Макс Шлее

Qt 5.3

ПРОФЕССИОНАЛЬНОЕ ПРОГРАММИРОВАНИЕ НА C++

- Кроссплатформенная реализация приложений для Windows, Mac OS X и Linux
- Программирование графики, мультимедиа, веб-приложений, баз данных, сети, таймера, многопоточности, XML, QML и JavaScript
- 230 завершенных программ

+ Материалы на www.bhv.ru

Наиболее полное руководство

В ПОДЛИННИКЕ®

УДК 004.438C++
ББК 32.973.26-018.1
Ш68

Шлее М.

Ш68 Qt 5.3. Профессиональное программирование на C++. — СПб.: БХВ-Петербург, 2015. — 928 с.: ил. — (В подлиннике)

ISBN 978-5-9775-3346-1

Книга посвящена разработке приложений для Windows, Mac OS X и Linux с использованием библиотеки Qt версии 5.3. Подробно рассмотрены возможности, предоставляемые этой библиотекой, и описаны особенности, выгодно отличающие ее от других библиотек. Описана интегрированная среда разработки Qt Creator и работа с технологией Qt Quick. Книга содержит исчерпывающую информацию о классах Qt 5, и так же даны практические рекомендации их применения, проиллюстрированные на большом количестве подробно прокомментированных примеров. Проекты примеров из книги размещены на сайте издательства.

Для программистов

УДК 004.438C++
ББК 32.973.26-018.1

Группа подготовки издания:

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

Подписано в печать 05.03.15.
Формат 70×100 1/16. Печать офсетная. Усл. печ. л. 74,82.
Тираж 1500 экз. Заказ №
"БХВ-Петербург", 191036, Санкт-Петербург, Гончарная ул., 20.

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

ISBN 978-5-9775-3346-1

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 такой пример не в состоянии продемонстрировать весь потенциал и возможности самой библиотеки, он дает представление о базовых понятиях. Этот пример позволяет оценить объем и сложность процесса реализации программ, использующих ту или иную библиотеку. Кроме того, на примере 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 <QtWidgets>
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 `QtWidgets`, представляющий собой файл модуля и включающий в себя заголовочные файлы для используемых в нашей программе классов: `QApplication` и `QLabel`. Конечно, мы могли бы обойтись и без

модуля QtWidgets, а непосредственно подключить заголовочные файлы для поддержки классов QApplication и QLabel, но при большем количестве включаемых классов, задействованных в программе, читаемость самой программы заметно бы ухудшилась. Кроме того, это ускоряет саму работу с кодом и, благодаря механизму предварительной компиляции заголовочных файлов (Precompiled Headers), не должно отразиться на скорости компиляции самой программы — конечно в том случае, если ваш компилятор ее поддерживает.

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, из 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 расходовало 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, являющемся базовым для QGuiApplication, от которого унаследован класс 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

У программистов, начинающих изучение классов новой библиотеки, из-за большого объема информации, которую надо усвоить, зачастую создается ощущение перенасыщения. Но иерархия классов Qt имеет четкую внутреннюю структуру, которую важно сразу понять, чтобы потом уметь хорошо и интуитивно в этой библиотеке ориентироваться.

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 нашего примера из листинга 1.1 — это три модуля: QtCore, QtGui и QtWidgets, они присутствуют во всех программах с графическим интерфейсом и поэтому определены в программе создания make-файлов (см. *главу 3*) по умолчанию. Для использования других модулей в своих про-

Таблица 1.1. *Некоторые модули Qt*

Библиотека	Обозначение в проектном файле	Назначение
QtCore	core	Основополагающий модуль, состоящий из классов, не связанных с графическим интерфейсом
QtGui	gui	Модуль базовых классов для программирования графического интерфейса
QtWidgets	widgets	Модуль, дополняющий QtGui «строительным материалом» для графического интерфейса в виде виджетов на C++
QtQuick	quick	Модуль, содержащий описательный фреймворк для быстрого создания графического интерфейса
QtQML	qml	Модуль, содержащий движок для языка QML и JavaScript
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
QtXmlPatterns	xmlpatterns	Модуль поддержки XPath, XQuery, XSLT и XmlShemaValidator
QtScript	script	Модуль поддержки языка сценариев
QtScriptTools	scripttools	Модуль дополнительных возможностей поддержки языка сценария. В настоящее время предоставляет отладчик
QtMultimedia	multimedia	Модуль мультимедиа
QtMultimediaWidgets	multimediawidgets	Модуль с виджетами для QtMultimedia
QtWebKit	webkit	Модуль для создания Web-приложений
QtWebKitWidgets	webkitwidgets	Модуль с виджетами для QtWebKit
QPrintSupport	printsupport	Модуль для работы с принтером
QtTest	test	Модуль, содержащий классы для тестирования кода

ектах необходимо перечислить их в проектном файле (см. *главу 3*). Например, чтобы добавить модули, нужно написать:

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

А чтобы исключить модуль из проекта:

```
QT -= gui
```

Наиболее значимый из перечисленных в табл. 1.1 модулей — это QtCore, так как он является базовым для всех остальных модулей. Далее идут модули, которые непосредственно зависят от QtCore, это: QtNetwork, QtGui, QtSql и QtXml. И, наконец, модули, зависящие от только что упомянутых модулей: QtOpenGL и QtSvg.

