Макс Шлее

Qt 4.8

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ НА C++

- Кроссплатформенная реализация приложений для Windows, Linux и Mac OS X
- Программирование графики, мультимедиа, веб-приложений, баз данных, сети, таймера, многопоточности, XML
- 222 завершенные программы
- Создание пользовательских интерфейсов с помощью Qt Quick и QML

+ Материалы на www.bhv.ru

Наиболее полное руководство

В ПОДЛИННИКЕ®

УДК 681.3.068+800.92C++
ББК 32.973.26-018.1
Ш68

Шлее М.

Ш68 Qt 4.8. Профессиональное программирование на C++. — СПб.: БХВ-Петербург, 2012. — 912 с.: ил. — (В подлиннике)

ISBN 978-5-9775-0736-3

Книга посвящена разработке приложений для Windows, Linux и Mac OS X с использованием библиотеки Qt версии 4.8. Подробно рассмотрены возможности, предоставляемые этой библиотекой, и описаны особенности, выгодно отличающие ее от других библиотек. Описана интегрированная среда разработки Qt Creator. Показано создание пользовательских интерфейсов с помощью Qt Quick и QML. Книга содержит исчерпывающую информацию о классах Qt 4, а также даны практические рекомендации их применения, проиллюстрированные на большом количестве подробно прокомментированных примеров. Проекты примеров из книги размещены на сайте издательства.

Для программистов

УДК 681.3.068+800.92C++
ББК 32.973.26-018.1

Группа подготовки издания:

Главный редактор	*Екатерина Кондукова*
Зам. главного редактора	*Игорь Шишигин*
Зав. редакцией	*Григорий Добин*
Редактор	*Юрий Рожко*
Компьютерная верстка	*Ольги Сергиенко*
Корректор	*Зинаида Дмитриева*
Дизайн серии	*Инны Тачиной*
Оформление обложки	*Марины Дамбиевой*

Подписано в печать 31.03.12.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 73,53.
Тираж 2000 экз. Заказ №
"БХВ-Петербург", 190005, Санкт-Петербург, Измайловский пр., 29.

Первая Академическая типография "Наука"
199034, Санкт-Петербург, 9 линия, 12/28

ISBN 978-5-9775-0736-3

Оглавление

ГЛАВА 1

Обзор иерархии классов Qt

Если вы хотите знать территорию — нужно сначала изучить карту.

Тони Бьюзен

Первая программа на Qt

Как заведено, в самом начале знакомства нужно поздороваться, и, чтобы никого не оставить без внимания, лучше всего обратиться сразу ко всему миру. Давайте для этого напишем короткую программу "Hello World" ("Здравствуй, Мир"), результат выполнения которой показан на рис. 1.1.

Написание подобного рода программ стало уже традицией при знакомстве с новым языком или библиотекой. Хотя подобный пример не в состоянии продемонстрировать весь потенциал и возможности самой библиотеки, он дает представление о базовых понятиях. Данный пример позволяет оценить объем и сложность процесса реализации программ, использующих эту библиотеку. Кроме того, при помощи примера можно убедиться, что все необходимое для компиляции и компоновки установлено правильно.

Рис. 1.1. Окно программы "Hello World"

Листинг 1.1. Программа "Hello World". Файл hello.cpp

```
#include <QtGui>
int main(int argc, char** argv)
{
    QApplication app(argc, argv);
    QLabel lbl("Hello, World !");
    lbl.show();
    return app.exec();
}
```

В первой строке листинга 1.1 подключается заголовочный файл `QtGui`, который представляет собой файл модуля, включающий в себя заголовочные файлы для используемых в нашей программе классов: `QApplication` и `QLabel`. Конечно, мы могли бы обойтись и без модуля

QtGui, а непосредственно подключить заголовочные файлы для поддержки классов QApplication и QLabel, но при большем количестве классов разных модулей, задействованных в программе, читаемость самой программы заметно бы ухудшилась. Кроме того, подключение модулей дает возможность ускорить компиляцию самой программы за счет предварительно откомпилированных заголовочных файлов (Precompiled Headers) в том случае, если ваш компилятор позволяет это делать.

