся в одном потоке, то эквивалентно значению `Qt::DirectConnection`, а если в разных потоках, то — `Qt::QueuedConnection`;
- `Qt::DirectConnection` — обработчик вызывается сразу после генерации сигнала. Обработчик выполняется в потоке источника сигнала;
- `Qt::QueuedConnection` — сигнал помещается в очередь обработки событий. Обработчик выполняется в потоке приемника сигнала;

- `Qt::BlockingQueuedConnection` — аналогично значению `Qt::QueuedConnection`, но, пока сигнал не обработан, поток будет заблокирован. Обратите внимание на то, что источник сигнала и обработчик должны быть обязательно расположены в разных потоках;
- `Qt::UniqueConnection` — обработчик можно назначить, только если он не был назначен ранее;
- `Qt::SingleShotConnection` — обработчик будет вызываться только один раз. Пример:

```
QObject::connect(btn1, SIGNAL(clicked()),
                 this, SLOT(on_btn1_clicked()),
                 Qt::SingleShotConnection);
```

4.2. Блокировка и удаление обработчика

Для блокировки и удаления обработчиков предназначены следующие методы из класса `QObject`:

- `blockSignals()` — временно блокирует прием сигналов, если параметр имеет значение `true`, и снимает блокировку, если параметр имеет значение `false`. Метод возвращает логическое представление предыдущего состояния соединения. Прототип метода:

```
bool blockSignals(bool block)
```

- `signalsBlocked()` — метод возвращает значение `true`, если блокировка установлена, и `false` — в противном случае. Прототип метода:

```
bool signalsBlocked() const
```

- `disconnect()` — удаляет обработчик. Метод является статическим и доступен без создания экземпляра класса. Прототипы метода:

```
static bool disconnect(const QObject *sender, const char *signal,
   const QObject *receiver, const char *method)
static bool disconnect(const QObject *sender,
   const QMetaMethod &signal, const QObject *receiver,
   const QMetaMethod &method)
static bool disconnect(const QMetaObject::Connection &connection)
static bool disconnect(const QObject *sender,
   PointerToMemberFunction signal, const QObject *receiver,
   PointerToMemberFunction method)
```

Существует также нестатический метод `disconnect()`. Прототипы метода:

```
bool disconnect(const char *signal = nullptr,
   const QObject *receiver = nullptr,
   const char *method = nullptr) const
bool disconnect(const QObject *receiver,
   const char *method = nullptr) const
```

Если обработчик успешно удален, то метод `disconnect()` возвращает значение `true`. Значения, указанные в методе `disconnect()`, должны совпадать со значениями, используемыми при назначении обработчика. Например, если обработчик назначался таким образом:

```
connect(btn1, SIGNAL(clicked()),
        this, SLOT(on_btn1_clicked()));
```

то удалить его можно следующим образом:

```
disconnect(btn1, SIGNAL(clicked()),
           this, SLOT(on_btn1_clicked()));
```

Создадим окно с четырьмя кнопками. Для кнопки **Нажми меня** назначим обработчик для сигнала `clicked()`. Чтобы информировать о нажатии кнопки, выведем сообщение в окно консоли. Для кнопок **Блокировать**, **Разблокировать** и **Удалить обработчик** создадим обработчики, которые будут изменять статус обработчика для кнопки **Нажми меня**. Содержимое файла widget.h приведено в листинге 4.6, файла widget.cpp — в листинге 4.7, а файла main.cpp — в листинге 4.8.

Листинг 4.6. Содержимое файла widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QVBoxLayout>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent=nullptr);
   ~Widget();
private slots:
   void on_btn1_clicked();
   void on_btn2_clicked();
   void on_btn3_clicked();
   void on_btn4_clicked();
private:
   QPushButton *btn1;
   QPushButton *btn2;
   QPushButton *btn3;
   QPushButton *btn4;
   QVBoxLayout *vbox;
};
#endif // WIDGET_H
```

Листинг 4.7. Содержимое файла widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   btn1 = new QPushButton("Нажми меня");
   btn2 = new QPushButton("Блокировать");
   btn3 = new QPushButton("Разблокировать");
   btn4 = new QPushButton("Удалить обработчик");
   btn3->setEnabled(false);
   vbox = new QVBoxLayout();
   vbox->addWidget(btn1);
   vbox->addWidget(btn2);
   vbox->addWidget(btn3);
   vbox->addWidget(btn4);
   setLayout(vbox);
   connect(btn1, SIGNAL(clicked()),
           this, SLOT(on_btn1_clicked()));
   connect(btn2, SIGNAL(clicked()),
           this, SLOT(on_btn2_clicked()));
   connect(btn3, SIGNAL(clicked()),
           this, SLOT(on_btn3_clicked()));
   connect(btn4, SIGNAL(clicked()),
           this, SLOT(on_btn4_clicked()));
}

void Widget::on_btn1_clicked()
{
   qDebug() << "Нажата кнопка 1";
}

void Widget::on_btn2_clicked()
{
   btn1->blockSignals(true);
   btn2->setEnabled(false);
   btn3->setEnabled(true);
}

void Widget::on_btn3_clicked()
{
   btn1->blockSignals(false);
   btn2->setEnabled(true);
   btn3->setEnabled(false);
}
```

```
void Widget::on_btn4_clicked()
{
   disconnect(btn1, SIGNAL(clicked()),
              this, SLOT(on_btn1_clicked()));
   btn2->setEnabled(false);
   btn3->setEnabled(false);
   btn4->setEnabled(false);
}
Widget::~Widget() {}
```

Листинг 4.8. Содержимое файла main.cpp

```
#include "widget.h"

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   Widget window;
   window.setWindowTitle("Блокировка и удаление обработчика");
   window.resize(350, 150);
   window.show();
   return app.exec();
}
```

Если после отображения окна нажать кнопку **Нажми меня**, то в окно консоли будет выведена строка `Нажата кнопка 1`. Нажатие кнопки **Блокировать** производит блокировку обработчика. Теперь при нажатии кнопки **Нажми меня** никаких сообщений в окно консоли не выводится. Отменить блокировку можно с помощью кнопки **Разблокировать**. Нажатие кнопки **Удалить обработчик** производит полное удаление обработчика. В этом случае, чтобы обрабатывать нажатие кнопки **Нажми меня**, необходимо заново назначить обработчик.

Отключить генерацию сигнала можно также, сделав компонент недоступным с помощью следующих методов из класса `QWidget`:

- `setEnabled()` — если в параметре указано значение `false`, то компонент станет недоступным. Чтобы сделать компонент опять доступным, следует передать значение `true`. Метод является слотом. Прототип метода:

  ```
  void setEnabled(bool)
  ```

- `setDisabled()` — если в параметре указано значение `true`, то компонент станет недоступным. Чтобы сделать компонент опять доступным, следует передать значение `false`. Метод является слотом. Прототип метода:

  ```
  void setDisabled(bool)
  ```

Проверить, доступен компонент или нет, позволяет метод `isEnabled()`. Метод возвращает значение `true`, если компонент доступен, и `false` — в противном случае. Прототип метода:

```
bool isEnabled() const
```

4.3. Генерация сигнала из программы

В некоторых случаях необходимо вызвать сигнал из программы. Например, при заполнении последнего текстового поля и нажатии клавиши <Enter> можно имитировать нажатие кнопки и тем самым запустить обработчик этого сигнала. Выполнить генерацию сигнала из программы позволяет инструкция `emit`. Формат инструкции:

```
emit [Объект->]<Сигнал>([<Данные через запятую>])
```

Пользовательские сигналы должны объявляться внутри секцией `signals` в объявлении класса. Методы сигналов никогда не возвращают значения и не требуют создания определения. Пример:

```
signals:
   void mysignal(int, int);
```

В качестве примера создадим окно с двумя кнопками. Для этих кнопок назначим обработчики сигнала `clicked()` (нажатие кнопки). Внутри обработчика щелчка на первой кнопке сгенерируем два сигнала. Первый сигнал будет имитировать нажатие второй кнопки, а второй сигнал будет пользовательским. Внутри обработчиков выведем сообщения в окно консоли. Содержимое файла widget.h приведено в листинге 4.9, файла widget.cpp — в листинге 4.10, а файла main.cpp — в листинге 4.11.

Листинг 4.9. Содержимое файла widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QApplication>
#include <QWidget>
#include <QPushButton>
#include <QVBoxLayout>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent=nullptr);
   ~Widget();
signals:
   void mysignal(int, int);
private slots:
   void on_btn1_clicked();
   void on_btn2_clicked();
   void on_mysignal(int, int);
```

```
private:
   QPushButton *btn1;
   QPushButton *btn2;
   QVBoxLayout *vbox;
};
#endif // WIDGET_H
```

Листинг 4.10. Содержимое файла widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   btn1 = new QPushButton("Нажми меня");
   btn2 = new QPushButton("Кнопка 2");
   vbox = new QVBoxLayout();
   vbox->addWidget(btn1);
   vbox->addWidget(btn2);
   setLayout(vbox);
   connect(btn1, SIGNAL(clicked()),
           this, SLOT(on_btn1_clicked()));
   connect(btn2, SIGNAL(clicked()),
           this, SLOT(on_btn2_clicked()));
   connect(this, SIGNAL(mysignal(int,int)),
           this, SLOT(on_mysignal(int,int)));
}

void Widget::on_btn1_clicked()
{
   qDebug() << "Нажата кнопка 1";
   // Генерируем сигналы
   emit btn2->clicked();
   emit mysignal(10, 20);
}

void Widget::on_btn2_clicked()
{
   qDebug() << "Нажата кнопка 2";
}

void Widget::on_mysignal(int a, int b)
{
   qDebug() << "on_mysignal" << a << b;
}

Widget::~Widget() {}
```

Листинг 4.11. Содержимое файла main.cpp

```
#include "widget.h"

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   Widget window;
   window.setWindowTitle("Генерация сигнала из программы");
   window.resize(350, 150);
   window.show();
   return app.exec();
}
```

Результат выполнения после нажатия первой кнопки:

```
Нажата кнопка 1
Нажата кнопка 2
on_mysignal 10 20
```

Сгенерировать сигнал можно не только с помощью `emit`. Некоторые компоненты предоставляют методы, которые посылают сигнал. Например, у кнопок существует метод `click()`. Используя этот метод, инструкцию

```
emit btn2->clicked();
```

можно записать следующим образом:

```
btn2->click();
```

Более подробно такие методы мы будем рассматривать при изучении конкретных компонентов.

4.4. Использование таймеров

Таймеры позволяют через определенный интервал времени выполнять метод с предопределенным названием `timerEvent()`. Для назначения таймера используется метод `startTimer()` из класса `QObject`. Прототипы метода:

```
int startTimer(int interval, Qt::TimerType timerType = Qt::CoarseTimer)
int startTimer(std::chrono::milliseconds time,
               Qt::TimerType timerType = Qt::CoarseTimer)
```

Метод `startTimer()` возвращает идентификатор таймера, с помощью которого можно остановить таймер. Первый параметр задает промежуток времени в миллисекундах, по истечении которого выполняется метод `timerEvent()`. Прототип метода:

```
virtual void timerEvent(QTimerEvent *event)
```

Внутри метода `timerEvent()` можно получить идентификатор таймера с помощью метода `timerId()` объекта класса `QTimerEvent`. Прототип метода:

```
int timerId() const
```

Минимальное значение интервала зависит от операционной системы. Если в первом параметре указать значение `0`, то таймер будет срабатывать много раз при отсутствии других необработанных событий.

Чтобы остановить таймер, необходимо воспользоваться методом `killTimer()` из класса `QObject`. Прототип метода:

```
void killTimer(int id)
```

В качестве параметра указывается идентификатор, возвращаемый методом `startTimer()`.

Создадим часы в окне, которые будут отображать текущее системное время с точностью до секунды, и добавим возможность запуска и остановки часов с помощью кнопок. Содержимое файла widget.h приведено в листинге 4.12, файла widget.cpp — в листинге 4.13, а файла main.cpp — в листинге 4.14.

Листинг 4.12. Содержимое файла widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QPushButton>
#include <QVBoxLayout>
#include <QTimerEvent>
#include <QTime>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent=nullptr);
   ~Widget();
protected:
   void timerEvent(QTimerEvent *event) override;
private slots:
   void on_btn1_clicked();
   void on_btn2_clicked();
private:
   QLabel *label;
   QPushButton *btn1;
```

```
    QPushButton *btn2;
    QVBoxLayout *vbox;
    int timer_id;
};
#endif // WIDGET_H
```

Листинг 4.13. Содержимое файла widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    timer_id = -1;
    label = new QLabel("");
    label->setAlignment(Qt::AlignCenter);
    btn1 = new QPushButton("Запустить");
    btn2 = new QPushButton("Остановить");
    btn2->setEnabled(false);
    vbox = new QVBoxLayout();
    vbox->addWidget(label);
    vbox->addWidget(btn1);
    vbox->addWidget(btn2);
    setLayout(vbox);
    connect(btn1, SIGNAL(clicked()),
            this, SLOT(on_btn1_clicked()));
    connect(btn2, SIGNAL(clicked()),
            this, SLOT(on_btn2_clicked()));
}

void Widget::on_btn1_clicked()
{
    timer_id = startTimer(1000); // 1 секунда
    btn1->setEnabled(false);
    btn2->setEnabled(true);
}

void Widget::on_btn2_clicked()
{
    if (timer_id != -1) {
        killTimer(timer_id);
        timer_id = -1;
    }
    btn1->setEnabled(true);
    btn2->setEnabled(false);
}
```

```
void Widget::timerEvent(QTimerEvent *event)
{
   // qDebug() << "ID =" << event->timerId();
   QTime t = QTime::currentTime();
   label->setText(t.toString("HH:mm:ss"));
}

Widget::~Widget() {}
```

Листинг 4.14. Содержимое файла main.cpp

```
#include "widget.h"

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   Widget window;
   window.setWindowTitle("Часы в окне");
   window.resize(350, 100);
   window.show();
   return app.exec();
}
```

4.5. Класс *QTimer:* таймер

Вместо методов `startTimer()` и `killTimer()` можно воспользоваться классом `QTimer`. Конструктор класса имеет следующий формат:

```
#include <QTimer>
QTimer(QObject *parent = nullptr)
```

Методы класса:

- `setInterval()` — задает промежуток времени в миллисекундах, по истечении которого генерируется сигнал `timeout()`. Минимальное значение интервала зависит от операционной системы. Если в параметре указать значение `0`, то таймер будет срабатывать много раз при отсутствии других необработанных сигналов. Прототипы метода:

  ```
  void setInterval(int msec)
  void setInterval(std::chrono::milliseconds value)
  ```

- `start()` — запускает таймер. В необязательном параметре можно указать промежуток времени в миллисекундах. Если параметр не указан, то используется значение, возвращаемое методом `interval()`. Метод является слотом. Прототипы метода:

  ```
  void start()
  void start(int msec)
  void start(std::chrono::milliseconds msec)
  ```

- `stop()` — останавливает таймер. Метод является слотом. Прототип метода:

  ```
  void stop()
  ```

- `setSingleShot()` — если в параметре указано значение `true`, то таймер будет срабатывать только один раз, в противном случае — многократно. Прототип метода:

  ```
  void setSingleShot(bool singleShot)
  ```

- `interval()` — возвращает установленный интервал. Прототип метода:

  ```
  int interval() const
  ```

- `timerId()` — возвращает идентификатор таймера или значение `-1`. Прототип метода:

  ```
  int timerId() const
  ```

- `isSingleShot()` — возвращает значение `true`, если таймер будет срабатывать только один раз, и `false` — в противном случае. Прототип метода:

  ```
  bool isSingleShot() const
  ```

- `isActive()` — возвращает значение `true`, если таймер генерирует сигналы, и `false` — в противном случае. Прототип метода:

  ```
  bool isActive() const
  ```

Переделаем предыдущий пример и используем класс `QTimer` вместо методов `startTimer()` и `killTimer()`. Содержимое файла widget.h приведено в листинге 4.15, файла widget.cpp — в листинге 4.16, а файла main.cpp — в листинге 4.17.

Листинг 4.15. Содержимое файла widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QPushButton>
#include <QVBoxLayout>
#include <QTime>
#include <QTimer>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent=nullptr);
   ~Widget();
```

```
private slots:
   void on_btn1_clicked();
   void on_btn2_clicked();
   void on_timeout();
private:
   QLabel *label;
   QPushButton *btn1;
   QPushButton *btn2;
   QVBoxLayout *vbox;
   QTimer *timer;
};
#endif // WIDGET_H
```

Листинг 4.16. Содержимое файла widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   timer = new QTimer(this);
   label = new QLabel("");
   label->setAlignment(Qt::AlignCenter);
   btn1 = new QPushButton("Запустить");
   btn2 = new QPushButton("Остановить");
   btn2->setEnabled(false);
   vbox = new QVBoxLayout();
   vbox->addWidget(label);
   vbox->addWidget(btn1);
   vbox->addWidget(btn2);
   setLayout(vbox);
   connect(btn1, SIGNAL(clicked()),
           this, SLOT(on_btn1_clicked()));
   connect(btn2, SIGNAL(clicked()),
           this, SLOT(on_btn2_clicked()));
   connect(timer, SIGNAL(timeout()),
           this, SLOT(on_timeout()));
}

void Widget::on_btn1_clicked()
{
   timer->start(1000); // 1 секунда
   btn1->setEnabled(false);
   btn2->setEnabled(true);
}
```

```
void Widget::on_btn2_clicked()
{
   timer->stop();
   btn1->setEnabled(true);
   btn2->setEnabled(false);
}

void Widget::on_timeout()
{
   QTime t = QTime::currentTime();
   label->setText(t.toString("HH:mm:ss"));
}

Widget::~Widget() {}
```

Листинг 4.17. Содержимое файла main.cpp

```
#include "widget.h"

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   Widget window;
   window.setWindowTitle("Использование класса QTimer");
   window.resize(350, 100);
   window.show();
   return app.exec();
}
```

Кроме перечисленных методов в классе `QTimer` определен статический метод `singleShot()`, предназначенный для вызова указанного обработчика через заданный промежуток времени. Таймер срабатывает только один раз. Прототипы метода:

```
static void singleShot(int msec, const QObject *receiver, const char *member)
static void singleShot(int msec, Qt::TimerType timerType,
                       const QObject *receiver, const char *member)
static void singleShot(int msec, const QObject *receiver,
                       PointerToMemberFunction method)
static void singleShot(int msec, Qt::TimerType timerType,
       const QObject *receiver, PointerToMemberFunction method)
static void singleShot(int msec, Functor functor)
static void singleShot(int msec, Qt::TimerType timerType, Functor functor)
static void singleShot(int msec, const QObject *context, Functor functor)
static void singleShot(int msec, Qt::TimerType timerType,
                       const QObject *context, Functor functor)
static void singleShot(std::chrono::milliseconds msec,
                       const QObject *receiver, const char *member)
```

```
static void singleShot(std::chrono::milliseconds msec,
       Qt::TimerType timerType, const QObject *receiver,
       const char *member)
```

Примеры использования статического метода `singleShot()`:

```
QTimer::singleShot(2000, this, SLOT(on_timeout()));
QTimer::singleShot(10000, qApp, SLOT(quit()));
```

4.6. Перехват всех событий

В предыдущих разделах мы рассмотрели обработку сигналов, которые позволяют обмениваться сообщениями между компонентами. Обработка внешних событий, например нажатий клавиш, осуществляется несколько иначе. Чтобы обработать событие, необходимо наследовать класс и переопределить в нем метод со специальным названием. Например, чтобы обработать нажатие клавиши, следует переопределить метод `keyPressEvent()`. Специальные методы принимают объект, содержащий детальную информацию о событии, например код нажатой клавиши. Все эти объекты являются наследниками класса `QEvent` и наследуют следующие методы:

- `accept()` — устанавливает флаг, который является признаком согласия с дальнейшей обработкой события. Например, если в методе `closeEvent()` вызывать метод `accept()` через объект события, то окно будет закрыто. Флаг обычно устанавливается по умолчанию. Прототип метода:

  ```
  void accept()
  ```

- `ignore()` — сбрасывает флаг. Например, если в методе `closeEvent()` вызывать метод `ignore()` через объект события, то окно закрыто не будет. Прототип метода:

  ```
  void ignore()
  ```

- `setAccepted()` — если в качестве параметра указано значение `true`, то флаг будет установлен (аналогично вызову метода `accept()`), а если `false`, то сброшен (аналогично вызову метода `ignore()`). Прототип метода:

  ```
  virtual void setAccepted(bool accepted)
  ```

- `isAccepted()` — возвращает текущее состояние флага. Прототип метода:

  ```
  bool isAccepted() const
  ```

- `registerEventType()` — позволяет зарегистрировать пользовательский тип события. Метод возвращает идентификатор зарегистрированного события. В качестве параметра можно указать значение в пределах от `QEvent::User` (1000) до `QEvent::MaxUser` (65 535). Метод является статическим. Прототип метода:

  ```
  static int registerEventType(int hint = -1)
  ```

- `spontaneous()` — возвращает `true`, если событие сгенерировано системой, и `false`, если событие сгенерировано внутри программы искусственно. Прототип метода:

  ```
  bool spontaneous() const
  ```

- `type()` — возвращает тип события. Прототип метода:

  ```
  QEvent::Type type() const
  ```

  Перечислим основные типы событий (полный список смотрите в документации по классу `QEvent`):

  - `0` — `None` — нет события;
  - `1` — `Timer` — событие таймера;
  - `2` — `MouseButtonPress` — нажата кнопка мыши;
  - `3` — `MouseButtonRelease` — отпущена кнопка мыши;
  - `4` — `MouseButtonDblClick` — двойной щелчок мышью;
  - `5` — `MouseMove` — перемещение мыши;
  - `6` — `KeyPress` — клавиша клавиатуры нажата;
  - `7` — `KeyRelease` — клавиша клавиатуры отпущена;
  - `8` — `FocusIn` — получен фокус ввода с клавиатуры;
  - `9` — `FocusOut` — потерян фокус ввода с клавиатуры;
  - `10` — `Enter` — указатель мыши входит в область компонента;
  - `11` — `Leave` — указатель мыши покидает область компонента;
  - `12` — `Paint` — перерисовка компонента;
  - `13` — `Move` — позиция компонента изменилась;
  - `14` — `Resize` — изменился размер компонента;
  - `17` — `Show` — компонент отображен;
  - `18` — `Hide` — компонент скрыт;
  - `19` — `Close` — окно закрыто;
  - `24` — `WindowActivate` — окно стало активным;
  - `25` — `WindowDeactivate` — окно стало неактивным;
  - `26` — `ShowToParent` — дочерний компонент отображен;
  - `27` — `HideToParent` — дочерний компонент скрыт;
  - `31` — `Wheel` — прокручено колесико мыши;
  - `40` — `Clipboard` — содержимое буфера обмена изменено;
  - `60` — `DragEnter` — указатель мыши входит в область компонента при операции перетаскивания;

- 61 — DragMove — производится операция перетаскивания;
- 62 — DragLeave — указатель мыши покидает область компонента при операции перетаскивания;
- 63 — Drop — операция перетаскивания завершена;
- 68 — ChildAdded — добавлен дочерний компонент;
- 69 — ChildPolished — производится настройка дочернего компонента;
- 71 — ChildRemoved — удален дочерний компонент;
- 74 — PolishRequest — компонент настроен;
- 82 — ContextMenu — событие контекстного меню;
- 99 — ActivationChange — изменился статус активности окна верхнего уровня;
- 103 — WindowBlocked — окно блокировано модальным окном;
- 104 — WindowUnblocked — текущее окно разблокировано после закрытия модального окна;
- 105 — WindowStateChange — статус окна изменился;
- 1000 — User — пользовательское событие;
- 65535 — MaxUser — максимальный идентификатор пользовательского события.

Перехват всех событий осуществляется с помощью метода с предопределенным названием event(). Прототип метода:

```
virtual bool event(QEvent *e)
```

Через параметр доступен объект с дополнительной информацией о событии. Этот объект отличается для разных типов событий, например для события MouseButtonPress объект будет экземпляром класса QMouseEvent, а для события KeyPress — экземпляром класса QKeyEvent. Какие методы содержат эти классы, мы рассмотрим в следующих разделах.

Внутри метода event() следует вернуть значение true, если событие принято, и false — в противном случае. Если возвращается значение true, то родительский компонент не получит событие. Чтобы продолжить распространение события, необходимо вызвать метод event() базового класса и передать ему текущий объект события. Обычно это делается так:

```
return QWidget::event(e);
```

В этом случае пользовательский класс является наследником класса QWidget и переопределяет метод event(). Если вы наследуете другой класс, то необходимо вызывать метод именно этого класса. Например, при наследовании класса QLabel инструкция будет выглядеть так:

```
return QLabel::event(e);
```

Рассмотрим пример перехвата нажатия клавиши, щелчка мышью и закрытия окна. Содержимое файла widget.h приведено в листинге 4.18, файла widget.cpp — в листинге 4.19, а файла main.cpp — в листинге 4.20.

Листинг 4.18. Содержимое файла widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QApplication>
#include <QWidget>
#include <QKeyEvent>
#include <QMouseEvent>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent=nullptr);
   ~Widget();
   bool event(QEvent *e) override;
};
#endif // WIDGET_H
```

Листинг 4.19. Содержимое файла widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{}

bool Widget::event(QEvent *e)
{
   if (e->type() == QEvent::KeyPress) {
      qDebug() << "Нажата клавиша на клавиатуре";
      QKeyEvent *k = static_cast<QKeyEvent*>(e);
      qDebug() << "Код:" << k->key() << "текст:" << k->text();
   }
   else if (e->type() == QEvent::Close) {
      qDebug() << "Окно закрыто";
   }
   else if (e->type() == QEvent::MouseButtonPress) {
      qDebug() << "Щелчок мышью";
      QMouseEvent *m = static_cast<QMouseEvent*>(e);
      qDebug() << "Координаты:" << m->position();
   }
```

```
    return QWidget::event(e); // Отправляем дальше
}

Widget::~Widget() {}
```

Листинг 4.20. Содержимое файла main.cpp

```
#include "widget.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    Widget window;
    window.setWindowTitle("Обработка событий");
    window.resize(350, 100);
    window.show();
    return app.exec();
}
```

4.7. События окна

Перехватывать все события следует только в самом крайнем случае. В обычных ситуациях нужно использовать методы, предназначенные для обработки определенного события. Например, чтобы обработать закрытие окна, достаточно переопределить метод `closeEvent()`. Какие методы требуется переопределять для обработки событий окна, мы сейчас и рассмотрим.

4.7.1. Изменение состояния окна

Отследить изменение состояния окна (сворачивание, разворачивание, сокрытие и отображение) позволяют следующие методы:

- `changeEvent()` — вызывается при изменении состояния окна, приложения или компонента. Иными словами, метод вызывается не только при изменении статуса окна, но и при изменении заголовка окна, палитры, статуса активности окна верхнего уровня, языка, локали и др. (полный список смотрите в документации). При обработке события `WindowStateChange` через параметр доступен экземпляр класса `QWindowStateChangeEvent`. Этот класс содержит только метод `oldState()`, с помощью которого можно получить предыдущий статус окна. Прототипы методов:

  ```
  virtual void changeEvent(QEvent *event)
  Qt::WindowStates oldState() const
  ```

- `showEvent()` — вызывается при отображении компонента. Через параметр доступен экземпляр класса `QShowEvent`. Прототип метода:

  ```
  virtual void showEvent(QShowEvent *event)
  ```

- `hideEvent()` — вызывается при сокрытии компонента. Через параметр доступен экземпляр класса `QHideEvent`. Прототип метода:

```
virtual void hideEvent(QHideEvent *event)
```

Для примера выведем текущее состояние окна в консоль при сворачивании, разворачивании, сокрытии и отображении окна. Содержимое файла widget.h приведено в листинге 4.21, файла widget.cpp — в листинге 4.22, а файла main.cpp — в листинге 4.23.

Листинг 4.21. Содержимое файла widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QApplication>
#include <QWidget>
#include <QShowEvent>
#include <QHideEvent>
#include <QPushButton>
#include <QVBoxLayout>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent=nullptr);
   ~Widget();
protected:
   void changeEvent(QEvent *e) override;
   void showEvent(QShowEvent *e) override;
   void hideEvent(QHideEvent *e) override;
private:
   QPushButton *btn1;
   QPushButton *btn2;
   QVBoxLayout *vbox;
};
#endif // WIDGET_H
```

Листинг 4.22. Содержимое файла widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   btn1 = new QPushButton("Полноэкранный режим");
   btn2 = new QPushButton("Нормальный режим");
```

```
    vbox = new QVBoxLayout();
    vbox->addWidget(btn1);
    vbox->addWidget(btn2);
    setLayout(vbox);
    QObject::connect(btn1, SIGNAL(clicked()),
                     this, SLOT(showFullScreen()));
    QObject::connect(btn2, SIGNAL(clicked()),
                     this, SLOT(showNormal()));
}

void Widget::changeEvent(QEvent *e)
{
    if (e->type() == QEvent::WindowStateChange) {
        if (isMinimized()) {
            qDebug() << "Окно свернуто";
        }
        else if (isMaximized()) {
            qDebug() << "Окно раскрыто до максимальных размеров";
        }
        else if (isFullScreen()) {
            qDebug() << "Полноэкранный режим";
        }
        else if (isActiveWindow()) {
            qDebug() << "Окно находится в фокусе ввода";
        }
    }
    QWidget::changeEvent(e); // Отправляем дальше
}

void Widget::showEvent(QShowEvent *e)
{
    qDebug() << "Окно отображено";
    QWidget::showEvent(e); // Отправляем дальше
}

void Widget::hideEvent(QHideEvent *e)
{
    qDebug() << "Окно скрыто";
    QWidget::hideEvent(e); // Отправляем дальше
}

Widget::~Widget() {}
```

Листинг 4.23. Содержимое файла main.cpp

```
#include "widget.h"

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

```
{
   QApplication app(argc, argv);
   Widget window;
   window.setWindowTitle("Отслеживание состояния окна");
   window.resize(350, 100);
   window.show();
   return app.exec();
}
```

4.7.2. Изменение положения окна и его размеров

При перемещении окна и изменении размеров вызываются следующие методы:

- `moveEvent()` — вызывается непрерывно при перемещении окна. Через параметр доступен экземпляр класса `QMoveEvent`. Прототип метода:

  ```
  virtual void moveEvent(QMoveEvent *event)
  ```

  Получить координаты окна позволяют следующие методы из класса `QMoveEvent`:

  - `pos()` — возвращает экземпляр класса `QPoint` с текущими координатами;
  - `oldPos()` — возвращает экземпляр класса `QPoint` с предыдущими координатами.

  Прототипы методов:

  ```
  const QPoint &pos() const
  const QPoint &oldPos() const
  ```

- `resizeEvent()` — вызывается непрерывно при изменении размеров окна. Через параметр доступен экземпляр класса `QResizeEvent`. Прототип метода:

  ```
  virtual void resizeEvent(QResizeEvent *event)
  ```

  Получить размеры окна позволяют следующие методы из класса `QResizeEvent`:

  - `size()` — возвращает экземпляр класса `QSize` с текущими размерами;
  - `oldSize()` — возвращает экземпляр класса `QSize` с предыдущими размерами.

  Прототипы методов:

  ```
  const QSize &size() const
  const QSize &oldSize() const
  ```

Рассмотрим пример обработки изменения положения окна и его размеров. Содержимое файла widget.h приведено в листинге 4.24, файла widget.cpp — в листинге 4.25, а файла main.cpp — в листинге 4.26.

Листинг 4.24. Содержимое файла widget.h

```
#ifndef WIDGET_H
#define WIDGET_H
```

```
#include <QApplication>
#include <QWidget>
#include <QMoveEvent>
#include <QResizeEvent>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent=nullptr);
   ~Widget();
protected:
   void moveEvent(QMoveEvent *e) override;
   void resizeEvent(QResizeEvent *e) override;
};
#endif // WIDGET_H
```

Листинг 4.25. Содержимое файла widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{}

void Widget::moveEvent(QMoveEvent *e)
{
   qDebug() << "moveEvent" << e->pos().x() << e->pos().y();
   QWidget::moveEvent(e);
}

void Widget::resizeEvent(QResizeEvent *e)
{
   qDebug() << "resizeEvent"
            << e->size().width() << e->size().height();
   QWidget::resizeEvent(e);
}

Widget::~Widget() {}
```

Листинг 4.26. Содержимое файла main.cpp

```
#include "widget.h"

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   Widget window;
```

```
    window.setWindowTitle("Изменение положения окна и его размеров");
    window.resize(350, 100);
    window.show();
    return app.exec();
}
```

4.7.3. Перерисовка окна или его части

Когда компонент (или его часть) становится видимым, требуется выполнить перерисовку компонента или только его части. В этом случае вызывается метод с названием `paintEvent()`. Прототип метода:

```
virtual void paintEvent(QPaintEvent *event)
```

Через параметр доступен экземпляр класса `QPaintEvent`, который содержит следующие методы:

- `rect()` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, которую требуется перерисовать. Прототип метода:

  ```
  const QRect &rect() const
  ```

- `region()` — возвращает экземпляр класса `QRegion` с регионом, требующим перерисовки. Прототип метода:

  ```
  const QRegion &region() const
  ```

С помощью этих методов можно получать координаты области, которая, например, была ранее перекрыта другим окном и теперь оказалась в зоне видимости. Перерисовывая только область, а не весь компонент, можно достичь более эффективного расходования ресурсов компьютера. Следует также заметить, что в целях эффективности последовательность событий перерисовки может быть объединена в одно событие с общей областью перерисовки.

В некоторых случаях перерисовку окна необходимо выполнить вне зависимости от внешних действий системы или пользователя, например при изменении каких-либо значений нужно обновить график. Вызвать событие перерисовки компонента позволяют следующие методы из класса `QWidget`:

- `repaint()` — незамедлительно вызывает метод `paintEvent()` для перерисовки компонента при условии, что компонент не скрыт и обновление не запрещено с помощью метода `setUpdatesEnabled()`. Прототипы методов (первый прототип является слотом):

  ```
  void repaint()
  void repaint(int x, int y, int w, int h)
  void repaint(const QRect &rect)
  void repaint(const QRegion &rgn)
  void setUpdatesEnabled(bool enable)
  ```

- `update()` — посылает сообщение о необходимости перерисовки компонента при условии, что компонент не скрыт и обновление не запрещено. Событие будет

обработано на следующей итерации основного цикла приложения. Если посылаются сразу несколько сообщений, то они объединяются в одно сообщение. Благодаря этому можно избежать неприятного мерцания. Метод `update()` предпочтительнее использовать вместо метода `repaint()`. Прототипы метода (первый прототип является слотом):

```
void update()
void update(int x, int y, int w, int h)
void update(const QRect &rect)
void update(const QRegion &rgn)
```

4.7.4. Предотвращение закрытия окна

При нажатии кнопки **Закрыть** в заголовке окна или при вызове метода `close()` вызывается метод `closeEvent()`. Прототип метода:

```
virtual void closeEvent(QCloseEvent *event)
```

Через параметр доступен экземпляр класса `QCloseEvent`. Чтобы предотвратить закрытие окна, необходимо вызвать метод `ignore()` через объект события, в противном случае — метод `accept()`.

В качестве примера при нажатии кнопки **Закрыть** в заголовке окна выведем стандартное диалоговое окно с запросом подтверждения закрытия окна. Если пользователь нажимает кнопку **Yes**, то закроем окно, а если кнопку **No** или просто закрывает диалоговое окно, то прервем закрытие окна. Содержимое файла widget.h приведено в листинге 4.27, файла widget.cpp — в листинге 4.28, а файла main.cpp — в листинге 4.29.

Листинг 4.27. Содержимое файла widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QApplication>
#include <QWidget>
#include <QCloseEvent>
#include <QMessageBox>

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent=nullptr);
   ~Widget();
protected:
  void closeEvent(QCloseEvent *e) override;
};
#endif // WIDGET_H
```

Листинг 4.28. Содержимое файла widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{}

void Widget::closeEvent(QCloseEvent *e)
{
   QMessageBox::StandardButton result;
   result = QMessageBox::question(this,
               "Подтверждение закрытия окна",
               "Вы действительно хотите закрыть окно?",
               QMessageBox::Yes | QMessageBox::No,
               QMessageBox::No);
   if (result == QMessageBox::Yes) {
      e->accept();
      QWidget::closeEvent(e);
   }
   else {
      e->ignore();
   }
}

Widget::~Widget() {}
```

Листинг 4.29. Содержимое файла main.cpp

```
#include "widget.h"

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   Widget window;
   window.setWindowTitle("Обработка закрытия окна");
   window.resize(350, 100);
   window.show();
   return app.exec();
}
```

4.8. События клавиатуры

События клавиатуры очень часто обрабатываются внутри приложений. Например, при нажатии клавиши <F1> можно вывести справочную информацию, при нажатии клавиши <Enter> в однострочном текстовом поле можно перенести фокус ввода на другой компонент и т. д. Рассмотрим события клавиатуры подробно.

4.8.1. Установка фокуса ввода

В один момент времени только один компонент (или вообще ни одного) может иметь фокус ввода. Для управления фокусом ввода предназначены следующие методы из класса `QWidget`:

- `setFocus()` — устанавливает фокус ввода, если компонент находится в активном окне. Прототипы метода (первый прототип является слотом):

```
void setFocus()
void setFocus(Qt::FocusReason reason)
```

  В параметре `reason` можно указать причину изменения фокуса ввода. Указываются следующие константы:

  - `Qt::MouseFocusReason` — фокус изменен с помощью мыши;
  - `Qt::TabFocusReason` — нажата клавиша <Tab>;
  - `Qt::BacktabFocusReason` — нажата комбинация клавиш <Shift>+<Tab>;
  - `Qt::ActiveWindowFocusReason` — окно стало активным или неактивным;
  - `Qt::PopupFocusReason` — открыто или закрыто всплывающее окно;
  - `Qt::ShortcutFocusReason` — нажата комбинация клавиш быстрого доступа;
  - `Qt::MenuBarFocusReason` — фокус изменился из-за меню;
  - `Qt::OtherFocusReason` — другая причина;

- `clearFocus()` — убирает фокус ввода с компонента. Прототип метода:

```
void clearFocus()
```

- `hasFocus()` — возвращает значение `true`, если компонент находится в фокусе ввода, и `false` — в противном случае. Прототип метода:

```
bool hasFocus() const
```

- `focusWidget()` — возвращает указатель на последний компонент, для которого вызывался метод `setFocus()`. Для компонентов верхнего уровня возвращается указатель на компонент, который получит фокус после того, как окно станет активным. Прототип метода:

```
QWidget *focusWidget() const
```

- `setFocusProxy()` — позволяет передать указатель на компонент, который будет получать фокус ввода вместо текущего компонента. Прототип метода:

```
void setFocusProxy(QWidget *w)
```

- `focusProxy()` — возвращает указатель на компонент, который обрабатывает фокус ввода вместо текущего компонента. Если компонента нет, то метод возвращает нулевой указатель. Прототип метода:

```
QWidget *focusProxy() const
```

- `focusNextChild()` — находит следующий компонент, которому можно передать фокус, и передает фокус. Аналогично нажатию клавиши <Tab>. Метод возвращает значение `true`, если компонент найден, и `false` — в противном случае. Прототип метода:

```
protected:
   bool focusNextChild()
```

- `focusPreviousChild()` — находит предыдущий компонент, которому можно передать фокус, и передает фокус. Аналогично нажатию комбинации клавиш <Shift>+<Tab>. Метод возвращает значение `true`, если компонент найден, и `false` — в противном случае. Прототип метода:

```
protected:
   bool focusPreviousChild()
```

- `focusNextPrevChild()` — если в параметре указано значение `true`, то метод аналогичен методу `focusNextChild()`. Если в параметре указано значение `false`, то метод аналогичен методу `focusPreviousChild()`. Метод возвращает значение `true`, если компонент найден, и `false` — в противном случае. Прототип метода:

```
protected:
   virtual bool focusNextPrevChild(bool next)
```

- `setTabOrder()` — позволяет задать последовательность смены фокуса при нажатии клавиши <Tab>. Метод является статическим. Прототип метода:

```
static void setTabOrder(QWidget *first, QWidget *second)
```

  В параметре `second` передается указатель на компонент, на который переместится фокус с компонента `first`. Если компонентов много, то метод вызывается несколько раз. Пример указания цепочки перехода `widget1 -> widget2 -> widget3 -> widget4`:

```
QWidget::setTabOrder(widget1, widget2)
QWidget::setTabOrder(widget2, widget3)
QWidget::setTabOrder(widget3, widget4)
```

- `setFocusPolicy()` — задает способ получения фокуса компонентом. Прототип метода:

```
void setFocusPolicy(Qt::FocusPolicy policy)
```

  В качестве параметра указываются следующие константы:

  - `Qt::NoFocus` — компонент не может получать фокус;
  - `Qt::TabFocus` — получает фокус с помощью клавиши <Tab>;
  - `Qt::ClickFocus` — получает фокус с помощью щелчка мышью;
  - `Qt::StrongFocus` — получает фокус с помощью клавиши <Tab> и щелчка мышью;
  - `Qt::WheelFocus` — получает фокус с помощью клавиши <Tab>, щелчка мышью и колесика мыши;

- `focusPolicy()` — возвращает текущий способ получения фокуса. Прототип метода:

  ```
  Qt::FocusPolicy focusPolicy() const
  ```

- `grabKeyboard()` — захватывает ввод с клавиатуры. Другие компоненты не будут получать события клавиатуры, пока не будет вызван метод `releaseKeyboard()`. Прототип метода:

  ```
  void grabKeyboard()
  ```

- `releaseKeyboard()` — освобождает захваченный ранее ввод с клавиатуры. Прототип метода:

  ```
  void releaseKeyboard()
  ```

Получить указатель на компонент, находящийся в фокусе ввода, позволяет статический метод `focusWidget()` из класса `QApplication`. Прототип метода:

```
static QWidget *focusWidget()
```

Если ни один компонент не имеет фокуса ввода, то метод возвращает нулевой указатель. Не путайте этот метод с одноименным методом из класса `QWidget`.

Обработать получение и потерю фокуса ввода позволяют следующие методы:

- `focusInEvent()` — вызывается при получении фокуса ввода. Прототип метода:

  ```
  virtual void focusInEvent(QFocusEvent *event)
  ```

- `focusOutEvent()` — вызывается при потере фокуса ввода. Прототип метода:

  ```
  virtual void focusOutEvent(QFocusEvent *event)
  ```

Через параметр доступен экземпляр класса `QFocusEvent`, который содержит следующие методы:

- `gotFocus()` — возвращает значение `true`, если тип события `QEvent::FocusIn`, и `false` — в противном случае. Прототип метода:

  ```
  bool gotFocus() const
  ```

- `lostFocus()` — возвращает значение `true`, если тип события `QEvent::FocusOut`, и `false` — в противном случае. Прототип метода:

  ```
  bool lostFocus() const
  ```

- `reason()` — возвращает причину установки фокуса. Значение аналогично значению параметра `reason` в методе `setFocus()`. Прототип метода:

  ```
  Qt::FocusReason reason() const
  ```

Создадим окно с кнопкой и двумя однострочными полями ввода. Для полей ввода обработаем получение и потерю фокуса ввода, а при нажатии кнопки установим фокус ввода на второе поле. Кроме того, зададим последовательность перехода при нажатии клавиши <Tab>. Содержимое файла mylineedit.h приведено в листинге 4.30, файла mylineedit.cpp — в листинге 4.31, файла widget.h — в листинге 4.32, файла widget.cpp — в листинге 4.33, а файла main.cpp — в листинге 4.34.

Листинг 4.30. Содержимое файла mylineedit.h

```
#ifndef MYLINEEDIT_H
#define MYLINEEDIT_H

#include <QWidget>
#include <QLineEdit>
#include <QFocusEvent>

class MyLineEdit : public QLineEdit
{
   Q_OBJECT
public:
   MyLineEdit(int id, QWidget *parent=nullptr);
protected:
   void focusInEvent(QFocusEvent *e) override;
   void focusOutEvent(QFocusEvent *e) override;
private:
   int id_;
};

#endif // MYLINEEDIT_H
```

Листинг 4.31. Содержимое файла mylineedit.cpp

```
#include "mylineedit.h"

MyLineEdit::MyLineEdit(int id, QWidget *parent)
   : QLineEdit(parent), id_(id)
{}

void MyLineEdit::focusInEvent(QFocusEvent *e)
{
   qDebug() << "Получен фокус полем" << id_;
   QLineEdit::focusInEvent(e);
}

void MyLineEdit::focusOutEvent(QFocusEvent *e)
{
   qDebug() << "Потерян фокус полем" << id_;
   QLineEdit::focusOutEvent(e);
}
```

Листинг 4.32. Содержимое файла widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QApplication>
#include <QWidget>
```

```
#include <QPushButton>
#include <QVBoxLayout>
#include "mylineedit.h"

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent=nullptr);
   ~Widget();
private:
   MyLineEdit *line1;
   MyLineEdit *line2;
   QPushButton *btn1;
   QVBoxLayout *vbox;
};
#endif // WIDGET_H
```

Листинг 4.33. Содержимое файла widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   line1 = new MyLineEdit(1);
   line2 = new MyLineEdit(2);
   btn1 = new QPushButton("Установить фокус на поле 2");
   vbox = new QVBoxLayout();
   vbox->addWidget(btn1);
   vbox->addWidget(line1);
   vbox->addWidget(line2);
   setLayout(vbox);
   QObject::connect(btn1, SIGNAL(clicked()),
                    line2, SLOT(setFocus()));
   // Задаем порядок обхода с помощью клавиши <Tab>
   QWidget::setTabOrder(line1, line2);
   QWidget::setTabOrder(line2, btn1);
}

Widget::~Widget() {}
```

Листинг 4.34. Содержимое файла main.cpp

```
#include "widget.h"

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

```
{
   QApplication app(argc, argv);
   Widget window;
   window.setWindowTitle("Установка фокуса ввода");
   window.resize(350, 100);
   window.show();
   return app.exec();
}
```

4.8.2. Назначение клавиш быстрого доступа

Клавиши быстрого доступа (иногда их также называют *горячими клавишами*) позволяют установить фокус ввода с помощью нажатия специальной клавиши (например, <Alt> или <Ctrl>) и какой-либо дополнительной клавиши. Если после нажатия клавиш быстрого доступа в фокусе окажется кнопка (или пункт меню), то она будет нажата.

Чтобы задать клавиши быстрого доступа, следует в тексте надписи указать символ `&` перед буквой. В этом случае буква, перед которой указан символ `&`, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы компонент окажется в фокусе ввода. Некоторые компоненты, например текстовое поле, не имеют надписи. Чтобы задать клавиши быстрого доступа для таких компонентов, необходимо отдельно создать надпись и связать ее с компонентом с помощью метода `setBuddy()` из класса `QLabel`. Прототип метода:

```
void setBuddy(QWidget *buddy)
```

Если же создание надписи не представляется возможным, то можно воспользоваться следующими методами из класса `QWidget`:

- `grabShortcut()` — регистрирует клавиши быстрого доступа и возвращает идентификатор, с помощью которого можно управлять ими в дальнейшем. Прототип метода:

  ```
  int grabShortcut(const QKeySequence &key,
        Qt::ShortcutContext context = Qt::WindowShortcut)
  ```

  В параметре `key` указывается экземпляр класса `QKeySequence`. Создать экземпляр этого класса для комбинации <Alt>+<E> можно, например, так:

  ```
  QKeySequence::mnemonic("&e")
  QKeySequence("Alt+e")
  QKeySequence(Qt::ALT | Qt::Key_E)
  ```

  В параметре `context` можно указать константы `Qt::WidgetShortcut`, `Qt::WidgetWithChildrenShortcut`, `Qt::WindowShortcut` (значение по умолчанию) и `Qt::ApplicationShortcut`;

- `releaseShortcut()` — удаляет комбинацию с идентификатором `id`. Прототип метода:

  ```
  void releaseShortcut(int id)
  ```

- `setShortcutEnabled()` — если в качестве параметра `enable` указано значение `true` (значение по умолчанию), то клавиши быстрого доступа с идентификатором `id` разрешены. Значение `false` запрещает использование клавиш быстрого доступа. Прототип метода:

```
void setShortcutEnabled(int id, bool enable = true)
```

При нажатии клавиш быстрого доступа генерируется событие `QEvent::Shortcut`, которое можно обработать в методе `event()`. Через параметр доступен экземпляр класса `QShortcutEvent`, который содержит следующие методы:

- `shortcutId()` — возвращает идентификатор комбинации клавиш. Прототип метода:

```
int shortcutId() const
```

- `isAmbiguous()` — возвращает значение `true`, если событие отправлено сразу нескольким компонентам, и `false` — в противном случае. Прототип метода:

```
bool isAmbiguous() const
```

- `key()` — возвращает экземпляр класса `QKeySequence`. Прототип метода:

```
const QKeySequence &key() const
```

Создадим окно с надписью, двумя однострочными текстовыми полями и кнопкой. Для первого текстового поля назначим комбинацию клавиш (<Alt>+<B>) через надпись, а для второго поля (<Alt>+<E>) — с помощью метода `grabShortcut()`. Для кнопки назначим комбинацию клавиш (<Alt>+<У>) обычным образом через надпись на кнопке. Содержимое файла mylineedit.h приведено в листинге 4.35, файла mylineedit.cpp — в листинге 4.36, файла widget.h — в листинге 4.37, файла widget.cpp — в листинге 4.38, а файла main.cpp — в листинге 4.39.

Листинг 4.35. Содержимое файла mylineedit.h

```
#ifndef MYLINEEDIT_H
#define MYLINEEDIT_H

#include <QWidget>
#include <QLineEdit>
#include <QShortcutEvent>

class MyLineEdit : public QLineEdit
{
   Q_OBJECT
public:
   MyLineEdit(QWidget *parent=nullptr);
protected:
   bool event(QEvent *e) override;
public:
   int id;
};

#endif // MYLINEEDIT_H
```

Листинг 4.36. Содержимое файла mylineedit.cpp

```
#include "mylineedit.h"

MyLineEdit::MyLineEdit(QWidget *parent)
   : QLineEdit(parent), id(-1)
{}

bool MyLineEdit::event(QEvent *e)
{
   if (e->type() == QEvent::Shortcut) {
      QShortcutEvent *s = static_cast<QShortcutEvent*>(e);
      if (id == s->shortcutId()) {
         setFocus(Qt::ShortcutFocusReason);
         return true;
      }
   }
   return QLineEdit::event(e);
}
```

Листинг 4.37. Содержимое файла widget.h

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QPushButton>
#include <QVBoxLayout>
#include "mylineedit.h"

class Widget : public QWidget
{
   Q_OBJECT

public:
   Widget(QWidget *parent=nullptr);
   ~Widget();
private slots:
   void on_btn1_clicked();
private:
   QLabel *label;
   QLineEdit *line1;
   MyLineEdit *line2;
   QPushButton *btn1;
   QVBoxLayout *vbox;
};
#endif // WIDGET_H
```

Листинг 4.38. Содержимое файла widget.cpp

```
#include "widget.h"

Widget::Widget(QWidget *parent)
   : QWidget(parent)
{
   label = new QLabel("Устано&вить фокус на поле 1");
   line1 = new QLineEdit();
   label->setBuddy(line1);
   line2 = new MyLineEdit();
   line2->id = line2->grabShortcut(QKeySequence::mnemonic("&e"));
   btn1 = new QPushButton("&Убрать фокус с поля 1");
   vbox = new QVBoxLayout();
   vbox->addWidget(label);
   vbox->addWidget(line1);
   vbox->addWidget(line2);
   vbox->addWidget(btn1);
   setLayout(vbox);
   QObject::connect(btn1, SIGNAL(clicked()),
                    this, SLOT(on_btn1_clicked()));
}

void Widget::on_btn1_clicked()
{
   line1->clearFocus();
}

Widget::~Widget() {}
```

Листинг 4.39. Содержимое файла main.cpp

```
#include "widget.h"

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   Widget window;
   window.setWindowTitle("Назначение клавиш быстрого доступа");
   window.resize(350, 150);
   window.show();
   return app.exec();
}
```

Помимо рассмотренных способов для назначения клавиш быстрого доступа можно воспользоваться классом `QShortcut`. В этом случае назначение клавиш для первого текстового поля будет выглядеть так:

```
// widget.h
QShortcut *shortcut;
// widget.cpp
line1 = new QLineEdit();
shortcut = new QShortcut(QKeySequence::mnemonic("&ф"), this);
shortcut->setContext(Qt::WindowShortcut);
QObject::connect(shortcut, SIGNAL(activated()),
                 line1, SLOT(setFocus()));
```

Назначить комбинацию быстрых клавиш позволяет также класс `QAction`. Назначение клавиш для первого текстового поля выглядит следующим образом:

```
// widget.h
QAction *action;
// widget.cpp
line1 = new QLineEdit();
action = new QAction(this);
action->setShortcut(QKeySequence::mnemonic("&ф"));
QObject::connect(action, SIGNAL(triggered()),
                 line1, SLOT(setFocus()));
addAction(action);
```

4.8.3. Нажатие и отпускание клавиши клавиатуры

При нажатии и отпускании клавиши вызываются следующие методы:

- `keyPressEvent()` — вызывается при нажатии клавиши клавиатуры. Если клавишу удерживать нажатой, то событие генерируется постоянно, пока клавиша не будет отпущена. Прототип метода:

  ```
  virtual void keyPressEvent(QKeyEvent *event)
  ```

- `keyReleaseEvent()` — вызывается при отпускании ранее нажатой клавиши. Прототип метода:

  ```
  virtual void keyReleaseEvent(QKeyEvent *event)
  ```

Через параметр доступен экземпляр класса `QKeyEvent`, который позволяет получить дополнительную информацию о событии. Класс `QKeyEvent` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации по классу `QKeyEvent`):

- `key()` — возвращает код нажатой клавиши. Прототип метода:

  ```
  int key() const
  ```

  Пример определения клавиши:

  ```
  if (e->key() == Qt::Key_B) {
     qDebug() << "Нажата клавиша <B>";
  }
  ```

- `text()` — возвращает текстовое представление символа в кодировке Unicode. Если клавиша является специальной, то возвращается пустая строка. Прототип метода:

```
QString text() const
```

- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с клавишей. Прототип метода:

```
Qt::KeyboardModifiers modifiers() const
```

  Может содержать значения следующих констант (или комбинацию значений):

  - `Qt::NoModifier` — модификаторы не нажаты;
  - `Qt::ShiftModifier` — нажата клавиша <Shift>;
  - `Qt::ControlModifier` — нажата клавиша <Ctrl>;
  - `Qt::AltModifier` — нажата клавиша <Alt>;
  - `Qt::MetaModifier` — нажата клавиша <Meta>;
  - `Qt::KeypadModifier`;
  - `Qt::GroupSwitchModifier`.

  Пример определения модификатора <Shift>:

```
if (e->modifiers() & Qt::ShiftModifier) {
   qDebug() << "Нажат модификатор Shift";
}
```

- `isAutoRepeat()` — возвращает значение `true`, если событие вызвано повторно удержанием клавиши нажатой, и `false` — в противном случае. Прототип метода:

```
bool isAutoRepeat() const
```

- `matches()` — возвращает значение `true`, если нажата специальная комбинация клавиш, соответствующая указанному значению, и `false` — в противном случае. Прототип метода:

```
bool matches(QKeySequence::StandardKey key) const
```

  В качестве значения указываются константы из класса `QKeySequence`, например `QKeySequence::Copy` для комбинации клавиш <Ctrl>+<C> (копировать). Полный список констант смотрите в документации по классу `QKeySequence`. Пример:

```
if (e->matches(QKeySequence::Copy)) {
   qDebug() << "Нажата комбинация <Ctrl>+<C>";
}
```

При обработке нажатия клавиш следует учитывать, что:

- компонент должен иметь возможность принимать фокус ввода. Некоторые компоненты не могут принимать фокус ввода по умолчанию, например надпись. Чтобы изменить способ получения фокуса, следует воспользоваться методом `setFocusPolicy()`, который мы рассматривали в *разд. 4.8.1*;

- чтобы захватить эксклюзивный ввод с клавиатуры, следует воспользоваться методом `grabKeyboard()`, а чтобы освободить ввод — методом `releaseKeyboard()`;
- можно перехватить нажатие любых клавиш, кроме клавиши <Tab> и комбинации <Shift>+<Tab>. Эти клавиши используются для передачи фокуса следующему и предыдущему компоненту соответственно. Перехватить нажатие этих клавиш можно только в методе `event()`;
- если событие обработано, то нужно вызвать метод `accept()` через объект события. Чтобы родительский компонент мог получить событие, вместо метода `accept()` необходимо вызвать метод `ignore()`.

4.9. События мыши

События мыши обрабатываются не реже, чем события клавиатуры. С помощью специальных методов можно обработать нажатие и отпускание кнопки мыши, перемещение указателя, а также вхождение указателя в область компонента и выхода из этой области. В зависимости от ситуации можно изменить вид указателя, например при выполнении длительной операции отобразить указатель в виде песочных часов. В этом разделе мы рассмотрим изменение вида указателя мыши как для отдельного компонента, так и для всего приложения.

4.9.1. Нажатие и отпускание кнопки мыши

При нажатии и отпускании кнопки мыши вызываются следующие методы:

- `mousePressEvent()` — вызывается при нажатии кнопки мыши. Прототип метода:

```
virtual void mousePressEvent(QMouseEvent *event)
```

- `mouseReleaseEvent()` — вызывается при отпускании ранее нажатой кнопки мыши. Прототип метода:

```
virtual void mouseReleaseEvent(QMouseEvent *event)
```

- `mouseDoubleClickEvent()` — вызывается при двойном щелчке мышью в области компонента. Прототип метода:

```
virtual void mouseDoubleClickEvent(QMouseEvent *event)
```

Следует учитывать, что двойному щелчку предшествуют другие события. Последовательность событий при двойном щелчке выглядит так:

```
Событие MouseButtonPress
Событие MouseButtonRelease
Событие MouseButtonDblClick
Событие MouseButtonPress
Событие MouseButtonRelease
```

Задать интервал двойного щелчка позволяет метод `setDoubleClickInterval()` из класса `QApplication`. Получить значение интервала можно с помощью метода `doubleClickInterval()`. Прототипы методов:

```
void setDoubleClickInterval(int)
int doubleClickInterval()
```

Через параметр доступен экземпляр класса `QMouseEvent`, который позволяет получить дополнительную информацию о событии. Класс `QMouseEvent` содержит следующие методы:

- `position()` — возвращает экземпляр класса `QPointF` с вещественными координатами в пределах области компонента. Прототип метода:

  ```
  QPointF position() const
  ```

- `scenePosition()` — возвращает координаты относительно окна или сцены. Прототип метода:

  ```
  QPointF scenePosition() const
  ```

- `globalPosition()` — возвращает экземпляр класса `QPointF` с координатами в пределах экрана. Прототип метода:

  ```
  QPointF globalPosition() const
  ```

- `button()` — позволяет определить, какая кнопка мыши вызвала событие. Прототип метода:

  ```
  Qt::MouseButton button() const
  ```

  Возвращает значение одной из следующих констант:

  - `Qt::NoButton` — кнопки не нажаты. Это значение возвращается методом `button()` при перемещении указателя мыши;
  - `Qt::LeftButton` — нажата левая кнопка мыши;
  - `Qt::RightButton` — нажата правая кнопка мыши;
  - `Qt::MiddleButton` — нажата средняя кнопка мыши;

- `buttons()` — позволяет определить все кнопки, которые нажаты одновременно. Возвращает комбинацию значений констант `Qt::LeftButton`, `Qt::RightButton` и `Qt::MiddleButton`. Прототип метода:

  ```
  Qt::MouseButtons buttons() const
  ```

  Пример определения кнопки мыши:

  ```
  if (e->buttons() & Qt::LeftButton) {
     qDebug() << "Нажата левая кнопка мыши";
  }
  ```

- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с кнопкой мыши. Возможные значения мы рассматривали в *разд. 4.8.3*. Прототип метода:

  ```
  Qt::KeyboardModifiers modifiers() const
  ```

Если событие обработано, то нужно вызвать метод `accept()` через объект события. Чтобы родительский компонент мог получить событие, вместо метода `accept()` необходимо вызвать метод `ignore()`.

Если для компонента опция Qt::WA_NoMousePropagation установлена в истинное значение, то событие мыши не будет передаваться родительскому компоненту. Значение можно изменить с помощью метода setAttribute():

```
setAttribute(Qt::WA_NoMousePropagation, true);
```

По умолчанию событие мыши перехватывает компонент, над которым произведен щелчок мышью. Чтобы перехватывать нажатие и отпускание мыши вне компонента, следует захватить мышь с помощью метода grabMouse(). Освободить захваченную ранее мышь позволяет метод releaseMouse(). Прототипы методов:

```
void grabMouse()
void grabMouse(const QCursor &cursor)
void releaseMouse()
```

4.9.2. Перемещение указателя

Чтобы обработать перемещение указателя мыши, необходимо переопределить метод mouseMoveEvent(). Прототип метода:

```
virtual void mouseMoveEvent(QMouseEvent *event)
```

Через параметр доступен экземпляр класса QMouseEvent, который позволяет получить дополнительную информацию о событии. Методы этого класса мы рассматривали в предыдущем разделе. Следует учитывать, что метод button() при перемещении мыши возвращает значение константы Qt::NoButton.

По умолчанию метод mouseMoveEvent() вызывается только в том случае, если при перемещении удерживается нажатой какая-либо кнопка мыши. Это сделано специально, чтобы не создавать лишних событий при обычном перемещении указателя мыши. Если необходимо обрабатывать любые перемещения указателя в пределах компонента, то следует вызвать метод setMouseTracking() из класса QWidget и передать ему значение true. Чтобы обработать все перемещения внутри окна, нужно дополнительно захватить мышь с помощью метода grabMouse(). Прототипы методов:

```
void setMouseTracking(bool enable)
void grabMouse()
void grabMouse(const QCursor &cursor)
```

Метод position() объекта события возвращает позицию точки в системе координат компонента. Чтобы преобразовать эти координаты точки в систему координат родительского компонента или в глобальную систему координат, следует воспользоваться следующими методами из класса QWidget:

- mapToGlobal() — преобразует координаты точки из системы координат компонента в глобальную систему координат. Прототипы метода:

  ```
  QPoint mapToGlobal(const QPoint &pos) const
  QPointF mapToGlobal(const QPointF &pos) const
  ```

- mapFromGlobal() — преобразует координаты точки из глобальной системы координат в систему координат компонента. Прототипы метода:

```
QPoint mapFromGlobal(const QPoint &pos) const
QPointF mapFromGlobal(const QPointF &pos) const
```

- `mapToParent()` — преобразует координаты точки из системы координат компонента в систему координат родительского компонента. Если компонент не имеет родителя, то метод аналогичен методу `mapToGlobal()`. Прототипы метода:

```
QPoint mapToParent(const QPoint &pos) const
QPointF mapToParent(const QPointF &pos) const
```

- `mapFromParent()` — преобразует координаты точки из системы координат родительского компонента в систему координат данного компонента. Если компонент не имеет родителя, то метод аналогичен методу `mapFromGlobal()`. Прототипы метода:

```
QPoint mapFromParent(const QPoint &pos) const
QPointF mapFromParent(const QPointF &pos) const
```

- `mapTo()` — преобразует координаты точки из системы координат компонента в систему координат родительского компонента `parent`. Прототипы метода:

```
QPoint mapTo(const QWidget *parent, const QPoint &pos) const
QPointF mapTo(const QWidget *parent, const QPointF &pos) const
```

- `mapFrom()` — преобразует координаты точки из системы координат родительского компонента `parent` в систему координат данного компонента. Прототипы метода:

```
QPoint mapFrom(const QWidget *parent, const QPoint &pos) const
QPointF mapFrom(const QWidget *parent, const QPointF &pos) const
```

4.9.3. Наведение и выведение указателя

Обработать наведение указателя мыши на компонент и выведение указателя позволяют следующие методы:

- `enterEvent()` — вызывается при наведении указателя мыши на область компонента. Прототип метода:

```
virtual void enterEvent(QEnterEvent *event)
```

- `leaveEvent()` — вызывается, когда указатель мыши покидает область компонента. Прототип метода:

```
virtual void leaveEvent(QEvent *event)
```

4.9.4. Прокрутка колесика мыши

Некоторые мыши комплектуются колесиком, которое обычно используется для управления прокруткой некоторой области. Обработать поворот этого колесика позволяет метод `wheelEvent()`. Прототип метода:

```
virtual void wheelEvent(QWheelEvent *event)
```

Через параметр доступен экземпляр класса QWheelEvent, который позволяет получить дополнительную информацию о событии. Класс QWheelEvent содержит следующие методы:

- angleDelta() — возвращает угол поворота колесика по осям X и Y. Значение может быть положительным или отрицательным в зависимости от направления поворота. Прототип метода:

  ```
  QPoint angleDelta() const
  ```

- position() — возвращает экземпляр класса QPointF с вещественными координатами в пределах области компонента. Прототип метода:

  ```
  QPointF position() const
  ```

- scenePosition() — возвращает координаты относительно окна или сцены. Прототип метода:

  ```
  QPointF scenePosition() const
  ```

- globalPosition() — возвращает экземпляр класса QPointF с координатами в пределах экрана. Прототип метода:

  ```
  QPointF globalPosition() const
  ```

- buttons() — позволяет определить все кнопки, которые нажаты одновременно. Возвращает комбинацию значений констант Qt::LeftButton, Qt::RightButton и Qt::MiddleButton. Прототип метода:

  ```
  Qt::MouseButtons buttons() const
  ```

  Пример определения кнопки мыши:

  ```
  if (e->buttons() & Qt::LeftButton) {
     qDebug() << "Нажата левая кнопка мыши";
  }
  ```

- modifiers() — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с кнопкой мыши. Возможные значения мы рассматривали в *разд. 4.8.3*. Прототип метода:

  ```
  Qt::KeyboardModifiers modifiers() const
  ```

Если событие обработано, то нужно вызвать метод accept() через объект события. Чтобы родительский компонент мог получить событие, вместо метода accept() необходимо вызвать метод ignore().

4.9.5. Изменение внешнего вида указателя мыши

Для изменения внешнего вида указателя мыши при вхождении указателя в область компонента предназначены следующие методы из класса QWidget:

- setCursor() — задает внешний вид указателя мыши для компонента. Прототип метода:

  ```
  void setCursor(const QCursor &)
  ```

В качестве параметра указывается экземпляр класса `QCursor`. Конструктору класса можно передать следующие константы: `Qt::ArrowCursor` (стандартная стрелка), `Qt::UpArrowCursor` (стрелка, направленная вверх), `Qt::CrossCursor` (крестообразный указатель), `Qt::WaitCursor` (курсор выполнения операции, например песочные часы), `Qt::IBeamCursor` (I-образный указатель), `Qt::SizeVerCursor` (стрелки, направленные вверх и вниз), `Qt::SizeHorCursor` (стрелки, направленные влево и вправо), `Qt::SizeBDiagCursor`, `Qt::SizeFDiagCursor`, `Qt::SizeAllCursor` (стрелки, направленные вверх, вниз, влево и вправо), `Qt::BlankCursor` (пустой указатель), `Qt::SplitVCursor`, `Qt::SplitHCursor`, `Qt::PointingHandCursor` (указатель в виде руки), `Qt::ForbiddenCursor` (перечеркнутый круг), `Qt::OpenHandCursor` (разжатая рука), `Qt::ClosedHandCursor` (сжатая рука), `Qt::WhatsThisCursor` (стрелка с вопросительным знаком), `Qt::BusyCursor` (стрелка с песочными часами или вращающимся кругом), `Qt::DragMoveCursor`, `Qt::DragCopyCursor` и `Qt::DragLinkCursor`. Пример:

```
setCursor(QCursor(Qt::WaitCursor));
```

- `unsetCursor()` — отменяет установку указателя для компонента. В результате внешний вид указателя мыши будет наследоваться от родительского компонента. Прототип метода:

  ```
  void unsetCursor()
  ```

- `cursor()` — возвращает экземпляр класса `QCursor` с текущим курсором. Прототип метода:

  ```
  QCursor cursor() const
  ```

Управлять текущим видом курсора для всего приложения сразу можно с помощью следующих статических методов из класса `QApplication`:

- `setOverrideCursor()` — задает внешний вид указателя мыши для всего приложения. Прототип метода:

  ```
  static void setOverrideCursor(const QCursor &cursor)
  ```

  В качестве параметра указывается экземпляр класса `QCursor`. Для отмены установки необходимо вызвать метод `restoreOverrideCursor()`;

- `restoreOverrideCursor()` — отменяет изменение внешнего вида курсора для всего приложения. Прототип метода:

  ```
  static void restoreOverrideCursor()
  ```

  Пример:

  ```
  QApplication::setOverrideCursor(QCursor(Qt::WaitCursor));
  // Выполняем длительную операцию
  QApplication::restoreOverrideCursor();
  ```

- `changeOverrideCursor()` — изменяет внешний вид указателя мыши для всего приложения. Если до вызова этого метода не вызывался метод `setOverrideCursor()`, то указанное значение игнорируется. В качестве параметра указывается экземпляр класса `QCursor`. Прототип метода:

  ```
  static void changeOverrideCursor(const QCursor &cursor)
  ```

- `overrideCursor()` — возвращает указатель на экземпляр класса `QCursor` с текущим курсором или нулевой указатель. Прототип метода:

```
static QCursor *overrideCursor()
```

Изменять внешний вид указателя мыши для всего приложения принято на небольшой промежуток времени, обычно на время выполнения какой-либо операции, в процессе которой приложение не может нормально реагировать на действия пользователя. Чтобы информировать об этом пользователя, указатель принято выводить в виде песочных часов или вращающегося круга (константа `Qt::WaitCursor`).

Метод `setOverrideCursor()` может быть вызван несколько раз. В этом случае курсоры помещаются в стек. Каждый вызов метода `restoreOverrideCursor()` удаляет последний курсор, добавленный в стек. Для нормальной работы приложения необходимо вызывать методы `setOverrideCursor()` и `restoreOverrideCursor()` одинаковое количество раз.

Класс `QCursor` позволяет создать объект курсора с изображением любой формы. Прототипы конструкторов:

```
QCursor()
QCursor(Qt::CursorShape shape)
QCursor(const QPixmap &pixmap, int hotX = -1, int hotY = -1)
QCursor(const QBitmap &bitmap, const QBitmap &mask, int hotX = -1,
        int hotY = -1)
QCursor(const QCursor &c)
QCursor(QCursor &&other)
```

Чтобы загрузить изображение, следует передать путь к файлу конструктору класса `QPixmap`. Чтобы создать объект курсора, необходимо передать конструктору класса `QCursor` в первом параметре экземпляр класса `QPixmap`, а во втором и третьем параметрах — координаты «горячей» точки. Пример создания и установки пользовательского курсора:

```
QPixmap pix("C:\\cpp\\projectsQt\\Test\\cursor.png");
setCursor(QCursor(pix, 0, 0));
```

Класс `QCursor` содержит также два статических метода:

- `pos()` — возвращает экземпляр класса `QPoint` с координатами указателя мыши относительно экрана. Прототипы метода:

```
static QPoint pos()
static QPoint pos(const QScreen *screen)
```

Пример:

```
qDebug() << QCursor::pos(); // QPoint(1024,520)
```

- `setPos()` — позволяет задать позицию указателя мыши. Прототипы метода:

```
static void setPos(int x, int y)
static void setPos(const QPoint &p)
static void setPos(QScreen *screen, int x, int y)
static void setPos(QScreen *screen, const QPoint &p)
```

4.10. Технология drag & drop

Технология drag & drop позволяет обмениваться данными различных типов между компонентами как одного приложения, так и разных приложений путем перетаскивания и сбрасывания объектов с помощью мыши. Типичным примером использования технологии служит перемещение файлов в программе Проводник в Windows. Чтобы переместить файл в другой каталог, достаточно нажать левую кнопку мыши над значком файла и, не отпуская кнопку, перетащить файл на значок каталога, а затем отпустить кнопку мыши. Если необходимо скопировать файл, а не переместить, то следует дополнительно удерживать нажатой клавишу <Ctrl>.

4.10.1. Запуск перетаскивания

Операция перетаскивания состоит из двух частей. Первая часть запускает процесс, а вторая часть обрабатывает момент сброса объекта. Обе части могут обрабатываться как одним приложением, так и двумя разными приложениями. Запуск перетаскивания осуществляется следующим образом:

- внутри метода `mousePressEvent()` запоминаются координаты щелчка левой кнопкой мыши;
- внутри метода `mouseMoveEvent()` вычисляется пройденное расстояние или измеряется время операции. Это необходимо сделать, чтобы предотвратить случайное перетаскивание. Управлять задержкой позволяют следующие статические методы класса `QApplication`:
  - `startDragDistance()` — возвращает минимальное расстояние, после прохождения которого можно запускать операцию перетаскивания;
  - `setStartDragDistance()` — задает расстояние;
  - `startDragTime()` — возвращает время задержки в миллисекундах перед запуском операции перетаскивания;
  - `setStartDragTime()` — задает время задержки;

  Прототипы методов:

```
static int startDragDistance()
static void setStartDragDistance(int l)
static int startDragTime()
static void setStartDragTime(int ms)
```

- если пройдено минимальное расстояние или истек минимальный промежуток времени, то создается экземпляр класса `QDrag` и вызывается метод `exec()`, который после завершения операции возвращает действие, выполненное с данными (например, данные скопированы или перемещены).

Создать экземпляр класса `QDrag` можно так:

```
QDrag *drag = new QDrag(this);
```

Класс `QDrag` содержит следующие методы:

- `exec()` — запускает процесс перетаскивания и возвращает действие, которое было выполнено по завершении операции. Прототипы метода:

```
Qt::DropAction exec(Qt::DropActions supportedActions = Qt::MoveAction)
Qt::DropAction exec(Qt::DropActions supportedActions,
                    Qt::DropAction defaultDropAction)
```

  В параметре `supportedActions` указывается комбинация допустимых действий, а в параметре `defaultDropAction` — действие, которое используется, если не нажаты клавиши-модификаторы. Возможные действия могут быть заданы следующими константами: `Qt::CopyAction` (копирование), `Qt::MoveAction` (перемещение), `Qt::LinkAction` (ссылка), `Qt::IgnoreAction` (действие проигнорировано), `Qt::TargetMoveAction`. Пример:

```
Qt::DropAction action = drag->exec(
              Qt::MoveAction | Qt::CopyAction, Qt::MoveAction);
```

- `setMimeData()` — позволяет задать перемещаемые данные. В качестве значения указывается экземпляр класса `QMimeData`. Прототип метода:

```
void setMimeData(QMimeData *data)
```

  Пример передачи текста:

```
QMimeData *data = new QMimeData();
data->setText("Перетаскиваемый текст");
QDrag *drag = new QDrag(this);
drag->setMimeData(data);
```

- `mimeData()` — возвращает указатель на экземпляр класса `QMimeData` с перемещаемыми данными. Прототип метода:

```
QMimeData *mimeData() const
```

- `setPixmap()` — задает изображение, которое будет перемещаться вместе с указателем мыши. В качестве параметра указывается экземпляр класса `QPixmap`. Прототип метода:

```
void setPixmap(const QPixmap &pixmap)
```

  Пример:

```
drag->setPixmap(QPixmap("C:\\cpp\\projectsQt\\Test\\pixmap.png"));
```

- `pixmap()` — возвращает экземпляр класса `QPixmap` с изображением, которое перемещается вместе с указателем. Прототип метода:

```
QPixmap pixmap() const
```

- `setHotSpot()` — задает координаты «горячей» точки на перемещаемом изображении. В качестве параметра указывается экземпляр класса `QPoint`. Прототип метода:

```
void setHotSpot(const QPoint &hotspot)
```

Пример:

```
drag->setHotSpot(QPoint(40, 40));
```

- `hotSpot()` — возвращает экземпляр класса `QPoint` с координатами «горячей» точки на перемещаемом изображении. Прототип метода:

```
QPoint hotSpot() const
```

- `setDragCursor()` — позволяет изменить внешний вид указателя мыши для действия, указанного во втором параметре. Прототип метода:

```
void setDragCursor(const QPixmap &cursor, Qt::DropAction action)
```

В первом параметре указывается экземпляр класса `QPixmap`, а во втором параметре — константы `Qt::CopyAction`, `Qt::MoveAction` или `Qt::LinkAction`. Пример изменения указателя для перемещения:

```
drag->setDragCursor(
         QPixmap("C:\\cpp\\projectsQt\\Test\\cursor.png"),
         Qt::MoveAction);
```

- `source()` — возвращает указатель на компонент-источник. Прототип метода:

```
QObject *source() const
```

- `target()` — возвращает указатель на компонент-приемник или нулевой указатель, если компонент находится в другом приложении. Прототип метода:

```
QObject *target() const
```

Класс `QDrag` поддерживает два сигнала:

- `actionChanged(Qt::DropAction)` — генерируется при изменении действия;
- `targetChanged(QObject*)` — генерируется при изменении принимающего компонента.

Пример назначения обработчика сигнала `actionChanged()`:

```
connect(drag, SIGNAL(actionChanged(Qt::DropAction)),
        this, SLOT(on_action_changed(Qt::DropAction)));
```

4.10.2. Класс *QMimeData*

Перемещаемые данные и сведения о MIME-типе должны быть представлены классом `QMimeData`. Экземпляр этого класса необходимо передать в метод `setMimeData()` класса `QDrag`. Создание экземпляра класса `QMimeData` выглядит так:

```
// #include <QMimeData>
QMimeData *data = new QMimeData();
```

Класс `QMimeData` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации по классу `QMimeData`):

- `setText()` — устанавливает текстовые данные (MIME-тип — `text/plain`). Прототип метода:

```
void setText(const QString &text)
```

Пример указания значения:

```
data->setText("Перетаскиваемый текст");
```

- `text()` — возвращает текстовые данные (MIME-тип — `text/plain`). Прототип метода:

```
QString text() const
```

- `hasText()` — возвращает значение `true`, если объект содержит текстовые данные (MIME-тип — `text/plain`), и `false` — в противном случае. Прототип метода:

```
bool hasText() const
```

- `setHtml()` — устанавливает текстовые данные в формате HTML (MIME-тип — `text/html`). Прототип метода:

```
void setHtml(const QString &html)
```

Пример указания значения:

```
data->setHtml("<b>Перетаскиваемый HTML-текст</b>");
```

- `html()` — возвращает текстовые данные в формате HTML (MIME-тип — `text/html`). Прототип метода:

```
QString html() const
```

- `hasHtml()` — возвращает значение `true`, если объект содержит текстовые данные в формате HTML (MIME-тип — `text/html`), и `false` — в противном случае. Прототип метода:

```
bool hasHtml() const
```

- `setUrls()` — устанавливает список URI-адресов (MIME-тип — `text/uri-list`). В качестве значения указывается список с экземплярами класса `QUrl`. С помощью этого MIME-типа можно обработать перетаскивание файлов. Прототип метода:

```
void setUrls(const QList<QUrl> &urls)
```

Пример указания значения:

```
QList<QUrl> list;
list << QUrl("http://google.ru/");
data->setUrls(list);
```

- `urls()` — возвращает список URI-адресов (MIME-тип — `text/uri-list`). Прототип метода:

```
QList<QUrl> urls() const
```

Пример получения первого URI-адреса:

```
setText(e->mimeData()->urls()[0].toString());
```

- `hasUrls()` — возвращает значение `true`, если объект содержит список URI-адресов (MIME-тип — `text/uri-list`), и `false` — в противном случае. Прототип метода:

```
bool hasUrls() const
```

- `setImageData()` — устанавливает изображение (MIME-тип — `image/*`). В качестве значения можно указать, например, экземпляр класса `QImage` или `QPixmap`. Прототип метода:

  ```
  void setImageData(const QVariant &image)
  ```

  Пример указания значения:

  ```
  data->setImageData(QImage("C:\\cpp\\projectsQt\\Test\\pixmap.png"));
  ```

- `imageData()` — возвращает объект изображения. Прототип метода:

  ```
  QVariant imageData() const
  ```

  Пример:

  ```
  QImage image = qvariant_cast<QImage>(e->mimeData()->imageData());
  setPixmap(QPixmap::fromImage(image));
  ```

- `hasImage()` — возвращает значение `true`, если объект содержит изображение (MIME-тип — `image/*`), и `false` — в противном случае. Прототип метода:

  ```
  bool hasImage() const
  ```

- `setData()` — позволяет установить данные пользовательского MIME-типа. В первом параметре указывается MIME-тип в виде строки, а во втором параметре — экземпляр класса `QByteArray` с данными. Метод можно вызвать несколько раз с различными MIME-типами. Прототип метода:

  ```
  void setData(const QString &mimeType, const QByteArray &data)
  ```

- `data()` — возвращает экземпляр класса `QByteArray` с данными, соответствующими указанному MIME-типу. Прототип метода:

  ```
  QByteArray data(const QString &mimeType) const
  ```

- `hasFormat()` — возвращает значение `true`, если объект содержит данные в указанном MIME-типе, и `false` — в противном случае. Прототип метода:

  ```
  virtual bool hasFormat(const QString &mimeType) const
  ```

- `formats()` — возвращает список с поддерживаемыми объектом MIME-типами. Прототип метода:

  ```
  virtual QStringList formats() const
  ```

- `removeFormat()` — удаляет данные, соответствующие указанному MIME-типу. Прототип метода:

  ```
  void removeFormat(const QString &mimeType)
  ```

- `clear()` — удаляет все данные и информацию о MIME-типе. Прототип метода:

  ```
  void clear()
  ```

Если необходимо перетаскивать данные какого-либо специфического типа, то нужно наследовать класс `QMimeData` и переопределить методы `retrieveData()` и `formats()`. За подробной информацией по этому вопросу обращайтесь к документации.

4.10.3. Обработка сброса

Прежде чем обрабатывать перетаскивание и сбрасывание объекта, необходимо сообщить системе, что компонент может обрабатывать эти события. Для этого внутри конструктора компонента следует вызвать метод `setAcceptDrops()` из класса `QWidget` и передать ему значение `true`. Прототип метода:

```
void setAcceptDrops(bool on)
```

Пример:

```
setAcceptDrops(true);
```

Обработка перетаскивания и сброса объекта выполняется следующим образом:

- внутри метода `dragEnterEvent()` проверяется MIME-тип перетаскиваемых данных и действие. Если компонент способен обработать сброс этих данных и соглашается с предложенным действием, то необходимо вызвать метод `acceptProposedAction()` через объект события. Если нужно изменить действие, то методу `setDropAction()` передается новое действие, а затем вызывается метод `accept()`, а не метод `acceptProposedAction()`;
- если необходимо ограничить область сброса некоторым участком компонента, то можно дополнительно определить метод `dragMoveEvent()`. Этот метод будет постоянно вызываться при перетаскивании внутри области компонента. При согласии со сбрасыванием следует вызвать метод `accept()`, которому можно передать экземпляр класса `QRect` с координатами и размером участка. Если параметр указан, то при перетаскивании внутри участка метод `dragMoveEvent()` повторно вызываться не будет;
- внутри метода `dropEvent()` производится обработка сброса.