Теперь давайте разберем наш пример. Сначала создается объект класса QApplication, который осуществляет контроль и управление приложением. Для его создания в конструктор этого класса необходимо передать два аргумента. Первый аргумент представляет собой информацию о количестве аргументов в командной строке, с которой происходит обращение к программе, а второй — это указатель на массив символьных строк, содержащих аргументы, по одному в строке. Любая использующая Qt-программа с графическим интерфейсом должна создавать только один объект этого класса, и он должен быть создан до использования операций, связанных с пользовательским интерфейсом.

Затем создается объект класса QLabel. После создания элементы управления Qt по умолчанию невидимы, и для их отображения необходимо вызвать метод show(). Объект класса QLabel является *основным управляющим элементом приложения*, что позволяет завершить работу приложения при закрытии окна элемента. Если вдруг окажется, что в созданном приложении имеется сразу несколько независимых друг от друга элементов управления, то при закрытии окна последнего такого элемента управления завершится и само приложение. Это правильно, иначе приложение осталось бы в памяти компьютера и использовало бы его ресурсы.

Наконец, в последней строке программы приложение запускается вызовом QApplication::exec(). С его запуском приводится в действие цикл обработки событий, определенный в классе QCoreApplication, являющимся базовым для QApplication. Этот цикл передает получаемые от системы события на обработку соответствующим объектам. Он продолжается до тех пор, пока либо не будет вызван статический метод QCoreApplication::exit(), либо не закроется окно последнего элемента управления. По завершению работы приложения метод QApplication::exec() возвращает значение целого типа, содержащее код, информирующий о его завершении.

Модули Qt

В начале изучения классов новой библиотеки создается ощущение перенасыщения из-за большого объема информации. Иерархия классов Qt имеет четкую внутреннюю структуру, которую важно понять, чтобы уметь хорошо и интуитивно ориентироваться в этой библиотеке.

Библиотека Qt — это множество классов (более 500), которые охватывают большую часть функциональных возможностей операционных систем, предоставляя разработчику мощные механизмы, расширяющие и, вместе с тем, упрощающие разработку приложений. При этом не нарушается идеология операционной системы. Qt не является единым целым, она разбита на модули (табл. 1.1).

Любая Qt-программа так или иначе должна использовать хотя бы один из модулей, в большинстве случаев это QtCore и QtGui, поэтому эти два модуля определены в программе создания make-файлов (см. *главу 3*) по умолчанию. Для использования других модулей в своих проектах необходимо перечислить их в проектном файле (см. *главу 3*). Например, чтобы добавить модули, нужно написать

```
QT += opengl network sql
```

Таблица 1.1. *Модули Qt*

Библиотека	Обозначение в проектном файле	Назначение
QtCore	core	Основополагающий модуль, состоящий из классов, не связанных с графическим интерфейсом
QtGui	gui	Модуль для программирования графического интерфейса
QtNetwork	network	Модуль для программирования сети
QtOpenGL	opengl	Модуль для программирования графики OpenGL
QtSql	sql	Модуль для программирования баз данных
QtSvg	svg	Модуль для работы с SVG (Scalable Vector Graphics, масштабируемая векторная графика)
QtXml	xml	Модуль поддержки XML, классы, относящиеся к SAX и DOM
Qt3Support	qt3support	Модуль, содержащий классы для совместимости с предыдущей библиотекой Qt
QtScript	script	Модуль поддержки языка сценариев
Phonon	phonon	Модуль мультимедиа
QtWebKit	webkit	Модуль для создания Web-приложений
QtScriptTools	scripttools	Модуль дополнительных возможностей поддержки языка сценария. На настоящий момент предоставляет отладчик
QtTest	test	Модуль, содержащий классы для тестирования кода

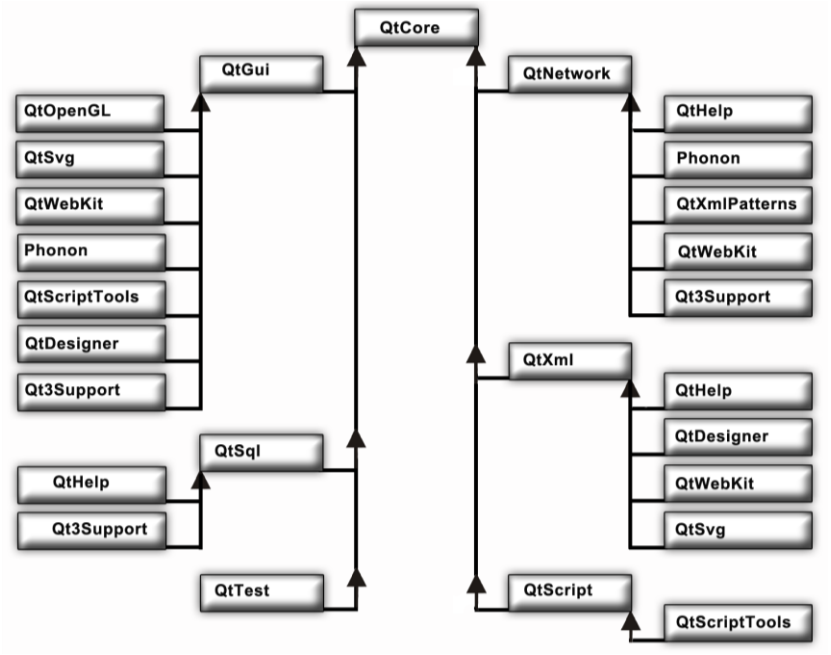

Рис. 1.2. Диаграмма некоторых модульных зависимостей

А чтобы исключить модуль из проекта:

```
QT -= gui
```

Наиболее значимый из перечисленных в табл. 1.1 модулей — это QtCore, т. к. он является базовым для всех остальных модулей (рис. 1.2). Далее идут модули, которые непосредственно зависят от QtCore, это — QtNetwork, QtGui, QtSql и QtXml. И, наконец, модули, зависящие от только что упомянутых модулей — Qt3Support, QtOpenGL и QtSvg.

Для каждого модуля Qt предоставляет отдельный заголовочный файл, содержащий заголовочные файлы всех классов этого модуля. Название этого заголовочного файла соответствует названию самого модуля. Например, для включения QtGui модуля нужно добавить в программу строку

```
#include <QtGui>
```

Пространство имен Qt

Пространство имен Qt содержит ряд типов перечислений и констант, которые часто применяются при программировании. Если вам необходимо получить доступ к какой-либо константе этого пространства имен, то вы должны указать префикс `Qt` (например, не `red`, а `Qt::red`). Если вы все-таки хотите опускать префикс `Qt`, то необходимо в начале файла с исходным кодом добавить следующую директиву:

```
using namespace Qt;
```

Модуль QtCore

Как уже было сказано ранее, базовым является модуль QtCore. Этот модуль является базовым для приложений и не содержит классов, относящихся к интерфейсу пользователя. Если вы собираетесь реализовать консольное приложение, то, вполне возможно, вы можете ограничиться одним этим модулем. В модуль QtCore входят более 200 классов, вот некоторые из них:

- контейнерные классы `QList`, `QVector`, `QMap` (см. *главу 4*);
- классы для ввода и вывода `QIODevice`, `QTextStream`, `QFile` (см. *главу 37*);
- классы процесса `QProcess` и для программирования многопоточности `QThread`, `QWaitCondition`, `QMutex` (см. *главу 39*);
- классы для работы с таймером `QBasicTimer` и `QTimer` (см. *главу 38*);
- классы для работы с датой и временем `QDate` и `QTime` (см. *главу 38*);
- класс `QObject`, являющийся *краеугольным камнем* объектной модели Qt (см. *главу 2*);
- базовый класс событий `QEvent` (см. *главу 14*);
- класс для сохранения настроек приложения `QSettings` (см. *главу 29*);
- класс приложения `QCoreApplication`, из объекта которого, если требуется, можно запустить цикл событий.