Обработать события, возникающие при перетаскивании и сбрасывании объектов, позволяют следующие методы:

- `dragEnterEvent()` — вызывается, когда перетаскиваемый объект входит в область компонента. Через параметр доступен указатель на экземпляр класса `QDragEnterEvent`. Прототип метода:

  ```
  virtual void dragEnterEvent(QDragEnterEvent *event)
  ```

- `dragLeaveEvent()` — вызывается, когда перетаскиваемый объект покидает область компонента. Через параметр доступен указатель на экземпляр класса `QDragLeaveEvent`. Прототип метода:

  ```
  virtual void dragLeaveEvent(QDragLeaveEvent *event)
  ```

- `dragMoveEvent()` — вызывается при перетаскивании объекта внутри области компонента. Через параметр доступен указатель на экземпляр класса `QDragMoveEvent`. Прототип метода:

  ```
  virtual void dragMoveEvent(QDragMoveEvent *event)
  ```

- `dropEvent()` — вызывается при сбрасывании объекта в области компонента. Через параметр доступен указатель на экземпляр класса `QDropEvent`. Прототип метода:

  ```
  virtual void dropEvent(QDropEvent *event)
  ```

Класс `QDragLeaveEvent` наследует класс `QEvent` и не несет никакой дополнительной информации. Достаточно просто знать, что перетаскиваемый объект покинул область компонента. Цепочка наследования остальных классов выглядит так:

`QEvent — QDropEvent — QDragMoveEvent — QDragEnterEvent`

Класс `QDragEnterEvent` не содержит собственных методов, но наследует все методы классов `QDropEvent` и `QDragMoveEvent`.

Класс `QDropEvent` содержит следующие методы:

- `mimeData()` — возвращает указатель на экземпляр класса `QMimeData` с перемещаемыми данными и информацией о MIME-типе. Прототип метода:

  ```
  const QMimeData *mimeData() const
  ```

- `position()` — возвращает экземпляр класса `QPointF` с координатами сбрасывания объекта. Прототип метода:

  ```
  QPointF position() const
  ```

- `possibleActions()` — возвращает комбинацию возможных действий при сбрасывании. Прототип метода:

  ```
  Qt::DropActions possibleActions() const
  ```

  Пример определения значений:

  ```
  if (e->possibleActions() & Qt::MoveAction) {
     qDebug() << "MoveAction";
  }
  if (e->possibleActions() & Qt::CopyAction) {
     qDebug() << "CopyAction";
  }
  ```

- `proposedAction()` — возвращает действие по умолчанию при сбрасывании. Прототип метода:

  ```
  Qt::DropAction proposedAction() const
  ```

- `acceptProposedAction()` — устанавливает флаг готовности принять перемещаемые данные и согласия с действием, возвращаемым методом `proposedAction()`. Метод `acceptProposedAction()` (или метод `accept()`) необходимо вызвать внутри метода `dragEnterEvent()`, иначе метод `dropEvent()` вызван не будет. Прототип метода:

  ```
  void acceptProposedAction()
  ```

- `setDropAction()` — позволяет изменить действие при сбрасывании. После изменения действия следует вызвать метод `accept()`, а не `acceptProposedAction()`. Прототип метода:

  ```
  void setDropAction(Qt::DropAction action)
  ```

- `dropAction()` — возвращает действие, которое должно быть выполнено при сбрасывании. Возвращаемое значение может не совпадать со значением, возвращаемым методом `proposedAction()`, если действие было изменено с помощью метода `setDropAction()`. Прототип метода:

```
Qt::DropAction dropAction() const
```

- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с кнопкой мыши. Возможные значения мы рассматривали в *разд. 4.8.3*. Прототип метода:

```
Qt::KeyboardModifiers modifiers() const
```

- `buttons()` — позволяет определить кнопки мыши, которые нажаты. Прототип метода:

```
Qt::MouseButtons buttons() const
```

- `source()` — возвращает указатель на компонент внутри приложения, являющийся источником события, или нулевой указатель. Прототип метода:

```
QObject *source() const
```

Теперь рассмотрим методы класса `QDragMoveEvent`:

- `accept()` — устанавливает флаг, который является признаком согласия с дальнейшей обработкой события. В качестве параметра можно указать экземпляр класса `QRect` с координатами и размерами прямоугольной области, в которой сбрасывание будет принято. Прототипы метода:

```
void accept()
void accept(const QRect &rectangle)
```

- `ignore()` — сбрасывает флаг, что является запретом на дальнейшую обработку события. В качестве параметра можно указать экземпляр класса `QRect` с координатами и размерами прямоугольной области, в которой сбрасывание выполнить нельзя. Прототипы метода:

```
void ignore()
void ignore(const QRect &rectangle)
```

- `answerRect()` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, в которой произойдет сбрасывание, если событие будет принято. Прототип метода:

```
QRect answerRect() const
```

Некоторые компоненты в Qt по умолчанию поддерживают технологию drag & drop, например в однострочное текстовое поле можно перетащить текст из другого приложения. Поэтому, прежде чем изобретать свой «велосипед», убедитесь, что поддержка технологии в компоненте не реализована.

4.11. Работа с буфером обмена

Помимо технологии drag & drop для обмена данными между приложениями используется *буфер обмена*. Одно приложение помещает данные в буфер обмена, а второе приложение (или то же самое) может извлечь их из буфера. Получить указатель на глобальный объект буфера обмена позволяет статический метод `clipboard()` из класса `QApplication`:

```
QClipboard *clipboard()
```

Пример получения текста из буфера обмена:

```
QString text = QApplication::clipboard()->text();
```

Класс `QClipboard` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации по классу `QClipboard`):

- `setText()` — копирует текст в буфер обмена. Прототип метода:

```
void setText(const QString &text,
             QClipboard::Mode mode = Clipboard)
```

- `text()` — возвращает текст из буфера обмена или пустую строку. Прототипы метода:

```
QString text(QClipboard::Mode mode = Clipboard) const
QString text(QString &subtype,
             QClipboard::Mode mode = Clipboard) const
```

- `setMimeData()` — позволяет сохранить в буфере данные любого типа. В качестве первого параметра указывается экземпляр класса `QMimeData`. Этот класс мы рассматривали при изучении технологии drag & drop *(см. разд. 4.10.2)*. Прототип метода:

```
void setMimeData(QMimeData *src,
                 QClipboard::Mode mode = Clipboard)
```

- `mimeData()` — возвращает указатель на экземпляр класса `QMimeData`. Прототип метода:

```
const QMimeData *mimeData(QClipboard::Mode mode = Clipboard) const
```

- `clear()` — очищает буфер обмена. Прототип метода:

```
void clear(QClipboard::Mode mode = Clipboard)
```

В необязательном параметре `mode` могут быть указаны константы `Clipboard` (используется по умолчанию), `Selection` или `FindBuffer`.

Отследить изменение состояния буфера обмена позволяет сигнал `dataChanged()`. Назначить обработчик этого сигнала можно так:

```
connect(QApplication::clipboard(), SIGNAL(dataChanged()),
        this, SLOT(on_change_clipboard()));
```

ГЛАВА 5

Размещение нескольких компонентов в окне

При размещении нескольких компонентов в окне обычно возникает вопрос взаимного расположения компонентов, а также их минимальных размеров. Следует помнить, что по умолчанию размеры окна можно изменить, взявшись мышью за границу окна, а значит, необходимо перехватывать событие изменения размеров и производить пересчет позиции и размера каждого компонента. Библиотека Qt избавляет нас от лишних проблем и предоставляет множество контейнеров, которые производят перерасчет автоматически. Все, что от нас требуется, — это выбрать нужный контейнер, добавить туда компоненты в определенном порядке, а затем поместить контейнер в окно или в другой контейнер.

5.1. Абсолютное позиционирование

Прежде чем изучать контейнеры, рассмотрим возможность абсолютного позиционирования компонентов в окне. Итак, если при создании компонента передан указатель на родительский компонент, то компонент выводится в позицию с координатами `(0, 0)`. Иными словами, если мы добавим несколько компонентов, то все они отобразятся на одной и той же позиции. Последний добавленный компонент будет на вершине этой кучи, а остальные компоненты будут видны лишь частично или вообще не видны. Размеры добавляемых компонентов будут соответствовать их содержимому.

Для перемещения компонента можно воспользоваться методом `move()`, а для изменения размеров — методом `resize()`. Выполнить одновременное изменение позиции и размеров позволяет метод `setGeometry()`. Все эти методы, а также множество других методов, позволяющих изменять позицию и размеры, мы рассматривали в *разд. 3.3* и *3.4*. Если компонент не имеет родителя, то эти методы изменяют характеристики окна, а если при создании компонента указан родительский компонент, то методы изменяют характеристики только самого компонента.

Для примера выведем внутри окна надпись и кнопку, указав позицию и размеры для каждого компонента (листинг 5.1).

Листинг 5.1. Абсолютное позиционирование

```
#include <QApplication>
#include <QWidget>
#include <QLabel>
#include <QPushButton>

int main(int argc, char *argv[])
{
   QApplication app(argc, argv);
   QWidget window;
   window.setWindowTitle("Абсолютное позиционирование");
   window.resize(300, 120);
   QLabel *label = new QLabel("Текст надписи", &window);
   QPushButton *btn1 = new QPushButton("Текст на кнопке", &window);
   label->setGeometry(10, 10, 280, 60);
   btn1->resize(280, 30);
   btn1->move(10, 80);
   window.show();
   return app.exec();
}
```

Абсолютное позиционирование имеет следующие недостатки:

- при изменении размеров окна необходимо пересчитывать и изменять характеристики всех компонентов вручную;
- при указании фиксированных размеров надписи на компонентах могут выходить за пределы компонента. Помните, что в разных операционных системах используются разные стили оформления, в том числе и характеристики шрифта. Подогнав размеры в одной операционной системе, можно прийти в ужас при виде приложения в другой операционной системе, где размер шрифта в два раза больше. Поэтому лучше вообще отказаться от указания фиксированных размеров или задавать размер и название шрифта для каждого компонента. Кроме того, приложение может поддерживать несколько языков интерфейса. Длина слов в разных языках отличается, что также станет причиной искажения компонентов.

5.2. Горизонтальное и вертикальное выравнивание

Контейнеры (их еще называют *менеджерами компоновки*, *менеджерами геометрии*) лишены недостатков абсолютного позиционирования. При изменении размеров окна производится автоматическое изменение характеристик всех компонентов, добавленных в контейнер. Настройки шрифта при этом также учитываются, поэтому изменение размеров шрифта в два раза приведет только к увеличению компонентов и размеров окна.

Для автоматического выравнивания компонентов используются два класса:

- `QHBoxLayout` — выстраивает все добавляемые компоненты по горизонтали (по умолчанию — слева направо). Конструкторы класса:

```
#include <QHBoxLayout>
QHBoxLayout()
QHBoxLayout(QWidget *parent)
```

- `QVBoxLayout` — выстраивает все добавляемые компоненты по вертикали (по умолчанию — сверху вниз). Конструкторы класса:

```
#include <QVBoxLayout>
QVBoxLayout()
QVBoxLayout(QWidget *parent)
```

Иерархия наследования для классов `QHBoxLayout` и `QVBoxLayout` выглядит так:

```
(QObject, QLayoutItem) — QLayout — QBoxLayout — QHBoxLayout
(QObject, QLayoutItem) — QLayout — QBoxLayout — QVBoxLayout
```

Обратите внимание на то, что классы не являются наследниками класса `QWidget`, следовательно, они не обладают собственным окном и не могут использоваться отдельно. Поэтому контейнеры обязательно должны быть привязаны к родительскому компоненту. Передать указатель на родительский компонент можно через конструктор классов `QHBoxLayout` и `QVBoxLayout`. Кроме того, можно передать указатель на контейнер в метод `setLayout()` родительского компонента. Прототип метода:

```
void setLayout(QLayout *layout)
```

После этого все компоненты, добавленные в контейнер, автоматически привязываются к родительскому компоненту. Типичный пример использования класса `QHBoxLayout` выглядит следующим образом:

```
QWidget window;                                  // Родительский компонент
window.setWindowTitle("Выравнивание по горизонтали");
window.resize(300, 120);
QPushButton *btn1 = new QPushButton("1");
QPushButton *btn2 = new QPushButton("2");
QPushButton *btn3 = new QPushButton("3");
QHBoxLayout *hbox = new QHBoxLayout();           // Создаем контейнер
hbox->addWidget(btn1);                           // Добавляем компоненты
hbox->addWidget(btn2);
hbox->addWidget(btn3);
window.setLayout(hbox);                          // Передаем родителю
window.show();
```

Добавить компоненты в контейнер и удалить их позволяют следующие методы:

- `addWidget()` — добавляет компонент в конец контейнера. Прототип метода:

```
void addWidget(QWidget *widget, int stretch = 0,
               Qt::Alignment alignment = Qt::Alignment())
```

В первом параметре передается указатель на компонент. Необязательный параметр `stretch` задает фактор растяжения для ячейки, а параметр `alignment` — выравнивание компонента внутри ячейки. Пример:

```
hbox->addWidget(btn1, 10, Qt::AlignRight);
```

- `insertWidget()` — добавляет компонент в указанную позицию контейнера. Прототип метода:

```
void insertWidget(int index, QWidget *widget, int stretch = 0,
                  Qt::Alignment alignment = Qt::Alignment())
```

Если в первом параметре указано значение `0`, то компонент будет добавлен в начало контейнера. Если указано отрицательное значение, то компонент добавляется в конец контейнера. Другое значение указывает определенную позицию. Остальные параметры аналогичны параметрам метода `addWidget()`. Пример:

```
hbox->addWidget(btn1);
hbox->insertWidget(-1, btn2); // Добавление в конец
hbox->insertWidget(0, btn3);  // Добавление в начало
```

- `removeWidget()` — удаляет компонент из контейнера. Прототип метода:

```
void removeWidget(QWidget *widget)
```

- `addLayout()` — добавляет другой контейнер в конец текущего контейнера. С помощью этого метода можно вкладывать один контейнер в другой, создавая таким образом структуру любой сложности. Прототип метода:

```
void addLayout(QLayout *layout, int stretch = 0)
```

- `insertLayout()` — добавляет другой контейнер в указанную позицию текущего контейнера. Если в первом параметре указано отрицательное значение, то контейнер добавляется в конец. Прототип метода:

```
void insertLayout(int index, QLayout *layout, int stretch = 0)
```

- `addSpacing()` — добавляет пустое пространство указанного размера в конец контейнера. Прототип метода:

```
void addSpacing(int size)
```

- `insertSpacing()` — добавляет пустое пространство указанного размера в определенную позицию. Если в первом параметре указано отрицательное значение, то пространство добавляется в конец. Прототип метода:

```
void insertSpacing(int index, int size)
```

- `addStretch()` — добавляет пустое растягиваемое пространство с нулевым минимальным размером и фактором растяжения `stretch` в конец контейнера. Это пространство можно сравнить с пружиной, вставленной между компонентами, а параметр `stretch` — с жесткостью пружины. Прототип метода:

```
void addStretch(int stretch = 0)
```

- `insertStretch()` — метод аналогичен методу `addStretch()`, но добавляет растягиваемое пространство в указанную позицию. Если в первом параметре указано

отрицательное значение, то пространство добавляется в конец контейнера. Прототип метода:

```
void insertStretch(int index, int stretch = 0)
```

Параметр `alignment` в методах `addWidget()` и `insertWidget()` задает выравнивание компонента внутри ячейки. В этом параметре можно указать следующие константы:

- `Qt::AlignLeft` — горизонтальное выравнивание по левому краю;
- `Qt::AlignRight` — горизонтальное выравнивание по правому краю;
- `Qt::AlignHCenter` — горизонтальное выравнивание по центру;
- `Qt::AlignJustify` — заполнение всего пространства;
- `Qt::AlignTop` — вертикальное выравнивание по верхнему краю;
- `Qt::AlignBottom` — вертикальное выравнивание по нижнему краю;
- `Qt::AlignVCenter` — вертикальное выравнивание по центру;
- `Qt::AlignBaseline` — вертикальное выравнивание по базовой линии;
- `Qt::AlignCenter` — горизонтальное и вертикальное выравнивание по центру;
- `Qt::AlignAbsolute` — если в методе `setLayoutDirection()` из класса `QWidget` указана константа `Qt::RightToLeft`, то константа `Qt::AlignLeft` задает выравнивание по правому краю, а константа `Qt::AlignRight` — по левому краю. Чтобы константа `Qt::AlignLeft` всегда соответствовала именно левому краю, необходимо указать комбинацию `Qt::AlignAbsolute | Qt::AlignLeft`. Аналогично следует поступить с константой `Qt::AlignRight`.

Можно задавать комбинацию констант. В комбинации допускается указывать только одну константу горизонтального выравнивания и только одну константу вертикального выравнивания. Например, комбинация `Qt::AlignLeft | Qt::AlignTop` задает выравнивание по левому и верхнему краю. Противоречивые значения приводят к непредсказуемым результатам.

Помимо рассмотренных методов контейнеры поддерживают следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setDirection()` — задает направление вывода компонентов. Прототип метода:

  ```
  void setDirection(QBoxLayout::Direction direction)
  ```

  В параметре можно указать следующие константы:

  - `QBoxLayout::LeftToRight` — слева направо (значение по умолчанию для горизонтального контейнера);
  - `QBoxLayout::RightToLeft` — справа налево;
  - `QBoxLayout::TopToBottom` — сверху вниз (значение по умолчанию для вертикального контейнера);
  - `QBoxLayout::BottomToTop` — снизу вверх;

- `setSpacing()` — задает расстояние между компонентами. Прототип метода:

```
void setSpacing(int)
```

- `setContentsMargins()` — задает величину отступа от границ контейнера до компонентов. Прототипы метода:

```
void setContentsMargins(int left, int top, int right, int bottom)
void setContentsMargins(const QMargins &margins)
```

5.3. Выравнивание по сетке

Помимо выравнивания компонентов по горизонтали и вертикали существует возможность размещения компонентов внутри ячеек сетки. Для выравнивания компонентов по сетке предназначен класс `QGridLayout`. Иерархия наследования:

```
(QObject, QLayoutItem) — QLayout — QGridLayout
```

Создать экземпляр класса `QGridLayout` можно с помощью такого конструктора:

```
#include <QGridLayout>
QGridLayout(QWidget *parent = nullptr)
```

В необязательном параметре можно передать указатель на родительский компонент. Если параметр не указан, то необходимо передать указатель на сетку в метод `setLayout()` родительского компонента. Типичный пример использования класса `QGridLayout` выглядит так:

```
QWidget window;                              // Родительский компонент
window.setWindowTitle("Выравнивание по сетке");
window.resize(300, 100);
QPushButton *btn1 = new QPushButton("1");
QPushButton *btn2 = new QPushButton("2");
QPushButton *btn3 = new QPushButton("3");
QPushButton *btn4 = new QPushButton("4");
QGridLayout *grid = new QGridLayout();       // Создаем сетку
grid->addWidget(btn1, 0, 0);                 // Добавляем компоненты
grid->addWidget(btn2, 0, 1);
grid->addWidget(btn3, 1, 0);
grid->addWidget(btn4, 1, 1);
window.setLayout(grid);                      // Передаем родителю
window.show();
```

Добавить компоненты и удалить их позволяют следующие методы:

- `addWidget()` — добавляет компонент в указанную ячейку сетки. Прототипы метода:

```
void addWidget(QWidget *widget, int row, int column,
               Qt::Alignment alignment = Qt::Alignment())
void addWidget(QWidget *widget, int fromRow, int fromColumn,
               int rowSpan, int columnSpan,
               Qt::Alignment alignment = Qt::Alignment())
```

В первом параметре передается указатель на компонент, во втором параметре передается индекс строки, а в третьем — индекс столбца. Нумерация строк и столбцов начинается с нуля. Параметр `rowSpan` задает количество объединенных ячеек по вертикали, а параметр `columnSpan` — по горизонтали. Параметр `alignment` задает выравнивание компонента внутри ячейки. Значения, которые можно указать в этом параметре, мы рассматривали в предыдущем разделе. Пример:

```
QGridLayout *grid = new QGridLayout();
grid->addWidget(btn1, 0, 0, Qt::AlignLeft);
grid->addWidget(btn2, 0, 1, Qt::AlignRight);
grid->addWidget(btn3, 1, 0, 1, 2);
```

- `addLayout()` — добавляет контейнер в указанную ячейку сетки. Прототипы метода:

```
void addLayout(QLayout *layout, int row, int column,
               Qt::Alignment alignment = Qt::Alignment())
void addLayout(QLayout *layout, int row, int column,
               int rowSpan, int columnSpan,
               Qt::Alignment alignment = Qt::Alignment())
```

В первом параметре передается указатель на контейнер. Остальные параметры аналогичны параметрам метода `addWidget()`.

Класс `QGridLayout` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setRowMinimumHeight()` — задает минимальную высоту строки с индексом `row`. Прототип метода:

```
void setRowMinimumHeight(int row, int minSize)
```

- `setColumnMinimumWidth()` — задает минимальную ширину столбца с индексом `column`. Прототип метода:

```
void setColumnMinimumWidth(int column, int minSize)
```

- `setRowStretch()` — задает фактор растяжения для строки с индексом `row`. Прототип метода:

```
void setRowStretch(int row, int stretch)
```

- `setColumnStretch()` — задает фактор растяжения для столбца с индексом `column`. Прототип метода:

```
void setColumnStretch(int column, int stretch)
```

- `setHorizontalSpacing()` — задает расстояние между компонентами по горизонтали. Прототип метода:

```
void setHorizontalSpacing(int spacing)
```

- `setVerticalSpacing()` — задает расстояние между компонентами по вертикали. Прототип метода:

```
void setVerticalSpacing(int spacing)
```

- `rowCount()` — возвращает количество строк сетки. Прототип метода:

```
int rowCount() const
```

- `columnCount()` — возвращает количество столбцов сетки. Прототип метода:

```
int columnCount() const
```

5.4. Выравнивание компонентов формы

Класс `QFormLayout` позволяет выравнивать компоненты формы. Контейнер по умолчанию состоит из двух столбцов. Первый столбец предназначен для вывода надписи, а второй столбец — для вывода компонента, например текстового поля. При этом надпись связывается с компонентом, что позволяет назначать клавиши быстрого доступа, указав символ `&` перед буквой внутри текста надписи. После нажатия комбинации клавиш быстрого доступа (комбинация <Alt> + буква) в фокусе окажется компонент, расположенный справа от надписи. Иерархия наследования выглядит так:

```
(QObject, QLayoutItem) — QLayout — QFormLayout
```

Создать экземпляр класса `QFormLayout` позволяет следующий конструктор:

```
#include <QFormLayout>
QFormLayout(QWidget *parent = nullptr)
```

В необязательном параметре можно передать указатель на родительский компонент. Если параметр не указан, то необходимо передать указатель на контейнер в метод `setLayout()` родительского компонента. Типичный пример использования класса `QFormLayout` выглядит так:

```
QWidget window;
window.setWindowTitle("QFormLayout");
window.resize(300, 150);
QLineEdit *lineEdit = new QLineEdit();
QTextEdit *textEdit = new QTextEdit();
QPushButton *btn1 = new QPushButton("О&тправить");
QPushButton *btn2 = new QPushButton("О&чистить");
QHBoxLayout *hbox = new QHBoxLayout();
hbox->addWidget(btn1);
hbox->addWidget(btn2);
QFormLayout *form = new QFormLayout();
form->addRow("&Название:", lineEdit);
form->addRow("&Описание:", textEdit);
form->addRow(hbox);
window.setLayout(form);
window.show();
```

Результат выполнения этого кода показан на рис. 5.1.

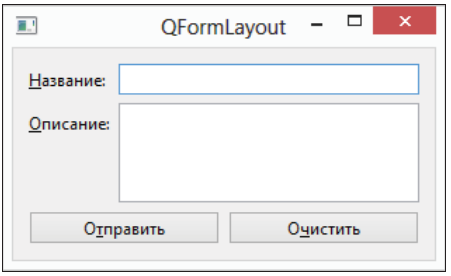

Рис. 5.1. Пример использования класса `QFormLayout`

Класс `QFormLayout` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `addRow()` — добавляет строку в конец контейнера. Прототипы метода:

```
void addRow(const QString &labelText, QWidget *field)
void addRow(const QString &labelText, QLayout *field)
void addRow(QWidget *label, QWidget *field)
void addRow(QWidget *label, QLayout *field)
void addRow(QWidget *widget)
void addRow(QLayout *layout)
```

  В параметре `labelText` можно указать текст, внутри которого перед какой-либо буквой указан символ `&`. В этом случае надпись связывается с компонентом, указанном во втором параметре. После нажатия комбинации клавиш быстрого доступа (комбинация <Alt> + буква) этот компонент окажется в фокусе ввода. Если в первом параметре указан экземпляр класса `QLabel`, то связь с компонентом необходимо устанавливать вручную, передав указатель на компонент в метод `setBuddy()`. Если указан только один параметр, то компонент (или контейнер) займет сразу два столбца;

- `insertRow()` — добавляет строку в указанную позицию контейнера. Если указано отрицательное значение в первом параметре, то компонент добавляется в конец контейнера. Прототипы метода:

```
void insertRow(int row, const QString &labelText, QWidget *field)
void insertRow(int row, const QString &labelText, QLayout *field)
void insertRow(int row, QWidget *label, QWidget *field)
void insertRow(int row, QWidget *label, QLayout *field)
void insertRow(int row, QWidget *widget)
void insertRow(int row, QLayout *layout)
```

- `setFormAlignment()` — задает режим выравнивания формы. Допустимые значения мы рассматривали в *разд. 5.2*. Прототип метода:

```
void setFormAlignment(Qt::Alignment alignment)
```

  Пример:

```
form->setFormAlignment(Qt::AlignLeft | Qt::AlignTop);
```

- `setLabelAlignment()` — задает режим выравнивания надписи. Допустимые значения мы рассматривали в *разд. 5.2*. Прототип метода:

  ```
  void setLabelAlignment(Qt::Alignment alignment)
  ```

  Пример выравнивания по правому краю:

  ```
  form->setLabelAlignment(Qt::AlignRight);
  ```

- `setRowWrapPolicy()` — задает местоположение надписи. Прототип метода:

  ```
  void setRowWrapPolicy(QFormLayout::RowWrapPolicy policy)
  ```

  В качестве параметра указываются следующие константы:

  - `QFormLayout::DontWrapRows` — надписи расположены слева от компонентов;
  - `QFormLayout::WrapLongRows` — длинные надписи могут находиться выше компонентов, а короткие надписи — слева от компонентов;
  - `QFormLayout::WrapAllRows` — надписи расположены выше компонентов;

- `setFieldGrowthPolicy()` — задает режим управления размерами компонентов. Прототип метода:

  ```
  void setFieldGrowthPolicy(QFormLayout::FieldGrowthPolicy policy)
  ```

  В качестве параметра указываются следующие константы:

  - `QFormLayout::FieldsStayAtSizeHint` — размеры компонентов будут соответствовать рекомендуемым (возвращаемым методом `sizeHint()`);
  - `QFormLayout::ExpandingFieldsGrow` — компоненты, для которых установлена политика изменения размеров `QSizePolicy::Expanding` или `QSizePolicy::MinimumExpanding`, будут занимать всю доступную ширину. Размеры остальных компонентов будут соответствовать рекомендуемым;
  - `QFormLayout::AllNonFixedFieldsGrow` — все компоненты (если это возможно) будут занимать всю доступную ширину;

- `setSpacing()` — задает расстояние между компонентами по горизонтали и вертикали. Прототип метода:

  ```
  void setSpacing(int)
  ```

- `setHorizontalSpacing()` — задает расстояние между компонентами по горизонтали. Прототип метода:

  ```
  void setHorizontalSpacing(int spacing)
  ```

- `setVerticalSpacing()` — задает расстояние между компонентами по вертикали. Прототип метода:

  ```
  void setVerticalSpacing(int spacing)
  ```

5.5. Классы *QStackedLayout* и *QStackedWidget*

Класс `QStackedLayout` реализует стек компонентов. В один момент времени показывается только один компонент. Иерархия наследования выглядит так:

```
(QObject, QLayoutItem) — QLayout — QStackedLayout
```

Создать экземпляр класса `QStackedLayout` позволяют следующие конструкторы:

```
#include <QStackedLayout>
QStackedLayout()
QStackedLayout(QWidget *parent)
QStackedLayout(QLayout *parentLayout)
```

В необязательном параметре можно передать указатель на родительский компонент или контейнер. Если параметр не указан, то необходимо передать указатель на контейнер в метод `setLayout()` родительского компонента.

Класс `QStackedLayout` содержит следующие методы:

- `setStackingMode()` — задает режим отображения компонентов. Прототип метода:

  ```
  void setStackingMode(QStackedLayout::StackingMode stackingMode)
  ```

  В параметре могут быть указаны следующие константы:

  - `QStackedLayout::StackOne` — только один компонент видим (значение по умолчанию);
  - `QStackedLayout::StackAll` — видны все компоненты;

- `stackingMode()` — возвращает режим отображения компонентов. Прототип метода:

  ```
  QStackedLayout::StackingMode stackingMode() const
  ```

- `addWidget()` — добавляет компонент в конец контейнера. Метод возвращает индекс добавленного компонента. Прототип метода:

  ```
  int addWidget(QWidget *widget)
  ```

- `insertWidget()` — добавляет компонент в указанную позицию контейнера. Метод возвращает индекс добавленного компонента. Прототип метода:

  ```
  int insertWidget(int index, QWidget *widget)
  ```

- `removeWidget()` — удаляет компонент из контейнера. Прототип метода:

  ```
  void removeWidget(QWidget *widget)
  ```

- `count()` — возвращает количество компонентов внутри контейнера. Прототип метода:

  ```
  int count() const
  ```

- `currentIndex()` — возвращает индекс видимого компонента. Прототип метода:

  ```
  int currentIndex() const
  ```

- `currentWidget()` — возвращает указатель на видимый компонент. Прототип метода:

```
QWidget *currentWidget() const
```

- `widget()` — возвращает указатель на компонент, который расположен по указанному индексу, или нулевой указатель. Прототип метода:

```
QWidget *widget(int index) const
```

- `setCurrentIndex()` — делает видимым компонент с указанным в параметре индексом. Метод является слотом. Прототип метода:

```
void setCurrentIndex(int index)
```

- `setCurrentWidget()` — делает видимым компонент, указатель на который указан в параметре. Метод является слотом. Прототип метода:

```
void setCurrentWidget(QWidget *widget)
```

Класс `QStackedLayout` содержит следующие сигналы:

- `currentChanged(int)` — генерируется при изменении видимого компонента. Через параметр внутри обработчика доступен индекс нового компонента;
- `widgetRemoved(int)` — генерируется при удалении компонента из контейнера. Через параметр внутри обработчика доступен индекс компонента.

Класс `QStackedWidget` также реализует стек компонентов, но создает новый компонент, а не контейнер. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QStackedWidget
```

Создать экземпляр класса `QStackedWidget` позволяет следующий конструктор:

```
#include <QStackedWidget>
QStackedWidget(QWidget *parent = nullptr)
```

Класс `QStackedWidget` содержит методы `addWidget()`, `insertWidget()`, `removeWidget()`, `count()`, `currentIndex()`, `currentWidget()`, `widget()`, `setCurrentIndex()` и `setCurrentWidget()`, которые выполняют аналогичные действия, что и одноименные методы в классе `QStackedLayout`. Кроме того, класс `QStackedWidget` наследует все методы из базовых классов и содержит метод `indexOf()`, который возвращает индекс компонента, указатель на который указан в параметре. Прототип метода:

```
int indexOf(const QWidget *widget) const
```

Чтобы отследить изменения внутри компонента, следует назначить обработчики сигналов `currentChanged(int)` и `widgetRemoved(int)`.

5.6. Класс *QSizePolicy*

Если в вертикальный контейнер большой высоты добавить надпись и кнопку, то под надпись будет выделено максимальное пространство, а кнопка займет пространство, достаточное для рекомендуемых размеров, которые возвращает метод

`sizeHint()`. Управление размерами компонентов внутри контейнера определяется правилами, установленными с помощью класса `QSizePolicy`. Установить правила для компонента можно с помощью метода `setSizePolicy()` из класса `QWidget`, а получить значение — с помощью метода `sizePolicy()`. Прототипы методов:

```
void setSizePolicy(QSizePolicy::Policy horizontal,
                   QSizePolicy::Policy vertical)
void setSizePolicy(QSizePolicy)
QSizePolicy sizePolicy() const
```

Создать экземпляр класса `QSizePolicy` позволяют следующие конструкторы:

```
#include <QSizePolicy>
QSizePolicy()
QSizePolicy(QSizePolicy::Policy horizontal, QSizePolicy::Policy vertical,
            QSizePolicy::ControlType type = DefaultType)
```

Если параметры не заданы, то размер компонента должен точно соответствовать размерам, возвращаемым методом `sizeHint()`. В первом и втором параметрах указываются следующие константы:

- `QSizePolicy::Fixed` — размер компонента должен точно соответствовать размерам, возвращаемым методом `sizeHint()`;
- `QSizePolicy::Minimum` — размер, возвращаемый методом `sizeHint()`, является минимальным для компонента. Размер может быть увеличен компоновщиком;
- `QSizePolicy::Maximum` — размер, возвращаемый методом `sizeHint()`, является максимальным для компонента. Размер может быть уменьшен компоновщиком;
- `QSizePolicy::Preferred` — размер, возвращаемый методом `sizeHint()`, является предпочтительным, но может быть как увеличен, так и уменьшен;
- `QSizePolicy::Expanding` — размер, возвращаемый методом `sizeHint()`, может быть как увеличен, так и уменьшен. Компоновщик должен предоставить компоненту столько пространства, сколько возможно;
- `QSizePolicy::MinimumExpanding` — размер, возвращаемый методом `sizeHint()`, является минимальным для компонента. Компоновщик должен предоставить компоненту столько пространства, сколько возможно;
- `QSizePolicy::Ignored` — размер, возвращаемый методом `sizeHint()`, игнорируется. Компонент получит столько пространства, сколько возможно.

Изменить значения уже после создания экземпляра класса `QSizePolicy` позволяют методы `setHorizontalPolicy()` и `setVerticalPolicy()`. Прототипы методов:

```
void setHorizontalPolicy(QSizePolicy::Policy policy)
void setVerticalPolicy(QSizePolicy::Policy policy)
```

С помощью методов `setHorizontalStretch()` и `setVerticalStretch()` можно указать фактор растяжения. Чем больше указанное значение относительно значения, заданного в других компонентах, тем больше места будет выделяться под компонент. Этот параметр можно сравнить с жесткостью пружины. Прототипы методов:

```
void setHorizontalStretch(int stretchFactor)
void setVerticalStretch(int stretchFactor)
```

Можно указать, что минимальная высота компонента зависит от его ширины. Для этого необходимо передать значение `true` в метод `setHeightForWidth()`. Кроме того, следует переопределить метод `heightForWidth()` в классе компонента. Метод должен возвращать высоту компонента в соответствии с указанной в параметре шириной. Прототипы методов:

```
void setHeightForWidth(bool dependent)
virtual int heightForWidth(int w) const
```

5.7. Объединение компонентов в группу

Состояние некоторых компонентов может зависеть от состояния других компонентов, например из нескольких переключателей можно выбрать только один. В этом случае компоненты объединяют в группу. Группа компонентов отображается внутри рамки, на границе которой выводится текст подсказки. Реализовать группу позволяет класс `QGroupBox`. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QGroupBox
```

Создать экземпляр класса `QGroupBox` позволяют следующие конструкторы:

```
#include <QGroupBox>
QGroupBox(QWidget *parent = nullptr)
QGroupBox(const QString &title, QWidget *parent = nullptr)
```

В необязательном параметре `parent` можно передать указатель на родительский компонент. Параметр `title` задает текст подсказки, которая отобразится на верхней границе рамки. Внутри текста подсказки символ `&`, указанный перед буквой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ `&`, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы первый компонент внутри группы окажется в фокусе ввода.

После создания экземпляра класса `QGroupBox` следует добавить компоненты в какой-либо контейнер, а затем передать указатель на контейнер в метод `setLayout()`. Типичный пример использования класса `QGroupBox` выглядит так:

```
QWidget window;
window.setWindowTitle("Класс QGroupBox");
window.resize(350, 80);
QRadioButton *radio1 = new QRadioButton("&Да");
QRadioButton *radio2 = new QRadioButton("&Нет");
QVBoxLayout *vbox = new QVBoxLayout();
QGroupBox *box = new QGroupBox("&Вы знаете язык C++?");
QHBoxLayout *hbox = new QHBoxLayout();
hbox->addWidget(radio1);
hbox->addWidget(radio2);
box->setLayout(hbox);
```

```
vbox->addWidget(box);
window.setLayout(vbox);
radio1->setChecked(true); // Выбираем первый переключатель
window.show();
```

Результат выполнения этого кода показан на рис. 5.2.

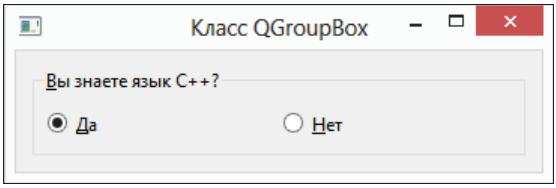

Рис. 5.2. Пример использования класса `QGroupBox`

Класс `QGroupBox` содержит следующие методы:

- `setTitle()` — задает текст подсказки. Прототип метода:

  ```
  void setTitle(const QString &title)
  ```

  Получение значения:

  ```
  QString title() const
  ```

- `setAlignment()` — задает горизонтальное местоположение текста подсказки. В параметре указываются следующие константы `Qt::AlignLeft`, `Qt::AlignHCenter` или `Qt::AlignRight`. Прототип метода:

  ```
  void setAlignment(int alignment)
  ```

  Пример:

  ```
  box->setAlignment(Qt::AlignRight);
  ```

  Получение значения:

  ```
  Qt::Alignment alignment() const
  ```

- `setCheckable()` — если в параметре указать значение `true`, то перед текстом подсказки будет отображен флажок. Если флажок установлен, то группа будет активной, а если флажок снят, то все компоненты внутри группы станут неактивными. По умолчанию флажок не отображается. Прототип метода:

  ```
  void setCheckable(bool checkable)
  ```

- `isCheckable()` — возвращает значение `true`, если флажок выводится перед надписью, и `false` — в противном случае. Прототип метода:

  ```
  bool isCheckable() const
  ```

- `setChecked()` — если в параметре указать значение `true`, то флажок, отображаемый перед текстом подсказки, будет установлен. Значение `false` сбрасывает флажок. Метод является слотом. Прототип метода:

  ```
  void setChecked(bool checked)
  ```

- `isChecked()` — возвращает значение `true`, если флажок, отображаемый перед текстом подсказки, установлен, и `false` — в противном случае. Прототип метода:

  ```
  bool isChecked() const
  ```

- `setFlat()` — если в параметре указано значение `true`, то отображается только верхняя граница рамки, а если `false`, то все границы рамки. Прототип метода:

  ```
  void setFlat(bool flat)
  ```

- `isFlat()` — возвращает значение `true`, если отображается только верхняя граница рамки, и `false` — если все границы рамки. Прототип метода:

  ```
  bool isFlat() const
  ```

Класс `QGroupBox` содержит следующие сигналы:

- `clicked(bool checked=false)` — генерируется при щелчке мышью на флажке, выводимом перед текстом подсказки. Если состояние флажка изменяется с помощью метода `setChecked()`, то сигнал не генерируется. Через параметр внутри обработчика доступно значение `true`, если флажок установлен, и `false` — если сброшен;
- `toggled(bool)` — генерируется при изменении статуса флажка, выводимого перед текстом подсказки. Через параметр внутри обработчика доступно значение `true`, если флажок установлен, и `false` — если сброшен.

5.8. Панель с рамкой

Класс `QFrame` расширяет возможности класса `QWidget` за счет добавления рамки различного стиля вокруг компонента. Этот класс наследуют, в свою очередь, некоторые компоненты, например: надписи, многострочные текстовые поля и др. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame
```

Конструктор класса `QFrame` имеет следующий прототип:

```
#include <QFrame>
QFrame(QWidget *parent = nullptr, Qt::WindowFlags f = Qt::WindowFlags())
```

В параметре `parent` передается указатель на родительский компонент. Если параметр не указан или имеет значение `nullptr`, то компонент будет обладать своим собственным окном. Если в параметре `f` указан тип окна, то компонент, имея родителя, будет обладать своим собственным окном, но будет привязан к родителю. Это позволяет, например, создать модальное окно, которое будет блокировать только окно родителя, а не все окна приложения. Какие именно значения можно указать в параметре `f`, мы рассматривали в *разд. 3.2*.

Класс `QFrame` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setFrameShape()` — задает форму рамки. Прототип метода:

  ```
  void setFrameShape(QFrame::Shape)
  ```

  Могут быть указаны следующие константы:

  - `QFrame::NoFrame` — нет рамки;
  - `QFrame::Box` — прямоугольная рамка;
  - `QFrame::Panel` — панель, которая может быть выпуклой или вогнутой;
  - `QFrame::WinPanel` — панель со стилем, принятым в Windows. Ширина границы 2 пиксела. Панель может быть выпуклой или вогнутой;
  - `QFrame::HLine` — горизонтальная линия. Используется как разделитель;
  - `QFrame::VLine` — вертикальная линия без содержимого;
  - `QFrame::StyledPanel` — панель, внешний вид которой зависит от текущего стиля. Панель может быть выпуклой или вогнутой.

  Получение значения:

  ```
  QFrame::Shape frameShape() const
  ```

- `setFrameShadow()` — задает стиль тени. Прототип метода:

  ```
  void setFrameShadow(QFrame::Shadow)
  ```

  Могут быть указаны следующие константы:

  - `QFrame::Plain` — нет эффектов;
  - `QFrame::Raised` — панель отображается выпуклой;
  - `QFrame::Sunken` — панель отображается вогнутой.

  Получение значения:

  ```
  QFrame::Shadow frameShadow() const
  ```

- `setFrameStyle()` — задает форму рамки и стиль тени одновременно. Прототип метода:

  ```
  void setFrameStyle(int style)
  ```

  В качестве значения указывается комбинация констант через оператор `|`. Пример:

  ```
  frame2->setFrameStyle(QFrame::Panel | QFrame::Raised);
  ```

  Получение значения:

  ```
  int frameStyle() const
  ```

- `setLineWidth()` — задает ширину линии рамки. Прототип метода:

  ```
  void setLineWidth(int)
  ```

  Получение значения:

  ```
  int lineWidth() const
  ```

- `setMidLineWidth()` — задает ширину средней линии рамки. Средняя линия используется для создания эффекта выпуклости и вогнутости и доступна только для форм рамки `Box`, `HLine` и `VLine`. Прототип метода:

  ```
  void setMidLineWidth(int)
  ```

  Получение значения:

  ```
  int midLineWidth() const
  ```

5.9. Панель с вкладками

Для создания панели с вкладками предназначен класс `QTabWidget`. Панель состоит из области заголовка с ярлыками и набора вкладок с различными компонентами. В один момент времени показывается содержимое только одной вкладки. Щелчок мышью на ярлыке в области заголовка приводит к отображению содержимого соответствующий вкладки. Иерархия наследования для класса `QTabWidget` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QTabWidget
```

Конструктор класса `QTabWidget` имеет следующий прототип:

```
#include <QTabWidget>
QTabWidget(QWidget *parent = nullptr)
```

В параметре `parent` передается указатель на родительский компонент. Если параметр не указан, то компонент будет обладать своим собственным окном. Типичный пример использования класса `QTabWidget` выглядит так:

```
QWidget window;
window.setWindowTitle("Класс QTabWidget");
window.resize(350, 120);
QTabWidget *tab = new QTabWidget();
tab->addTab(new QLabel("Содержимое вкладки 1"), "Вкладка &1");
tab->addTab(new QLabel("Содержимое вкладки 2"), "Вкладка &2");
tab->addTab(new QLabel("Содержимое вкладки 3"), "Вкладка &3");
tab->setCurrentIndex(0);
QVBoxLayout *vbox = new QVBoxLayout();
vbox->addWidget(tab);
window.setLayout(vbox);
window.show();
```

Результат выполнения этого кода показан на рис. 5.3.

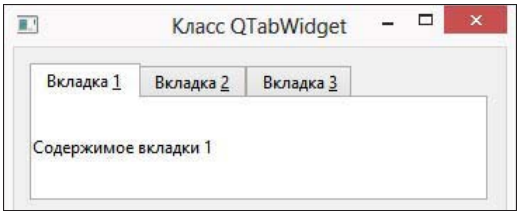

Рис. 5.3. Пример использования класса `QTabWidget`

Класс `QTabWidget` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `addTab()` — добавляет вкладку в конец контейнера. Метод возвращает индекс добавленной вкладки. Прототипы метода:

```
int addTab(QWidget *page, const QString &label)
int addTab(QWidget *page, const QIcon &icon, const QString &label)
```

  В параметре `page` передается указатель на компонент, который будет отображаться на вкладке. Чаще всего этот компонент является лишь родителем для других компонентов. Параметр `label` задает текст, который будет отображаться на ярлыке в области заголовка. Внутри текста заголовка символ `&`, указанный перед буквой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ `&`, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы соответствующая вкладка будет отображена. Параметр `icon` позволяет указать значок (экземпляр класса `QIcon`), который отобразится перед текстом в области заголовка. Пример указания стандартного значка:

```
QIcon icon1 = window.style()->standardIcon(QStyle::SP_DriveHDIcon);
tab->addTab(new QLabel("Содержимое вкладки 1"), icon1, "Вкладка &1");
```

  Пример загрузки значка из файла:

```
QIcon icon3("C:\\cpp\\projectsQt\\Test\\test.ico");
tab->addTab(new QLabel("Содержимое вкладки 3"), icon3, "Вкладка &3");
```

- `insertTab()` — добавляет вкладку в указанную позицию. Метод возвращает индекс добавленной вкладки. Прототипы метода:

```
int insertTab(int index, QWidget *page, const QString &label)
int insertTab(int index, QWidget *page, const QIcon &icon,
              const QString &label)
```

  Пример добавления вкладки в конец контейнера после нажатия кнопки:

```
void Widget::on_btn1_clicked()
{
   tab->setUpdatesEnabled(false); // Для предотвращения мерцания
   int ind = tab->insertTab(-1,
             new QLabel("Содержимое вкладки 3"), "Вкладка &3");
   tab->setCurrentIndex(ind);
   tab->setUpdatesEnabled(true);  // Устанавливаем обратно
   btn1->setEnabled(false);
}
```

- `removeTab()` — удаляет вкладку с указанным индексом. При этом компонент, который отображался на вкладке, не удаляется. Прототип метода:

```
void removeTab(int index)
```

- `clear()` — удаляет все вкладки. При этом компоненты, которые отображались на вкладках, не удаляются. Прототип метода:

  ```
  void clear()
  ```

- `setTabText()` — задает текст заголовка для вкладки с указанным индексом. Прототип метода:

  ```
  void setTabText(int index, const QString &label)
  ```

- `tabText()` — возвращает текст заголовка вкладки с указанным индексом. Прототип метода:

  ```
  QString tabText(int index) const
  ```

- `setElideMode()` — задает режим обрезки текста в названии вкладки, если он не помещается в отведенную область. В месте пропуска выводится многоточие. Прототип метода:

  ```
  void setElideMode(Qt::TextElideMode mode)
  ```

  Могут быть указаны следующие константы:

  - `Qt::ElideLeft` — текст обрезается слева;
  - `Qt::ElideRight` — текст обрезается справа;
  - `Qt::ElideMiddle` — текст обрезается посередине;
  - `Qt::ElideNone` — текст не обрезается.

  Получение значения:

  ```
  Qt::TextElideMode elideMode() const
  ```

- `setTabIcon()` — устанавливает значок перед текстом в заголовке вкладки с указанным индексом. Прототип метода:

  ```
  void setTabIcon(int index, const QIcon &icon)
  ```

  Получение значения:

  ```
  QIcon tabIcon(int index) const
  ```

- `setTabPosition()` — задает позицию области заголовка. Прототип метода:

  ```
  void setTabPosition(QTabWidget::TabPosition position)
  ```

  Могут быть указаны следующие константы:

  - `QTabWidget::North` — сверху;
  - `QTabWidget::South` — снизу;
  - `QTabWidget::West` — слева;
  - `QTabWidget::East` — справа.

  Пример указания значения:

  ```
  tab->setTabPosition(QTabWidget::East);
  ```

Получение значения:

```
QTabWidget::TabPosition tabPosition() const
```

- `setTabShape()` — задает форму углов ярлыка вкладки в области заголовка. Прототип метода:

  ```
  void setTabShape(QTabWidget::TabShape s)
  ```

  Могут быть указаны следующие константы:

  - `QTabWidget::Rounded` — скругленные углы (значение по умолчанию);
  - `QTabWidget::Triangular` — треугольная форма.

  Получение значения:

  ```
  QTabWidget::TabShape tabShape() const
  ```

- `setTabsClosable()` — если в качестве параметра указано значение `true`, то после текста заголовка будет отображена кнопка закрытия вкладки. При нажатии этой кнопки генерируется сигнал `tabCloseRequested(int)`. Прототип метода:

  ```
  void setTabsClosable(bool closeable)
  ```

  Получение значения:

  ```
  bool tabsClosable() const
  ```

- `setMovable()` — если в качестве параметра указано значение `true`, то ярлыки вкладок можно перемещать с помощью мыши. Прототип метода:

  ```
  void setMovable(bool movable)
  ```

  Получение значения:

  ```
  bool isMovable() const
  ```

- `setDocumentMode()` — если в качестве параметра указано значение `true`, то область компонента не будет отображаться как панель. Прототип метода:

  ```
  void setDocumentMode(bool set)
  ```

  Получение значения:

  ```
  bool documentMode() const
  ```

- `setUsesScrollButtons()` — если в качестве параметра указано значение `true`, то, когда все ярлыки вкладок не помещаются в область заголовка, появляются две кнопки, с помощью которых можно прокручивать область заголовка, тем самым отображая только часть ярлыков. Значение `false` запрещает сокрытие ярлыков. Прототип метода:

  ```
  void setUsesScrollButtons(bool useButtons)
  ```

  Получение значения:

  ```
  bool usesScrollButtons() const
  ```

- `setTabToolTip()` — задает текст всплывающей подсказки для ярлыка вкладки с указанным индексом. Прототип метода:

  ```
  void setTabToolTip(int index, const QString &tip)
  ```

Получение значения:

```
QString tabToolTip(int index) const
```

- `setTabWhatsThis()` — задает текст справки для ярлыка вкладки с указанным индексом. Прототип метода:

```
void setTabWhatsThis(int index, const QString &text)
```

Получение значения:

```
QString tabWhatsThis(int index) const
```

- `setTabEnabled()` — если во втором параметре указано значение `false`, то вкладка с указанным в первом параметре индексом станет недоступной. Значение `true` делает вкладку доступной. Прототип метода:

```
void setTabEnabled(int index, bool enable)
```

- `isTabEnabled()` — возвращает значение `true`, если вкладка с указанным индексом доступна, и `false` — в противном случае. Прототип метода:

```
bool isTabEnabled(int index) const
```

- `setTabVisible()` — если во втором параметре указано значение `false`, то вкладка с указанным в первом параметре индексом будет скрыта. Значение `true` делает вкладку видимой. Прототип метода:

```
void setTabVisible(int index, bool visible)
```

Получение значения:

```
bool isTabVisible(int index) const
```

- `setTabBarAutoHide()` — если в параметре указано значение `true`, то область ярлыков вкладок будет скрываться, если панель содержит меньше двух вкладок. Прототип метода:

```
void setTabBarAutoHide(bool enabled)
```

Получение значения:

```
bool tabBarAutoHide() const
```

- `count()` — возвращает количество вкладок. Прототип метода:

```
int count() const
```

- `currentIndex()` — возвращает индекс видимой вкладки. Прототип метода:

```
int currentIndex() const
```

- `currentWidget()` — возвращает указатель на компонент, расположенный на видимой вкладке. Прототип метода:

```
QWidget *currentWidget() const
```

- `widget()` — возвращает указатель на компонент, который расположен по указанному индексу, или нулевой указатель. Прототип метода:

```
QWidget *widget(int index) const
```

- `indexOf()` — возвращает индекс вкладки, на которой расположен компонент. Если компонент не найден, возвращается значение `-1`. Прототип метода:

```
int indexOf(const QWidget *w) const
```

- `setCurrentIndex()` — делает видимой вкладку с указанным в параметре индексом. Метод является слотом. Прототип метода:

```
void setCurrentIndex(int index)
```

- `setCurrentWidget()` — делает видимым компонент, указатель на который указан в параметре. Метод является слотом. Прототип метода:

```
void setCurrentWidget(QWidget *widget)
```

Класс `QTabWidget` содержит следующие сигналы:

- `currentChanged(int)` — генерируется при изменении вкладки. Через параметр внутри обработчика доступен индекс новой вкладки;
- `tabCloseRequested(int)` — генерируется при нажатии кнопки закрытия вкладки. Через параметр внутри обработчика доступен индекс вкладки;
- `tabBarClicked(int)` — генерируется при щелчке мышью в области заголовков вкладок. Через параметр внутри обработчика доступен индекс вкладки или значение `-1`;
- `tabCloseRequested(int)` — генерируется при двойном щелчке мышью в области заголовков вкладок. Через параметр внутри обработчика доступен индекс вкладки или значение `-1`.

5.10. Компонент «аккордеон»

Класс `QToolBox` позволяет создать компонент с несколькими вкладками. Изначально отображается содержимое только одной вкладки, а у остальных доступны только заголовки. После щелчка мышью на заголовке вкладки она открывается, а остальные сворачиваются. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QToolBox
```

Конструктор класса `QToolBox` имеет следующий формат:

```
#include <QToolBox>
QToolBox(QWidget *parent = nullptr,
         Qt::WindowFlags f = Qt::WindowFlags())
```

В параметре `parent` передается указатель на родительский компонент. Если параметр не указан или имеет значение `nullptr`, то компонент будет обладать своим собственным окном. В параметре `f` может быть указан тип окна. Пример использования класса `QToolBox`:

```
QWidget window;
window.setWindowTitle("Класс QToolBox");
window.resize(350, 150);
QToolBox *toolBox = new QToolBox();
```

```
toolBox->addItem(new QLabel("Содержимое вкладки 1"), "Вкладка &1");
toolBox->addItem(new QLabel("Содержимое вкладки 2"), "Вкладка &2");
toolBox->addItem(new QLabel("Содержимое вкладки 3"), "Вкладка &3");
toolBox->setCurrentIndex(0);
QVBoxLayout *vbox = new QVBoxLayout();
vbox->addWidget(toolBox);
window.setLayout(vbox);
window.show();
```

Класс `QToolBox` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `addItem()` — добавляет вкладку в конец контейнера. Метод возвращает индекс добавленной вкладки. Прототипы метода:

  ```
  int addItem(QWidget *widget, const QString &text)
  int addItem(QWidget *widget, const QIcon &iconSet, const QString &text)
  ```

  В параметре `widget` передается указатель на компонент, который будет отображаться на вкладке. Чаще всего этот компонент является лишь родителем для других компонентов. Параметр `text` задает текст, который будет отображаться на ярлыке в области заголовка. Внутри текста заголовка символ `&`, указанный перед буквой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ `&`, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы соответствующая вкладка будет отображена. Параметр `iconSet` позволяет указать значок (экземпляр класса `QIcon`), который отобразится перед текстом в области заголовка;

- `insertItem()` — добавляет вкладку в указанную позицию. Метод возвращает индекс добавленной вкладки. Прототипы метода:

  ```
  int insertItem(int index, QWidget *widget, const QString &text)
  int insertItem(int index, QWidget *widget, const QIcon &icon,
                 const QString &text)
  ```

- `removeItem()` — удаляет вкладку с указанным индексом. При этом компонент, который отображался на вкладке, не удаляется. Прототип метода:

  ```
  void removeItem(int index)
  ```

- `setItemText()` — задает текст заголовка для вкладки с указанным индексом. Прототип метода:

  ```
  void setItemText(int index, const QString &text)
  ```

- `itemText()` — возвращает текст заголовка вкладки с указанным индексом. Прототип метода:

  ```
  QString itemText(int index) const
  ```

- `setItemIcon()` — устанавливает иконку перед текстом в заголовке вкладки с указанным индексом. Прототип метода:

  ```
  void setItemIcon(int index, const QIcon &icon)
  ```

Получение значения:

```
QIcon itemIcon(int index) const
```

- `setItemToolTip()` — задает текст вплывающей подсказки для ярлыка вкладки с указанным индексом. Прототип метода:

  ```
  void setItemToolTip(int index, const QString &toolTip)
  ```

  Получение значения:

  ```
  QString itemToolTip(int index) const
  ```

- `setItemEnabled()` — если во втором параметре указано значение `false`, то вкладка с указанным в первом параметре индексом станет недоступной. Значение `true` делает вкладку доступной. Прототип метода:

  ```
  void setItemEnabled(int index, bool enabled)
  ```

- `isItemEnabled()` — возвращает значение `true`, если вкладка с указанным индексом доступна, и `false` — в противном случае. Прототип метода:

  ```
  bool isItemEnabled(int index) const
  ```

- `count()` — возвращает количество вкладок. Прототип метода:

  ```
  int count() const
  ```

- `currentIndex()` — возвращает индекс видимой вкладки. Прототип метода:

  ```
  int currentIndex() const
  ```

- `currentWidget()` — возвращает указатель на компонент, который расположен на видимой вкладке. Прототип метода:

  ```
  QWidget *currentWidget() const
  ```

- `widget()` — возвращает указатель на компонент, который расположен по указанному индексу, или нулевой указатель. Прототип метода:

  ```
  QWidget *widget(int index) const
  ```

- `indexOf()` — возвращает индекс вкладки, на которой расположен компонент. Если компонент не найден, возвращается значение `-1`. Прототип метода:

  ```
  int indexOf(const QWidget *widget) const
  ```

- `setCurrentIndex()` — делает видимой вкладку с указанным в параметре индексом. Метод является слотом. Прототип метода:

  ```
  void setCurrentIndex(int index)
  ```

- `setCurrentWidget()` — делает видимым компонент, указатель на который передан в параметре. Метод является слотом. Прототип метода:

  ```
  void setCurrentWidget(QWidget *widget)
  ```

При изменении вкладки генерируется сигнал `currentChanged(int)`. Через параметр внутри обработчика доступен индекс вкладки.

5.11. Панели с изменяемым размером

Класс `QSplitter` позволяет изменять размеры добавленных компонентов с помощью мыши. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QSplitter
```

Конструктор класса `QSplitter` имеет два формата:

```
#include <QSplitter>
QSplitter(QWidget *parent = nullptr)
QSplitter(Qt::Orientation orientation, QWidget *parent = nullptr)
```

В параметре `parent` передается указатель на родительский компонент. Если параметр не указан или имеет значение `nullptr`, то компонент будет обладать своим собственным окном. Параметр `orientation` задает ориентацию размещения компонентов. Могут быть заданы константы `Qt::Horizontal` (по горизонтали) или `Qt::Vertical` (по вертикали). Если параметр не указан, то компоненты размещаются по горизонтали. Пример использования класса `QSplitter`:

```
QWidget window;
window.setWindowTitle("Класс QSplitter");
window.resize(400, 250);
QSplitter *splitter = new QSplitter(Qt::Vertical);
QSplitter *splitter2 = new QSplitter(Qt::Horizontal);
QLabel *label1 = new QLabel("Содержимое 1");
QLabel *label2 = new QLabel("Содержимое 2");
QLabel *label3 = new QLabel("Содержимое 3");
label1->setFrameStyle(QFrame::Box | QFrame::Plain);
label2->setFrameStyle(QFrame::Box | QFrame::Plain);
label3->setFrameStyle(QFrame::Box | QFrame::Plain);
splitter2->addWidget(label1);
splitter2->addWidget(label2);
splitter->addWidget(splitter2);
splitter->addWidget(label3);
QVBoxLayout *vbox = new QVBoxLayout();
vbox->addWidget(splitter);
window.setLayout(vbox);
window.show();
```

Класс `QSplitter` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `addWidget()` — добавляет компонент в конец контейнера. Прототип метода:

  ```
  void addWidget(QWidget *widget)
  ```

- `insertWidget()` — добавляет компонент в указанную позицию. Если компонент был добавлен ранее, то он будет перемещен в новую позицию. Прототип метода:

  ```
  void insertWidget(int index, QWidget *widget)
  ```

- `setOrientation()` — задает ориентацию размещения компонентов. Могут быть заданы константы `Qt::Horizontal` (по горизонтали) или `Qt::Vertical` (по вертикали). Прототип метода:

  ```
  void setOrientation(Qt::Orientation)
  ```

  Получение значения:

  ```
  Qt::Orientation orientation() const
  ```

- `setHandleWidth()` — задает ширину компонента-разделителя, взявшись за который мышью, можно изменить размер области. Прототип метода:

  ```
  void setHandleWidth(int)
  ```

  Получение значения:

  ```
  int handleWidth() const
  ```

- `saveState()` — возвращает экземпляр класса `QByteArray` с размерами всех областей. Эти данные можно сохранить (например, в файл), а затем восстановить с помощью метода `restoreState()`. Прототипы методов:

  ```
  QByteArray saveState() const
  bool restoreState(const QByteArray &state)
  ```

- `setChildrenCollapsible()` — если в параметре указано значение `false`, то пользователь не сможет уменьшить размеры всех компонентов до нуля. По умолчанию размер может быть нулевым, даже если установлены минимальные размеры компонента. Прототип метода:

  ```
  void setChildrenCollapsible(bool)
  ```

  Получение значения:

  ```
  bool childrenCollapsible() const
  ```

- `setCollapsible()` — значение `false` во втором параметре запрещает уменьшение размеров до нуля для компонента с указанным индексом. Прототип метода:

  ```
  void setCollapsible(int index, bool collapse)
  ```

  Получение значения:

  ```
  bool isCollapsible(int index) const
  ```

- `setOpaqueResize()` — если в качестве параметра указано значение `false`, то размеры компонентов изменятся только после окончания перемещения границы и отпускания кнопки мыши. В процессе перемещения мыши вместе с ней будет перемещаться специальный компонент в виде линии. Прототип метода:

  ```
  void setOpaqueResize(bool opaque = true)
  ```

  Получение значения:

  ```
  bool opaqueResize() const
  ```

- `setStretchFactor()` — задает фактор растяжения для компонента с указанным индексом. Прототип метода:

  ```
  void setStretchFactor(int index, int stretch)
  ```

- `setSizes()` — задает размеры всех компонентов. Для горизонтального контейнера указывается список со значениями ширины каждого компонента, а для вертикального контейнера — список со значениями высоты каждого компонента. Прототип метода:

```
void setSizes(const QList<int> &list)
```

- `sizes()` — возвращает список с размерами (шириной или высотой). Прототип метода:

```
QList<int> sizes() const
```

- `count()` — возвращает количество компонентов. Прототип метода:

```
int count() const
```

- `widget()` — возвращает указатель на компонент, который расположен по указанному индексу. Прототип метода:

```
QWidget *widget(int index) const
```

- `indexOf()` — возвращает индекс области, в которой расположен компонент. Если компонент не найден, возвращается значение `-1`. Прототип метода:

```
int indexOf(QWidget *widget) const
```

При изменении размеров генерируется сигнал `splitterMoved(int,int)`. Через первый параметр внутри обработчика доступна новая позиция, а через второй параметр — индекс перемещаемого разделителя.

5.12. Область с полосами прокрутки

Класс `QScrollArea` реализует область с полосами прокрутки. Если компонент не помещается в размеры области, то автоматически отображаются полосы прокрутки. Изменение положения полос прокрутки с помощью мыши автоматически приводит к прокрутке содержимого области. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                        — QAbstractScrollArea — QScrollArea
```

Конструктор класса `QScrollArea` имеет следующий формат:

```
#include <QScrollArea>
QScrollArea(QWidget *parent = nullptr)
```

Класс `QScrollArea` содержит следующие методы:

- `setWidget()` — добавляет компонент в область прокрутки. Прототип метода:

```
void setWidget(QWidget *widget)
```

- `widget()` — возвращает указатель на компонент, который расположен внутри области. Прототип метода:

```
QWidget *widget() const
```

- `setWidgetResizable()` — если в качестве параметра указано значение `true`, то при изменении размеров области будут изменяться и размеры компонента. Значение `false` запрещает изменение размеров компонента. Прототип метода:

  ```
  void setWidgetResizable(bool resizable)
  ```

  Получение значения:

  ```
  bool widgetResizable() const
  ```

- `setAlignment()` — задает местоположение компонента внутри области, когда размеры области больше размеров компонента. Прототип метода:

  ```
  void setAlignment(Qt::Alignment)
  ```

  Пример:

  ```
  scrollArea.setAlignment(Qt::AlignCenter);
  ```

  Получение значения:

  ```
  Qt::Alignment alignment() const
  ```

- `ensureVisible()` — прокручивает область к точке с координатами `(x, y)` и полями `xmargin` и `ymargin`. Прототип метода:

  ```
  void ensureVisible(int x, int y,
                     int xmargin = 50, int ymargin = 50)
  ```

- `ensureWidgetVisible()` — прокручивает область таким образом, чтобы `childWidget` был видим. Прототип метода:

  ```
  void ensureWidgetVisible(QWidget *childWidget,
                           int xmargin = 50, int ymargin = 50)
  ```

- `takeWidget()` — удаляет компонент из области и возвращает указатель на него. Сам компонент не удаляется. Прототип метода:

  ```
  QWidget *takeWidget()
  ```

Класс `QScrollArea` наследует следующие методы из класса `QAbstractScrollArea` (перечислены только основные методы; полный список смотрите в документации):

- `horizontalScrollBar()` — возвращает указатель на горизонтальную полосу прокрутки (экземпляр класса `QScrollBar`). Прототип метода:

  ```
  QScrollBar *horizontalScrollBar() const
  ```

- `verticalScrollBar()` — возвращает указатель на вертикальную полосу прокрутки (экземпляр класса `QScrollBar`). Прототип метода:

  ```
  QScrollBar *verticalScrollBar() const
  ```

- `cornerWidget()` — возвращает указатель на компонент, расположенный в правом нижнем углу между двумя полосами прокрутки. Прототип метода:

  ```
  QWidget *cornerWidget() const
  ```

- `viewport()` — возвращает указатель на окно области прокрутки. Прототип метода:

  ```
  QWidget *viewport() const
  ```

- `setHorizontalScrollBarPolicy()` — устанавливает режим отображения горизонтальной полосы прокрутки. Прототип метода:

  ```
  void setHorizontalScrollBarPolicy(Qt::ScrollBarPolicy mode)
  ```

  Получение значения:

  ```
  Qt::ScrollBarPolicy horizontalScrollBarPolicy() const
  ```

- `setVerticalScrollBarPolicy()` — устанавливает режим отображения вертикальной полосы прокрутки. Прототип метода:

  ```
  void setVerticalScrollBarPolicy(Qt::ScrollBarPolicy mode)
  ```

  В параметре `mode` могут быть указаны следующие константы:

  - `Qt::ScrollBarAsNeeded` — полоса прокрутки отображается только в том случае, если размеры компонента больше размеров области;
  - `Qt::ScrollBarAlwaysOff` — полоса прокрутки никогда не отображается;
  - `Qt::ScrollBarAlwaysOn` — полоса прокрутки всегда отображается.

  Получение значения:

  ```
  Qt::ScrollBarPolicy verticalScrollBarPolicy() const
  ```

ГЛАВА 6

Основные компоненты

Практически все компоненты графического интерфейса наследуют классы `QObject` и `QWidget`. Следовательно, методы этих классов, которые мы рассматривали в предыдущих главах, доступны всем компонентам. Если компонент не имеет родителя, то он обладает собственным окном и, например, его положение отсчитывается относительно экрана. Если же компонент имеет родителя, то его положение отсчитывается относительно родительского компонента. Это обстоятельство важно учитывать при работе с компонентами. Обращайте внимание на иерархию наследования, которую мы будем показывать для каждого компонента.

6.1. Надпись

Надпись применяется для вывода подсказки пользователю, информирования пользователя о ходе выполнения операции, назначения клавиш быстрого доступа применительно к другому компоненту, а также для вывода изображений и анимации. Кроме того, надписи позволяют отображать текст в формате HTML, отформатированный с помощью CSS, что делает надпись самым настоящим браузером. В библиотеке Qt надпись реализуется с помощью класса `QLabel`. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QLabel
```

Конструктор класса `QLabel` имеет два формата:

```
#include <QLabel>
QLabel(const QString &text, QWidget *parent = nullptr,
       Qt::WindowFlags f = Qt::WindowFlags())
QLabel(QWidget *parent = nullptr, Qt::WindowFlags f = Qt::WindowFlags())
```

В параметре `parent` передается указатель на родительский компонент. Если параметр не указан или имеет значение `nullptr`, то компонент будет обладать своим собственным окном, тип которого можно задать с помощью параметра `f`. Параметр `text` позволяет задать текст, который будет отображен на надписи. Пример:

```
QLabel *label = new QLabel("Текст надписи");
```

Класс QLabel содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setText() — задает текст, который будет отображен на надписи. Можно указать как обычный текст, так и текст в формате HTML, который содержит форматирование с помощью CSS. Метод является слотом. Прототип метода:

  ```
  void setText(const QString &)
  ```

  Пример:

  ```
  label->setText("Текст <b>полужирный</b>");
  ```

  Перевод строки в простом тексте осуществляется с помощью символа \n, а в тексте в формате HTML с помощью тега
. Пример:

  ```
  label->setText("Текст\nна двух строках");
  label->setText("Текст<br>на двух строках");
  ```

  Внутри текста символ &, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ &, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы компонент, указатель на который передан в метод setBuddy(), окажется в фокусе ввода. Чтобы вывести символ &, необходимо его удвоить. Если надпись не связана с другим компонентом, то символ & выводится в составе текста. Пример:

  ```
  QLabel *label = new QLabel("&Пароль");
  QLineEdit *lineEdit = new QLineEdit();
  label->setBuddy(lineEdit);
  ```

  Получение значения:

  ```
  QString text() const
  ```

- setNum() — преобразует целое или вещественное число в строку и отображает ее на надписи. Метод является слотом. Прототипы метода:

  ```
  void setNum(int num)
  void setNum(double num)
  ```

- setWordWrap() — если в параметре указано значение true, то текст может переноситься на новую строку. По умолчанию перенос строк не осуществляется. Прототип метода:

  ```
  void setWordWrap(bool on)
  ```

  Получение значения:

  ```
  bool wordWrap() const
  ```

- setTextFormat() — задает режим отображения текста. Прототип метода:

  ```
  void setTextFormat(Qt::TextFormat)
  ```

Могут быть указаны следующие константы:

- `Qt::PlainText` — простой текст;
- `Qt::RichText` — форматированный текст;
- `Qt::AutoText` — автоматическое определение (режим по умолчанию). Если текст содержит теги, то используется режим `Qt::RichText`, в противном случае — режим `Qt::PlainText`;
- `Qt::MarkdownText` — текст в формате Markdown.

Получение значения:

```
Qt::TextFormat textFormat() const
```

- `setAlignment()` — задает режим выравнивания текста внутри надписи. Допустимые значения мы рассматривали в *разд. 5.2*. Прототип метода:

```
void setAlignment(Qt::Alignment)
```

Пример:

```
label->setAlignment(Qt::AlignRight | Qt::AlignBottom);
```

Получение значения:

```
Qt::Alignment alignment() const
```

- `setOpenExternalLinks()` — если в качестве параметра указано значение `true`, то щелчок на гиперссылке приведет к открытию браузера, используемого в системе по умолчанию, и загрузке указанной страницы. Прототип метода:

```
void setOpenExternalLinks(bool open)
```

Пример:

```
label->setText(
          "<a href=\"https://google.ru/\">Это гиперссылка</a>");
label->setOpenExternalLinks(true);
```

Получение значения:

```
bool openExternalLinks() const
```

- `setBuddy()` — позволяет связать надпись с другим компонентом. В этом случае в тексте надписи можно задавать клавиши быстрого доступа, указав символ `&` перед буквой или цифрой. После нажатия комбинации клавиш в фокусе ввода окажется компонент, указатель на который передан в качестве параметра. Прототип метода:

```
void setBuddy(QWidget *buddy)
```

Получение значения:

```
QWidget *buddy() const
```

- `setPixmap()` — позволяет вывести изображение на надпись. Метод является слотом. Прототип метода:

```
void setPixmap(const QPixmap &)
```

Пример:

```
label->setPixmap(QPixmap("C:\\cpp\\projectsQt\\Test\\photo.jpg"));
```

Получение значения:

```
QPixmap pixmap() const
```

- `setPicture()` — позволяет вывести рисунок. Метод является слотом. Прототип метода:

  ```
  void setPicture(const QPicture &picture)
  ```

  Получение значения:

  ```
  QPicture picture() const
  ```

- `setMovie()` — позволяет вывести анимацию. Метод является слотом. Прототип метода:

  ```
  void setMovie(QMovie *movie)
  ```

  Получение значения:

  ```
  QMovie *movie() const
  ```

- `setScaledContents()` — если в параметре указано значение `true`, то при изменении размеров надписи размер содержимого также будет изменяться. По умолчанию изменение размеров содержимого не осуществляется. Прототип метода:

  ```
  void setScaledContents(bool)
  ```

  Получение значения:

  ```
  bool hasScaledContents() const
  ```

- `setMargin()` — задает отступ от рамки до содержимого надписи. Прототип метода:

  ```
  void setMargin(int)
  ```

  Получение значения:

  ```
  int margin() const
  ```

- `setIndent()` — задает отступ от рамки до текста надписи в зависимости от значения выравнивания. Если выравнивание производится по левой стороне, то задает отступ слева; если по правой стороне, то справа и т. д. Прототип метода:

  ```
  void setIndent(int)
  ```

  Получение значения:

  ```
  int indent() const
  ```

- `clear()` — удаляет содержимое надписи. Метод является слотом. Прототип метода:

  ```
  void clear()
  ```

- `setTextInteractionFlags()` — задает режим взаимодействия пользователя с текстом надписи. Прототип метода:

  ```
  void setTextInteractionFlags(Qt::TextInteractionFlags flags)
  ```

Можно указать следующие константы (или их комбинацию через оператор |):

- Qt::NoTextInteraction — пользователь не может взаимодействовать с текстом надписи;
- Qt::TextSelectableByMouse — текст можно выделить и скопировать в буфер обмена;
- Qt::TextSelectableByKeyboard — текст можно выделить с помощью клавиш клавиатуры. Внутри надписи будет отображен текстовый курсор;
- Qt::LinksAccessibleByMouse — на гиперссылке можно щелкнуть мышью и скопировать ее адрес;
- Qt::LinksAccessibleByKeyboard — с гиперссылкой можно взаимодействовать с помощью клавиатуры. Перемещаться между гиперссылками можно с помощью клавиши <Tab>, а переходить по гиперссылке — при нажатии клавиши <Enter>;
- Qt::TextEditable — текст надписи можно редактировать;
- Qt::TextEditorInteraction — комбинация TextSelectableByMouse | TextSelectableByKeyboard | TextEditable;
- Qt::TextBrowserInteraction — комбинация TextSelectableByMouse | LinksAccessibleByMouse | LinksAccessibleByKeyboard.

Получение значения:

```
Qt::TextInteractionFlags textInteractionFlags() const
```

- setSelection() — выделяет фрагмент длиной length, начиная с позиции start. Прототип метода:

```
void setSelection(int start, int length)
```

- selectionStart() — возвращает начальный индекс выделенного фрагмента или значение -1, если ничего не выделено. Прототип метода:

```
int selectionStart() const
```

- selectedText() — возвращает выделенный текст или пустую строку. Прототип метода:

```
QString selectedText() const
```

- hasSelectedText() — возвращает значение true, если существует выделенный фрагмент, и false — в противном случае. Прототип метода:

```
bool hasSelectedText() const
```

Класс QLabel содержит следующие сигналы:

- linkActivated(const QString&) — генерируется при переходе по гиперссылке. Через параметр внутри обработчика доступен URL-адрес;
- linkHovered(const QString&) — генерируется при наведении указателя мыши на гиперссылку и выведении указателя. Через параметр внутри обработчика доступен URL-адрес или пустая строка.

6.2. Командная кнопка

Командная кнопка является наиболее часто используемым компонентом. При нажатии кнопки внутри обработчика обычно выполняется какая-либо операция. Кнопка реализуется с помощью класса `QPushButton`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QPushButton
```

Конструктор класса `QPushButton` имеет три формата:

```
#include <QPushButton>
QPushButton(const QString &text, QWidget *parent = nullptr)
QPushButton(const QIcon &icon, const QString &text,
            QWidget *parent = nullptr)
QPushButton(QWidget *parent = nullptr)
```

В параметре `parent` передается указатель на родительский компонент. Если параметр не указан или имеет значение `nullptr`, то компонент будет обладать своим собственным окном. Параметр `text` позволяет задать текст, который будет отображен на кнопке, а параметр `icon` позволяет добавить перед текстом значок.

Класс `QPushButton` наследует следующие методы из класса `QAbstractButton` (перечислены только основные методы; полный список смотрите в документации):

- `setText()` — задает текст, который будет отображен на кнопке. Внутри текста символ `&`, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае буква, перед которой указан символ `&`, будет подчеркнута, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутой буквы кнопка будет нажата. Чтобы вывести символ `&`, необходимо его удвоить. Прототип метода:

  ```
  void setText(const QString &text)
  ```

- `text()` — возвращает текст, отображаемый на кнопке. Прототип метода:

  ```
  QString text() const
  ```

- `setShortcut()` — задает комбинацию клавиш быстрого доступа. Прототип метода:

  ```
  void setShortcut(const QKeySequence &key)
  ```

  Примеры указания значения:

  ```
  btn1->setShortcut(QKeySequence("Alt+B"));
  btn1->setShortcut(QKeySequence(Qt::ALT | Qt::Key_E));
  btn1->setShortcut(QKeySequence::mnemonic("&B"));
  ```

  Получение значения:

  ```
  QKeySequence shortcut() const
  ```

- `setIcon()` — позволяет вставить значок перед текстом. Прототип метода:

  ```
  void setIcon(const QIcon &icon)
  ```

Получение значения:

```
QIcon icon() const
```

- `setIconSize()` — задает размеры значка. Метод является слотом. Прототип метода:

  ```
  void setIconSize(const QSize &size)
  ```

  Получение значения:

  ```
  QSize iconSize() const
  ```

- `setAutoRepeat()` — если в качестве параметра указано значение `true`, то сигнал `clicked()` будет периодически генерироваться, пока кнопка находится в нажатом состоянии. Примером являются кнопки, изменяющие значение полосы прокрутки. Прототип метода:

  ```
  void setAutoRepeat(bool)
  ```

  Получение значения:

  ```
  bool autoRepeat() const
  ```

- `setAutoRepeatDelay()` — задает начальную задержку в миллисекундах перед периодической генерацией сигнала `clicked()`. Прототип метода:

  ```
  void setAutoRepeatDelay(int)
  ```

  Получение значения:

  ```
  int autoRepeatDelay() const
  ```

- `setAutoRepeatInterval()` — задает интервал в миллисекундах перед повторной генерацией сигнала `clicked()`. Прототип метода:

  ```
  void setAutoRepeatInterval(int)
  ```

  Получение значения:

  ```
  int autoRepeatInterval() const
  ```

- `animateClick()` — имитирует нажатие кнопки пользователем. Метод является слотом. Прототип метода:

  ```
  void animateClick()
  ```

- `click()` — имитирует нажатие кнопки без анимации. Метод является слотом. Прототип метода:

  ```
  void click()
  ```

- `toggle()` — переключает кнопку. Метод является слотом. Прототип метода:

  ```
  void toggle()
  ```

- `setCheckable()` — если в качестве параметра указано значение `true`, то кнопка является переключателем, который может находиться в двух состояниях — установленном и неустановленном. Прототип метода:

  ```
  void setCheckable(bool)
  ```

Получение значения:

```
bool isCheckable() const
```

- `setChecked()` — если в качестве параметра указано значение `true`, то кнопка-переключатель будет находиться в установленном состоянии. Метод является слотом. Прототип метода:

```
void setChecked(bool)
```

- `isChecked()` — возвращает значение `true`, если кнопка находится в установленном состоянии, и `false` — в противном случае. Прототип метода:

```
bool isChecked() const
```

- `setAutoExclusive()` — если в качестве параметра указано значение `true`, то внутри группы только одна кнопка-переключатель может быть установлена. Прототип метода:

```
void setAutoExclusive(bool)
```

Получение значения:

```
bool autoExclusive() const
```

- `setDown()` — если в качестве параметра указано значение `true`, то кнопка будет находиться в нажатом состоянии. Прототип метода:

```
void setDown(bool)
```

- `isDown()` — возвращает значение `true`, если кнопка находится в нажатом состоянии, и `false` — в противном случае. Прототип метода:

```
bool isDown() const
```

Кроме перечисленных состояний кнопка может находиться в неактивном состоянии. Для этого необходимо передать значение `false` в метод `setEnabled()` из класса `QWidget`. Метод является слотом. Проверить, активна кнопка или нет, позволяет метод `isEnabled()`. Метод возвращает значение `true`, если кнопка находится в активном состоянии, и `false` — в противном случае. Прототипы методов:

```
void setEnabled(bool)
bool isEnabled() const
```

Класс `QAbstractButton` содержит следующие сигналы:

- `pressed()` — генерируется при нажатии кнопки;
- `released()` — генерируется при отпускании ранее нажатой кнопки;
- `clicked(bool checked=false)` — генерируется при нажатии, а затем отпускании кнопки мыши над кнопкой. Именно для этого сигнала обычно назначают обработчики;
- `toggled(bool)` — генерируется при переключении кнопки. Через параметр внутри обработчика доступно текущее состояние кнопки.

Класс `QPushButton` содержит дополнительные методы:

- `setFlat()` — если в качестве параметра указано значение `true`, то кнопка будет отображаться без границ. Прототип метода:

  ```
  void setFlat(bool)
  ```

  Получение значения:

  ```
  bool isFlat() const
  ```

- `setAutoDefault()` — если в качестве параметра указано значение `true`, то кнопка может быть нажата с помощью клавиши <Enter> при условии, что она находится в фокусе. По умолчанию нажать кнопку позволяет только клавиша <Пробел>. В диалоговых окнах для всех кнопок по умолчанию указано значение `true`, а для остальных окон — значение `false`. Прототип метода:

  ```
  void setAutoDefault(bool)
  ```

  Получение значения:

  ```
  bool autoDefault() const
  ```

- `setDefault()` — задает кнопку по умолчанию. Метод работает только в диалоговых окнах. Эта кнопка может быть нажата с помощью клавиши <Enter>, когда фокус ввода установлен на другой компонент, например на текстовое поле. Прототип метода:

  ```
  void setDefault(bool)
  ```

  Получение значения:

  ```
  bool isDefault() const
  ```

- `setMenu()` — устанавливает всплывающее меню, которое будет отображаться при нажатии кнопки. Прототип метода:

  ```
  void setMenu(QMenu *menu)
  ```

  Получение значения:

  ```
  QMenu *menu() const
  ```

- `showMenu()` — отображает всплывающее меню. Метод является слотом. Прототип метода:

  ```
  void showMenu()
  ```

6.3. Переключатель

Переключатели (иногда их называют *радиокнопками*) обычно используются в группе. Внутри группы может быть включен только один переключатель. При попытке включить другой переключатель ранее включенный переключатель автоматически отключается. Для объединения переключателей в группу можно воспользоваться классом `QGroupBox`, который мы рассматривали в *разд. 5.7*, а также

классом QButtonGroup. Переключатель реализуется с помощью класса QRadioButton. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QRadioButton
```

Конструктор класса QRadioButton имеет два формата:

```
#include <QRadioButton>
QRadioButton(const QString &text, QWidget *parent = nullptr)
QRadioButton(QWidget *parent = nullptr)
```

Класс QRadioButton наследует все методы из класса QAbstractButton *(см. разд. 6.2)*. Включить или отключить переключатель позволяет метод setChecked(), проверить текущий статус можно с помощью метода isChecked(), а чтобы перехватить переключение, следует назначить обработчик сигнала toggled(bool). Через параметр внутри обработчика доступно текущее состояние переключателя.