Давайте немного остановимся на классе `QCoreApplication`. Объект класса приложения `QCoreApplication` можно образно сравнить с сосудом, содержащим объекты, подсоединенные к контексту операционной системы. Срок жизни объекта класса `QCoreApplication` соответствует продолжительности работы всего приложения, и он остается доступным в любой

момент работы программы. Объект класса `QCoreApplication` должен создаваться в приложении только один раз. К задачам этого объекта можно отнести:

- управление событиями между приложением и операционной системой;
- передачу и предоставление аргументов командной строки.

Кроме того, `QCoreApplication` можно унаследовать, для того чтобы перезаписать некоторые методы, а также задействовать сам объект для дополнительных глобальных данных, используемых внутри приложения. Такой подход может избавить вас от нежелательного использования шаблона проектирования Singleton.

Модуль QtGui

Этот модуль содержит в себе классы, необходимые для программирования графического интерфейса пользователя. В этот модуль входят около 300 классов. Вот некоторые из них:

- класс `QWidget` — это базовый класс для всех элементов управления библиотеки Qt. По своему внешнему виду это не что иное, как заполненный четырехугольник, но за этой внешней простотой скрывается большой потенциал непростых функциональных возможностей. Этот класс насчитывает 254 метода и 53 свойства. В *главе 5* этому элементу уделено особое внимание;
- классы для автоматического размещения элементов `QVBoxLayout`, `QHBoxLayout` (см. *главу 6*);
- классы элементов отображения `QLabel`, `QLCDNumber` (см. *главу 7*);
- классы кнопок `QPushButton`, `QCheckBox`, `QRadioButton` (см. *главу 8*);
- классы элементов установок `QSlider`, `QScrollBar` (см. *главу 9*);
- классы элементов ввода `QLineEdit`, `QSpinBox` (см. *главу 10*);
- классы элементов выбора `QComboBox`, `QToolBox` (см. *главу 11*);
- классы меню `QMainWindow` и `QMenu` (см. *главы 32* и *35*);
- классы окон сообщений и диалоговых окон `QMessageBox`, `QDialog` (см. *главу 33*);
- классы для рисования `QPainter`, `QBrush`, `QPen`, `QColor` (см. *главу 18*);
- классы для растровых изображений `QImage`, `QPixmap` (см. *главу 19*);
- классы стилей `QMotifStyle`, `QWindowsStyle` и др. (см. *главу 26*). Как отдельному элементу, так и всему приложению может быть присвоен определенный стиль, изменяющий их внешний облик;
- класс приложения `QApplication`, который предоставляет цикл событий.

Давайте рассмотрим немного поподробнее последний класс, класс `QApplication`, с которым мы встречались в самом первом примере. Все, что было сказано выше о классе `QCoreApplication`, относится также и к этому классу, т. к. он является прямым его наследником. Объект класса `QApplication` представляет собой центральный контрольный пункт всех Qt-приложений, имеющих пользовательский интерфейс. Данный объект используется для получения событий клавиатуры, мыши, таймера и других событий, на которые приложение должно реагировать соответствующим образом. Например, окно даже самого простого приложения может быть изменено по величине или быть перекрыто окном другого приложения, и на все подобные события необходима правильная реакция.

В назначение `QApplication` входит:

- установка стиля приложения. Таким образом можно устанавливать стиль `Motif`, `Windows`, а также многие другие *виды и поведения* (Look & Feel) приложения, включая и свои собственные (см. *главу 26*);
- получение указателя на объект *рабочего стола* (desktop);
- получение доступа к буферу обмена (см. *главу 30*);
- управление глобальными манипуляциями с мышью (например, установка интервала двойного щелчка кнопкой мыши) и регистрация движения мыши в пределах и за пределами окна приложения;
- выдача предупреждающего звукового сигнала (см. *главу 27*);
- обеспечение правильного завершения работающего приложения при завершении работы операционной системы (см. *главу 29*);
- инициализация необходимых настроек приложения, например палитры для расцветки элементов управления (см. *главу 13*).