6.4. Флажок

Флажок предназначен для включения или выключения каких-либо опций настроек программы пользователем и может иметь несколько состояний: установлен, сброшен и частично установлен. Флажок реализуется с помощью класса QCheckBox. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QCheckBox
```

Конструктор класса QCheckBox имеет два формата:

```
#include <QCheckBox>
QCheckBox(const QString &text, QWidget *parent = nullptr)
QCheckBox(QWidget *parent = nullptr)
```

Класс QCheckBox наследует все методы из класса QAbstractButton *(см. разд. 6.2)*, а также добавляет несколько новых:

- setCheckState() — задает статус флажка. Прототип метода:

  ```
  void setCheckState(Qt::CheckState state)
  ```

  Могут быть указаны следующие константы:

  - Qt::Unchecked — флажок сброшен;
  - Qt::PartiallyChecked — флажок частично установлен;
  - Qt::Checked — флажок установлен;

- checkState() — возвращает текущий статус флажка. Прототип метода:

  ```
  Qt::CheckState checkState() const
  ```

- setTristate() — если в качестве параметра указано значение true (значение по умолчанию), то флажок может поддерживать все три статуса. По умолчанию поддерживаются только статусы установлен и сброшен. Прототип метода:

  ```
  void setTristate(bool y = true)
  ```

- `isTristate()` — возвращает значение `true`, если флажок поддерживает три статуса, и `false` — в противном случае. Прототип метода:

  ```
  bool isTristate() const
  ```

Чтобы перехватить смену статуса флажка, следует назначить обработчик сигнала `stateChanged(int)`. Через параметр внутри обработчика доступен текущий статус флажка.

Если используется флажок, поддерживающий только два состояния, то установить или сбросить флажок позволяет метод `setChecked()`, проверить текущий статус можно с помощью метода `isChecked()`, а чтобы перехватить изменение статуса, следует назначить обработчик сигнала `toggled(bool)`. Через параметр внутри обработчика доступен текущий статус флажка.

6.5. Однострочное текстовое поле

Однострочное текстовое поле предназначено для ввода и редактирования текста небольшого объема. С его помощью можно также отобразить вводимые символы в виде звездочек (например, чтобы скрыть пароль) или вообще не отображать их (например, чтобы скрыть длину пароля). Поле по умолчанию поддерживает технологию drag & drop, стандартные комбинации клавиш быстрого доступа, работу с буфером обмена и многое другое. Однострочное текстовое поле реализуется с помощью класса `QLineEdit`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QLineEdit
```

Конструктор класса `QLineEdit` имеет два формата:

```
#include <QLineEdit>
QLineEdit(const QString &contents, QWidget *parent = nullptr)
QLineEdit(QWidget *parent = nullptr)
```

В параметре `parent` передается указатель на родительский компонент. Если параметр не указан или имеет значение `nullptr`, то компонент будет обладать своим собственным окном. Параметр `contents` позволяет задать текст, который будет отображен в однострочном текстовом поле. Пример:

```
QLineEdit *lineEdit = new QLineEdit("Начальное значение");
```

6.5.1. Основные методы и сигналы

Класс `QLineEdit` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setEchoMode()` — задает режим отображения текста. Прототип метода:

  ```
  void setEchoMode(QLineEdit::EchoMode)
  ```

  Могут быть указаны следующие константы:

  - `QLineEdit::Normal` — показывать символы, как они были введены;
  - `QLineEdit::NoEcho` — не показывать вводимые символы;

- `QLineEdit::Password` — вместо символов показывать символ `*`;
- `QLineEdit::PasswordEchoOnEdit` — показывать символы при вводе, а при потере фокуса отображать символ `*`.

Получение значения:

```
QLineEdit::EchoMode echoMode() const
```

- `setCompleter()` — позволяет предлагать возможные варианты значений, начинающиеся с введенных пользователем символов. Прототип метода:

```
void setCompleter(QCompleter *c)
```

Пример:

```
QLineEdit *lineEdit = new QLineEdit();
QStringList list;
list << "кадр" << "каменный" << "камень" << "камера";
QCompleter *completer = new QCompleter(list, &window);
completer->setCaseSensitivity(Qt::CaseInsensitive);
lineEdit->setCompleter(completer);
```

Получение значения:

```
QCompleter *completer() const
```

- `setReadOnly()` — если в качестве параметра указано значение `true`, то поле будет доступно только для чтения. Прототип метода:

```
void setReadOnly(bool)
```

- `isReadOnly()` — возвращает значение `true`, если поле доступно только для чтения, и `false` — в противном случае. Прототип метода:

```
bool isReadOnly() const
```

- `setAlignment()` — задает выравнивание текста внутри поля. Прототип метода:

```
void setAlignment(Qt::Alignment flag)
```

Получение значения:

```
Qt::Alignment alignment() const
```

- `setMaxLength()` — задает максимальное количество символов. Прототип метода:

```
void setMaxLength(int)
```

Получение значения:

```
int maxLength() const
```

- `setFrame()` — если в качестве параметра указано значение `false`, то поле будет отображаться без рамки. Прототип метода:

```
void setFrame(bool)
```

Получение значения:

```
bool hasFrame() const
```

- `setClearButtonEnabled()` — если в качестве параметра указано значение `true`, то в правой части поля будет отображаться значок, при щелчке на котором поле очищается. Если поле пустое, то значок не отображается. Прототип метода:

  ```
  void setClearButtonEnabled(bool enable)
  ```

  Получение значения:

  ```
  bool isClearButtonEnabled() const
  ```

- `setDragEnabled()` — если в качестве параметра указано значение `true`, то режим перетаскивания текста из текстового поля с помощью мыши будет включен. По умолчанию однострочное текстовое поле только принимает перетаскиваемый текст. Прототип метода:

  ```
  void setDragEnabled(bool)
  ```

  Получение значения:

  ```
  bool dragEnabled() const
  ```

- `setPlaceholderText()` — задает текст подсказки пользователю, который будет выводиться в поле, когда оно не содержит значения и находится вне фокуса ввода. Прототип метода:

  ```
  void setPlaceholderText(const QString &)
  ```

  Получение значения:

  ```
  QString placeholderText() const
  ```

- `setText()` — вставляет указанный текст в поле. Метод является слотом. Прототип метода:

  ```
  void setText(const QString &)
  ```

- `insert()` — вставляет текст в текущую позицию текстового курсора. Если в поле был выделен фрагмент, то он будет удален. Прототип метода:

  ```
  void insert(const QString &)
  ```

- `text()` — возвращает текст, содержащийся в текстовом поле. Прототип метода:

  ```
  QString text() const
  ```

- `displayText()` — возвращает текст, который видит пользователь. Результат зависит от режима отображения, заданного с помощью метода `setEchoMode()`. Например, в режиме `Password` строка будет состоять из символов `*`. Прототип метода:

  ```
  QString displayText() const
  ```

- `selectedText()` — возвращает выделенный фрагмент или пустую строку. Прототип метода:

  ```
  QString selectedText() const
  ```

- `clear()` — удаляет весь текст из поля. Метод является слотом. Прототип метода:

  ```
  void clear()
  ```

- `backspace()` — удаляет выделенный фрагмент. Если выделенного фрагмента нет, то удаляет символ, стоящий слева от текстового курсора. Прототип метода:

```
void backspace()
```

- `del()` — удаляет выделенный фрагмент. Если выделенного фрагмента нет, то удаляет символ, стоящий справа от текстового курсора. Прототип метода:

```
void del()
```

- `setSelection()` — выделяет фрагмент длиной `length`, начиная с позиции `start`. Во втором параметре можно указать отрицательное значение. Прототип метода:

```
void setSelection(int start, int length)
```

- `selectAll()` — выделяет весь текст в поле. Метод является слотом. Прототип метода:

```
void selectAll()
```

- `selectionStart()` — возвращает начальный индекс выделенного фрагмента или значение `-1`, если ничего не выделено. Прототип метода:

```
int selectionStart() const
```

- `selectionEnd()` — возвращает конечный индекс выделенного фрагмента или значение `-1`, если ничего не выделено. Прототип метода:

```
int selectionEnd() const
```

- `selectionLength()` — возвращает длину выделенного фрагмента. Прототип метода:

```
int selectionLength() const
```

- `hasSelectedText()` — возвращает значение `true`, если поле содержит выделенный фрагмент, и `false` — в противном случае. Прототип метода:

```
bool hasSelectedText() const
```

- `deselect()` — снимает выделение. Прототип метода:

```
void deselect()
```

- `setCursorPosition()` — задает положение текстового курсора. Прототип метода:

```
void setCursorPosition(int)
```

- `cursorPosition()` — возвращает текущее положение текстового курсора. Прототип метода:

```
int cursorPosition() const
```

- `cursorForward()` — перемещает текстовый курсор вперед на указанное во втором параметре количество символов. Если в первом параметре указано значение `true`, то фрагмент выделяется. Прототип метода:

```
void cursorForward(bool mark, int steps = 1)
```

- `cursorBackward()` — перемещает текстовый курсор назад на указанное во втором параметре количество символов. Если в первом параметре указано значение `true`, то фрагмент выделяется. Прототип метода:

```
void cursorBackward(bool mark, int steps = 1)
```

- `cursorWordForward()` — перемещает текстовый курсор вперед на одно слово. Если в параметре указано значение `true`, то фрагмент выделяется. Прототип метода:

```
void cursorWordForward(bool mark)
```

- `cursorWordBackward()` — перемещает текстовый курсор назад на одно слово. Если в параметре указано значение `true`, то фрагмент выделяется. Прототип метода:

```
void cursorWordBackward(bool mark)
```

- `home()` — перемещает текстовый курсор в начало поля. Если в параметре указано значение `true`, то фрагмент выделяется. Прототип метода:

```
void home(bool mark)
```

- `end()` — перемещает текстовый курсор в конец поля. Если в параметре указано значение `true`, то фрагмент выделяется. Прототип метода:

```
void end(bool mark)
```

- `cut()` — копирует выделенный текст в буфер обмена, а затем удаляет его из поля при условии, что есть выделенный фрагмент и используется режим `Normal`. Метод является слотом. Прототип метода:

```
void cut()
```

- `copy()` — копирует выделенный текст в буфер обмена при условии, что есть выделенный фрагмент и используется режим `Normal`. Метод является слотом. Прототип метода:

```
void copy() const
```

- `paste()` — вставляет текст из буфера обмена в текущую позицию текстового курсора при условии, что поле доступно для редактирования. Метод является слотом. Прототип метода:

```
void paste()
```

- `undo()` — отменяет последнюю операцию ввода пользователем при условии, что отмена возможна. Метод является слотом. Прототип метода:

```
void undo()
```

- `redo()` — повторяет последнюю отмененную операцию ввода пользователем, если это возможно. Метод является слотом. Прототип метода:

```
void redo()
```

- `isUndoAvailable()` — возвращает значение `true`, если можно отменить последнюю операцию ввода, и `false` — в противном случае. Прототип метода:

```
bool isUndoAvailable() const
```

- `isRedoAvailable()` — возвращает значение `true`, если можно повторить последнюю отмененную операцию ввода, и `false` — в противном случае. Прототип метода:

```
bool isRedoAvailable() const
```

- `createStandardContextMenu()` — создает стандартное меню, которое отображается при щелчке правой кнопкой мыши в текстовом поле. Чтобы изменить стандартное меню, следует создать класс, наследующий класс `QLineEdit`, и переопределить метод `contextMenuEvent()`. Внутри этого метода можно создать свое собственное меню или добавить новый пункт в стандартное меню. Прототипы методов:

```
QMenu *createStandardContextMenu()
virtual void contextMenuEvent(QContextMenuEvent *event)
```

- `setTextMargins()` — задает отступы между линией рамки и текстом. Прототипы метода:

```
void setTextMargins(int left, int top, int right, int bottom)
void setTextMargins(const QMargins &margins)
```

  Получение значения:

```
QMargins textMargins() const
```

Класс `QLineEdit` содержит следующие сигналы:

- `cursorPositionChanged(int,int)` — генерируется при перемещении текстового курсора. Внутри обработчика через первый параметр доступна старая позиция курсора, а через второй параметр — новая позиция;
- `editingFinished()` — генерируется при нажатии клавиши <Enter> или потере полем фокуса ввода;
- `returnPressed()` — генерируется при нажатии клавиши <Enter>;
- `selectionChanged()` — генерируется при изменении выделения;
- `inputRejected()` — генерируется при недопустимом вводе;
- `textChanged(const QString&)` — генерируется при изменении текста внутри поля пользователем или программно. Внутри обработчика через параметр доступно новое значение;
- `textEdited(const QString&)` — генерируется при изменении текста внутри поля пользователем. Сигнал не генерируется при изменении текста с помощью метода `setText()`. Внутри обработчика через параметр доступно новое значение.

6.5.2. Ввод данных по маске

С помощью метода `setInputMask()` можно ограничить ввод символов допустимым диапазоном значений. Прототип метода:

```
void setInputMask(const QString &inputMask)
```

Получение значения:

```
QString inputMask() const
```

В качестве параметра указывается строка, имеющая следующий формат:

```
"<Последовательность символов>[;<Символ-заполнитель>]"
```

В первом параметре указывается комбинация из следующих специальных символов:

- `9` — обязательна цифра от `0` до `9`;
- `0` — разрешена, но не обязательна цифра от `0` до `9`;
- `D` — обязательна цифра от `1` до `9`;
- `d` — разрешена, но не обязательна цифра от `1` до `9`;
- `B` — обязательна цифра `0` или `1`;
- `b` — разрешена, но не обязательна цифра `0` или `1`;
- `H` — обязателен шестнадцатеричный символ (`0-9`, `A-F`, `a-f`);
- `h` — разрешен, но не обязателен шестнадцатеричный символ (`0-9`, `A-F`, `a-f`);
- `#` — разрешена, но не обязательна цифра или знак «плюс» или «минус»;
- `A` — обязательна латинская буква в любом регистре;
- `a` — разрешена, но не обязательна латинская буква в любом регистре;
- `N` — обязательна латинская буква в любом регистре или цифра от `0` до `9`;
- `n` — разрешена, но не обязательна латинская буква в любом регистре или цифра от `0` до `9`;
- `X` — обязателен любой символ;
- `x` — разрешен, но не обязателен любой символ;
- `>` — все последующие буквы переводятся в верхний регистр;
- `<` — все последующие буквы переводятся в нижний регистр;
- `!` — отключает изменение регистра;
- `\` — используется для отмены действия спецсимволов.

Все остальные символы трактуются как есть. В необязательном параметре `<Символ-заполнитель>` можно указать символ, который будет отображаться в поле, обозначая место ввода. Если параметр не указан, то символом является пробел. Пример:

```
lineEdit->setInputMask("Дата: 99.B9.9999;#");  // Дата: ##.##.####
lineEdit->setInputMask("Дата: 99.B9.9999;_");  // Дата: __.__.____
lineEdit->setInputMask("Дата: 99.B9.9999 г."); // Дата:   .  .     г.
```

Проверить соответствие введенных данных маске позволяет метод `hasAcceptableInput()`. Если данные соответствуют маске, то метод возвращает значение `true`, в противном случае — `false`. Прототип метода:

```
bool hasAcceptableInput() const
```

6.5.3. Контроль ввода

Контролировать ввод данных позволяет метод `setValidator()`. Прототип метода:

```
void setValidator(const QValidator *v)
```

В качестве значения указывается экземпляр класса, наследующего класс `QValidator`. Существуют следующие стандартные классы, позволяющие контролировать ввод данных:

- `QIntValidator` — допускает ввод только целых чисел. Функциональность класса зависит от настройки локали. Форматы конструктора:

  ```
  #include <QIntValidator>
  QIntValidator(int minimum, int maximum, QObject *parent = nullptr
  QIntValidator(QObject *parent = nullptr)
  ```

  Пример ограничения ввода диапазоном целых чисел от `0` до `100`:

  ```
  lineEdit->setValidator(new QIntValidator(0, 100, this));
  ```

- `QDoubleValidator` — допускает ввод только вещественных чисел. Функциональность класса зависит от настройки локали. Форматы конструктора:

  ```
  #include <QDoubleValidator>
  QDoubleValidator(double bottom, double top, int decimals,
                   QObject *parent = nullptr)
  QDoubleValidator(QObject *parent = nullptr)
  ```

  Пример ограничения ввода диапазоном вещественных чисел от `0.0` до `100.0` и двумя цифрами после десятичной точки:

  ```
  lineEdit->setValidator(new QDoubleValidator(0.0, 100.0, 2, this));
  ```

  Чтобы позволить вводить числа в экспоненциальной форме, необходимо передать значение константы `ScientificNotation` в метод `setNotation()`. Если передать значение константы `StandardNotation`, то число должно быть только в десятичной форме. Пример:

  ```
  QDoubleValidator *validator = new QDoubleValidator(0.0, 100.0, 2, this);
  validator->setNotation(QDoubleValidator::StandardNotation);
  lineEdit->setValidator(validator);
  ```

- `QRegularExpressionValidator` — позволяет проверить данные на соответствие шаблону регулярного выражения. Форматы конструктора:

  ```
  #include <QRegularExpressionValidator>
  QRegularExpressionValidator(const QRegularExpression &re,
                              QObject *parent = nullptr)
  QRegularExpressionValidator(QObject *parent = nullptr)
  ```

Пример ввода только цифр от 0 до 9:

```
QRegularExpression p("[0-9]+");
QRegularExpressionValidator *validator =
      new QRegularExpressionValidator(p, this);
lineEdit->setValidator(validator);
```

Обратите внимание на то, что производится проверка полного соответствия шаблону, поэтому символы ^ и $ явным образом указывать не нужно.

Проверить соответствие введенных данных условию позволяет метод `hasAcceptableInput()`. Если данные соответствуют условию, то метод возвращает значение `true`, в противном случае — `false`. Прототип метода:

```
bool hasAcceptableInput() const
```

6.6. Многострочное текстовое поле

Многострочное текстовое поле предназначено для ввода и редактирования как простого текста, так и текста в формате HTML. Поле по умолчанию поддерживает технологию drag & drop, стандартные комбинации клавиш быстрого доступа, работу с буфером обмена и многое другое. Многострочное текстовое поле реализуется с помощью класса `QTextEdit`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                        — QAbstractScrollArea — QTextEdit
```

Конструктор класса `QTextEdit` имеет два формата:

```
#include <QTextEdit>
QTextEdit(const QString &text, QWidget *parent = nullptr)
QTextEdit(QWidget *parent = nullptr)
```

В параметре `parent` передается указатель на родительский компонент. Если параметр не указан или имеет значение `nullptr`, то компонент будет обладать своим собственным окном. Параметр `text` позволяет задать текст в формате HTML, который будет отображен в текстовом поле.

> ***Примечание***
>
> Класс `QTextEdit` предназначен для отображения как простого текста, так и текста в формате HTML. Если поддержка HTML не нужна, то следует воспользоваться классом `QPlainTextEdit`, который оптимизирован для работы с простым текстом большого объема.

6.6.1. Основные методы и сигналы

Класс `QTextEdit` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setText()` — вставляет указанный текст в поле. Текст может быть простым или в формате HTML. Метод является слотом. Прототип метода:

  ```
  void setText(const QString &text)
  ```

- `setPlainText()` — вставляет простой текст. Метод является слотом. Прототип метода:

  ```
  void setPlainText(const QString &text)
  ```

- `setHtml()` — вставляет текст в формате HTML. Метод является слотом. Прототип метода:

  ```
  void setHtml(const QString &text)
  ```

- `insertPlainText()` — вставляет простой текст в текущую позицию текстового курсора. Если в поле был выделен фрагмент, то он будет удален. Метод является слотом. Прототип метода:

  ```
  void insertPlainText(const QString &text)
  ```

- `insertHtml()` — вставляет текст в формате HTML в текущую позицию текстового курсора. Если в поле был выделен фрагмент, то он будет удален. Метод является слотом. Прототип метода:

  ```
  void insertHtml(const QString &text)
  ```

- `append()` — добавляет новый абзац с указанным текстом в формате HTML в конец поля. Метод является слотом. Прототип метода:

  ```
  void append(const QString &text)
  ```

- `setPlaceholderText()` — задает текст подсказки пользователю, который будет выводиться в поле, когда оно не содержит значения и находится вне фокуса ввода. Прототип метода:

  ```
  void setPlaceholderText(const QString &)
  ```

  Получение значения:

  ```
  QString placeholderText() const
  ```

- `setDocumentTitle()` — задает текст заголовка документа (для тега `<title>`). Прототип метода:

  ```
  void setDocumentTitle(const QString &title)
  ```

- `documentTitle()` — возвращает текст заголовка (из тега `<title>`). Прототип метода:

  ```
  QString documentTitle() const
  ```

- `toPlainText()` — возвращает простой текст, содержащийся в текстовом поле. Прототип метода:

  ```
  QString toPlainText() const
  ```

- `toHtml()` — возвращает текст в формате HTML. Прототип метода:

  ```
  QString toHtml() const
  ```

- `clear()` — удаляет весь текст из поля. Метод является слотом. Прототип метода:

  ```
  void clear()
  ```

- `selectAll()` — выделяет весь текст в поле. Метод является слотом. Прототип метода:

```
void selectAll()
```

- `zoomIn()` — увеличивает масштаб шрифта. Метод является слотом. Прототип метода:

```
void zoomIn(int range = 1)
```

- `zoomOut()` — уменьшает масштаб шрифта. Метод является слотом. Прототип метода:

```
void zoomOut(int range = 1)
```

- `cut()` — копирует выделенный текст в буфер обмена, а затем удаляет его из поля при условии, что есть выделенный фрагмент. Метод является слотом. Прототип метода:

```
void cut()
```

- `copy()` — копирует выделенный текст в буфер обмена при условии, что есть выделенный фрагмент. Метод является слотом. Прототип метода:

```
void copy()
```

- `paste()` — вставляет текст из буфера обмена в текущую позицию текстового курсора при условии, что поле доступно для редактирования. Метод является слотом. Прототип метода:

```
void paste()
```

- `canPaste()` — возвращает `true`, если из буфера обмена можно вставить текст, и `false` — в противном случае. Прототип метода:

```
bool canPaste() const
```

- `setAcceptRichText()` — если в качестве параметра указано значение `true`, то в поле можно будет вставить текст в формате HTML из буфера обмена или с помощью перетаскивания. Значение `false` отключает эту возможность. Прототип метода:

```
void setAcceptRichText(bool accept)
```

- `acceptRichText()` — возвращает значение `true`, если в поле можно вставить текст в формате HTML, и `false` — в противном случае. Прототип метода:

```
bool acceptRichText() const
```

- `undo()` — отменяет последнюю операцию ввода пользователем при условии, что отмена возможна. Метод является слотом. Прототип метода:

```
void undo()
```

- `redo()` — повторяет последнюю отмененную операцию ввода пользователем, если это возможно. Метод является слотом. Прототип метода:

```
void redo()
```

- `setUndoRedoEnabled()` — если в качестве значения указано значение `true`, то операции отмены и повтора действий разрешены, а если `false`, то запрещены. Прототип метода:

```
void setUndoRedoEnabled(bool enable)
```

- `isUndoRedoEnabled()` — возвращает значение `true`, если операции отмены и повтора действий разрешены, и `false`, если запрещены. Прототип метода:

```
bool isUndoRedoEnabled() const
```

- `createStandardContextMenu()` — создает стандартное меню, которое отображается при щелчке правой кнопкой мыши в текстовом поле. Чтобы изменить стандартное меню, следует создать класс, наследующий класс `QTextEdit`, и переопределить метод `contextMenuEvent()`. Внутри этого метода можно создать свое собственное меню или добавить новый пункт в стандартное меню. Прототипы метода:

```
QMenu *createStandardContextMenu()
QMenu *createStandardContextMenu(const QPoint &position)
```

- `ensureCursorVisible()` — прокручивает область таким образом, чтобы текстовый курсор оказался в зоне видимости. Прототип метода:

```
void ensureCursorVisible()
```

- `find()` — производит поиск фрагмента (по умолчанию в прямом направлении без учета регистра символов) в текстовом поле. Если фрагмент найден, то он выделяется и метод возвращает значение `true`, в противном случае — значение `false`. Прототипы метода:

```
bool find(const QString &exp,
   QTextDocument::FindFlags options = QTextDocument::FindFlags())
bool find(const QRegularExpression &exp,
   QTextDocument::FindFlags options = QTextDocument::FindFlags())
```

  В необязательном параметре `options` можно указать комбинацию (через оператор `|`) следующих констант:

  - `QTextDocument::FindBackward` — поиск в обратном направлении, а не в прямом;
  - `QTextDocument::FindCaseSensitively` — поиск с учетом регистра символов;
  - `QTextDocument::FindWholeWords` — поиск слов целиком, а не фрагментов;

- `print()` — отправляет содержимое текстового поля на печать. Прототип метода:

```
void print(QPagedPaintDevice *printer) const
```

Класс `QTextEdit` содержит следующие сигналы:

- `currentCharFormatChanged(const QTextCharFormat&)` — генерируется при изменении формата. Внутри обработчика через параметр доступен новый формат;
- `cursorPositionChanged()` — генерируется при изменении положения текстового курсора;

- `selectionChanged()` — генерируется при изменении выделения;
- `textChanged()` — генерируется при изменении текста внутри поля;
- `copyAvailable(bool)` — генерируется при изменении возможности скопировать фрагмент. Внутри обработчика через параметр доступно значение `true`, если фрагмент можно скопировать, и `false` — в противном случае;
- `undoAvailable(bool)` — генерируется при изменении возможности отменить операцию ввода. Внутри обработчика через параметр доступно значение `true`, если можно отменить операцию ввода, и `false` — в противном случае;
- `redoAvailable(bool)` — генерируется при изменении возможности повторить отмененную операцию ввода. Внутри обработчика через параметр доступно значение `true`, если можно повторить отмененную операцию ввода, и `false` — в противном случае.

6.6.2. Изменение настроек поля

Для изменения настроек предназначены следующие методы из класса `QTextEdit` (перечислены только основные методы; полный список смотрите в документации):

- `setTextInteractionFlags()` — задает режим взаимодействия пользователя с текстом. Прототип метода:

  ```
  void setTextInteractionFlags(Qt::TextInteractionFlags flags)
  ```

  Можно указать следующие константы (или их комбинацию через оператор `|`):

  - `Qt::NoTextInteraction` — пользователь не может взаимодействовать с текстом;
  - `Qt::TextSelectableByMouse` — текст можно выделить и скопировать в буфер обмена;
  - `Qt::TextSelectableByKeyboard` — текст можно выделить с помощью клавиш клавиатуры. Внутри поля будет отображен текстовый курсор;
  - `Qt::LinksAccessibleByMouse` — на гиперссылке можно щелкнуть мышью и скопировать ее адрес;
  - `Qt::LinksAccessibleByKeyboard` — с гиперссылкой можно взаимодействовать с помощью клавиатуры. Перемещаться между гиперссылками можно с помощью клавиши <Tab>, а переходить по гиперссылке — при нажатии клавиши <Enter>;
  - `Qt::TextEditable` — текст можно редактировать;
  - `Qt::TextEditorInteraction` — комбинация `Qt::TextSelectableByMouse | Qt::TextSelectableByKeyboard | Qt::TextEditable`;
  - `Qt::TextBrowserInteraction` — комбинация `Qt::TextSelectableByMouse | Qt::LinksAccessibleByMouse | Qt::LinksAccessibleByKeyboard`.

Получение значения:

```
Qt::TextInteractionFlags textInteractionFlags() const
```

- `setReadOnly()` — если в качестве параметра указано значение `true`, то поле будет доступно только для чтения. Прототип метода:

  ```
  void setReadOnly(bool)
  ```

- `isReadOnly()` — возвращает значение `true`, если поле доступно только для чтения, и `false` — в противном случае. Прототип метода:

  ```
  bool isReadOnly() const
  ```

- `setLineWrapMode()` — задает режим переноса строк. Прототип метода:

  ```
  void setLineWrapMode(QTextEdit::LineWrapMode mode)
  ```

  В качестве значения могут быть указаны следующие константы:

  - `QTextEdit::NoWrap` — перенос строк не производится;
  - `QTextEdit::WidgetWidth` — перенос строки при достижении ширины поля;
  - `QTextEdit::FixedPixelWidth` — перенос строки при достижении фиксированной ширины в пикселах, которую можно задать с помощью метода `setLineWrapColumnOrWidth()`;
  - `QTextEdit::FixedColumnWidth` — перенос строки при достижении фиксированной ширины в буквах, которую можно задать с помощью метода `setLineWrapColumnOrWidth()`.

  Получение значения:

  ```
  QTextEdit::LineWrapMode lineWrapMode() const
  ```

- `setLineWrapColumnOrWidth()` — задает ширину колонки. Прототип метода:

  ```
  void setLineWrapColumnOrWidth(int w)
  ```

  Получение значения:

  ```
  int lineWrapColumnOrWidth() const
  ```

- `setWordWrapMode()` — задает режим переноса по словам. Прототип метода:

  ```
  void setWordWrapMode(QTextOption::WrapMode policy)
  ```

  В качестве значения могут быть указаны следующие константы:

  - `QTextOption::NoWrap` — перенос по словам не производится;
  - `QTextOption::WordWrap` — перенос строк только по словам;
  - `QTextOption::ManualWrap` — аналогичен режиму `NoWrap`;
  - `QTextOption::WrapAnywhere` — перенос строки может быть внутри слова;
  - `QTextOption::WrapAtWordBoundaryOrAnywhere` — по возможности перенос по словам. Если это невозможно, то перенос строки может быть внутри слова.

Получение значения:

```
QTextOption::WrapMode wordWrapMode() const
```

- `setOverwriteMode()` — если в качестве параметра указано значение `true`, то вводимый текст будет замещать ранее введенный. Значение `false` отключает замещение. Прототип метода:

```
void setOverwriteMode(bool overwrite)
```

- `overwriteMode()` — возвращает значение `true`, если вводимый текст замещает ранее введенный, и `false` — в противном случае. Прототип метода:

```
bool overwriteMode() const
```

- `setAutoFormatting()` — задает режим автоматического форматирования. Прототип метода:

```
void setAutoFormatting(QTextEdit::AutoFormatting features)
```

В качестве значения могут быть указаны следующие константы:

- `QTextEdit::AutoNone` — автоматическое форматирование не используется;
- `QTextEdit::AutoBulletList` — автоматически создавать маркированный список при вводе пользователем в начале строки символа `*`;
- `QTextEdit::AutoAll` — включить все автоматические режимы. Эквивалентно режиму `AutoBulletList`.

Получение значения:

```
QTextEdit::AutoFormatting autoFormatting() const
```

- `setCursorWidth()` — задает ширину текстового курсора. Прототип метода:

```
void setCursorWidth(int width)
```

Получение значения:

```
int cursorWidth() const
```

- `setTabChangesFocus()` — если в качестве параметра указано значение `false`, то с помощью нажатия клавиши <Tab> можно вставить символ табуляции в поле. Если указано значение `true`, то клавиша <Tab> используется для передачи фокуса между компонентами. Прототип метода:

```
void setTabChangesFocus(bool)
```

Получение значения:

```
bool tabChangesFocus() const
```

- `setTabStopDistance()` — задает ширину символа табуляции в пикселах. Прототип метода:

```
void setTabStopDistance(qreal distance)
```

Получение значения:

```
qreal tabStopDistance() const
```

6.6.3. Изменение характеристик текста и фона

Для изменения характеристик текста и фона предназначены следующие методы из класса `QTextEdit` (перечислены только основные методы; полный список смотрите в документации):

- `setCurrentFont()` — задает текущий шрифт. Метод является слотом. Прототип метода:

  ```
  void setCurrentFont(const QFont &f)
  ```

  В качестве параметра указывается экземпляр класса `QFont`. Конструктор класса `QFont` имеет следующие форматы:

  ```
  QFont()
  QFont(const QStringList &families, int pointSize = -1,
        int weight = -1, bool italic = false)
  QFont(const QFont &font)
  QFont(const QFont &font, const QPaintDevice *pd)
  ```

  В параметре `families` указывается название шрифта. Необязательный параметр `pointSize` задает размер шрифта. В параметре `weight` можно указать степень жирности шрифта в виде констант `Light`, `Normal`, `DemiBold`, `Bold` или `Black`. Если в параметре `italic` указано значение `true`, то шрифт будет курсивным;

- `currentFont()` — возвращает экземпляр класса `QFont` с текущими характеристиками шрифта. Прототип метода:

  ```
  QFont currentFont() const
  ```

- `setFontFamily()` — задает название текущего шрифта. Метод является слотом. Прототип метода:

  ```
  void setFontFamily(const QString &fontFamily)
  ```

- `fontFamily()` — возвращает название текущего шрифта. Прототип метода:

  ```
  QString fontFamily() const
  ```

- `setFontPointSize()` — задает размер текущего шрифта. Метод является слотом. Прототип метода:

  ```
  void setFontPointSize(qreal)
  ```

- `fontPointSize()` — возвращает размер текущего шрифта. Прототип метода:

  ```
  qreal fontPointSize() const
  ```

- `setFontWeight()` — задает жирность текущего шрифта. Метод является слотом. Прототип метода:

  ```
  void setFontWeight(int weight)
  ```

- `fontWeight()` — возвращает жирность текущего шрифта. Прототип метода:

  ```
  int fontWeight() const
  ```

- `setFontItalic()` — если в качестве параметра указано значение `true`, то шрифт будет курсивным. Метод является слотом. Прототип метода:

  ```
  void setFontItalic(bool italic)
  ```

- `fontItalic()` — возвращает `true`, если шрифт курсивный, и `false` — в противном случае. Прототип метода:

  ```
  bool fontItalic() const
  ```

- `setFontUnderline()` — если в качестве параметра указано значение `true`, то текст будет подчеркнутым. Метод является слотом. Прототип метода:

  ```
  void setFontUnderline(bool underline)
  ```

- `fontUnderline()` — возвращает `true`, если текст подчеркнутый, и `false` — в противном случае. Прототип метода:

  ```
  bool fontUnderline() const
  ```

- `setTextColor()` — задает цвет текста. В качестве значения можно указать константу (например: `Qt::black`, `Qt::white` и т. д.) или экземпляр класса `QColor` (например: `QColor("red")`, `QColor(255, 0, 0)` и др.). Метод является слотом. Прототип метода:

  ```
  void setTextColor(const QColor &c)
  ```

- `textColor()` — возвращает экземпляр класса `QColor` с цветом текущего текста. Прототип метода:

  ```
  QColor textColor() const
  ```

- `setTextBackgroundColor()` — задает цвет фона. В качестве значения можно указать константу (например: `Qt::black`, `Qt::white` и т. д.) или экземпляр класса `QColor` (например: `QColor("red")`, `QColor(255, 0, 0)` и др.). Метод является слотом. Прототип метода:

  ```
  void setTextBackgroundColor(const QColor &c)
  ```

- `textBackgroundColor()` — возвращает экземпляр класса `QColor` с цветом фона. Прототип метода:

  ```
  QColor textBackgroundColor() const
  ```

- `setAlignment()` — задает горизонтальное выравнивание текста внутри абзаца. Допустимые значения мы рассматривали в *разд. 5.2*. Метод является слотом. Прототип метода:

  ```
  void setAlignment(Qt::Alignment)
  ```

- `alignment()` — возвращает значение выравнивания текста внутри абзаца. Прототип метода:

  ```
  Qt::Alignment alignment() const
  ```

Задать формат символов можно также с помощью класса `QTextCharFormat`, который содержит дополнительные настройки. После создания экземпляр класса следует передать в метод `setCurrentCharFormat()`. Прототип метода:

```
void setCurrentCharFormat(const QTextCharFormat &format)
```

Получить экземпляр класса с текущими настройками позволяет метод `currentCharFormat()`. Прототип метода:

```
QTextCharFormat currentCharFormat() const
```

За подробной информацией по классу `QTextCharFormat` обращайтесь к документации.

6.6.4. Класс *QTextDocument*

Класс `QTextDocument` реализует документ, который отображается в многострочном текстовом поле. Получить указатель на текущий документ позволяет метод `document()` из класса `QTextEdit`. Прототип метода:

```
QTextDocument *document() const
```

Установить новый документ можно с помощью метода `setDocument()`. Прототип метода:

```
void setDocument(QTextDocument *document)
```

Иерархия наследования:

```
QObject — QTextDocument
```

Конструктор класса `QTextDocument` имеет два формата:

```
#include <QTextDocument>
QTextDocument(const QString &text, QObject *parent = nullptr)
QTextDocument(QObject *parent = nullptr)
```

В параметре `parent` передается указатель на родительский компонент. Параметр `text` позволяет задать текст в простом формате (не в HTML-формате), который будет отображен в текстовом поле.

Класс `QTextDocument` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setPlainText()` — вставляет простой текст. Прототип метода:

  ```
  void setPlainText(const QString &text)
  ```

- `setHtml()` — вставляет текст в формате HTML. Прототип метода:

  ```
  void setHtml(const QString &html)
  ```

- `toPlainText()` — возвращает простой текст, содержащийся в документе. Прототип метода:

  ```
  QString toPlainText() const
  ```

- `toHtml()` — возвращает текст в формате HTML. Прототип метода:

  ```
  QString toHtml() const
  ```

- `clear()` — удаляет весь текст из документа. Прототип метода:

  ```
  virtual void clear()
  ```

- `isEmpty()` — возвращает значение `true`, если документ пустой, и `false` — в противном случае. Прототип метода:

  ```
  bool isEmpty() const
  ```

- `isModified()` — возвращает значение `true`, если документ был изменен, и `false` — в противном случае. Прототип метода:

  ```
  bool isModified() const
  ```

  Изменить значение позволяет следующий слот:

  ```
  void setModified(bool m = true)
  ```

- `undo()` — отменяет последнюю операцию ввода пользователем при условии, что отмена возможна. Метод является слотом. Прототип метода:

  ```
  void undo()
  ```

- `redo()` — повторяет последнюю отмененную операцию ввода пользователем, если это возможно. Метод является слотом. Прототип метода:

  ```
  void redo()
  ```

- `isUndoAvailable()` — возвращает значение `true`, если можно отменить последнюю операцию ввода, и `false` — в противном случае. Прототип метода:

  ```
  bool isUndoAvailable() const
  ```

- `isRedoAvailable()` — возвращает значение `true`, если можно повторить последнюю отмененную операцию ввода, и `false` — в противном случае. Прототип метода:

  ```
  bool isRedoAvailable() const
  ```

- `setUndoRedoEnabled()` — если в качестве параметра указано значение `true`, то операции отмены и повтора действий разрешены, а если `false`, то запрещены. Прототип метода:

  ```
  void setUndoRedoEnabled(bool enable)
  ```

- `isUndoRedoEnabled()` — возвращает значение `true`, если операции отмены и повтора действий разрешены, и `false`, если запрещены. Прототип метода:

  ```
  bool isUndoRedoEnabled() const
  ```

- `availableUndoSteps()` — возвращает количество возможных операций отмены. Прототип метода:

  ```
  int availableUndoSteps() const
  ```

- `availableRedoSteps()` — возвращает количество возможных повторов отмененных операций. Прототип метода:

  ```
  int availableRedoSteps() const
  ```

- `clearUndoRedoStacks()` — очищает список возможных отмен и/или повторов. Прототип метода:

```
void clearUndoRedoStacks(QTextDocument::Stacks
                         stacksToClear = UndoAndRedoStacks)
```

В качестве параметра можно указать следующие константы:

- `QTextDocument::UndoStack` — только список возможных отмен;
- `QTextDocument::RedoStack` — только список возможных повторов;
- `QTextDocument::UndoAndRedoStacks` — очищаются оба списка;

❐ `print()` — отправляет содержимое документа на печать. Прототип метода:

```
void print(QPagedPaintDevice *printer) const
```

❐ `find()` — производит поиск фрагмента в документе. Метод возвращает экземпляр класса `QTextCursor`. Если фрагмент не найден, то объект курсора будет нулевым. Проверить успешность операции можно с помощью метода `isNull()` объекта курсора. Прототипы метода:

```
QTextCursor find(const QString &subString,
    const QTextCursor &cursor,
    QTextDocument::FindFlags options = FindFlags()) const
QTextCursor find(const QString &subString, int position = 0,
    QTextDocument::FindFlags options = FindFlags()) const
QTextCursor find(const QRegularExpression &expr, int from = 0,
    QTextDocument::FindFlags options = FindFlags()) const
QTextCursor find(const QRegularExpression &expr,
    const QTextCursor &cursor,
    QTextDocument::FindFlags options = FindFlags()) const
```

Параметр `subString` задает искомый фрагмент, а параметр `expr` позволяет указать регулярное выражение. По умолчанию обычный поиск производится без учета регистра символов в прямом направлении, начиная с позиции `position` или от текстового курсора, указанного в параметре `cursor`. В необязательном параметре `options` можно указать комбинацию (через оператор `|`) следующих констант:

- `QTextDocument::FindBackward` — поиск в обратном направлении, а не в прямом;
- `QTextDocument::FindCaseSensitively` — поиск с учетом регистра символов;
- `QTextDocument::FindWholeWords` — поиск слов целиком, а не фрагментов;

❐ `setDefaultFont()` — задает шрифт по умолчанию для документа. Прототип метода:

```
void setDefaultFont(const QFont &font)
```

Получение значения:

```
QFont defaultFont() const
```

В качестве параметра указывается экземпляр класса `QFont`. Конструктор класса `QFont` имеет следующие форматы:

```
QFont()
QFont(const QStringList &families, int pointSize = -1,
      int weight = -1, bool italic = false)
QFont(const QFont &font)
QFont(const QFont &font, const QPaintDevice *pd)
```

В параметре `families` указывается название шрифта. Необязательный параметр `pointSize` задает размер шрифта. В параметре `weight` можно указать степень жирности шрифта в виде констант `Light`, `Normal`, `DemiBold`, `Bold` или `Black`. Если в параметре `italic` указано значение `true`, то шрифт будет курсивным;

- `setDefaultStyleSheet()` — устанавливает таблицу стилей CSS по умолчанию для документа. Прототип метода:

  ```
  void setDefaultStyleSheet(const QString &sheet)
  ```

  Получение значения:

  ```
  QString defaultStyleSheet() const
  ```

- `setDocumentMargin()` — задает отступ от краев поля до текста. Прототип метода:

  ```
  void setDocumentMargin(qreal margin)
  ```

- `documentMargin()` — возвращает величину отступа от краев поля до текста. Прототип метода:

  ```
  qreal documentMargin() const
  ```

- `setMaximumBlockCount()` — задает максимальное количество текстовых блоков в документе. Если количество блоков становится больше указанного значения, то первый блок будет удален. Прототип метода:

  ```
  void setMaximumBlockCount(int maximum)
  ```

- `maximumBlockCount()` — возвращает максимальное количество текстовых блоков. Прототип метода:

  ```
  int maximumBlockCount() const
  ```

- `characterCount()` — возвращает количество символов в документе. Прототип метода:

  ```
  int characterCount() const
  ```

- `lineCount()` — возвращает количество абзацев в документе. Прототип метода:

  ```
  int lineCount() const
  ```

- `blockCount()` — возвращает количество текстовых блоков в документе. Прототип метода:

  ```
  int blockCount() const
  ```

- `firstBlock()` — возвращает экземпляр класса `QTextBlock`, который содержит первый текстовый блок документа. Прототип метода:

  ```
  QTextBlock firstBlock() const
  ```

- `lastBlock()` — возвращает экземпляр класса `QTextBlock`, который содержит последний текстовый блок документа. Прототип метода:

```
QTextBlock lastBlock() const
```

- `findBlock()` — возвращает экземпляр класса `QTextBlock`, который содержит текстовый блок документа, включающий символ с указанным индексом. Прототип метода:

```
QTextBlock findBlock(int pos) const
```

- `findBlockByNumber()` — возвращает экземпляр класса `QTextBlock`, который содержит текстовый блок документа с указанным индексом. Прототип метода:

```
QTextBlock findBlockByNumber(int blockNumber) const
```

Класс `QTextDocument` содержит следующие основные сигналы (полный список смотрите в документации):

- `undoAvailable(bool)` — генерируется при изменении возможности отменить операцию ввода. Внутри обработчика через параметр доступно значение `true`, если можно отменить операцию ввода, и `false` — в противном случае;
- `redoAvailable(bool)` — генерируется при изменении возможности повторить отмененную операцию ввода. Внутри обработчика через параметр доступно значение `true`, если можно повторить отмененную операцию ввода, и `false` — в противном случае;
- `undoCommandAdded()` — генерируется при добавлении операции ввода в список возможных отмен;
- `blockCountChanged(int)` — генерируется при изменении количества текстовых блоков. Внутри обработчика через параметр доступно новое количество текстовых блоков;
- `cursorPositionChanged(const QTextCursor&)` — генерируется при изменении позиции текстового курсора из-за операции редактирования. Обратите внимание на то, что при простом перемещении текстового курсора сигнал не генерируется;
- `contentsChange(int,int,int)` — генерируется при изменении текста. Внутри обработчика через первый параметр доступен индекс позиции внутри документа, через второй параметр — количество удаленных символов, а через третий параметр — количество добавленных символов;
- `contentsChanged()` — генерируется при любом изменении документа;
- `modificationChanged(bool)` — генерируется при изменении статуса документа.

6.6.5. Класс *QTextCursor*

Класс `QTextCursor` реализует текстовый курсор, выделение и позволяет изменять документ. Конструктор класса `QTextCursor` имеет следующие форматы:

```
#include <QTextCursor>
QTextCursor()
QTextCursor(QTextDocument *document)
QTextCursor(QTextFrame *frame)
QTextCursor(const QTextBlock &block)
QTextCursor(const QTextCursor &cursor)
```

Создать текстовый курсор, установить его в документе и управлять им позволяют следующие методы из класса `QTextEdit`:

- `textCursor()` — возвращает видимый в данный момент текстовый курсор (экземпляр класса `QTextCursor`). Чтобы изменения затронули текущий документ, необходимо передать этот объект в метод `setTextCursor()`. Прототип метода:

  ```
  QTextCursor textCursor() const
  ```

- `setTextCursor()` — устанавливает текстовый курсор, экземпляр которого указан в качестве параметра. Прототип метода:

  ```
  void setTextCursor(const QTextCursor &cursor)
  ```

- `cursorForPosition()` — возвращает текстовый курсор, который соответствует позиции, указанной в качестве параметра. Позиция задается с помощью экземпляра класса `QPoint` в координатах области. Прототип метода:

  ```
  QTextCursor cursorForPosition(const QPoint &pos) const
  ```

- `moveCursor()` — перемещает текстовый курсор внутри документа. Прототип метода:

  ```
  void moveCursor(QTextCursor::MoveOperation operation,
       QTextCursor::MoveMode mode = QTextCursor::MoveAnchor)
  ```

  В первом параметре можно указать следующие константы:

  - `QTextCursor::NoMove` — не перемещать курсор;
  - `QTextCursor::Start` — в начало документа;
  - `QTextCursor::Up` — на одну строку вверх;
  - `QTextCursor::StartOfLine` — в начало текущей строки;
  - `QTextCursor::StartOfBlock` — в начало текущего текстового блока;
  - `QTextCursor::StartOfWord` — в начало текущего слова;
  - `QTextCursor::PreviousBlock` — в начало предыдущего текстового блока;
  - `QTextCursor::PreviousCharacter` — сдвинуть на один символ влево;
  - `QTextCursor::PreviousWord` — в начало предыдущего слова;
  - `QTextCursor::Left` — сдвинуть на один символ влево;
  - `QTextCursor::WordLeft` — влево на одно слово;
  - `QTextCursor::End` — в конец документа;

- `QTextCursor::Down` — на одну строку вниз;
- `QTextCursor::EndOfLine` — в конец текущей строки;
- `QTextCursor::EndOfWord` — в конец текущего слова;
- `QTextCursor::EndOfBlock` — в конец текущего текстового блока;
- `QTextCursor::NextBlock` — в начало следующего текстового блока;
- `QTextCursor::NextCharacter` — сдвинуть на один символ вправо;
- `QTextCursor::NextWord` — в начало следующего слова;
- `QTextCursor::Right` — сдвинуть на один символ вправо;
- `QTextCursor::WordRight` — в начало следующего слова.

Помимо перечисленных констант существуют также константы `NextCell`, `PreviousCell`, `NextRow` и `PreviousRow`, позволяющие перемещать текстовый курсор внутри таблицы. В необязательном параметре `mode` можно указать следующие константы:

- `QTextCursor::MoveAnchor` — если существует выделенный фрагмент, то выделение будет снято и текстовый курсор переместится в новое место (значение по умолчанию);
- `QTextCursor::KeepAnchor` — фрагмент текста от старой позиции курсора до новой будет выделен.

Класс `QTextCursor` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `isNull()` — возвращает значение `true`, если объект курсора является нулевым (создан с помощью конструктора без параметра), и `false` — в противном случае. Прототип метода:

```
bool isNull() const
```

- `setPosition()` — перемещает текстовый курсор внутри документа. В первом параметре указывается позиция внутри документа. Необязательный параметр `mode` аналогичен одноименному параметру в методе `moveCursor()` из класса `QTextEdit`. Прототип метода:

```
void setPosition(int pos, QTextCursor::MoveMode m = MoveAnchor)
```

- `movePosition()` — перемещает текстовый курсор внутри документа. Прототип метода:

```
bool movePosition(QTextCursor::MoveOperation operation,
     QTextCursor::MoveMode mode = MoveAnchor, int n = 1)
```

Параметры `operation` и `mode` аналогичны одноименным параметрам в методе `moveCursor()` из класса `QTextEdit`. Необязательный параметр `n` позволяет указать количество перемещений. Например, переместить курсор на `10` символов вперед можно так:

```
QTextCursor cur = textEdit->textCursor();
cur.movePosition(QTextCursor::NextCharacter,
                 QTextCursor::MoveAnchor, 10);
textEdit->setTextCursor(cur);
```

Метод `movePosition()` возвращает значение `true`, если операция успешно выполнена указанное количество раз. Если было выполнено меньшее количество перемещений (например, из-за достижения конца документа), то метод возвращает значение `false`;

- `position()` — возвращает позицию текстового курсора внутри документа. Прототип метода:

  ```
  int position() const
  ```

- `positionInBlock()` — возвращает позицию текстового курсора внутри блока. Прототип метода:

  ```
  int positionInBlock() const
  ```

- `block()` — возвращает экземпляр класса `QTextBlock`, который описывает текстовый блок, содержащий курсор. Прототип метода:

  ```
  QTextBlock block() const
  ```

- `blockNumber()` — возвращает индекс текстового блока, содержащего курсор. Прототип метода:

  ```
  int blockNumber() const
  ```

- `atStart()` — возвращает значение `true`, если текстовый курсор находится в начале документа, и `false` — в противном случае. Прототип метода:

  ```
  bool atStart() const
  ```

- `atEnd()` — возвращает значение `true`, если текстовый курсор находится в конце документа, и `false` — в противном случае. Прототип метода:

  ```
  bool atEnd() const
  ```

- `atBlockStart()` — возвращает значение `true`, если текстовый курсор находится в начале блока, и `false` — в противном случае. Прототип метода:

  ```
  bool atBlockStart() const
  ```

- `atBlockEnd()` — возвращает значение `true`, если текстовый курсор находится в конце блока, и `false` — в противном случае. Прототип метода:

  ```
  bool atBlockEnd() const
  ```

- `select()` — выделяет фрагмент в документе в соответствии с указанным режимом. Прототип метода:

  ```
  void select(QTextCursor::SelectionType selection)
  ```

  В качестве параметра можно указать следующие константы:

  - `QTextCursor::WordUnderCursor` — выделяет слово, в котором расположен курсор;

- `QTextCursor::LineUnderCursor` — выделяет текущую строку;
- `QTextCursor::BlockUnderCursor` — выделяет текущий текстовый блок;
- `QTextCursor::Document` — выделяет весь документ;

❐ `hasSelection()` — возвращает значение `true`, если существует выделенный фрагмент, и `false` — в противном случае. Прототип метода:

```
bool hasSelection() const
```

❐ `hasComplexSelection()` — возвращает значение `true`, если выделенный фрагмент содержит сложное форматирование, а не просто текст, и `false` — в противном случае. Прототип метода:

```
bool hasComplexSelection() const
```

❐ `clearSelection()` — снимает выделение. Прототип метода:

```
void clearSelection()
```

❐ `selectionStart()` — возвращает начальную позицию выделенного фрагмента. Прототип метода:

```
int selectionStart() const
```

❐ `selectionEnd()` — возвращает конечную позицию выделенного фрагмента. Прототип метода:

```
int selectionEnd() const
```

❐ `selectedText()` — возвращает текст выделенного фрагмента. Обратите внимание: если выделенный фрагмент занимает несколько строк, то вместо символа перевода строки вставляется символ с кодом `\u2029`. Прототип метода:

```
QString selectedText() const
```

❐ `selection()` — возвращает экземпляр класса `QTextDocumentFragment`, который описывает выделенный фрагмент. Получить текст позволяют методы `toPlainText()` (возвращает простой текст) и `toHtml()` (возвращает текст в формате HTML) из этого класса. Прототип метода:

```
QTextDocumentFragment selection() const
```

❐ `removeSelectedText()` — удаляет выделенный фрагмент. Прототип метода:

```
void removeSelectedText()
```

❐ `deleteChar()` — если нет выделенного фрагмента, то удаляет символ справа от курсора, в противном случае удаляет выделенный фрагмент. Прототип метода:

```
void deleteChar()
```

❐ `deletePreviousChar()` — если нет выделенного фрагмента, то удаляет символ слева от курсора, в противном случае удаляет выделенный фрагмент. Прототип метода:

```
void deletePreviousChar()
```

- `beginEditBlock()` и `endEditBlock()` — задают начало и конец блока инструкций. Эти инструкции могут быть отменены или повторены как единое целое с помощью методов `undo()` и `redo()`. Прототипы методов:

```
void beginEditBlock()
void endEditBlock()
```

- `joinPreviousEditBlock()` — делает последующие инструкции частью предыдущего блока инструкций. Прототип метода:

```
void joinPreviousEditBlock()
```

- `setKeepPositionOnInsert()` — если в качестве параметра указано значение `true`, то после операции вставки курсор сохранит свою предыдущую позицию. По умолчанию позиция курсора при вставке изменяется. Прототип метода:

```
void setKeepPositionOnInsert(bool)
```

- `insertText()` — вставляет простой текст. Прототипы метода:

```
void insertText(const QString &text)
void insertText(const QString &text,
                const QTextCharFormat &format)
```

- `insertHtml()` — вставляет текст в формате HTML. Прототип метода:

```
void insertHtml(const QString &html)
```

С помощью методов `insertBlock()`, `insertFragment()`, `insertFrame()`, `insertImage()`, `insertList()` и `insertTable()` можно вставить различные элементы, например изображения, списки и др. Изменить формат выделенного фрагмента позволяют методы `mergeBlockCharFormat()`, `mergeBlockFormat()` и `mergeCharFormat()`. За подробной информацией по этим методам обращайтесь к документации.

6.7. Текстовый браузер

Класс `QTextBrowser` расширяет возможности класса `QTextEdit` и реализует текстовый браузер с возможностью перехода по гиперссылкам при щелчке мышью. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                        — QAbstractScrollArea — QTextEdit — QTextBrowser
```

Формат конструктора класса `QTextBrowser`:

```
#include <QTextBrowser>
QTextBrowser(QWidget *parent = nullptr)
```

Класс `QTextBrowser` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setSource()` — загружает ресурс. Метод является слотом. Прототип метода:

```
void setSource(const QUrl &url, QTextDocument::ResourceType
               type = QTextDocument::UnknownResource)
```

- `source()` — возвращает экземпляр класса `QUrl` с адресом текущего ресурса. Прототип метода:

```
QUrl source() const
```

- `reload()` — перезагружает текущий ресурс. Метод является слотом. Прототип метода:

```
virtual void reload()
```

- `home()` — загружает первый ресурс из списка истории. Метод является слотом. Прототип метода:

```
virtual void home()
```

- `backward()` — загружает предыдущий ресурс из списка истории. Метод является слотом. Прототип метода:

```
virtual void backward()
```

- `forward()` — загружает следующий ресурс из списка истории. Метод является слотом. Прототип метода:

```
virtual void forward()
```

- `backwardHistoryCount()` — возвращает количество предыдущих ресурсов. Прототип метода:

```
int backwardHistoryCount() const
```

- `forwardHistoryCount()` — возвращает количество следующих ресурсов. Прототип метода:

```
int forwardHistoryCount() const
```

- `isBackwardAvailable()` — возвращает значение `true`, если существует предыдущий ресурс в списке истории, и `false` — в противном случае. Прототип метода:

```
bool isBackwardAvailable() const
```

- `isForwardAvailable()` — возвращает значение `true`, если существует следующий ресурс в списке истории, и `false` — в противном случае. Прототип метода:

```
bool isForwardAvailable() const
```

- `clearHistory()` — очищает список истории. Прототип метода:

```
void clearHistory()
```

- `historyTitle()` — если в качестве параметра указано отрицательное число, то возвращает заголовок предыдущего ресурса, если `0` — заголовок текущего ресурса, а если положительное число — заголовок следующего ресурса. Прототип метода:

```
QString historyTitle(int) const
```

- `historyUrl()` — если в качестве параметра указано отрицательное число, то возвращает URL-адрес (экземпляр класса `QUrl`) предыдущего ресурса, если `0` —

URL-адрес текущего ресурса, а если положительное число — URL-адрес следующего ресурса. Прототип метода:

```
QUrl historyUrl(int) const
```

- `setOpenLinks()` — если в качестве параметра указано значение `true`, то автоматический переход по гиперссылкам разрешен (значение по умолчанию). Значение `false` запрещает переход. Прототип метода:

```
void setOpenLinks(bool)
```

Класс `QTextBrowser` содержит следующие сигналы:

- `anchorClicked(const QUrl&)` — генерируется при переходе по гиперссылке. Внутри обработчика через параметр доступен URL-адрес гиперссылки;
- `backwardAvailable(bool)` — генерируется при изменении статуса списка предыдущих ресурсов. Внутри обработчика через параметр доступен статус;
- `forwardAvailable(bool)` — генерируется при изменении статуса списка следующих ресурсов. Внутри обработчика через параметр доступен статус;
- `highlighted(const QUrl&)` — генерируется при наведении указателя мыши на гиперссылку и выведении его. Внутри обработчика через параметр доступен URL-адрес ссылки (экземпляр класса `QUrl`) или пустой объект;
- `historyChanged()` — генерируется при изменении списка истории;
- `sourceChanged(const QUrl&)` — генерируется при загрузке нового ресурса. Внутри обработчика через параметр доступен URL-адрес ресурса.

6.8. Поля для ввода целых и вещественных чисел

Для ввода чисел предназначены классы `QSpinBox` (поле для ввода целых чисел) и `QDoubleSpinBox` (поле для ввода вещественных чисел). Поля могут содержать две кнопки, которые позволяют увеличивать и уменьшать значение внутри поля с помощью щелчка мышью. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QSpinBox
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDoubleSpinBox
```

Форматы конструкторов классов `QSpinBox` и `QDoubleSpinBox`:

```
#include <QSpinBox>
QSpinBox(QWidget *parent = nullptr)
#include <QDoubleSpinBox>
QDoubleSpinBox(QWidget *parent = nullptr)
```

Классы `QSpinBox` и `QDoubleSpinBox` наследуют следующие методы из класса `QAbstractSpinBox` (перечислены только основные методы; полный список смотрите в документации):

- `setButtonSymbols()` — задает режим отображения кнопок, предназначенных для изменения значения поля с помощью мыши. Прототип метода:

  ```
  void setButtonSymbols(QAbstractSpinBox::ButtonSymbols)
  ```

  Можно указать следующие константы:

  - `QAbstractSpinBox::UpDownArrows` — отображаются кнопки со стрелками;
  - `QAbstractSpinBox::PlusMinus` — отображаются кнопки с символами + и –. Обратите внимание на то, что при использовании некоторых стилей данное значение может быть проигнорировано;
  - `QAbstractSpinBox::NoButtons` — кнопки не отображаются.

  Получение значения:

  ```
  QAbstractSpinBox::ButtonSymbols buttonSymbols() const
  ```

- `setAlignment()` — задает режим выравнивания значения внутри поля. Прототип метода:

  ```
  void setAlignment(Qt::Alignment flag)
  ```

  Получение значения:

  ```
  Qt::Alignment alignment() const
  ```

- `setWrapping()` — если в качестве параметра указано значение `true`, то значение внутри поля будет изменяться по кругу при нажатии кнопок, например максимальное значение сменится минимальным. Прототип метода:

  ```
  void setWrapping(bool)
  ```

  Получение значения:

  ```
  bool wrapping() const
  ```

- `setSpecialValueText()` — позволяет задать строку, которая будет отображаться внутри поля вместо минимального значения. Прототип метода:

  ```
  void setSpecialValueText(const QString &txt)
  ```

  Получение значения:

  ```
  QString specialValueText() const
  ```

- `setReadOnly()` — если в качестве параметра указано значение `true`, то поле будет доступно только для чтения. Прототип метода:

  ```
  void setReadOnly(bool)
  ```

  Получение значения:

  ```
  bool isReadOnly() const
  ```

- `setKeyboardTracking()` — если в качестве параметра указано значение `false`, то сигналы `valueChanged()` и `textChanged()` не будут генерироваться при вводе значения с клавиатуры. Прототип метода:

  ```
  void setKeyboardTracking(bool)
  ```

Получение значения:

```
bool keyboardTracking() const
```

- `setGroupSeparatorShown()` — если в качестве параметра указано значение `true`, то будут отображаться разделители тысяч. Прототип метода:

  ```
  void setGroupSeparatorShown(bool shown)
  ```

  Получение значения:

  ```
  bool isGroupSeparatorShown() const
  ```

- `setFrame()` — если в качестве параметра указано значение `false`, то поле будет отображаться без рамки. Прототип метода:

  ```
  void setFrame(bool)
  ```

  Получение значения:

  ```
  bool hasFrame() const
  ```

- `stepDown()` — уменьшает значение на одно приращение. Метод является слотом. Прототип метода:

  ```
  void stepDown()
  ```

- `stepUp()` — увеличивает значение на одно приращение. Метод является слотом. Прототип метода:

  ```
  void stepUp()
  ```

- `stepBy()` — увеличивает (при положительном значении) или уменьшает (при отрицательном значении) значение поля на указанное количество приращений. Прототип метода:

  ```
  virtual void stepBy(int steps)
  ```

- `text()` — возвращает текст, содержащийся внутри поля. Прототип метода:

  ```
  QString text() const
  ```

- `clear()` — очищает поле. Метод является слотом. Прототип метода:

  ```
  virtual void clear()
  ```

- `selectAll()` — выделяет все содержимое поля. Метод является слотом. Прототип метода:

  ```
  void selectAll()
  ```

Класс `QAbstractSpinBox` содержит сигнал `editingFinished()`, который генерируется при потере полем фокуса ввода или нажатии клавиши <Enter>.

Классы `QSpinBox` и `QDoubleSpinBox` содержат следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setValue()` — задает значение поля. Метод является слотом. Прототипы метода:

  ```
  void setValue(int val)     // QSpinBox
  void setValue(double val)  // QDoubleSpinBox
  ```

- `value()` — возвращает число, содержащееся в поле. Прототипы метода:

```
int value() const          // QSpinBox
double value() const       // QDoubleSpinBox
```

- `cleanText()` — возвращает число в виде строки. Прототип метода:

```
QString cleanText() const
```

- `setRange()`, `setMinimum()` и `setMaximum()` — задают минимальное и максимальное допустимые значения. Прототипы методов:

```
// QSpinBox
void setRange(int minimum, int maximum)
void setMinimum(int min)
void setMaximum(int max)
// QDoubleSpinBox
void setRange(double minimum, double maximum)
void setMinimum(double min)
void setMaximum(double max)
```

- `setPrefix()` — задает текст, который будет отображаться внутри поля перед значением. Прототип метода:

```
void setPrefix(const QString &prefix)
```

- `setSuffix()` — задает текст, который будет отображаться внутри поля после значения. Прототип метода:

```
void setSuffix(const QString &suffix)
```

- `setSingleStep()` — задает число, которое будет прибавляться или вычитаться из текущего значения поля на каждом шаге. Прототипы метода:

```
void setSingleStep(int val)          // QSpinBox
void setSingleStep(double val)       // QDoubleSpinBox
```

Класс `QDoubleSpinBox` содержит также метод `setDecimals()`, который задает количество цифр после десятичной точки. Прототип метода:

```
void setDecimals(int prec)
```

Классы `QSpinBox` и `QDoubleSpinBox` содержат сигналы `valueChanged(int)` (только в классе `QSpinBox`), `valueChanged(double)` (только в классе `QDoubleSpinBox`) и `textChanged(const QString&)`, которые генерируются при изменении значения внутри поля. Внутри обработчика через параметр доступно новое значение в виде числа или строки в зависимости от сигнала.

6.9. Поля для ввода даты и времени

Для ввода даты и времени предназначены классы `QDateTimeEdit` (поле для ввода даты и времени), `QDateEdit` (поле для ввода даты) и `QTimeEdit` (поле для ввода времени). Поля могут содержать две кнопки, которые позволяют увеличивать и уменьшать значение внутри поля с помощью щелчка мышью. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDateTimeEdit
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDateTimeEdit —
                        — QDateEdit
(QObject, QPaintDevice) — QWidget — QAbstractSpinBox — QDateTimeEdit —
                        — QTimeEdit
```

Форматы конструкторов классов:

```
#include <QDateTimeEdit>
QDateTimeEdit(QTime time, QWidget *parent = nullptr)
QDateTimeEdit(QDate date, QWidget *parent = nullptr)
QDateTimeEdit(const QDateTime &datetime, QWidget *parent = nullptr)
QDateTimeEdit(QWidget *parent = nullptr)
#include <QDateEdit>
QDateEdit(QDate date, QWidget *parent = nullptr)
QDateEdit(QWidget *parent = nullptr)
#include <QTimeEdit>
QTimeEdit(QTime time, QWidget *parent = nullptr)
QTimeEdit(QWidget *parent = nullptr)
```

Класс `QDateTimeEdit` наследует все методы из класса `QAbstractSpinBox` *(см. разд. 6.8)* и дополнительно реализует следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setDateTime()` — устанавливает дату и время. Метод является слотом. Прототип метода:

  ```
  void setDateTime(const QDateTime &dateTime)
  ```

- `setDate()` — устанавливает дату. Метод является слотом. Прототип метода:

  ```
  void setDate(QDate date)
  ```

- `setTime()` — устанавливает время. Метод является слотом. Прототип метода:

  ```
  void setTime(QTime time)
  ```

- `dateTime()` — возвращает экземпляр класса `QDateTime` с датой и временем. Прототип метода:

  ```
  QDateTime dateTime() const
  ```

- `date()` — возвращает экземпляр класса `QDate` с датой. Прототип метода:

  ```
  QDate date() const
  ```

- `time()` — возвращает экземпляр класса `QTime` с временем. Прототип метода:

  ```
  QTime time() const
  ```

- `setDateTimeRange()` — задает диапазон допустимых значений для даты и времени. Прототип метода:

  ```
  void setDateTimeRange(const QDateTime &min, const QDateTime &max)
  ```

- `setMinimumDateTime()` и `setMaximumDateTime()` — задают минимальное и максимальное допустимые значения для даты и времени. Прототипы методов:

```
void setMinimumDateTime(const QDateTime &dt)
void setMaximumDateTime(const QDateTime &dt)
```

- `setDateRange()`, `setMinimumDate()` и `setMaximumDate()` — задают минимальное и максимальное допустимые значения для даты. Прототипы методов:

```
void setDateRange(QDate min, QDate max)
void setMinimumDate(QDate min)
void setMaximumDate(QDate max)
```

- `setTimeRange()`, `setMinimumTime()` и `setMaximumTime()` — задают минимальное и максимальное допустимые значения для времени. Прототипы методов:

```
void setTimeRange(QTime min, QTime max)
void setMinimumTime(QTime min)
void setMaximumTime(QTime max)
```

- `setDisplayFormat()` — задает формат отображения даты и времени. Прототип метода:

```
void setDisplayFormat(const QString &format)
```

  В качестве параметра указывается строка, содержащая специальные символы. Пример указания строки формата:

```
dateTimeEdit->setDisplayFormat("dd.MM.yyyy HH:mm:ss");
```

- `setTimeSpec()` — задает зону времени. В качестве параметра можно указать константы `Qt::LocalTime`, `Qt::UTC` или `Qt::OffsetFromUTC`. Прототип метода:

```
void setTimeSpec(Qt::TimeSpec spec)
```

- `setCalendarPopup()` — если в качестве параметра указано значение `true`, то дату можно будет выбрать с помощью календаря. Прототип метода:

```
void setCalendarPopup(bool enable)
```

- `setSelectedSection()` — выделяет указанную секцию. В качестве параметра можно указать константы `NoSection`, `DaySection`, `MonthSection`, `YearSection`, `HourSection`, `MinuteSection`, `SecondSection`, `MSecSection` или `AmPmSection` из пространства имен `QDateTimeEdit`. Прототип метода:

```
void setSelectedSection(QDateTimeEdit::Section section)
```

- `setCurrentSection()` — делает указанную секцию текущей. Прототип метода:

```
void setCurrentSection(QDateTimeEdit::Section section)
```

- `setCurrentSectionIndex()` — делает секцию с указанным индексом текущей. Прототип метода:

```
void setCurrentSectionIndex(int index)
```

- `currentSection()` — возвращает тип текущей секции. Прототип метода:

```
QDateTimeEdit::Section currentSection() const
```

- `currentSectionIndex()` — возвращает индекс текущей секции. Прототип метода:

  ```
  int currentSectionIndex() const
  ```

- `sectionCount()` — возвращает количество секций внутри поля. Прототип метода:

  ```
  int sectionCount() const
  ```

- `sectionAt()` — возвращает тип секции по указанному индексу. Прототип метода:

  ```
  QDateTimeEdit::Section sectionAt(int index) const
  ```

- `sectionText()` — возвращает текст указанной секции. Прототип метода:

  ```
  QString sectionText(QDateTimeEdit::Section section) const
  ```

При изменении внутри поля значений даты или времени генерируются сигналы `timeChanged(QTime)`, `dateChanged(QDate)` и `dateTimeChanged(const QDateTime&)`. Внутри обработчиков через параметр доступно новое значение.

Классы `QDateEdit` (поле для ввода даты) и `QTimeEdit` (поле для ввода времени) созданы для удобства и отличаются от класса `QDateTimeEdit` только форматом отображаемых данных. Эти классы наследуют методы базовых классов и не добавляют больше никаких своих методов.

6.10. Календарь

Класс `QCalendarWidget` реализует календарь с возможностью выбора даты и перемещения по месяцам с помощью мыши и клавиатуры. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QCalendarWidget
```

Формат конструктора класса `QCalendarWidget`:

```
#include <QCalendarWidget>
QCalendarWidget(QWidget *parent = nullptr)
```

Класс `QCalendarWidget` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setSelectedDate()` — устанавливает указанную дату. Метод является слотом. Прототип метода:

  ```
  void setSelectedDate(QDate date)
  ```

- `selectedDate()` — возвращает экземпляр класса `QDate` с выбранной датой. Прототип метода:

  ```
  QDate selectedDate() const
  ```

- `setDateRange()`, `setMinimumDate()` и `setMaximumDate()` — задают минимальное и максимальное допустимые значения для даты. Метод `setDateRange()` является слотом. Прототипы методов:

  ```
  void setDateRange(QDate min, QDate max)
  void setMinimumDate(QDate date)
  void setMaximumDate(QDate date)
  ```

- `setCurrentPage()` — делает текущей страницу с указанным годом и месяцем. Выбранная дата при этом не изменяется. Метод является слотом. Прототип метода:

```
void setCurrentPage(int year, int month)
```

- `monthShown()` — возвращает месяц (число от `1` до `12`), отображаемый на текущей странице. Прототип метода:

```
int monthShown() const
```

- `yearShown()` — возвращает год, отображаемый на текущей странице. Прототип метода:

```
int yearShown() const
```

- `showSelectedDate()` — отображает страницу с выбранной датой. Выбранная дата при этом не изменяется. Метод является слотом. Прототип метода:

```
void showSelectedDate()
```

- `showToday()` — отображает страницу с сегодняшней датой. Выбранная дата при этом не изменяется. Метод является слотом. Прототип метода:

```
void showToday()
```

- `showPreviousMonth()` — отображает страницу с предыдущим месяцем. Выбранная дата при этом не изменяется. Метод является слотом. Прототип метода:

```
void showPreviousMonth()
```

- `showNextMonth()` — отображает страницу со следующим месяцем. Выбранная дата при этом не изменяется. Метод является слотом. Прототип метода:

```
void showNextMonth()
```

- `showPreviousYear()` — отображает страницу с текущим месяцем в предыдущем году. Выбранная дата при этом не изменяется. Метод является слотом. Прототип метода:

```
void showPreviousYear()
```

- `showNextYear()` — отображает страницу с текущим месяцем в следующем году. Выбранная дата при этом не изменяется. Метод является слотом. Прототип метода:

```
void showNextYear()
```

- `setFirstDayOfWeek()` — задает первый день недели. По умолчанию первый день недели зависит от локали. Прототип метода:

```
void setFirstDayOfWeek(Qt::DayOfWeek dayOfWeek)
```

- `setNavigationBarVisible()` — если в качестве параметра указано значение `false`, то панель навигации выводиться не будет. Метод является слотом. Прототип метода:

```
void setNavigationBarVisible(bool visible)
```

- `setHorizontalHeaderFormat()` — задает формат горизонтального заголовка. Прототип метода:

```
void setHorizontalHeaderFormat(
        QCalendarWidget::HorizontalHeaderFormat format)
```

  В качестве параметра можно указать следующие константы:

  - `QCalendarWidget::NoHorizontalHeader` — заголовок не отображается;
  - `QCalendarWidget::SingleLetterDayNames` — отображается только первая буква из названия дня недели;
  - `QCalendarWidget::ShortDayNames` — отображается сокращенное название дня недели;
  - `QCalendarWidget::LongDayNames` — отображается полное название дня недели;

- `setVerticalHeaderFormat()` — задает формат вертикального заголовка. Прототип метода:

```
void setVerticalHeaderFormat(QCalendarWidget::VerticalHeaderFormat format)
```

  В качестве параметра можно указать следующие константы:

  - `QCalendarWidget::NoVerticalHeader` — заголовок не отображается;
  - `QCalendarWidget::ISOWeekNumbers` — отображается номер недели в году;

- `setGridVisible()` — если в качестве параметра указано значение `true`, то будут отображены линии сетки. Метод является слотом. Прототип метода:

```
void setGridVisible(bool show)
```

- `setSelectionMode()` — задает режим выделения даты. Прототип метода:

```
void setSelectionMode(QCalendarWidget::SelectionMode mode)
```

  В качестве параметра можно указать следующие константы:

  - `QCalendarWidget::NoSelection` — дата не может быть выбрана пользователем;
  - `QCalendarWidget::SingleSelection` — может быть выбрана одна дата;

- `setHeaderTextFormat()` — задает формат ячеек заголовка. Прототип метода:

```
void setHeaderTextFormat(const QTextCharFormat &format)
```

- `setWeekdayTextFormat()` — задает формат ячеек для указанного дня недели. В первом параметре указываются константы `Monday`, `Tuesday`, `Wednesday`, `Thursday`, `Friday`, `Saturday` или `Sunday`. Прототип метода:

```
void setWeekdayTextFormat(Qt::DayOfWeek dayOfWeek,
                          const QTextCharFormat &format)
```

- `setDateTextFormat()` — задает формат ячейки с указанной датой. Прототип метода:

```
void setDateTextFormat(QDate date, const QTextCharFormat &format)
```

Класс QCalendarWidget содержит следующие сигналы:

- activated(QDate) — генерируется при двойном щелчке мышью или нажатии клавиши <Enter>. Внутри обработчика через параметр доступна дата;
- clicked(QDate) — генерируется при щелчке мышью на доступной дате. Внутри обработчика через параметр доступна выбранная дата;
- currentPageChanged(int,int) — генерируется при изменении страницы. Внутри обработчика через первый параметр доступен год, а через второй — месяц;
- selectionChanged() — генерируется при изменении выбранной даты пользователем или из программы.

6.11. Электронный индикатор

Класс QLCDNumber реализует электронный индикатор, в котором цифры и буквы отображаются отдельными сегментами, как на электронных часах или дисплее калькулятора. Индикатор позволяет отображать числа в двоичной, восьмеричной, десятичной и шестнадцатеричной системах счисления. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QLCDNumber
```

Форматы конструктора класса QLCDNumber:

```
#include <QLCDNumber>
QLCDNumber(uint numDigits, QWidget *parent = nullptr)
QLCDNumber(QWidget *parent = nullptr)
```

В параметре numDigits указывается количество отображаемых цифр. Если параметр не указан, то по умолчанию используется значение 5.

Класс QLCDNumber содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- display() — задает новое значение. Метод является слотом. Прототипы метода:

```
void display(int num)
void display(double num)
void display(const QString &s)
```

- checkOverflow() — возвращает значение true, если целое или вещественное число, указанное в параметре, не может быть отображено индикатором. В противном случае возвращает значение false. Прототипы метода:

```
bool checkOverflow(int num) const
bool checkOverflow(double num) const
```

- intValue() — возвращает значение индикатора в виде целого числа. Прототип метода:

```
int intValue() const
```

- `value()` — возвращает значение индикатора в виде вещественного числа. Прототип метода:

  ```
  double value() const
  ```

- `setSegmentStyle()` — задает стиль индикатора. В качестве параметра можно указать константы `Outline`, `Filled` или `Flat` из пространства имен `QLCDNumber`. Прототип метода:

  ```
  void setSegmentStyle(QLCDNumber::SegmentStyle)
  ```

- `setMode()` — задает режим отображения чисел. Прототип метода:

  ```
  void setMode(QLCDNumber::Mode)
  ```

  В качестве параметра можно указать следующие константы:

  - `QLCDNumber::Hex` — шестнадцатеричное значение;
  - `QLCDNumber::Dec` — десятичное значение;
  - `QLCDNumber::Oct` — восьмеричное значение;
  - `QLCDNumber::Bin` — двоичное значение.

  Вместо метода `setMode()` удобнее воспользоваться слотами `setHexMode()`, `setDecMode()`, `setOctMode()` и `setBinMode()`. Прототипы методов:

  ```
  void setHexMode()
  void setDecMode()
  void setOctMode()
  void setBinMode()
  ```

- `setSmallDecimalPoint()` — если в качестве параметра указано значение `true`, то десятичная точка будет отображаться как отдельный элемент (при этом значение выводится более компактно без пробелов до и после точки), а если значение `false`, то десятичная точка будет занимать позицию цифры (значение используется по умолчанию). Метод является слотом. Прототип метода:

  ```
  void setSmallDecimalPoint(bool)
  ```

- `setDigitCount()` — задает количество отображаемых цифр. Если в методе `setSmallDecimalPoint()` указано значение `false`, то десятичная точка считается одной цифрой. Прототип метода:

  ```
  void setDigitCount(int numDigits)
  ```

Класс `QLCDNumber` содержит сигнал `overflow()`, который генерируется при попытке задать значение, которое не может быть отображено индикатором.

6.12. Индикатор процесса

Класс `QProgressBar` реализует индикатор процесса, с помощью которого можно информировать пользователя о текущем состоянии выполнения длительной операции. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QProgressBar
```

Формат конструктора класса `QProgressBar`:

```
#include <QProgressBar>
QProgressBar(QWidget *parent = nullptr)
```

Класс `QProgressBar` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setValue()` — задает новое значение. Метод является слотом. Прототип метода:

  ```
  void setValue(int value)
  ```

- `value()` — возвращает текущее значение индикатора. Прототип метода:

  ```
  int value() const
  ```

- `text()` — возвращает текст, отображаемый на индикаторе или рядом с ним. Прототип метода:

  ```
  virtual QString text() const
  ```

- `setRange()`, `setMinimum()` и `setMaximum()` — задают минимальное и максимальное значения. Если оба значения равны нулю, то внутри индикатора будут постоянно по кругу перемещаться сегменты, показывая ход выполнения процесса с неопределенным количеством шагов. Методы являются слотами. Прототипы методов:

  ```
  void setRange(int minimum, int maximum)
  void setMinimum(int minimum)
  void setMaximum(int maximum)
  ```

- `reset()` — сбрасывает значение индикатора. Метод является слотом. Прототип метода:

  ```
  void reset()
  ```

- `setOrientation()` — задает ориентацию индикатора. В качестве значения указываются константы `Qt::Horizontal` или `Qt::Vertical`. Метод является слотом. Прототип метода:

  ```
  void setOrientation(Qt::Orientation)
  ```

- `setTextVisible()` — если в качестве параметра указано значение `false`, то текст с текущим значением индикатора отображаться не будет. Прототип метода:

  ```
  void setTextVisible(bool visible)
  ```

- `setTextDirection()` — задает направление вывода текста при вертикальной ориентации индикатора. Обратите внимание на то, что при использовании некоторых стилей при вертикальной ориентации текст вообще не отображается. Прототип метода:

  ```
  void setTextDirection(QProgressBar::Direction textDirection)
  ```

  В качестве значения указываются следующие константы:

  - `QProgressBar::TopToBottom` — текст поворачивается на 90 градусов по часовой стрелке;

- `QProgressBar::BottomToTop` — текст поворачивается на 90 градусов против часовой стрелки;

❐ `setInvertedAppearance()` — если в качестве параметра указано значение `true`, то направление увеличения значения будет изменено на противоположное (например, не слева направо, а справа налево при горизонтальной ориентации). Прототип метода:

```
void setInvertedAppearance(bool invert)
```

При изменении значения индикатора генерируется сигнал `valueChanged(int)`. Внутри обработчика через параметр доступно новое значение.

6.13. Шкала с ползунком

Класс `QSlider` реализует шкалу с ползунком, который можно перемещать с помощью клавиатуры или указателя мыши. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QAbstractSlider — QSlider
```

Форматы конструктора класса `QSlider`:

```
#include <QSlider>
QSlider(Qt::Orientation orientation, QWidget *parent = nullptr)
QSlider(QWidget *parent = nullptr)
```

Параметр `orientation` позволяет задать ориентацию шкалы. В качестве значения указываются константы `Qt::Horizontal` или `Qt::Vertical` (значение по умолчанию).

Класс `QSlider` наследует следующие методы из класса `QAbstractSlider` (перечислены только основные методы; полный список смотрите в документации):

❐ `setValue()` — задает новое значение. Метод является слотом. Прототип метода:

```
void setValue(int)
```

❐ `value()` — возвращает установленное положение ползунка в виде числа. Прототип метода:

```
int value() const
```

❐ `setSliderPosition()` — задает текущее положение ползунка. Прототип метода:

```
void setSliderPosition(int)
```

❐ `sliderPosition()` — возвращает текущее положение ползунка в виде числа. Если отслеживание перемещения ползунка включено (по умолчанию), то возвращаемое значение будет совпадать со значением, возвращаемым методом `value()`. Если отслеживание выключено, то при перемещении метод `sliderPosition()` вернет текущее положение, а метод `value()` — положение, которое имел ползунок до перемещения. Прототип метода:

```
int sliderPosition() const
```

- `setRange()`, `setMinimum()` и `setMaximum()` — задают минимальное и максимальное значения. Метод `setRange()` является слотом. Прототипы методов:

```
void setRange(int min, int max)
void setMinimum(int min)
void setMaximum(int max)
```

- `setOrientation()` — задает ориентацию шкалы. В качестве значения указываются константы `Qt::Horizontal` или `Qt::Vertical`. Метод является слотом. Прототип метода:

```
void setOrientation(Qt::Orientation)
```

- `setSingleStep()` — задает значение, на которое сдвинется ползунок при нажатии клавиш со стрелками. Прототип метода:

```
void setSingleStep(int)
```

- `setPageStep()` — задает значение, на которое сдвинется ползунок при нажатии клавиш <PageUp> и <PageDown>, повороте колесика мыши или щелчке мышью на шкале. Прототип метода:

```
void setPageStep(int)
```

- `setInvertedAppearance()` — если в качестве параметра указано значение `true`, то направление увеличения значения будет изменено на противоположное (например, не слева направо, а справа налево при горизонтальной ориентации). Прототип метода:

```
void setInvertedAppearance(bool)
```

- `setInvertedControls()` — если в качестве параметра указано значение `false`, то при изменении направления увеличения значения будет изменено и направление перемещения ползунка при нажатии клавиш <PageUp> и <PageDown>, повороте колесика мыши и нажатии клавиш <↑> и <↓>. Прототип метода:

```
void setInvertedControls(bool)
```

- `setTracking()` — если в качестве параметра указано значение `true`, то отслеживание перемещения ползунка будет включено (значение по умолчанию). При этом сигнал `valueChanged(int)` будет генерироваться постоянно при перемещении ползунка. Если в качестве параметра указано значение `false`, то сигнал `valueChanged(int)` будет сгенерирован только при отпускании ползунка. Прототип метода:

```
void setTracking(bool enable)
```

- `hasTracking()` — возвращает значение `true`, если отслеживание перемещения ползунка включено, и `false` — в противном случае. Прототип метода:

```
bool hasTracking() const
```

Класс `QAbstractSlider` содержит следующие сигналы:

- `actionTriggered(int)` — генерируется, когда производится взаимодействие с ползунком, например при нажатии клавиши <PageUp>. Внутри обработчика

через параметр доступно произведенное действие, которое описывается константами SliderNoAction, SliderSingleStepAdd, SliderSingleStepSub, SliderPageStepAdd, SliderPageStepSub, SliderToMinimum, SliderToMaximum и SliderMove из пространства имен QAbstractSlider;

- rangeChanged(int,int) — генерируется при изменении диапазона значений. Внутри обработчика через первый параметр доступно новое минимальное значение, а через второй параметр — новое максимальное значение;
- sliderPressed() — генерируется при нажатии ползунка;
- sliderMoved(int) — генерируется постоянно при перемещении ползунка. Внутри обработчика через параметр доступно новое положение ползунка;
- sliderReleased() — генерируется при отпускании ранее нажатого ползунка;
- valueChanged(int) — генерируется при изменении значения. Внутри обработчика через параметр доступно новое значение.

Класс QSlider содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setTickPosition() — задает позицию рисок. Прототип метода:

```
void setTickPosition(QSlider::TickPosition position)
```

  В качестве параметра указываются следующие константы:

  - QSlider::NoTicks — без рисок;
  - QSlider::TicksBothSides — риски по обе стороны;
  - QSlider::TicksAbove — риски выводятся сверху;
  - QSlider::TicksBelow — риски выводятся снизу;
  - QSlider::TicksLeft — риски выводятся слева;
  - QSlider::TicksRight — риски выводятся справа;

- setTickInterval() — задает расстояние между рисками. Прототип метода:

```
void setTickInterval(int)
```

6.14. Класс *QDial*

Класс QDial реализует круглую шкалу с ползунком круглой или треугольной формы (вид зависит от используемого стиля), который можно перемещать по кругу с помощью клавиатуры или указателя мыши. Компонент напоминает регулятор, используемый в различных устройствах для изменения или отображения каких-либо настроек. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSlider — QDial
```

Формат конструктора класса QDial:

```
#include <QDial>
QDial(QWidget *parent = nullptr)
```

Класс `QDial` наследует все методы и сигналы из класса `QAbstractSlider` *(см. разд. 6.13)* и содержит несколько дополнительных методов (перечислены только основные методы; полный список смотрите в документации):

- `setNotchesVisible()` — если в качестве параметра указано значение `true`, то риски будут отображены. По умолчанию риски не выводятся. Метод является слотом. Прототип метода:

  ```
  void setNotchesVisible(bool visible)
  ```

- `setNotchTarget()` — задает рекомендуемое количество пикселов между рисками. В качестве параметра указывается вещественное число. Прототип метода:

  ```
  void setNotchTarget(double target)
  ```

- `setWrapping()` — если в качестве параметра указано значение `true`, то начало шкалы будет совпадать с ее концом. По умолчанию между началом шкалы и концом расположено пустое пространство. Метод является слотом. Прототип метода:

  ```
  void setWrapping(bool on)
  ```

6.15. Полоса прокрутки

Класс `QScrollBar` реализует горизонтальную и вертикальную полосы прокрутки. Изменить значение можно с помощью нажатия кнопок, расположенных по краям полосы, щелчка мышью на полосе, путем перемещения ползунка, нажатия клавиш клавиатуры, а также выбрав соответствующий пункт из контекстного меню. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QAbstractSlider — QScrollBar
```

Форматы конструктора класса `QScrollBar`:

```
#include <QScrollBar>
QScrollBar(Qt::Orientation orientation, QWidget *parent = nullptr)
QScrollBar(QWidget *parent = nullptr)
```

Параметр `orientation` позволяет задать ориентацию полосы прокрутки. В качестве значения указываются константы `Qt::Horizontal` или `Qt::Vertical` (значение по умолчанию).

Класс `QScrollBar` наследует все методы и сигналы из класса `QAbstractSlider` *(см. разд. 6.13)* и не содержит дополнительных методов.

> ***Примечание***
>
> Полоса прокрутки редко используется отдельно. Гораздо удобнее воспользоваться областью с полосами прокрутки, которую реализует класс `QScrollArea` *(см. разд. 5.12)*.

ГЛАВА 7

Списки и таблицы

В Qt имеется широкий выбор компонентов, позволяющих отображать как одномерный список строк (в свернутом или развернутом состоянии), так и табличные данные. Кроме того, можно отобразить данные, которые имеют очень сложную структуру, например иерархическую. Благодаря поддержке концепции «модель/представление», позволяющей отделить данные от их отображения, одни и те же данные можно отображать сразу в нескольких компонентах без их дублирования.

7.1. Раскрывающийся список

Класс `QComboBox` реализует раскрывающийся список с возможностью выбора одного пункта. При щелчке мышью на поле появляется список возможных вариантов, а при выборе пункта список сворачивается. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QComboBox
```

Формат конструктора класса `QComboBox`:

```
#include <QComboBox>
QComboBox(QWidget *parent = nullptr)
```

7.1.1. Добавление, изменение и удаление элементов

Для добавления, изменения, удаления и получения значения элементов предназначены следующие методы из класса `QComboBox`:

- `addItem()` — добавляет один элемент в конец списка. Прототипы метода:

  ```
  void addItem(const QString &text,
               const QVariant &userData = QVariant())
  void addItem(const QIcon &icon, const QString &text,
               const QVariant &userData = QVariant())
  ```

  В параметре `text` задается текст элемента списка, а в параметре `icon` — значок, который будет отображен перед текстом. Необязательный параметр `userData`

позволяет сохранить пользовательские данные, например индекс в таблице базы данных;

- `addItems()` — добавляет несколько элементов в конец списка. Прототип метода:

```
void addItems(const QStringList &texts)
```

- `insertItem()` — вставляет один элемент в указанную позицию списка. Все остальные элементы сдвигаются в конец списка. Прототипы метода:

```
void insertItem(int index, const QString &text,
                const QVariant &userData = QVariant())
void insertItem(int index, const QIcon &icon,
                const QString &text,
                const QVariant &userData = QVariant())
```

- `insertItems()` — вставляет несколько элементов в указанную позицию списка. Все остальные элементы сдвигаются в конец списка. Прототип метода:

```
void insertItems(int index, const QStringList &list)
```

- `insertSeparator()` — вставляет разделительную линию в указанную позицию. Прототип метода:

```
void insertSeparator(int index)
```

- `setItemText()` — изменяет текст элемента с указанным индексом. Прототип метода:

```
void setItemText(int index, const QString &text)
```

- `setItemIcon()` — изменяет значок элемента с указанным индексом. Прототип метода:

```
void setItemIcon(int index, const QIcon &icon)
```

- `setItemData()` — изменяет данные для элемента с указанным индексом. Необязательный параметр `role` позволяет указать роль, для которой задаются данные. Например, если указать константу `Qt::ToolTipRole`, то данные задают текст вплывающей подсказки, которая будет отображена при наведении указателя мыши на элемент. По умолчанию изменяются пользовательские данные. Прототип метода:

```
void setItemData(int index, const QVariant &value, int role = Qt::UserRole)
```

- `setCurrentIndex()` — делает элемент с указанным индексом текущим. Метод является слотом. Прототип метода:

```
void setCurrentIndex(int index)
```

- `currentIndex()` — возвращает индекс текущего элемента. Прототип метода:

```
int currentIndex() const
```

- `currentText()` — возвращает текст текущего элемента. Прототип метода:

```
QString currentText() const
```

- `itemText()` — возвращает текст элемента с указанным индексом. Прототип метода:

```
QString itemText(int index) const
```

- `itemData()` — возвращает данные, сохраненные в роли `role` элемента с индексом `index`. Прототип метода:

```
QVariant itemData(int index, int role = Qt::UserRole) const
```

- `currentData()` — возвращает данные, сохраненные в роли `role` текущего элемента. Прототип метода:

```
QVariant currentData(int role = Qt::UserRole) const
```

- `count()` — возвращает общее количество элементов списка. Прототип метода:

```
int count() const
```

- `removeItem()` — удаляет элемент с указанным индексом. Прототип метода:

```
void removeItem(int index)
```

- `clear()` — удаляет все элементы списка. Метод является слотом. Прототип метода:

```
void clear()
```

7.1.2. Изменение настроек

Управлять настройками раскрывающегося списка позволяют следующие методы:

- `setEditable()` — если в качестве параметра указано значение `true`, то пользователь сможет добавлять новые элементы в список путем ввода текста в поле и последующего нажатия клавиши <Enter>. Прототип метода:

```
void setEditable(bool editable)
```

- `setInsertPolicy()` — задает режим добавления нового элемента пользователем. Прототип метода:

```
void setInsertPolicy(QComboBox::InsertPolicy policy)
```

  В качестве параметра указываются следующие константы:

  - `QComboBox::NoInsert` — элемент не будет добавлен;
  - `QComboBox::InsertAtTop` — элемент вставляется в начало списка;
  - `QComboBox::InsertAtCurrent` — будет изменен текст текущего элемента;
  - `QComboBox::InsertAtBottom` — элемент вставляется в конец списка;
  - `QComboBox::InsertAfterCurrent` — элемент вставляется после текущего элемента;
  - `QComboBox::InsertBeforeCurrent` — элемент вставляется перед текущим элементом;

- `QComboBox::InsertAlphabetically` — при вставке учитывается алфавитный порядок следования элементов;

- `setEditText()` — вставляет текст в поле редактирования. Метод является слотом. Прототип метода:

```
void setEditText(const QString &text)
```

- `clearEditText()` — удаляет текст из поля редактирования. Метод является слотом. Прототип метода:

```
void clearEditText()
```

- `setCompleter()` — позволяет предлагать возможные варианты значений, начинающиеся с введенных пользователем символов. Прототип метода:

```
#include <QCompleter>
void setCompleter(QCompleter *completer)
```

Пример:

```
QComboBox *comboBox = new QComboBox();
comboBox->setEditable(true);
comboBox->setInsertPolicy(QComboBox::InsertAtTop);
QStringList list;
list << "кадр" << "каменный" << "камень" << "камера";
QCompleter *completer = new QCompleter(list, &window);
completer->setCaseSensitivity(Qt::CaseInsensitive);
comboBox->setCompleter(completer);
```

- `setValidator()` — устанавливает контроль ввода. В качестве значения указывается экземпляр класса, наследующего класс `QValidator` *(см. разд. 6.5.3)*. Прототип метода:

```
void setValidator(const QValidator *validator)
```

- `setDuplicatesEnabled()` — если в качестве параметра указано значение `true`, то пользователь может добавить элемент с повторяющимся текстом. По умолчанию повторы запрещены. Прототип метода:

```
void setDuplicatesEnabled(bool enable)
```

- `setMaxCount()` — задает максимальное количество элементов в списке. Если до вызова метода количество элементов превышало указанное количество, то лишние элементы будут удалены. Прототип метода:

```
void setMaxCount(int max)
```

- `setMaxVisibleItems()` — задает максимальное количество видимых элементов в раскрывающемся списке. Прототип метода:

```
void setMaxVisibleItems(int maxItems)
```

- `setMinimumContentsLength()` — задает минимальное количество отображаемых символов. Прототип метода:

```
void setMinimumContentsLength(int characters)
```

- `setSizeAdjustPolicy()` — устанавливает режим изменения ширины при изменении содержимого. Прототип метода:

  ```
  void setSizeAdjustPolicy(QComboBox::SizeAdjustPolicy policy)
  ```

  В качестве параметра указываются следующие константы:

  - `QComboBox::AdjustToContents` — ширина будет соответствовать содержимому;
  - `QComboBox::AdjustToContentsOnFirstShow` — ширина будет соответствовать ширине, используемой при первом отображении списка;
  - `QComboBox::AdjustToMinimumContentsLengthWithIcon` — используется значение минимальной ширины, которое установлено с помощью метода `setMinimumContentsLength()`, плюс ширина значка;

- `setFrame()` — если в качестве параметра указано значение `false`, то поле будет отображаться без рамки. Прототип метода:

  ```
  void setFrame(bool)
  ```

- `setIconSize()` — задает максимальный размер значков. Прототип метода:

  ```
  void setIconSize(const QSize &size)
  ```

- `showPopup()` — отображает список. Прототип метода:

  ```
  virtual void showPopup()
  ```

- `hidePopup()` — скрывает список. Прототип метода:

  ```
  virtual void hidePopup()
  ```

7.1.3. Поиск элемента внутри списка

Произвести поиск элемента внутри списка позволяют методы `findText()` (поиск в тексте элемента) и `findData()` (поиск данных в указанной роли). Методы возвращают индекс найденного элемента или значение `-1`, если элемент не найден. Прототипы методов:

```
int findText(const QString &text,
   Qt::MatchFlags flags = Qt::MatchExactly|Qt::MatchCaseSensitive) const
int findData(const QVariant &data, int role = Qt::UserRole,
   Qt::MatchFlags flags = static_cast<Qt::MatchFlags>(Qt::MatchExactly|
   Qt::MatchCaseSensitive)) const
```

Параметр `flags` задает режим поиска. В качестве значения можно указать комбинацию (через оператор `|`) следующих констант:

- `Qt::MatchExactly` — поиск полного соответствия;
- `Qt::MatchFixedString` — поиск полного соответствия внутри строки, выполняемый по умолчанию без учета регистра символов;
- `Qt::MatchContains` — поиск совпадения с любой частью;
- `Qt::MatchStartsWith` — совпадение с началом;

- `Qt::MatchEndsWith` — совпадение с концом;
- `Qt::MatchRegularExpression` — поиск с помощью регулярного выражения;
- `Qt::MatchWildcard` — используются подстановочные знаки;
- `Qt::MatchCaseSensitive` — поиск с учетом регистра символов;
- `Qt::MatchWrap` — поиск по кругу;
- `Qt::MatchRecursive` — просмотр всей иерархии.

7.1.4. Сигналы

Класс `QComboBox` содержит следующие сигналы:

- `activated(int)` — генерируется при выборе пункта в списке (даже если индекс не изменился) пользователем. Внутри обработчика доступен индекс элемента;
- `textActivated(const QString&)` — генерируется при выборе пункта в списке (даже если индекс не изменился) пользователем. Внутри обработчика доступен текст элемента;
- `currentIndexChanged(int)` — генерируется при изменении текущего индекса. Внутри обработчика доступен индекс (значение `-1`, если список пуст) элемента;
- `currentTextChanged(const QString&)` — генерируется при изменении текущего индекса. Внутри обработчика доступен текст элемента;
- `editTextChanged(const QString&)` — генерируется при изменении текста в поле. Внутри обработчика через параметр доступен новый текст;
- `highlighted(int)` — генерируется при наведении указателя мыши на пункт в списке. Внутри обработчика доступен индекс элемента;
- `textHighlighted(const QString&)` — генерируется при наведении указателя мыши на пункт в списке. Внутри обработчика доступен текст элемента.

7.2. Список для выбора шрифта

Класс `QFontComboBox` реализует раскрывающийся список с названиями шрифтов. Шрифт можно выбрать из списка или ввести название в поле, при этом будут отображаться названия, начинающиеся с введенных букв. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QComboBox — QFontComboBox
```

Формат конструктора класса `QFontComboBox`:

```
#include <QFontComboBox>
QFontComboBox(QWidget *parent = nullptr)
```

Класс `QFontComboBox` наследует все методы и сигналы из класса `QComboBox` *(см. разд. 7.1)* и содержит несколько дополнительных методов:

- `setCurrentFont()` — делает текущим элемент, соответствующий указанному шрифту. Метод является слотом. Прототип метода:

  ```
  void setCurrentFont(const QFont &f)
  ```

Пример:

```
comboBox->setCurrentFont(QFont("Verdana"));
```

- `currentFont()` — возвращает экземпляр класса `QFont` с выбранным шрифтом. Прототип метода:

  ```
  QFont currentFont() const
  ```

  Пример вывода названия шрифта:

  ```
  qDebug() << comboBox->currentFont().family();
  ```

- `setFontFilters()` — ограничивает список указанными типами шрифтов. Прототип метода:

  ```
  void setFontFilters(QFontComboBox::FontFilters filters)
  ```

  В качестве параметра указывается комбинация следующих констант:

  - `QFontComboBox::AllFonts` — все типы шрифтов;
  - `QFontComboBox::ScalableFonts` — масштабируемые шрифты;
  - `QFontComboBox::NonScalableFonts` — немасштабируемые шрифты;
  - `QFontComboBox::MonospacedFonts` — моноширинные шрифты;
  - `QFontComboBox::ProportionalFonts` — пропорциональные шрифты.

Класс `QFontComboBox` содержит сигнал `currentFontChanged(const QFont&)`, который генерируется при изменении текущего шрифта. Внутри обработчика доступен экземпляр класса `QFont` с текущим шрифтом.

7.3. Роли элементов

Каждый элемент списка содержит данные, распределенные по ролям. С помощью этих данных можно указать текст элемента, каким шрифтом и цветом отображается текст, задать текст всплывающей подсказки и многое другое. Перечислим роли элементов:

- `Qt::DisplayRole` — `0` — отображаемые данные (обычно текст);
- `Qt::DecorationRole` — `1` — изображение (обычно значок);
- `Qt::EditRole` — `2` — данные в виде, удобном для редактирования;
- `Qt::ToolTipRole` — `3` — текст всплывающей подсказки;
- `Qt::StatusTipRole` — `4` — текст для строки состояния;
- `Qt::WhatsThisRole` — `5` — текст для справки;
- `Qt::FontRole` — `6` — шрифт элемента. Указывается экземпляр класса `QFont`;
- `Qt::TextAlignmentRole` — `7` — выравнивание текста внутри элемента;
- `Qt::BackgroundRole` — `8` — фон элемента. Указывается экземпляр класса `QBrush`;
- `Qt::ForegroundRole` — `9` — цвет текста. Указывается экземпляр класса `QBrush`;

- `Qt::CheckStateRole` — `10` — статус флажка. Могут быть указаны следующие константы:
  - `Qt::Unchecked` — `0` — флажок сброшен;
  - `Qt::PartiallyChecked` — `1` — флажок частично установлен;
  - `Qt::Checked` — `2` — флажок установлен;
- `Qt::AccessibleTextRole` — `11` — текст для плагинов;
- `Qt::AccessibleDescriptionRole` — `12` — описание элемента;
- `Qt::UserRole` — `32` — любые пользовательские данные, например индекс элемента в базе данных. Пример:

```
comboBox->setItemData(0, QVariant(50), Qt::UserRole);
comboBox->setItemData(0, "Это текст всплывающей подсказки",
                      Qt::ToolTipRole);
```

7.4. Модели

Для отображения данных в виде списков и таблиц применяется концепция «модель/представление», позволяющая отделить данные от их представления и избежать дублирования данных. В основе концепции лежат следующие составляющие:

- *модель* — является «оберткой» над данными. Позволяет добавлять, изменять и удалять данные, а также содержит методы для чтения данных и управления ими;
- *представление* — предназначено для отображения элементов модели. В нескольких представлениях можно установить одну модель;
- *модель выделения* — позволяет управлять выделением. Если одна модель выделения установлена сразу в нескольких представлениях, то выделение элемента в одном представлении приведет к выделению соответствующего элемента в другом представлении;
- *промежуточная модель* — является прослойкой между моделью и представлением. Позволяет производить сортировку и фильтрацию данных на основе базовой модели без изменения порядка следования элементов в базовой модели;
- *делегат* — выполняет рисование каждого элемента в отдельности, а также позволяет произвести редактирование данных. В своих программах вы можете наследовать стандартные классы делегатов и полностью управлять отображением данных и их редактированием. За подробной информацией по классам делегатов обращайтесь к документации.

7.4.1. Доступ к данным внутри модели

Доступ к данным внутри модели реализуется с помощью класса `QModelIndex`. Формат конструктора класса:

```
#include <QModelIndex>
QModelIndex()
```

Наиболее часто экземпляр класса `QModelIndex` создается с помощью метода `index()` из класса модели или метода `currentIndex()` из класса `QAbstractItemView`.

Класс `QModelIndex` содержит следующие методы:

- `isValid()` — возвращает значение `true`, если объект является валидным, и `false` — в противном случае. Прототип метода:

  ```
  bool isValid() const
  ```

- `data()` — возвращает данные, хранящиеся в указанной роли. Прототип метода:

  ```
  QVariant data(int role = Qt::DisplayRole) const
  ```

- `flags()` — содержит свойства элемента. Прототип метода:

  ```
  Qt::ItemFlags flags() const
  ```

  Возвращает комбинацию следующих констант:

  - `Qt::NoItemFlags` — `0` — свойства не установлены;
  - `Qt::ItemIsSelectable` — `1` — можно выделить;
  - `Qt::ItemIsEditable` — `2` — можно редактировать;
  - `Qt::ItemIsDragEnabled` — `4` — можно перетаскивать;
  - `Qt::ItemIsDropEnabled` — `8` — можно сбрасывать перетаскиваемые данные;
  - `Qt::ItemIsUserCheckable` — `16` — может быть включен и выключен;
  - `Qt::ItemIsEnabled` — `32` — пользователь может взаимодействовать с элементом;
  - `Qt::ItemIsAutoTristate` — `64`;
  - `Qt::ItemNeverHasChildren` — `128`;
  - `Qt::ItemIsUserTristate` — `256`;

- `row()` — возвращает индекс строки. Прототип метода:

  ```
  int row() const
  ```

- `column()` — возвращает индекс столбца. Прототип метода:

  ```
  int column() const
  ```

- `parent()` — возвращает индекс элемента (экземпляр класса `QModelIndex`), расположенного на один уровень выше по иерархии. Если элемента нет, то возвращается невалидный экземпляр класса `QModelIndex`. Прототип метода:

  ```
  QModelIndex parent() const
  ```

- `sibling()` — возвращает индекс элемента (экземпляр класса `QModelIndex`), расположенного на том же уровне вложенности на указанной позиции. Если элемента нет, то возвращается невалидный экземпляр класса `QModelIndex`. Прототип метода:

  ```
  QModelIndex sibling(int row, int column) const
  ```

- `model()` — возвращает указатель на модель. Прототип метода:

```
const QAbstractItemModel *model() const
```

Следует учитывать, что модель может измениться и экземпляр класса `QModelIndex` будет ссылаться на несуществующий уже элемент. Если необходимо сохранить ссылку на элемент, то следует воспользоваться классом `QPersistentModelIndex`, который содержит те же самые методы, но обеспечивает валидность ссылки.

7.4.2. Класс *QStringListModel*

Класс `QStringListModel` реализует одномерную модель, содержащую список строк. Модель можно отобразить с помощью классов `QListView`, `QComboBox` и др., передав в метод `setModel()`. Иерархия наследования:

```
QObject — QAbstractItemModel — QAbstractListModel — QStringListModel
```

Форматы конструктора класса `QStringListModel`:

```
#include <QStringListModel>
QStringListModel(const QStringList &strings, QObject *parent = nullptr)
QStringListModel(QObject *parent = nullptr)
```

Класс `QStringListModel` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setStringList()` — устанавливает список строк. Прототип метода:

```
void setStringList(const QStringList &strings)
```

- `stringList()` — возвращает список строк, хранимых в модели. Прототип метода:

```
QStringList stringList() const
```

- `insertRows()` — вставляет указанное количество элементов в позицию, заданную первым параметром. Остальные элементы сдвигаются в конец списка. Метод возвращает значение `true`, если операция успешно выполнена. Прототип метода:

```
virtual bool insertRows(int row, int count,
                const QModelIndex &parent = QModelIndex())
```

- `removeRows()` — удаляет указанное количество элементов, начиная с позиции, заданной первым параметром. Метод возвращает значение `true`, если операция успешно выполнена. Прототип метода:

```
virtual bool removeRows(int row, int count,
                const QModelIndex &parent = QModelIndex())
```

- `setData()` — задает значение для роли `role` элемента, на который указывает индекс `index`. Метод возвращает значение `true`, если операция успешно выполнена. Прототип метода:

```
virtual bool setData(const QModelIndex &index,
                const QVariant &value, int role = Qt::EditRole)
```

- `data()` — возвращает данные, хранимые в указанной роли элемента, на который ссылается индекс `index`. Прототип метода:

```
virtual QVariant data(const QModelIndex &index,
                      int role = Qt::DisplayRole) const
```

- `rowCount()` — возвращает количество строк в модели. Прототип метода:

```
virtual int rowCount(const QModelIndex &parent = QModelIndex()) const
```

- `sort()` — производит сортировку. Если во втором параметре указана константа `Qt::AscendingOrder`, то сортировка производится в прямом порядке, а если `Qt::DescendingOrder`, то в обратном. Прототип метода:

```
virtual void sort(int column, Qt::SortOrder order = Qt::AscendingOrder)
```

Класс `QStringListModel` наследует метод `index()` из класса `QAbstractListModel`, который возвращает индекс (экземпляр класса `QModelIndex`) внутри модели. Прототип метода:

```
virtual QModelIndex index(int row, int column = 0,
                    const QModelIndex &parent = QModelIndex()) const
```

7.4.3. Класс *QStandardItemModel*

Класс `QStandardItemModel` реализует двумерную (таблица) и иерархическую модели. Каждый элемент такой модели представлен классом `QStandardItem`. Модель можно отобразить с помощью классов `QTableView`, `QTreeView` и др., передав в метод `setModel()`. Иерархия наследования:

```
QObject — QAbstractItemModel — QStandardItemModel
```

Форматы конструктора класса `QStandardItemModel`:

```
#include <QStandardItemModel>
QStandardItemModel(int rows, int columns, QObject *parent = nullptr)
QStandardItemModel(QObject *parent = nullptr)
```

Класс `QStandardItemModel` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setRowCount()` — задает количество строк. Прототип метода:

```
void setRowCount(int rows)
```

- `setColumnCount()` — задает количество столбцов. Прототип метода:

```
void setColumnCount(int columns)
```

- `rowCount()` — возвращает количество строк. Прототип метода:

```
virtual int rowCount(const QModelIndex &parent = QModelIndex()) const
```

- `columnCount()` — возвращает количество столбцов. Прототип метода:

```
virtual int columnCount(const QModelIndex &parent = QModelIndex()) const
```

- `setItem()` — устанавливает элемент в указанную ячейку. Прототипы метода:

```
void setItem(int row, int column, QStandardItem *item)
void setItem(int row, QStandardItem *item)
```

Пример заполнения таблицы:

```
QTableView *view = new QTableView();
QStandardItemModel *model = new QStandardItemModel(&window);
model->setRowCount(3);
model->setColumnCount(4);
for (int row = 0, r = model->rowCount(); row < r; ++row) {
   for (int col = 0, c = model->columnCount(); col < c; ++col) {
      QStandardItem *item = new QStandardItem(
                     QString("%1, %2").arg(row).arg(col));
      model->setItem(row, col, item);
   }
}
view->setModel(model);
```

- `appendRow()` — добавляет одну строку в конец модели. Прототипы метода:

```
void appendRow(const QList<QStandardItem *> &items)
void appendRow(QStandardItem *item)
```

- `appendColumn()` — добавляет один столбец в конец модели. Прототип метода:

```
void appendColumn(const QList<QStandardItem *> &items)
```

- `insertRow()` — добавляет одну строку в указанную позицию модели. Прототипы метода:

```
void insertRow(int row, const QList<QStandardItem *> &items)
void insertRow(int row, QStandardItem *item)
bool insertRow(int row, const QModelIndex &parent = QModelIndex())
```

- `insertRows()` — добавляет несколько строк в указанную позицию модели. Метод возвращает значение `true`, если операция успешно выполнена. Прототип метода:

```
virtual bool insertRows(int row, int count,
                  const QModelIndex &parent = QModelIndex())
```

- `insertColumn()` — добавляет один столбец в указанную позицию модели. Прототипы метода:

```
void insertColumn(int column, const QList<QStandardItem *> &items)
bool insertColumn(int column, const QModelIndex &parent = QModelIndex())
```

- `insertColumns()` — добавляет несколько столбцов в указанную позицию. Метод возвращает значение `true`, если операция успешно выполнена. Прототип метода:

```
virtual bool insertColumns(int column, int count,
                  const QModelIndex &parent = QModelIndex())
```

- `removeRows()` — удаляет указанное количество строк, начиная со строки, имеющей индекс `row`. Метод возвращает значение `true`, если операция успешно выполнена. Прототип метода:

```
virtual bool removeRows(int row, int count,
                  const QModelIndex &parent = QModelIndex())
```

- `removeColumns()` — удаляет указанное количество столбцов, начиная со столбца, имеющего индекс `column`. Метод возвращает значение `true`, если операция успешно выполнена. Прототип метода:

```
virtual bool removeColumns(int column, int count,
                  const QModelIndex &parent = QModelIndex())
```

- `takeItem()` — удаляет указанный элемент из модели и возвращает его. Прототип метода:

```
QStandardItem *takeItem(int row, int column = 0)
```

- `takeRow()` — удаляет указанную строку из модели и возвращает ее. Прототип метода:

```
QList<QStandardItem *> takeRow(int row)
```

- `takeColumn()` — удаляет указанный столбец из модели и возвращает его. Прототип метода:

```
QList<QStandardItem *> takeColumn(int column)
```

- `clear()` — удаляет все элементы из модели. Прототип метода:

```
void clear()
```

- `item()` — возвращает указатель на элемент, расположенный в указанной ячейке. Прототип метода:

```
QStandardItem *item(int row, int column = 0) const
```

- `invisibleRootItem()` — возвращает указатель на невидимый корневой элемент модели. Прототип метода:

```
QStandardItem *invisibleRootItem() const
```

- `itemFromIndex()` — возвращает указатель на элемент, на который ссылается индекс `index`. Прототип метода:

```
QStandardItem *itemFromIndex(const QModelIndex &index) const
```

- `index()` — возвращает индекс элемента (экземпляр класса `QModelIndex`), расположенного в указанной ячейке. Прототип метода:

```
virtual QModelIndex index(int row, int column,
              const QModelIndex &parent = QModelIndex()) const
```

- `indexFromItem()` — возвращает индекс элемента (экземпляр класса `QModelIndex`), указатель на который передан в качестве параметра. Прототип метода:

```
QModelIndex indexFromItem(const QStandardItem *item) const
```

- `setData()` — задает значение для роли `role` элемента, на который указывает индекс `index`. Метод возвращает значение `true`, если операция успешно выполнена. Прототип метода:

```
virtual bool setData(const QModelIndex &index,
                const QVariant &value, int role = Qt::EditRole)
```

- `data()` — возвращает данные, хранимые в указанной роли элемента, на который ссылается индекс `index`. Прототип метода:

```
virtual QVariant data(const QModelIndex &index,
                      int role = Qt::DisplayRole) const
```

- `setHorizontalHeaderLabels()` — задает заголовки столбцов. В качестве параметра указывается список строк. Прототип метода:

```
void setHorizontalHeaderLabels(const QStringList &labels)
```

- `setVerticalHeaderLabels()` — задает заголовки строк. В качестве параметра указывается список строк. Прототип метода:

```
void setVerticalHeaderLabels(const QStringList &labels)
```

- `setHorizontalHeaderItem()` — задает заголовок столбца. Прототип метода:

```
void setHorizontalHeaderItem(int column, QStandardItem *item)
```

- `setVerticalHeaderItem()` — задает заголовок строки. Прототип метода:

```
void setVerticalHeaderItem(int row, QStandardItem *item)
```

- `horizontalHeaderItem()` — возвращает указатель на заголовок столбца. Прототип метода:

```
QStandardItem *horizontalHeaderItem(int column) const
```

- `verticalHeaderItem()` — возвращает указатель на заголовок строки. Прототип метода:

```
QStandardItem *verticalHeaderItem(int row) const
```

- `setHeaderData()` — задает данные для указанной роли заголовка. В первом параметре указывается индекс строки или столбца, а во втором параметре — ориентация (константы `Qt::Horizontal` или `Qt::Vertical`). Если параметр `role` не указан, то используется значение `EditRole`. Метод возвращает значение `true`, если операция успешно выполнена. Прототип метода:

```
virtual bool setHeaderData(int section,
        Qt::Orientation orientation, const QVariant &value,
        int role = Qt::EditRole)
```

- `headerData()` — возвращает данные, хранящиеся в указанной роли заголовка. В первом параметре указывается индекс строки или столбца, а во втором параметре — ориентация. Если параметр `role` не указан, то используется значение `DisplayRole`. Прототип метода:

```
virtual QVariant headerData(int section,
                        Qt::Orientation orientation,
                        int role = Qt::DisplayRole) const
```

- `findItems()` — производит поиск элемента внутри модели в указанном в параметре `column` столбце. Допустимые значения параметра `flags` мы рассматривали в *разд. 7.1.3*. В качестве значения метод возвращает список элементов или пустой список. Прототип метода:

```
QList<QStandardItem *> findItems(const QString &text,
                        Qt::MatchFlags flags = Qt::MatchExactly,
                        int column = 0) const
```

- `sort()` — производит сортировку. Если во втором параметре указана константа `Qt::AscendingOrder`, то сортировка производится в прямом порядке, а если `Qt::DescendingOrder`, то в обратном. Прототип метода:

```
virtual void sort(int column, Qt::SortOrder order = Qt::AscendingOrder)
```

- `setSortRole()` — задает роль *(см. разд. 7.3)*, по которой производится сортировка. Прототип метода:

```
void setSortRole(int role)
```

- `parent()` — возвращает индекс (экземпляр класса `QModelIndex`) родительского элемента. В качестве параметра указывается индекс (экземпляр класса `QModelIndex`) элемента-потомка. Прототип метода:

```
virtual QModelIndex parent(const QModelIndex &child) const
```

- `hasChildren()` — возвращает значение `true`, если существует элемент, расположенный на один уровень ниже по иерархии, и `false` — в противном случае. Прототип метода:

```
virtual bool hasChildren(const QModelIndex &parent = QModelIndex()) const
```

При изменении значения элемента генерируется сигнал `itemChanged(QStandardItem *)`. Внутри обработчика через параметр доступен указатель на элемент.

7.4.4. Класс *QStandardItem*

Каждый элемент модели `QStandardItemModel` представлен классом `QStandardItem`. Этот класс не только описывает элемент, но и позволяет создавать вложенные структуры. Форматы конструктора класса:

```
QStandardItem()
QStandardItem(const QString &text)
QStandardItem(const QIcon &icon, const QString &text)
QStandardItem(int rows, int columns = 1)
```

Класс `QStandardItem` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setRowCount()` — задает количество дочерних строк. Прототип метода:

```
void setRowCount(int rows)
```

- `setColumnCount()` — задает количество дочерних столбцов. Прототип метода:

```
void setColumnCount(int columns)
```

- `rowCount()` — возвращает количество дочерних строк. Прототип метода:

```
int rowCount() const
```

- `columnCount()` — возвращает количество дочерних столбцов. Прототип метода:

```
int columnCount() const
```

- `row()` — возвращает индекс строки в дочерней таблице родительского элемента или значение `-1`, если элемент не содержит родителя. Прототип метода:

```
int row() const
```

- `column()` — возвращает индекс столбца в дочерней таблице родительского элемента или значение `-1`, если элемент не содержит родителя. Прототип метода:

```
int column() const
```

- `setChild()` — устанавливает элемент в указанную ячейку дочерней таблицы. Прототипы метода:

```
void setChild(int row, int column, QStandardItem *item)
void setChild(int row, QStandardItem *item)
```

Пример создания иерархии:

```
QStandardItem *parent = new QStandardItem(3, 4);
parent->setText("Элемент-родитель");
for (int row = 0; row < 3; ++row) {
   for (int col = 0; col < 4; ++col) {
      QStandardItem *item = new QStandardItem(
                     QString("%1, %2").arg(row).arg(col));
      parent->setChild(row, col, item);
   }
}
model->appendRow(parent);
```

- `appendRow()` — добавляет одну строку в конец дочерней таблицы. Прототипы метода:

```
void appendRow(const QList<QStandardItem *> &items)
void appendRow(QStandardItem *item)
```

- `appendRows()` — добавляет несколько строк в конец дочерней таблицы. Прототип метода:

```
void appendRows(const QList<QStandardItem *> &items)
```

- `appendColumn()` — добавляет один столбец в конец дочерней таблицы. Прототип метода:

```
void appendColumn(const QList<QStandardItem *> &items)
```

- `insertRow()` — добавляет одну строку в указанную позицию дочерней таблицы. Прототипы метода:

```
void insertRow(int row, const QList<QStandardItem *> &items)
void insertRow(int row, QStandardItem *item)
```

- `insertRows()` — добавляет несколько строк в указанную позицию дочерней таблицы. Прототипы метода:

```
void insertRows(int row, const QList<QStandardItem *> &items)
void insertRows(int row, int count)
```

- `insertColumn()` — добавляет один столбец в указанную позицию дочерней таблицы. Прототип метода:

```
void insertColumn(int column, const QList<QStandardItem *> &items)
```

- `insertColumns()` — добавляет несколько столбцов в указанную позицию. Прототип метода:

```
void insertColumns(int column, int count)
```

- `removeRow()` — удаляет строку с указанным индексом. Прототип метода:

```
void removeRow(int row)
```

- `removeRows()` — удаляет указанное количество строк, начиная со строки, имеющей индекс `row`. Прототип метода:

```
void removeRows(int row, int count)
```

- `removeColumn()` — удаляет столбец с указанным индексом. Прототип метода:

```
void removeColumn(int column)
```

- `removeColumns()` — удаляет указанное количество столбцов, начиная со столбца, имеющего индекс `column`. Прототип метода:

```
void removeColumns(int column, int count)
```

- `takeChild()` — удаляет указанный дочерний элемент и возвращает его. Прототип метода:

```
QStandardItem *takeChild(int row, int column = 0)
```

- `takeRow()` — удаляет указанную строку из дочерней таблицы и возвращает ее. Прототип метода:

```
QList<QStandardItem *> takeRow(int row)
```

- `takeColumn()` — удаляет указанный столбец из дочерней таблицы и возвращает его. Прототип метода:

```
QList<QStandardItem *> takeColumn(int column)
```

- `parent()` — возвращает указатель на родительский элемент или нулевой указатель. Прототип метода:

```
QStandardItem *parent() const
```

- `child()` — возвращает указатель на дочерний элемент или нулевой указатель. Прототип метода:

```
QStandardItem *child(int row, int column = 0) const
```

- `hasChildren()` — возвращает значение `true`, если существует дочерний элемент, и `false` — в противном случае. Прототип метода:

```
bool hasChildren() const
```

- `setData()` — устанавливает значение для указанной роли. Прототип метода:

```
virtual void setData(const QVariant &value, int role = Qt::UserRole + 1)
```

- `data()` — возвращает значение, хранимое в указанной роли. Прототип метода:

```
virtual QVariant data(int role = Qt::UserRole + 1) const
```

- `setText()` — задает текст элемента. Прототип метода:

```
void setText(const QString &text)
```

- `text()` — возвращает текст элемента. Прототип метода:

```
QString text() const
```

- `setTextAlignment()` — задает выравнивание текста внутри элемента. Прототип метода:

```
void setTextAlignment(Qt::Alignment alignment)
```

- `setIcon()` — задает значок, который будет отображен перед текстом. Прототип метода:

```
void setIcon(const QIcon &icon)
```

- `setToolTip()` — задает текст всплывающей подсказки. Прототип метода:

```
void setToolTip(const QString &toolTip)
```

- `setWhatsThis()` — задает текст для справки. Прототип метода:

```
void setWhatsThis(const QString &whatsThis)
```

- `setFont()` — задает шрифт элемента. Прототип метода:

```
void setFont(const QFont &font)
```

- `setBackground()` — задает цвет фона. Прототип метода:

```
void setBackground(const QBrush &brush)
```

- `setForeground()` — задает цвет текста. Прототип метода:

```
void setForeground(const QBrush &brush)
```

- `setCheckable()` — если в качестве параметра указано значение `true`, то пользователь может взаимодействовать с флажком. Прототип метода:

```
void setCheckable(bool checkable)
```

- `isCheckable()` — возвращает значение `true`, если пользователь может взаимодействовать с флажком, и `false` — в противном случае. Прототип метода:

  ```
  bool isCheckable() const
  ```

- `setCheckState()` — задает статус флажка. Прототип метода:

  ```
  void setCheckState(Qt::CheckState state)
  ```

  Могут быть указаны следующие константы:

  - `Qt::Unchecked` — флажок сброшен;
  - `Qt::PartiallyChecked` — флажок частично установлен;
  - `Qt::Checked` — флажок установлен;

- `checkState()` — возвращает текущий статус флажка. Прототип метода:

  ```
  Qt::CheckState checkState() const
  ```

- `setAutoTristate()` — если в качестве параметра указано значение `true`, то флажок может иметь три состояния: установлен, сброшен и частично установлен. Управление состоянием родительского флажка выполняется автоматически. Прототип метода:

  ```
  void setAutoTristate(bool tristate)
  ```

  Получение значения:

  ```
  bool isAutoTristate() const
  ```

- `setFlags()` — задает свойства элемента *(см. разд. 7.4.1)*. Прототип метода:

  ```
  void setFlags(Qt::ItemFlags flags)
  ```

- `flags()` — возвращает значение установленных свойств элемента. Прототип метода:

  ```
  Qt::ItemFlags flags() const
  ```

- `setSelectable()` — если в качестве параметра указано значение `true`, то пользователь может выделить элемент. Прототип метода:

  ```
  void setSelectable(bool selectable)
  ```

- `setEditable()` — если в качестве параметра указано значение `true`, то пользователь может редактировать текст элемента. Прототип метода:

  ```
  void setEditable(bool editable)
  ```

- `setDragEnabled()` — если в качестве параметра указано значение `true`, то перетаскивание элемента разрешено. Прототип метода:

  ```
  void setDragEnabled(bool dragEnabled)
  ```

- `setDropEnabled()` — если в качестве параметра указано значение `true`, то сброс разрешен. Прототип метода:

  ```
  void setDropEnabled(bool dropEnabled)
  ```

- `setEnabled()` — если в качестве параметра указано значение `true`, то пользователь может взаимодействовать с элементом. Значение `false` делает элемент недоступным. Прототип метода:

  ```
  void setEnabled(bool enabled)
  ```

- `clone()` — возвращает копию элемента. Прототип метода:

  ```
  virtual QStandardItem *clone() const
  ```

- `index()` — возвращает индекс элемента (экземпляр класса `QModelIndex`). Прототип метода:

  ```
  QModelIndex index() const
  ```

- `model()` — возвращает указатель на модель. Прототип метода:

  ```
  QStandardItemModel *model() const
  ```

- `sortChildren()` — производит сортировку дочерней таблицы. Если во втором параметре указана константа `Qt::AscendingOrder`, то сортировка производится в прямом порядке, а если `Qt::DescendingOrder`, то в обратном. Прототип метода:

  ```
  void sortChildren(int column, Qt::SortOrder order = Qt::AscendingOrder)
  ```

7.5. Представления

Для отображения элементов модели предназначены следующие классы представлений:

- `ListView` — реализует простой список с возможностью выбора как одного, так и нескольких пунктов. Кроме того, с помощью этого класса можно отображать значки;
- `QTableView` — реализует таблицу;
- `QTreeView` — реализует иерархический список.

Помимо этих классов для отображения элементов модели можно воспользоваться классами `QComboBox` (раскрывающийся список; *см. разд. 7.1*), `QListWidget` (простой список), `QTableWidget` (таблица) и `QTreeWidget` (иерархический список). Последние три класса нарушают концепцию «модель/представление», хотя и базируются на этой концепции. За подробной информацией по этим классам обращайтесь к документации.

7.5.1. Класс *QAbstractItemView*

Абстрактный класс `QAbstractItemView` является базовым классом для всех представлений. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                        — QAbstractItemView
```

Класс `QAbstractItemView` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setCurrentIndex()` — делает элемент с указанным индексом (экземпляр класса `QModelIndex`) текущим. Метод является слотом. Прототип метода:

  ```
  void setCurrentIndex(const QModelIndex &index)
  ```

- `currentIndex()` — возвращает индекс (экземпляр класса `QModelIndex`) текущего элемента. Прототип метода:

  ```
  QModelIndex currentIndex() const
  ```

- `setRootIndex()` — задает корневой элемент. В качестве параметра указывается экземпляр класса `QModelIndex`. Метод является слотом. Прототип метода:

  ```
  virtual void setRootIndex(const QModelIndex &index)
  ```

- `rootIndex()` — возвращает индекс (экземпляр класса `QModelIndex`) корневого элемента. Прототип метода:

  ```
  QModelIndex rootIndex() const
  ```

- `setAlternatingRowColors()` — если в качестве параметра указано значение `true`, то четные и нечетные строки будут иметь разный цвет фона. Прототип метода:

  ```
  void setAlternatingRowColors(bool enable)
  ```

- `setIndexWidget()` — устанавливает компонент в позицию, указанную индексом (экземпляр класса `QModelIndex`). Прототип метода:

  ```
  void setIndexWidget(const QModelIndex &index, QWidget *widget)
  ```

- `indexWidget()` — возвращает указатель на компонент, установленный ранее в позицию, указанную индексом (экземпляр класса `QModelIndex`). Прототип метода:

  ```
  QWidget *indexWidget(const QModelIndex &index) const
  ```

- `setSelectionModel()` — устанавливает модель выделения. Прототип метода:

  ```
  virtual void setSelectionModel(QItemSelectionModel *selectionModel)
  ```

- `selectionModel()` — возвращает указатель на модель выделения. Прототип метода:

  ```
  QItemSelectionModel *selectionModel() const
  ```

- `setSelectionMode()` — задает режим выделения элементов. Прототип метода:

  ```
  void setSelectionMode(QAbstractItemView::SelectionMode mode)
  ```

  В качестве параметра указываются следующие константы:

  - `QAbstractItemView::NoSelection` — элементы не могут быть выделены;
  - `QAbstractItemView::SingleSelection` — можно выделить только один элемент;
  - `QAbstractItemView::MultiSelection` — можно выделить несколько элементов. Повторный щелчок на элементе снимает выделение;

- `QAbstractItemView::ExtendedSelection` — можно выделить несколько элементов, удерживая нажатой клавишу <Ctrl> или щелкнув мышью на элементе левой кнопкой мыши и перемещая мышь, не отпуская кнопку. Если удерживать нажатой клавишу <Shift>, все элементы от текущей позиции до позиции щелчка мышью выделяются;
- `QAbstractItemView::ContiguousSelection` — можно выделить несколько элементов, щелкнув мышью на элементе левой кнопкой мыши и перемещая мышь, не отпуская кнопку. Если удерживать нажатой клавишу <Shift>, все элементы от текущей позиции до позиции щелчка мышью выделяются;

❐ `setSelectionBehavior()` — задает режим выделения. Прототип метода:

```
void setSelectionBehavior(QAbstractItemView::SelectionBehavior behavior)
```

В качестве параметра указываются следующие константы:

- `QAbstractItemView::SelectItems` — выделяется отдельный элемент;
- `QAbstractItemView::SelectRows` — выделяется строка целиком;
- `QAbstractItemView::SelectColumns` — выделяется столбец целиком;

❐ `selectAll()` — выделяет все элементы. Метод является слотом. Прототип метода:

```
virtual void selectAll()
```

❐ `clearSelection()` — снимает выделение. Метод является слотом. Прототип метода:

```
void clearSelection()
```

❐ `setEditTriggers()` — задает действие, при котором производится начало редактирования текста элемента. Прототип метода:

```
void setEditTriggers(QAbstractItemView::EditTriggers triggers)
```

В качестве параметра указывается комбинация следующих констант:

- `QAbstractItemView::NoEditTriggers` — редактировать нельзя;
- `QAbstractItemView::CurrentChanged` — при выделении элемента;
- `QAbstractItemView::DoubleClicked` — при двойном щелчке мышью;
- `QAbstractItemView::SelectedClicked` — при одинарном щелчке мышью на выделенном элементе;
- `QAbstractItemView::EditKeyPressed` — при нажатии клавиши <F2>;
- `QAbstractItemView::AnyKeyPressed` — при нажатии любой символьной клавиши;
- `QAbstractItemView::AllEditTriggers` — при любом вышеперечисленном действии;

❐ `setIconSize()` — задает размер значков. Прототип метода:

```
void setIconSize(const QSize &size)
```

- `setTextElideMode()` — задает режим обрезки текста, если он не помещается в отведенную область. В месте пропуска выводится многоточие. Прототип метода:

  ```
  void setTextElideMode(Qt::TextElideMode mode)
  ```

  Могут быть указаны следующие константы:

  - `Qt::ElideLeft` — текст обрезается слева;
  - `Qt::ElideRight` — текст обрезается справа;
  - `Qt::ElideMiddle` — текст обрезается посередине;
  - `Qt::ElideNone` — текст не обрезается;

- `setTabKeyNavigation()` — если в качестве параметра указано значение `true`, то между элементами можно перемещаться с помощью клавиши <Tab>. Прототип метода:

  ```
  void setTabKeyNavigation(bool enable)
  ```

- `scrollTo()` — прокручивает представление таким образом, чтобы элемент, на который ссылается индекс (экземпляр класса `QModelIndex`), был видим. Прототип метода:

  ```
  virtual void scrollTo(const QModelIndex &index,
          QAbstractItemView::ScrollHint hint = EnsureVisible)
  ```

  В параметре `hint` указываются следующие константы:

  - `QAbstractItemView::EnsureVisible` — элемент должен быть в области видимости;
  - `QAbstractItemView::PositionAtTop` — элемент отображается в верхней части;
  - `QAbstractItemView::PositionAtBottom` — элемент отображается в нижней части;
  - `QAbstractItemView::PositionAtCenter` — элемент отображается в центре;

- `scrollToTop()` — прокручивает представление в самое начало. Метод является слотом. Прототип метода:

  ```
  void scrollToTop()
  ```

- `scrollToBottom()` — прокручивает представление в самый конец. Метод является слотом. Прототип метода:

  ```
  void scrollToBottom()
  ```

- `setDragEnabled()` — если в качестве параметра указано значение `true`, то перетаскивание элементов разрешено. Прототип метода:

  ```
  void setDragEnabled(bool enable)
  ```

- `setDragDropMode()` — задает режим технологии drag & drop. Прототип метода:

  ```
  void setDragDropMode(QAbstractItemView::DragDropMode behavior)
  ```

В качестве параметра указываются следующие константы:

- `QAbstractItemView::NoDragDrop` — drag & drop не поддерживается;
- `QAbstractItemView::DragOnly` — поддерживается только перетаскивание;
- `QAbstractItemView::DropOnly` — поддерживается только сбрасывание;
- `QAbstractItemView::DragDrop` — поддерживается перетаскивание и сбрасывание;
- `QAbstractItemView::InternalMove` — перетаскивание и сбрасывание самого элемента, а не его копии, представление принимает операции только от себя;

❐ `setDefaultDropAction()` — задает действие по умолчанию при перетаскивании. Прототип метода:

```
void setDefaultDropAction(Qt::DropAction dropAction)
```

❐ `setDropIndicatorShown()` — если в качестве параметра указано значение `true`, то позиция возможного сброса элемента будет выделена. Прототип метода:

```
void setDropIndicatorShown(bool enable)
```

❐ `setAutoScroll()` — если в качестве параметра указано значение `true`, то при перетаскивании пункта будет производиться автоматическая прокрутка. Прототип метода:

```
void setAutoScroll(bool enable)
```

❐ `setAutoScrollMargin()` — задает расстояние от края области, при достижении которого будет производиться автоматическая прокрутка области. Прототип метода:

```
void setAutoScrollMargin(int margin)
```

Класс `QAbstractItemView` содержит следующие основные сигналы:

❐ `activated(const QModelIndex&)` — генерируется при активизации элемента, например путем нажатия клавиши <Enter>. Внутри обработчика через параметр доступен индекс элемента (экземпляр класса `QModelIndex`);

❐ `pressed(const QModelIndex&)` — генерируется при нажатии кнопки мыши над элементом. Внутри обработчика через параметр доступен индекс элемента (экземпляр класса `QModelIndex`);

❐ `clicked(const QModelIndex&)` — генерируется при нажатии и отпускании кнопки мыши над элементом. Внутри обработчика через параметр доступен индекс элемента (экземпляр класса `QModelIndex`);

❐ `doubleClicked(const QModelIndex&)` — генерируется при двойном щелчке мышью над элементом. Внутри обработчика через параметр доступен индекс элемента (экземпляр класса `QModelIndex`);

❐ `entered(const QModelIndex&)` — генерируется при вхождении указателя мыши в область элемента. Чтобы сигнал сработал, необходимо включить обработку

перемещения указателя с помощью метода `setMouseTracking()` из класса `QWidget`. Внутри обработчика через параметр доступен индекс элемента (экземпляр класса `QModelIndex`);

- `viewportEntered()` — генерируется при вхождении указателя мыши в область компонента. Чтобы сигнал сработал, необходимо включить обработку перемещения указателя с помощью метода `setMouseTracking()` из класса `QWidget`.

7.5.2. Простой список

Класс `QListView` реализует простой список с возможностью выбора как одного, так и нескольких пунктов. Кроме того, с помощью этого класса можно отображать значки. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                        — QAbstractItemView — QListView
```

Формат конструктора класса `QListView`:

```
#include <QListView>
QListView(QWidget *parent = nullptr)
```

Класс `QListView` наследует все методы и сигналы из класса `QAbstractItemView` *(см. разд. 7.5.1)* и дополнительно содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setModel()` — устанавливает модель. Прототип метода:

  ```
  virtual void setModel(QAbstractItemModel *model)
  ```

- `model()` — возвращает указатель на модель. Прототип метода:

  ```
  QAbstractItemModel *model() const
  ```

- `setModelColumn()` — задает индекс отображаемого столбца в табличной модели. Прототип метода:

  ```
  void setModelColumn(int column)
  ```

- `setViewMode()` — задает режим отображения элементов. Прототип метода:

  ```
  void setViewMode(QListView::ViewMode mode)
  ```

  В качестве параметра указываются следующие константы:

  - `QListView::ListMode` — элементы размещаются сверху вниз, а значки имеют маленькие размеры;
  - `QListView::IconMode` — элементы размещаются слева направо, а значки имеют большие размеры. Элементы можно свободно перемещать с помощью мыши;

- `setMovement()` — задает режим перемещения элементов. Прототип метода:

  ```
  void setMovement(QListView::Movement movement)
  ```

В качестве параметра указываются следующие константы:

- `QListView::Static` — пользователь не может перемещать элементы;
- `QListView::Free` — свободное перемещение;
- `QListView::Snap` — перемещение по сетке (размеры задаются методом `setGridSize()`);

❐ `setGridSize()` — задает размеры вспомогательной сетки. Прототип метода:

```
void setGridSize(const QSize &size)
```

❐ `setResizeMode()` — задает режим положения элементов при изменении размера списка. Прототип метода:

```
void setResizeMode(QListView::ResizeMode mode)
```

В качестве параметра указываются следующие константы:

- `QListView::Fixed` — элементы остаются в том же положении;
- `QListView::Adjust` — положение элементов изменяется при изменении размеров;

❐ `setFlow()` — задает порядок вывода элементов. Прототип метода:

```
void setFlow(QListView::Flow flow)
```

В качестве параметра указываются следующие константы:

- `QListView::LeftToRight` — слева направо;
- `QListView::TopToBottom` — сверху вниз;

❐ `setWrapping()` — если в качестве параметра указано значение `false`, то перенос элементов на новую строку (если они не помещаются в ширину области) запрещен. Прототип метода:

```
void setWrapping(bool enable)
```

❐ `setWordWrap()` — если в качестве параметра указано значение `true`, то текст элемента может быть перенесен на другую строку. Прототип метода:

```
void setWordWrap(bool on)
```

❐ `setLayoutMode()` — задает режим размещения элементов. Прототип метода:

```
void setLayoutMode(QListView::LayoutMode mode)
```

В качестве параметра указываются следующие константы:

- `QListView::SinglePass` — элементы размещаются все сразу. Если список слишком большой, то окно будет заблокировано, пока все элементы не будут отображены;
- `QListView::Batched` — элементы размещаются блоками. Размер блока задается с помощью метода `setBatchSize()`;

❐ `setUniformItemSizes()` — если в качестве параметра указано значение `true`, то все элементы будут иметь одинаковый размер. Прототип метода:

```
void setUniformItemSizes(bool enable)
```

- `setSpacing()` — задает отступ вокруг элемента. Прототип метода:

  ```
  void setSpacing(int space)
  ```

- `setSelectionRectVisible()` — если в качестве параметра указано значение `true`, то при выделении будет отображаться вспомогательная рамка, показывающая область выделения. Метод доступен только при использовании режима множественного выделения. Прототип метода:

  ```
  void setSelectionRectVisible(bool show)
  ```

- `setRowHidden()` — если во втором параметре указано значение `true`, то строка с индексом, указанным в первом параметре, будет скрыта. Значение `false` отображает строку. Прототип метода:

  ```
  void setRowHidden(int row, bool hide)
  ```

- `isRowHidden()` — возвращает значение `true`, если строка с указанным индексом скрыта, и `false` — в противном случае. Прототип метода:

  ```
  bool isRowHidden(int row) const
  ```

Класс `QListView` содержит сигнал `indexesMoved(const QModelIndexList&)`, который генерируется при перемещении элементов. Внутри обработчика через параметр доступен список экземпляров класса `QModelIndex`.

7.5.3. Таблица

Класс `QTableView` реализует таблицу. Иерархия наследования выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                        — QAbstractItemView — QTableView
```

Формат конструктора класса `QTableView`:

```
#include <QTableView>
QTableView(QWidget *parent = nullptr)
```

Класс `QTableView` наследует все методы и сигналы из класса `QAbstractItemView` *(см. разд. 7.5.1)* и дополнительно содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setModel()` — устанавливает модель. Прототип метода:

  ```
  virtual void setModel(QAbstractItemModel *model)
  ```

- `model()` — возвращает указатель на модель. Прототип метода:

  ```
  QAbstractItemModel *model() const
  ```

- `horizontalHeader()` — возвращает указатель на горизонтальный заголовок (экземпляр класса `QHeaderView`). Прототип метода:

  ```
  QHeaderView *horizontalHeader() const
  ```

- `verticalHeader()` — возвращает указатель на вертикальный заголовок (экземпляр класса `QHeaderView`). Прототип метода:

  ```
  QHeaderView *verticalHeader() const
  ```

Скрыть заголовки можно так:

```
view->horizontalHeader()->hide();
view->verticalHeader()->hide();
```

- `setRowHeight()` — задает высоту строки с указанным в первом параметре индексом. Прототип метода:

```
void setRowHeight(int row, int height)
```

- `setColumnWidth()` — задает ширину столбца с указанным в первом параметре индексом. Прототип метода:

```
void setColumnWidth(int column, int width)
```

- `rowHeight()` — возвращает высоту строки. Прототип метода:

```
int rowHeight(int row) const
```

- `columnWidth()` — возвращает ширину столбца. Прототип метода:

```
int columnWidth(int column) const
```

- `resizeRowToContents()` — изменяет размер указанной строки таким образом, чтобы поместилось все содержимое. Метод является слотом. Прототип метода:

```
void resizeRowToContents(int row)
```

- `resizeRowsToContents()` — изменяет размер всех строк таким образом, чтобы поместилось все содержимое. Метод является слотом. Прототип метода:

```
void resizeRowsToContents()
```

- `resizeColumnToContents()` — изменяет размер указанного столбца таким образом, чтобы поместилось все содержимое. Метод является слотом. Прототип метода:

```
void resizeColumnToContents(int column)
```

- `resizeColumnsToContents()` — изменяет размер всех столбцов таким образом, чтобы поместилось все содержимое. Метод является слотом. Прототип метода:

```
void resizeColumnsToContents()
```

- `setSpan()` — растягивает элемент с указанными в первых двух параметрах индексами на заданное количество строк и столбцов. Происходит как бы объединение ячеек таблицы. Прототип метода:

```
void setSpan(int row, int column, int rowSpanCount, int columnSpanCount)
```

- `rowSpan()` — возвращает количество ячеек в строке, которое занимает элемент с указанными индексами. Прототип метода:

```
int rowSpan(int row, int column) const
```

- `columnSpan()` — возвращает количество ячеек в столбце, которое занимает элемент с указанными индексами. Прототип метода:

```
int columnSpan(int row, int column) const
```

- `clearSpans()` — отменяет все объединения ячеек. Прототип метода:

```
void clearSpans()
```

- `setRowHidden()` — если во втором параметре указано значение `true`, то строка с индексом, указанным в первом параметре, будет скрыта. Значение `false` отображает строку. Прототип метода:

```
void setRowHidden(int row, bool hide)
```

- `hideRow()` — скрывает строку с указанным индексом. Метод является слотом. Прототип метода:

```
void hideRow(int row)
```

- `showRow()` — отображает строку с указанным индексом. Метод является слотом. Прототип метода:

```
void showRow(int row)
```

- `setColumnHidden()` — если во втором параметре указано значение `true`, то столбец с индексом, указанным в первом параметре, будет скрыт. Значение `false` отображает столбец. Прототип метода:

```
void setColumnHidden(int column, bool hide)
```

- `hideColumn()` — скрывает столбец с указанным индексом. Метод является слотом. Прототип метода:

```
void hideColumn(int column)
```

- `showColumn()` — отображает столбец с указанным индексом. Метод является слотом. Прототип метода:

```
void showColumn(int column)
```

- `isRowHidden()` — возвращает значение `true`, если строка с указанным индексом скрыта, и `false` — в противном случае. Прототип метода:

```
bool isRowHidden(int row) const
```

- `isColumnHidden()` — возвращает значение `true`, если столбец с указанным индексом скрыт, и `false` — в противном случае. Прототип метода:

```
bool isColumnHidden(int column) const
```

- `selectRow()` — выделяет строку с указанным индексом. Метод является слотом. Прототип метода:

```
void selectRow(int row)
```

- `selectColumn()` — выделяет столбец с указанным индексом. Метод является слотом. Прототип метода:

```
void selectColumn(int column)
```

- `setGridStyle()` — задает стиль линий сетки. Прототип метода:

```
void setGridStyle(Qt::PenStyle style)
```

В качестве параметра указываются следующие константы:

- `Qt::NoPen` — линии не выводятся;
- `Qt::SolidLine` — сплошная линия;
- `Qt::DashLine` — штриховая линия;
- `Qt::DotLine` — пунктирная линия;
- `Qt::DashDotLine` — штрих и точка, штрих и точка и т. д.;
- `Qt::DashDotDotLine` — штрих и две точки, штрих и две точки и т. д.;

- `setShowGrid()` — если в качестве параметра указано значение `true`, то сетка будет отображена, а если `false`, то скрыта. Метод является слотом. Прототип метода:

```
void setShowGrid(bool show)
```

- `setSortingEnabled()` — если в качестве параметра указано значение `true`, то столбцы можно сортировать с помощью щелчка мышью на заголовке столбца. При этом в заголовке показывается текущее направление сортировки. Прототип метода:

```
void setSortingEnabled(bool enable)
```

- `setCornerButtonEnabled()` — если в качестве параметра указано значение `true`, то с помощью кнопки в левом верхнем углу заголовка можно выделить всю таблицу. Значение `false` отключает кнопку. Прототип метода:

```
void setCornerButtonEnabled(bool enable)
```

- `setWordWrap()` — если в качестве параметра указано значение `true`, то текст элемента может быть перенесен на другую строку. Прототип метода:

```
void setWordWrap(bool on)
```

- `sortByColumn()` — производит сортировку. Если во втором параметре указана константа `Qt::AscendingOrder`, то сортировка производится в прямом порядке, а если `Qt::DescendingOrder`, то в обратном. Метод является слотом. Прототип метода:

```
void sortByColumn(int column, Qt::SortOrder order)
```

7.5.4. Иерархический список

Класс `QTreeView` реализует иерархический список. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                        — QAbstractItemView — QTreeView
```

Формат конструктора класса `QTreeView`:

```
#include <QTreeView>
QTreeView(QWidget *parent = nullptr)
```

Класс `QTreeView` наследует все методы и сигналы из класса `QAbstractItemView` *(см. разд. 7.5.1)* и дополнительно содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setModel()` — устанавливает модель. Прототип метода:

  ```
  virtual void setModel(QAbstractItemModel *model)
  ```

- `model()` — возвращает указатель на модель. Прототип метода:

  ```
  QAbstractItemModel *model() const
  ```

- `header()` — возвращает указатель на горизонтальный заголовок (экземпляр класса `QHeaderView`). Прототип метода:

  ```
  QHeaderView *header() const
  ```

- `setColumnWidth()` — задает ширину столбца с указанным в первом параметре индексом. Прототип метода:

  ```
  void setColumnWidth(int column, int width)
  ```

- `columnWidth()` — возвращает ширину столбца. Прототип метода:

  ```
  int columnWidth(int column) const
  ```

- `resizeColumnToContents()` — изменяет ширину указанного столбца таким образом, чтобы поместилось все содержимое. Метод является слотом. Прототип метода:

  ```
  void resizeColumnToContents(int column)
  ```

- `setUniformRowHeights()` — если в качестве параметра указано значение `true`, то все элементы будут иметь одинаковую высоту. Прототип метода:

  ```
  void setUniformRowHeights(bool uniform)
  ```

- `setHeaderHidden()` — если в качестве параметра указано значение `true`, то заголовок будет скрыт. Значение `false` отображает заголовок. Прототип метода:

  ```
  void setHeaderHidden(bool hide)
  ```

- `isHeaderHidden()` — возвращает значение `true`, если заголовок скрыт, и `false` — в противном случае. Прототип метода:

  ```
  bool isHeaderHidden() const
  ```

- `setColumnHidden()` — если во втором параметре указано значение `true`, то столбец с индексом, указанным в первом параметре, будет скрыт. Значение `false` отображает столбец. Прототип метода:

  ```
  void setColumnHidden(int column, bool hide)
  ```

- `hideColumn()` — скрывает столбец с указанным индексом. Метод является слотом. Прототип метода:

  ```
  void hideColumn(int column)
  ```

- `showColumn()` — отображает столбец с указанным индексом. Метод является слотом. Прототип метода:

```
void showColumn(int column)
```

- `isColumnHidden()` — возвращает значение `true`, если столбец с указанным индексом скрыт, и `false` — в противном случае. Прототип метода:

```
bool isColumnHidden(int column) const
```

- `setRowHidden()` — если в третьем параметре указано значение `true`, то строка с индексом `row` и родителем `parent` будет скрыта. Значение `false` отображает строку. Прототип метода:

```
void setRowHidden(int row, const QModelIndex &parent, bool hide)
```

- `isRowHidden()` — возвращает значение `true`, если строка с указанным индексом `row` и родителем `parent` скрыта, и `false` — в противном случае. Прототип метода:

```
bool isRowHidden(int row, const QModelIndex &parent) const
```

- `setExpanded()` — если во втором параметре указано значение `true`, то элементы, которые являются дочерними для элемента с указанным в первом параметре индексом, будут отображены, а если `false`, то скрыты. Прототип метода:

```
void setExpanded(const QModelIndex &index, bool expanded)
```

- `expand()` — отображает элементы, которые являются дочерними для элемента с указанным индексом. Метод является слотом. Прототип метода:

```
void expand(const QModelIndex &index)
```

- `expandToDepth()` — отображает все дочерние элементы до указанного уровня. Метод является слотом. Прототип метода:

```
void expandToDepth(int depth)
```

- `expandAll()` — отображает все дочерние элементы. Метод является слотом. Прототип метода:

```
void expandAll()
```

- `collapse()` — скрывает элементы, которые являются дочерними для элемента с указанным индексом. Метод является слотом. Прототип метода:

```
void collapse(const QModelIndex &index)
```

- `collapseAll()` — скрывает все дочерние элементы. Метод является слотом. Прототип метода:

```
void collapseAll()
```

- `isExpanded()` — возвращает значение `true`, если элементы, которые являются дочерними для элемента с указанным индексом, отображены, и `false` — в противном случае. Прототип метода:

```
bool isExpanded(const QModelIndex &index) const
```

- `setItemsExpandable()` — если в качестве параметра указано значение `false`, то пользователь не сможет отображать или скрывать дочерние элементы. Прототип метода:

  ```
  void setItemsExpandable(bool enable)
  ```

- `setAnimated()` — если в качестве параметра указано значение `true`, то отображение и сокрытие дочерних элементов будет производиться с анимацией. Прототип метода:

  ```
  void setAnimated(bool enable)
  ```

- `setIndentation()` — задает отступ для дочерних элементов. Прототип метода:

  ```
  void setIndentation(int)
  ```

- `setRootIsDecorated()` — если в качестве параметра указано значение `false`, то для элементов верхнего уровня не будут показываться компоненты, с помощью которых производится отображение и сокрытие дочерних элементов. Прототип метода:

  ```
  void setRootIsDecorated(bool show)
  ```

- `setSortingEnabled()` — если в качестве параметра указано значение `true`, то столбцы можно сортировать с помощью щелчка мышью на заголовке столбца. При этом в заголовке показывается текущее направление сортировки. Прототип метода:

  ```
  void setSortingEnabled(bool enable)
  ```

- `sortByColumn()` — производит сортировку. Если во втором параметре указана константа `Qt::AscendingOrder`, то сортировка производится в прямом порядке, а если `Qt::DescendingOrder`, то в обратном. Метод является слотом. Прототип метода:

  ```
  void sortByColumn(int column, Qt::SortOrder order)
  ```

- `setWordWrap()` — если в качестве параметра указано значение `true`, то текст элемента может быть перенесен на другую строку. Прототип метода:

  ```
  void setWordWrap(bool on)
  ```

Класс `QTreeView` содержит следующие сигналы:

- `expanded(const QModelIndex&)` — генерируется при отображении дочерних элементов. Внутри обработчика через параметр доступен индекс (экземпляр класса `QModelIndex`) элемента;
- `collapsed(const QModelIndex&)` — генерируется при сокрытии дочерних элементов. Внутри обработчика через параметр доступен индекс (экземпляр класса `QModelIndex`) элемента.