Бывает так, что приложение может быть неактивным, а есть необходимость обратить на себя внимание пользователя. Для этой цели класс `QApplication` предоставляет статический метод `alert()`. Его вызов приведет к подскакиванию значка приложения на док-панели в ОС Mac OS X, а в ОС Windows на панели задач произойдет его пульсация, как это показано на рис. 1.3 и 1.4.

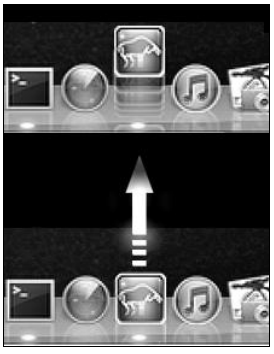

Рис. 1.3. Подскакивание значка приложения на док-панели в ОС Mac OS X

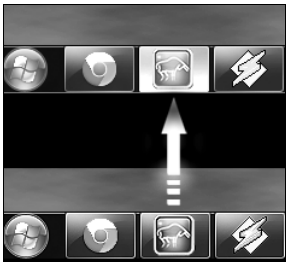

Рис. 1.4. Пульсация значка приложения на панели задач в Windows 7

Модуль QtNetwork

Сетевой модуль предоставляет инструментарий для программирования TCP- и UDP-сокетов (классы `QTcpSocket` и `QUdpSocket`), а также для реализации программ-клиентов, использующих HTTP- и FTP-протоколы (классы `QHttp` и `QFtp`). Этот модуль описывается в *главе 40*.

Модуль QtXml

Этот модуль предназначен для работы с XML посредством SAX2- и DOM-интерфейсов, которые определяют классы Qt (см. *главу 41*).

Модуль QtSql

Этот модуль предназначен для работы с базами данных. В него входят классы, предоставляющие возможность для манипулирования значениями баз данных (см. *главу 42*).

Модуль QtOpenGL

Модуль QtOpenGL делает возможным использование OpenGL в Qt-программах для двух- и трехмерной графики. Основным классом этого модуля является `QGLWidget`, который унаследован от `QWidget` (см. *главу 23*).

Модуль QtWebKit

Этот модуль позволяет очень просто интегрировать в приложение содержание из Web. Возможно также расширять элементы Web своими собственными виджетами. Более подробно с этим модулем можно ознакомиться в *главе 47*.

Модуль QtSvg

Модуль поддержки графического векторного формата SVG, базирующегося на XML. Этот формат предоставляет возможность не только для вывода одного кадра векторного изображения, но может быть использован и для векторной анимации (см. *главу 22*).

Модуль Qt3Support

Если сравнивать третью версию Qt с четвертой, то можно заметить, что библиотека подверглась очень большим изменениям. Некоторые классы были удалены из новой версии библиотеки, а другие претерпели такие изменения, которые сделали их несовместимыми со старым программным кодом. Основное назначение этого модуля состоит в облегчении адаптации старых Qt-программ к Qt4. Разумеется, этот модуль не нужно использовать в ваших новых программах, написанных на базе Qt4. Некоторые рекомендации по адаптации старых Qt-программ к Qt4 можно найти в *главе 49*.

Резюме

Библиотека Qt не является монолитной библиотекой, она разбита на отдельные модули: QtCore, QtGui, QtNetwork, QtOpenGL, QtSql, QtXml и QtSvg. Каждый модуль имеет свое назначение, например программирование интерфейса пользователя, графики, баз данных и др. Классы модулей предоставляют разработчику механизмы, расширяющие возможности программистов и, вместе с тем, упрощающие создание приложений. Вершиной модульной иерархии является модуль QtCore, который позволяет реализовывать приложения без графического интерфейса пользователя (консольные приложения). Объект класса `QCoreApplication` должен быть создан в приложении только один раз.

Для реализации приложений с графическим интерфейсом пользователя необходим модуль QtGui. Класс `QApplication` является стержнем любого Qt-приложения с графическим интерфейсом. Объект этого класса также не должен создаваться в приложении больше одного раза. Данный объект используется для получения событий клавиатуры, мыши и др.

ГЛАВА 2

Философия объектной модели

Те, кого первое знакомство с квантовой теорией не повергло в шок, скорее всего, вовсе ее не поняли.
Макс Борн

Объектная модель Qt подразумевает, что все построено на объектах. Фактически, класс `QObject` — основной, базовый класс. Подавляющее большинство классов Qt являются его наследниками. Классы, имеющие сигналы и слоты, должны быть унаследованы от этого класса.

> ***Примечание***
>
> При множественном наследовании важно помнить, что при определении класса имя класса `QObject` (или унаследованного от него) должно стоять первым, чтобы MOC (Meta Object Compiler, метаобъектный компилятор) мог правильно распознать его. Другой порядок приведет к ошибке при компиляции. В листинге 2.1 приведен правильный порядок для множественного наследования.

Листинг 2.1. Порядок наследования

```
class MyClass : public QObject, public AnotherClass {
...
};
```

> ***Примечание***
>
> При множественном наследовании также важно учитывать, что от класса `QObject` должен быть унаследован только один из базовых классов. Другими словами, нельзя производить наследование сразу от нескольких классов, наследующих класс `QObject`.

Класс `QObject` содержит в себе поддержку:

- сигналов и слотов (signal/slot);
- таймера;
- механизма объединения объектов в иерархии;
- событий и механизма их фильтрации;
- организации объектных иерархий;
- метаобъектной информации;

- приведения типов;
- свойств.

Сигналы и слоты — это средства, позволяющие эффективно производить обмен информацией о событиях, вырабатываемых объектами. О них мы подробно поговорим позже в этой главе.

Поддержка таймера дает возможность каждому из классов, унаследованных от класса `QObject`, не создавать дополнительно объект таймера. Тем самым экономится время на разработку. Подробнее о таймерах говорится в *главе 38*.

Механизм объединения объектов в иерархические структуры позволяет резко сократить временные затраты при разработке приложений, не заботясь об освобождении памяти создаваемых объектов, т. к. объекты-предки сами отвечают за уничтожение своих потомков.

Механизм фильтрации событий позволяет осуществить их перехват. Фильтр событий может быть установлен в любом классе, унаследованном от класса `QObject`, благодаря чему можно изменять реакцию объектов на происходящие события без изменения исходного кода класса (см. *главу 15*).

Метаобъектная информация включает в себя информацию о наследовании классов, что позволяет определять, являются ли классы непосредственными наследниками, а также узнать имя класса.

Для приведения типов Qt предоставляет шаблонную функцию `qobject_cast<T>()`, базирующуюся на метаинформации, создаваемой метаобъектным компилятором MOC (см. *главу 3*), для классов, унаследованных от `QObject`.

Свойства — это поля, для которых обязательно должны существовать методы чтения. С их помощью можно получать доступ к атрибутам объектов извне, например из языка сценариев Qt Script (см. *часть VII*). Свойства также широко используются в визуальной среде разработки пользовательского интерфейса Qt Designer (см. *главу 45*). Этот механизм реализован в Qt при помощи директив препроцессора. Задается свойство при помощи макроса `Q_PROPERTY`. Определение свойства в общем виде выглядит следующим образом:

```
Q_PROPERTY(type name
           READ getFunction
           [WRITE setFunction]
           [RESET resetFunction]
           [DESIGNABLE bool]
           [SCRIPTABLE bool]
           [STORED bool]
          )
```

Первыми задаются тип и имя свойства, вторым — имя метода чтения (`READ`). Определение остальных параметров не является обязательным. Третий параметр задает имя метода записи (`WRITE`), четвертый — имя метода сброса значения (`RESET`), пятый (`DESIGNABLE`) является логическим (булевым) значением, говорящим о том, должно ли свойство появляться в инспекторе свойств Qt Designer. Шестой параметр (`SCRIPTABLE`) — также логическое значение, которое управляет тем, будет ли свойство доступно для языка сценариев Qt Script. Последний, седьмой параметр (`STORED`) управляет сериализацией, т. е. тем, будет ли свойство запоминаться во время сохранения объекта.