7.5.5. Управление заголовками строк и столбцов

Класс `QHeaderView` реализует заголовки строк и столбцов в представлениях `QTableView` и `QTreeView`. Получить указатели на заголовки в классе `QTableView` по-

зволяют методы `horizontalHeader()` и `verticalHeader()`, а для установки заголовков предназначены методы `setHorizontalHeader()` и `setVerticalHeader()`. Получить указатель на заголовок в классе `QTreeView` позволяет метод `header()`, а для установки заголовка предназначен метод `setHeader()`. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame — QAbstractScrollArea —
                        — QAbstractItemView — QHeaderView
```

Формат конструктора класса `QHeaderView`:

```
#include <QHeaderView>
QHeaderView(Qt::Orientation orientation, QWidget *parent = nullptr)
```

Параметр `orientation` позволяет задать ориентацию заголовка. В качестве значения указывается константа `Qt::Horizontal` или `Qt::Vertical`.

Класс `QHeaderView` наследует все методы и сигналы из класса `QAbstractItemView` *(см. разд. 7.5.1)* и дополнительно содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `count()` — возвращает количество секций в заголовке. Прототип метода:

  ```
  int count() const
  ```

- `setDefaultSectionSize()` — задает размер секций по умолчанию. Прототип метода:

  ```
  void setDefaultSectionSize(int size)
  ```

- `defaultSectionSize()` — возвращает размер секций по умолчанию. Прототип метода:

  ```
  int defaultSectionSize() const
  ```

- `setMinimumSectionSize()` — задает минимальный размер секций. Прототип метода:

  ```
  void setMinimumSectionSize(int size)
  ```

- `minimumSectionSize()` — возвращает минимальный размер секций. Прототип метода:

  ```
  int minimumSectionSize() const
  ```

- `setMaximumSectionSize()` — задает максимальный размер секций. Прототип метода:

  ```
  void setMaximumSectionSize(int size)
  ```

- `maximumSectionSize()` — возвращает максимальный размер секций. Прототип метода:

  ```
  int maximumSectionSize() const
  ```

- `resizeSection()` — изменяет размер секции с указанным индексом. Прототип метода:

  ```
  void resizeSection(int logicalIndex, int size)
  ```

- `sectionSize()` — возвращает размер секции с указанным индексом. Прототип метода:

  ```
  int sectionSize(int logicalIndex) const
  ```

- `setSectionResizeMode()` — задает режим изменения размеров для секций. Прототипы метода:

  ```
  void setSectionResizeMode(QHeaderView::ResizeMode mode)
  void setSectionResizeMode(int logicalIndex, QHeaderView::ResizeMode mode)
  ```

  В качестве параметра `mode` могут быть указаны следующие константы:

  - `QHeaderView::Interactive` — размер может быть изменен пользователем или программно;
  - `QHeaderView::Stretch` — секции автоматически равномерно распределяют свободное пространство между собой. Размер не может быть изменен ни пользователем, ни программно;
  - `QHeaderView::Fixed` — размер может быть изменен только программно;
  - `QHeaderView::ResizeToContents` — размер определяется автоматически по содержимому секции. Размер не может быть изменен ни пользователем, ни программно;

- `setStretchLastSection()` — если в качестве параметра указано значение `true`, то последняя секция будет занимать все свободное пространство. Прототип метода:

  ```
  void setStretchLastSection(bool stretch)
  ```

- `setCascadingSectionResizes()` — если в качестве параметра указано значение `true`, то изменение размеров одной секции может привести к изменению размеров других секций. Прототип метода:

  ```
  void setCascadingSectionResizes(bool enable)
  ```

- `setSectionHidden()` — если во втором параметре указано значение `true`, то секция с индексом, указанным в первом параметре, будет скрыта. Значение `false` отображает секцию. Прототип метода:

  ```
  void setSectionHidden(int logicalIndex, bool hide)
  ```

- `hideSection()` — скрывает секцию с указанным индексом. Прототип метода:

  ```
  void hideSection(int logicalIndex)
  ```

- `showSection()` — отображает секцию с указанным индексом. Прототип метода:

  ```
  void showSection(int logicalIndex)
  ```

- `isSectionHidden()` — возвращает значение `true`, если секция с указанным индексом скрыта, и `false` — в противном случае. Прототип метода:

  ```
  bool isSectionHidden(int logicalIndex) const
  ```

- `sectionsHidden()` — возвращает значение `true`, если существует скрытая секция, и `false` — в противном случае. Прототип метода:

  ```
  bool sectionsHidden() const
  ```

- `hiddenSectionCount()` — возвращает количество скрытых секций. Прототип метода:

```
int hiddenSectionCount() const
```

- `setDefaultAlignment()` — задает выравнивание текста внутри заголовков. Прототип метода:

```
void setDefaultAlignment(Qt::Alignment alignment)
```

- `setHighlightSections()` — если в качестве параметра указано значение `true`, то текст заголовка текущей секции будет выделен. Прототип метода:

```
void setHighlightSections(bool highlight)
```

- `setSectionsClickable()` — если в качестве параметра указано значение `true`, то заголовок будет реагировать на щелчок мышью, при этом выделяя все элементы секции. Прототип метода:

```
void setSectionsClickable(bool clickable)
```

- `setSectionsMovable()` — если в качестве параметра указано значение `true`, то пользователь может перемещать секции с помощью мыши. Прототип метода:

```
void setSectionsMovable(bool movable)
```

- `sectionsMovable()` — возвращает значение `true`, если пользователь может перемещать секции с помощью мыши, и `false` — в противном случае. Прототип метода:

```
bool sectionsMovable() const
```

- `moveSection()` — позволяет переместить секцию. В параметрах указываются визуальные индексы. Прототип метода:

```
void moveSection(int from, int to)
```

- `swapSections()` — меняет две секции местами. В параметрах указываются визуальные индексы. Прототип метода:

```
void swapSections(int first, int second)
```

- `visualIndex()` — преобразует логический (первоначальный порядок следования) индекс в визуальный (отображаемый порядок следования) индекс. Если преобразование прошло неудачно, то возвращается значение `-1`. Прототип метода:

```
int visualIndex(int logicalIndex) const
```

- `logicalIndex()` — преобразует визуальный (отображаемый порядок следования) индекс в логический (первоначальный порядок следования) индекс. Если преобразование прошло неудачно, то возвращается значение `-1`. Прототип метода:

```
int logicalIndex(int visualIndex) const
```

- `saveState()` — возвращает экземпляр класса `QByteArray` с текущими размерами и положением секций. Прототип метода:

```
QByteArray saveState() const
```

- `restoreState()` — восстанавливает размеры и положение секций на основе экземпляра класса `QByteArray`, возвращаемого методом `saveState()`. Прототип метода:

```
bool restoreState(const QByteArray &state)
```

Класс `QHeaderView` содержит следующие сигналы (перечислены только основные сигналы; полный список смотрите в документации):

- `sectionPressed(int)` — генерируется при нажатии левой кнопки мыши над заголовком секции. Внутри обработчика через параметр доступен логический индекс секции;
- `sectionClicked(int)` — генерируется при нажатии и отпускании левой кнопки мыши над заголовком секции. Внутри обработчика через параметр доступен логический индекс секции;
- `sectionDoubleClicked(int)` — генерируется при двойном щелчке мышью на заголовке секции. Внутри обработчика через параметр доступен логический индекс секции;
- `sectionMoved(int,int,int)` — генерируется при изменении положения секции. Внутри обработчика через первый параметр доступен логический индекс секции, через второй параметр — старый визуальный индекс, а через третий — новый визуальный индекс;
- `sectionResized(int,int,int)` — генерируется непрерывно при изменении размера секции. Внутри обработчика через первый параметр доступен логический индекс секции, через второй параметр — старый размер, а через третий — новый размер.

7.6. Управление выделением элементов

Класс `QItemSelectionModel` реализует модель, позволяющую централизованно управлять выделением сразу в нескольких представлениях. Установить модель выделения позволяет метод `setSelectionModel()` из класса `QAbstractItemView`, а получить указатель на модель можно с помощью метода `selectionModel()`. Если одна модель выделения установлена сразу в нескольких представлениях, то выделение элемента в одном представлении приведет к выделению соответствующего элемента в другом представлении. Иерархия наследования выглядит так:

```
QObject — QItemSelectionModel
```

Форматы конструктора класса `QItemSelectionModel`:

```
#include <QItemSelectionModel>
QItemSelectionModel(QAbstractItemModel *model, QObject *parent)
QItemSelectionModel(QAbstractItemModel *model = nullptr)
```

Класс `QItemSelectionModel` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `hasSelection()` — возвращает значение `true`, если существует выделенный элемент, и `false` — в противном случае. Прототип метода:

```
bool hasSelection() const
```

- `isSelected()` — возвращает значение `true`, если элемент с указанным индексом (экземпляр класса `QModelIndex`) выделен, и `false` — в противном случае. Прототип метода:

```
bool isSelected(const QModelIndex &index) const
```

- `isRowSelected()` — возвращает значение `true`, если строка с индексом `row` и родителем `parent` выделена, и `false` — в противном случае. Прототип метода:

```
bool isRowSelected(int row, const QModelIndex &parent = QModelIndex()) const
```

- `isColumnSelected()` — возвращает значение `true`, если столбец с индексом `column` и родителем `parent` выделен, и `false` — в противном случае. Прототип метода:

```
bool isColumnSelected(int column,
            const QModelIndex &parent = QModelIndex()) const
```

- `rowIntersectsSelection()` — возвращает значение `true`, если строка с индексом `row` и родителем `parent` содержит выделенный элемент, и `false` — в противном случае. Прототип метода:

```
bool rowIntersectsSelection(int row,
        const QModelIndex &parent = QModelIndex()) const
```

- `columnIntersectsSelection()` — возвращает значение `true`, если столбец с индексом `column` и родителем `parent` содержит выделенный элемент, и `false` — в противном случае. Прототип метода:

```
bool columnIntersectsSelection(int column,
           const QModelIndex &parent = QModelIndex()) const
```

- `selectedIndexes()` — возвращает список индексов (экземпляры класса `QModelIndex`) выделенных элементов или пустой список. Прототип метода:

```
QModelIndexList selectedIndexes() const
```

- `selectedRows()` — возвращает список индексов (экземпляры класса `QModelIndex`) выделенных элементов из указанного столбца. Элемент попадет в список только в том случае, если строка выделена полностью. Прототип метода:

```
QModelIndexList selectedRows(int column = 0) const
```

- `selectedColumns()` — возвращает список индексов (экземпляры класса `QModelIndex`) выделенных элементов из указанной строки. Элемент попадет в список только в том случае, если столбец выделен полностью. Прототип метода:

```
QModelIndexList selectedColumns(int row = 0) const
```

- `selection()` — возвращает экземпляр класса `QItemSelection`. Прототип метода:

```
const QItemSelection selection() const
```

- `select()` — изменяет выделение элемента. Метод является слотом. Прототипы метода:

```
virtual void select(const QItemSelection &selection,
        QItemSelectionModel::SelectionFlags command)
virtual void select(const QModelIndex &index,
        QItemSelectionModel::SelectionFlags command)
```

  Во втором параметре указываются следующие константы (или их комбинация через оператор `|`):

  - `QItemSelectionModel::NoUpdate` — без изменений;
  - `QItemSelectionModel::Clear` — снимает выделение всех элементов;
  - `QItemSelectionModel::Select` — выделяет элемент;
  - `QItemSelectionModel::Deselect` — снимает выделение с элемента;
  - `QItemSelectionModel::Toggle` — выделяет элемент, если он не выделен, или снимает выделение, если элемент был выделен;
  - `QItemSelectionModel::Current` — изменяет выделение текущего элемента;
  - `QItemSelectionModel::Rows` — индекс будет расширен так, чтобы охватить всю строку;
  - `QItemSelectionModel::Columns` — индекс будет расширен так, чтобы охватить весь столбец;
  - `QItemSelectionModel::SelectCurrent` — комбинация `Select | Current`;
  - `QItemSelectionModel::ToggleCurrent` — комбинация `Toggle | Current`;
  - `QItemSelectionModel::ClearAndSelect` — комбинация `Clear | Select`;

- `setCurrentIndex()` — делает элемент текущим и изменяет режим выделения. Метод является слотом. Прототип метода:

```
virtual void setCurrentIndex(const QModelIndex &index,
        QItemSelectionModel::SelectionFlags command)
```

- `currentIndex()` — возвращает индекс (экземпляр класса `QModelIndex`) текущего элемента. Прототип метода:

```
QModelIndex currentIndex() const
```

- `clearSelection()` — снимает все выделения. Метод является слотом. Прототип метода:

```
void clearSelection()
```

Класс `QItemSelectionModel` содержит следующие основные сигналы:

- `currentChanged(const QModelIndex&, const QModelIndex&)` — генерируется при изменении индекса текущего элемента. Внутри обработчика через первый параметр доступен индекс текущего элемента, а через второй параметр — индекс предыдущего элемента;

- `currentRowChanged(const QModelIndex&, const QModelIndex&)` — генерируется, когда текущий элемент перемещается в другую строку. Внутри обработчика через первый параметр доступен индекс текущего элемента, а через второй параметр — индекс предыдущего элемента;
- `currentColumnChanged(const QModelIndex&, const QModelIndex&)` — генерируется, когда текущий элемент перемещается в другой столбец. Внутри обработчика через первый параметр доступен индекс текущего элемента, а через второй параметр — индекс предыдущего элемента;
- `selectionChanged(const QItemSelection&, const QItemSelection&)` — генерируется при изменении выделения.

7.7. Промежуточные модели

Как вы уже знаете, одну модель можно установить в нескольких представлениях. При этом изменение порядка следования элементов в одном представлении повлечет за собой изменение порядка следования элементов в другом представлении. Чтобы предотвратить изменение порядка следования элементов в базовой модели, следует создать промежуточную модель с помощью класса `QSortFilterProxyModel` и установить ее в представлении. Иерархия наследования для класса `QSortFilterProxyModel` выглядит так:

```
QObject — QAbstractItemModel — QAbstractProxyModel —
                             — QSortFilterProxyModel
```

Формат конструктора класса `QSortFilterProxyModel`:

```
#include <QSortFilterProxyModel>
QSortFilterProxyModel(QObject *parent = nullptr)
```

Класс `QSortFilterProxyModel` наследует следующие методы из класса `QAbstractProxyModel` (перечислены только основные методы; полный список смотрите в документации):

- `setSourceModel()` — устанавливает базовую модель. Прототип метода:

  ```
  virtual void setSourceModel(QAbstractItemModel *sourceModel)
  ```

- `sourceModel()` — возвращает указатель на базовую модель. Прототип метода:

  ```
  QAbstractItemModel *sourceModel() const
  ```

Класс `QSortFilterProxyModel` поддерживает основные методы обычных моделей и дополнительно содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `sort()` — производит сортировку. Прототип метода:

  ```
  virtual void sort(int column, Qt::SortOrder order = Qt::AscendingOrder)
  ```

  Если во втором параметре указана константа `Qt::AscendingOrder`, то сортировка производится в прямом порядке, а если `Qt::DescendingOrder`, то в обратном.

Если в параметре column указать значение -1, то будет использован порядок следования элементов из базовой модели.

Примечание

Чтобы включить сортировку столбцов пользователем, следует передать значение true в метод setSortingEnabled() объекта представления.

- setSortRole() — задает роль *(см. разд. 7.3)*, по которой производится сортировка. По умолчанию сортировка производится по роли DisplayRole. Прототип метода:

```
void setSortRole(int role)
```

- setSortCaseSensitivity() — если в качестве параметра указать константу Qt::CaseInsensitive, то при сортировке не будет учитываться регистр символов, а если Qt::CaseSensitive, то регистр будет учитываться. Прототип метода:

```
void setSortCaseSensitivity(Qt::CaseSensitivity)
```

- setSortLocaleAware() — если в качестве параметра указать значение true, то при сортировке будут учитываться настройки локали. Прототип метода:

```
void setSortLocaleAware(bool)
```

- setFilterFixedString() — в результат попадут только строки, которые содержат заданный фрагмент. Если указать пустую строку, то в результат попадут все строки из базовой модели. Метод является слотом. Прототип метода:

```
void setFilterFixedString(const QString &pattern)
```

- setFilterRegularExpression() — производит фильтрацию элементов в соответствии с указанным регулярным выражением. Если указать пустую строку, то в результат попадут все строки из базовой модели. Метод является слотом. Прототипы метода:

```
void setFilterRegularExpression(const QRegularExpression &regularExpression)
void setFilterRegularExpression(const QString &pattern)
```

- setFilterWildcard() — производит фильтрацию элементов в соответствии с указанной строкой, содержащей подстановочные знаки:
  - ? — один любой символ;
  - * — нуль или более любых символов;
  - [...] — диапазон значений.

  Остальные символы трактуются как есть. Если в качестве параметра указать пустую строку, то в результат попадут все строки из базовой модели. Метод является слотом. Прототип метода:

```
void setFilterWildcard(const QString &pattern)
```

- setFilterKeyColumn() — задает индекс столбца, по которому будет производиться фильтрация. Если в качестве параметра указать значение -1, то будут про-

сматриваться элементы во всех столбцах. По умолчанию фильтрация производится по первому столбцу. Прототип метода:

```
void setFilterKeyColumn(int column)
```

- `setFilterRole()` — задает роль *(см. разд. 7.3)*, по которой производится фильтрация. По умолчанию сортировка производится по роли `DisplayRole`. Прототип метода:

```
void setFilterRole(int role)
```

- `setFilterCaseSensitivity()` — если в качестве параметра указать константу `Qt::CaseInsensitive`, то при фильтрации не будет учитываться регистр символов, а если `Qt::CaseSensitive`, то регистр будет учитываться. Прототип метода:

```
void setFilterCaseSensitivity(Qt::CaseSensitivity)
```

- `setDynamicSortFilter()` — если в качестве параметра указано значение `true`, то при изменении базовой модели будет производиться повторная сортировка или фильтрация. Прототип метода:

```
void setDynamicSortFilter(bool enable)
```

ГЛАВА 8

Работа с графикой

Все компоненты, которые мы рассматривали в предыдущих главах, на самом деле нарисованы. Когда компонент становится видимым (в первый раз при отображении части компонента, ранее перекрытой другим окном, или после изменения настроек), вызывается метод с названием `paintEvent()` (описание метода см. в *разд. 4.7.3*). Вызвать событие перерисовки компонента можно также искусственно с помощью методов `repaint()` и `update()` из класса `QWidget`. Внутри метода `paintEvent()` выполняется рисование компонента с помощью методов класса `QPainter`.

Класс `QPainter` содержит все необходимые средства, позволяющие выполнять рисование геометрических фигур и текста на поверхности, которая реализуется классом `QPaintDevice`. Класс `QWidget` наследует класс `QPaintDevice`. В свою очередь, класс `QWidget` наследуют все компоненты, поэтому мы можем рисовать на поверхности любого компонента. Класс `QPaintDevice` наследуют также классы `QPicture`, `QPixmap`, `QImage`, `QPrinter` и некоторые другие. Класс `QPicture` позволяет сохранить команды рисования в метафайл, а затем считать их из файла и воспроизвести на какой-либо поверхности. Классы `QPixmap` и `QImage` позволяют обрабатывать изображения, а класс `QPrinter` предназначен для вывода данных на принтер или в PDF-файл. Основные методы этих классов мы рассмотрим далее в этой главе.

Библиотека Qt позволяет также работать с SVG-графикой, анимацией, видео, а также содержит поддержку библиотеки OpenGL, предназначенной для обработки двумерной и трехмерной графики. Рассмотрение этих возможностей выходит за рамки данной книги; за подробной информацией обращайтесь к документации.

8.1. Вспомогательные классы

Прежде чем изучать работу с графикой, необходимо рассмотреть несколько вспомогательных классов, с помощью которых производится настройка различных параметров (например, цвета, характеристик шрифта, стиля пера и кисти). Кроме того, мы рассмотрим классы, описывающие геометрические фигуры (например, линию и многоугольник).

8.1.1. Класс *QColor:* цвет

Класс `QColor` описывает цвет в цветовых моделях RGB, CMYK, HSV или HSL. Форматы конструктора класса `QColor`:

```
#include <QColor>
QColor()
QColor(int r, int g, int b, int a = 255)
QColor(const char *name)
QColor(const QString &name)
QColor(QLatin1String name)
QColor(Qt::GlobalColor color)
QColor(QRgba64 rgba64)
QColor(QRgb color)
```

Первый конструктор создает невалидный объект. Проверить объект на валидность можно с помощью метода `isValid()`. Метод возвращает значение `true`, если объект является валидным, и `false` — в противном случае. Прототип метода:

```
bool isValid() const
```

Второй конструктор позволяет указать целочисленные значения красной, зеленой и синей составляющих цвета модели RGB. В качестве значений указываются числа от `0` до `255`. Необязательный параметр `a` задает степень прозрачности цвета. Значение `0` соответствует прозрачному цвету, а значение `255` — полностью непрозрачному. Пример указания красного цвета:

```
QColor red(255, 0, 0);
```

В третьем, четвертом и пятом конструкторах указывается название цвета (`transparent` для прозрачного цвета). Пример указания белого цвета:

```
QColor white("white");
```

Можно также передать значение в следующих форматах: "`#RGB`", "`#RRGGBB`", "`#RRRGGGBBB`", "`#RRRRGGGGBBBB`". Пример (выводится предупреждающее сообщение о том, что лучше в этом случае использовать второй конструктор):

```
QColor white("#ffffff");
```

Получить список всех поддерживаемых названий цветов позволяет статический метод `colorNames()`. Проверить правильность строки с названием цвета можно с помощью статического метода `isValidColor()`. Метод возвращает значение `true`, если строка является допустимой, и `false` — в противном случае. Прототипы методов:

```
static QStringList colorNames()
static bool isValidColor(const QString &name)
static bool isValidColor(QStringView name)
static bool isValidColor(QLatin1String name)
```

Пример:

```
qDebug() << QColor::colorNames();
// QList("aliceblue", "antiquewhite" ...)
qDebug() << QColor::isValidColor("lightcyan"); // true
```

В шестом конструкторе указываются следующие константы: `white`, `black`, `red`, `darkRed`, `green`, `darkGreen`, `blue`, `darkBlue`, `cyan`, `darkCyan`, `magenta`, `darkMagenta`, `yellow`, `darkYellow`, `gray`, `darkGray`, `lightGray`, `color0`, `color1` или `transparent` (прозрачный цвет). Константы `color0` (прозрачный цвет) и `color1` (непрозрачный цвет) используются в двухцветных изображениях. Пример:

```
QColor white(Qt::white);
```

Задать или получить значения в цветовой модели RGB (red, green, blue; красный, зеленый, синий) позволяют следующие методы:

- `setNamedColor()` — задает название цвета в виде строки в форматах `"#RGB"`, `"#RRGGBB"`, `"#RRRGGGBBB"`, `"#RRRRGGGGBBBB"`, `"Название цвета"` или `"transparent"` (для прозрачного цвета). Прототип метода:

  ```
  void setNamedColor(const QString &name)
  void setNamedColor(QStringView name)
  void setNamedColor(QLatin1String name)
  ```

- `name()` — возвращает строковое представление цвета. Прототип метода:

  ```
  QString name(QColor::NameFormat format = HexRgb) const
  ```

  Пример:

  ```
  QColor white("white");
  qDebug() << white.name(); // "#ffffff"
  ```

- `setRgb()` — задает целочисленные значения красной, зеленой и синей составляющих цвета модели RGB. В качестве параметров указываются числа от `0` до `255`. Необязательный параметр `a` задает степень прозрачности цвета. Значение `0` соответствует прозрачному цвету, а значение `255` — полностью непрозрачному. Прототипы метода:

  ```
  void setRgb(int r, int g, int b, int a = 255)
  void setRgb(QRgb rgb)
  ```

- `setRgbF()` — задает целочисленные значения красной, зеленой и синей составляющих цвета модели RGB. В качестве параметров указываются вещественные числа от `0.0` до `1.0`. Необязательный параметр `a` задает степень прозрачности цвета. Значение `0.0` соответствует прозрачному цвету, а значение `1.0` — полностью непрозрачному. Прототип метода:

  ```
  void setRgbF(float r, float g, float b, float a = 1.0)
  ```

- `setRed()`, `setGreen()`, `setBlue()` и `setAlpha()` — задают значения отдельных составляющих цвета. В качестве параметров указываются числа от `0` до `255`. Прототипы методов:

```
void setRed(int red)
void setGreen(int green)
void setBlue(int blue)
void setAlpha(int alpha)
```

- `setRedF()`, `setGreenF()`, `setBlueF()` и `setAlphaF()` — задают значения отдельных составляющих цвета. В качестве параметров указываются вещественные числа от `0.0` до `1.0`. Прототипы методов:

```
void setRedF(float red)
void setGreenF(float green)
void setBlueF(float blue)
void setAlphaF(float alpha)
```

- `fromRgb()` — возвращает экземпляр класса `QColor` с указанными значениями. В качестве параметров указываются числа от `0` до `255`. Метод является статическим. Прототипы метода:

```
static QColor fromRgb(int r, int g, int b, int a = 255)
static QColor fromRgb(QRgb rgb)
```

  Пример:

```
QColor white = QColor::fromRgb(255, 255, 255, 255);
```

- `fromRgbF()` — возвращает экземпляр класса `QColor` с указанными значениями. В качестве параметров указываются вещественные числа от `0.0` до `1.0`. Метод является статическим. Прототип метода:

```
static QColor fromRgbF(float r, float g, float b, float a = 1.0)
```

- `getRgb()` — позволяет получить целочисленные значения составляющих цвета. Прототип метода:

```
void getRgb(int *r, int *g, int *b, int *a = nullptr) const
```

- `getRgbF()` — позволяет получить вещественные значения составляющих цвета. Прототип метода:

```
void getRgbF(float *r, float *g, float *b,
             float *a = nullptr) const
```

- `red()`, `green()`, `blue()` и `alpha()` — возвращают целочисленные значения отдельных составляющих цвета. Прототипы методов:

```
int red() const
int green() const
int blue() const
int alpha() const
```

- `redF()`, `greenF()`, `blueF()` и `alphaF()` — возвращают вещественные значения отдельных составляющих цвета. Прототипы методов:

```
float redF() const
float greenF() const
```

```
float blueF() const
float alphaF() const
```

- `lighter()` — если параметр имеет значение больше `100`, то возвращает новый объект с более светлым цветом, а если меньше `100`, то с более темным. Прототип метода:

```
QColor lighter(int factor = 150) const
```

- `darker()` — если параметр имеет значение больше `100`, то возвращает новый объект с более темным цветом, а если меньше `100`, то с более светлым. Прототип метода:

```
QColor darker(int factor = 200) const
```

Задать или получить значения в цветовой модели CMYK (cyan, magenta, yellow, key; голубой, пурпурный, желтый, «ключевой» (черный)) позволяют следующие методы:

- `setCmyk()` — задает целочисленные значения составляющих цвета модели CMYK. В качестве параметров указываются числа от `0` до `255`. Необязательный параметр `a` задает степень прозрачности цвета. Значение `0` соответствует прозрачному цвету, а значение `255` — полностью непрозрачному. Прототип метода:

```
void setCmyk(int c, int m, int y, int k, int a = 255)
```

- `fromCmyk()` — возвращает экземпляр класса `QColor` с указанными значениями. В качестве параметров задаются числа от `0` до `255`. Метод является статическим. Прототип метода:

```
static QColor fromCmyk(int c, int m, int y, int k, int a = 255)
```

  Пример:

```
QColor white = QColor::fromCmyk(0, 0, 0, 0, 255);
```

- `getCmyk()` — позволяет получить значения отдельных составляющих цвета. Прототип метода:

```
void getCmyk(int *c, int *m, int *y, int *k,
             int *a = nullptr) const
```

- `cyan()`, `magenta()`, `yellow()`, `black()` и `alpha()` — возвращают целочисленные значения отдельных составляющих цвета. Прототипы методов:

```
int cyan() const
int magenta() const
int yellow() const
int black() const
int alpha() const
```

- `setCmykF()` — задает значения составляющих цвета модели CMYK. В качестве параметров указываются вещественные числа от `0.0` до `1.0`. Необязательный параметр `a` задает степень прозрачности цвета. Значение `0.0` соответствует прозрачному цвету, а значение `1.0` — полностью непрозрачному. Прототип метода:

```
void setCmykF(float c, float m, float y, float k, float a = 1.0)
```

- `fromCmykF()` — возвращает экземпляр класса `QColor` с указанными значениями. В качестве параметров задаются вещественные числа от `0.0` до `1.0`. Метод является статическим. Прототип метода:

```
static QColor fromCmykF(float c, float m, float y, float k, float a = 1.0)
```

- `getCmykF()` — позволяет получить значения отдельных составляющих цвета. Прототип метода:

```
void getCmykF(float *c, float *m, float *y, float *k,
              float *a = nullptr) const
```

- `cyanF()`, `magentaF()`, `yellowF()`, `blackF()` и `alphaF()` — возвращают вещественные значения отдельных составляющих цвета. Прототипы методов:

```
float cyanF() const
float magentaF() const
float yellowF() const
float blackF() const
float alphaF() const
```

Задать или получить значения в цветовой модели HSV (hue, saturation, value; оттенок, насыщенность, значение (яркость)) позволяют следующие методы:

- `setHsv()` — задает целочисленные значения составляющих цвета модели HSV. В первом параметре указывается число от `0` до `359`, а в остальных параметрах — числа от `0` до `255`. Прототип метода:

```
void setHsv(int h, int s, int v, int a = 255)
```

- `fromHsv()` — возвращает экземпляр класса `QColor` с указанными значениями. Метод является статическим. Прототип метода:

```
static QColor fromHsv(int h, int s, int v, int a = 255)
```

  Пример:

```
QColor white = QColor::fromHsv(0, 0, 255, 255);
```

- `getHsv()` — позволяет получить значения отдельных составляющих цвета. Прототип метода:

```
void getHsv(int *h, int *s, int *v, int *a = nullptr) const
```

- `hsvHue()`, `hsvSaturation()`, `value()` и `alpha()` — возвращают целочисленные значения отдельных составляющих цвета. Прототипы методов:

```
int hsvHue() const
int hsvSaturation() const
int value() const
int alpha() const
```

- `setHsvF()` — задает значения составляющих цвета модели HSV. В качестве параметров указываются вещественные числа от `0.0` до `1.0`. Прототип метода:

```
void setHsvF(float h, float s, float v, float a = 1.0)
```

- `fromHsvF()` — возвращает экземпляр класса `QColor` с указанными значениями. В качестве параметров задаются вещественные числа от `0.0` до `1.0`. Метод является статическим. Прототип метода:

```
static QColor fromHsvF(float h, float s, float v, float a = 1.0)
```

- `getHsvF()` — позволяет получить значения отдельных составляющих цвета. Прототип метода:

```
void getHsvF(float *h, float *s, float *v,
             float *a = nullptr) const
```

- `hsvHueF()`, `hsvSaturationF()`, `valueF()` и `alphaF()` — возвращают вещественные значения отдельных составляющих цвета. Прототипы методов:

```
float hsvHueF() const
float hsvSaturationF() const
float valueF() const
float alphaF() const
```

Цветовая модель HSL (hue, saturation, lightness) отличается от модели HSV только последней составляющей. Описание этой модели и полный перечень методов для установки и получения значений смотрите в документации.

Для получения типа используемой модели и преобразования между моделями предназначены следующие методы:

- `spec()` — позволяет узнать тип используемой модели. Возвращает значение одной из следующих констант: `Invalid` (`0`), `Rgb` (`1`), `ExtendedRgb` (`5`), `Hsv` (`2`), `Cmyk` (`3`) или `Hsl` (`4`). Прототип метода:

```
QColor::Spec spec() const
```

- `convertTo()` — преобразует тип модели. В качестве параметра указываются константы, которые перечислены в описании метода `spec()`. Метод возвращает новый объект. Прототип метода:

```
QColor convertTo(QColor::Spec colorSpec) const
```

  Пример преобразования:

```
QColor whiteHSV = QColor::fromHsv(0, 0, 255);
QColor whiteRGB = whiteHSV.convertTo(QColor::Rgb);
```

  Вместо метода `convertTo()` удобнее воспользоваться методами `toRgb()`, `toExtendedRgb()`, `toCmyk()`, `toHsv()` или `toHsl()`, которые возвращают новый объект. Прототипы методов:

```
QColor toRgb() const
QColor toExtendedRgb() const
QColor toCmyk() const
QColor toHsv() const
QColor toHsl() const
```

8.1.2. Класс *QPen:* перо

Класс `QPen` описывает перо, с помощью которого производится рисование точек, линий и контуров фигур. Форматы конструктора класса:

```
#include <QPen>
QPen()
QPen(const QColor &color)
QPen(Qt::PenStyle style)
QPen(const QBrush &brush, qreal width,
     Qt::PenStyle style = Qt::SolidLine,
     Qt::PenCapStyle cap = Qt::SquareCap,
     Qt::PenJoinStyle join = Qt::BevelJoin)
QPen(const QPen &pen)
QPen(QPen &&pen)
```

Первый конструктор создает перо черного цвета с настройками по умолчанию. Второй конструктор задает только цвет пера с помощью экземпляра класса `QColor`. Третий конструктор позволяет указать стиль линии. В качестве значения указываются следующие константы:

- `Qt::NoPen` — линия не выводится;
- `Qt::SolidLine` — сплошная линия;
- `Qt::DashLine` — штриховая линия;
- `Qt::DotLine` — пунктирная линия;
- `Qt::DashDotLine` — штрих и точка, штрих и точка и т. д.;
- `Qt::DashDotDotLine` — штрих и две точки, штрих и две точки и т. д.;
- `Qt::CustomDashLine` — пользовательский стиль.

Четвертый конструктор позволяет задать все характеристики пера за один раз. В первом параметре указывается экземпляр класса `QBrush` или класса `QColor`. Ширина линии передается во втором параметре, а стиль линии в необязательном параметре `style`. Необязательный параметр `cap` задает стиль концов линии. В качестве значения указываются следующие константы:

- `Qt::FlatCap` — квадратный конец линии. Длина линии не превышает указанных граничных точек;
- `Qt::SquareCap` — квадратный конец линии. Длина линии увеличивается с обоих концов на половину ширины линии;
- `Qt::RoundCap` — скругленные концы. Длина линии увеличивается с обоих концов на половину ширины линии.

Необязательный параметр `join` задает стиль перехода одной линии в другую. В качестве значения указываются следующие константы:

- `Qt::MiterJoin` — линии соединяются под острым углом;
- `Qt::BevelJoin` — пространство между концами линий заполняется цветом линии;

- `Qt::RoundJoin` — скругленные углы;
- `Qt::SvgMiterJoin`.

Класс `QPen` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setColor()` — задает цвет линии. Прототип метода:

```
void setColor(const QColor &color)
```

- `setBrush()` — задает кисть. Прототип метода:

```
void setBrush(const QBrush &brush)
```

- `setWidth()` и `setWidthF()` — задают ширину линии. Прототипы методов:

```
void setWidth(int width)
void setWidthF(qreal width)
```

- `setStyle()` — задает стиль линии (см. значения параметра `style` в третьем формате конструктора класса `QPen`). Прототип метода:

```
void setStyle(Qt::PenStyle style)
```

- `setCapStyle()` — задает стиль концов линии (см. значения параметра `cap` в четвертом формате конструктора класса `QPen`). Прототип метода:

```
void setCapStyle(Qt::PenCapStyle style)
```

- `setJoinStyle()` — задает стиль перехода одной линии в другую (см. значения параметра `join` в четвертом формате конструктора класса `QPen`). Прототип метода:

```
void setJoinStyle(Qt::PenJoinStyle style)
```

8.1.3. Класс *QBrush:* кисть

Класс `QBrush` описывает кисть, с помощью которой производится заполнение фона фигур. Форматы конструктора класса:

```
#include <QBrush>
QBrush()
QBrush(const QColor &color, Qt::BrushStyle style = Qt::SolidPattern)
QBrush(Qt::GlobalColor color, Qt::BrushStyle style = Qt::SolidPattern)
QBrush(Qt::BrushStyle style)
QBrush(const QGradient &gradient)
QBrush(const QColor &color, const QPixmap &pixmap)
QBrush(Qt::GlobalColor color, const QPixmap &pixmap)
QBrush(const QPixmap &pixmap)
QBrush(const QImage &image)
QBrush(const QBrush &other)
```

Параметр `color` задает цвет кисти. Можно передать экземпляр класса `QColor` или константу, например `Qt::black`.

В параметре style указываются константы, задающие стиль кисти, например: Qt::NoBrush, Qt::SolidPattern, Qt::Dense1Pattern, Qt::Dense2Pattern, Qt::Dense3Pattern, Qt::Dense4Pattern, Qt::Dense5Pattern, Qt::Dense6Pattern, Qt::Dense7Pattern, Qt::CrossPattern и др. С помощью этого параметра можно сделать цвет сплошным (SolidPattern) или имеющим текстуру (например, константа CrossPattern задает текстуру в виде сетки).

Параметр gradient позволяет установить градиентную заливку. В качестве значения указываются экземпляры классов QLinearGradient (линейный градиент), QConicalGradient (конический градиент) или QRadialGradient (радиальный градиент). За подробной информацией по этим классам обращайтесь к документации.

Параметры pixmap и image предназначены для установки изображения в качестве текстуры.

Класс QBrush содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- setColor() — задает цвет кисти. Прототипы метода:

```
void setColor(const QColor &color)
void setColor(Qt::GlobalColor color)
```

- setStyle() — задает стиль кисти (см. значения параметра style в конструкторе класса QBrush). Прототип метода:

```
void setStyle(Qt::BrushStyle style)
```

- setTexture() — устанавливает растровое изображение. В качестве параметра можно указать экземпляр классов QPixmap или QBitmap. Прототип метода:

```
void setTexture(const QPixmap &pixmap)
```

- setTextureImage() — устанавливает изображение. Прототип метода:

```
void setTextureImage(const QImage &image)
```

8.1.4. Класс *QLine:* линия

Класс QLine описывает координаты линии. Форматы конструктора класса:

```
#include <QLine>
QLine()
QLine(const QPoint &p1, const QPoint &p2)
QLine(int x1, int y1, int x2, int y2)
```

Первый конструктор создает нулевой объект. Во втором и третьем конструкторах указываются координаты начальной и конечной точек в виде экземпляров класса QPoint или значений через запятую.

> ***Примечание***
>
> Класс QLine предназначен для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать класс QLineF.

Класс `QLine` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `isNull()` — возвращает значение `true`, если начальная или конечная точка не установлена, и `false` — в противном случае. Прототип метода:

```
bool isNull() const
```

- `setPoints()` — задает координаты начальной и конечной точек в виде экземпляров класса `QPoint`. Прототип метода:

```
void setPoints(const QPoint &p1, const QPoint &p2)
```

- `setLine()` — задает координаты начальной и конечной точек в виде значений через запятую. Прототип метода:

```
void setLine(int x1, int y1, int x2, int y2)
```

- `setP1()` — задает координаты начальной точки. Прототип метода:

```
void setP1(const QPoint &p1)
```

- `setP2()` — задает координаты конечной точки. Прототип метода:

```
void setP2(const QPoint &p2)
```

- `p1()` — возвращает координаты (экземпляр класса `QPoint`) начальной точки. Прототип метода:

```
QPoint p1() const
```

- `p2()` — возвращает координаты (экземпляр класса `QPoint`) конечной точки. Прототип метода:

```
QPoint p2() const
```

- `x1()`, `y1()`, `x2()` и `y2()` — возвращают значения отдельных составляющих координат начальной и конечной точек в виде целых чисел. Прототипы методов:

```
int x1() const
int x2() const
int y1() const
int y2() const
```

- `dx()` — возвращает горизонтальную составляющую вектора линии. Прототип метода:

```
int dx() const
```

- `dy()` — возвращает вертикальную составляющую вектора линии. Прототип метода:

```
int dy() const
```

8.1.5. Класс *QPolygon:* многоугольник

Класс `QPolygon` описывает координаты вершин многоугольника. Форматы конструктора класса:

```
#include <QPolygon>
QPolygon()
QPolygon(const QList<QPoint> &points)
QPolygon(const QRect &rectangle, bool closed = false)
```

Первый конструктор создает пустой объект. Заполнить объект координатами вершин можно с помощью оператора <<. Пример добавления координат вершин треугольника:

```
QPolygon polygon;
polygon << QPoint(20, 20) << QPoint(120, 20) << QPoint(20, 120);
```

Во втором конструкторе указывается список с экземплярами класса `QPoint`, которые задают координаты отдельных вершин. Пример:

```
QList<QPoint> list;
list << QPoint(180, 20) << QPoint(280, 20) << QPoint(280, 120);
QPolygon polygon2(list);
```

Третий конструктор создает многоугольник на основе экземпляра класса `QRect`. Если параметр `closed` имеет значение `false`, то будут созданы четыре вершины, а если значение `true`, то пять вершин (образуют замкнутый контур).

> ***Примечание***
>
> Класс `QPolygon` предназначен для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать класс `QPolygonF`.

Класс `QPolygon` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setPoints()` — устанавливает координаты вершин. Ранее установленные значения удаляются. Количество устанавливаемых точек задается параметром `nPoints`. Прототипы метода:

  ```
  void setPoints(int nPoints, const int *points)
  void setPoints(int nPoints, int firstx, int firsty, ...)
  ```

- `setPoint()` — задает координаты для вершины с указанным индексом. Прототипы метода:

  ```
  void setPoint(int index, int x, int y)
  void setPoint(int index, const QPoint &point)
  ```

- `point()` — возвращает экземпляр класса `QPoint` с координатами вершины, индекс которой указан в параметре. Получить координаты можно также, передав адреса переменных в качестве параметров. Прототипы метода:

  ```
  QPoint point(int index) const
  void point(int index, int *x, int *y) const
  ```

- `boundingRect()` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, в которую вписан многоугольник. Прототип метода:

  ```
  QRect boundingRect() const
  ```

8.1.6. Класс *QFont:* шрифт

Класс `QFont` описывает характеристики шрифта. Форматы конструктора класса:

```
#include <QFont>
QFont()
QFont(const QStringList &families, int pointSize = -1, int weight = -1,
      bool italic = false)
QFont(const QFont &font)
QFont(const QFont &font, const QPaintDevice *pd)
```

Первый конструктор создает объект шрифта с настройками, используемыми приложением по умолчанию. Установить шрифт приложения по умолчанию позволяет статический метод `setFont()` из класса `QApplication`. Прототип метода:

```
static void setFont(const QFont &font, const char *className = nullptr)
```

Получить значение позволяет статический метод `font()` из класса `QApplication`:

```
static QFont font()
static QFont font(const QWidget *widget)
static QFont font(const char *className)
```

Второй конструктор позволяет указать основные характеристики шрифта. В первом параметре указывается название шрифта или семейства в виде строки. Необязательный параметр `pointSize` задает размер шрифта. В параметре `weight` можно указать степень жирности шрифта: значение констант `QFont::Light`, `QFont::Normal`, `QFont::DemiBold`, `QFont::Bold` или `QFont::Black`. Если в параметре `italic` указано значение `true`, то шрифт будет курсивным.

Класс `QFont` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setFamily()` — задает название шрифта или семейства. Прототип метода:

  ```
  void setFamily(const QString &family)
  ```

- `family()` — возвращает название шрифта. Прототип метода:

  ```
  QString family() const
  ```

- `setPointSize()` и `setPointSizeF()` — задают размер шрифта в пунктах. Прототипы методов:

  ```
  void setPointSize(int pointSize)
  void setPointSizeF(qreal pointSize)
  ```

- `pointSize()` — возвращает размер шрифта в пунктах в виде целого числа или значение `-1`, если размер шрифта был установлен в пикселах. Прототип метода:

  ```
  int pointSize() const
  ```

- `pointSizeF()` — возвращает размер шрифта в пунктах в виде вещественного числа или значение `-1`, если размер шрифта был установлен в пикселах. Прототип метода:

  ```
  qreal pointSizeF() const
  ```

- `setPixelSize()` — задает размер шрифта в пикселах. Прототип метода:

  ```
  void setPixelSize(int pixelSize)
  ```

- `pixelSize()` — возвращает размер шрифта в пикселах или значение `-1`, если размер шрифта был установлен в пунктах. Прототип метода:

  ```
  int pixelSize() const
  ```

- `setWeight()` — задает степень жирности шрифта: значение констант `QFont::Light`, `QFont::Normal`, `QFont::DemiBold`, `QFont::Bold` или `QFont::Black`. Прототип метода:

  ```
  void setWeight(QFont::Weight weight)
  ```

- `weight()` — возвращает степень жирности шрифта. Прототип метода:

  ```
  QFont::Weight weight() const
  ```

- `setBold()` — если в качестве параметра указано значение `true`, то жирность шрифта устанавливается равной значению константы `Bold`, а если `false`, то равной значению константы `Normal`. Прототип метода:

  ```
  void setBold(bool enable)
  ```

- `bold()` — возвращает значение `true`, если степень жирности шрифта больше значения константы `Normal`, и `false` — в противном случае. Прототип метода:

  ```
  bool bold() const
  ```

- `setItalic()` — если в качестве параметра указано значение `true`, то шрифт будет курсивным, а если `false`, то нормальным. Прототип метода:

  ```
  void setItalic(bool enable)
  ```

- `italic()` — возвращает значение `true`, если шрифт курсивный, и `false` — в противном случае. Прототип метода:

  ```
  bool italic() const
  ```

- `setUnderline()` — если в качестве параметра указано значение `true`, то текст будет подчеркнутым, а если `false`, то не подчеркнутым. Прототип метода:

  ```
  void setUnderline(bool enable)
  ```

- `underline()` — возвращает значение `true`, если текст будет подчеркнут, и `false` — в противном случае. Прототип метода:

  ```
  bool underline() const
  ```

- `setOverline()` — если в качестве параметра указано значение `true`, то над текстом будет выводиться черта. Прототип метода:

  ```
  void setOverline(bool enable)
  ```

- `overline()` — возвращает значение `true`, если над текстом будет выводиться черта, и `false` — в противном случае. Прототип метода:

  ```
  bool overline() const
  ```

- `setStrikeOut()` — если в качестве параметра указано значение `true`, то текст будет перечеркнутым. Прототип метода:

  ```
  void setStrikeOut(bool enable)
  ```

- `strikeOut()` — возвращает значение `true`, если текст будет перечеркнутым, и `false` — в противном случае. Прототип метода:

  ```
  bool strikeOut() const
  ```

Получить список всех доступных в системе шрифтов позволяет статический метод `families()` из класса `QFontDatabase`. Метод возвращает список строк. Прототип метода:

```
static QStringList families(QFontDatabase::WritingSystem writingSystem = Any)
```

Пример:

```
// #include <QFontDatabase>
qDebug() << QFontDatabase::families();
```

Чтобы получить список доступных стилей для указанного шрифта, следует воспользоваться статическим методом `styles()` из класса `QFontDatabase`. Прототип метода:

```
static QStringList styles(const QString &family)
```

Пример:

```
qDebug() << QFontDatabase::styles("Tahoma");
// QList("Обычный", "Полужирный")
```

Получить допустимые размеры для указанного стиля можно с помощью статического метода метода `smoothSizes()` из класса `QFontDatabase`. Прототип метода:

```
static QList<int> smoothSizes(const QString &family,
                              const QString &styleName)
```

Пример:

```
qDebug() << QFontDatabase::smoothSizes("Tahoma", "Обычный");
// QList(6, 7, 8, 9, 10, 11, 12, 14, 16, 18, 20, 22, 24, 26,
// 28, 36, 48, 72)
```

Очень часто необходимо произвести выравнивание выводимого текста внутри некоторой области. Чтобы это сделать, нужно знать размеры области, в которую вписан текст. Получить эти значения позволяют следующие методы из класса `QFontMetrics`:

- `height()` — возвращает высоту шрифта. Прототип метода:

  ```
  int height() const
  ```

- `boundingRect()` — возвращает экземпляр класса `QRect` с координатами и размерами прямоугольной области, в которую вписан текст. Прототипы метода:

  ```
  QRect boundingRect(QChar ch) const
  QRect boundingRect(const QString &text) const
  ```

```
QRect boundingRect(const QRect &rect, int flags,
      const QString &text, int tabStops = 0,
      int *tabArray = nullptr) const
QRect boundingRect(int x, int y, int width, int height,
      int flags, const QString &text, int tabStops = 0,
      int *tabArray = nullptr) const
```

Пример получения размеров области:

```
QFont font("Tahoma", 16);
QFontMetrics fm(font);
qDebug() << fm.height();                // 25
qDebug() << fm.boundingRect("Строка"); // QRect(0,-21 65x25)
```

Примечание

Класс `QFontMetrics` предназначен для работы с целыми числами. Чтобы работать с вещественными числами, необходимо использовать класс `QFontMetricsF`.

8.2. Класс *QPainter*

Класс `QPainter` содержит все необходимые средства, позволяющие выполнять рисование геометрических фигур и текста на поверхности, которая реализуется классом `QPaintDevice`. Класс `QPaintDevice` наследуют классы `QWidget`, `QPicture`, `QPixmap`, `QImage`, `QPrinter` и некоторые другие. Таким образом, мы можем рисовать как на поверхности любого компонента, так и на изображении. Форматы конструктора класса:

```
#include <QPainter>
QPainter()
QPainter(QPaintDevice *device)
```

Первый конструктор создает объект, который не подключен ни к одному устройству. Чтобы подключиться к устройству и захватить контекст рисования, необходимо вызвать метод `begin()` и передать ему указатель на экземпляр класса, являющегося наследником класса `QPaintDevice`. Метод возвращает значение `true`, если контекст успешно захвачен, и `false` — в противном случае. Прототип метода:

```
bool begin(QPaintDevice *device)
```

В один момент времени только один объект может рисовать на устройстве, поэтому после окончания рисования необходимо освободить контекст рисования с помощью метода `end()`. Прототип метода:

```
bool end()
```

С учетом сказанного код, позволяющий рисовать на компоненте, будет выглядеть так:

```
void Widget::paintEvent(QPaintEvent *e)
{
   QPainter painter;
   painter.begin(this);
```

```
    // Здесь производим рисование на компоненте
    painter.end();
}
```

Второй конструктор принимает указатель на экземпляр класса, являющегося наследником класса `QPaintDevice`, подключается к этому устройству и сразу захватывает контекст рисования. Контекст рисования автоматически освобождается внутри деструктора класса `QPainter` при уничтожении объекта. Так как объект автоматически уничтожается при выходе из метода `paintEvent()`, то метод `end()` можно и не вызывать. Пример рисования на компоненте:

```
void Widget::paintEvent(QPaintEvent *e)
{
    QPainter painter(this);
    // Здесь производим рисование на компоненте
}
```

Проверить успешность захвата контекста рисования можно с помощью метода `isActive()`. Метод возвращает значение `true`, если контекст захвачен, и `false` — в противном случае. Прототип метода:

```
bool isActive() const
```

8.2.1. Рисование линий и фигур

После захвата контекста рисования следует установить перо и кисть. С помощью пера производится рисование точек, линий и контуров фигур, а с помощью кисти — заполнение фона фигур. Установить перо позволяет метод `setPen()` из класса `QPainter`. Прототипы метода:

```
void setPen(const QPen &pen)
void setPen(const QColor &color)
void setPen(Qt::PenStyle style)
```

Для установки кисти предназначен метод `setBrush()`. Прототипы метода:

```
void setBrush(const QBrush &brush)
void setBrush(Qt::BrushStyle style)
```

Устанавливать перо или кисть нужно будет перед каждой операцией рисования, требующей изменения цвета или стиля. Если перо или кисть не установлены, то будут использоваться объекты с настройками по умолчанию. После установки пера и кисти можно приступать к рисованию точек, линий, фигур, текста и др.

Для рисования точек, линий и фигур класс `QPainter` предоставляет следующие методы (перечислены только основные методы; полный список смотрите в документации по классу `QPainter`):

- `drawPoint()` — рисует точку. Прототипы метода:

```
void drawPoint(int x, int y)
void drawPoint(const QPoint &position)
void drawPoint(const QPointF &position)
```

- `drawPoints()` — рисует несколько точек. Прототипы метода:

```
void drawPoints(const QPoint *points, int pointCount)
void drawPoints(const QPointF *points, int pointCount)
void drawPoints(const QPolygon &points)
void drawPoints(const QPolygonF &points)
```

- `drawLine()` — рисует линию. Прототипы метода:

```
void drawLine(const QLine &line)
void drawLine(const QLineF &line)
void drawLine(const QPoint &p1, const QPoint &p2)
void drawLine(const QPointF &p1, const QPointF &p2)
void drawLine(int x1, int y1, int x2, int y2)
```

- `drawLines()` — рисует несколько отдельных линий. Прототипы метода:

```
void drawLines(const QList<QLine> &lines)
void drawLines(const QList<QLineF> &lines)
void drawLines(const QList<QPoint> &pointPairs)
void drawLines(const QList<QPointF> &pointPairs)
void drawLines(const QLine *lines, int lineCount)
void drawLines(const QLineF *lines, int lineCount)
void drawLines(const QPoint *pointPairs, int lineCount)
void drawLines(const QPointF *pointPairs, int lineCount)
```

- `drawPolyline()` — рисует несколько линий, которые соединяют указанные точки. Первая и последняя точки не соединяются. Прототипы метода:

```
void drawPolyline(const QPoint *points, int pointCount)
void drawPolyline(const QPointF *points, int pointCount)
void drawPolyline(const QPolygon &points)
void drawPolyline(const QPolygonF &points)
```

- `drawRect()` — рисует прямоугольник с границей и заливкой. Чтобы убрать границу, следует использовать перо со стилем `NoPen`, а чтобы убрать заливку — кисть со стилем `NoBrush`. Прототипы метода:

```
void drawRect(int x, int y, int width, int height)
void drawRect(const QRect &rectangle)
void drawRect(const QRectF &rectangle)
```

- `drawRects()` — рисует несколько прямоугольников. Прототипы метода:

```
void drawRects(const QRect *rectangles, int rectCount)
void drawRects(const QRectF *rectangles, int rectCount)
void drawRects(const QList<QRect> &rectangles)
void drawRects(const QList<QRectF> &rectangles)
```

- `fillRect()` — рисует прямоугольник с заливкой без границы. Прототипы метода:

```
void fillRect(int x, int y, int width, int height,
              const QColor &color)
```

```
void fillRect(int x, int y, int width, int height,
              Qt::GlobalColor color)
void fillRect(int x, int y, int width, int height,
              const QBrush &brush)
void fillRect(int x, int y, int width, int height,
              Qt::BrushStyle style)
void fillRect(int x, int y, int width, int height,
              QGradient::Preset preset)
void fillRect(const QRect &rectangle, const QColor &color)
void fillRect(const QRect &rectangle, Qt::GlobalColor color)
void fillRect(const QRect &rectangle, const QBrush &brush)
void fillRect(const QRect &rectangle, Qt::BrushStyle style)
void fillRect(const QRect &rectangle, QGradient::Preset preset)
void fillRect(const QRectF &rectangle, const QColor &color)
void fillRect(const QRectF &rectangle, Qt::GlobalColor color)
void fillRect(const QRectF &rectangle, const QBrush &brush)
void fillRect(const QRectF &rectangle, Qt::BrushStyle style)
void fillRect(const QRectF &rectangle, QGradient::Preset preset)
```

- `drawRoundedRect()` — рисует прямоугольник с границей, заливкой и скругленными краями. Прототипы метода:

```
void drawRoundedRect(int x, int y, int w, int h, qreal xRadius,
     qreal yRadius, Qt::SizeMode mode = Qt::AbsoluteSize)
void drawRoundedRect(const QRect &rect, qreal xRadius,
     qreal yRadius, Qt::SizeMode mode = Qt::AbsoluteSize)
void drawRoundedRect(const QRectF &rect, qreal xRadius,
     qreal yRadius, Qt::SizeMode mode = Qt::AbsoluteSize)
```

- `drawPolygon()` — рисует многоугольник с границей и заливкой. Прототипы метода:

```
void drawPolygon(const QPoint *points, int pointCount,
                 Qt::FillRule fillRule = Qt::OddEvenFill)
void drawPolygon(const QPointF *points, int pointCount,
                 Qt::FillRule fillRule = Qt::OddEvenFill)
void drawPolygon(const QPolygon &points,
                 Qt::FillRule fillRule = Qt::OddEvenFill)
void drawPolygon(const QPolygonF &points,
                 Qt::FillRule fillRule = Qt::OddEvenFill)
```

- `drawEllipse()` — рисует эллипс с границей и заливкой. Прототипы метода:

```
void drawEllipse(int x, int y, int width, int height)
void drawEllipse(const QRect &rectangle)
void drawEllipse(const QRectF &rectangle)
void drawEllipse(const QPoint &center, int rx, int ry)
void drawEllipse(const QPointF &center, qreal rx, qreal ry)
```

В первых трех форматах указываются координаты и размеры прямоугольника, в который необходимо вписать эллипс. В двух последних форматах первый

параметр задает координаты центра, параметр `rx` — радиус по оси `X`, а параметр `ry` — радиус по оси `Y`;

- `drawArc()` — рисует дугу. Прототипы метода:

```
void drawArc(int x, int y, int width, int height, int startAngle,
             int spanAngle)
void drawArc(const QRect &rectangle, int startAngle, int spanAngle)
void drawArc(const QRectF &rectangle, int startAngle, int spanAngle)
```

  Следует учитывать, что значения углов задаются в значениях `1/16` градуса. Полный круг эквивалентен значению `16 * 360`. Нулевой угол находится в позиции трех часов. Положительные значения углов отсчитываются против часовой стрелки, а отрицательные — по часовой стрелке;

- `drawChord()` — рисует замкнутую дугу. Метод `drawChord()` аналогичен методу `drawArc()`, но замыкает крайние точки дуги прямой линией. Прототипы метода:

```
void drawChord(int x, int y, int width, int height,
               int startAngle, int spanAngle)
void drawChord(const QRect &rectangle, int startAngle, int spanAngle)
void drawChord(const QRectF &rectangle, int startAngle, int spanAngle)
```

- `drawPie()` — рисует замкнутый сектор. Метод `drawPie()` аналогичен методу `drawArc()`, но замыкает крайние точки дуги с центром окружности. Прототипы метода:

```
void drawPie(int x, int y, int width, int height,
             int startAngle, int spanAngle)
void drawPie(const QRect &rectangle, int startAngle, int spanAngle)
void drawPie(const QRectF &rectangle, int startAngle, int spanAngle)
```

При выводе некоторых фигур (например, эллипса) контур может отображаться в виде «лесенки». Чтобы сгладить контуры фигур, следует вызвать метод `setRenderHint()` и передать ему константу `Antialiasing`. Прототип метода:

```
void setRenderHint(QPainter::RenderHint hint, bool on = true)
```

Пример:

```
painter.setRenderHint(QPainter::Antialiasing);
```

8.2.2. Вывод текста

Вывести текст позволяет метод `drawText()` из класса `QPainter`. Прототипы метода:

```
void drawText(int x, int y, const QString &text)
void drawText(const QPoint &position, const QString &text)
void drawText(const QPointF &position, const QString &text)
void drawText(int x, int y, int width, int height, int flags,
              const QString &text, QRect *boundingRect = nullptr)
void drawText(const QRect &rectangle, int flags, const QString &text,
              QRect *boundingRect = nullptr)
```

```
void drawText(const QRectF &rectangle, int flags, const QString &text,
              QRectF *boundingRect = nullptr)
void drawText(const QRectF &rectangle, const QString &text,
              const QTextOption &option = QTextOption())
```

Первые три формата метода выводят текст, начиная с указанных координат.

Следующие три формата выводят текст в указанную прямоугольную область. Если текст не помещается в прямоугольную область, то он будет обрезан, если не указан флаг `TextDontClip`. В параметре `flags` через оператор `|` указываются константы `AlignLeft`, `AlignRight`, `AlignHCenter`, `AlignTop`, `AlignBottom`, `AlignVCenter` или `AlignCenter`, задающие выравнивание текста внутри прямоугольной области, а также следующие константы:

- `Qt::TextDontClip` — текст может выйти за пределы указанной прямоугольной области;
- `Qt::TextSingleLine` — все специальные символы трактуются как пробелы, и текст выводится в одну строку;
- `Qt::TextWordWrap` — если текст не помещается на одной строке, то будет произведен перенос по границам слова;
- `Qt::TextShowMnemonic` — символ, перед которым указан символ `&`, будет подчеркнут. Чтобы вывести символ `&`, его необходимо удвоить;
- `Qt::TextExpandTabs` — символы табуляции будут обрабатываться.

Седьмой формат метода `drawText()` также выводит текст в указанную прямоугольную область, но выравнивание текста и другие опции задаются с помощью экземпляра класса `QTextOption`. Например, с помощью этого класса можно отобразить непечатаемые символы (символ пробела, табуляцию и др.).

Получить координаты и размеры прямоугольника, в который вписывается текст, позволяет метод `boundingRect()`. Прототипы метода:

```
QRect boundingRect(int x, int y, int w, int h, int flags,
                   const QString &text)
QRect boundingRect(const QRect &rectangle, int flags,
                   const QString &text)
QRectF boundingRect(const QRectF &rectangle, int flags,
                    const QString &text)
QRectF boundingRect(const QRectF &rectangle, const QString &text,
                    const QTextOption &option = QTextOption())
```

При выводе текста линии букв могут отображаться в виде «лесенки». Чтобы сгладить контуры, следует вызвать метод `setRenderHint()` и передать ему константу `TextAntialiasing`. Пример:

```
painter.setRenderHint(QPainter::TextAntialiasing);
```

8.2.3. Вывод изображения

Для вывода растровых изображений на рисунок предназначены методы `drawPixmap()` и `drawImage()` из класса `QPainter`. Метод `drawPixmap()` предназначен для вывода изображений, хранимых в экземпляре класса `QPixmap`. Прототипы метода `drawPixmap()`:

```
void drawPixmap(int x, int y, const QPixmap &pixmap)
void drawPixmap(const QPoint &point, const QPixmap &pixmap)
void drawPixmap(const QPointF &point, const QPixmap &pixmap)
void drawPixmap(int x, int y, int width, int height, const QPixmap &pixmap)
void drawPixmap(const QRect &rectangle, const QPixmap &pixmap)
void drawPixmap(int x, int y, const QPixmap &pixmap,
                int sx, int sy, int sw, int sh)
void drawPixmap(const QPoint &point, const QPixmap &pixmap,
                const QRect &source)
void drawPixmap(const QPointF &point, const QPixmap &pixmap,
                const QRectF &source)
void drawPixmap(int x, int y, int w, int h, const QPixmap &pixmap,
                int sx, int sy, int sw, int sh)
void drawPixmap(const QRect &target, const QPixmap &pixmap,
                const QRect &source)
void drawPixmap(const QRectF &target, const QPixmap &pixmap,
                const QRectF &source)
```

Первые три формата задают координаты, в которые будет установлен левый верхний угол изображения, и экземпляр класса `QPixmap`. Пример:

```
QPixmap pixmap("C:\\cpp\\projectsQt\\Test\\photo.jpg");
painter.drawPixmap(3, 3, pixmap);
```

Четвертый и пятый форматы позволяют ограничить вывод изображения указанной прямоугольной областью. Если размеры области не совпадают с размерами изображения, то производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

Шестой, седьмой и восьмой форматы задают координаты, в которые будет установлен левый верхний угол фрагмента изображения. Координаты и размеры вставляемого фрагмента изображения указываются после экземпляра класса `QPixmap` в виде отдельных составляющих или экземпляров классов `QRect` или `QRectF`.

Последние три формата ограничивают вывод фрагмента изображения указанной прямоугольной областью. Координаты и размеры вставляемого фрагмента изображения указываются после экземпляра класса `QPixmap` в виде отдельных составляющих или экземпляров классов `QRect` или `QRectF`. Если размеры области не совпадают с размерами фрагмента изображения, то производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

Метод `drawImage()` предназначен для вывода изображений, хранимых в экземпляре класса `QImage`. При выводе экземпляр класса `QImage` преобразуется в экземпляр

класса `QPixmap`. Тип преобразования задается с помощью необязательного параметра `flags`. Прототипы метода `drawImage()`:

```
void drawImage(const QPoint &point, const QImage &image)
void drawImage(const QPointF &point, const QImage &image)
void drawImage(const QRect &rectangle, const QImage &image)
void drawImage(const QRectF &rectangle, const QImage &image)
void drawImage(int x, int y, const QImage &image,
               int sx = 0, int sy = 0, int sw = -1, int sh = -1,
               Qt::ImageConversionFlags flags = Qt::AutoColor)
void drawImage(const QPoint &point, const QImage &image,
               const QRect &source,
               Qt::ImageConversionFlags flags = Qt::AutoColor)
void drawImage(const QPointF &point, const QImage &image,
               const QRectF &source,
               Qt::ImageConversionFlags flags = Qt::AutoColor)
void drawImage(const QRect &target, const QImage &image,
               const QRect &source,
               Qt::ImageConversionFlags flags = Qt::AutoColor)
void drawImage(const QRectF &target, const QImage &image,
               const QRectF &source,
               Qt::ImageConversionFlags flags = Qt::AutoColor)
```

Первые два формата (а также пятый формат со значениями по умолчанию) задают координаты, в которые будет установлен левый верхний угол изображения, и экземпляр класса `QImage`. Пример:

```
QImage img("C:\\cpp\\projectsQt\\Test\\photo.jpg");
painter.drawImage(3, 3, img);
```

Третий и четвертый форматы позволяют ограничить вывод изображения указанной прямоугольной областью. Если размеры области не совпадают с размерами изображения, то производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

Пятый, шестой и седьмой форматы задают координаты, в которые будет установлен левый верхний угол фрагмента изображения. Координаты и размеры вставляемого фрагмента изображения указываются после экземпляра класса `QImage` в виде отдельных составляющих или экземпляров классов `QRect` или `QRectF`.

Последние два формата ограничивают вывод фрагмента изображения указанной прямоугольной областью. Координаты и размеры вставляемого фрагмента изображения указываются после экземпляра класса `QImage` в виде экземпляров классов `QRect` или `QRectF`. Если размеры области не совпадают с размерами фрагмента изображения, то производится масштабирование изображения. При несоответствии пропорций изображение может быть искажено.

8.2.4. Преобразование систем координат

Существуют две системы координат: *физическая* (`viewport`; система координат устройства) и *логическая* (`window`). При рисовании координаты из логической системы координат преобразуются в систему координат устройства. По умолчанию эти две системы координат совпадают. В некоторых случаях возникает необходимость изменить координаты. Выполнить изменение физической системы координат позволяет метод `setViewport()`, а получить текущее значение можно с помощью метода `viewport()`. Прототипы методов:

```
void setViewport(int x, int y, int width, int height)
void setViewport(const QRect &rectangle)
QRect viewport() const
```

Выполнить изменение логической системы координат позволяет метод `setWindow()`, а получить текущее значение можно с помощью метода `window()`. Прототипы методов:

```
void setWindow(int x, int y, int width, int height)
void setWindow(const QRect &rectangle)
QRect window() const
```

Произвести дополнительную трансформацию системы координат позволяют следующие методы из класса `QPainter`:

- `translate()` — перемещает начало координат в указанную точку. По умолчанию начало координат находится в левом верхнем углу. Положительная ось `X` направлена вправо, а положительная ось `Y` — вниз. Прототипы метода:

  ```
  void translate(qreal dx, qreal dy)
  void translate(const QPoint &offset)
  void translate(const QPointF &offset)
  ```

- `rotate()` — поворачивает систему координат на указанное количество градусов (указывается вещественное число). Положительное значение вызывает поворот по часовой стрелке, а отрицательное значение — против часовой стрелки. Прототип метода:

  ```
  void rotate(qreal angle)
  ```

- `scale()` — масштабирует систему координат. В качестве значений указываются вещественные числа. Если значение меньше единицы, то выполняется уменьшение, а если больше единицы, то увеличение. Прототип метода:

  ```
  void scale(qreal sx, qreal sy)
  ```

- `shear()` — сдвигает систему координат. Прототип метода:

  ```
  void shear(qreal sh, qreal sv)
  ```

Все указанные трансформации влияют на последующие операции рисования и не изменяют ранее нарисованные фигуры. Чтобы после трансформации восстановить систему координат, следует предварительно сохранить состояние в стеке с помощью метода `save()`, а после окончания рисования вызвать метод `restore()`:

```
painter.save();     // Сохраняем состояние
                    // Трансформируем и рисуем
painter.restore(); // Восстанавливаем состояние
```

8.2.5. Сохранение команд рисования в файл

Класс `QPicture` исполняет роль устройства для рисования с возможностью сохранения команд рисования в файл специального формата и последующего восстановления команд. Иерархия наследования:

```
QPaintDevice — QPicture
```

Форматы конструктора класса:

```
#include <QPicture>
QPicture(int formatVersion = -1)
QPicture(const QPicture &pic)
```

Первый конструктор создает пустой рисунок. Необязательный параметр `formatVersion` задает версию формата. Если параметр не указан, то используется формат, принятый в текущей версии Qt. Второй конструктор создает копию рисунка.

Для сохранения и загрузки рисунка предназначены следующие методы:

- `save()` — сохраняет рисунок в файл. Метод возвращает значение `true`, если рисунок успешно сохранен, и `false` — в противном случае. Прототипы метода:

  ```
  bool save(const QString &fileName)
  bool save(QIODevice *dev)
  ```

- `load()` — загружает рисунок из файла. Метод возвращает значение `true`, если рисунок успешно загружен, и `false` — в противном случае. Прототипы метода:

  ```
  bool load(const QString &fileName)
  bool load(QIODevice *dev)
  ```

Для вывода загруженного рисунка на устройство рисования предназначен метод `drawPicture()` из класса `QPainter`. Прототипы метода:

```
void drawPicture(int x, int y, const QPicture &picture)
void drawPicture(const QPoint &point, const QPicture &picture)
void drawPicture(const QPointF &point, const QPicture &picture)
```

Пример сохранения рисунка:

```
QPainter painter;
QPicture pic;
painter.begin(&pic);
// Здесь что-то рисуем
painter.end();
pic.save("C:\\cpp\\projectsQt\\Test\\pic.dat");
```

Пример вывода загруженного рисунка на поверхность компонента:

```
void Widget::paintEvent(QPaintEvent *e)
{
   QPainter painter(this);
   QPicture pic;
   pic.load("C:\\cpp\\projectsQt\\Test\\pic.dat");
   painter.drawPicture(0, 0, pic);
}
```

8.3. Работа с изображениями

Библиотека Qt содержит несколько классов, позволяющих работать с растровыми изображениями в контекстно-зависимом (классы `QPixmap` и `QBitmap`) и контекстно-независимом (класс `QImage`) представлениях. Получить список форматов, которые можно загрузить, позволяет статический метод `supportedImageFormats()` из класса `QImageReader`. Прототип метода:

```
#include <QImageReader>
static QList<QByteArray> supportedImageFormats()
```

Получим список поддерживаемых форматов для чтения:

```
qDebug() << QImageReader::supportedImageFormats();
```

Результат выполнения:

```
QList("bmp", "cur", "gif", "ico", "jpeg", "jpg", "pbm", "pgm", "png",
"ppm", "svg", "svgz", "xbm", "xpm")
```

Получить список форматов, в которых можно сохранить изображение, позволяет статический метод `supportedImageFormats()` из класса `QImageWriter`. Прототип метода:

```
#include <QImageWriter>
static QList<QByteArray> supportedImageFormats()
```

Получим список поддерживаемых форматов для записи:

```
qDebug() << QImageWriter::supportedImageFormats();
```

Результат выполнения:

```
QList("bmp", "cur", "ico", "jpeg", "jpg", "pbm", "pgm", "png", "ppm",
"xbm", "xpm")
```

Обратите внимание на то, что мы можем загрузить изображение в формате GIF, но не имеем возможности сохранить изображение в этом формате, т. к. алгоритм сжатия, используемый в этом формате, защищен патентом.

8.3.1. Класс *QPixmap*

Класс `QPixmap` предназначен для работы с изображениями в контекстно-зависимом представлении. Данные хранятся в виде, позволяющем отображать изображение на

экране наиболее эффективным способом, поэтому класс часто используется в качестве буфера для рисования перед выводом на экран. Иерархия наследования:

```
QPaintDevice — QPixmap
```

Так как класс `QPixmap` наследует класс `QPaintDevice`, мы можем использовать его как поверхность для рисования. Вывести изображение на рисунок позволяет метод `drawPixmap()` из класса `QPainter` *(см. разд. 8.2.3)*.

Форматы конструктора класса:

```
#include <QPixmap>
QPixmap()
QPixmap(int width, int height)
QPixmap(const QSize &size)
QPixmap(const QString &fileName, const char *format = nullptr,
        Qt::ImageConversionFlags flags = Qt::AutoColor)
QPixmap(const QPixmap &pixmap)
QPixmap(QPixmap &&other)
QPixmap(const char *const [] xpm)
```

Первый конструктор создает нулевой объект изображения. Второй и третий конструкторы позволяют указать размеры изображения. Если размеры равны нулю, то будет создан нулевой объект изображения. Четвертый конструктор предназначен для загрузки изображения из файла, а пятый конструктор создает копию изображения.

Класс `QPixmap` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `isNull()` — возвращает значение `true`, если объект является нулевым, и `false` — в противном случае. Прототип метода:

  ```
  bool isNull() const
  ```

- `load()` — загружает изображение из файла. В первом параметре указывается абсолютный или относительный путь к файлу. Во втором параметре можно указать тип изображения. Если параметр не указан, то тип определяется автоматически. Необязательный параметр `flags` задает тип преобразования. Метод возвращает значение `true`, если изображение успешно загружено, и `false` — в противном случае. Прототип метода:

  ```
  bool load(const QString &fileName, const char *format = nullptr,
            Qt::ImageConversionFlags flags = Qt::AutoColor)
  ```

- `loadFromData()` — загружает изображение из массива или экземпляра класса `QByteArray`. Метод возвращает значение `true`, если изображение успешно загружено, и `false` — в противном случае. Прототипы метода:

  ```
  bool loadFromData(const uchar *data, uint len,
                    const char *format = nullptr,
                    Qt::ImageConversionFlags flags = Qt::AutoColor)
  ```

```
bool loadFromData(const QByteArray &data,
                  const char *format = nullptr,
                  Qt::ImageConversionFlags flags = Qt::AutoColor)
```

- `save()` — сохраняет изображение в файл. В параметре `fileName` указывается абсолютный или относительный путь к файлу. В параметре `format` можно задать тип изображения в виде строки. Если параметр не указан, то тип определяется автоматически по расширению файла. Необязательный параметр `quality` позволяет задать качество изображения. Можно передать значение в диапазоне от `0` до `100`. Метод возвращает значение `true`, если изображение успешно сохранено, и `false` — в противном случае. Прототипы метода:

```
bool save(const QString &fileName, const char *format = nullptr,
          int quality = -1) const
bool save(QIODevice *device, const char *format = nullptr,
          int quality = -1) const
```

- `convertFromImage()` — преобразует экземпляр класса `QImage` в экземпляр класса `QPixmap`. Метод возвращает значение `true`, если изображение успешно преобразовано, и `false` — в противном случае. Прототип метода:

```
bool convertFromImage(const QImage &image,
            Qt::ImageConversionFlags flags = Qt::AutoColor)
```

- `fromImage()` — преобразует экземпляр класса `QImage` в экземпляр класса `QPixmap` и возвращает его. Метод является статическим. Прототипы метода:

```
static QPixmap fromImage(const QImage &image,
               Qt::ImageConversionFlags flags = Qt::AutoColor)
static QPixmap fromImage(QImage &&image,
               Qt::ImageConversionFlags flags = Qt::AutoColor)
```

- `toImage()` — преобразует экземпляр класса `QPixmap` в экземпляр класса `QImage` и возвращает его. Прототип метода:

```
QImage toImage() const
```

- `fill()` — производит заливку изображения указанным цветом. Прототип метода:

```
void fill(const QColor &color = Qt::white)
```

- `width()` — возвращает ширину изображения. Прототип метода:

```
int width() const
```

- `height()` — возвращает высоту изображения. Прототип метода:

```
int height() const
```

- `size()` — возвращает экземпляр класса `QSize` с размерами изображения. Прототип метода:

```
QSize size() const
```

- `rect()` — возвращает экземпляр класса `QRect` с координатами и размерами и прямоугольной области, ограничивающей изображение. Прототип метода:

```
QRect rect() const
```

- `depth()` — возвращает глубину цвета. Прототип метода:

```
int depth() const
```

- `isQBitmap()` — возвращает значение `true`, если глубина цвета равна одному биту, и `false` — в противном случае. Прототип метода:

```
bool isQBitmap() const
```

- `setMask()` — устанавливает маску. Прототип метода:

```
void setMask(const QBitmap &mask)
```

- `mask()` — возвращает экземпляр класса `QBitmap` с маской изображения. Прототип метода:

```
QBitmap mask() const
```

- `copy()` — возвращает экземпляр класса `QPixmap` с фрагментом изображения. Если параметр `rect` не указан, то изображение копируется полностью. Прототипы метода:

```
QPixmap copy(int x, int y, int width, int height) const
QPixmap copy(const QRect &rect = QRect()) const
```

- `scaled()` — изменяет размер изображения и возвращает экземпляр класса `QPixmap`. Исходное изображение не изменяется. Прототипы метода:

```
QPixmap scaled(int width, int height,
        Qt::AspectRatioMode am = Qt::IgnoreAspectRatio,
        Qt::TransformationMode tm = Qt::FastTransformation) const
QPixmap scaled(const QSize &size,
        Qt::AspectRatioMode am = Qt::IgnoreAspectRatio,
        Qt::TransformationMode tm = Qt::FastTransformation) const
```

  В необязательном параметре `am` могут быть указаны следующие константы:

  - `Qt::IgnoreAspectRatio` — непосредственно изменяет размеры без сохранения пропорций сторон;
  - `Qt::KeepAspectRatio` — производится попытка масштабирования старой области внутри новой области без нарушения пропорций;
  - `Qt::KeepAspectRatioByExpanding` — производится попытка полностью заполнить новую область без нарушения пропорций старой области.

  В необязательном параметре `tm` могут быть указаны следующие константы:

  - `Qt::FastTransformation` — сглаживание выключено;
  - `Qt::SmoothTransformation` — сглаживание включено;

- `scaledToWidth()` — изменяет ширину изображения и возвращает экземпляр класса `QPixmap`. Высота изображения изменяется пропорционально. Исходное изображение не изменяется. Параметр `mode` аналогичен параметру `tm` в методе `scaled()`. Прототип метода:

```
QPixmap scaledToWidth(int width,
   Qt::TransformationMode mode = Qt::FastTransformation) const
```

- `scaledToHeight()` — изменяет высоту изображения и возвращает экземпляр класса `QPixmap`. Ширина изображения изменяется пропорционально. Исходное изображение не изменяется. Параметр `mode` аналогичен параметру `tm` в методе `scaled()`. Прототип метода:

```
QPixmap scaledToHeight(int height,
   Qt::TransformationMode mode = Qt::FastTransformation) const
```

- `transformed()` — производит трансформацию изображения (например, поворот) и возвращает экземпляр класса `QPixmap`. Исходное изображение не изменяется. Трансформация задается с помощью экземпляра класса `QTransform`. Параметр `mode` аналогичен параметру `tm` в методе `scaled()`. Прототип метода:

```
QPixmap transformed(const QTransform &transform,
   Qt::TransformationMode mode = Qt::FastTransformation) const
```

8.3.2. Класс *QBitmap*

Класс `QBitmap` предназначен для работы с изображениями, имеющими глубину цвета равной одному биту, в контекстно-зависимом представлении. Наиболее часто класс `QBitmap` используется для создания масок изображений. Иерархия наследования:

```
QPaintDevice — QPixmap — QBitmap
```

Так как класс `QBitmap` наследует класс `QPaintDevice`, мы можем использовать его как поверхность для рисования. Цвет пера и кисти задается константами `Qt::color0` (прозрачный цвет) и `Qt::color1` (непрозрачный цвет). Вывести изображение на рисунок позволяет метод `drawPixmap()` из класса `QPainter` *(см. разд. 8.2.3)*.

Форматы конструктора класса:

```
#include <QBitmap>
QBitmap()
QBitmap(int width, int height)
QBitmap(const QSize &size)
QBitmap(const QString &fileName, const char *format = nullptr)
```

Класс `QBitmap` наследует все методы из класса `QPixmap` и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `fromImage()` — преобразует экземпляр класса `QImage` в экземпляр класса `QBitmap` и возвращает его. Метод является статическим. Прототипы метода:

```
static QBitmap fromImage(const QImage &image,
       Qt::ImageConversionFlags flags = Qt::AutoColor)
static QBitmap fromImage(QImage &&image,
       Qt::ImageConversionFlags flags = Qt::AutoColor)
```

- `fromPixmap()` — преобразует экземпляр класса `QPixmap` в экземпляр класса `QBitmap` и возвращает его. Метод является статическим. Прототип метода:

```
static QBitmap fromPixmap(const QPixmap &pixmap)
```

- `transformed()` — производит трансформацию изображения (например, поворот) и возвращает экземпляр класса `QBitmap`. Исходное изображение не изменяется. Прототип метода:

```
QBitmap transformed(const QTransform &matrix) const
```

- `clear()` — устанавливает все биты изображения в значение `Qt::color0`. Прототип метода:

```
void clear()
```

8.3.3. Класс *QImage*

Класс `QImage` предназначен для работы с изображениями в контекстно-независимом представлении. Перед выводом на экран экземпляр класса `QImage` преобразуется в экземпляр класса `QPixmap`. Иерархия наследования:

```
QPaintDevice — QImage
```

Так как класс `QImage` наследует класс `QPaintDevice`, мы можем использовать его как поверхность для рисования. Однако следует учитывать, что не на всех форматах изображения можно рисовать. Для рисования лучше использовать изображение формата `Format_ARGB32_Premultiplied`. Вывести изображение на рисунок позволяет метод `drawImage()` из класса `QPainter` *(см. разд. 8.2.3)*.

Форматы конструктора класса:

```
#include <QImage>
QImage()
QImage(int width, int height, QImage::Format format)
QImage(const QSize &size, QImage::Format format)
QImage(const QString &fileName, const char *f = nullptr)
QImage(const char *const [] xpm)
QImage(uchar *data, int width, int height, qsizetype bytesPerLine,
       QImage::Format format,
       QImageCleanupFunction cleanupFunction = nullptr,
       void *cleanupInfo = nullptr)
QImage(const uchar *data, int width, int height, qsizetype bytesPerLine,
       QImage::Format format,
       QImageCleanupFunction cleanupFunction = nullptr,
       void *cleanupInfo = nullptr)
QImage(const uchar *data, int width, int height, QImage::Format format,
       QImageCleanupFunction cleanupFunction = nullptr,
       void *cleanupInfo = nullptr)
QImage(uchar *data, int width, int height, QImage::Format format,
       QImageCleanupFunction cleanupFunction = nullptr,
       void *cleanupInfo = nullptr)
```

```
QImage(const QImage &image)
QImage(QImage &&other)
```

Первый конструктор создает нулевой объект изображения. Второй и третий конструкторы позволяют указать размеры изображения. Если размеры равны нулю, то будет создан нулевой объект изображения. Четвертый конструктор предназначен для загрузки изображения из файла. Во втором параметре указывается тип изображения в виде строки. Если второй параметр не указан, то формат определяется автоматически.

В параметре `format` можно указать следующие константы:

- `QImage::Format_Invalid` — формат не определен;
- `QImage::Format_Mono` — глубина цвета 1 бит;
- `QImage::Format_MonoLSB` — глубина цвета 1 бит;
- `QImage::Format_Indexed8` — глубина цвета 8 бит;
- `QImage::Format_RGB32` — RGB без альфа-канала, глубина цвета 32 бита;
- `QImage::Format_ARGB32` — RGB с альфа-каналом, глубина цвета 32 бита;
- `QImage::Format_ARGB32_Premultiplied` — RGB с альфа-каналом, глубина цвета 32 бита. Этот формат лучше использовать для рисования;
- `QImage::Format_RGB16`;
- `QImage::Format_ARGB8565_Premultiplied`;
- `QImage::Format_RGB666`;
- `QImage::Format_ARGB6666_Premultiplied`;
- `QImage::Format_RGB555`;
- `QImage::Format_ARGB8555_Premultiplied`;
- `QImage::Format_RGB888`;
- `QImage::Format_RGB444`;
- `QImage::Format_ARGB4444_Premultiplied`;
- `QImage::Format_RGBX8888`;
- `QImage::Format_RGBA8888`;
- `QImage::Format_RGBA8888_Premultiplied`;
- `QImage::Format_BGR30`;
- `QImage::Format_A2BGR30_Premultiplied`;
- `QImage::Format_RGB30`;
- `QImage::Format_A2RGB30_Premultiplied`;
- `QImage::Format_Alpha8`;
- `QImage::Format_Grayscale8`;
- `QImage::Format_Grayscale16`;

- `QImage::Format_RGBX64`;
- `QImage::Format_RGBA64`;
- `QImage::Format_RGBA64_Premultiplied`;
- `QImage::Format_BGR888`.

Класс `QImage` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `isNull()` — возвращает значение `true`, если объект является нулевым, и `false` — в противном случае. Прототип метода:

```
bool isNull() const
```

- `load()` — загружает изображение из файла. В параметре `fileName` задается абсолютный или относительный путь к файлу. В параметре `format` указывается тип изображения. Если параметр не указан, то тип определяется автоматически. Метод возвращает значение `true`, если изображение успешно загружено, и `false` — в противном случае. Прототип метода:

```
bool load(const QString &fileName, const char *format = nullptr)
bool load(QIODevice *device, const char *format)
```

- `loadFromData()` — загружает изображение из массива или экземпляра класса `QByteArray`. Метод возвращает значение `true`, если изображение успешно загружено, и `false` — в противном случае. Прототипы метода:

```
bool loadFromData(const uchar *data, int len, const char *format = nullptr)
bool loadFromData(const QByteArray &data, const char *format = nullptr)
```

- `fromData()` — загружает изображение из массива или экземпляра класса `QByteArray` и возвращает экземпляр класса `QImage`. Метод является статическим. Прототипы метода:

```
static QImage fromData(const uchar *data, int size,
                       const char *format = nullptr)
static QImage fromData(const QByteArray &data,
                       const char *format = nullptr)
```

- `save()` — сохраняет изображение в файл. В параметре `fileName` указывается абсолютный или относительный путь к файлу. Во втором параметре можно задать тип изображения в виде строки. Если параметр не указан, то тип определяется автоматически по расширению файла. Необязательный параметр `quality` позволяет задать качество изображения. Можно передать значение в диапазоне от `0` до `100`. Метод возвращает значение `true`, если изображение успешно сохранено, и `false` — в противном случае. Прототипы метода:

```
bool save(const QString &fileName, const char *format = nullptr,
          int quality = -1) const
bool save(QIODevice *device, const char *format = nullptr,
          int quality = -1) const
```

- `fill()` — производит заливку изображения определенным цветом. Прототипы метода:

```
void fill(const QColor &color)
void fill(Qt::GlobalColor color)
void fill(uint pixelValue)
```

- `width()` — возвращает ширину изображения. Прототип метода:

```
int width() const
```

- `height()` — возвращает высоту изображения. Прототип метода:

```
int height() const
```

- `size()` — возвращает экземпляр класса `QSize` с размерами изображения. Прототип метода:

```
QSize size() const
```

- `rect()` — возвращает экземпляр класса `QRect` с координатами и размерами и прямоугольной области, ограничивающей изображение. Прототип метода:

```
QRect rect() const
```

- `depth()` — возвращает глубину цвета. Прототип метода:

```
int depth() const
```

- `format()` — возвращает формат изображения (см. значения параметра `format` в конструкторе класса `QImage`). Прототип метода:

```
QImage::Format format() const
```

- `setPixel()` — задает цвет указанного пиксела. В параметре `index_or_rgb` для 8-битных изображений задается индекс цвета в палитре, а для 32-битных — целочисленное значение цвета. Прототипы метода:

```
void setPixel(int x, int y, uint index_or_rgb)
void setPixel(const QPoint &position, uint index_or_rgb)
```

- `setPixelColor()` — задает цвет указанного пиксела. Прототипы метода:

```
void setPixelColor(int x, int y, const QColor &color)
void setPixelColor(const QPoint &position, const QColor &color)
```

- `pixel()` — возвращает целочисленное значение цвета указанного пиксела. Прототипы метода:

```
QRgb pixel(int x, int y) const
QRgb pixel(const QPoint &position) const
```

- `pixelColor()` — возвращает цвет указанного пиксела. Прототипы метода:

```
QColor pixelColor(int x, int y) const
QColor pixelColor(const QPoint &position) const
```

- `convertToFormat()` — преобразует формат изображения (см. значения параметра `format` в конструкторе класса `QImage`) и возвращает экземпляр класса `QImage`. Прототипы метода:

```
QImage convertToFormat(QImage::Format format,
   Qt::ImageConversionFlags flags = Qt::AutoColor) const &
QImage convertToFormat(QImage::Format format,
   Qt::ImageConversionFlags flags = Qt::AutoColor) &&
QImage convertToFormat(QImage::Format format,
   const QList<QRgb> &colorTable,
   Qt::ImageConversionFlags flags = Qt::AutoColor) const
```

- `copy()` — возвращает экземпляр класса `QImage` с фрагментом изображения. Если параметр `rect` не указан, то изображение копируется полностью. Прототипы метода:

```
QImage copy(const QRect &rect = QRect()) const
QImage copy(int x, int y, int width, int height) const
```

- `scaled()` — изменяет размер изображения и возвращает экземпляр класса `QImage`. Исходное изображение не изменяется. Прототипы метода:

```
QImage scaled(int width, int height,
       Qt::AspectRatioMode am = Qt::IgnoreAspectRatio,
       Qt::TransformationMode tm = Qt::FastTransformation) const
QImage scaled(const QSize &size,
       Qt::AspectRatioMode am = Qt::IgnoreAspectRatio,
       Qt::TransformationMode tm = Qt::FastTransformation) const
```

  В необязательном параметре `am` могут быть указаны следующие константы:

  - `Qt::IgnoreAspectRatio` — непосредственно изменяет размеры без сохранения пропорций сторон;
  - `Qt::KeepAspectRatio` — производится попытка масштабирования старой области внутри новой области без нарушения пропорций;
  - `Qt::KeepAspectRatioByExpanding` — производится попытка полностью заполнить новую область без нарушения пропорций старой области.

  В необязательном параметре `tm` могут быть указаны следующие константы:

  - `Qt::FastTransformation` — сглаживание выключено;
  - `Qt::SmoothTransformation` — сглаживание включено;

- `scaledToWidth()` — изменяет ширину изображения и возвращает экземпляр класса `QImage`. Высота изображения изменяется пропорционально. Исходное изображение не изменяется. Параметр `mode` аналогичен параметру `tm` в методе `scaled()`. Прототип метода:

```
QImage scaledToWidth(int width,
   Qt::TransformationMode mode = Qt::FastTransformation) const
```

- `scaledToHeight()` — изменяет высоту изображения и возвращает экземпляр класса `QImage`. Ширина изображения изменяется пропорционально. Исходное изображение не изменяется. Параметр `mode` аналогичен параметру `tm` в методе `scaled()`. Прототип метода:

```
QImage scaledToHeight(int height,
   Qt::TransformationMode mode = Qt::FastTransformation) const
```

- `transformed()` — производит трансформацию изображения (например, поворот) и возвращает экземпляр класса `QImage`. Исходное изображение не изменяется. Прототип метода:

```
QImage transformed(const QTransform &matrix,
   Qt::TransformationMode mode = Qt::FastTransformation) const
```

- `invertPixels()` — инвертирует значения всех пикселов в изображении. В необязательном параметре `mode` могут быть указаны константы `QImage::InvertRgb` или `QImage::InvertRgba`. Прототип метода:

```
void invertPixels(QImage::InvertMode mode = InvertRgb)
```

- `mirrored()` — создает зеркальный образ изображения. Прототипы метода:

```
QImage mirrored(bool horizontal = false, bool vertical = true) const &
QImage mirrored(bool horizontal = false, bool vertical = true) &&
```

8.3.4. Класс *QIcon*

Класс `QIcon` описывает значки в различных размерах, режимах и состояниях. Обратите внимание на то, что класс `QIcon` не наследует класс `QPaintDevice`, следовательно, мы не можем использовать его как поверхность для рисования. Форматы конструктора:

```
#include <QIcon>
QIcon()
QIcon(const QString &fileName)
QIcon(const QPixmap &pixmap)
QIcon(QIconEngine *engine)
QIcon(const QIcon &other)
QIcon(QIcon &&other)
```

Первый конструктор создает нулевой объект значка. Второй конструктор предназначен для загрузки значка из файла. Обратите внимание на то, что файл загружается при первой попытке использования, а не сразу. Третий конструктор создает значок на основе экземпляра класса `QPixmap`.

Класс `QIcon` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `isNull()` — возвращает значение `true`, если объект является нулевым, и `false` — в противном случае. Прототип метода:

```
bool isNull() const
```

- `addFile()` — добавляет значок для указанного размера, режима и состояния. Можно добавить несколько значков, вызывая метод с разными значениями параметров. Параметр `size` задает размер значка (с помощью экземпляра класса `QSize`). Так как загрузка значка производится при первой попытке использова-

ния, заранее размер значка неизвестен. В параметре `mode` можно указать следующие константы: `QIcon::Normal`, `QIcon::Disabled`, `QIcon::Active` или `QIcon::Selected`. В параметре `state` указываются константы `QIcon::Off` или `QIcon::On`. Прототип метода:

```
void addFile(const QString &fileName,
   const QSize &size = QSize(), QIcon::Mode mode = Normal,
   QIcon::State state = Off)
```

- `addPixmap()` — добавляет значок для указанного режима и состояния. Значок загружается из экземпляра класса `QPixmap`. Прототип метода:

```
void addPixmap(const QPixmap &pixmap, QIcon::Mode mode = Normal,
               QIcon::State state = Off)
```

- `availableSizes()` — возвращает доступные размеры (список с экземплярами класса `QSize`) значков для указанного режима и состояния. Прототип метода:

```
QList<QSize> availableSizes(QIcon::Mode mode = Normal,
                            QIcon::State state = Off) const
```

- `actualSize()` — возвращает фактический размер (экземпляр класса `QSize`) для указанного размера, режима и состояния. Фактический размер может быть меньше размера, указанного в первом параметре, но не больше. Прототип метода:

```
QSize actualSize(const QSize &size, QIcon::Mode mode = Normal,
                 QIcon::State state = Off) const
```

- `pixmap()` — возвращает значок (экземпляр класса `QPixmap`), который примерно соответствует указанному размеру, режиму и состоянию. Прототипы метода:

```
QPixmap pixmap(int w, int h, QIcon::Mode mode = Normal,
               QIcon::State state = Off) const
QPixmap pixmap(const QSize &size, QIcon::Mode mode = Normal,
               QIcon::State state = Off) const
QPixmap pixmap(int extent, QIcon::Mode mode = Normal,
               QIcon::State state = Off) const
QPixmap pixmap(const QSize &size, qreal devicePixelRatio,
   QIcon::Mode mode = Normal, QIcon::State state = Off) const
```

Вместо загрузки значка из файла можно воспользоваться одним из встроенных значков. Загрузить стандартный значок позволяет следующий код:

```
// #include <QIcon>
// #include <QStyle>
QIcon ico = window.style()->standardIcon(QStyle::SP_MessageBoxCritical);
window.setWindowIcon(ico);
```

Посмотреть список всех встроенных значков можно в документации к классу `QStyle`.

ГЛАВА 9

Графическая сцена

Графическая сцена позволяет отображать объекты (например, линию, прямоугольник и др.) и производить с ними различные манипуляции (например, перемещать с помощью мыши, трансформировать и др.). Для отображения графических объектов применяется концепция «модель/представление», позволяющая отделить данные от их отображения и избежать дублирования данных. Благодаря этому одну и ту же сцену можно отобразить сразу в нескольких представлениях без дублирования. В основе концепции лежат следующие классы:

- `QGraphicsScene` — исполняет роль сцены, на которой расположены графические объекты. Этот класс содержит также множество методов для управления этими объектами;
- `QGraphicsView` — предназначен для отображения сцены. Одну сцену можно отображать с помощью нескольких представлений;
- `QGraphicsItem` — является базовым классом для графических объектов. Можно наследовать этот класс и реализовать свой графический объект или воспользоваться готовыми классами, например: `QGraphicsRectItem` (прямоугольник), `QGraphicsEllipseItem` (эллипс) и др.

9.1. Класс *QGraphicsScene:* сцена

Класс `QGraphicsScene` исполняет роль сцены, на которой расположены графические объекты. Этот класс содержит также множество методов для управления этими объектами. Иерархия наследования выглядит так:

```
QObject — QGraphicsScene
```

Форматы конструктора:

```
#include <QGraphicsScene>
QGraphicsScene(QObject *parent = nullptr)
QGraphicsScene(qreal x, qreal y, qreal width, qreal height,
               QObject *parent = nullptr)
QGraphicsScene(const QRectF &sceneRect, QObject *parent = nullptr)
```

Первый конструктор создает сцену, не имеющую определенного размера. Второй и третий конструкторы позволяют указать размеры сцены в виде вещественных чисел или экземпляра класса QRectF. В качестве параметра parent можно передать указатель на родительский компонент.

9.1.1. Настройка параметров сцены

Для настройки различных параметров сцены предназначены следующие методы из класса QGraphicsScene:

- setSceneRect() — задает координаты и размеры сцены. Прототипы метода:

```
void setSceneRect(qreal x, qreal y, qreal w, qreal h)
void setSceneRect(const QRectF &rect)
```

- sceneRect() — возвращает экземпляр класса QRectF с координатами и размерами сцены. Прототип метода:

```
QRectF sceneRect() const
```

- width() и height() — возвращают ширину и высоту сцены соответственно в виде вещественного числа. Прототипы методов:

```
qreal width() const
qreal height() const
```

- itemsBoundingRect() — возвращает экземпляр класса QRectF с координатами и размерами прямоугольника, в который можно вписать все объекты, расположенные на сцене. Прототип метода:

```
QRectF itemsBoundingRect() const
```

- setBackgroundBrush() — задает кисть для заднего плана (расположен под графическими объектами). Чтобы изменить задний фон, можно также переопределить метод drawBackground() и произвести в нем рисование. Прототип метода:

```
void setBackgroundBrush(const QBrush &brush)
```

- setForegroundBrush() — задает кисть для переднего плана (расположен над графическими объектами). Чтобы изменить передний фон, можно также переопределить метод drawForeground() и произвести в нем рисование. Прототип метода:

```
void setForegroundBrush(const QBrush &brush)
```

- setFont() — задает шрифт сцены по умолчанию. Прототип метода:

```
void setFont(const QFont &font)
```

- setItemIndexMethod() — задает режим индексации объектов сцены. Прототип метода:

```
void setItemIndexMethod(QGraphicsScene::ItemIndexMethod method)
```

  В качестве параметра указываются следующие константы:

  - QGraphicsScene::BspTreeIndex — для поиска объектов используется индекс в виде бинарного дерева. Этот режим следует использовать для сцен, на ко-

торых большинство объектов являются статическими (которые нельзя перемещать);

- `QGraphicsScene::NoIndex` — индекс не используется. Этот режим следует использовать для динамических сцен;

- `setBspTreeDepth()` — задает глубину дерева при использовании режима `BspTreeIndex`. По умолчанию установлено значение `0`, которое означает, что глубина выбирается автоматически. Прототип метода:

```
void setBspTreeDepth(int depth)
```

- `bspTreeDepth()` — возвращает текущее значение глубины дерева при использовании режима `BspTreeIndex`. Прототип метода:

```
int bspTreeDepth() const
```

9.1.2. Добавление и удаление графических объектов

Для добавления графических объектов на сцену и удаления их предназначены следующие методы из класса `QGraphicsScene`:

- `addItem()` — добавляет графический объект на сцену. В качестве значения указывается экземпляр класса, который наследует класс `QGraphicsItem`, например `QGraphicsEllipseItem` (эллипс). Прототип метода:

```
void addItem(QGraphicsItem *item)
```

- `addLine()` — создает линию, добавляет ее на сцену и возвращает указатель на нее (экземпляр класса `QGraphicsLineItem`). Прототипы метода:

```
QGraphicsLineItem *addLine(qreal x1, qreal y1, qreal x2, qreal y2,
                           const QPen &pen = QPen())
QGraphicsLineItem *addLine(const QLineF &line,
                           const QPen &pen = QPen())
```

- `addRect()` — создает прямоугольник, добавляет его на сцену и возвращает указатель на него (экземпляр класса `QGraphicsRectItem`). Прототипы метода:

```
QGraphicsRectItem *addRect(qreal x, qreal y, qreal w, qreal h,
   const QPen &pen = QPen(), const QBrush &brush = QBrush())
QGraphicsRectItem *addRect(const QRectF &rect,
   const QPen &pen = QPen(), const QBrush &brush = QBrush())
```

- `addPolygon()` — создает многоугольник, добавляет его на сцену и возвращает указатель на него (экземпляр класса `QGraphicsPolygonItem`). Прототип метода:

```
QGraphicsPolygonItem *addPolygon(const QPolygonF &polygon,
   const QPen &pen = QPen(), const QBrush &brush = QBrush())
```

- `addEllipse()` — создает эллипс, добавляет его на сцену и возвращает указатель на него (экземпляр класса `QGraphicsEllipseItem`). Прототипы метода:

```
QGraphicsEllipseItem *addEllipse(qreal x, qreal y, qreal w,
   qreal h, const QPen &pen = QPen(),
   const QBrush &brush = QBrush())
QGraphicsEllipseItem *addEllipse(const QRectF &rect,
   const QPen &pen = QPen(), const QBrush &brush = QBrush())
```

- `addPixmap()` — создает изображение, добавляет его на сцену и возвращает указатель на него (экземпляр класса `QGraphicsPixmapItem`). Прототип метода:

```
QGraphicsPixmapItem *addPixmap(const QPixmap &pixmap)
```

- `addSimpleText()` — создает экземпляр класса `QGraphicsSimpleTextItem`, добавляет его на сцену в позицию с координатами `(0, 0)` и возвращает указатель на него. Прототип метода:

```
QGraphicsSimpleTextItem *addSimpleText(const QString &text,
                                       const QFont &font = QFont())
```

- `addText()` — создает экземпляр класса `QGraphicsTextItem`, добавляет его на сцену в позицию с координатами `(0, 0)` и возвращает указатель на него. Прототип метода:

```
QGraphicsTextItem *addText(const QString &text, const QFont &font = QFont())
```

- `addPath()` — создает экземпляр класса `QGraphicsPathItem`, добавляет его на сцену и возвращает указатель на него. Прототип метода:

```
QGraphicsPathItem *addPath(const QPainterPath &path,
   const QPen &pen = QPen(), const QBrush &brush = QBrush())
```

- `removeItem()` — убирает графический объект (и всех его потомков) со сцены. Графический объект при этом не удаляется и, например, может быть добавлен на другую сцену. Прототип метода:

```
void removeItem(QGraphicsItem *item)
```

- `clear()` — удаляет все элементы со сцены. Метод является слотом. Прототип метода:

```
void clear()
```

- `createItemGroup()` — группирует объекты, добавляет группу на сцену и возвращает указатель на экземпляр класса `QGraphicsItemGroup`. Прототип метода:

```
QGraphicsItemGroup *createItemGroup(const QList<QGraphicsItem *> &items)
```

- `destroyItemGroup()` — удаляет группу со сцены. Прототип метода:

```
void destroyItemGroup(QGraphicsItemGroup *group)
```

9.1.3. Добавление компонентов на сцену

Помимо графических объектов на сцену можно добавить компоненты, которые будут функционировать как обычно. Добавить компонент на сцену позволяет метод `addWidget()` из класса `QGraphicsScene`. Прототип метода:

```
QGraphicsProxyWidget *addWidget(QWidget *widget,
                        Qt::WindowFlags wFlags = Qt::WindowFlags())
```

В первом параметре указывается экземпляр класса, который наследует класс `QWidget`. Во втором параметре задается тип окна *(см. разд. 3.2)*. Метод возвращает указатель на добавленный компонент (экземпляр класса `QGraphicsProxyWidget`).

Чтобы изменить текст в заголовке окна, следует воспользоваться методом `setWindowTitle()` из класса `QGraphicsWidget`.

Сделать окно активным позволяет метод `setActiveWindow()`. Получить указатель на активное окно можно с помощью метода `activeWindow()`. Если активного окна нет, то метод возвращает нулевой указатель. Прототипы методов:

```
void setActiveWindow(QGraphicsWidget *widget)
QGraphicsWidget *activeWindow() const
```

9.1.4. Поиск объектов

Для поиска объектов предназначены следующие методы из класса `QGraphicsScene`:

- `itemAt()` — возвращает указатель на верхний видимый объект, который расположен по указанным координатам, или нулевой указатель, если объекта нет. Прототипы метода:

```
QGraphicsItem *itemAt(qreal x, qreal y,
                      const QTransform &deviceTransform) const
QGraphicsItem *itemAt(const QPointF &position,
                      const QTransform &deviceTransform) const
```

- `collidingItems()` — возвращает список с указателями на объекты, которые сталкиваются с указанным в первом параметре объектом. Если таких объектов нет, то метод возвращает пустой список. Прототип метода:

```
QList<QGraphicsItem *> collidingItems(const QGraphicsItem *item,
   Qt::ItemSelectionMode mode = Qt::IntersectsItemShape) const
```

- `items()` — возвращает список с указателями на все объекты или на объекты, расположенные по указанным координатам, или объекты, попадающие в указанную область. Если объектов нет, то возвращается пустой список. Прототипы метода:

```
QList<QGraphicsItem *> items(
   Qt::SortOrder order = Qt::DescendingOrder) const
QList<QGraphicsItem *> items(const QPointF &pos,
   Qt::ItemSelectionMode mode = Qt::IntersectsItemShape,
   Qt::SortOrder order = Qt::DescendingOrder,
   const QTransform &deviceTransform = QTransform()) const
QList<QGraphicsItem *> items(qreal x, qreal y, qreal w, qreal h,
   Qt::ItemSelectionMode mode, Qt::SortOrder order,
   const QTransform &deviceTransform = QTransform()) const
```

```
QList<QGraphicsItem *> items(const QRectF &rect,
   Qt::ItemSelectionMode mode = Qt::IntersectsItemShape,
   Qt::SortOrder order = Qt::DescendingOrder,
   const QTransform &deviceTransform = QTransform()) const
QList<QGraphicsItem *> items(const QPolygonF &polygon,
   Qt::ItemSelectionMode mode = Qt::IntersectsItemShape,
   Qt::SortOrder order = Qt::DescendingOrder,
   const QTransform &deviceTransform = QTransform()) const
QList<QGraphicsItem *> items(const QPainterPath &path,
   Qt::ItemSelectionMode mode = Qt::IntersectsItemShape,
   Qt::SortOrder order = Qt::DescendingOrder,
   const QTransform &deviceTransform = QTransform()) const
```

В параметре `order` указываются константы `Qt::AscendingOrder` (в прямом порядке) или `Qt::DescendingOrder` (в обратном порядке).

В параметре `mode` могут быть указаны следующие константы:

- `Qt::ContainsItemShape` — объект попадет в список, если все точки объекта находятся внутри области;
- `Qt::IntersectsItemShape` — объект попадет в список, если любая точка объекта попадет в область;
- `Qt::ContainsItemBoundingRect` — объект попадет в список, если охватывающий прямоугольник полностью находится внутри области;
- `Qt::IntersectsItemBoundingRect` — объект попадет в список, если любая точка охватывающего прямоугольника попадет в область.

9.1.5. Управление фокусом ввода

Обладать фокусом ввода с клавиатуры может как сцена в целом, так и отдельный объект на сцене. Если фокус установлен на отдельный объект, то все события клавиатуры перенаправляются этому объекту. Чтобы объект мог принимать фокус ввода, необходимо установить флаг `ItemIsFocusable`, например с помощью метода `setFlag()` из класса `QGraphicsItem`. Для управления фокусом ввода предназначены следующие методы из класса `QGraphicsScene`:

- `setFocus()` — устанавливает фокус ввода на сцену. В параметре `focusReason` можно указать причину изменения фокуса ввода *(см. разд. 4.8.1)*. Прототип метода:

  ```
  void setFocus(Qt::FocusReason focusReason = Qt::OtherFocusReason)
  ```

- `setFocusItem()` — устанавливает фокус ввода на указанный графический объект на сцене. Если сцена была вне фокуса ввода, то она автоматически получит фокус. Если объект не может принимать фокус, то метод просто убирает фокус с объекта, который обладает фокусом ввода в данный момент. В параметре `focusReason` можно указать причину изменения фокуса ввода *(см. разд. 4.8.1)*. Прототип метода:

```
void setFocusItem(QGraphicsItem *item,
        Qt::FocusReason focusReason = Qt::OtherFocusReason)
```

- `clearFocus()` — убирает фокус ввода со сцены. Объект на сцене, который обладает фокусом ввода в данный момент, потеряет фокус, но получит его снова, когда фокус будет снова установлен на сцену. Прототип метода:

```
void clearFocus()
```

- `hasFocus()` — возвращает значение `true`, если сцена имеет фокус ввода, и `false` — в противном случае. Прототип метода:

```
bool hasFocus() const
```

- `focusItem()` — возвращает указатель на объект, который обладает фокусом ввода, или нулевой указатель. Прототип метода:

```
QGraphicsItem *focusItem() const
```

- `setStickyFocus()` — если в качестве параметра указано значение `true`, то при щелчке мышью на фоне сцены или на объекте, который не может принимать фокус, объект, владеющий фокусом, не потеряет фокус ввода. По умолчанию фокус убирается. Прототип метода:

```
void setStickyFocus(bool enabled)
```

- `stickyFocus()` — возвращает значение `true`, если фокус ввода не будет убран с объекта при щелчке мышью на фоне или на объекте, который не может принимать фокус. Прототип метода:

```
bool stickyFocus() const
```

9.1.6. Управление выделением объектов

Чтобы объект можно было выделить (с помощью мыши или из программы), необходимо установить флаг `ItemIsSelectable`, например с помощью метода `setFlag()` из класса `QGraphicsItem`. Для управления выделением объектов предназначены следующие методы из класса `QGraphicsScene`:

- `setSelectionArea()` — выделяет объекты внутри указанной области. Чтобы выделить только один объект, следует воспользоваться методом `setSelected()` из класса `QGraphicsItem`. Прототипы метода:

```
void setSelectionArea(const QPainterPath &path,
        const QTransform &deviceTransform)
void setSelectionArea(const QPainterPath &path,
        Qt::ItemSelectionOperation selectionOperation =
        Qt::ReplaceSelection, Qt::ItemSelectionMode mode =
        Qt::IntersectsItemShape,
        const QTransform &deviceTransform = QTransform())
```

  В параметре `mode` могут быть указаны следующие константы:

  - `Qt::ContainsItemShape` — объект будет выделен, если все точки объекта находятся внутри области выделения;

- `Qt::IntersectsItemShape` — объект будет выделен, если любая точка объекта попадет в область выделения;
- `Qt::ContainsItemBoundingRect` — объект будет выделен, если охватывающий прямоугольник полностью находится внутри области выделения;
- `Qt::IntersectsItemBoundingRect` — объект будет выделен, если любая точка охватывающего прямоугольника попадет в область выделения;

❐ `selectionArea()` — возвращает область выделения (экземпляр класса `QPainterPath`). Прототип метода:

```
QPainterPath selectionArea() const
```

❐ `selectedItems()` — возвращает список с указателями на выделенные объекты или пустой список, если выделенных объектов нет. Прототип метода:

```
QList<QGraphicsItem *> selectedItems() const
```

❐ `clearSelection()` — снимает выделение. Метод является слотом. Прототип метода:

```
void clearSelection()
```

9.1.7. Прочие методы и сигналы

Помимо рассмотренных методов класс `QGraphicsScene` содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

❐ `isActive()` — возвращает значение `true`, если сцена отображается представлением, и `false` — в противном случае. Прототип метода:

```
bool isActive() const
```

❐ `views()` — возвращает список с представлениями (экземпляры класса `QGraphicsView`), к которым подключена сцена. Если сцена не подключена к представлениям, то возвращается пустой список. Прототип метода:

```
QList<QGraphicsView *> views() const
```

❐ `mouseGrabberItem()` — возвращает указатель на объект, который владеет мышью, или нулевой указатель, если такого объекта нет. Прототип метода:

```
QGraphicsItem *mouseGrabberItem() const
```

❐ `render()` — позволяет вывести содержимое сцены на принтер или на устройство рисования. Прототип метода:

```
void render(QPainter *painter, const QRectF &target = QRectF(),
     const QRectF &source = QRectF(),
     Qt::AspectRatioMode aspectRatioMode = Qt::KeepAspectRatio)
```

Параметр `target` задает координаты и размеры устройства рисования, а параметр `source` — координаты и размеры прямоугольной области на сцене. Если параметры не указаны, то используются размеры устройства рисования и сцены;

- `invalidate()` — вызывает перерисовку указанных слоев внутри прямоугольной области на сцене. Прототипы метода (второй прототип является слотом):

```
void invalidate(qreal x, qreal y, qreal w, qreal h,
                QGraphicsScene::SceneLayers layers = AllLayers)
void invalidate(const QRectF &rect = QRectF(),
                QGraphicsScene::SceneLayers layers = AllLayers)
```

  В параметре `layers` могут быть указаны следующие константы:

  - `QGraphicsScene::ItemLayer` — слой объекта;
  - `QGraphicsScene::BackgroundLayer` — слой заднего плана;
  - `QGraphicsScene::ForegroundLayer` — слой переднего плана;
  - `QGraphicsScene::AllLayers` — все слои. Сначала отрисовывается слой заднего плана, затем слой объекта и в конце слой переднего плана;

- `update()` — вызывает перерисовку указанной прямоугольной области сцены. Прототипы метода (второй прототип является слотом):

```
void update(qreal x, qreal y, qreal w, qreal h)
void update(const QRectF &rect = QRectF())
```

Класс `QGraphicsScene` содержит следующие сигналы:

- `changed(const QList<QRectF>&)` — генерируется при изменении сцены. Внутри обработчика через параметр доступен список с экземплярами класса `QRectF` или пустой список;
- `sceneRectChanged(const QRectF&)` — генерируется при изменении размеров сцены. Внутри обработчика через параметр доступен экземпляр класса `QRectF` с новыми координатами и размерами сцены;
- `selectionChanged()` — генерируется при изменении выделения объектов;
- `focusItemChanged(QGraphicsItem*, QGraphicsItem*, Qt::FocusReason)` — генерируется при изменении фокуса.

9.2. Класс *QGraphicsView:* представление

Класс `QGraphicsView` предназначен для отображения сцены. Одну сцену можно отображать с помощью нескольких представлений. Иерархия наследования:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                        — QAbstractScrollArea — QGraphicsView
```

Форматы конструктора класса:

```
#include <QGraphicsView>
QGraphicsView(QWidget *parent = nullptr)
QGraphicsView(QGraphicsScene *scene, QWidget *parent = nullptr)
```

9.2.1. Настройка параметров представления

Для настройки различных параметров представления предназначены следующие методы из класса `QGraphicsView` (перечислены только основные методы; полный список смотрите в документации):

- `setScene()` — устанавливает сцену в представление. Прототип метода:

  ```
  void setScene(QGraphicsScene *scene)
  ```

- `scene()` — возвращает указатель на сцену. Прототип метода:

  ```
  QGraphicsScene *scene() const
  ```

- `setSceneRect()` — задает координаты и размеры сцены. Прототипы метода:

  ```
  void setSceneRect(qreal x, qreal y, qreal w, qreal h)
  void setSceneRect(const QRectF &rect)
  ```

- `sceneRect()` — возвращает экземпляр класса `QRectF` с координатами и размерами сцены. Прототип метода:

  ```
  QRectF sceneRect() const
  ```

- `setBackgroundBrush()` — задает кисть для заднего плана сцены (расположен под графическими объектами). Прототип метода:

  ```
  void setBackgroundBrush(const QBrush &brush)
  ```

- `setForegroundBrush()` — задает кисть для переднего плана сцены (расположен над графическими объектами). Прототип метода:

  ```
  void setForegroundBrush(const QBrush &brush)
  ```

- `setCacheMode()` — задает режим кеширования. Прототип метода:

  ```
  void setCacheMode(QGraphicsView::CacheMode mode)
  ```

  В качестве параметра могут быть указаны следующие константы:

  - `QGraphicsView::CacheNone` — без кеширования;
  - `QGraphicsView::CacheBackground` — кешируется задний фон;

- `resetCachedContent()` — сбрасывает кеш. Прототип метода:

  ```
  void resetCachedContent()
  ```

- `setAlignment()` — задает выравнивание сцены внутри представления при отсутствии полос прокрутки. По умолчанию сцена центрируется по горизонтали и вертикали. Прототип метода:

  ```
  void setAlignment(Qt::Alignment alignment)
  ```

- `setInteractive()` — если в качестве параметра указано значение `true`, то пользователь может взаимодействовать с объектами на сцене (интерактивный режим используется по умолчанию). Значение `false` устанавливает режим только для чтения. Прототип метода:

  ```
  void setInteractive(bool allowed)
  ```

- `isInteractive()` — возвращает значение `true`, если используется интерактивный режим, и `false` — в противном случае. Прототип метода:

  ```
  bool isInteractive() const
  ```

- `setDragMode()` — задает действие, которое производится при щелчке левой кнопкой мыши на фоне и перемещении мыши. Получить текущее действие позволяет метод `dragMode()`. Прототипы методов:

  ```
  void setDragMode(QGraphicsView::DragMode mode)
  QGraphicsView::DragMode dragMode() const
  ```

  В качестве параметра `mode` могут быть указаны следующие константы:

  - `QGraphicsView::NoDrag` — действие не определено;
  - `QGraphicsView::ScrollHandDrag` — перемещение мыши при нажатой левой кнопке будет приводить к прокрутке сцены. При этом указатель мыши примет вид сжатой или разжатой руки;
  - `QGraphicsView::RubberBandDrag` — создается область выделения. Объекты, частично или полностью (задается с помощью метода `setRubberBandSelectionMode()`) попавшие в эту область, будут выделены (при условии, что установлен флаг `ItemIsSelectable`). Действие выполняется только в интерактивном режиме;

- `setRubberBandSelectionMode()` — задает режим выделения объектов при установленном флаге `RubberBandDrag`. Прототип метода:

  ```
  void setRubberBandSelectionMode(Qt::ItemSelectionMode mode)
  ```

  В параметре `mode` могут быть указаны следующие константы:

  - `Qt::ContainsItemShape` — объект будет выделен, если все точки объекта находятся внутри области выделения;
  - `Qt::IntersectsItemShape` — объект будет выделен, если любая точка объекта попадет в область выделения;
  - `Qt::ContainsItemBoundingRect` — объект будет выделен, если охватывающий прямоугольник полностью находится внутри области выделения;
  - `Qt::IntersectsItemBoundingRect` — объект будет выделен, если любая точка охватывающего прямоугольника попадет в область выделения.

9.2.2. Преобразования между координатами представления и сцены

Для преобразования между координатами представления и сцены предназначены следующие методы из класса `QGraphicsView`:

- `mapFromScene()` — преобразует координаты из системы координат сцены в систему координат представления. Прототипы метода:

```
QPoint mapFromScene(qreal x, qreal y) const
QPoint mapFromScene(const QPointF &point) const
QPolygon mapFromScene(qreal x, qreal y, qreal w, qreal h) const
QPolygon mapFromScene(const QRectF &rect) const
QPolygon mapFromScene(const QPolygonF &polygon) const
QPainterPath mapFromScene(const QPainterPath &path) const
```

- `mapToScene()` — преобразует координаты из системы координат представления в систему координат сцены. Прототипы метода:

```
QPointF mapToScene(int x, int y) const
QPointF mapToScene(const QPoint &point) const
QPolygonF mapToScene(int x, int y, int w, int h) const
QPolygonF mapToScene(const QRect &rect) const
QPolygonF mapToScene(const QPolygon &polygon) const
QPainterPath mapToScene(const QPainterPath &path) const
```

9.2.3. Поиск объектов

Для поиска объектов на сцене предназначены следующие методы:

- `itemAt()` — возвращает указатель на верхний видимый объект, который расположен по указанным координатам, или нулевой указатель, если объекта нет. В качестве значений указываются координаты в системе координат представления, а не сцены. Прототипы метода:

```
QGraphicsItem *itemAt(int x, int y) const
QGraphicsItem *itemAt(const QPoint &pos) const
```

- `items()` — возвращает список с указателями на все объекты или на объекты, расположенные по указанным координатам, или объекты, попадающие в указанную область. Если объектов нет, то возвращается пустой список. В качестве значений указываются координаты в системе координат представления, а не сцены. Прототипы метода:

```
QList<QGraphicsItem *> items() const
QList<QGraphicsItem *> items(int x, int y) const
QList<QGraphicsItem *> items(const QPoint &pos) const
QList<QGraphicsItem *> items(int x, int y, int w, int h,
   Qt::ItemSelectionMode mode = Qt::IntersectsItemShape) const
QList<QGraphicsItem *> items(const QRect &rect,
   Qt::ItemSelectionMode mode = Qt::IntersectsItemShape) const
QList<QGraphicsItem *> items(const QPolygon &polygon,
   Qt::ItemSelectionMode mode = Qt::IntersectsItemShape) const
QList<QGraphicsItem *> items(const QPainterPath &path,
   Qt::ItemSelectionMode mode = Qt::IntersectsItemShape) const
```

Допустимые значения параметра `mode` мы рассматривали в *разд. 9.1.4*.

9.2.4. Трансформация систем координат

Произвести трансформацию системы координат позволяют следующие методы из класса `QGraphicsView`:

- `translate()` — перемещает начало координат в указанную точку. По умолчанию начало координат находится в левом верхнем углу. Положительная ось `X` направлена вправо, а положительная ось `Y` — вниз. Прототип метода:

```
void translate(qreal dx, qreal dy)
```

- `rotate()` — поворачивает систему координат на указанное количество градусов (указывается вещественное число). Положительное значение вызывает поворот по часовой стрелке, а отрицательное значение — против часовой стрелки. Прототип метода:

```
void rotate(qreal angle)
```

- `scale()` — масштабирует систему координат. Если значение меньше единицы, то выполняется уменьшение, а если больше единицы, то увеличение. Прототип метода:

```
void scale(qreal sx, qreal sy)
```

- `shear()` — сдвигает систему координат. Прототип метода:

```
void shear(qreal sh, qreal sv)
```

- `resetTransform()` — отменяет все трансформации. Прототип метода:

```
void resetTransform()
```

Несколько трансформаций можно произвести последовательно друг за другом. В этом случае следует учитывать, что порядок следования трансформаций имеет значение. Если одна и та же последовательность выполняется несколько раз, то ее можно сохранить в экземпляре класса `QTransform`, а затем установить с помощью метода `setTransform()`. Получить установленную матрицу позволяет метод `transform()`. Прототипы методов:

```
void setTransform(const QTransform &matrix, bool combine = false)
QTransform transform() const
```

9.2.5. Прочие методы

Помимо рассмотренных методов класс `QGraphicsView` содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `centerOn()` — прокручивает область таким образом, чтобы указанная точка или объект находились в центре видимой области представления. Прототипы метода:

```
void centerOn(qreal x, qreal y)
void centerOn(const QPointF &pos)
void centerOn(const QGraphicsItem *item)
```

- `ensureVisible()` — прокручивает область таким образом, чтобы указанный прямоугольник или объект находились в видимой области представления. Прототипы метода:

```
void ensureVisible(qreal x, qreal y, qreal w, qreal h,
                   int xmargin = 50, int ymargin = 50)
void ensureVisible(const QRectF &rect,
                   int xmargin = 50, int ymargin = 50)
void ensureVisible(const QGraphicsItem *item,
                   int xmargin = 50, int ymargin = 50)
```

- `fitInView()` — прокручивает и масштабирует область таким образом, чтобы указанный прямоугольник или объект занимали всю видимую область представления. Прототипы метода:

```
void fitInView(qreal x, qreal y, qreal w, qreal h,
   Qt::AspectRatioMode aspectRatioMode = Qt::IgnoreAspectRatio)
void fitInView(const QRectF &rect,
   Qt::AspectRatioMode aspectRatioMode = Qt::IgnoreAspectRatio)
void fitInView(const QGraphicsItem *item,
   Qt::AspectRatioMode aspectRatioMode = Qt::IgnoreAspectRatio)
```

- `render()` — позволяет вывести содержимое представления на принтер или на устройство рисования. Прототип метода:

```
void render(QPainter *painter, const QRectF &target = QRectF(),
   const QRect &source = QRect(),
   Qt::AspectRatioMode aspectRatioMode = Qt::KeepAspectRatio)
```

- `invalidateScene()` — вызывает перерисовку указанных слоев внутри прямоугольной области на сцене. Метод является слотом. Прототип метода:

```
void invalidateScene(const QRectF &rect = QRectF(),
 QGraphicsScene::SceneLayers layers = QGraphicsScene::AllLayers)
```

- `updateSceneRect()` — вызывает перерисовку указанной прямоугольной области сцены. Метод является слотом. Прототип метода:

```
void updateSceneRect(const QRectF &rect)
```

- `updateScene()` — вызывает перерисовку указанных прямоугольных областей. Метод является слотом. Прототип метода:

```
void updateScene(const QList<QRectF> &rects)
```

9.3. Класс *QGraphicsItem:* базовый класс для графических объектов

Абстрактный класс `QGraphicsItem` является базовым классом для графических объектов. Формат конструктора класса:

```
QGraphicsItem(QGraphicsItem *parent = nullptr)
```

В параметре `parent` может быть передан указатель на родительский объект (экземпляр класса, наследующего класс `QGraphicsItem`).

Так как класс `QGraphicsItem` является абстрактным, создать экземпляр этого класса нельзя. Чтобы создать новый графический объект, следует наследовать класс `QGraphicsItem` и переопределить как минимум методы `boundingRect()` и `paint()`. Метод `boundingRect()` должен возвращать экземпляр класса `QRectF` с координатами и размерами прямоугольной области, ограничивающей объект. Внутри метода `paint()` необходимо выполнить рисование объекта. Прототипы методов:

```
virtual QRectF boundingRect() const = 0
virtual void paint(QPainter *painter,
  const QStyleOptionGraphicsItem *option, QWidget *widget = nullptr) = 0
```

Для обработки столкновений следует также переопределить метод `shape()`. Метод должен возвращать экземпляр класса `QPainterPath`. Прототип метода:

```
virtual QPainterPath shape() const
```

9.3.1. Настройка параметров объекта

Для настройки различных параметров объекта предназначены следующие методы из класса `QGraphicsItem` (перечислены только основные методы; полный список смотрите в документации):

- `setPos()` — задает позицию объекта относительно родителя или сцены (при отсутствии родителя). Прототипы метода:

  ```
  void setPos(qreal x, qreal y)
  void setPos(const QPointF &pos)
  ```

- `pos()` — возвращает экземпляр класса `QPointF` с текущими координатами относительно родителя или сцены (при отсутствии родителя). Прототип метода:

  ```
  QPointF pos() const
  ```

- `scenePos()` — возвращает экземпляр класса `QPointF` с текущими координатами относительно сцены. Прототип метода:

  ```
  QPointF scenePos() const
  ```

- `sceneBoundingRect()` — возвращает экземпляр класса `QRectF`, который содержит координаты (относительно сцены) и размеры прямоугольника, ограничивающего объект. Прототип метода:

  ```
  QRectF sceneBoundingRect() const
  ```

- `setX()` и `setY()` — задают позицию объекта по отдельным осям. Прототипы методов:

  ```
  void setX(qreal x)
  void setY(qreal y)
  ```

- `x()` и `y()` — возвращают позицию объекта по отдельным осям. Прототипы методов:

```
qreal x() const
qreal y() const
```

- `setZValue()` — задает позицию объекта по оси `Z`. Объект с большим значением рисуется выше объекта с меньшим значением. По умолчанию для всех объектов значение равно `0.0`. Прототип метода:

```
void setZValue(qreal z)
```

- `zValue()` — возвращает позицию объекта по оси `Z`. Прототип метода:

```
qreal zValue() const
```

- `moveBy()` — сдвигает объект на указанное смещение относительно текущей позиции. Прототип метода:

```
void moveBy(qreal dx, qreal dy)
```

- `prepareGeometryChange()` — этот метод следует вызвать перед изменением размеров объекта, чтобы поддержать индекс сцены в актуальном состоянии. Прототип метода:

```
protected: void prepareGeometryChange()
```

- `scene()` — возвращает указатель на сцену. Прототип метода:

```
QGraphicsScene *scene() const
```

- `setFlag()` — устанавливает (если второй параметр имеет значение `true`) или сбрасывает (если второй параметр имеет значение `false`) указанный флаг. Прототип метода:

```
void setFlag(QGraphicsItem::GraphicsItemFlag flag,
             bool enabled = true)
```

  В первом параметре могут быть указаны следующие константы:

  - `QGraphicsItem::ItemIsMovable` — объект можно перемещать с помощью мыши;
  - `QGraphicsItem::ItemIsSelectable` — объект можно выделять;
  - `QGraphicsItem::ItemIsFocusable` — объект может получить фокус ввода;
  - `QGraphicsItem::ItemClipsToShape`;
  - `QGraphicsItem::ItemClipsChildrenToShape`;
  - `QGraphicsItem::ItemIgnoresTransformations` — объект игнорирует наследуемые трансформации;
  - `QGraphicsItem::ItemIgnoresParentOpacity` — объект игнорирует прозрачность родителя;
  - `QGraphicsItem::ItemDoesntPropagateOpacityToChildren` — прозрачность объекта не распространяется на потомков;
  - `QGraphicsItem::ItemStacksBehindParent` — объект располагается позади родителя;

- `QGraphicsItem::ItemUsesExtendedStyleOption`;
- `QGraphicsItem::ItemHasNoContents`;
- `QGraphicsItem::ItemSendsGeometryChanges`;
- `QGraphicsItem::ItemAcceptsInputMethod`;
- `QGraphicsItem::ItemNegativeZStacksBehindParent`;
- `QGraphicsItem::ItemIsPanel` — объект является панелью;
- `QGraphicsItem::ItemSendsScenePositionChanges`;

❐ `setFlags()` — устанавливает несколько флагов. Константы (см. описание метода `setFlag()`) указываются через оператор `|`. Прототип метода:

```
void setFlags(QGraphicsItem::GraphicsItemFlags flags)
```

❐ `flags()` — возвращает комбинацию установленных флагов (см. описание метода `setFlag()`). Прототип метода:

```
QGraphicsItem::GraphicsItemFlags flags() const
```

❐ `setOpacity()` — задает степень прозрачности объекта. В качестве значения указывается вещественное число от `0.0` (полностью прозрачный) до `1.0` (полностью непрозрачный). Прототип метода:

```
void setOpacity(qreal opacity)
```

❐ `opacity()` — возвращает степень прозрачности объекта. Прототип метода:

```
qreal opacity() const
```

❐ `setToolTip()` — задает текст всплывающей подсказки. Прототип метода:

```
void setToolTip(const QString &toolTip)
```

❐ `setCursor()` — задает внешний вид указателя мыши при наведении указателя на объект. Прототип метода:

```
void setCursor(const QCursor &cursor)
```

❐ `unsetCursor()` — отменяет изменение указателя мыши. Прототип метода:

```
void unsetCursor()
```

❐ `setVisible()` — если в качестве параметра указано значение `true`, то объект будет видим. Значение `false` скрывает объект. Прототип метода:

```
void setVisible(bool visible)
```

❐ `show()` — делает объект видимым. Прототип метода:

```
void show()
```

❐ `hide()` — скрывает объект. Прототип метода:

```
void hide()
```

❐ `isVisible()` — возвращает значение `true`, если объект видим, и `false`, если скрыт. Прототип метода:

```
bool isVisible() const
```

- `setEnabled()` — если в качестве параметра указано значение `true`, то объект будет доступен. Значение `false` делает объект недоступным. Недоступный объект не получает никаких событий, и его нельзя выделить. Прототип метода:

  ```
  void setEnabled(bool enabled)
  ```

- `isEnabled()` — возвращает значение `true`, если объект доступен, и `false`, если недоступен. Прототип метода:

  ```
  bool isEnabled() const
  ```

- `setSelected()` — если в качестве параметра указано значение `true`, то объект будет выделен. Значение `false` снимает выделение. Чтобы объект можно было выделить, необходимо установить флаг `ItemIsSelectable`, например с помощью метода `setFlag()` из класса `QGraphicsItem`. Прототип метода:

  ```
  void setSelected(bool selected)
  ```

- `isSelected()` — возвращает значение `true`, если объект выделен, и `false` — в противном случае. Прототип метода:

  ```
  bool isSelected() const
  ```

- `setFocus()` — устанавливает фокус ввода на объект. В параметре `focusReason` можно указать причину изменения фокуса ввода. Чтобы объект мог принимать фокус ввода, необходимо установить флаг `ItemIsFocusable`, например с помощью метода `setFlag()` из класса `QGraphicsItem`. Прототип метода:

  ```
  void setFocus(Qt::FocusReason focusReason = Qt::OtherFocusReason)
  ```

- `clearFocus()` — убирает фокус ввода с объекта. Прототип метода:

  ```
  void clearFocus()
  ```

- `hasFocus()` — возвращает значение `true`, если объект находится в фокусе ввода, и `false` — в противном случае. Прототип метода:

  ```
  bool hasFocus() const
  ```

- `grabKeyboard()` — захватывает ввод с клавиатуры. Прототип метода:

  ```
  void grabKeyboard()
  ```

- `ungrabKeyboard()` — освобождает ввод с клавиатуры. Прототип метода:

  ```
  void ungrabKeyboard()
  ```

- `grabMouse()` — захватывает мышь. Прототип метода:

  ```
  void grabMouse()
  ```

- `ungrabMouse()` — освобождает мышь. Прототип метода:

  ```
  void ungrabMouse()
  ```

9.3.2. Трансформация объекта

Произвести трансформацию графического объекта позволяют следующие методы из класса `QGraphicsItem`:

- `setTransformOriginPoint()` — перемещает начало координат в указанную точку. Прототипы метода:

```
void setTransformOriginPoint(qreal x, qreal y)
void setTransformOriginPoint(const QPointF &origin)
```

- `setRotation()` — поворачивает объект на указанное количество градусов (указывается вещественное число). Положительное значение вызывает поворот по часовой стрелке, а отрицательное значение — против часовой стрелки. Прототип метода:

```
void setRotation(qreal angle)
```

- `rotation()` — возвращает текущий угол поворота. Прототип метода:

```
qreal rotation() const
```

- `setScale()` — масштабирует систему координат. Если значение меньше единицы, то выполняется уменьшение, а если больше единицы, то увеличение. Прототип метода:

```
void setScale(qreal factor)
```

- `scale()` — возвращает текущий масштаб. Прототип метода:

```
qreal scale() const
```

Установить матрицу трансформаций позволяет метод `setTransform()`, а получить матрицу можно с помощью метода `transform()`. Для получения матрицы трансформации сцены предназначен метод `sceneTransform()`. Сбросить все трансформации позволяет метод `resetTransform()`. Прототипы методов:

```
void setTransform(const QTransform &matrix, bool combine = false)
QTransform transform() const
QTransform sceneTransform() const
void resetTransform()
```

9.3.3. Прочие методы

Помимо рассмотренных методов класс `QGraphicsItem` содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setParentItem()` — задает родительский объект. Местоположение дочернего объекта задается в координатах родительского объекта. Прототип метода:

```
void setParentItem(QGraphicsItem *newParent)
```

- `parentItem()` — возвращает указатель на родительский объект. Прототип метода:

```
QGraphicsItem *parentItem() const
```

- `topLevelItem()` — возвращает указатель на родительский объект верхнего уровня. Прототип метода:

```
QGraphicsItem *topLevelItem() const
```

- `childItems()` — возвращает список с указателями на дочерние объекты. Прототип метода:

```
QList<QGraphicsItem *> childItems() const
```

- `collidingItems()` — возвращает список с указателями на объекты, которые сталкиваются с данным объектом. Если таких объектов нет, то метод возвращает пустой список. Возможные значения параметра `mode` см. в *разд. 9.1.4*. Прототип метода:

```
QList<QGraphicsItem *> collidingItems(
   Qt::ItemSelectionMode mode = Qt::IntersectsItemShape) const
```

- `collidesWithItem()` — возвращает значение `true`, если данный объект сталкивается с объектом, указанным в первом параметре. Возможные значения параметра `mode` см. в *разд. 9.1.4*. Прототип метода:

```
virtual bool collidesWithItem(const QGraphicsItem *other,
   Qt::ItemSelectionMode mode = Qt::IntersectsItemShape) const
```

- `ensureVisible()` — прокручивает область таким образом, чтобы указанный прямоугольник находился в видимой области представления. Прототипы метода:

```
void ensureVisible(qreal x, qreal y, qreal w, qreal h,
                   int xmargin = 50, int ymargin = 50)
void ensureVisible(const QRectF &rect = QRectF(),
                   int xmargin = 50, int ymargin = 50)
```

- `update()` — вызывает перерисовку указанной прямоугольной области. Прототипы метода:

```
void update(qreal x, qreal y, qreal width, qreal height)
void update(const QRectF &rect = QRectF())
```

9.4. Графические объекты

Вместо создания собственного объекта путем наследования класса `QGraphicsItem` можно воспользоваться следующими стандартными классами:

- `QGraphicsLineItem` — линия;
- `QGraphicsRectItem` — прямоугольник;
- `QGraphicsPolygonItem` — многоугольник;
- `QGraphicsEllipseItem` — эллипс;
- `QGraphicsPixmapItem` — изображение;
- `QGraphicsSimpleTextItem` — простой текст;

- `QGraphicsTextItem` — форматированный текст;
- `QGraphicsPathItem` — путь;
- `QGraphicsSvgItem` — SVG-графика.

9.4.1. Линия

Класс `QGraphicsLineItem` описывает линию. Иерархия наследования:

```
QGraphicsItem — QGraphicsLineItem
```

Форматы конструктора класса:

```
#include <QGraphicsLineItem>
QGraphicsLineItem(QGraphicsItem *parent = nullptr)
QGraphicsLineItem(qreal x1, qreal y1, qreal x2, qreal y2,
                  QGraphicsItem *parent = nullptr)
QGraphicsLineItem(const QLineF &line, QGraphicsItem *parent = nullptr)
```

Класс `QGraphicsLineItem` наследует все методы из класса `QGraphicsItem` и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setLine()` — устанавливает линию. Прототипы метода:

  ```
  void setLine(qreal x1, qreal y1, qreal x2, qreal y2)
  void setLine(const QLineF &line)
  ```

- `line()` — возвращает экземпляр класса `QLineF`. Прототип метода:

  ```
  QLineF line() const
  ```

- `setPen()` — устанавливает перо. Прототип метода:

  ```
  void setPen(const QPen &pen)
  ```

9.4.2. Класс *QAbstractGraphicsShapeItem*

Класс `QAbstractGraphicsShapeItem` является базовым классом для графических фигур. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem
```

Класс `QAbstractGraphicsShapeItem` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setPen()` — устанавливает перо. Прототип метода:

  ```
  void setPen(const QPen &pen)
  ```

- `setBrush()` — устанавливает кисть. Прототип метода:

  ```
  void setBrush(const QBrush &brush)
  ```

9.4.3. Прямоугольник

Класс `QGraphicsRectItem` описывает прямоугольник. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsRectItem
```

Форматы конструктора класса:

```
#include <QGraphicsRectItem>
QGraphicsRectItem(QGraphicsItem *parent = nullptr)
QGraphicsRectItem(qreal x, qreal y, qreal width, qreal height,
                  QGraphicsItem *parent = nullptr)
QGraphicsRectItem(const QRectF &rect, QGraphicsItem *parent = nullptr)
```

Класс `QGraphicsRectItem` наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setRect()` — устанавливает прямоугольник. Прототипы метода:

  ```
  void setRect(qreal x, qreal y, qreal width, qreal height)
  void setRect(const QRectF &rectangle)
  ```

- `rect()` — возвращает экземпляр класса `QRectF`. Прототип метода:

  ```
  QRectF rect() const
  ```

9.4.4. Многоугольник

Класс `QGraphicsPolygonItem` описывает многоугольник. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsPolygonItem
```

Форматы конструктора класса:

```
#include <QGraphicsPolygonItem>
QGraphicsPolygonItem(QGraphicsItem *parent = nullptr)
QGraphicsPolygonItem(const QPolygonF &polygon,
                     QGraphicsItem *parent = nullptr)
```

Класс `QGraphicsPolygonItem` наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setPolygon()` — устанавливает многоугольник. Прототип метода:

  ```
  void setPolygon(const QPolygonF &polygon)
  ```

- `polygon()` — возвращает экземпляр класса `QPolygonF`. Прототип метода:

  ```
  QPolygonF polygon() const
  ```

9.4.5. Эллипс

Класс `QGraphicsEllipseItem` описывает эллипс. Иерархия наследования:

```
QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsEllipseItem
```

Форматы конструктора класса:

```
#include <QGraphicsEllipseItem>
QGraphicsEllipseItem(QGraphicsItem *parent = nullptr)
QGraphicsEllipseItem(qreal x, qreal y, qreal width, qreal height,
                     QGraphicsItem *parent = nullptr)
QGraphicsEllipseItem(const QRectF &rect, QGraphicsItem *parent = nullptr)
```

Класс `QGraphicsEllipseItem` наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setRect()` — устанавливает прямоугольник, в который необходимо вписать эллипс. Прототипы метода:

  ```
  void setRect(qreal x, qreal y, qreal width, qreal height)
  void setRect(const QRectF &rect)
  ```

- `rect()` — возвращает экземпляр класса `QRectF`. Прототип метода:

  ```
  QRectF rect() const
  ```

- `setStartAngle()` и `setSpanAngle()` — задают начальный и конечный углы сектора соответственно. Следует учитывать, что величины углов задаются в значениях `1/16` градуса. Полный круг эквивалентен значению `16 * 360`. Нулевой угол находится в позиции трех часов. Положительные значения углов отсчитываются против часовой стрелки, а отрицательные — по часовой стрелке. Прототипы методов:

  ```
  void setStartAngle(int angle)
  void setSpanAngle(int angle)
  ```

- `startAngle()` и `spanAngle()` — возвращают значения начального и конечного углов сектора соответственно. Прототипы методов:

  ```
  int startAngle() const
  int spanAngle() const
  ```

9.4.6. Изображение

Класс `QGraphicsPixmapItem` описывает изображение. Иерархия наследования:

```
QGraphicsItem — QGraphicsPixmapItem
```

Форматы конструктора класса:

```
#include <QGraphicsPixmapItem>
QGraphicsPixmapItem(QGraphicsItem *parent = nullptr)
QGraphicsPixmapItem(const QPixmap &pixmap, QGraphicsItem *parent = nullptr)
```

Класс `QGraphicsPixmapItem` наследует все методы из класса `QGraphicsItem` и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setPixmap()` — устанавливает изображение. Прототип метода:

```
void setPixmap(const QPixmap &pixmap)
```

- `pixmap()` — возвращает экземпляр класса `QPixmap`. Прототип метода:

```
QPixmap pixmap() const
```

- `setOffset()` — задает местоположение изображения. Прототипы метода:

```
void setOffset(qreal x, qreal y)
void setOffset(const QPointF &offset)
```

- `offset()` — возвращает местоположение изображения (экземпляр класса `QPointF`). Прототип метода:

```
QPointF offset() const
```

- `setShapeMode()` — задает режим определения формы изображения. Прототип метода:

```
void setShapeMode(QGraphicsPixmapItem::ShapeMode mode)
```

  В качестве параметра могут быть указаны следующие константы:

  - `QGraphicsPixmapItem::MaskShape` — используется результат выполнения метода `mask()` из класса `QPixmap` (значение по умолчанию);
  - `QGraphicsPixmapItem::BoundingRectShape` — форма определяется по контуру изображения;
  - `QGraphicsPixmapItem::HeuristicMaskShape` — используется результат выполнения метода `createHeuristicMask()` из класса `QPixmap`;

- `setTransformationMode()` — задает режим сглаживания. Прототип метода:

```
void setTransformationMode(Qt::TransformationMode mode)
```

  В качестве параметра могут быть указаны следующие константы:

  - `Qt::FastTransformation` — сглаживание выключено (по умолчанию);
  - `Qt::SmoothTransformation` — сглаживание включено.

9.4.7. Простой текст

Класс `QGraphicsSimpleTextItem` описывает простой текст. Иерархия наследования:

`QGraphicsItem — QAbstractGraphicsShapeItem — QGraphicsSimpleTextItem`

Форматы конструктора класса:

```
#include <QGraphicsSimpleTextItem>
QGraphicsSimpleTextItem(QGraphicsItem *parent = nullptr)
QGraphicsSimpleTextItem(const QString &text, QGraphicsItem *parent = nullptr)
```

Класс `QGraphicsSimpleTextItem` наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setText()` — задает текст. Прототип метода:

```
void setText(const QString &text)
```

- `text()` — возвращает текст. Прототип метода:

```
QString text() const
```

- `setFont()` — устанавливает шрифт. Прототип метода:

```
void setFont(const QFont &font)
```

- `font()` — возвращает объект шрифта. Прототип метода:

```
QFont font() const
```

9.4.8. Форматированный текст

Класс `QGraphicsTextItem` описывает форматированный текст. Иерархия наследования:

```
(QObject, QGraphicsItem) — QGraphicsObject — QGraphicsTextItem
```

Форматы конструктора класса:

```
#include <QGraphicsTextItem>
QGraphicsTextItem(QGraphicsItem *parent = nullptr)
QGraphicsTextItem(const QString &text, QGraphicsItem *parent = nullptr)
```

Класс `QGraphicsTextItem` наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setPlainText()` — задает простой текст. Прототип метода:

```
void setPlainText(const QString &text)
```

- `toPlainText()` — возвращает простой текст. Прототип метода:

```
QString toPlainText() const
```

- `setHtml()` — задает HTML-текст. Прототип метода:

```
void setHtml(const QString &text)
```

- `toHtml()` — возвращает HTML-текст. Прототип метода:

```
QString toHtml() const
```

- `setFont()` — устанавливает шрифт. Прототип метода:

```
void setFont(const QFont &font)
```

- `font()` — возвращает объект шрифта. Прототип метода:

```
QFont font() const
```

- `setDefaultTextColor()` — задает цвет шрифта по умолчанию. Прототип метода:

```
void setDefaultTextColor(const QColor &col)
```

- `setTextWidth()` — задает предпочитаемую ширину строки. Если текст не помещается в установленную ширину, то он будет перенесен на новую строку. Прототип метода:

```
void setTextWidth(qreal width)
```

- `textWidth()` — возвращает предпочитаемую ширину текста. Прототип метода:

```
qreal textWidth() const
```

- `setDocument()` — устанавливает объект документа. Прототип метода:

```
void setDocument(QTextDocument *document)
```

- `document()` — возвращает указатель на объект документа. Прототип метода:

```
QTextDocument *document() const
```

- `setTextCursor()` — устанавливает объект курсора. Прототип метода:

```
void setTextCursor(const QTextCursor &cursor)
```

- `textCursor()` — возвращает объект курсора. Прототип метода:

```
QTextCursor textCursor() const
```

- `setTextInteractionFlags()` — задает режим взаимодействия пользователя с текстом. По умолчанию используется режим `NoTextInteraction`, при котором пользователь не может взаимодействовать с текстом. Прототип метода:

```
void setTextInteractionFlags(Qt::TextInteractionFlags flags)
```

- `setTabChangesFocus()` — если в качестве параметра указано значение `false`, то с помощью нажатия клавиши <Tab> можно вставить символ табуляции. Если указано значение `true`, то клавиша <Tab> используется для передачи фокуса. Прототип метода:

```
void setTabChangesFocus(bool b)
```

- `setOpenExternalLinks()` — если в качестве параметра указано значение `true`, то щелчок на гиперссылке приведет к открытию браузера, используемого в системе по умолчанию, и загрузке указанной страницы. Метод работает только при использовании режима `TextBrowserInteraction`. Прототип метода:

```
void setOpenExternalLinks(bool open)
```

Класс `QGraphicsTextItem` содержит следующие сигналы:

- `linkActivated(const QString&)` — генерируется при переходе по гиперссылке. Через параметр внутри обработчика доступен URL-адрес;
- `linkHovered(const QString&)` — генерируется при наведении указателя мыши на гиперссылку и выведении указателя. Через параметр внутри обработчика доступен URL-адрес или пустая строка.

9.5. Группировка объектов

Объединить несколько объектов в группу позволяет класс `QGraphicsItemGroup`. После группировки над объектами можно выполнять различные преобразования, например перемещать одновременно все объекты группы. Иерархия наследования для класса `QGraphicsItemGroup` выглядит так:

```
QGraphicsItem — QGraphicsItemGroup
```

Формат конструктора класса:

```
#include <QGraphicsItemGroup>
QGraphicsItemGroup(QGraphicsItem *parent = nullptr)
```

Класс `QGraphicsItemGroup` наследует все методы из класса `QGraphicsItem` и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `addToGroup()` — добавляет объект в группу. Прототип метода:

  ```
  void addToGroup(QGraphicsItem *item)
  ```

- `removeFromGroup()` — удаляет объект из группы. Прототип метода:

  ```
  void removeFromGroup(QGraphicsItem *item)
  ```

Создать группу и добавить ее на сцену можно также с помощью метода `createItemGroup()` из класса `QGraphicsScene`. Метод возвращает указатель на группу (экземпляр класса `QGraphicsItemGroup`). Удалить группу со сцены позволяет метод `destroyItemGroup()` из класса `QGraphicsScene`. Прототипы методов:

```
QGraphicsItemGroup *createItemGroup(const QList<QGraphicsItem *> &items)
void destroyItemGroup(QGraphicsItemGroup *group)
```

Добавить объект в группу позволяет также метод `setGroup()` из класса `QGraphicsItem`. Получить указатель на группу, к которой прикреплен объект, можно с помощью метода `group()` из класса `QGraphicsItem`. Если объект не прикреплен к группе, то метод возвращает нулевой указатель. Прототипы методов:

```
void setGroup(QGraphicsItemGroup *group)
QGraphicsItemGroup *group() const
```

9.6. Эффекты

К графическим объектам можно применить различные эффекты, например: изменить прозрачность или цвет, отобразить тень или сделать объект размытым. Наследуя класс `QGraphicsEffect` и переопределяя метод `draw()`, можно создать свой эффект.

Для установки эффекта и получения указателя на него предназначены следующие методы из класса `QGraphicsItem`:

- `setGraphicsEffect()` — устанавливает эффект. Прототип метода:

  ```
  void setGraphicsEffect(QGraphicsEffect *effect)
  ```

- `graphicsEffect()` — возвращает указатель на эффект или нулевой указатель, если эффект не был установлен. Прототип метода:

```
QGraphicsEffect *graphicsEffect() const
```

9.6.1. Класс *QGraphicsEffect*

Класс `QGraphicsEffect` является базовым классом для всех эффектов. Иерархия наследования выглядит так:

```
QObject — QGraphicsEffect
```

Формат конструктора класса:

```
#include <QGraphicsEffect>
QGraphicsEffect(QObject *parent = nullptr)
```

Класс `QGraphicsEffect` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `draw()` — производит рисование эффекта. Этот абстрактный метод должен быть переопределен в производных классах. Прототип метода:

```
virtual void draw(QPainter *painter) = 0
```

- `setEnabled()` — если в качестве параметра указано значение `false`, то эффект отключен. Значение `true` разрешает использование эффекта. Метод является слотом. Прототип метода:

```
void setEnabled(bool enable)
```

- `isEnabled()` — возвращает значение `true`, если эффект разрешено использовать, и `false` — в противном случае. Прототип метода:

```
bool isEnabled() const
```

- `update()` — вызывает перерисовку эффекта. Метод является слотом. Прототип метода:

```
void update()
```

Класс `QGraphicsEffect` содержит сигнал `enabledChanged(bool)`, который генерируется при изменении статуса эффекта. Внутри обработчика через параметр доступно значение `true`, если эффект разрешено использовать, и `false` — в противном случае.

9.6.2. Тень

Класс `QGraphicsDropShadowEffect` реализует тень. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsDropShadowEffect
```

Формат конструктора класса:

```
#include <QGraphicsDropShadowEffect>
QGraphicsDropShadowEffect(QObject *parent = nullptr)
```

Класс `QGraphicsDropShadowEffect` наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setColor()` — задает цвет тени. Метод является слотом. Прототип метода:

```
void setColor(const QColor &color)
```

- `color()` — возвращает цвет тени. Прототип метода:

```
QColor color() const
```

- `setBlurRadius()` — задает радиус размытия тени. Метод является слотом. Прототип метода:

```
void setBlurRadius(qreal blurRadius)
```

- `blurRadius()` — возвращает радиус размытия тени. Прототип метода:

```
qreal blurRadius() const
```

- `setOffset()` — задает смещение тени. Метод является слотом. Прототипы метода:

```
void setOffset(qreal d)
void setOffset(qreal dx, qreal dy)
void setOffset(const QPointF &ofs)
```

- `offset()` — возвращает смещение тени. Прототип метода:

```
QPointF offset() const
```

- `setXOffset()` — задает смещение по оси `X`. Метод является слотом. Прототип метода:

```
void setXOffset(qreal dx)
```

- `xOffset()` — возвращает смещение по оси `X`. Прототип метода:

```
qreal xOffset() const
```

- `setYOffset()` — задает смещение по оси `Y`. Метод является слотом. Прототип метода:

```
void setYOffset(qreal dy)
```

- `yOffset()` — возвращает смещение по оси `Y`. Прототип метода:

```
qreal yOffset() const
```

Класс `QGraphicsDropShadowEffect` содержит следующие сигналы:

- `colorChanged(const QColor&)` — генерируется при изменении цвета тени. Внутри обработчика через параметр доступен новый цвет (экземпляр класса `QColor`);
- `blurRadiusChanged(qreal)` — генерируется при изменении радиуса размытия. Внутри обработчика через параметр доступно новое значение;
- `offsetChanged(const QPointF&)` — генерируется при изменении смещения. Внутри обработчика через параметр доступно новое значение (экземпляр класса `QPointF`).

9.6.3. Размытие

Класс `QGraphicsBlurEffect` реализует эффект размытия. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsBlurEffect
```

Формат конструктора класса:

```
#include <QGraphicsBlurEffect>
QGraphicsBlurEffect(QObject *parent = nullptr)
```

Класс `QGraphicsBlurEffect` наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setBlurRadius()` — задает радиус размытия. Метод является слотом. Прототип метода:

  ```
  void setBlurRadius(qreal blurRadius)
  ```

- `blurRadius()` — возвращает радиус размытия. Прототип метода:

  ```
  qreal blurRadius() const
  ```

Класс `QGraphicsBlurEffect` содержит сигнал `blurRadiusChanged(qreal)`, который генерируется при изменении радиуса размытия. Внутри обработчика через параметр доступно новое значение.

9.6.4. Изменение цвета

Класс `QGraphicsColorizeEffect` реализует эффект изменения цвета. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsColorizeEffect
```

Формат конструктора класса:

```
#include <QGraphicsColorizeEffect>
QGraphicsColorizeEffect(QObject *parent = nullptr)
```

Класс `QGraphicsColorizeEffect` наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setColor()` — задает цвет. Метод является слотом. Прототип метода:

  ```
  void setColor(const QColor &c)
  ```

- `color()` — возвращает текущий цвет. Прототип метода:

  ```
  QColor color() const
  ```

- `setStrength()` — задает интенсивность цвета. В качестве значения указывается вещественное число от `0.0` до `1.0` (значение по умолчанию). Метод является слотом. Прототип метода:

  ```
  void setStrength(qreal strength)
  ```

- `strength()` — возвращает интенсивность цвета. Прототип метода:

```
qreal strength() const
```

Класс `QGraphicsColorizeEffect` содержит следующие сигналы:

- `colorChanged(const QColor&)` — генерируется при изменении цвета. Внутри обработчика через параметр доступен новый цвет (экземпляр класса `QColor`);
- `strengthChanged(qreal)` — генерируется при изменении интенсивности цвета. Внутри обработчика через параметр доступно новое значение.

9.6.5. Изменение прозрачности

Класс `QGraphicsOpacityEffect` реализует эффект прозрачности. Иерархия наследования:

```
QObject — QGraphicsEffect — QGraphicsOpacityEffect
```

Формат конструктора класса:

```
#include <QGraphicsOpacityEffect>
QGraphicsOpacityEffect(QObject *parent = nullptr)
```

Класс `QGraphicsOpacityEffect` наследует все методы из базовых классов и содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setOpacity()` — задает степень прозрачности. В качестве значения указывается вещественное число от `0.0` до `1.0`. По умолчанию используется значение `0.7`. Метод является слотом. Прототип метода:

```
void setOpacity(qreal opacity)
```

- `opacity()` — возвращает степень прозрачности. Прототип метода:

```
qreal opacity() const
```

- `setOpacityMask()` — задает маску. Метод является слотом. Прототип метода:

```
void setOpacityMask(const QBrush &mask)
```

- `opacityMask()` — возвращает маску. Прототип метода:

```
QBrush opacityMask() const
```

Класс `QGraphicsOpacityEffect` содержит следующие сигналы:

- `opacityChanged(qreal)` — генерируется при изменении степени прозрачности. Внутри обработчика через параметр доступно новое значение;
- `opacityMaskChanged(const QBrush&)` — генерируется при изменении маски.

9.7. Обработка событий

Первоначально события получают представление, затем представление преобразует события и передает их объекту сцены. В свою очередь, сцена передает событие объекту, который способен обработать данное событие. Например, щелчок мыши передается объекту, который расположен по координатам щелчка.

Обработка событий в классе представления ничем не отличается от обычной обработки событий, рассмотренной в *главе 4*. Обработка событий в классе объекта имеет свои отличия, которые мы и рассмотрим в этом разделе.

9.7.1. События клавиатуры

При обработке событий клавиатуры следует учитывать, что:

- графический объект должен иметь возможность принимать фокус ввода. Для этого необходимо установить флаг `ItemIsFocusable`, например с помощью метода `setFlag()` из класса `QGraphicsItem`;
- объект должен быть в фокусе ввода. Методы, позволяющие управлять фокусом ввода, мы рассматривали в *разд. 9.1.5* и *9.3.1*;
- чтобы захватить эксклюзивный ввод с клавиатуры, следует воспользоваться методом `grabKeyboard()`, а чтобы освободить ввод — методом `ungrabKeyboard()`;
- можно перехватить нажатие любых клавиш, кроме клавиши <Tab> и комбинации <Shift>+<Tab>. Эти клавиши используются для передачи фокуса следующему и предыдущему объектам соответственно;
- если событие обработано, то нужно вызвать метод `accept()` через объект события. В противном случае необходимо вызвать метод `ignore()`.

Для обработки событий клавиатуры следует наследовать класс, реализующий графический объект, и переопределить следующие методы:

- `focusInEvent()` — вызывается при получении фокуса ввода. Прототип метода:

```
virtual void focusInEvent(QFocusEvent *event)
```

- `focusOutEvent()` — вызывается при потере фокуса ввода. Прототип метода:

```
virtual void focusOutEvent(QFocusEvent *event)
```

- `keyPressEvent()` — вызывается при нажатии клавиши клавиатуры. Если клавишу удерживать нажатой, то событие генерируется постоянно, пока клавиша не будет отпущена. Прототип метода:

```
virtual void keyPressEvent(QKeyEvent *event)
```

- `keyReleaseEvent()` — вызывается при отпускании ранее нажатой клавиши. Прототип метода:

```
virtual void keyReleaseEvent(QKeyEvent *event)
```

С помощью метода setFocusProxy() из класса QGraphicsItem можно указать объект, который будет обрабатывать события клавиатуры вместо текущего объекта. Получить указатель на такой объект позволяет метод focusProxy(). Прототипы методов:

```
void setFocusProxy(QGraphicsItem *item)
QGraphicsItem *focusProxy() const
```

9.7.2. События мыши

Для обработки нажатия кнопки мыши и перемещения мыши следует наследовать класс, реализующий графический объект, и переопределить следующие методы:

- mousePressEvent() — вызывается при нажатии кнопки мыши над объектом. Если событие принято, то необходимо вызвать метод accept() через объект события, в противном случае следует вызвать метод ignore(). Если вызван метод ignore(), то методы mouseReleaseEvent() и mouseMoveEvent() вызваны не будут. Прототип метода:

  ```
  virtual void mousePressEvent(QGraphicsSceneMouseEvent *event)
  ```

  С помощью метода setAcceptedMouseButtons() из класса QGraphicsItem можно указать кнопки, события от которых объект принимает. По умолчанию объект принимает события от всех кнопок мыши. Если в параметре buttons указать константу NoButton, то обработка событий мыши будет отключена. Прототип метода:

  ```
  void setAcceptedMouseButtons(Qt::MouseButtons buttons)
  ```

- mouseReleaseEvent() — вызывается при отпускании ранее нажатой кнопки мыши. Прототип метода:

  ```
  virtual void mouseReleaseEvent(QGraphicsSceneMouseEvent *event)
  ```

- mouseDoubleClickEvent() — вызывается при двойном щелчке мышью в области объекта. Прототип метода:

  ```
  virtual void mouseDoubleClickEvent(QGraphicsSceneMouseEvent *event)
  ```

- mouseMoveEvent() — вызывается при перемещении мыши с нажатой кнопкой. Прототип метода:

  ```
  virtual void mouseMoveEvent(QGraphicsSceneMouseEvent *event)
  ```

Класс QGraphicsSceneMouseEvent наследует все методы из классов QGraphicsSceneEvent и QEvent и добавляет следующие методы:

- pos() — возвращает экземпляр класса QPointF с координатами в пределах области объекта. Прототип метода:

  ```
  QPointF pos() const
  ```

- scenePos() — возвращает экземпляр класса QPointF с координатами в пределах сцены. Прототип метода:

  ```
  QPointF scenePos() const
  ```

- `screenPos()` — возвращает экземпляр класса `QPoint` с координатами в пределах экрана. Прототип метода:

```
QPoint screenPos() const
```

- `lastPos()` — возвращает экземпляр класса `QPointF` с координатами последней запомненной представлением позиции в пределах области объекта. Прототип метода:

```
QPointF lastPos() const
```

- `lastScenePos()` — возвращает экземпляр класса `QPointF` с координатами последней запомненной представлением позиции в пределах сцены. Прототип метода:

```
QPointF lastScenePos() const
```

- `lastScreenPos()` — возвращает экземпляр класса `QPoint` с координатами последней запомненной представлением позиции в пределах экрана. Прототип метода:

```
QPoint lastScreenPos() const
```

- `buttonDownPos()` — возвращает экземпляр класса `QPointF` с координатами щелчка указанной кнопки мыши в пределах области объекта. Прототип метода:

```
QPointF buttonDownPos(Qt::MouseButton button) const
```

- `buttonDownScenePos()` — возвращает экземпляр класса `QPointF` с координатами щелчка указанной кнопки мыши в пределах сцены. Прототип метода:

```
QPointF buttonDownScenePos(Qt::MouseButton button) const
```

- `buttonDownScreenPos()` — возвращает экземпляр класса `QPoint` с координатами щелчка указанной кнопки мыши в пределах экрана. Прототип метода:

```
QPoint buttonDownScreenPos(Qt::MouseButton button) const
```

- `button()` — позволяет определить, какая кнопка мыши вызвала событие. Прототип метода:

```
Qt::MouseButton button() const
```

- `buttons()` — позволяет определить все кнопки, которые нажаты одновременно. Прототип метода:

```
Qt::MouseButtons buttons() const
```

- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты вместе с кнопкой мыши. Прототип метода:

```
Qt::KeyboardModifiers modifiers() const
```

По умолчанию событие мыши перехватывает объект, над которым произведен щелчок мышью. Чтобы перехватывать нажатие и отпускание мыши вне объекта, следует захватить мышь с помощью метода `grabMouse()` из класса `QGraphicsItem`. Освободить захваченную ранее мышь позволяет метод `ungrabMouse()`. Получить указатель на объект, захвативший мышь, можно с помощью метода `mouseGrabberItem()` из класса `QGraphicsScene`. Прототип метода:

```
QGraphicsItem *mouseGrabberItem() const
```

Для обработки прочих событий мыши следует наследовать класс, реализующий графический объект, и переопределить следующие методы:

- `hoverEnterEvent()` — вызывается при наведении указателя мыши на область объекта. Прототип метода:

```
virtual void hoverEnterEvent(QGraphicsSceneHoverEvent *event)
```

- `hoverLeaveEvent()` — вызывается, когда указатель мыши покидает область объекта. Прототип метода:

```
virtual void hoverLeaveEvent(QGraphicsSceneHoverEvent *event)
```

- `hoverMoveEvent()` — вызывается при перемещении указателя мыши внутри области объекта. Прототип метода:

```
virtual void hoverMoveEvent(QGraphicsSceneHoverEvent *event)
```

- `wheelEvent()` — вызывается при повороте колесика мыши при нахождении указателя мыши над объектом. Чтобы обрабатывать событие в любом случае, следует захватить мышь. Прототип метода:

```
virtual void wheelEvent(QGraphicsSceneWheelEvent *event)
```

Следует учитывать, что методы `hoverEnterEvent()`, `hoverLeaveEvent()` и `hoverMoveEvent()` будут вызваны только в том случае, если обработка этих событий разрешена. Чтобы разрешить обработку событий перемещения мыши, следует вызвать метод `setAcceptHoverEvents()` из класса `QGraphicsItem` и передать ему значение `true`. Значение `false` запрещает обработку событий перемещения указателя. Получить текущее состояние позволяет метод `acceptHoverEvents()`. Прототипы методов:

```
void setAcceptHoverEvents(bool enabled)
bool acceptHoverEvents() const
```

Класс `QGraphicsSceneHoverEvent` наследует все методы из классов `QGraphicsSceneEvent` и `QEvent` и добавляет следующие методы:

- `pos()` — возвращает экземпляр класса `QPointF` с координатами в пределах области объекта. Прототип метода:

```
QPointF pos() const
```

- `scenePos()` — возвращает экземпляр класса `QPointF` с координатами в пределах сцены. Прототип метода:

```
QPointF scenePos() const
```

- `screenPos()` — возвращает экземпляр класса `QPoint` с координатами в пределах экрана. Прототип метода:

```
QPoint screenPos() const
```

- `lastPos()` — возвращает экземпляр класса `QPointF` с координатами последней запомненной представлением позиции в пределах области объекта. Прототип метода:

```
QPointF lastPos() const
```

- `lastScenePos()` — возвращает экземпляр класса `QPointF` с координатами последней запомненной представлением позиции в пределах сцены. Прототип метода:

  ```
  QPointF lastScenePos() const
  ```

- `lastScreenPos()` — возвращает экземпляр класса `QPoint` с координатами последней запомненной представлением позиции в пределах экрана. Прототип метода:

  ```
  QPoint lastScreenPos() const
  ```

- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты. Прототип метода:

  ```
  Qt::KeyboardModifiers modifiers() const
  ```

Класс `QGraphicsSceneWheelEvent` наследует все методы из классов `QGraphicsSceneEvent` и `QEvent` и добавляет следующие методы:

- `delta()` — возвращает расстояние поворота колесика. Прототип метода:

  ```
  int delta() const
  ```

- `orientation()` — возвращает ориентацию в виде значения одной из констант:

  - `Qt::Horizontal` — по горизонтали;
  - `Qt::Vertical` — по вертикали.

  Прототип метода:

  ```
  Qt::Orientation orientation() const
  ```

- `pos()` — возвращает экземпляр класса `QPointF` с координатами указателя мыши в пределах области объекта. Прототип метода:

  ```
  QPointF pos() const
  ```

- `scenePos()` — возвращает экземпляр класса `QPointF` с координатами указателя мыши в пределах сцены. Прототип метода:

  ```
  QPointF scenePos() const
  ```

- `screenPos()` — возвращает экземпляр класса `QPoint` с координатами указателя мыши в пределах экрана. Прототип метода:

  ```
  QPoint screenPos() const
  ```

- `buttons()` — позволяет определить кнопки, которые нажаты одновременно с поворотом колесика. Прототип метода:

  ```
  Qt::MouseButtons buttons() const
  ```

- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты. Прототип метода:

  ```
  Qt::KeyboardModifiers modifiers() const
  ```

9.7.3. Обработка перетаскивания и сброса

Прежде чем обрабатывать перетаскивание и сброс, необходимо сообщить системе, что графический объект может обрабатывать эти события. Для этого следует вызвать метод `setAcceptDrops()` из класса `QGraphicsItem` и передать ему значение `true`. Прототип метода:

```
void setAcceptDrops(bool on)
```

Обработка перетаскивания и сброса выполняется следующим образом:

- внутри метода `dragEnterEvent()` проверяется MIME-тип перетаскиваемых данных и действие. Если графический объект способен обработать сброс этих данных и соглашается с предложенным действием, то необходимо вызвать метод `acceptProposedAction()` через объект события. Если нужно изменить действие, то методу `setDropAction()` передается новое действие, а затем вызывается метод `accept()`, а не метод `acceptProposedAction()`;
- если необходимо ограничить область сброса некоторым участком графического объекта, то можно дополнительно определить метод `dragMoveEvent()`. Этот метод будет постоянно вызываться при перетаскивании внутри области графического объекта. При согласии со сбрасыванием следует вызвать метод `accept()`;
- внутри метода `dropEvent()` производится обработка сброса.

Обработать события, возникающие при перетаскивании и сбрасывании объектов, позволяют следующие методы:

- `dragEnterEvent()` — вызывается, когда перетаскиваемый объект входит в область графического объекта. Прототип метода:

  ```
  virtual void dragEnterEvent(QGraphicsSceneDragDropEvent *event)
  ```

- `dragLeaveEvent()` — вызывается, когда перетаскиваемый объект покидает область графического объекта. Прототип метода:

  ```
  virtual void dragLeaveEvent(QGraphicsSceneDragDropEvent *event)
  ```

- `dragMoveEvent()` — вызывается при перетаскивании объекта внутри области графического объекта. Прототип метода:

  ```
  virtual void dragMoveEvent(QGraphicsSceneDragDropEvent *event)
  ```

- `dropEvent()` — вызывается при сбрасывании объекта в области графического объекта. Прототип метода:

  ```
  virtual void dropEvent(QGraphicsSceneDragDropEvent *event)
  ```

Класс `QGraphicsSceneDragDropEvent` наследует все методы из классов `QGraphicsSceneEvent` и `QEvent` и добавляет следующие методы:

- `mimeData()` — возвращает указатель на экземпляр класса `QMimeData` с перемещаемыми данными и информацией о MIME-типе. Прототип метода:

  ```
  const QMimeData *mimeData() const
  ```

- `pos()` — возвращает экземпляр класса `QPointF` с координатами в пределах области объекта. Прототип метода:

```
QPointF pos() const
```

- `scenePos()` — возвращает экземпляр класса `QPointF` с координатами в пределах сцены. Прототип метода:

```
QPointF scenePos() const
```

- `screenPos()` — возвращает экземпляр класса `QPoint` с координатами в пределах экрана. Прототип метода:

```
QPoint screenPos() const
```

- `possibleActions()` — возвращает комбинацию возможных действий при сбрасывании. Прототип метода:

```
Qt::DropActions possibleActions() const
```

- `proposedAction()` — возвращает действие по умолчанию при сбрасывании. Прототип метода:

```
Qt::DropAction proposedAction() const
```

- `acceptProposedAction()` — устанавливает флаг готовности принять перемещаемые данные и согласия с действием, возвращаемым методом `proposedAction()`. Прототип метода:

```
void acceptProposedAction()
```

- `setDropAction()` — позволяет изменить действие при сбрасывании. После изменения действия следует вызвать метод `accept()`, а не `acceptProposedAction()`. Прототип метода:

```
void setDropAction(Qt::DropAction action)
```

- `dropAction()` — возвращает действие, которое должно быть выполнено при сбрасывании. Прототип метода:

```
Qt::DropAction dropAction() const
```

- `modifiers()` — позволяет определить, какие клавиши-модификаторы (<Shift>, <Ctrl>, <Alt> и др.) были нажаты. Прототип метода:

```
Qt::KeyboardModifiers modifiers() const
```

- `buttons()` — позволяет определить кнопки мыши, которые нажаты. Прототип метода:

```
Qt::MouseButtons buttons() const
```

- `source()` — возвращает указатель на источник события. Прототип метода:

```
QWidget *source() const
```

9.7.4. Фильтрация событий

События можно перехватывать еще до того, как они будут переданы специализированному методу. Для этого необходимо переопределить метод `sceneEvent()` в классе объекта. Через параметр `event` доступен объект с дополнительной информацией о событии. Тип этого объекта отличается для разных типов событий. Внутри метода следует вернуть значение `true`, если событие обработано, и `false` — в противном случае. Если вернуть значение `true`, то специализированный метод (например, `mousePressEvent()`) вызван не будет. Прототип метода:

```
virtual bool sceneEvent(QEvent *event)
```

Чтобы произвести фильтрацию событий какого-либо объекта, необходимо переопределить метод `sceneEventFilter()` в классе объекта. Прототип метода:

```
virtual bool sceneEventFilter(QGraphicsItem *watched, QEvent *event)
```

Через параметр `watched` доступен указатель на объект, а через параметр `event` — объект с дополнительной информацией о событии. Тип этого объекта отличается для разных типов событий. Внутри метода следует вернуть значение `true`, если событие обработано, и `false` — в противном случае. Если вернуть значение `true`, то объект не получит событие.