Итак, теперь, когда вы познакомились с понятием "свойство", давайте в качестве простого примера определим в классе свойство для управления режимом только чтения (листинг 2.2).

Листинг 2.2. Определение свойства для управления режимом только чтения

```
class MyClass : public QObject {
Q_OBJECT
Q_PROPERTY(bool readOnly READ isReadOnly WRITE setReadOnly)

private:
    bool m_bReadOnly;

public:
    MyClass(QObject* pobj = 0) : QObject(pobj)
                               , m_bReadOnly(false)
    {
    }

public:
    void setReadOnly(bool bReadOnly)
    {
        m_bReadOnly = bReadOnly;
    }

    bool isReadOnly() const
    {
        return m_bReadOnly;
    }
}
```

Класс `MyClass`, показанный в листинге 2.2, наследуется от класса `QObject`. Мы определяем атрибут `m_bReadOnly`, в котором будут запоминаться значения состояния. Этот атрибут инициализируется в конструкторе значением `false`. Для получения и изменения значения атрибута в классе `MyClass` определены методы `isReadOnly()` и `setReadOnly()`. Эти методы регистрируются в макросе `Q_PROPERTY`. Метод `isReadOnly()` служит для получения значения, поэтому указывается в секции `READ`, а метод `setReadOnly()` — для изменения значения, поэтому пишется в секции `WRITE`.

Из программы мы можем изменить значение нашего свойства следующим образом:

```
pobj->setProperty("readOnly", true);
```

А так можно получить текущее значение:

```
bool bReadOnly = pobj->property("readOnly").toBool();
```

Механизм сигналов и слотов

Элементы графического интерфейса определенным образом реагируют на действия пользователя и посылают сообщения. Существует несколько вариантов такого решения.

Старая концепция *функций обратного вызова* (callback functions), лежащая в основе X Window System, основана на использовании обычных функций, которые должны вызываться в результате действий пользователя. Применение такой концепции значительно

усложняет исходный код программы, делая его менее понятным. Кроме того, отсутствует возможность производить проверку типов возвращаемых значений, потому что во всех случаях возвращается указатель на пустой тип `void`. Например, для того чтобы сопоставить код с кнопкой, необходимо передать в функцию указатель на кнопку. Если пользователь нажимает на кнопку, функция будет вызвана. Сами библиотеки не проверяют, были ли аргументы, переданные в функцию, требуемого типа, а это часто является причиной сбоев. Другой недостаток функций обратного вызова заключается в том, что элементы графического интерфейса пользователя тесно связаны с функциональными частями программы и это, в свою очередь, заметно усложняет разработку классов независимо друг от друга. Одним из ярких представителей этой концепции является библиотека Motif.

Важно помнить, что Motif и Windows API предназначены для процедурного программирования, и с реализацией объектно-ориентированных проектов, наверняка, появятся трудности.

Для программирования в ОС Windows существуют специальные библиотеки классов языка C++, облегчающие программирование для этой операционной системы. Самой популярной библиотекой является Microsoft Foundation Classes (MFC). Ее можно, с большой натяжкой, назвать объектно-ориентированной, т. к. она создавалась людьми, не подозревающими о существовании самых элементарных принципов объектно-ориентированного подхода. Одна из самых фундаментальных заповедей объектно-ориентированного подхода — это *инкапсуляция*, которая запрещает оставлять атрибуты классов незащищенными (ведь тогда объекты могут читать и изменять данные без ведома объекта-владельца), но, несмотря на это, во многих MFC-классах это требование не соблюдено. Сама библиотека MFC является надстройкой, предоставляющей доступ к функциям Windows, реализованным на языке C, что заставляет разработчиков время от времени использовать устаревшие структуры, не вписывающиеся в рамки концепции объектно-ориентированного подхода. Интересно также отметить, что сама Microsoft для реализации широко известной программы Microsoft Word не использует MFC вообще.

При использовании MFC для обеспечения связей сообщения и методов обработки используются специальные макросы — так называемые карты сообщений (листинг 2.3). Они очень сильно загромождают исходный код программы, заметно снижая ее читаемость.

Листинг 2.3. Отрывок программы, реализованной с помощью MFC

```
class CPhotoStylerApp : public CWinApp {
public:
    CPhotoStylerApp();
public:
    virtual BOOL InitInstance();

    afx_msg void OnAppAbout();
    afx_msg void OnFileNew();

    DECLARE_MESSAGE_MAP()
};
BEGIN_MESSAGE_MAP(CPhotoStylerApp, CWinApp)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
    ON_COMMAND(ID_FILE_NEW, OnFileNew)
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
```

```
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

Конструкции, подобные показанной в листинге 2.3, очень неудобны для человеческого восприятия и приводят в замешательство при проведении анализа кода программы. Пусть многие рассказывают об удобстве использования средств для автоматического создания подобного кода, но созданы они были не от хорошей жизни. Так, непродуманность самой библиотеки вынуждает разработчика при внесении незначительных изменений модифицировать код самой программы сразу в нескольких местах. Например, для того чтобы добавить в диалоговое окно текстовое поле, необходимо провести целый ряд операций. Во-первых, нужно создать в классе диалога атрибут, предназначенный для хранения значений, вводимых в текстовом поле. Во-вторых, надо задать идентификатор ресурса текстового поля. В-третьих, поставить идентификатор ресурса и атрибут в методе `DoDataExchange()` в соответствие друг с другом при помощи метода `DDX_Text()`, после чего будет осуществляться обмен данными между текстовым полем и атрибутом. В-четвертых, этим обменом необходимо управлять, передавая в методе `UpdateData()` значения булевого типа `true` или `false`. И лишь с помощью средств автоматического создания кода можно частично избавиться от этой проблемы, заставив выполнить эти изменения за вас и получив взамен другие недостатки, например дополнительное засорение кода программы ненужной информацией и возможное несовпадение созданного кода с утвержденными для проекта требованиями для форматирования и нотации (если не используется венгерская нотация). Я не противник обоснованного применения подобного рода средств, но, по моему мнению, они не должны создаваться как средство устранения изъянов плохого дизайна самой библиотеки.

В данной ситуации часть вины скрыта в самом языке C++. Дело в том, что C++ не создавался как средство для написания пользовательского интерфейса, и поэтому не предоставляет соответствующей поддержки, делающей программирование в этой области более удобным. Например, если бы работа по передаче событий реализовывалась средствами самого языка, тогда отпадала бы необходимость в использовании подобного рода макросов. До настоящего времени не удавалось сделать ничего подобного, именно поэтому библиотека Qt явилась "как гром среди ясного неба". В отличие от большинства других библиотек программирования, Qt расширяет язык C++ дополнительными ключевыми словами.

Проблема расширения языка C++ решена в Qt с помощью специального препроцессора MOC (Meta Object Compiler, метаобъектный компилятор). Он анализирует классы на наличие специального макроса `Q_OBJECT` в их определении и внедряет в отдельный файл всю необходимую дополнительную информацию. Это происходит автоматически, без непосредственного участия разработчика. Подобная операция автоматического создания кода не противоречит привычному процессу программирования на C++, ведь стандартный препроцессор перед компиляцией самой программы тоже создает промежуточный код, содержащий исполненные команды препроцессора. Подобным образом действует и MOC, записывая всю необходимую дополнительную информацию в отдельный файл, содержимое которого может быть проигнорировано. Макрос `Q_OBJECT` должен располагаться сразу на следующей строке после ключевого слова `class` с определением имени класса. Очень важно помнить, что после макроса не должно стоять точки с запятой. Внедрять макрос в определение класса имеет смысл в тех случаях, когда созданный класс использует механизм сигналов и слотов или если ему необходима информация о свойствах.

Механизм сигналов и слотов полностью замещает старую модель функций обратного вызова, он очень гибок и полностью объектно-ориентирован. Сигналы и слоты — это краеугольный концепт программирования с использованием Qt, позволяющий соединить вместе