Указать, события какого объекта фильтруются, позволяют следующие методы из класса `QGraphicsItem`:

- `installSceneEventFilter()` — задает объект, который будет производить фильтрацию событий текущего объекта. Прототип метода:

  ```
  void installSceneEventFilter(QGraphicsItem *filterItem)
  ```

- `removeSceneEventFilter()` — удаляет фильтр. Прототип метода:

  ```
  void removeSceneEventFilter(QGraphicsItem *filterItem)
  ```

- `setFiltersChildEvents()` — если в качестве параметра указано значение `true`, то объект будет производить фильтрацию событий всех своих дочерних объектов. Прототип метода:

  ```
  void setFiltersChildEvents(bool enabled)
  ```

9.7.5. Обработка изменения состояния объекта

Чтобы обработать изменение состояния объекта, следует переопределить метод `itemChange()` в классе объекта. Прототип метода:

```
virtual QVariant itemChange(QGraphicsItem::GraphicsItemChange change,
                            const QVariant &value)
```

Метод должен возвращать новое значение. Через параметр `change` доступно состояние, которое было изменено, в виде значения одной из констант:

- `QGraphicsItem::ItemEnabledChange` — изменилось состояние доступности;
- `QGraphicsItem::ItemEnabledHasChanged` — изменилось состояние доступности. Возвращаемое значение игнорируется;

- `QGraphicsItem::ItemPositionChange` — изменилась позиция объекта. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`;
- `QGraphicsItem::ItemPositionHasChanged` — изменилась позиция объекта. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`. Возвращаемое значение игнорируется;
- `QGraphicsItem::ItemScenePositionHasChanged` — изменилась позиция объекта на сцене. Метод будет вызван, только если установлен флаг `ItemSendsScenePositionChanges`. Возвращаемое значение игнорируется;
- `QGraphicsItem::ItemTransformChange` — изменилась матрица трансформаций. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`;
- `QGraphicsItem::ItemTransformHasChanged` — изменилась матрица трансформаций. Метод будет вызван, только если установлен флаг `ItemSendsGeometryChanges`. Возвращаемое значение игнорируется;
- `QGraphicsItem::ItemRotationChange`;
- `QGraphicsItem::ItemRotationHasChanged`;
- `QGraphicsItem::ItemScaleChange`;
- `QGraphicsItem::ItemScaleHasChanged`;
- `QGraphicsItem::ItemTransformOriginPointChange`;
- `QGraphicsItem::ItemTransformOriginPointHasChanged`;
- `QGraphicsItem::ItemSelectedChange` — изменилось выделение объекта;
- `QGraphicsItem::ItemSelectedHasChanged` — изменилось выделение объекта. Возвращаемое значение игнорируется;
- `QGraphicsItem::ItemVisibleChange` — изменилось состояние видимости объекта;
- `QGraphicsItem::ItemVisibleHasChanged` — изменилось состояние видимости объекта. Возвращаемое значение игнорируется;
- `QGraphicsItem::ItemParentChange`;
- `QGraphicsItem::ItemParentHasChanged`;
- `QGraphicsItem::ItemChildAddedChange`;
- `QGraphicsItem::ItemChildRemovedChange`;
- `QGraphicsItem::ItemSceneChange`;
- `QGraphicsItem::ItemSceneHasChanged`;
- `QGraphicsItem::ItemCursorChange` — изменился курсор;
- `QGraphicsItem::ItemCursorHasChanged` — изменился курсор. Возвращаемое значение игнорируется;
- `QGraphicsItem::ItemToolTipChange` — изменилась всплывающая подсказка;
- `QGraphicsItem::ItemToolTipHasChanged` — изменилась всплывающая подсказка. Возвращаемое значение игнорируется;

- `QGraphicsItem::ItemFlagsChange` — изменились флаги;
- `QGraphicsItem::ItemFlagsHaveChanged` — изменились флаги. Возвращаемое значение игнорируется;
- `QGraphicsItem::ItemZValueChange` — изменилось положение по оси `Z`;
- `QGraphicsItem::ItemZValueHasChanged` — изменилось положение по оси `Z`. Возвращаемое значение игнорируется;
- `QGraphicsItem::ItemOpacityChange` — изменилась прозрачность объекта;
- `QGraphicsItem::ItemOpacityHasChanged` — изменилась прозрачность объекта. Возвращаемое значение игнорируется.

> ***Внимание!***
>
> Вызов некоторых методов из метода `itemChange()` может привести к рекурсии. За подробной информацией обращайтесь к документации по классу `QGraphicsItem`.

ГЛАВА 10

Диалоговые окна

Диалоговые окна предназначены для информирования пользователя, а также для получения данных от пользователя. В большинстве случаев диалоговые окна являются модальными (т. е. блокирующими все окна приложения или только родительское окно) и отображаются на непродолжительный промежуток времени. Для работы с диалоговыми окнами в Qt предназначен класс `QDialog`, который предоставляет множество специальных методов, позволяющих дождаться закрытия окна, определить статус завершения и многое другое. Класс `QDialog` наследуют другие классы, которые реализуют готовые диалоговые окна. Например, класс `QMessageBox` предоставляет готовые диалоговые окна для вывода сообщений, класс `QInputDialog` — для ввода данных, класс `QFileDialog` — для выбора каталога или файла и т. д.

> ***Примечание***
>
> Помимо рассмотренных в этой главе классов в Qt существуют классы для реализации диалога изменения настроек принтера (класс `QPrintDialog`) и предварительного просмотра перед печатью (класс `QPrintPreviewDialog`). За подробной информацией по этим классам обращайтесь к документации.

10.1. Пользовательские диалоговые окна

Класс `QDialog` реализует диалоговое окно. По умолчанию окно выводится с рамкой и заголовком, в котором расположена кнопка **Закрыть**. Размеры окна можно изменять с помощью мыши. Иерархия наследования для класса `QDialog` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog
```

Конструктор класса `QDialog` имеет следующий формат:

```
#include <QDialog>
QDialog(QWidget *parent = nullptr, Qt::WindowFlags f = Qt::WindowFlags())
```

В параметре `parent` передается указатель на родительское окно. Если параметр не указан или имеет значение `nullptr`, то диалоговое окно будет центрироваться относительно экрана. Если передан указатель на родительское окно, то диалоговое окно будет центрироваться относительно родительского окна. Кроме того, это позволяет

создать модальное диалоговое окно, которое будет блокировать только окно родителя, а не все окна приложения. Какие именно значения можно указать в параметре `f`, мы рассматривали в *разд. 3.2*. Тип окна по умолчанию — `Dialog`.

Класс `QDialog` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `exec()` — отображает модальное диалоговое окно и возвращает код возврата в виде значения следующих констант:
  - `QDialog::Accepted` — нажата кнопка **OK**;
  - `QDialog::Rejected` — нажата кнопка **Cancel**, кнопка **Закрыть** в заголовке окна или клавиша <Esc>.

  Метод является слотом. Прототип метода:

  ```
  virtual int exec()
  ```

  Пример отображения диалогового окна и обработки статуса внутри обработчика нажатия кнопки (класс `MyDialog` является наследником класса `QDialog`, а `this` — указатель на главное окно):

  ```
  void Widget::on_btn1_clicked()
  {
     MyDialog dialog(this);
     int result = dialog.exec();
     if (result == QDialog::Accepted) {
        qDebug() << dialog.lineEdit->text();
     }
     else {
        qDebug() << "Нажата кнопка Cancel" << result;
     }
  }
  ```

- `accept()` — скрывает модальное диалоговое окно и устанавливает код возврата равным значению константы `QDialog::Accepted`. Метод является слотом. Прототип метода:

  ```
  virtual void accept()
  ```

  Метод `accept()` следует соединить с сигналом нажатия кнопки **OK**:

  ```
  connect(btnOK, SIGNAL(clicked()), this, SLOT(accept()));
  ```

- `reject()` — скрывает модальное диалоговое окно и устанавливает код возврата равным значению константы `QDialog::Rejected`. Метод является слотом. Прототип метода:

  ```
  virtual void reject()
  ```

  Метод `reject()` следует соединить с сигналом нажатия кнопки **Cancel**:

  ```
  connect(btnCancel, SIGNAL(clicked()), this, SLOT(reject()));
  ```

- `done()` — скрывает модальное диалоговое окно и устанавливает код возврата равным значению параметра. Метод является слотом. Прототип метода:

  ```
  virtual void done(int)
  ```

- `setResult()` — устанавливает код возврата. Прототип метода:

  ```
  void setResult(int)
  ```

- `result()` — возвращает код завершения. Прототип метода:

  ```
  int result() const
  ```

- `setSizeGripEnabled()` — если в качестве параметра указано значение `true`, то в правом нижнем углу диалогового окна будет отображен значок изменения размера, а если `false`, то скрыт (значение по умолчанию). Прототип метода:

  ```
  void setSizeGripEnabled(bool)
  ```

- `isSizeGripEnabled()` — возвращает значение `true`, если значок изменения размера отображается в правом нижнем углу диалогового окна, и `false` — в противном случае. Прототип метода:

  ```
  bool isSizeGripEnabled() const
  ```

- `setVisible()` — если в параметре указано значение `true`, то диалоговое окно будет отображено, а если значение `false`, то скрыто. Вместо этого метода можно воспользоваться методами `show()` и `hide()` из класса `QWidget`. Прототип метода:

  ```
  void setVisible(bool)
  ```

- `open()` — отображает диалоговое окно в модальном режиме. Блокируется только родительское окно, а не все окна приложения. Метод является слотом. Прототип метода:

  ```
  virtual void open()
  ```

- `setModal()` — если в качестве параметра указано значение `true`, то окно будет модальным, а если `false`, то обычным. Обратите внимание на то, что окно, открываемое с помощью метода `exec()`, всегда будет модальным независимо от значения метода `setModal()`. Чтобы диалоговое окно было не модальным, следует отображать его с помощью метода `show()` или `setVisible()`. После вызова этих методов следует вызвать методы `raise()` (чтобы поместить окно поверх всех окон) и `activateWindow()` (чтобы сделать окно активным (имеющим фокус ввода)).

  Указать, что окно является модальным, позволяет также метод `setWindowModality()` из класса `QWidget`. Прототип метода:

  ```
  void setWindowModality(Qt::WindowModality windowModality)
  ```

  В качестве параметра могут быть указаны следующие константы:

  - `Qt::NonModal` — окно не является модальным;
  - `Qt::WindowModal` — окно блокирует только родительские окна в пределах иерархии;
  - `Qt::ApplicationModal` — окно блокирует все окна в приложении.

Окна, открытые из модального окна, не блокируются. Следует также учитывать, что метод `setWindowModality()` должен быть вызван до отображения окна.

Получить текущее значение позволяет метод `windowModality()` из класса `QWidget`. Проверить, является ли окно модальным, можно с помощью метода `isModal()` из класса `QWidget`. Метод возвращает `true`, если окно является модальным, и `false` — в противном случае. Прототипы методов:

```
Qt::WindowModality windowModality() const
bool isModal() const
```

Класс `QDialog` содержит следующие сигналы:

- `accepted()` — генерируется при установке флага `Accepted` (нажата кнопка **OK**). Сигнал не генерируется при сокрытии окна с помощью метода `hide()` или `setVisible()`;
- `rejected()` — генерируется при установке флага `Rejected` (нажата кнопка **Cancel**, кнопка **Закрыть** в заголовке окна или клавиша <Esc>). Сигнал не генерируется при сокрытии окна с помощью метода `hide()` или `setVisible()`;
- `finished(int)` — генерируется при установке кода завершения, например пользователем или с помощью методов `accept()`, `reject()` или `done()`. Внутри обработчика через параметр доступен код завершения. Сигнал не генерируется при сокрытии окна с помощью метода `hide()` или `setVisible()`.

Для всех кнопок, добавляемых в диалоговое окно, автоматически вызывается метод `setAutoDefault()`, и ему передается значение `true`. В этом случае кнопка может быть нажата с помощью клавиши <Enter>, при условии что она находится в фокусе. По умолчанию нажать кнопку позволяет только клавиша <Пробел>.

С помощью метода `setDefault()` можно указать кнопку по умолчанию. Эта кнопка может быть нажата с помощью клавиши <Enter>, когда фокус ввода установлен на другой компонент, например на текстовое поле.

10.2. Класс *QDialogButtonBox*

Класс `QDialogButtonBox` реализует контейнер, в который можно добавить различные кнопки, как пользовательские, так и стандартные. Внешний вид контейнера и расположение кнопок в нем зависят от используемой операционной системы. Иерархия наследования для класса `QDialogButtonBox`:

```
(QObject, QPaintDevice) — QWidget — QDialogButtonBox
```

Форматы конструктора класса `QDialogButtonBox`:

```
#include <QDialogButtonBox>
QDialogButtonBox(QWidget *parent = nullptr)
QDialogButtonBox(Qt::Orientation orientation, QWidget *parent = nullptr)
QDialogButtonBox(QDialogButtonBox::StandardButtons buttons,
                 QWidget *parent = nullptr)
```

```
QDialogButtonBox(QDialogButtonBox::StandardButtons buttons,
            Qt::Orientation orientation, QWidget *parent = nullptr)
```

В параметре `parent` может быть передан указатель на родительский компонент. Параметр `orientation` задает порядок расположения кнопок внутри контейнера. В качестве значения указываются константы `Qt::Horizontal` (по горизонтали; значение по умолчанию) или `Qt::Vertical` (по вертикали). В параметре `buttons` указываются следующие константы (или их комбинация через оператор `|`):

- `QDialogButtonBox::NoButton` — кнопки не установлены;
- `QDialogButtonBox::Ok` — кнопка **OK** с ролью `AcceptRole`;
- `QDialogButtonBox::Cancel` — кнопка **Cancel** с ролью `RejectRole`;
- `QDialogButtonBox::Yes` — кнопка **Yes** с ролью `YesRole`;
- `QDialogButtonBox::YesToAll` — кнопка **Yes to All** с ролью `YesRole`;
- `QDialogButtonBox::No` — кнопка **No** с ролью `NoRole`;
- `QDialogButtonBox::NoToAll` — кнопка **No to All** с ролью `NoRole`;
- `QDialogButtonBox::Open` — кнопка **Open** с ролью `AcceptRole`;
- `QDialogButtonBox::Close` — кнопка **Close** с ролью `RejectRole`;
- `QDialogButtonBox::Save` — кнопка **Save** с ролью `AcceptRole`;
- `QDialogButtonBox::SaveAll` — кнопка **Save All** с ролью `AcceptRole`;
- `QDialogButtonBox::Discard` — кнопка **Discard** или **Don't Save** (надпись на кнопке зависит от операционной системы) с ролью `DestructiveRole`;
- `QDialogButtonBox::Apply` — кнопка **Apply** с ролью `ApplyRole`;
- `QDialogButtonBox::Reset` — кнопка **Reset** с ролью `ResetRole`;
- `QDialogButtonBox::RestoreDefaults` — кнопка **Restore Defaults** с ролью `ResetRole`;
- `QDialogButtonBox::Help` — кнопка **Help** с ролью `HelpRole`;
- `QDialogButtonBox::Abort` — кнопка **Abort** с ролью `RejectRole`;
- `QDialogButtonBox::Retry` — кнопка **Retry** с ролью `AcceptRole`;
- `QDialogButtonBox::Ignore` — кнопка **Ignore** с ролью `AcceptRole`.

Поведение кнопок описывается с помощью ролей. В качестве роли можно указать следующие константы:

- `QDialogButtonBox::InvalidRole` — ошибочная роль;
- `QDialogButtonBox::AcceptRole` — нажатие кнопки устанавливает код возврата равным значению константы `Accepted`;
- `QDialogButtonBox::RejectRole` — нажатие кнопки устанавливает код возврата равным значению константы `Rejected`;
- `QDialogButtonBox::DestructiveRole` — кнопка для отказа от изменений;
- `QDialogButtonBox::ActionRole`;

- `QDialogButtonBox::HelpRole` — кнопка для отображения справки;
- `QDialogButtonBox::YesRole` — кнопка **Yes**;
- `QDialogButtonBox::NoRole` — кнопка **No**;
- `QDialogButtonBox::ResetRole` — кнопка для установки значений по умолчанию;
- `QDialogButtonBox::ApplyRole` — кнопка для принятия изменений.

Класс `QDialogButtonBox` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setOrientation()` — задает порядок расположения кнопок внутри контейнера. В качестве значения указываются константы `Qt::Horizontal` (по горизонтали) или `Qt::Vertical` (по вертикали). Прототип метода:

  ```
  void setOrientation(Qt::Orientation orientation)
  ```

- `setStandardButtons()` — устанавливает несколько стандартных кнопок. Прототип метода:

  ```
  void setStandardButtons(QDialogButtonBox::StandardButtons buttons)
  ```

  Пример:

  ```
  box->setStandardButtons(QDialogButtonBox::Ok |
                          QDialogButtonBox::Cancel);
  ```

- `addButton()` — добавляет кнопку в контейнер. Прототипы метода:

  ```
  void addButton(QAbstractButton *button,
                 QDialogButtonBox::ButtonRole role)
  QPushButton *addButton(const QString &text,
                 QDialogButtonBox::ButtonRole role)
  QPushButton *addButton(QDialogButtonBox::StandardButton button)
  ```

- `button()` — возвращает указатель на кнопку, соответствующую указанному значению, или нулевой указатель, если стандартная кнопка не была добавлена в контейнер ранее. Прототип метода:

  ```
  QPushButton *button(QDialogButtonBox::StandardButton which) const
  ```

- `buttonRole()` — возвращает роль указанной в параметре кнопки. Если кнопка не была добавлена в контейнер, то метод возвращает значение константы `InvalidRole`. Прототип метода:

  ```
  QDialogButtonBox::ButtonRole buttonRole(QAbstractButton *button) const
  ```

- `buttons()` — возвращает список с указателями на кнопки, которые были добавлены в контейнер. Прототип метода:

  ```
  QList<QAbstractButton *> buttons() const
  ```

- `removeButton()` — удаляет кнопку из контейнера, при этом не удаляя объект кнопки. Прототип метода:

  ```
  void removeButton(QAbstractButton *button)
  ```

- `clear()` — очищает контейнер и удаляет все кнопки. Прототип метода:

```
void clear()
```

- `setCenterButtons()` — если в качестве параметра указано значение `true`, то кнопки будут выравниваться по центру контейнера. Прототип метода:

```
void setCenterButtons(bool center)
```

Класс `QDialogButtonBox` содержит следующие сигналы:

- `accepted()` — генерируется при нажатии кнопки с ролью `AcceptRole` или `YesRole`. Этот сигнал можно соединить со слотом `accept()` объекта диалогового окна. Пример:

```
connect(box, SIGNAL(accepted()), this, SLOT(accept()));
```

- `rejected()` — генерируется при нажатии кнопки с ролью `RejectRole` или `NoRole`. Этот сигнал можно соединить со слотом `reject()` объекта диалогового окна. Пример:

```
connect(box, SIGNAL(rejected()), this, SLOT(reject()));
```

- `helpRequested()` — генерируется при нажатии кнопки с ролью `HelpRole`;
- `clicked(QAbstractButton *)` — генерируется при нажатии любой кнопки внутри контейнера. Внутри обработчика через параметр доступен указатель на кнопку.

10.3. Класс *QMessageBox*

Класс `QMessageBox` реализует модальные диалоговые окна для вывода сообщений. Иерархия наследования для класса `QMessageBox` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QMessageBox
```

Форматы конструктора класса `QMessageBox`:

```
#include <QMessageBox>
QMessageBox(QWidget *parent = nullptr)
QMessageBox(QMessageBox::Icon icon, const QString &title,
   const QString &text, QMessageBox::StandardButtons buttons = NoButton,
   QWidget *parent = nullptr,
   Qt::WindowFlags f = Qt::Dialog | Qt::MSWindowsFixedSizeDialogHint)
```

Если в параметре `parent` передан указатель на родительское окно, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Параметр `f` задает тип окна *(см. разд. 3.2)*. В параметре `icon` могут быть указаны следующие константы:

- `QMessageBox::NoIcon` — нет значка;
- `QMessageBox::Question` — значок со знаком вопроса;
- `QMessageBox::Information` — значок информационного сообщения;

- `QMessageBox::Warning` — значок предупреждающего сообщения;
- `QMessageBox::Critical` — значок критического сообщения.

В параметре `buttons` указываются следующие константы (или их комбинация через оператор `|`):

- `QMessageBox::NoButton` — кнопки не установлены;
- `QMessageBox::Ok` — кнопка **OK** с ролью `AcceptRole`;
- `QMessageBox::Cancel` — кнопка **Cancel** с ролью `RejectRole`;
- `QMessageBox::Yes` — кнопка **Yes** с ролью `YesRole`;
- `QMessageBox::YesToAll` — кнопка **Yes to All** с ролью `YesRole`;
- `QMessageBox::No` — кнопка **No** с ролью `NoRole`;
- `QMessageBox::NoToAll` — кнопка **No to All** с ролью `NoRole`;
- `QMessageBox::Open` — кнопка **Open** с ролью `AcceptRole`;
- `QMessageBox::Close` — кнопка **Close** с ролью `RejectRole`;
- `QMessageBox::Save` — кнопка **Save** с ролью `AcceptRole`;
- `QMessageBox::SaveAll` — кнопка **Save All** с ролью `AcceptRole`;
- `QMessageBox::Discard` — кнопка **Discard** или **Don't Save** (надпись на кнопке зависит от операционной системы) с ролью `DestructiveRole`;
- `QMessageBox::Apply` — кнопка **Apply** с ролью `ApplyRole`;
- `QMessageBox::Reset` — кнопка **Reset** с ролью `ResetRole`;
- `QMessageBox::RestoreDefaults` — кнопка **Restore Defaults** с ролью `ResetRole`;
- `QMessageBox::Help` — кнопка **Help** с ролью `HelpRole`;
- `QMessageBox::Abort` — кнопка **Abort** с ролью `RejectRole`;
- `QMessageBox::Retry` — кнопка **Retry** с ролью `AcceptRole`;
- `QMessageBox::Ignore` — кнопка **Ignore** с ролью `AcceptRole`.

Поведение кнопок описывается с помощью ролей. В качестве роли можно указать следующие константы:

- `QMessageBox::InvalidRole` — ошибочная роль;
- `QMessageBox::AcceptRole` — нажатие кнопки устанавливает код возврата равным значению константы `Accepted`;
- `QMessageBox::RejectRole` — нажатие кнопки устанавливает код возврата равным значению константы `Rejected`;
- `QMessageBox::DestructiveRole` — кнопка для отказа от изменений;
- `QMessageBox::ActionRole`;
- `QMessageBox::HelpRole` — кнопка для отображения справки;
- `QMessageBox::YesRole` — кнопка **Yes**;

- `QMessageBox::NoRole` — кнопка **No**;
- `QMessageBox::ResetRole` — кнопка для установки значений по умолчанию;
- `QMessageBox::ApplyRole` — кнопка для принятия изменений.

После создания экземпляра класса следует вызвать метод `exec()` для отображения окна. Метод возвращает код нажатой кнопки. Пример:

```
QMessageBox dialog(QMessageBox::Critical,
                   "Текст заголовка", "Текст сообщения",
                   QMessageBox::Ok | QMessageBox::Cancel,
                   this);
int result = dialog.exec();
```

10.3.1. Основные методы и сигналы

Класс `QMessageBox` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setIcon()` — устанавливает стандартный значок. Прототип метода:

  ```
  void setIcon(QMessageBox::Icon)
  ```

- `setIconPixmap()` — устанавливает пользовательский значок. Прототип метода:

  ```
  void setIconPixmap(const QPixmap &pixmap)
  ```

- `setWindowTitle()` — задает текст заголовка окна. Прототип метода:

  ```
  void setWindowTitle(const QString &title)
  ```

- `setText()` — задает текст сообщения. Можно указать как обычный текст, так и текст в формате HTML. Перенос строки в обычной строке осуществляется с помощью символа `\n`, а в строке в формате HTML — с помощью тега `<br>`. Прототип метода:

  ```
  void setText(const QString &text)
  ```

- `setInformativeText()` — задает дополнительный текст сообщения, который отображается под обычным текстом сообщения. Можно указать как обычный текст, так и текст в формате HTML. Прототип метода:

  ```
  void setInformativeText(const QString &text)
  ```

- `setDetailedText()` — задает текст описания деталей сообщения. Если текст задан, то будет добавлена кнопка **Show Details**, с помощью которой можно отобразить скрытую панель с описанием. Прототип метода:

  ```
  void setDetailedText(const QString &text)
  ```

- `setTextFormat()` — задает режим отображения текста сообщения. Прототип метода:

  ```
  void setTextFormat(Qt::TextFormat format)
  ```

Могут быть указаны следующие константы:

- `Qt::PlainText` — простой текст;
- `Qt::RichText` — форматированный текст;
- `Qt::AutoText` — автоматическое определение (режим по умолчанию). Если текст содержит теги, то используется режим `RichText`, в противном случае — режим `PlainText`;
- `Qt::MarkdownText`;

❐ `setStandardButtons()` — устанавливает несколько стандартных кнопок. Прототип метода:

```
void setStandardButtons(QMessageBox::StandardButtons buttons)
```

❐ `addButton()` — добавляет кнопку в окно. Прототипы методов:

```
void addButton(QAbstractButton *button,
               QMessageBox::ButtonRole role)
QPushButton *addButton(const QString &text,
               QMessageBox::ButtonRole role)
QPushButton *addButton(QMessageBox::StandardButton button)
```

❐ `setDefaultButton()` — задает кнопку по умолчанию. Прототипы метода:

```
void setDefaultButton(QPushButton *button)
void setDefaultButton(QMessageBox::StandardButton button)
```

❐ `setEscapeButton()` — задает кнопку, которая будет нажата при нажатии клавиши <Esc>. Прототипы метода:

```
void setEscapeButton(QAbstractButton *button)
void setEscapeButton(QMessageBox::StandardButton button)
```

❐ `clickedButton()` — возвращает указатель на кнопку, которая была нажата, или нулевой указатель. Прототип метода:

```
QAbstractButton *clickedButton() const
```

❐ `button()` — возвращает указатель на кнопку, соответствующую указанному значению, или нулевой указатель, если стандартная кнопка не была добавлена в окно ранее. Прототип метода:

```
QAbstractButton *button(QMessageBox::StandardButton which) const
```

❐ `buttonRole()` — возвращает роль указанной в параметре кнопки. Если кнопка не была добавлена в окно, то метод возвращает значение константы `InvalidRole`. Прототип метода:

```
QMessageBox::ButtonRole buttonRole(QAbstractButton *button) const
```

❐ `buttons()` — возвращает список с указателями на кнопки, которые были добавлены в окно. Прототип метода:

```
QList<QAbstractButton *> buttons() const
```

- `removeButton()` — удаляет кнопку из окна, при этом не удаляя объект кнопки. Прототип метода:

```
void removeButton(QAbstractButton *button)
```

Класс `QMessageBox` содержит сигнал `buttonClicked(QAbstractButton *)`, который генерируется при нажатии кнопки в окне. Внутри обработчика через параметр доступен указатель на кнопку.

10.3.2. Окно для вывода обычного сообщения

Помимо рассмотренных методов класс `QMessageBox` содержит несколько статических методов, реализующих готовые диалоговые окна. Для вывода модального окна с информационным сообщением предназначен статический метод `information()`. Прототип метода:

```
static QMessageBox::StandardButton information(QWidget *parent,
   const QString &title, const QString &text,
   QMessageBox::StandardButtons buttons = Ok,
   QMessageBox::StandardButton defaultButton = NoButton)
```

В параметре `parent` передается указатель на родительское окно или нулевой указатель. Если передан указатель, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать отображаемые стандартные кнопки (константы, задающие стандартные кнопки, указываются через оператор `|`). По умолчанию отображается кнопка **OK**. Параметр `defaultButton` назначает кнопку по умолчанию. Метод `information()` возвращает код нажатой кнопки. Пример:

```
QMessageBox::information(this, "Текст заголовка",
                         "Текст сообщения",
                         QMessageBox::Close,
                         QMessageBox::Close);
```

Результат выполнения этого кода показан на рис. 10.1.

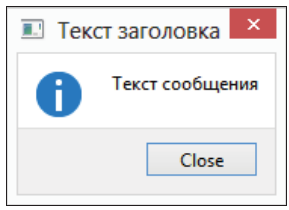

Рис. 10.1. Окно для вывода обычного сообщения

10.3.3. Окно запроса подтверждения

Для вывода модального окна с запросом подтверждения каких-либо действий предназначен статический метод `question()`. Прототип метода:

```
static QMessageBox::StandardButton question(QWidget *parent,
   const QString &title, const QString &text,
   QMessageBox::StandardButtons buttons = StandardButtons(Yes | No),
   QMessageBox::StandardButton defaultButton = NoButton)
```

В параметре `parent` передается указатель на родительское окно или нулевой указатель. Если передан указатель, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать отображаемые стандартные кнопки (константы, задающие стандартные кнопки, указываются через оператор `|`). Параметр `defaultButton` назначает кнопку по умолчанию. Метод `question()` возвращает код нажатой кнопки. Пример:

```
QMessageBox::StandardButton result = QMessageBox::question(
              this, "Текст заголовка",
              "Вы действительно хотите выполнить действие?",
              QMessageBox::Yes | QMessageBox::No |
              QMessageBox::Cancel,
              QMessageBox::Cancel);
```

Результат выполнения этого кода показан на рис. 10.2.

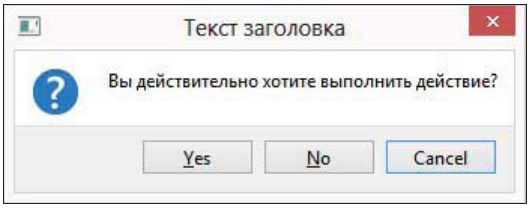

Рис. 10.2. Окно запроса подтверждения

10.3.4. Окно для вывода предупреждающего сообщения

Для вывода модального окна с предупреждающим сообщением предназначен статический метод `warning()`. Прототип метода:

```
static QMessageBox::StandardButton warning(QWidget *parent,
   const QString &title, const QString &text,
   QMessageBox::StandardButtons buttons = Ok,
   QMessageBox::StandardButton defaultButton = NoButton)
```

В параметре `parent` передается указатель на родительское окно или нулевой указатель. Если передан указатель, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать отображаемые стандартные кнопки (константы, задающие стандартные кнопки, указываются через оператор `|`). По умолчанию отображается кнопка **OK**. Параметр `defaultButton` назначает кнопку по умолчанию. Метод `warning()` возвращает код нажатой кнопки. Пример:

```
QMessageBox::StandardButton result = QMessageBox::warning(
              this, "Текст заголовка",
              "Действие может быть опасным. Продолжить?",
              QMessageBox::Yes | QMessageBox::No |
              QMessageBox::Cancel,
              QMessageBox::Cancel);
```

Результат выполнения этого кода показан на рис. 10.3.

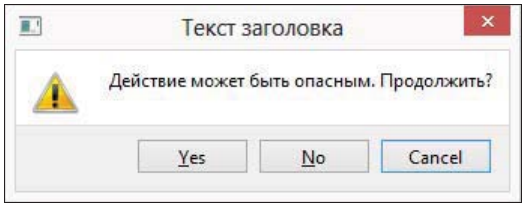

Рис. 10.3. Окно для вывода предупреждающего сообщения

10.3.5. Окно для вывода критического сообщения

Для вывода модального окна с критическим сообщением предназначен статический метод `critical()`. Прототип метода:

```
static QMessageBox::StandardButton critical(QWidget *parent,
   const QString &title, const QString &text,
   QMessageBox::StandardButtons buttons = Ok,
   QMessageBox::StandardButton defaultButton = NoButton)
```

В параметре `parent` передается указатель на родительское окно или нулевой указатель. Если передан указатель, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `buttons` позволяет указать отображаемые стандартные кнопки (константы, задающие стандартные кнопки, указываются через оператор `|`). По умолчанию отображается кнопка **OK**. Параметр `defaultButton` назначает кнопку по умолчанию. Метод `critical()` возвращает код нажатой кнопки. Пример:

```
QMessageBox::critical(this, "Текст заголовка",
   "Программа выполнила недопустимую ошибку и будет закрыта",
   QMessageBox::Ok, QMessageBox::Ok);
```

Результат выполнения этого кода показан на рис. 10.4.

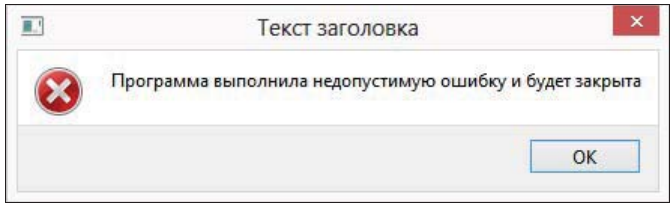

Рис. 10.4. Окно для вывода критического сообщения

10.3.6. Окно «О программе»

Для вывода модального окна с описанием программы и информацией об авторских правах предназначен статический метод `about()`. Прототип метода:

```
static void about(QWidget *parent, const QString &title, const QString &text)
```

В параметре `parent` передается указатель на родительское окно или нулевой указатель. Если передан указатель, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Слева от текста сообщения отображается значок приложения, если он был установлен. Пример:

```
QMessageBox::about(this, "Текст заголовка", "Описание программы");
```

Результат выполнения этого кода показан на рис. 10.5.

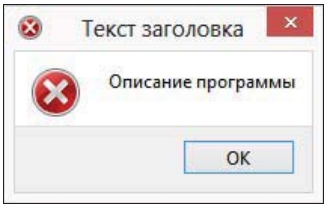

Рис. 10.5. Окно «О программе»

10.3.7. Окно «About Qt»

Для вывода модального окна с описанием используемой версии библиотеки Qt предназначен статический метод `aboutQt()`. Прототип метода:

```
static void aboutQt(QWidget *parent, const QString &title = QString())
```

В параметре `parent` передается указатель на родительское окно или нулевой указатель. Если передан указатель, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. В параметре `title` можно указать текст, выводимый в заголовке окна. Если параметр не указан, то выводится заголовок «About Qt». Пример:

```
QMessageBox::aboutQt(this);
```

10.4. Класс *QInputDialog*

Класс `QInputDialog` реализует модальные диалоговые окна для ввода различных данных. Иерархия наследования для класса `QInputDialog` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QInputDialog
```

Формат конструктора класса `QInputDialog`:

```
#include <QInputDialog>
QInputDialog(QWidget *parent = nullptr,
             Qt::WindowFlags flags = Qt::WindowFlags())
```

Если в параметре `parent` передан указатель на родительское окно, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Параметр `flags` задает тип окна *(см. разд. 3.2)*.

После создания экземпляра класса следует вызвать метод `exec()` для отображения окна. Метод возвращает код возврата в виде значения следующих констант: `Accepted` или `Rejected`. Пример отображения диалогового окна и обработки статуса внутри обработчика нажатия кнопки:

```
void Widget::on_btn1_clicked()
{
   QInputDialog dialog(this);
   int result = dialog.exec();
   if (result == QDialog::Accepted) {
       qDebug() << "Нажата кнопка OK";
       // Здесь получаем данные из диалогового окна
   }
   else {
      qDebug() << "Нажата кнопка Cancel";
   }
}
```

10.4.1. Основные методы и сигналы

Класс `QInputDialog` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setLabelText()` — задает текст, отображаемый над полем ввода. Прототип метода:

  ```
  void setLabelText(const QString &text)
  ```

- `setOkButtonText()` — задает текст, отображаемый на кнопке **OK**. Прототип метода:

  ```
  void setOkButtonText(const QString &text)
  ```

- `setCancelButtonText()` — задает текст, отображаемый на кнопке **Cancel**. Прототип метода:

  ```
  void setCancelButtonText(const QString &text)
  ```

- `setInputMode()` — задает режим ввода данных. Прототип метода:

  ```
  void setInputMode(QInputDialog::InputMode mode)
  ```

  В качестве параметра указываются следующие константы:

  - `QInputDialog::TextInput` — ввод текста;
  - `QInputDialog::IntInput` — ввод целого числа;
  - `QInputDialog::DoubleInput` — ввод вещественного числа;

- `setTextEchoMode()` — задает режим отображения текста в поле. Прототип метода:

  ```
  void setTextEchoMode(QLineEdit::EchoMode mode)
  ```

  Могут быть указаны следующие константы:

  - `QLineEdit::Normal` — показывать символы, как они были введены;
  - `QLineEdit::NoEcho` — не показывать вводимые символы;
  - `QLineEdit::Password` — вместо символов указывать символ `*`;
  - `QLineEdit::PasswordEchoOnEdit` — показывать символы при вводе, а при потере фокуса отображать символ `*`;

- `setTextValue()` — задает текст по умолчанию, отображаемый в текстовом поле. Прототип метода:

  ```
  void setTextValue(const QString &text)
  ```

- `textValue()` — возвращает текст, введенный в текстовое поле. Прототип метода:

  ```
  QString textValue() const
  ```

- `setIntValue()` — задает целочисленное значение при использовании режима `IntInput`. Прототип метода:

  ```
  void setIntValue(int value)
  ```

- `intValue()` — возвращает целочисленное значение, введенное в поле, при использовании режима `IntInput`. Прототип метода:

  ```
  int intValue() const
  ```

- `setIntRange()`, `setIntMinimum()` и `setIntMaximum()` — задают диапазон допустимых целочисленных значений при использовании режима `IntInput`. Прототипы методов:

  ```
  void setIntRange(int min, int max)
  void setIntMinimum(int min)
  void setIntMaximum(int max)
  ```

- `setIntStep()` — задает шаг приращения значения при нажатии кнопок со стрелками в правой части поля при использовании режима `IntInput`. Прототип метода:

  ```
  void setIntStep(int step)
  ```

- `setDoubleValue()` — задает вещественное значение при использовании режима `DoubleInput`. Прототип метода:

  ```
  void setDoubleValue(double value)
  ```

- `doubleValue()` — возвращает вещественное значение, введенное в поле, при использовании режима `DoubleInput`. Прототип метода:

  ```
  double doubleValue() const
  ```

- `setDoubleRange()`, `setDoubleMinimum()` и `setDoubleMaximum()` — задают диапазон допустимых вещественных значений при использовании режима `DoubleInput`. Прототипы методов:

```
void setDoubleRange(double min, double max)
void setDoubleMinimum(double min)
void setDoubleMaximum(double max)
```

- `setDoubleStep()` — задает шаг приращения значения при нажатии кнопок со стрелками в правой части поля при использовании режима `DoubleInput`. Прототип метода:

```
void setDoubleStep(double step)
```

- `setDoubleDecimals()` — задает количество цифр после десятичной точки при использовании режима `DoubleInput`. Прототип метода:

```
void setDoubleDecimals(int decimals)
```

- `setComboBoxItems()` — задает строки, которые будут отображаться в раскрывающемся списке. Прототип метода:

```
void setComboBoxItems(const QStringList &items)
```

- `setComboBoxEditable()` — если в качестве параметра указано значение `true`, то значения из раскрывающегося списка можно будет редактировать. Прототип метода:

```
void setComboBoxEditable(bool editable)
```

- `setOption()` — если во втором параметре указано значение `true`, то производит установку указанной в первом параметре опции, а если `false`, то сбрасывает опцию. Прототип метода:

```
void setOption(QInputDialog::InputDialogOption option, bool on = true)
```

  В первом параметре можно указать следующие константы:

  - `QInputDialog::NoButtons` — кнопки не отображаются;
  - `QInputDialog::UseListViewForComboBoxItems` — для отображения списка строк будет использоваться класс `QListView`, а не `QComboBox`;
  - `QInputDialog::UsePlainTextEditForTextInput` — для ввода строки использовать компонент `QPlainTextEdit`;

- `setOptions()` — устанавливает несколько опций (см. описание метода `setOption()`) сразу.

```
void setOptions(QInputDialog::InputDialogOptions options)
```

Класс `QInputDialog` содержит следующие сигналы:

- `textValueChanged(const QString&)` — генерируется при изменении значения в текстовом поле. Внутри обработчика через параметр доступно новое значение. Сигнал генерируется при использовании режима `TextInput`;

- `textValueSelected(const QString&)` — генерируется при нажатии кнопки **OK**. Внутри обработчика через параметр доступно введенное значение. Сигнал генерируется при использовании режима `TextInput`;
- `intValueChanged(int)` — генерируется при изменении значения в поле. Внутри обработчика через параметр доступно новое значение. Сигнал генерируется при использовании режима `IntInput`;
- `intValueSelected(int)` — генерируется при нажатии кнопки **OK**. Внутри обработчика через параметр доступно введенное значение. Сигнал генерируется при использовании режима `IntInput`;
- `doubleValueChanged(double)` — генерируется при изменении значения в поле. Внутри обработчика через параметр доступно новое значение. Сигнал генерируется при использовании режима `DoubleInput`;
- `doubleValueSelected(double)` — генерируется при нажатии кнопки **OK**. Внутри обработчика через параметр доступно введенное значение. Сигнал генерируется при использовании режима `DoubleInput`.

10.4.2. Окна для ввода строки

Помимо рассмотренных методов класс `QInputDialog` содержит несколько статических методов, реализующих готовые диалоговые окна. Окно для ввода обычной строки или пароля реализуется с помощью статического метода `getText()`. Прототип метода:

```
static QString getText(QWidget *parent,
   const QString &title, const QString &label,
   QLineEdit::EchoMode mode = QLineEdit::Normal,
   const QString &text = QString(), bool *ok = nullptr,
   Qt::WindowFlags flags = Qt::WindowFlags(),
   Qt::InputMethodHints inputMethodHints = Qt::ImhNone)
```

В параметре `parent` передается указатель на родительское окно или нулевой указатель. Если передан указатель, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `mode` задает режим отображения текста в поле (см. описание метода `setTextEchoMode()` в *разд. 10.4.1*). Параметр `text` устанавливает значение поля по умолчанию, а в параметре `flags` можно указать тип окна. Статус операции доступен через параметр `ok` (переменная будет иметь значение `true`, если была нажата кнопка **OK**, или значение `false`, если была нажата кнопка **Cancel**, клавиша <Esc> или кнопка **Закрыть** в заголовке окна). Метод `getText()` возвращает значение, введенное в поле. Пример:

```
bool ok;
QString text = QInputDialog::getText(this, "Это заголовок окна",
                 "Это текст подсказки", QLineEdit::Normal,
                 "Значение по умолчанию", &ok);
```

```
if (ok) {
   qDebug() << "Введено значение:" << text;
}
```

Результат выполнения этого кода показан на рис. 10.6.

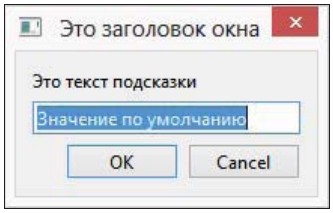

Рис. 10.6. Окно для ввода строки

Окно для ввода многострочного текста реализуется с помощью статического метода `getMultiLineText()`. Прототип метода:

```
static QString getMultiLineText(QWidget *parent,
   const QString &title, const QString &label,
   const QString &text = QString(), bool *ok = nullptr,
   Qt::WindowFlags flags = Qt::WindowFlags(),
   Qt::InputMethodHints inputMethodHints = Qt::ImhNone)
```

Пример:

```
bool ok;
QString text = QInputDialog::getMultiLineText(
                  this, "Это заголовок окна",
                  "Это текст подсказки",
                  "Значение по умолчанию", &ok);
if (ok) {
   qDebug() << "Введено значение:" << text;
}
```

10.4.3. Окно для ввода целого числа

Окно для ввода целого числа реализуется с помощью статического метода `getInt()`. Прототип метода:

```
static int getInt(QWidget *parent,
   const QString &title, const QString &label,
   int value = 0, int min = -2147483647, int max = 2147483647,
   int step = 1, bool *ok = nullptr,
   Qt::WindowFlags flags = Qt::WindowFlags())
```

В параметре `parent` передается указатель на родительское окно или нулевой указатель. Если передан указатель, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `value` устанавливает значение поля по умолчанию. Параметр `min` задает минималь-

ное значение, параметр `max` — максимальное значение, а параметр `step` — шаг приращения. Параметр `flags` позволяет указать тип окна. Статус операции доступен через параметр `ok` (переменная будет иметь значение `true`, если была нажата кнопка **OK**, или значение `false`, если была нажата кнопка **Cancel**, клавиша <Esc> или кнопка **Закрыть** в заголовке окна). Метод возвращает значение, введенное в поле. Пример:

```
bool ok;
int n = QInputDialog::getInt(this, "Это заголовок окна",
                      "Это текст подсказки",
                      50, 0, 100, 2, &ok);
if (ok) {
   qDebug() << "Введено значение:" << n;
}
```

Результат выполнения этого кода показан на рис. 10.7.

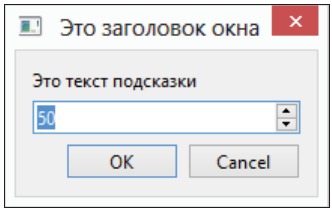

Рис. 10.7. Окно для ввода целого числа

10.4.4. Окно для ввода вещественного числа

Окно для ввода вещественного числа реализуется с помощью статического метода `getDouble()`. Прототип метода:

```
static double getDouble(QWidget *parent,
   const QString &title, const QString &label,
   double value = 0, double min = -2147483647, double max = 2147483647,
   int decimals = 1, bool *ok = nullptr,
   Qt::WindowFlags flags = Qt::WindowFlags(), double step = 1)
```

В параметре `parent` передается указатель на родительское окно или нулевой указатель. Если передан указатель, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `value` устанавливает значение поля по умолчанию. Параметр `min` задает минимальное значение, параметр `max` — максимальное значение, а параметр `decimals` — количество цифр после десятичной точки. Параметр `flags` позволяет указать тип окна. Статус операции доступен через параметр `ok` (переменная будет иметь значение `true`, если была нажата кнопка **OK**, или значение `false`, если была нажата кнопка **Cancel**, клавиша <Esc> или кнопка **Закрыть** в заголовке окна). Метод возвращает значение, введенное в поле. Пример:

```
bool ok;
double n = QInputDialog::getDouble(this, "Это заголовок окна",
                      "Это текст подсказки",
                      50.0, 0.0, 100.0, 2, &ok);
if (ok) {
   qDebug() << "Введено значение:" << n;
}
```

10.4.5. Окно для выбора пункта из списка

Окно для выбора пункта из списка реализуется с помощью статического метода `getItem()`. Прототип метода:

```
static QString getItem(QWidget *parent,
   const QString &title, const QString &label,
   const QStringList &items, int current = 0,
   bool editable = true, bool *ok = nullptr,
   Qt::WindowFlags flags = Qt::WindowFlags(),
   Qt::InputMethodHints inputMethodHints = Qt::ImhNone)
```

В параметре `parent` передается указатель на родительское окно или нулевой указатель. Если передан указатель, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Необязательный параметр `current` устанавливает индекс пункта, выбранного по умолчанию. Если в параметре `editable` указано значение `true`, то текст пункта списка можно редактировать. Параметр `flags` позволяет указать тип окна. Статус операции доступен через параметр `ok` (переменная будет иметь значение `true`, если была нажата кнопка **OK**, или значение `false`, если была нажата кнопка **Cancel**, клавиша <Esc> или кнопка **Закрыть** в заголовке окна). Метод возвращает текст выбранного пункта в списке. Пример:

```
bool ok;
QStringList list;
list << "Пункт 1" << "Пункт 2" << "Пункт 3";
QString text = QInputDialog::getItem(this,
                "Это заголовок окна", "Это текст подсказки",
                list, 1, false, &ok);
if (ok) {
   qDebug() << "Выбрано значение:" << text;
}
```

Результат выполнения этого кода показан на рис. 10.8.

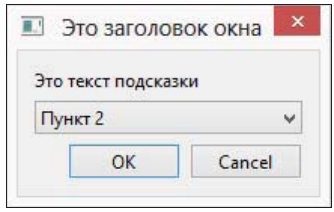

Рис. 10.8. Окно для выбора пункта из списка

10.5. Класс *QFileDialog*

Класс `QFileDialog` реализует модальные диалоговые окна для выбора файла или каталога. Иерархия наследования для класса `QFileDialog` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QFileDialog
```

Форматы конструктора класса `QFileDialog`:

```
#include <QFileDialog>
QFileDialog(QWidget *parent, Qt::WindowFlags flags)
QFileDialog(QWidget *parent = nullptr,
   const QString &caption = QString(),
   const QString &directory = QString(),
   const QString &filter = QString())
```

Если в параметре `parent` передан указатель на родительское окно, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Параметр `flags` задает тип окна *(см. разд. 3.2)*. Необязательный параметр `caption` позволяет указать заголовок окна, параметр `directory` — начальный каталог, а параметр `filter` — ограничивает отображение файлов указанным фильтром.

После создания экземпляра класса следует вызвать метод `exec()` для отображения окна. Метод возвращает код возврата в виде значения следующих констант: `Accepted` или `Rejected`.

10.5.1. Основные методы и сигналы

Класс `QFileDialog` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setAcceptMode()` — задает тип окна. Прототип метода:

  ```
  void setAcceptMode(QFileDialog::AcceptMode mode)
  ```

  В качестве параметра указываются следующие константы:

  - `QFileDialog::AcceptOpen` — окно для открытия файла (по умолчанию);
  - `QFileDialog::AcceptSave` — окно для сохранения файла;

- `setViewMode()` — задает режим отображения файлов. Прототип метода:

  ```
  void setViewMode(QFileDialog::ViewMode mode)
  ```

  В качестве параметра указываются следующие константы:

  - `QFileDialog::Detail` — отображается детальная информация о файлах;
  - `QFileDialog::List` — отображается только значок и название файла;

- `setFileMode()` — задает тип возвращаемого значения. Прототип метода:

  ```
  void setFileMode(QFileDialog::FileMode mode)
  ```

В качестве параметра указываются следующие константы:

- `QFileDialog::AnyFile` — имя файла независимо от того, существует он или нет;
- `QFileDialog::ExistingFile` — имя существующего файла;
- `QFileDialog::Directory` — имя каталога;
- `QFileDialog::ExistingFiles` — список из нескольких существующих файлов. Несколько файлов можно выбрать, удерживая нажатой клавишу <Ctrl>;

❐ `setOption()` — если во втором параметре указано значение `true`, то производит установку указанной в первом параметре опции, а если `false`, то сбрасывает опцию. Прототип метода:

```
void setOption(QFileDialog::Option option, bool on = true)
```

В первом параметре можно указать следующие константы:

- `QFileDialog::ShowDirsOnly` — отображать только названия каталогов. Опция работает только при использовании типа возвращаемого значения `Directory`;
- `QFileDialog::DontResolveSymlinks`;
- `QFileDialog::DontConfirmOverwrite` — не спрашивать разрешения, если выбран существующий файл;
- `QFileDialog::DontUseNativeDialog`;
- `QFileDialog::ReadOnly` — режим только для чтения;
- `QFileDialog::HideNameFilterDetails` — скрывает детали фильтра;
- `QFileDialog::DontUseCustomDirectoryIcons`;

❐ `setOptions()` — позволяет установить сразу несколько опций сразу. Прототип метода:

```
void setOptions(QFileDialog::Options options)
```

❐ `setDirectory()` — задает отображаемый каталог. Прототипы метода:

```
void setDirectory(const QString &directory)
void setDirectory(const QDir &directory)
```

❐ `directory()` — возвращает экземпляр класса `QDir` с путем к отображаемому каталогу. Прототип метода:

```
QDir directory() const
```

❐ `setNameFilter()` — устанавливает фильтр. Чтобы установить несколько фильтров, необходимо перечислить их через две «точки с запятой». Прототип метода:

```
void setNameFilter(const QString &filter)
```

Пример:

```
dialog.setNameFilter("All (*);;Images (*.png *.jpg)");
```

- `setNameFilters()` — устанавливает несколько фильтров. Прототип метода:

  ```
  void setNameFilters(const QStringList &filters)
  ```

  Пример:

  ```
  QStringList list;
  list << "All (*)" << "Images (*.png *.jpg)";
  dialog.setNameFilters(list);
  ```

- `selectFile()` — выбирает указанный файл. Прототип метода:

  ```
  void selectFile(const QString &filename)
  ```

- `selectedFiles()` — возвращает список с выбранными файлами. Прототип метода:

  ```
  QStringList selectedFiles() const
  ```

- `setDefaultSuffix()` — задает расширение, которое добавляется к файлу при отсутствии указанного расширения. Прототип метода:

  ```
  void setDefaultSuffix(const QString &suffix)
  ```

- `setHistory()` — задает список истории. Прототип метода:

  ```
  void setHistory(const QStringList &paths)
  ```

- `setSidebarUrls()` — задает список URL-адресов, отображаемый на боковой панели. Прототип метода:

  ```
  void setSidebarUrls(const QList<QUrl> &urls)
  ```

- `setLabelText()` — позволяет изменить текст указанной надписи. Прототип метода:

  ```
  void setLabelText(QFileDialog::DialogLabel label, const QString &text)
  ```

  В первом параметре указываются следующие константы:

  - `QFileDialog::LookIn` — надпись слева от списка с каталогами;
  - `QFileDialog::FileName` — надпись слева от поля с названием файла;
  - `QFileDialog::FileType` — надпись слева от поля с типами файлов;
  - `QFileDialog::Accept` — надпись на кнопке, нажатие которой приведет к принятию диалога (по умолчанию — `Open` или `Save`);
  - `QFileDialog::Reject` — надпись на кнопке, нажатие которой приведет к отмене диалога (по умолчанию — `Cancel`);

- `saveState()` – возвращает экземпляр класса `QByteArray` с текущими настройками. Прототип метода:

  ```
  QByteArray saveState() const
  ```

- `restoreState()` — восстанавливает настройки и возвращает статус успешности выполненной операции. Прототип метода:

  ```
  bool restoreState(const QByteArray &state)
  ```

Класс `QFileDialog` содержит следующие основные сигналы (полный список смотрите в документации):

- `currentChanged(const QString&)` — генерируется при изменении текущего файла. Внутри обработчика через параметр доступен новый путь;
- `directoryEntered(const QString&)` — генерируется при изменении каталога. Внутри обработчика через параметр доступен новый путь;
- `filterSelected(const QString&)` — генерируется при изменении фильтра. Внутри обработчика через параметр доступен новый фильтр;
- `fileSelected(const QString&)` — генерируется при выборе файла и принятии диалога. Внутри обработчика через параметр доступен путь;
- `filesSelected(const QStringList&)` — генерируется при выборе нескольких файлов и принятии диалога. Внутри обработчика через параметр доступен список с путями.

10.5.2. Окно для выбора каталога

Помимо рассмотренных методов класс `QFileDialog` содержит несколько статических методов, реализующих готовые диалоговые окна. Окно для выбора каталога реализуется с помощью статического метода `getExistingDirectory()`. Прототип метода:

```
static QString getExistingDirectory(QWidget *parent = nullptr,
   const QString &caption = QString(), const QString &dir = QString(),
   QFileDialog::Options options = ShowDirsOnly)
```

В параметре `parent` передается указатель на родительское окно или нулевой указатель. Необязательный параметр `caption` задает текст заголовка окна, параметр `dir` — текущий каталог, а параметр `options` устанавливает опции (см. описание метода `setOption()` в *разд. 10.5.1*). Метод возвращает выбранный каталог или пустую строку. Пример:

```
QString dir = QFileDialog::getExistingDirectory(this,
                            "Заголовок окна",
                            QDir::currentPath());
qDebug() << dir;
```

Результат выполнения этого кода показан на рис. 10.9.

Можно также воспользоваться статическим методом `getExistingDirectoryUrl()`:

```
static QUrl getExistingDirectoryUrl(QWidget *parent = nullptr,
   const QString &caption = QString(), const QUrl &dir = QUrl(),
   QFileDialog::Options options = ShowDirsOnly,
   const QStringList &supportedSchemes = QStringList())
```

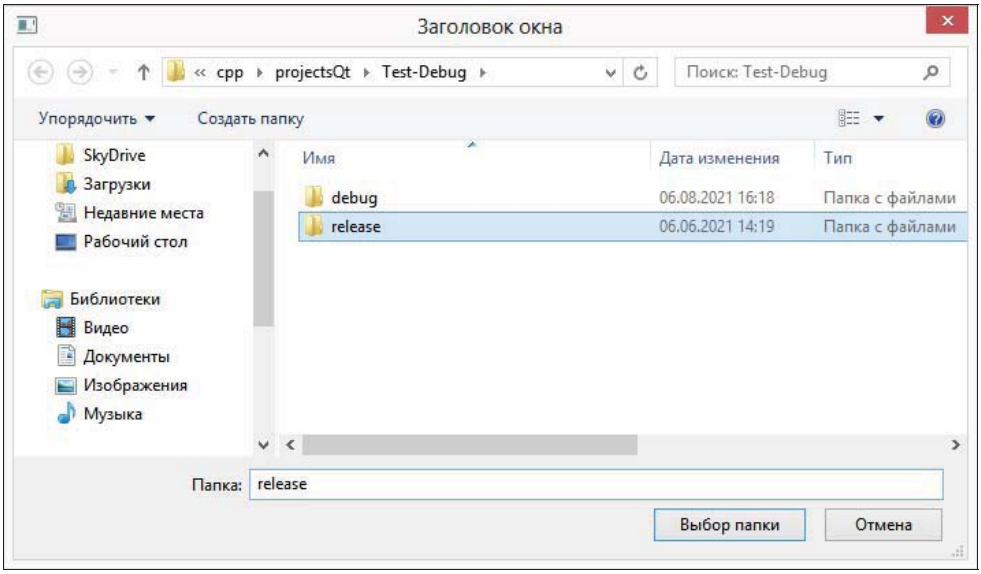

Рис. 10.9. Окно для выбора каталога

10.5.3. Окно для открытия файла

Окно для открытия одного файла реализуется с помощью статического метода `getOpenFileName()`. Прототип метода:

```
static QString getOpenFileName(QWidget *parent = nullptr,
   const QString &caption = QString(), const QString &dir = QString(),
   const QString &filter = QString(), QString *selectedFilter = nullptr,
   QFileDialog::Options options = Options())
```

В параметре `parent` передается указатель на родительское окно или нулевой указатель. Необязательный параметр `caption` задает текст заголовка окна, параметр `dir` — текущий каталог, параметр `filter` — фильтр, а параметр `options` устанавливает опции (см. описание метода `setOption()` в *разд. 10.5.1*). Метод возвращает выбранный файл или пустую строку. Пример:

```
QString f = QFileDialog::getOpenFileName(this,
                "Заголовок окна", "C:\\cpp\\projectsQt\\Test",
                "All (*);;Images (*.png *.jpg)");
qDebug() << f;
```

Результат выполнения этого кода показан на рис. 10.10.

Можно также воспользоваться статическим методом `getOpenFileUrl()`:

```
static QUrl getOpenFileUrl(QWidget *parent = nullptr,
   const QString &caption = QString(), const QUrl &dir = QUrl(),
   const QString &filter = QString(), QString *selectedFilter = nullptr,
   QFileDialog::Options options = Options(),
   const QStringList &supportedSchemes = QStringList())
```

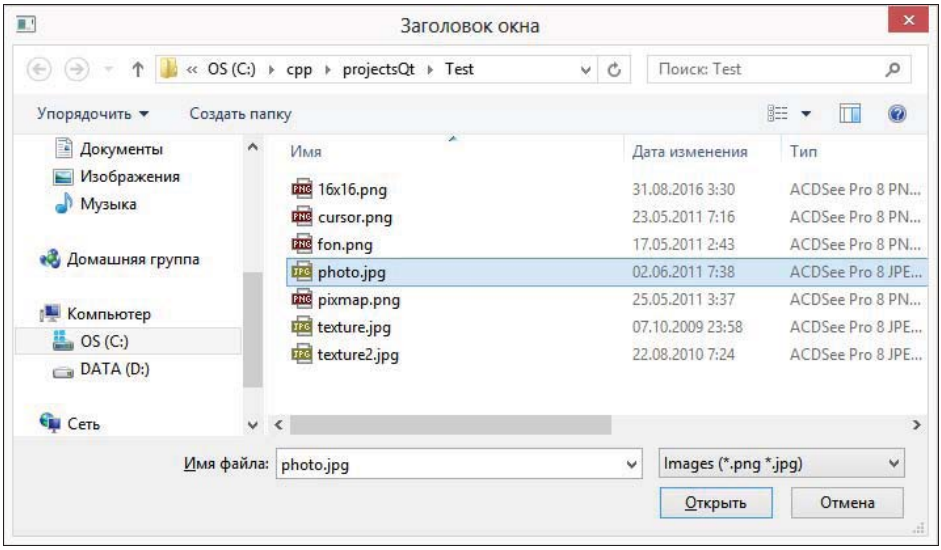

Рис. 10.10. Окно для открытия файла

Окно для открытия нескольких файлов реализуется с помощью статического метода `getOpenFileNames()`. Прототип метода:

```
static QStringList getOpenFileNames(QWidget *parent = nullptr,
   const QString &caption = QString(), const QString &dir = QString(),
   const QString &filter = QString(), QString *selectedFilter = nullptr,
   QFileDialog::Options options = Options())
```

Метод возвращает список с выбранными файлами или пустой список. Пример:

```
QStringList list = QFileDialog::getOpenFileNames(this,
                "Заголовок окна", "C:\\cpp\\projectsQt\\Test",
                "All (*);;Images (*.png *.jpg)");
qDebug() << list;
```

Можно также воспользоваться статическим методом `getOpenFileUrls()`:

```
static QList<QUrl> getOpenFileUrls(QWidget *parent = nullptr,
   const QString &caption = QString(), const QUrl &dir = QUrl(),
   const QString &filter = QString(), QString *selectedFilter = nullptr,
   QFileDialog::Options options = Options(),
   const QStringList &supportedSchemes = QStringList())
```

10.5.4. Окно для сохранения файла

Окно для сохранения файла реализуется с помощью статического метода `getSaveFileName()`. Прототип метода:

```
static QString getSaveFileName(QWidget *parent = nullptr,
   const QString &caption = QString(), const QString &dir = QString(),
   const QString &filter = QString(), QString *selectedFilter = nullptr,
   QFileDialog::Options options = Options())
```

В параметре `parent` передается указатель на родительское окно или нулевой указатель. Необязательный параметр `caption` задает текст заголовка окна, параметр `dir` — текущий каталог, параметр `filter` — фильтр, а параметр `options` устанавливает опции (см. описание метода `setOption()` в *разд. 10.5.1*). Метод возвращает выбранный файл или пустую строку. Пример:

```
QString f = QFileDialog::getSaveFileName(this,
                "Заголовок окна", "C:\\cpp\\projectsQt\\Test",
                "All (*);;Images (*.png *.jpg)");
qDebug() << f;
```

Результат выполнения этого кода показан на рис. 10.11.

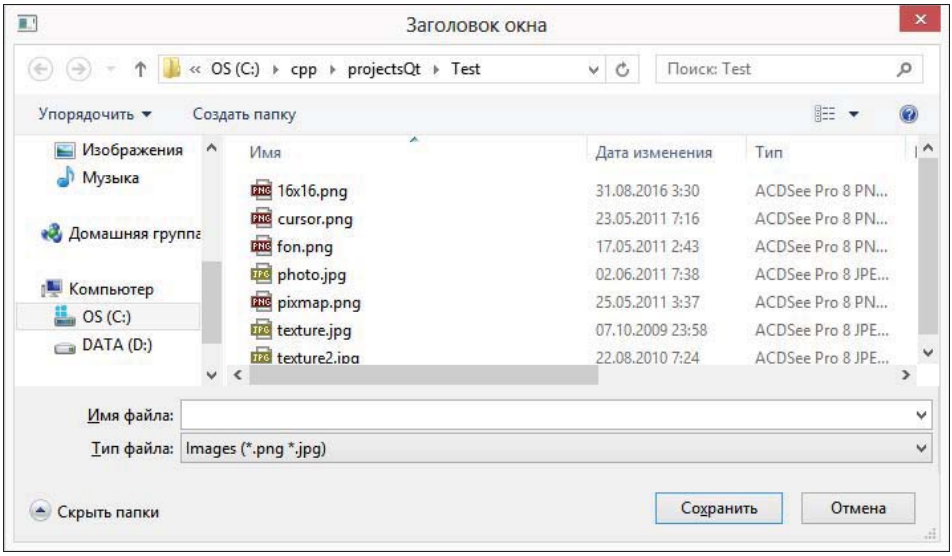

Рис. 10.11. Окно для сохранения файла

Можно также воспользоваться статическим методом `getOpenFileUrls()`:

```
static QUrl getSaveFileUrl(QWidget *parent = nullptr,
   const QString &caption = QString(), const QUrl &dir = QUrl(),
   const QString &filter = QString(), QString *selectedFilter = nullptr,
   QFileDialog::Options options = Options(),
   const QStringList &supportedSchemes = QStringList())
```

10.6. Окно для выбора цвета

Окно для выбора цвета (рис. 10.12) реализуется с помощью статического метода `getColor()` из класса `QColorDialog`. Прототип метода:

```
#include <QColorDialog>
static QColor getColor(const QColor &initial = Qt::white,
   QWidget *parent = nullptr, const QString &title = QString(),
   QColorDialog::ColorDialogOptions options = ColorDialogOptions())
```

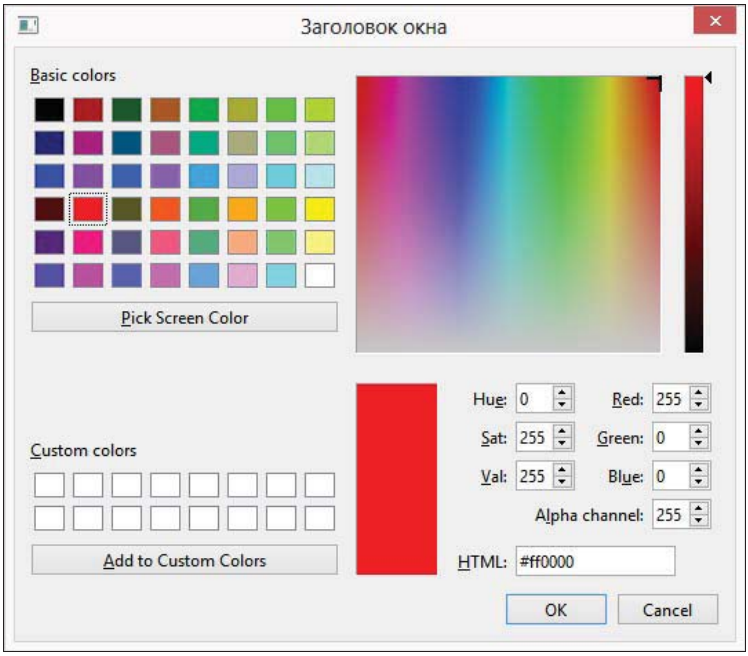

Рис. 10.12. Окно для выбора цвета

В параметре `parent` передается указатель на родительское окно или нулевой указатель. Параметр `initial` задает начальный цвет. В параметре `options` могут быть указаны следующие константы (или их комбинация):

- `QColorDialog::ShowAlphaChannel` — пользователь может выбрать значение альфа-канала;
- `QColorDialog::NoButtons` — кнопки **OK** и **Cancel** не отображаются;
- `QColorDialog::DontUseNativeDialog`.

Метод возвращает экземпляр класса `QColor`. Если пользователь нажимает кнопку **Cancel**, то объект будет невалидным. Пример:

```
QColor color = QColorDialog::getColor(QColor(255, 0, 0), this,
               "Заголовок окна", QColorDialog::ShowAlphaChannel);
if (color.isValid()) {
   qDebug() << color.red() << color.green() << color.blue()
            << color.alpha();
}
```

10.7. Окно для выбора шрифта

Окно для выбора шрифта реализуется с помощью статического метода `getFont()` из класса `QFontDialog`. Прототипы метода:

```
#include <QFontDialog>
static QFont getFont(bool *ok, QWidget *parent = nullptr)
```

```
static QFont getFont(bool *ok, const QFont &initial,
   QWidget *parent = nullptr, const QString &title = QString(),
   QFontDialog::FontDialogOptions options = FontDialogOptions())
```

В параметре `parent` передается указатель на родительское окно или нулевой указатель. Параметр `initial` задает начальный шрифт. В параметре `options` могут быть указаны константы `NoButtons`, `DontUseNativeDialog` и др. Статус операции доступен через параметр `ok` (переменная будет иметь значение `true`, если шрифт был выбран). Метод возвращает экземпляр класса `QFont` с выбранным шрифтом. Пример:

```
bool ok;
QFont font = QFontDialog::getFont(&ok, QFont("Tahoma", 16),
                                  this, "Заголовок окна");
if (ok) {
   qDebug() << font.family() << font.pointSize();
}
```

Результат выполнения этого кода показан на рис. 10.13.

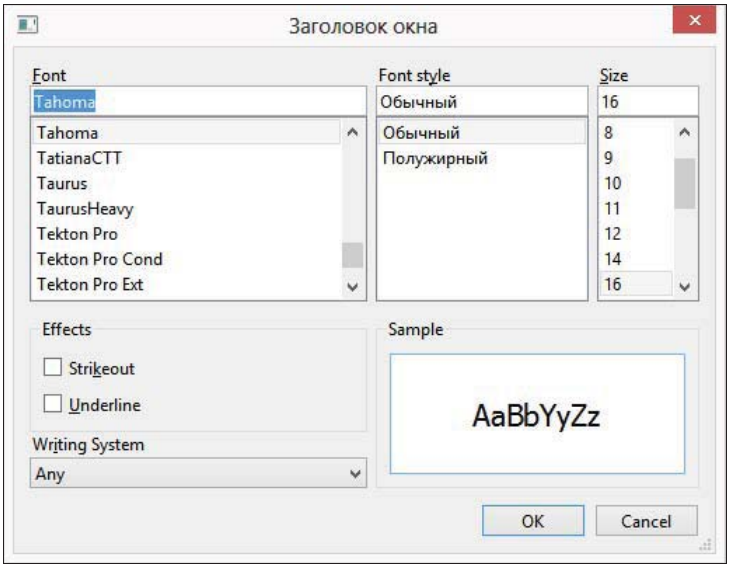

Рис. 10.13. Окно для выбора шрифта

10.8. Окно для вывода сообщения об ошибке

Класс `QErrorMessage` реализует немодальное диалоговое окно для вывода сообщения об ошибке (рис. 10.14). Окно содержит текстовое поле, в которое можно вставить текст сообщения об ошибке, и флажок. Если пользователь снимает флажок, то окно больше отображаться не будет. Иерархия наследования для класса `QErrorMessage` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QErrorMessage
```

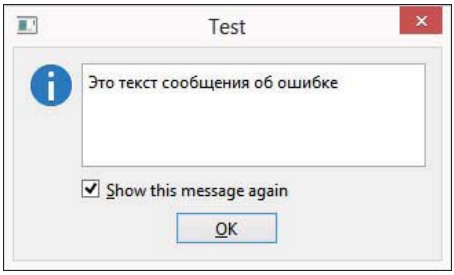

Рис. 10.14. Окно для вывода сообщения об ошибке

Формат конструктора класса `QErrorMessage`:

```
#include <QErrorMessage>
QErrorMessage(QWidget *parent = nullptr)
```

Для отображения окна предназначен метод `showMessage()`. Метод является слотом. Прототипы метода:

```
void showMessage(const QString &message)
void showMessage(const QString &message, const QString &type)
```

10.9. Окно с индикатором хода процесса

Класс `QProgressDialog` реализует диалоговое окно с индикатором хода процесса и кнопкой **Cancel**. Иерархия наследования для класса `QProgressDialog` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QProgressDialog
```

Форматы конструктора класса `QProgressDialog`:

```
#include <QProgressDialog>
QProgressDialog(QWidget *parent = nullptr,
                Qt::WindowFlags f = Qt::WindowFlags())
QProgressDialog(const QString &labelText,
                const QString &cancelButtonText, int minimum,
                int maximum, QWidget *parent = nullptr,
                Qt::WindowFlags f = Qt::WindowFlags())
```

Если в параметре `parent` передан указатель на родительское окно, то диалоговое окно будет центрироваться относительно родительского окна, а не относительно экрана. Параметр `f` задает тип окна *(см. разд. 3.2)*.

Класс `QProgressDialog` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setLabel()` — позволяет заменить объект надписи. Прототип метода:

  ```
  void setLabel(QLabel *label)
  ```

- `setBar()` — позволяет заменить объект индикатора. Прототип метода:

  ```
  void setBar(QProgressBar *bar)
  ```

- `setCancelButton()` — позволяет заменить объект кнопки. Прототип метода:

```
void setCancelButton(QPushButton *cancelButton)
```

- `setValue()` — задает новое значение индикатора. Если диалоговое окно является модальным, то при установке значения автоматически вызывается метод `processEvents()` объекта приложения. Метод является слотом. Прототип метода:

```
void setValue(int progress)
```

- `value()` — возвращает текущее значение индикатора в виде числа. Прототип метода:

```
int value() const
```

- `setLabelText()` — задает надпись, выводимую над индикатором. Метод является слотом. Прототип метода:

```
void setLabelText(const QString &text)
```

- `setCancelButtonText()` — задает надпись, выводимую на кнопке **Cancel**. Метод является слотом. Прототип метода:

```
void setCancelButtonText(const QString &cancelButtonText)
```

- `setRange()`, `setMinimum()` и `setMaximum()` — задают минимальное и максимальное значения. Если оба значения равны нулю, то внутри индикатора будут постоянно по кругу перемещаться сегменты, показывая ход выполнения процесса с неопределенным количеством шагов. Методы являются слотами. Прототипы методов:

```
void setRange(int min, int max)
void setMinimum(int min)
void setMaximum(int max)
```

- `setMinimumDuration()` — задает промежуток времени в миллисекундах перед отображением окна (по умолчанию значение равно `4000`). Окно может быть отображено ранее этого срока при установке значения. Метод является слотом. Прототип метода:

```
void setMinimumDuration(int ms)
```

- `reset()` — сбрасывает значение индикатора. Метод является слотом. Прототип метода:

```
void reset()
```

- `cancel()` — имитирует нажатие кнопки **Cancel**. Метод является слотом. Прототип метода:

```
void cancel()
```

- `setAutoClose()` — если в качестве параметра указано значение `true`, то при сбросе значения окно скрывается. Прототип метода:

```
void setAutoClose(bool close)
```

- `setAutoReset()` — если в качестве параметра указано значение `true`, то при достижении максимального значения будет автоматически произведен сброс значения. Прототип метода:

  ```
  void setAutoReset(bool reset)
  ```

- `wasCanceled()` — возвращает значение `true`, если была нажата кнопка **Cancel**. Прототип метода:

  ```
  bool wasCanceled() const
  ```

Класс `QProgressDialog` содержит сигнал `canceled()`, который генерируется при нажатии кнопки **Cancel**.

10.10. Создание многостраничного мастера

С помощью классов `QWizard` и `QWizardPage` можно создать диалоговое окно, в котором последовательно (или в произвольном порядке) отображаются различные страницы при нажатии кнопок **Back (Назад)** и **Next (Далее)**. Класс `QWizard` реализует контейнер для страниц, а отдельная страница описывается с помощью класса `QWizardPage`.

10.10.1. Класс *QWizard*

Класс `QWizard` реализует набор страниц, отображаемых последовательно или в произвольном порядке. Иерархия наследования для класса `QWizard` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDialog — QWizard
```

Формат конструктора класса `QWizard`:

```
#include <QWizard>
QWizard(QWidget *parent = nullptr,
        Qt::WindowFlags flags = Qt::WindowFlags())
```

Класс `QWizard` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `addPage()` — добавляет страницу в конец мастера и возвращает ее идентификатор. Прототип метода:

  ```
  int addPage(QWizardPage *page)
  ```

- `setPage()` — добавляет страницу в указанную позицию. Прототип метода:

  ```
  void setPage(int id, QWizardPage *page)
  ```

- `removePage()` — удаляет страницу с указанным идентификатором. Прототип метода:

  ```
  void removePage(int id)
  ```

- `page()` — возвращает указатель на страницу (экземпляр класса `QWizardPage`), соответствующую указанному идентификатору, или нулевой указатель, если страницы не существует. Прототип метода:

```
QWizardPage *page(int id) const
```

- `pageIds()` — возвращает список с идентификаторами страниц. Прототип метода:

```
QList<int> pageIds() const
```

- `currentId()` — возвращает идентификатор текущей страницы. Прототип метода:

```
int currentId() const
```

- `currentPage()` — возвращает указатель на текущую страницу (экземпляр класса `QWizardPage`) или нулевой указатель, если страницы не существует. Прототип метода:

```
QWizardPage *currentPage() const
```

- `setStartId()` — задает идентификатор начальной страницы. Прототип метода:

```
void setStartId(int id)
```

- `startId()` — возвращает идентификатор начальной страницы. Прототип метода:

```
int startId() const
```

- `visitedIds()` — возвращает список с идентификаторами посещенных страниц или пустой список. Прототип метода:

```
QList<int> visitedIds() const
```

- `hasVisitedPage()` — возвращает значение `true`, если страница была посещена, и `false` — в противном случае. Прототип метода:

```
bool hasVisitedPage(int id) const
```

- `back()` — имитирует нажатие кнопки **Back**. Метод является слотом. Прототип метода:

```
void back()
```

- `next()` — имитирует нажатие кнопки **Next**. Метод является слотом. Прототип метода:

```
void next()
```

- `restart()` — перезапускает мастера. Метод является слотом. Прототип метода:

```
void restart()
```

- `nextId()` — этот метод следует переопределить в классе, наследующем класс `QWizard`, если необходимо изменить порядок отображения страниц. Метод вызывается при нажатии кнопки **Next**. Метод должен возвращать идентификатор следующей страницы или значение `-1`. Прототип метода:

```
virtual int nextId() const
```

- `initializePage()` — этот метод следует переопределить в классе, наследующем класс `QWizard`, если необходимо производить настройку свойств компонентов на основе данных, введенных на предыдущих страницах. Метод вызывается при нажатии кнопки **Next** на предыдущей странице, но до отображения следующей страницы. Если установлена опция `IndependentPages`, то метод вызывается только при первом отображении страницы. Прототип метода:

```
virtual void initializePage(int id)
```

- `cleanupPage()` — этот метод следует переопределить в классе, наследующем класс `QWizard`, если необходимо контролировать нажатие кнопки **Back**. Метод вызывается при нажатии кнопки **Back** на текущей странице, но до отображения предыдущей страницы. Если установлена опция `IndependentPages`, то метод не вызывается. Прототип метода:

```
virtual void cleanupPage(int id)
```

- `validateCurrentPage()` — этот метод следует переопределить в классе, наследующем класс `QWizard`, если необходимо производить проверку данных, введенных на текущей странице. Метод вызывается при нажатии кнопки **Next** или **Finish**. Метод должен возвратить значение `true`, если данные корректны, и `false` — в противном случае. Если метод возвращает значение `false`, то переход на следующую страницу не производится. Прототип метода:

```
virtual bool validateCurrentPage()
```

- `setField()` — устанавливает значение указанного свойства. Создание свойства производится с помощью метода `registerField()` из класса `QWizardPage`. С помощью этого метода можно изменять значения компонентов, расположенных на разных страницах мастера. Прототип метода:

```
void setField(const QString &name, const QVariant &value)
```

- `field()` — возвращает значение указанного свойства. Создание свойства производится с помощью метода `registerField()` из класса `QWizardPage`. С помощью этого метода можно получить значения компонентов, расположенных на разных страницах мастера. Прототип метода:

```
QVariant field(const QString &name) const
```

- `setWizardStyle()` — задает стилевое оформление мастера. Прототип метода:

```
void setWizardStyle(QWizard::WizardStyle style)
```

  В качестве параметра указываются следующие константы:

  - `QWizard::ClassicStyle`;
  - `QWizard::ModernStyle`;
  - `QWizard::MacStyle`;
  - `QWizard::AeroStyle`;

- `setTitleFormat()` — задает формат отображения названия страницы. Прототип метода:

  ```
  void setTitleFormat(Qt::TextFormat format)
  ```

  В качестве параметра указываются следующие константы:

  - `Qt::PlainText` — простой текст;
  - `Qt::RichText` — форматированный текст;
  - `Qt::AutoText` — автоматическое определение (режим по умолчанию). Если текст содержит теги, то используется режим `RichText`, в противном случае — режим `PlainText`;

- `setSubTitleFormat()` — задает формат отображения описания страницы. Допустимые значения см. в описании метода `setTitleFormat()`. Прототип метода:

  ```
  void setSubTitleFormat(Qt::TextFormat format)
  ```

- `setButton()` — добавляет кнопку для указанной роли. Прототип метода:

  ```
  void setButton(QWizard::WizardButton which, QAbstractButton *button)
  ```

  В первом параметре указываются следующие константы:

  - `QWizard::BackButton` — кнопка **Back**;
  - `QWizard::NextButton` — кнопка **Next**;
  - `QWizard::CommitButton` — кнопка **Commit**;
  - `QWizard::FinishButton` — кнопка **Finish**;
  - `QWizard::CancelButton` — кнопка **Cancel** (если установлена опция `NoCancelButton`, то кнопка не отображается);
  - `QWizard::HelpButton` — кнопка **Help** (чтобы отобразить кнопку, необходимо установить опцию `HaveHelpButton`);
  - `QWizard::CustomButton1` — первая пользовательская кнопка (чтобы отобразить кнопку, необходимо установить опцию `HaveCustomButton1`);
  - `QWizard::CustomButton2` — вторая пользовательская кнопка (чтобы отобразить кнопку, необходимо установить опцию `HaveCustomButton2`);
  - `QWizard::CustomButton3` — третья пользовательская кнопка (чтобы отобразить кнопку, необходимо установить опцию `HaveCustomButton3`);

- `button()` — возвращает указатель на кнопку с заданной ролью. Прототип метода:

  ```
  QAbstractButton *button(QWizard::WizardButton which) const
  ```

- `setButtonText()` — устанавливает текст надписи для кнопки с указанной ролью. Прототип метода:

  ```
  void setButtonText(QWizard::WizardButton which, const QString &text)
  ```

- `buttonText()` — возвращает текст надписи кнопки с указанной ролью. Прототип метода:

  ```
  QString buttonText(QWizard::WizardButton which) const
  ```

- `setButtonLayout()` — задает порядок отображения кнопок. В качестве параметра указывается список с ролями кнопок. Список может также содержать значение константы `Stretch`, которая задает фактор растяжения. Прототип метода:

  ```
  void setButtonLayout(const QList<QWizard::WizardButton> &layout)
  ```

- `setPixmap()` — добавляет изображение для указанной роли. Прототип метода:

  ```
  void setPixmap(QWizard::WizardPixmap which, const QPixmap &pixmap)
  ```

  В первом параметре указываются следующие константы:

  - `QWizard::WatermarkPixmap` — изображение, которое занимает всю левую сторону при использовании стилей `ClassicStyle` или `ModernStyle`;
  - `QWizard::LogoPixmap` — небольшое изображение, отображаемое в правой части заголовка при использовании стилей `ClassicStyle` или `ModernStyle`;
  - `QWizard::BannerPixmap` — фоновое изображение, отображаемое в заголовке страницы при использовании стиля `ModernStyle`;
  - `QWizard::BackgroundPixmap` — фоновое изображение при использовании стиля `MacStyle`;

- `setOption()` — если во втором параметре указано значение `true`, то производит установку опции, указанной в первом параметре, а если указано значение `false`, то сбрасывает опцию. Прототип метода:

  ```
  void setOption(QWizard::WizardOption option, bool on = true)
  ```

  В первом параметре указываются следующие константы:

  - `QWizard::IndependentPages` — страницы не зависят друг от друга. Если опция установлена, то метод `initializePage()` будет вызван только при первом отображении страницы, а метод `cleanupPage()` не вызывается;
  - `QWizard::IgnoreSubTitles` — не отображать описание страницы в заголовке;
  - `QWizard::ExtendedWatermarkPixmap`;
  - `QWizard::NoDefaultButton` — не делать кнопки **Next** и **Finish** кнопками по умолчанию;
  - `QWizard::NoBackButtonOnStartPage` — не отображать кнопку **Back** на стартовой странице;
  - `QWizard::NoBackButtonOnLastPage` — не отображать кнопку **Back** на последней странице;
  - `QWizard::DisabledBackButtonOnLastPage` — сделать кнопку **Back** неактивной на последней странице;
  - `QWizard::HaveNextButtonOnLastPage` — показать неактивную кнопку **Next** на последней странице;
  - `QWizard::HaveFinishButtonOnEarlyPages` — показать неактивную кнопку **Finish** на не последних страницах;

- `QWizard::NoCancelButton` — не отображать кнопку **Cancel**;
- `QWizard::CancelButtonOnLeft` — поместить кнопку **Cancel** слева от кнопки **Back** (по умолчанию кнопка расположена справа от кнопок **Next** и **Finish**);
- `QWizard::HaveHelpButton` — показать кнопку **Help**;
- `QWizard::HelpButtonOnRight` — поместить кнопку **Help** у правого края окна;
- `QWizard::HaveCustomButton1` — показать пользовательскую кнопку с ролью `CustomButton1`;
- `QWizard::HaveCustomButton2` — показать пользовательскую кнопку с ролью `CustomButton2`;
- `QWizard::HaveCustomButton3` — показать пользовательскую кнопку с ролью `CustomButton3`;
- `QWizard::NoCancelButtonOnLastPage` — не отображать кнопку **Cancel** на последней странице;

❐ `setOptions()` — устанавливает несколько опций. Прототип метода:

```
void setOptions(QWizard::WizardOptions options)
```

Класс `QWizard` содержит следующие сигналы:

❐ `currentIdChanged(int)` — генерируется при изменении текущей страницы. Внутри обработчика через параметр доступен идентификатор текущей страницы;

❐ `customButtonClicked(int)` — генерируется при нажатии кнопок с ролями `CustomButton1`, `CustomButton2` и `CustomButton3`;

❐ `helpRequested()` — генерируется при нажатии кнопки **Help**;

❐ `pageAdded(int)` — генерируется при добавлении страницы;

❐ `pageRemoved(int)` — генерируется при удалении страницы.

10.10.2. Класс *QWizardPage*

Класс `QWizardPage` описывает одну страницу в многостраничном мастере. Иерархия наследования для класса `QWizardPage` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QWizardPage
```

Формат конструктора класса `QWizardPage`:

```
#include <QWizardPage>
QWizardPage(QWidget *parent = nullptr)
```

Класс `QWizardPage` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

❐ `wizard()` — возвращает указатель на объект мастера (экземпляр класса `QWizard`) или нулевой указатель, если страница не была добавлена. Прототип метода:

```
protected: QWizard *wizard() const
```

- `setTitle()` — задает название страницы. Прототип метода:

  ```
  void setTitle(const QString &title)
  ```

- `title()` — возвращает название страницы. Прототип метода:

  ```
  QString title() const
  ```

- `setSubTitle()` — задает описание страницы. Прототип метода:

  ```
  void setSubTitle(const QString &subTitle)
  ```

- `subTitle()` — возвращает описание страницы. Прототип метода:

  ```
  QString subTitle() const
  ```

- `setButtonText()` — устанавливает текст надписи для кнопки с указанной ролью (допустимые значения параметра `which` смотрите в описании метода `setButton()` из класса `QWizard`). Прототип метода:

  ```
  void setButtonText(QWizard::WizardButton which, const QString &text)
  ```

- `buttonText()` — возвращает текст надписи кнопки с указанной ролью. Прототип метода:

  ```
  QString buttonText(QWizard::WizardButton which) const
  ```

- `setPixmap()` — добавляет изображение для указанной роли (допустимые значения параметра `which` смотрите в описании метода `setPixmap()` из класса `QWizard`). Прототип метода:

  ```
  void setPixmap(QWizard::WizardPixmap which, const QPixmap &pixmap)
  ```

- `registerField()` — регистрирует свойство, с помощью которого можно получить доступ к значению компонента с любой страницы мастера. Прототип метода:

  ```
  protected: void registerField(const QString &name,
     QWidget *widget, const char *property = nullptr,
     const char *changedSignal = nullptr)
  ```

  В параметре `name` указывается произвольное название свойства в виде строки. Если в конце строки указать символ `*`, то компонент должен обязательно иметь значение (например, в поле должно быть введено какое-либо значение), иначе кнопки **Next** и **Finish** будут недоступны. Во втором параметре передается указатель на компонент. После регистрации свойства изменить значение компонента позволяет метод `setField()`, а получить значение — метод `field()`.

  В параметре `property` может быть указано свойство для получения и изменения значения, а в параметре `changedSignal` — сигнал, генерируемый при изменении значения. Назначить эти параметры для определенного класса позволяет также метод `setDefaultProperty()` из класса `QWizard`. Для стандартных классов по умолчанию используются следующие свойства и сигналы:

Класс:	**Свойство:**	**Сигнал:**
`QAbstractButton`	`checked`	`toggled()`
`QAbstractSlider`	`value`	`valueChanged()`

Класс:	Свойство:	Сигнал:
QComboBox	currentIndex	currentIndexChanged()
QDateTimeEdit	dateTime	dateTimeChanged()
QLineEdit	text	textChanged()
QListWidget	currentRow	currentRowChanged()
QSpinBox	value	valueChanged()

- `setField()` — устанавливает значение указанного свойства. С помощью этого метода можно изменять значения компонентов, расположенных на разных страницах мастера. Прототип метода:

  ```
  protected: void setField(const QString &name, const QVariant &value)
  ```

- `field()` — возвращает значение указанного свойства. С помощью этого метода можно получить значения компонентов, расположенных на разных страницах мастера. Прототип метода:

  ```
  protected: QVariant field(const QString &name) const
  ```

- `setFinalPage()` — если в качестве параметра указано значение `true`, то на странице будет отображаться кнопка **Finish**. Прототип метода:

  ```
  void setFinalPage(bool)
  ```

- `isFinalPage()` — возвращает значение `true`, если на странице будет отображаться кнопка **Finish**, и `false` — в противном случае. Прототип метода:

  ```
  bool isFinalPage() const
  ```

- `setCommitPage()` — если в качестве параметра указано значение `true`, то на странице будет отображаться кнопка **Commit**. Прототип метода:

  ```
  void setCommitPage(bool)
  ```

- `isCommitPage()` — возвращает значение `true`, если на странице будет отображаться кнопка **Commit**, и `false` — в противном случае. Прототип метода:

  ```
  bool isCommitPage() const
  ```

- `isComplete()` — этот метод вызывается, чтобы определить, должны ли кнопки **Next** и **Finish** быть доступными (метод возвращает значение `true`) или недоступными (метод возвращает значение `false`). Метод можно переопределить в классе, наследующем класс `QWizardPage`, и реализовать собственную проверку правильности ввода данных. При изменении возвращаемого значения необходимо генерировать сигнал `completeChanged()`. Прототип метода:

  ```
  virtual bool isComplete() const
  ```

- `nextId()` — этот метод следует переопределить в классе, наследующем класс `QWizardPage`, если необходимо изменить порядок отображения страниц. Метод вызывается при нажатии кнопки **Next**. Метод должен возвращать идентификатор следующей страницы или значение `-1`. Прототип метода:

  ```
  virtual int nextId() const
  ```

- `initializePage()` — этот метод следует переопределить в классе, наследующем класс `QWizardPage`, если необходимо производить настройку свойств компонентов на основе данных, введенных на предыдущих страницах. Метод вызывается при нажатии кнопки **Next** на предыдущей странице, но до отображения следующей страницы. Если установлена опция `IndependentPages`, то метод вызывается только при первом отображении страницы. Прототип метода:

  ```
  virtual void initializePage()
  ```

- `cleanupPage()` — этот метод следует переопределить в классе, наследующем класс `QWizardPage`, если необходимо контролировать нажатие кнопки **Back**. Метод вызывается при нажатии кнопки **Back** на текущей странице, но до отображения предыдущей страницы. Если установлена опция `IndependentPages`, то метод не вызывается. Прототип метода:

  ```
  virtual void cleanupPage()
  ```

- `validatePage()` — этот метод следует переопределить в классе, наследующем класс `QWizardPage`, если необходимо производить проверку данных, введенных на текущей странице. Метод вызывается при нажатии кнопки **Next** или **Finish**. Метод должен возвратить значение `true`, если данные корректны, и `false` — в противном случае. Если метод возвращает значение `false`, то переход на следующую страницу не производится. Прототип метода:

  ```
  virtual bool validatePage()
  ```

ГЛАВА 11

Создание SDI- и MDI-приложений

В Qt существует поддержка двух типов приложений:

- *SDI-приложения* (*Single Document Interface*) — приложение для отображения одного документа. Чтобы отобразить новый документ, необходимо закрыть предыдущий или запустить отдельный экземпляр приложения. Типичным примером таких приложений являются программы Блокнот, WordPad и Paint в операционной системе Windows. Чтобы создать SDI-приложение, следует установить центральный компонент с помощью метода `setCentralWidget()` из класса `QMainWindow`;
- *MDI-приложения* (*Multiple Document Interface*) — центральный компонент может отображать сразу несколько документов одновременно в разных вложенных окнах. Типичным примером MDI-приложения является программа Photoshop, позволяющая редактировать сразу несколько фотографий одновременно. Чтобы создать MDI-приложение, следует в качестве центрального компонента установить компонент `QMdiArea` с помощью метода `setCentralWidget()` из класса `QMainWindow`. Отдельное окно внутри MDI-области реализуется с помощью класса `QMdiSubWindow`.

11.1. Создание главного окна приложения

Класс `QMainWindow` реализует главное окно приложения, содержащее меню, панели инструментов, прикрепляемые панели, центральный компонент и строку состояния. Иерархия наследования для класса `QMainWindow` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QMainWindow
```

Конструктор класса `QMainWindow` имеет следующий формат:

```
#include <QMainWindow>
QMainWindow(QWidget *parent = nullptr,
            Qt::WindowFlags flags = Qt::WindowFlags())
```

В параметре `parent` передается указатель на родительское окно. Какие именно значения можно указать в параметре `flags`, мы рассматривали в *разд. 3.2*.

Класс `QMainWindow` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setCentralWidget()` — делает указанный компонент центральным компонентом главного окна. Прототип метода:

```
void setCentralWidget(QWidget *widget)
```

- `centralWidget()` — возвращает указатель на центральный компонент или нулевой указатель, если компонент не был установлен. Прототип метода:

```
QWidget *centralWidget() const
```

- `menuBar()` — возвращает указатель на главное меню. Прототип метода:

```
QMenuBar *menuBar() const
```

- `menuWidget()` — возвращает указатель на компонент, в котором расположено главное меню. Прототип метода:

```
QWidget *menuWidget() const
```

- `setMenuBar()` — позволяет установить пользовательское меню вместо стандартного. Прототип метода:

```
void setMenuBar(QMenuBar *menuBar)
```

- `setMenuWidget()` — позволяет установить компонент главного меню. Прототип метода:

```
void setMenuWidget(QWidget *menuBar)
```

- `createPopupMenu()` — создает контекстное меню с пунктами, позволяющими отобразить или скрыть панели инструментов и прикрепляемые панели, и возвращает указатель на меню (экземпляр класса `QMenu`). Это меню по умолчанию отображается при щелчке правой кнопкой мыши в области меню, панели инструментов или прикрепляемых панелей. Переопределив этот метод, можно реализовать собственное контекстное меню. Прототип метода:

```
virtual QMenu *createPopupMenu()
```

- `statusBar()` — возвращает указатель на строку состояния. Прототип метода:

```
QStatusBar *statusBar() const
```

- `setStatusBar()` — позволяет заменить стандартную строку состояния. Прототип метода:

```
void setStatusBar(QStatusBar *statusbar)
```

- `addToolBar()` — добавляет панель инструментов. Прототипы метода:

```
void addToolBar(QToolBar *toolbar)
void addToolBar(Qt::ToolBarArea area, QToolBar *toolbar)
QToolBar *addToolBar(const QString &title)
```

Первый прототип добавляет панель инструментов в верхнюю часть окна. Второй прототип позволяет указать местоположение панели. В качестве параметра `area` могут быть указаны следующие константы: `Qt::LeftToolBarArea` (слева), `Qt::RightToolBarArea` (справа), `Qt::TopToolBarArea` (сверху) или `Qt::BottomToolBarArea` (снизу). Третий прототип создает панель инструментов с указанным именем, добавляет ее в верхнюю область окна и возвращает указатель на нее;

- `insertToolBar()` — добавляет панель `toolbar` перед панелью `before`. Прототип метода:

  ```
  void insertToolBar(QToolBar *before, QToolBar *toolbar)
  ```

- `removeToolBar()` — удаляет панель инструментов из окна и скрывает ее. При этом объект панели инструментов не удаляется и далее может быть добавлен в другое место. Прототип метода:

  ```
  void removeToolBar(QToolBar *toolbar)
  ```

- `toolBarArea()` — возвращает местоположение указанной панели инструментов в виде значений констант `Qt::LeftToolBarArea` (слева), `Qt::RightToolBarArea` (справа), `Qt::TopToolBarArea` (сверху), `Qt::BottomToolBarArea` (снизу) или `Qt::NoToolBarArea` (положение не определено). Прототип метода:

  ```
  Qt::ToolBarArea toolBarArea(const QToolBar *toolbar) const
  ```

- `setToolButtonStyle()` — задает стиль кнопок на панели инструментов. Прототип метода:

  ```
  void setToolButtonStyle(Qt::ToolButtonStyle toolButtonStyle)
  ```

  В качестве параметра указываются следующие константы:

  - `Qt::ToolButtonIconOnly` — отображается только значок;
  - `Qt::ToolButtonTextOnly` — отображается только текст;
  - `Qt::ToolButtonTextBesideIcon` — текст отображается справа от значка;
  - `Qt::ToolButtonTextUnderIcon` — текст отображается под значком;
  - `Qt::ToolButtonFollowStyle` — зависит от используемого стиля;

- `toolButtonStyle()` — возвращает стиль кнопок на панели инструментов. Прототип метода:

  ```
  Qt::ToolButtonStyle toolButtonStyle() const
  ```

- `setIconSize()` — задает размеры значков. Прототип метода:

  ```
  void setIconSize(const QSize &iconSize)
  ```

- `iconSize()` — возвращает размеры значков. Прототип метода:

  ```
  QSize iconSize() const
  ```

- `setAnimated()` — если в качестве параметра указано значение `true` (используется по умолчанию), то вставка панелей инструментов и прикрепляемых панелей

в новое место по окончании перемещения будет производиться с анимацией. Метод является слотом. Прототип метода:

```
void setAnimated(bool enabled)
```

- `addToolBarBreak()` — вставляет разрыв в указанное место после всех добавленных ранее панелей. По умолчанию панели добавляются друг за другом на одной строке. С помощью этого метода можно поместить панели инструментов на двух и более строках. Прототип метода:

```
void addToolBarBreak(Qt::ToolBarArea area = Qt::TopToolBarArea)
```

- `insertToolBarBreak()` — вставляет разрыв перед указанной панелью инструментов. Прототип метода:

```
void insertToolBarBreak(QToolBar *before)
```

- `removeToolBarBreak()` — удаляет разрыв перед указанной панелью. Прототип метода:

```
void removeToolBarBreak(QToolBar *before)
```

- `toolBarBreak()` — возвращает значение `true`, если перед указанной панелью инструментов существует разрыв, и `false` — в противном случае. Прототип метода:

```
bool toolBarBreak(QToolBar *toolbar) const
```

- `addDockWidget()` — добавляет прикрепляемую панель. Прототипы метода:

```
void addDockWidget(Qt::DockWidgetArea area,
                   QDockWidget *dockwidget)
void addDockWidget(Qt::DockWidgetArea area,
                   QDockWidget *dockwidget, Qt::Orientation orientation)
```

  Первый прототип добавляет прикрепляемую панель в указанную область окна. В качестве параметра `area` могут быть указаны следующие константы: `Qt::LeftDockWidgetArea` (слева), `Qt::RightDockWidgetArea` (справа), `Qt::TopDockWidgetArea` (сверху) или `Qt::BottomDockWidgetArea` (снизу). Второй прототип позволяет дополнительно указать ориентацию при добавлении панели. В качестве параметра `orientation` могут быть указаны следующие константы: `Qt::Horizontal` или `Qt::Vertical`. Если указана константа `Qt::Horizontal`, то добавляемая панель будет расположена справа от ранее добавленной панели, а если `Qt::Vertical`, то снизу;

- `removeDockWidget()` — удаляет панель из окна и скрывает ее. При этом объект панели не удаляется и далее может быть добавлен в другую область. Прототип метода:

```
void removeDockWidget(QDockWidget *dockwidget)
```

- `dockWidgetArea()` — возвращает местоположение указанной панели в виде значений констант `Qt::LeftDockWidgetArea` (слева), `Qt::RightDockWidgetArea` (спра-

ва), `Qt::TopDockWidgetArea` (сверху), `Qt::BottomDockWidgetArea` (снизу) или `Qt::NoDockWidgetArea` (положение не определено). Прототип метода:

```
Qt::DockWidgetArea dockWidgetArea(QDockWidget *dockwidget) const
```

- `setDockOptions()` — устанавливает опции для прикрепляемых панелей. Значение по умолчанию: `AnimatedDocks | AllowTabbedDocks`. Прототип метода:

  ```
  void setDockOptions(QMainWindow::DockOptions options)
  ```

  В качестве значения указывается комбинация (через оператор `|`) следующих констант:

  - `QMainWindow::AnimatedDocks` — если опция установлена, то вставка панелей в новое место по окончании перемещения будет производиться с анимацией;
  - `QMainWindow::AllowNestedDocks` — если опция установлена, то области (в которые можно добавить панели) могут быть разделены, чтобы вместить больше панелей;
  - `QMainWindow::AllowTabbedDocks` — если опция установлена, то одна панель может быть наложена пользователем на другую. При этом добавляется панель с вкладками. С помощью щелчка мышью на заголовке вкладки можно выбрать отображаемую в данный момент панель;
  - `QMainWindow::ForceTabbedDocks` — если опция установлена, то панели не могут быть расположены рядом друг с другом. При этом опция `AllowNestedDocks` игнорируется;
  - `QMainWindow::VerticalTabs` — если опция установлена, то заголовки вкладок отображаются с внешнего края области (если область справа, то заголовки вкладок справа; если область слева, то заголовки слева; если область сверху, то заголовки вкладок сверху; если область снизу, то заголовки вкладок снизу). Если опция не установлена, то заголовки вкладок отображаются снизу. Опция `AllowTabbedDocks` должна быть установлена.

  ОБРАТИТЕ ВНИМАНИЕ

  Опции необходимо устанавливать до добавления прикрепляемых панелей. Исключением являются опции `AnimatedDocks` и `VerticalTabs`.

- `dockOptions()` — возвращает комбинацию установленных опций. Прототип метода:

  ```
  QMainWindow::DockOptions dockOptions() const
  ```

- `setDockNestingEnabled()` — если указано значение `true`, то метод устанавливает опцию `AllowNestedDocks`, а если указано значение `false`, то сбрасывает опцию. Метод является слотом. Прототип метода:

  ```
  void setDockNestingEnabled(bool enabled)
  ```

- `isDockNestingEnabled()` — возвращает значение `true`, если опция `AllowNestedDocks` установлена, и `false` — в противном случае. Прототип метода:

  ```
  bool isDockNestingEnabled() const
  ```

- `setTabPosition()` — задает позицию отображения заголовков вкладок прикрепляемых панелей для указанной области. По умолчанию заголовки вкладок отображаются снизу. Прототип метода:

```
void setTabPosition(Qt::DockWidgetAreas areas,
                    QTabWidget::TabPosition tabPosition)
```

  В качестве параметра `tabPosition` могут быть указаны следующие константы:

  - `QTabWidget::North` — сверху;
  - `QTabWidget::South` — снизу;
  - `QTabWidget::West` — слева;
  - `QTabWidget::East` — справа;

- `tabPosition()` — возвращает позицию отображения заголовков вкладок прикрепляемых панелей для указанной области. Прототип метода:

```
QTabWidget::TabPosition tabPosition(Qt::DockWidgetArea area) const
```

- `setTabShape()` — задает форму углов ярлыков вкладок в области заголовка. Прототип метода:

```
void setTabShape(QTabWidget::TabShape tabShape)
```

  Могут быть указаны следующие константы:

  - `QTabWidget::Rounded` — скругленные углы (значение по умолчанию);
  - `QTabWidget::Triangular` — треугольная форма;

- `tabShape()` — возвращает форму углов ярлыков вкладок в области заголовка. Прототип метода:

```
QTabWidget::TabShape tabShape() const
```

- `setCorner()` — позволяет закрепить указанный угол за определенной областью. По умолчанию верхние углы закреплены за верхней областью, а нижние углы — за нижней областью. Прототип метода:

```
void setCorner(Qt::Corner corner, Qt::DockWidgetArea area)
```

  В качестве параметра `area` могут быть указаны следующие константы: `Qt::LeftDockWidgetArea` (слева), `Qt::RightDockWidgetArea` (справа), `Qt::TopDockWidgetArea` (сверху) или `Qt::BottomDockWidgetArea` (снизу). В параметре `corner` указываются следующие константы:

  - `Qt::TopLeftCorner` — левый верхний угол;
  - `Qt::TopRightCorner` — правый верхний угол;
  - `Qt::BottomLeftCorner` — левый нижний угол;
  - `Qt::BottomRightCorner` — правый нижний угол;

- `corner()` — возвращает область, за которой закреплен указанный угол. Прототип метода:

```
Qt::DockWidgetArea corner(Qt::Corner corner) const
```

- `splitDockWidget()` — разделяет область, занимаемую панелью `first`, и добавляет панель `first` в первую часть, а панель `second` — во вторую часть. Порядок расположения частей зависит от параметра `orientation`. В качестве параметра `orientation` могут быть указаны следующие константы: `Qt::Horizontal` или `Qt::Vertical`. Если указана константа `Qt::Horizontal`, то панель `second` будет расположена справа, а если `Qt::Vertical`, то снизу. Если панель `first` расположена на вкладке, то панель `second` будет добавлена на новую вкладку, при этом разделения области не происходит. Прототип метода:

```
void splitDockWidget(QDockWidget *first, QDockWidget *second,
                     Qt::Orientation orientation)
```

- `tabifyDockWidget()` — размещает панель `second` над панелью `first`, создавая таким образом область с вкладками. Прототип метода:

```
void tabifyDockWidget(QDockWidget *first, QDockWidget *second)
```

- `tabifiedDockWidgets()` — возвращает список указателей на панели, которые расположены на других вкладках в области панели, указанной в качестве параметра. Прототип метода:

```
QList<QDockWidget *> tabifiedDockWidgets(QDockWidget *dockwidget) const
```

- `saveState()` — возвращает экземпляр класса `QByteArray` с размерами и положением всех панелей инструментов и прикрепляемых панелей. Эти данные можно сохранить (например, в файл), а затем восстановить с помощью метода `restoreState()`. Прототипы методов:

```
QByteArray saveState(int version = 0) const
bool restoreState(const QByteArray &state, int version = 0)
```

  Обратите внимание на то, что все панели инструментов и прикрепляемые панели должны иметь уникальные объектные имена. Задать объектное имя можно с помощью метода `setObjectName()` из класса `QObject`. Прототип метода:

```
void setObjectName(const QString &name)
```

  Чтобы сохранить размеры окна, следует воспользоваться методом `saveGeometry()` из класса `QWidget`. Метод возвращает экземпляр класса `QByteArray` с размерами окна. Чтобы восстановить размеры окна, следует воспользоваться методом `restoreGeometry()`. Прототипы методов:

```
QByteArray saveGeometry() const
bool restoreGeometry(const QByteArray &geometry)
```

- `restoreDockWidget()` — восстанавливает состояние указанной панели, если она создана после вызова метода `restoreState()`. Метод возвращает значение `true`, если состояние панели успешно восстановлено, и `false` — в противном случае. Прототип метода:

```
bool restoreDockWidget(QDockWidget *dockwidget)
```

11.2. Меню

Главное меню является основным компонентом пользовательского интерфейса, позволяющим компактно поместить множество команд, объединяя их в логические группы. Главное меню состоит из горизонтальной панели (реализуемой классом `QMenuBar`), на которой расположены отдельные меню (реализуются с помощью класса `QMenu`) верхнего уровня. Каждое меню может содержать множество пунктов (реализуемых с помощью класса `QAction`), разделители, а также вложенные меню. Пункт меню может содержать значок, текст и флажок, превращающий команду в переключатель.

Помимо главного меню в приложениях часто используются контекстные меню, которые обычно отображаются при щелчке правой кнопкой мыши в области компонента. Контекстное меню реализуется с помощью класса `QMenu` и отображается внутри метода с предопределенным названием `contextMenuEvent()` с помощью метода `exec()`, в который передаются глобальные координаты щелчка мышью.

11.2.1. Класс *QMenuBar*

Класс `QMenuBar` описывает горизонтальную панель меню. Панель меню реализована в главном окне приложения по умолчанию. Получить указатель на нее можно с помощью метода `menuBar()` из класса `QMainWindow`. Установить свою панель позволяет метод `setMenuBar()`. Иерархия наследования для класса `QMenuBar` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QMenuBar
```

Конструктор класса `QMenuBar` имеет следующий формат:

```
#include <QMenuBar>
QMenuBar(QWidget *parent = nullptr)
```

Класс `QMenuBar` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `addMenu()` — добавляет меню на панель и возвращает указатель на экземпляр класса `QAction`, с помощью которого, например, можно скрыть меню (с помощью метода `setVisible()`) или сделать его неактивным (с помощью метода `setEnabled()`). Прототип метода:

  ```
  QAction *addMenu(QMenu *menu)
  ```

- `addMenu()` — создает меню, добавляет его на панель и возвращает указатель на него (экземпляр класса `QMenu`). Внутри текста в параметре `title` символ `&`, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае символ, перед которым указан символ `&`, будет подчеркнут, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутого символа меню будет выбрано. Чтобы вывести символ `&`, необходимо его удвоить. Прототипы метода:

  ```
  QMenu *addMenu(const QString &title)
  QMenu *addMenu(const QIcon &icon, const QString &title)
  ```

- `insertMenu()` — добавляет меню `menu` перед пунктом `before`. Прототип метода:

```
QAction *insertMenu(QAction *before, QMenu *menu)
```

- `addAction()` — добавляет пункт в меню. Прототипы метода:

```
QAction *addAction(const QString &text)
QAction *addAction(const QString &text, const QObject *receiver,
                   const char *member)
QAction *addAction(const QString &text, const Obj *receiver,
                   PointerToMemberFunctionOrFunctor method)
QAction *addAction(const QString &text, Functor functor)
```

- `clear()` — удаляет все действия из панели меню. Прототип метода:

```
void clear()
```

- `setActiveAction()` — делает активным указанное действие. Прототип метода:

```
void setActiveAction(QAction *act)
```

- `activeAction()` — возвращает активное действие (указатель на экземпляр класса `QAction`) или нулевой указатель. Прототип метода:

```
QAction *activeAction() const
```

- `setDefaultUp()` — если в качестве параметра указано значение `true`, то пункты меню будут отображаться выше панели меню, а не ниже. Прототип метода:

```
void setDefaultUp(bool)
```

- `setVisible()` — если в качестве параметра указано значение `false`, то панель меню будет скрыта. Значение `true` отображает панель меню. Прототип метода:

```
virtual void setVisible(bool visible)
```

Класс `QMenuBar` содержит следующие сигналы:

- `hovered(QAction *)` — генерируется при наведении указателя мыши на пункт меню. Внутри обработчика через параметр доступен указатель на экземпляр класса `QAction`;
- `triggered(QAction *)` — генерируется при выборе пункта меню. Внутри обработчика через параметр доступен указатель на экземпляр класса `QAction`.

11.2.2. Класс *QMenu*

Класс `QMenu` реализует отдельное меню на панели меню, а также вложенное, плавающее и контекстное меню. Иерархия наследования для класса `QMenu` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QMenu
```

Форматы конструктора класса `QMenu`:

```
#include <QMenu>
QMenu(QWidget *parent = nullptr)
QMenu(const QString &title, QWidget *parent = nullptr)
```

В параметре parent передается указатель на родительский компонент. Внутри текста в параметре title символ &, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае символ, перед которым указан символ &, будет подчеркнут, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутого символа меню будет выбрано. Чтобы вывести символ &, необходимо его удвоить.

Класс QMenu наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- addAction() — добавляет пункт в меню. Прототипы метода:

```
void addAction(QAction *action)
QAction *addAction(const QString &text)
QAction *addAction(const QIcon &icon, const QString &text)
QAction *addAction(const QString &text, const QObject *receiver,
   const char *member, const QKeySequence &shortcut = {})
QAction *addAction(const QIcon &icon, const QString &text,
   const QObject *receiver, const char *member,
   const QKeySequence &shortcut = {})
QAction *addAction(const QString &text, Functor functor,
   const QKeySequence &shortcut = 0)
QAction *addAction(const QString &text, const QObject *context,
   Functor functor, const QKeySequence &shortcut = 0)
QAction *addAction(const QIcon &icon, const QString &text,
   Functor functor, const QKeySequence &shortcut = 0)
QAction *addAction(const QIcon &icon, const QString &text,
   const QObject *context, Functor functor,
   const QKeySequence &shortcut = 0)
```

  Внутри текста в параметре text символ &, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае символ, перед которым указан символ &, будет подчеркнут, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутого символа пункт меню будет выбран. Чтобы вывести символ &, необходимо его удвоить. Нажатие комбинации клавиш быстрого доступа сработает только в том случае, если меню, в котором находится пункт, является активным.

  Параметр shortcut задает комбинацию «горячих» клавиш, нажатие которых аналогично выбору пункта в меню. Нажатие комбинации «горячих» клавиш сработает даже в том случае, если меню не является активным.

  Добавить пункты в меню и удалить их позволяют следующие методы из класса QWidget:

  - addActions() — добавляет несколько пунктов в конец меню;
  - insertAction() — добавляет пункт action перед пунктом before;
  - insertActions() — добавляет несколько пунктов, указанных во втором параметре перед пунктом before;

- `actions()` — возвращает список с действиями;
- `removeAction()` — удаляет указанное действие из меню.

Прототипы методов:

```
void addActions(const QList<QAction *> &actions)
void insertAction(QAction *before, QAction *action)
void insertActions(QAction *before, const QList<QAction *> &actions)
QList<QAction *> actions() const
void removeAction(QAction *action)
```

- `addSeparator()` — добавляет разделитель в меню и возвращает указатель на экземпляр класса `QAction`. Прототип метода:

```
QAction *addSeparator()
```

- `insertSeparator()` — добавляет разделитель перед указанным пунктом и возвращает указатель на экземпляр класса `QAction`. Прототип метода:

```
QAction *insertSeparator(QAction *before)
```

- `addMenu()` — добавляет вложенное меню и возвращает указатель на экземпляр класса `QAction`. Прототип метода:

```
QAction *addMenu(QMenu *menu)
```

- `addMenu()` — создает вложенное меню, добавляет его в меню и возвращает указатель на него (экземпляр класса `QMenu`). Прототипы метода:

```
QMenu *addMenu(const QString &title)
QMenu *addMenu(const QIcon &icon, const QString &title)
```

- `insertMenu()` — добавляет вложенное меню `menu` перед пунктом `before`. Прототип метода:

```
QAction *insertMenu(QAction *before, QMenu *menu)
```

- `clear()` — удаляет все действия из меню. Прототип метода:

```
void clear()
```

- `isEmpty()` — возвращает значение `true`, если меню не содержит видимых пунктов, и `false` — в противном случае. Прототип метода:

```
bool isEmpty() const
```

- `menuAction()` — возвращает объект действия (экземпляр класса `QAction`), связанный с данным меню. С помощью этого объекта можно скрыть меню (с помощью метода `setVisible()`) или сделать его неактивным (с помощью метода `setEnabled()`). Прототип метода:

```
QAction *menuAction() const
```

- `setTitle()` — задает название меню. Прототип метода:

```
void setTitle(const QString &title)
```

- `title()` — возвращает название меню. Прототип метода:

  ```
  QString title() const
  ```

- `setIcon()` — задает значок меню. Прототип метода:

  ```
  void setIcon(const QIcon &icon)
  ```

- `icon()` — возвращает значок меню. Прототип метода:

  ```
  QIcon icon() const
  ```

- `setActiveAction()` — делает активным указанный пункт. Прототип метода:

  ```
  void setActiveAction(QAction *act)
  ```

- `activeAction()` — возвращает указатель на активный пункт или нулевой указатель. Прототип метода:

  ```
  QAction *activeAction() const
  ```

- `setDefaultAction()` — задает пункт по умолчанию. Прототип метода:

  ```
  void setDefaultAction(QAction *act)
  ```

- `defaultAction()` — возвращает указатель на пункт по умолчанию или нулевой указатель. Прототип метода:

  ```
  QAction *defaultAction() const
  ```

- `setTearOffEnabled()` — если в качестве параметра указано значение `true`, то в начало меню добавляется пункт с пунктирной линией, с помощью щелчка на котором можно оторвать меню от панели и сделать его плавающим (отображаемым в отдельном окне, которое можно разместить в любой части экрана). Прототип метода:

  ```
  void setTearOffEnabled(bool)
  ```

- `isTearOffEnabled()` — возвращает значение `true`, если меню может быть плавающим, и `false` — в противном случае. Прототип метода:

  ```
  bool isTearOffEnabled() const
  ```

- `isTearOffMenuVisible()` — возвращает значение `true`, если плавающее меню отображается в отдельном окне, и `false` — в противном случае. Прототип метода:

  ```
  bool isTearOffMenuVisible() const
  ```

- `hideTearOffMenu()` — скрывает плавающее меню. Прототип метода:

  ```
  void hideTearOffMenu()
  ```

- `setSeparatorsCollapsible()` — если в качестве параметра указано значение `true`, то вместо нескольких разделителей, идущих подряд, будет отображаться один разделитель. Кроме того, разделители, расположенные по краям меню, также будут скрыты. Прототип метода:

  ```
  void setSeparatorsCollapsible(bool collapse)
  ```

- `popup()` — отображает меню по указанным глобальным координатам. Если указан второй параметр, то меню отображается таким образом, чтобы по координатам был расположен указанный пункт меню. Прототип метода:

```
void popup(const QPoint &p, QAction *atAction = nullptr)
```

- `exec()` — отображает меню по указанным глобальным координатам и возвращает указатель на экземпляр класса `QAction` (соответствующий выбранному пункту) или нулевой указатель (если пункт не выбран, например нажата клавиша <Esc>). Если указан второй параметр, то меню отображается таким образом, чтобы по координатам был расположен указанный пункт меню. Прототипы метода:

```
QAction *exec()
QAction *exec(const QPoint &p, QAction *action = nullptr)
```

  Для отображения меню можно также воспользоваться статическим методом `exec()`. Метод отображает меню по указанным глобальным координатам и возвращает указатель на экземпляр класса `QAction` (соответствующий выбранному пункту) или нулевой указатель (если пункт не выбран, например нажата клавиша <Esc>). Прототипы метода:

```
static QAction *exec(const QList<QAction *> &actions,
       const QPoint &pos,
       QAction *at = nullptr, QWidget *parent = nullptr)
```

  Если указан параметр `at`, то меню отображается таким образом, чтобы по координатам был расположен указанный пункт меню. В параметре `parent` передается указатель на родительский компонент.

Класс `QMenu` содержит следующие сигналы:

- `hovered(QAction *)` — генерируется при наведении указателя мыши на пункт меню. Внутри обработчика через параметр доступен указатель на экземпляр класса `QAction`;
- `triggered(QAction *)` — генерируется при выборе пункта меню. Внутри обработчика через параметр доступен указатель на экземпляр класса `QAction`;
- `aboutToShow()` — генерируется перед отображением меню;
- `aboutToHide()` — генерируется перед сокрытием меню.

11.2.3. Контекстное меню

Чтобы создать контекстное меню, необходимо наследовать класс компонента и переопределить метод с названием `contextMenuEvent()`. Прототип метода:

```
virtual void contextMenuEvent(QContextMenuEvent *event)
```

Этот метод будет автоматически вызываться при щелчке правой кнопкой мыши в области компонента. Внутри метода через параметр `event` доступен экземпляр

класса QContextMenuEvent, который позволяет получить дополнительную информацию о событии. Класс QContextMenuEvent содержит следующие основные методы:

- x() и y() — возвращают координаты по осям X и Y соответственно в пределах области компонента. Прототипы методов:

```
int x() const
int y() const
```

- pos() — возвращает экземпляр класса QPoint с целочисленными координатами в пределах области компонента. Прототип метода:

```
const QPoint &pos() const
```

- globalX() и globalY() — возвращают координаты по осям X и Y соответственно в пределах экрана. Прототипы методов:

```
int globalX() const
int globalY() const
```

- globalPos() — возвращает экземпляр класса QPoint с координатами в пределах экрана. Прототип метода:

```
const QPoint &globalPos() const
```

Чтобы отобразить контекстное меню внутри метода contextMenuEvent(), следует вызвать метод exec() объекта меню и передать ему результат выполнения метода globalPos().

11.2.4. Класс *QAction*

Класс QAction описывает объект действия, который можно добавить в меню, на панель инструментов или просто прикрепить к какому-либо компоненту. Один и тот же объект действия допускается добавлять в несколько мест. Например, можно продублировать пункт меню на панели инструментов, что позволит одновременно сделать действие недоступным. Иерархия наследования для класса QAction выглядит так:

```
QObject — QAction
```

Форматы конструктора класса QAction:

```
#include <QAction>
QAction(QObject *parent = nullptr)
QAction(const QString &text, QObject *parent = nullptr)
QAction(const QIcon &icon, const QString &text, QObject *parent = nullptr)
```

В параметре parent передается указатель на родительский компонент. Внутри текста в параметре text символ &, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. В этом случае символ, перед которым указан символ &, будет подчеркнут, что является подсказкой пользователю. При одновременном нажатии клавиши <Alt> и подчеркнутого символа меню будет выбрано. Чтобы вывести символ &, необходимо его удвоить. Параметр icon устанавливает значок.

Класс `QAction` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `setText()` — задает название действия. Внутри текста в параметре `text` символ `&`, указанный перед буквой или цифрой, задает комбинацию клавиш быстрого доступа. Нажатие комбинации клавиш быстрого доступа сработает только в том случае, если меню, в котором находится пункт, является активным. Прототип метода:

  ```
  void setText(const QString &text)
  ```

- `text()` — возвращает название действия. Прототип метода:

  ```
  QString text() const
  ```

- `setIcon()` — устанавливает значок. Прототип метода:

  ```
  void setIcon(const QIcon &icon)
  ```

- `icon()` — возвращает значок. Прототип метода:

  ```
  QIcon icon() const
  ```

- `setIconVisibleInMenu()` — если в качестве параметра указано значение `false`, то значок в меню отображаться не будет. Прототип метода:

  ```
  void setIconVisibleInMenu(bool visible)
  ```

- `setSeparator()` — если в качестве параметра указано значение `true`, то объект является разделителем. Прототип метода:

  ```
  void setSeparator(bool)
  ```

- `isSeparator()` — возвращает значение `true`, если объект является разделителем, и `false` — в противном случае. Прототип метода:

  ```
  bool isSeparator() const
  ```

- `setShortcut()` — задает комбинацию «горячих» клавиш. Нажатие комбинации «горячих» клавиш по умолчанию сработает даже в том случае, если меню не является активным. Прототип метода:

  ```
  void setShortcut(const QKeySequence &shortcut)
  ```

- `setShortcuts()` — позволяет задать сразу несколько комбинаций «горячих» клавиш. Прототипы метода:

  ```
  void setShortcuts(const QList<QKeySequence> &shortcuts)
  void setShortcuts(QKeySequence::StandardKey key)
  ```

- `setShortcutContext()` — задает область действия комбинации «горячих» клавиш. Прототип метода:

  ```
  void setShortcutContext(Qt::ShortcutContext context)
  ```

  В качестве параметра указываются следующие константы:

  - `Qt::WidgetShortcut` — комбинация доступна, если родительский компонент имеет фокус;

- `Qt::WidgetWithChildrenShortcut` — комбинация доступна, если родительский компонент имеет фокус или любой дочерний компонент;
- `Qt::WindowShortcut` — комбинация доступна, если окно, в котором расположен компонент, является активным;
- `Qt::ApplicationShortcut` — комбинация доступна, если любое окно приложения является активным;

- `setToolTip()` — задает текст всплывающей подсказки. Прототип метода:

```
void setToolTip(const QString &tip)
```

- `toolTip()` — возвращает текст всплывающей подсказки. Прототип метода:

```
QString toolTip() const
```

- `setWhatsThis()` — задает текст справки. Прототип метода:

```
void setWhatsThis(const QString &what)
```

- `whatsThis()` — возвращает текст справки. Прототип метода:

```
QString whatsThis() const
```

- `setStatusTip()` — задает текст, который будет отображаться в строке состояния при наведении указателя мыши на пункт меню. Прототип метода:

```
void setStatusTip(const QString &statusTip)
```

- `statusTip()` — возвращает текст для строки состояния. Прототип метода:

```
QString statusTip() const
```

- `setCheckable()` — если в качестве параметра указано значение `true`, то действие является переключателем, который может находиться в двух состояниях — установленном и не установленном. Прототип метода:

```
void setCheckable(bool)
```

- `isCheckable()` — возвращает значение `true`, если действие является переключателем, и `false` — в противном случае. Прототип метода:

```
bool isCheckable() const
```

- `setChecked()` — если в качестве параметра указано значение `true`, то действие-переключатель будет находиться в установленном состоянии. Метод является слотом. Прототип метода:

```
void setChecked(bool)
```

- `isChecked()` — возвращает значение `true`, если действие-переключатель находится в установленном состоянии, и `false` — в противном случае. Прототип метода:

```
bool isChecked() const
```

- `setDisabled()` — если в качестве параметра указано значение `true`, то действие станет недоступным. Значение `false` делает действие снова доступным. Метод является слотом. Прототип метода:

```
void setDisabled(bool)
```

- `setEnabled()` — если в качестве параметра указано значение `false`, то действие станет недоступным. Значение `true` делает действие снова доступным. Метод является слотом. Прототип метода:

  ```
  void setEnabled(bool)
  ```

- `isEnabled()` — возвращает значение `true`, если действие доступно, и `false` — в противном случае. Прототип метода:

  ```
  bool isEnabled() const
  ```

- `setVisible()` — если в качестве параметра указано значение `false`, то действие будет скрыто. Значение `true` делает действие снова доступным. Метод является слотом. Прототип метода:

  ```
  void setVisible(bool)
  ```

- `isVisible()` — возвращает значение `false`, если действие скрыто, и `true` — в противном случае. Прототип метода:

  ```
  bool isVisible() const
  ```

- `setFont()` — устанавливает шрифт. Прототип метода:

  ```
  void setFont(const QFont &font)
  ```

- `font()` — возвращает экземпляр класса `QFont` с текущими настройками шрифта. Прототип метода:

  ```
  QFont font() const
  ```

- `setAutoRepeat()` — если в качестве параметра указано значение `true` (значение по умолчанию), то действие будет повторяться, пока удерживается нажатой комбинация «горячих» клавиш. Прототип метода:

  ```
  void setAutoRepeat(bool)
  ```

- `setPriority()` — задает приоритет действия. В качестве параметра указываются константы: `QAction::LowPriority` (низкий), `QAction::NormalPriority` (нормальный) или `QAction::HighPriority` (высокий). Прототип метода:

  ```
  void setPriority(QAction::Priority priority)
  ```

- `priority()` — возвращает текущий приоритет действия. Прототип метода:

  ```
  QAction::Priority priority() const
  ```

- `setData()` — позволяет сохранить пользовательские данные любого типа в объекте действия. Прототип метода:

  ```
  void setData(const QVariant &data)
  ```

- `data()` — возвращает пользовательские данные, сохраненные ранее с помощью метода `setData()`. Прототип метода:

  ```
  QVariant data() const
  ```

- `setActionGroup()` — добавляет действие в указанную группу. Прототип метода:

  ```
  void setActionGroup(QActionGroup *group)
  ```

- `actionGroup()` — возвращает указатель на группу или нулевой указатель. Прототип метода:

  ```
  QActionGroup *actionGroup() const
  ```

- `showStatusText()` — отправляет событие `StatusTip` указанному компоненту и возвращает значение `true`, если событие успешно отправлено. Прототип метода:

  ```
  bool showStatusText(QObject *object = nullptr)
  ```

- `hover()` — посылает сигнал `hovered()`. Метод является слотом. Прототип метода:

  ```
  void hover()
  ```

- `toggle()` — производит изменение состояния переключателя на противоположное. Метод является слотом. Прототип метода:

  ```
  void toggle()
  ```

- `trigger()` — посылает сигнал `triggered()`. Метод является слотом. Прототип метода:

  ```
  void trigger()
  ```

Класс `QAction` содержит следующие сигналы:

- `changed()` — генерируется при изменении действия;
- `hovered()` — генерируется при наведении указателя мыши на объект действия;
- `toggled(bool)` — генерируется при изменении состояния переключателя. Внутри обработчика через параметр доступно текущее состояние;
- `triggered(bool checked = false)` — генерируется при выборе пункта меню, нажатии кнопки на панели инструментов, нажатии комбинации клавиш или вызове метода `trigger()`.

11.2.5. Объединение переключателей в группу

Класс `QActionGroup` позволяет объединить несколько переключателей в группу. По умолчанию внутри группы может быть включен только один переключатель. При попытке включить другой переключатель ранее включенный переключатель автоматически отключается. Иерархия наследования для класса `QActionGroup` выглядит так:

```
QObject — QActionGroup
```

Конструктор класса `QActionGroup` имеет следующий формат:

```
#include <QActionGroup>
QActionGroup(QObject *parent)
```

В параметре `parent` передается указатель на родительский компонент. После создания объекта группы он может быть указан в качестве родителя при создании объектов действия. В этом случае действие автоматически добавляется в группу.

Класс `QActionGroup` содержит следующие основные методы:

- `addAction()` — добавляет объект действия в группу. Прототипы метода:

```
QAction *addAction(QAction *action)
QAction *addAction(const QString &text)
QAction *addAction(const QIcon &icon, const QString &text)
```

- `removeAction()` — удаляет объект действия из группы. Прототип метода:

```
void removeAction(QAction *action)
```

- `actions()` — возвращает список с объектами класса `QAction`, которые были добавлены в группу, или пустой список. Прототип метода:

```
QList<QAction *> actions() const
```

- `checkedAction()` — возвращает указатель на установленный переключатель внутри группы при использовании эксклюзивного режима или нулевой указатель. Прототип метода:

```
QAction *checkedAction() const
```

- `setExclusive()` — если в качестве параметра указано значение `true` (значение по умолчанию), то внутри группы может быть включен только один переключатель. При попытке включить другой переключатель ранее включенный переключатель автоматически отключается. Значение `false` отключает эксклюзивный режим. Метод является слотом. Прототип метода:

```
void setExclusive(bool)
```

- `setDisabled()` — если в качестве параметра указано значение `true`, то все действия внутри группы станут недоступными. Значение `false` делает действия снова доступными. Метод является слотом. Прототип метода:

```
void setDisabled(bool)
```

- `setEnabled()` — если в качестве параметра указано значение `false`, то все действия внутри группы станут недоступными. Значение `true` делает действия снова доступными. Метод является слотом. Прототип метода:

```
void setEnabled(bool)
```

- `isEnabled()` — возвращает значение `true`, если действия доступны, и `false` — в противном случае. Прототип метода:

```
bool isEnabled() const
```

- `setVisible()` — если в качестве параметра указано значение `false`, то все действия внутри группы будут скрыты. Значение `true` делает действия снова доступными. Метод является слотом. Прототип метода:

```
void setVisible(bool)
```

- `isVisible()` — возвращает значение `false`, если действия скрыты, и `true` — в противном случае. Прототип метода:

```
bool isVisible() const
```

11.3. Панели инструментов

Панели инструментов предназначены для отображения часто используемых команд. Добавить панель инструментов в главное окно позволяют методы `addToolBar()` и `insertToolBar()` из класса `QMainWindow`. Один из форматов метода `addToolBar()` позволяет указать область, к которой изначально прикреплена панель. По умолчанию с помощью мыши пользователь может переместить панель в другую область окна или отобразить панель в отдельном окне. Ограничить перечень областей, к которым можно прикрепить панель, позволяет метод `setAllowedAreas()` из класса `QToolBar`, а запретить отображение панели в отдельном окне позволяет метод `setFloatable()`.

11.3.1. Класс *QToolBar*

Класс `QToolBar` реализует панель инструментов, которую можно перемещать с помощью мыши. Иерархия наследования для класса `QToolBar` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QToolBar
```

Форматы конструктора класса `QToolBar`:

```
#include <QToolBar>
QToolBar(QWidget *parent = nullptr)
QToolBar(const QString &title, QWidget *parent = nullptr)
```

В параметре `parent` передается указатель на родительский компонент, а в параметре `title` задается название панели, которое отображается в контекстном меню при щелчке правой кнопкой мыши в области меню, панелей инструментов или на заголовке прикрепляемых панелей. С помощью контекстного меню можно скрыть или отобразить панель инструментов.

Класс `QToolBar` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `addAction()` — добавляет действие на панель инструментов. Прототипы метода:

```
void addAction(QAction *action);
QAction *addAction(const QString &text)
QAction *addAction(const QIcon &icon, const QString &text)
QAction *addAction(const QString &text, const QObject *receiver,
                   const char *member)
QAction *addAction(const QIcon &icon, const QString &text,
                   const QObject *receiver, const char *member)
QAction *addAction(const QString &text, Functor functor)
QAction *addAction(const QString &text, const QObject *context,
                   Functor functor)
QAction *addAction(const QIcon &icon, const QString &text, Functor functor)
QAction *addAction(const QIcon &icon, const QString &text,
                   const QObject *context, Functor functor)
```

Добавить действия на панель инструментов и удалить их позволяют следующие методы из класса `QWidget`: `addActions()`, `insertAction()`, `insertActions()`, `actions()`, `removeAction()`;

- `addSeparator()` — добавляет разделитель и возвращает указатель на экземпляр класса `QAction`. Прототип метода:

```
QAction *addSeparator()
```

- `insertSeparator()` — добавляет разделитель перед указанным действием и возвращает указатель на экземпляр класса `QAction`. Прототип метода:

```
QAction *insertSeparator(QAction *before)
```

- `addWidget()` — позволяет добавить компонент, например раскрывающийся список. Прототип метода:

```
QAction *addWidget(QWidget *widget)
```

- `insertWidget()` — добавляет компонент перед указанным действием и возвращает указатель на экземпляр класса `QAction`. Прототип метода:

```
QAction *insertWidget(QAction *before, QWidget *widget)
```

- `widgetForAction()` — возвращает указатель на компонент, который связан с указанным действием. Прототип метода:

```
QWidget *widgetForAction(QAction *action) const
```

- `clear()` — удаляет все действия из панели инструментов. Прототип метода:

```
void clear()
```

- `setAllowedAreas()` — задает области, к которым можно прикрепить панель инструментов. В качестве параметра указываются константы (или их комбинация через оператор `|`) `Qt::LeftToolBarArea` (слева), `Qt::RightToolBarArea` (справа), `Qt::TopToolBarArea` (сверху), `Qt::BottomToolBarArea` (снизу) или `Qt::AllToolBarAreas`. Прототип метода:

```
void setAllowedAreas(Qt::ToolBarAreas areas)
```

- `setMovable()` — если в качестве параметра указано значение `true` (значение по умолчанию), то панель можно перемещать с помощью мыши. Значение `false` запрещает перемещение. Прототип метода:

```
void setMovable(bool movable)
```

- `isMovable()` — возвращает значение `true`, если панель можно перемещать с помощью мыши, и `false` — в противном случае. Прототип метода:

```
bool isMovable() const
```

- `setFloatable()` — если в качестве параметра указано значение `true` (значение по умолчанию), то панель можно использовать как отдельное окно, а если указано значение `false`, то нет. Прототип метода:

```
void setFloatable(bool floatable)
```

- `isFloatable()` — возвращает значение `true`, если панель можно использовать как отдельное окно, и `false` — в противном случае. Прототип метода:

  ```
  bool isFloatable() const
  ```

- `isFloating()` — возвращает значение `true`, если панель отображается в отдельном окне, и `false` — в противном случае. Прототип метода:

  ```
  bool isFloating() const
  ```

- `setToolButtonStyle()` — задает стиль кнопок на панели инструментов. Метод является слотом. Прототип метода:

  ```
  void setToolButtonStyle(Qt::ToolButtonStyle toolButtonStyle)
  ```

  В качестве параметра указываются следующие константы:

  - `Qt::ToolButtonIconOnly` — отображается только значок;
  - `Qt::ToolButtonTextOnly` — отображается только текст;
  - `Qt::ToolButtonTextBesideIcon` — текст отображается справа от значка;
  - `Qt::ToolButtonTextUnderIcon` — текст отображается под значком;
  - `Qt::ToolButtonFollowStyle` — зависит от используемого стиля;

- `toolButtonStyle()` — возвращает стиль кнопок на панели инструментов. Прототип метода:

  ```
  Qt::ToolButtonStyle toolButtonStyle() const
  ```

- `setIconSize()` — задает размеры значков. Метод является слотом. Прототип метода:

  ```
  void setIconSize(const QSize &iconSize)
  ```

- `iconSize()` — возвращает размеры значков. Прототип метода:

  ```
  QSize iconSize() const
  ```

- `toggleViewAction()` — возвращает объект действия (указатель на экземпляр класса `QAction`), с помощью которого можно скрыть или отобразить панель. Прототип метода:

  ```
  QAction *toggleViewAction() const
  ```

Класс `QToolBar` содержит следующие сигналы (перечислены только основные сигналы; полный список смотрите в документации):

- `actionTriggered(QAction *)` — генерируется при нажатии кнопки на панели. Внутри обработчика через параметр доступен указатель на экземпляр класса `QAction`;
- `visibilityChanged(bool)` — генерируется при изменении видимости панели. Внутри обработчика через параметр доступно значение `true`, если панель видима, и `false` — если скрыта;

- `topLevelChanged(bool)` — генерируется при изменении положения панели. Внутри обработчика через параметр доступно значение `true`, если панель отображается в отдельном окне, и `false` — если прикреплена к области.

11.3.2. Класс *QToolButton*

При добавлении действия на панель инструментов автоматически создается кнопка, реализуемая классом `QToolButton`. Получить указатель на кнопку позволяет метод `widgetForAction()` из класса `QToolBar`. Иерархия наследования для класса `QToolButton` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QAbstractButton — QToolButton
```

Конструктор класса `QToolButton` имеет следующий формат:

```
#include <QToolButton>
QToolButton(QWidget *parent = nullptr)
```

Класс `QToolButton` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setDefaultAction()` — связывает объект действия с кнопкой. Метод является слотом. Прототип метода:

  ```
  void setDefaultAction(QAction *action)
  ```

- `defaultAction()` — возвращает указатель на объект действия (экземпляр класса `QAction`), связанный с кнопкой. Прототип метода:

  ```
  QAction *defaultAction() const
  ```

- `setToolButtonStyle()` — задает стиль кнопки. Метод является слотом. Допустимые значения параметра `style` см. в *разд. 11.3.1* в описании метода `setToolButtonStyle()`. Прототип метода:

  ```
  void setToolButtonStyle(Qt::ToolButtonStyle style)
  ```

- `toolButtonStyle()` — возвращает стиль кнопки. Прототип метода:

  ```
  Qt::ToolButtonStyle toolButtonStyle() const
  ```

- `setMenu()` — добавляет меню. Прототип метода:

  ```
  void setMenu(QMenu *menu)
  ```

- `menu()` — возвращает указатель на меню (экземпляр класса `QMenu`) или нулевой указатель. Прототип метода:

  ```
  QMenu *menu() const
  ```

- `showMenu()` — отображает меню, связанное с кнопкой. Метод является слотом. Прототип метода:

  ```
  void showMenu()
  ```

- `setPopupMode()` — задает режим отображения связанного с кнопкой меню. Прототип метода:

  ```
  void setPopupMode(QToolButton::ToolButtonPopupMode mode)
  ```

  В качестве параметра указываются следующие константы:

  - `QToolButton::DelayedPopup` — меню отображается при удержании кнопки нажатой некоторый промежуток времени;
  - `QToolButton::MenuButtonPopup` — справа от кнопки отображается кнопка со стрелкой, нажатие которой приводит к немедленному отображению меню;
  - `QToolButton::InstantPopup` — нажатие кнопки приводит к немедленному отображению меню. Сигнал `triggered()` при этом не генерируется;

- `popupMode()` — возвращает режим отображения связанного с кнопкой меню. Прототип метода:

  ```
  QToolButton::ToolButtonPopupMode popupMode() const
  ```

- `setArrowType()` — позволяет вместо стандартного значка действия установить значок в виде стрелки, указывающей в заданном направлении. В качестве параметра указываются константы `NoArrow` (значение по умолчанию), `UpArrow`, `DownArrow`, `LeftArrow` или `RightArrow`. Прототип метода:

  ```
  void setArrowType(Qt::ArrowType type)
  ```

- `setAutoRaise()` — если в качестве параметра указано значение `false`, то кнопка будет отображаться с рамкой. По умолчанию кнопка сливается с фоном, а при наведении указателя мыши становится выпуклой. Прототип метода:

  ```
  void setAutoRaise(bool enable)
  ```

Класс `QToolButton` содержит сигнал `triggered(QAction *)`, который генерируется при нажатии кнопки или комбинации клавиш, а также при выборе пункта в связанном меню. Внутри обработчика через параметр доступен указатель на экземпляр класса `QAction`.

11.4. Прикрепляемые панели

Если возможностей панелей инструментов недостаточно и необходимо добавить компоненты, занимающие много места (например, таблицу или иерархический список), то можно воспользоваться прикрепляемыми панелями. Прикрепляемые панели реализуются с помощью класса `QDockWidget`. Иерархия наследования для класса `QDockWidget` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QDockWidget
```

Форматы конструктора класса `QDockWidget`:

```
#include <QDockWidget>
QDockWidget(QWidget *parent = nullptr,
            Qt::WindowFlags flags = Qt::WindowFlags())
```

```
QDockWidget(const QString &title, QWidget *parent = nullptr,
            Qt::WindowFlags flags = Qt::WindowFlags())
```

В параметре `title` задается название панели, которое отображается в заголовке панели и в контекстном меню при щелчке правой кнопкой мыши в области меню, панелей инструментов или на заголовке прикрепляемых панелей. С помощью контекстного меню можно скрыть или отобразить прикрепляемую панель.

Класс `QDockWidget` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setWidget()` — устанавливает компонент, который будет отображаться на прикрепляемой панели. Прототип метода:

  ```
  void setWidget(QWidget *widget)
  ```

- `widget()` — возвращает указатель на компонент, расположенный внутри панели. Прототип метода:

  ```
  QWidget *widget() const
  ```

- `setTitleBarWidget()` — позволяет указать пользовательский компонент, отображаемый в заголовке панели. Прототип метода:

  ```
  void setTitleBarWidget(QWidget *widget)
  ```

- `titleBarWidget()` — возвращает указатель на пользовательский компонент, расположенный в заголовке панели. Прототип метода:

  ```
  QWidget *titleBarWidget() const
  ```

- `setAllowedAreas()` — задает области, к которым можно прикрепить панель. В качестве параметра указываются константы (или их комбинация через оператор `|`) `Qt::LeftDockWidgetArea` (слева), `Qt::RightDockWidgetArea` (справа), `Qt::TopDockWidgetArea` (сверху), `Qt::BottomDockWidgetArea` (снизу) или `Qt::AllDockWidgetAreas`. Прототип метода:

  ```
  void setAllowedAreas(Qt::DockWidgetAreas areas)
  ```

- `setFloating()` — если в качестве параметра указано значение `true`, то панель будет отображаться в отдельном окне, а если указано значение `false`, то панель будет прикреплена к какой-либо области. Прототип метода:

  ```
  void setFloating(bool floating)
  ```

- `isFloating()` — возвращает значение `true`, если панель отображается в отдельном окне, и `false` — в противном случае. Прототип метода:

  ```
  bool isFloating() const
  ```

- `setFeatures()` — устанавливает свойства панели. Прототип метода:

  ```
  void setFeatures(QDockWidget::DockWidgetFeatures features)
  ```

  В качестве параметра указывается комбинация (через оператор `|`) следующих констант:

- `QDockWidget::DockWidgetClosable` — панель можно закрыть;
- `QDockWidget::DockWidgetMovable` — панель можно перемещать с помощью мыши;
- `QDockWidget::DockWidgetFloatable` — панель может отображаться в отдельном окне;
- `QDockWidget::DockWidgetVerticalTitleBar` — заголовок панели отображается с левой стороны, а не сверху;
- `QDockWidget::NoDockWidgetFeatures` — панель нельзя закрыть, перемещать, и она не может отображаться в отдельном окне;

❐ `features()` — возвращает комбинацию установленных свойств панели. Прототип метода:

```
QDockWidget::DockWidgetFeatures features() const
```

❐ `toggleViewAction()` — возвращает объект действия (указатель на экземпляр класса `QAction`), с помощью которого можно скрыть или отобразить панель. Прототип метода:

```
QAction *toggleViewAction() const
```

Класс `QDockWidget` содержит следующие сигналы (перечислены только основные сигналы; полный список смотрите в документации):

❐ `dockLocationChanged(Qt::DockWidgetArea)` — генерируется при изменении области. Внутри обработчика через параметр доступна новая область, к которой прикреплена панель;

❐ `visibilityChanged(bool)` — генерируется при изменении видимости панели. Внутри обработчика через параметр доступно значение `true`, если панель видима, и `false` — если скрыта;

❐ `topLevelChanged(bool)` — генерируется при изменении положения панели. Внутри обработчика через параметр доступно значение `true`, если панель отображается в отдельном окне, и `false` — если прикреплена к области.

11.5. Управление строкой состояния

Класс `QStatusBar` реализует строку состояния, в которую можно выводить различные сообщения. Помимо текстовой информации в строку состояния можно добавить различные компоненты, например индикатор выполнения процесса. Строка состояния состоит из трех секций:

❐ *секция для временных сообщений*. Секция реализована по умолчанию. В эту секцию, например, выводятся сообщения, сохраненные в объекте действия с помощью метода `setStatusTip()`, при наведении указателя мыши на пункт меню или кнопку на панели инструментов. Вывести пользовательское сообщение во временную секцию можно с помощью метода `showMessage()`;

- *обычная секция*. При выводе временного сообщения содержимое обычной секции скрывается. Чтобы отображать сообщения в обычной секции, необходимо предварительно добавить туда компоненты с помощью метода `addWidget()` или `insertWidget()`. Добавленные компоненты выравниваются по левой стороне строки состояния;
- *постоянная секция*. При выводе временного сообщения содержимое постоянной секции не скрывается. Чтобы отображать сообщения в постоянной секции, необходимо предварительно добавить туда компоненты с помощью метода `addPermanentWidget()` или `insertPermanentWidget()`. Добавленные компоненты выравниваются по правой стороне строки состояния.

Получить указатель на строку состояния, установленную в главном окне, позволяет метод `statusBar()` из класса `QMainWindow`, а установить пользовательскую панель вместо стандартной можно с помощью метода `setStatusBar()`. Иерархия наследования для класса `QStatusBar` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QStatusBar
```

Формат конструктора класса `QStatusBar`:

```
#include <QStatusBar>
QStatusBar(QWidget *parent = nullptr)
```

Класс `QStatusBar` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `showMessage()` — выводит временное сообщение в строку состояния. Во втором параметре можно указать время в миллисекундах, на которое показывается сообщение. Если во втором параметре указано значение `0`, то сообщение показывается до тех пор, пока не будет выведено новое сообщение или вызван метод `clearMessage()`. Метод является слотом. Прототип метода:

  ```
  void showMessage(const QString &message, int timeout = 0)
  ```

- `currentMessage()` — возвращает временное сообщение, отображаемое в строке состояния. Прототип метода:

  ```
  QString currentMessage() const
  ```

- `clearMessage()` — удаляет временное сообщение из строки состояния. Метод является слотом. Прототип метода:

  ```
  void clearMessage()
  ```

- `addWidget()` — добавляет указанный компонент в конец обычной секции. В параметре `stretch` может быть указан фактор растяжения. Прототип метода:

  ```
  void addWidget(QWidget *widget, int stretch = 0)
  ```

- `insertWidget()` — добавляет компонент в указанную позицию обычной секции и возвращает индекс позиции. В параметре `stretch` может быть указан фактор растяжения. Прототип метода:

  ```
  int insertWidget(int index, QWidget *widget, int stretch = 0)
  ```

- `addPermanentWidget()` — добавляет указанный компонент в конец постоянной секции. В параметре `stretch` может быть указан фактор растяжения. Прототип метода:

  ```
  void addPermanentWidget(QWidget *widget, int stretch = 0)
  ```

- `insertPermanentWidget()` — добавляет компонент в указанную позицию постоянной секции и возвращает индекс позиции. В параметре `stretch` может быть указан фактор растяжения. Прототип метода:

  ```
  int insertPermanentWidget(int index, QWidget *widget, int stretch = 0)
  ```

- `removeWidget()` — удаляет компонент из обычной или постоянной секции. Обратите внимание на то, что сам компонент не удаляется, а только скрывается и лишается родителя. В дальнейшем компонент можно добавить в другое место. Прототип метода:

  ```
  void removeWidget(QWidget *widget)
  ```

- `setSizeGripEnabled()` — если в качестве параметра указано значение `true`, то в правом нижнем углу строки состояния будет отображаться маркер изменения размера. Значение `false` скрывает маркер. Прототип метода:

  ```
  void setSizeGripEnabled(bool)
  ```

Класс `QStatusBar` содержит сигнал `messageChanged(const QString&)`, который генерируется при изменении текста во временной секции. Внутри обработчика через параметр доступно новое сообщение или пустая строка.

11.6. MDI-приложения

MDI-приложения (*Multiple Document Interface*) позволяют отображать сразу несколько документов одновременно в разных вложенных окнах. Чтобы создать MDI-приложение, следует в качестве центрального компонента установить компонент `QMdiArea` с помощью метода `setCentralWidget()` из класса `QMainWindow`. Отдельное окно внутри MDI-области реализуется с помощью класса `QMdiSubWindow`.

11.6.1. Класс *QMdiArea*

Класс `QMdiArea` реализует MDI-область, внутри которой могут располагаться вложенные окна (экземпляры класса `QMdiSubWindow`). Иерархия наследования для класса `QMdiArea` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QFrame —
                        — QAbstractScrollArea — QMdiArea
```

Конструктор класса `QMdiArea` имеет следующий формат:

```
#include <QMdiArea>
QMdiArea(QWidget *parent = nullptr)
```

Класс `QMdiArea` содержит следующие методы (перечислены только основные методы; полный список смотрите в документации):

- `addSubWindow()` — создает вложенное окно, добавляет в него компонент `widget` и возвращает указатель на созданное окно (экземпляр класса `QMdiSubWindow`). Прототип метода:

```
QMdiSubWindow *addSubWindow(QWidget *widget,
                Qt::WindowFlags windowFlags = Qt::WindowFlags())
```

  Чтобы окно автоматически удалялось при закрытии, необходимо установить опцию `WA_DeleteOnClose`, а чтобы отобразить окно, следует вызвать метод `show()`;

- `activeSubWindow()` — возвращает указатель на активное вложенное окно (экземпляр класса `QMdiSubWindow`) или нулевой указатель. Прототип метода:

```
QMdiSubWindow *activeSubWindow() const
```

- `currentSubWindow()` — возвращает указатель на текущее вложенное окно (экземпляр класса `QMdiSubWindow`) или нулевой указатель. Результат выполнения этого метода аналогичен результату выполнения метода `activeSubWindow()`, если MDI-область находится в активном окне. Прототип метода:

```
QMdiSubWindow *currentSubWindow() const
```

- `subWindowList()` — возвращает список с указателями на все вложенные окна (экземпляры класса `QMdiSubWindow`), добавленные в MDI-область, или пустой список. Прототип метода:

```
QList<QMdiSubWindow *> subWindowList(
   QMdiArea::WindowOrder order = CreationOrder) const
```

  В параметре `order` указываются следующие константы:

  - `QMdiArea::CreationOrder` — окна в списке расположены в порядке добавления в MDI-область;
  - `QMdiArea::StackingOrder` — окна в списке расположены в порядке наложения. Последний элемент в списке будет содержать указатель на самое верхнее окно, а последний элемент — указатель на самое нижнее окно;
  - `QMdiArea::ActivationHistoryOrder` — окна в списке расположены в порядке истории получения фокуса. Последний элемент в списке будет содержать указатель на окно, получившее фокус последним;

- `removeSubWindow()` — удаляет вложенное окно из MDI-области. Прототип метода:

```
void removeSubWindow(QWidget *widget)
```

- `setActiveSubWindow()` — делает указанное вложенное окно активным. Если в качестве параметра указано значение `nullptr`, то все окна станут неактивными. Метод является слотом. Прототип метода:

```
void setActiveSubWindow(QMdiSubWindow *window)
```

- `setActivationOrder()` — задает режим передачи фокуса при использовании методов `activatePreviousSubWindow()`, `activateNextSubWindow()` и др. В параметре `order` указываются такие же константы, как и в параметре `order` метода `subWindowList()`. Прототип метода:

  ```
  void setActivationOrder(QMdiArea::WindowOrder order)
  ```

- `activationOrder()` — возвращает режим передачи фокуса. Прототип метода:

  ```
  QMdiArea::WindowOrder activationOrder() const
  ```

- `activatePreviousSubWindow()` — делает активным предыдущее вложенное окно. Метод является слотом. Порядок передачи фокуса устанавливается с помощью метода `setActivationOrder()`. Прототип метода:

  ```
  void activatePreviousSubWindow()
  ```

- `activateNextSubWindow()` — делает активным следующее вложенное окно. Метод является слотом. Порядок передачи фокуса устанавливается с помощью метода `setActivationOrder()`. Прототип метода:

  ```
  void activateNextSubWindow()
  ```

- `closeActiveSubWindow()` — закрывает активное вложенное окно. Метод является слотом. Прототип метода:

  ```
  void closeActiveSubWindow()
  ```

- `closeAllSubWindows()` — закрывает все вложенные окна. Метод является слотом. Прототип метода:

  ```
  void closeAllSubWindows()
  ```

- `cascadeSubWindows()` — упорядочивает вложенные окна, располагая их каскадом. Метод является слотом. Прототип метода:

  ```
  void cascadeSubWindows()
  ```

- `tileSubWindows()` — упорядочивает вложенные окна, располагая их мозаикой. Метод является слотом. Прототип метода:

  ```
  void tileSubWindows()
  ```

- `setViewMode()` — задает режим отображения документов в MDI-области. Прототип метода:

  ```
  void setViewMode(QMdiArea::ViewMode mode)
  ```

  В параметре `mode` указываются следующие константы:

  - `QMdiArea::SubWindowView` — в отдельном окне с рамкой (по умолчанию);
  - `QMdiArea::TabbedView` — на отдельной вкладке на панели с вкладками;

- `viewMode()` — возвращает режим отображения документов в MDI-области. Прототип метода:

  ```
  QMdiArea::ViewMode viewMode() const
  ```

- `setTabPosition()` — задает позицию отображения заголовков вкладок при использовании режима `TabbedView`. По умолчанию заголовки вкладок отображаются сверху. Прототип метода:

  ```
  void setTabPosition(QTabWidget::TabPosition position)
  ```

  В качестве параметра `position` могут быть указаны следующие константы:

  - `QTabWidget::North` — сверху;
  - `QTabWidget::South` — снизу;
  - `QTabWidget::West` — слева;
  - `QTabWidget::East` — справа;

- `tabPosition()` — возвращает позицию отображения заголовков вкладок при использовании режима `TabbedView`. Прототип метода:

  ```
  QTabWidget::TabPosition tabPosition() const
  ```

- `setTabShape()` — задает форму углов ярлыков вкладок при использовании режима `TabbedView`. Прототип метода:

  ```
  void setTabShape(QTabWidget::TabShape shape)
  ```

  Могут быть указаны следующие константы:

  - `QTabWidget::Rounded` — скругленные углы (значение по умолчанию);
  - `QTabWidget::Triangular` — треугольная форма;

- `tabShape()` — возвращает форму углов ярлыков вкладок в области заголовка при использовании режима `TabbedView`. Прототип метода:

  ```
  QTabWidget::TabShape tabShape() const
  ```

- `setBackground()` — задает кисть для заполнения фона MDI-области. Прототип метода:

  ```
  void setBackground(const QBrush &background)
  ```

- `setOption()` — если во втором параметре указано значение `true`, то производит установку опции, а если `false`, то сбрасывает опцию. В параметре `option` может быть указана константа `DontMaximizeSubWindowOnActivation`. Если эта опция установлена, то при передаче фокуса из максимально раскрытого окна отображаемое окно не будет максимально раскрываться. Прототип метода:

  ```
  void setOption(QMdiArea::AreaOption option, bool on = true)
  ```

- `testOption()` — возвращает значение `true`, если указанная опция установлена, и `false` — в противном случае. Прототип метода:

  ```
  bool testOption(QMdiArea::AreaOption option) const
  ```

Класс `QMdiArea` содержит сигнал `subWindowActivated(QMdiSubWindow *)`, который генерируется при изменении активности вложенных окон. Внутри обработчика через параметр доступен указатель на активное вложенное окно (экземпляр класса `QMdiSubWindow`) или нулевой указатель.

11.6.2. Класс *QMdiSubWindow*

Класс `QMdiSubWindow` реализует окно, которое может быть отображено внутри MDI-области. Иерархия наследования для класса `QMdiSubWindow` выглядит так:

```
(QObject, QPaintDevice) — QWidget — QMdiSubWindow
```

Формат конструктора класса `QMdiSubWindow`:

```
#include <QMdiSubWindow>
QMdiSubWindow(QWidget *parent = nullptr,
              Qt::WindowFlags flags = Qt::WindowFlags())
```

Класс `QMdiSubWindow` наследует все методы из базовых классов и содержит следующие дополнительные методы (перечислены только основные методы; полный список смотрите в документации):

- `setWidget()` — добавляет компонент в окно. Прототип метода:

  ```
  void setWidget(QWidget *widget)
  ```

- `widget()` — возвращает указатель на компонент внутри окна. Прототип метода:

  ```
  QWidget *widget() const
  ```

- `mdiArea()` — возвращает указатель на MDI-область. Прототип метода:

  ```
  QMdiArea *mdiArea() const
  ```

- `setSystemMenu()` — позволяет установить пользовательское системное меню окна вместо стандартного. Прототип метода:

  ```
  void setSystemMenu(QMenu *systemMenu)
  ```

- `systemMenu()` — возвращает указатель на системное меню окна. Прототип метода:

  ```
  QMenu *systemMenu() const
  ```

- `showSystemMenu()` — вызывает отображение системного меню окна. Метод является слотом. Прототип метода:

  ```
  void showSystemMenu()
  ```

- `setKeyboardSingleStep()` — устанавливает шаг изменения размера окна или его положения с помощью клавиш со стрелками. Чтобы изменить размеры окна или его положение с помощью клавиатуры, необходимо в системном меню окна выбрать пункт **Move** или **Size**. Значение по умолчанию `5`. Прототип метода:

  ```
  void setKeyboardSingleStep(int step)
  ```

- `setKeyboardPageStep()` — устанавливает шаг изменения размера окна или его положения с помощью клавиш со стрелками при удержании нажатой клавиши <Shift>. Значение по умолчанию `20`. Прототип метода:

  ```
  void setKeyboardPageStep(int step)
  ```

- `showShaded()` — сворачивает содержимое окна, оставляя только заголовок. Метод является слотом. Прототип метода:

  ```
  void showShaded()
  ```

- `isShaded()` — возвращает значение `true`, если отображается только заголовок, и `false` — в противном случае. Прототип метода:

  ```
  bool isShaded() const
  ```

- `setOption()` — если во втором параметре указано значение `true`, то производит установку опции, а если `false`, то сбрасывает опцию. Прототип метода:

  ```
  void setOption(QMdiSubWindow::SubWindowOption option, bool on = true)
  ```

  В параметре `option` могут быть указаны следующие константы:

  - `QMdiSubWindow::RubberBandResize` — если опция установлена, то при изменении размеров окна будут изменяться размеры вспомогательного компонента, а не самого окна. По окончании изменения размеров будут изменены размеры окна;
  - `QMdiSubWindow::RubberBandMove` — если опция установлена, то при изменении положения окна будет перемещаться вспомогательный компонент, а не само окно. По окончании перемещения будет изменено положение окна;

- `testOption()` — возвращает значение `true`, если указанная опция установлена, и `false` — в противном случае. Прототип метода:

  ```
  bool testOption(QMdiSubWindow::SubWindowOption option) const
  ```

Класс `QMdiSubWindow` содержит следующие сигналы:

- `aboutToActivate()` — генерируется перед отображением вложенного окна;
- `windowStateChanged(Qt::WindowStates,Qt::WindowStates)` — генерируется при изменении статуса окна. Внутри обработчика через первый параметр доступен старый статус, а через второй параметр — новый статус.

11.7. Добавление значка приложения в область уведомлений

Класс `QSystemTrayIcon` позволяет добавить значок приложения в область уведомлений, расположенную в правой части **Панели задач** в Windows. Иерархия наследования для класса `QSystemTrayIcon` выглядит так:

```
QObject — QSystemTrayIcon
```

Форматы конструктора класса `QSystemTrayIcon`:

```
#include <QSystemTrayIcon>
QSystemTrayIcon(QObject *parent = nullptr)
QSystemTrayIcon(const QIcon &icon, QObject *parent = nullptr)
```

Класс `QSystemTrayIcon` содержит следующие основные методы:

- `isSystemTrayAvailable()` — возвращает значение `true`, если можно отобразить значок в области уведомлений, и `false` — в противном случае. Метод является статическим. Прототип метода:

  ```
  static bool isSystemTrayAvailable()
  ```

- `setIcon()` — устанавливает значок. Установить значок можно также в конструкторе класса. Прототип метода:

```
void setIcon(const QIcon &icon)
```

- `icon()` — возвращает значок. Прототип метода:

```
QIcon icon() const
```

- `setContextMenu()` — устанавливает контекстное меню, отображаемое при щелчке правой кнопкой мыши на значке. Прототип метода:

```
void setContextMenu(QMenu *menu)
```

- `contextMenu()` — возвращает указатель на контекстное меню. Прототип метода:

```
QMenu *contextMenu() const
```

- `setToolTip()` — задает текст всплывающей подсказки. Прототип метода:

```
void setToolTip(const QString &tip)
```

- `toolTip()` — возвращает текст всплывающей подсказки. Прототип метода:

```
QString toolTip() const
```

- `setVisible()` — если в качестве параметра указано значение `true`, то отображает значок, а если `false`, то скрывает значок. Метод является слотом. Прототип метода:

```
void setVisible(bool visible)
```

- `show()` — отображает значок. Метод является слотом. Прототип метода:

```
void show()
```

- `hide()` — скрывает значок. Метод является слотом. Прототип метода:

```
void hide()
```

- `isVisible()` — возвращает значение `true`, если значок виден, и `false` — в противном случае. Прототип метода:

```
bool isVisible() const
```

- `geometry()` — возвращает экземпляр класса `QRect` с размерами и координатами значка на экране. Прототип метода:

```
QRect geometry() const
```

- `showMessage()` — позволяет отобразить сообщение в области уведомлений. Метод является слотом. Прототипы метода:

```
void showMessage(const QString &title, const QString &message,
   QSystemTrayIcon::MessageIcon ico =
   QSystemTrayIcon::Information,
   int millisecondsTimeoutHint = 10000)
void showMessage(const QString &title, const QString &message,
   const QIcon &icon, int millisecondsTimeoutHint = 10000)
```

Необязательный параметр `ico` задает значок, который отображается слева от заголовка сообщения. В качестве значения можно указать константы `NoIcon`, `Information`, `Warning` или `Critical`. Необязательный параметр `millisecondsTimeoutHint` задает промежуток времени, на который отображается сообщение. Обратите внимание на то, что сообщение может не показываться вообще, кроме того, значение параметра в некоторых операционных системах игнорируется;

- `supportsMessages()` — возвращает значение `true`, если вывод сообщений поддерживается, и `false` — в противном случае. Метод является статическим. Прототип метода:

```
static bool supportsMessages()
```

Класс `QSystemTrayIcon` содержит следующие сигналы:

- `activated(QSystemTrayIcon::ActivationReason)` — генерируется при щелчке мышью на значке. Внутри обработчика через параметр доступна причина в виде значения следующих констант:
  - `QSystemTrayIcon::Unknown` — неизвестная причина;
  - `QSystemTrayIcon::Context` — нажата правая кнопка мыши;
  - `QSystemTrayIcon::DoubleClick` — двойной щелчок мышью;
  - `QSystemTrayIcon::Trigger` — нажата левая кнопка мыши;
  - `QSystemTrayIcon::MiddleClick` — нажата средняя кнопка мыши;
- `messageClicked()` — генерируется при щелчке мышью в области сообщения.

Заключение

Вот и закончилось наше путешествие в мир Qt. Материал книги описывает лишь основы этой замечательной технологии. А здесь мы уточним, где найти дополнительную информацию.

Самым важным источником информации является официальный сайт Qt: **https://www.qt.io/**. На этом сайте вы найдете новости, а также ссылки на все другие ресурсы в Интернете.

На сайте **https://doc.qt.io/** расположена документация, которая обновляется в режиме реального времени. Библиотека постоянно совершенствуется, появляются новые классы, изменяются параметры методов и т. д. Регулярно посещайте этот сайт, и вы будете в курсе самых свежих новшеств.

Если в процессе изучения библиотеки Qt у вас возникнут какие-либо недопонимания, то не следует забывать, что Интернет предоставляет множество ответов на самые разнообразные вопросы. Достаточно в строке запроса поискового портала (например, **https://www.google.com/**) набрать свой вопрос. Наверняка уже кто-то сталкивался с подобной проблемой и описал ее решение на каком-либо сайте.

Свои замечания и пожелания вы можете оставить на странице книги на сайте издательства «БХВ-Петербург»: **http://www.bhv.ru/**. Все замеченные опечатки и неточности просим присылать на E-mail: **mail@bhv.ru** — не забудьте только указать название книги и имя автора.

ПРИЛОЖЕНИЕ

Описание электронного архива

Электронный архив к книге выложен на сервер издательства по адресу: **https://zip.bhv.ru/9785977511803.zip**. Ссылка доступна и со страницы книги на сайте **www.bhv.ru**.

Структура архива представлена в табл. П.1.

Таблица П.1. *Структура электронного архива*

Файл	Описание
Listings.doc	Содержит все пронумерованные листинги из книги
Examples.zip	Содержит дополнительные примеры
Readme.txt	Описание электронного архива

Предметный указатель

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

П

Р

С

Т

У

Ф

Ц

Ш

Э