Для каждого модуля Qt предоставляет отдельный заголовочный файл, содержащий заголовочные файлы всех классов этого модуля. Название такого заголовочного файла соответствует названию самого модуля. Например, для включения модуля `QtWidgets` нужно добавить в программу строку, как мы это уже сделали в листинге 1.1:

```
#include <QtWidgets>
```

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`. При этом он является базовым для приложений и не содержит классов, относящихся к интерфейсу пользователя. Если вы собираетесь реализовать консольное приложение, то, вполне возможно, можете ограничиться одним этим модулем. В модуль `QtCore` входят более 200 классов, вот некоторые из них:

- контейнерные классы: `QList`, `QVector`, `QMap`, `QVariant`, `QString` и т. д. (см. *главу 4*);
- классы для ввода и вывода: `QIODevice`, `QTextStream`, `QFile` (см. *главу 36*);
- классы процесса `QProcess` и для программирования многопоточности: `QThread`, `QWaitCondition`, `QMutex` (см. *главу 38*);
- классы для работы с таймером: `QBasicTimer` и `QTimer` (см. *главу 37*);
- классы для работы с датой и временем: `QDate` и `QTime` (см. *главу 37*);
- 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 28*);
- класс приложения `QCoreApplication`, из объекта которого, если требуется, можно запустить цикл событий;
- классы поддержки анимации: `QAbstractAnimation`, `QVariantAnimation` и т. д. (см. *главу 22*);
- классы для машины состояний: `QStateMachine`, `QState` и т. д. (см. *главу 22*);
- классы моделей интервью: `QAbstractItemModel`, `QStringListModel`, `QAbstractProxyModel` (см. *главу 12*).

Модуль содержит так же механизмы поддержки файлов ресурсов (см. *главу 3*).

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, 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 предоставляет классы интеграции с оконной системой, с OpenGL и OpenGL ES. Он содержит класс `QWindow`, который является элементарной областью с возможностью получения событий пользовательского ввода, изменения фокуса и размеров, а так же позволяющей производить графические операции и рисование на своей поверхности.

Класс приложения этого модуля `QGuiApplication`. Этот класс содержит механизм цикла событий и обладает так же возможностями:

- получения доступа к буферу обмена (см. *главу 29*);
- инициализации необходимых настроек приложения — например, палитры для расцветки элементов управления (см. *главу 13*);
- управления формой курсора мыши.

Модуль *QtWidgets*

Этот модуль содержит в себе классы виджетов, представляющие собой «строительный материал» 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 он 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* ему 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 31* и *34*);
- классы окон сообщений и диалоговых окон: `QMessageBox`, `QDialog` (см. *главу 32*);
- классы для рисования: `QPainter`, `QBrush`, `QPen`, `QColor` (см. *главу 18*);
- классы для растровых изображений: `QImage`, `QPixmap` (см. *главу 19*);

- классы стилей (см. *главу 26*) — 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 подробнее последний класс — класс `QApplication`, с которым мы встречались в самом первом примере. Все, что было сказано ранее о классе `QCoreApplication`, относится также и к этому классу, поскольку он является его наследником. Объект класса `QApplication` представляет собой центральный контрольный пункт Qt-приложений, имеющих пользовательский интерфейс на базе виджетов. Этот 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.

Класс `QApplication` напрямую унаследован от `QGuiApplication` и дополняет его следующими возможностями:

- установка стиля приложения. Таким способом можно устанавливать *виды и поведения* (Look & Feel) приложения, включая и свои собственные (см. *главу 26*);
- получение указателя на объект *рабочего стола* (desktop);
- управление глобальными манипуляциями с мышью (например, установка интервала двойного щелчка кнопкой мыши) и регистрация движения мыши в пределах и за пределами окна приложения;
- обеспечение правильного завершения работающего приложения при завершении работы операционной системы (см. *главу 28*).

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 (рис. 1.2) и его пульсации на панели задач в ОС Windows (рис. 1.3).

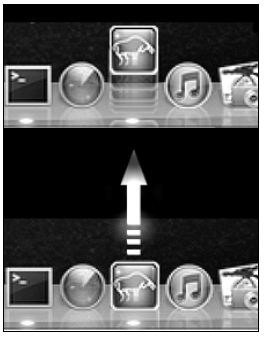

Рис. 1.2. Подскакивание значка приложения на док-панели в ОС Mac OS X

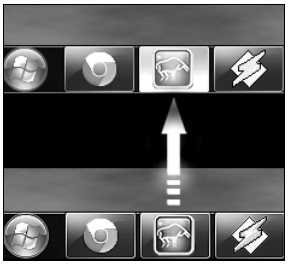

Рис. 1.3. Пульсация значка приложения на панели задач в Windows 7

Модули *QtQuick* и *QtQML*

Это альтернатива виджетам — модули, представляющие собой набор технологий для быстрой разработки графических интерфейсов нового поколения на базе описательного языка QML, языка программирования JavaScript и всех остальных возможностей библиотеки Qt (см. *главу 53*).

Модуль *QtNetwork*

Сетевой модуль `QtNetwork` предоставляет инструментарий для программирования TCP- и UDP-сокетов (классы `QTcpSocket` и `QUdpSocket`), а также для реализации программ-клиентов, использующих HTTP- и FTP-протоколы (класс `QNetworkAccessManager`). Этот модуль описывается в *главе 39*.

Модули *QtXml* и *QtXmlPatterns*

Модуль `QtXml` предназначен для работы с базовыми возможностями XML посредством SAX2- и DOM-интерфейсов, которые определяют классы Qt (см. *главу 40*). А модуль `QtXmlPatterns` идет дальше и предоставляет поддержку для дополнительных технологий XML — таких как: XPath, XQuery, XSLT и XmlShemaValidator.

Модуль *QtSql*

Этот модуль предназначен для работы с базами данных. В него входят классы, предоставляющие возможность для манипулирования значениями баз данных (см. *главу 41*).

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*).

Модули *QtWebKit* и *QtWebKitWidgets*

Модуль `QtWebKit` позволяет очень просто интегрировать в приложение возможости Web. А модуль `QtWebKiteWidgets` предоставляет готовые к интеграции в приложение элементы в виде виджетов с возможностью также расширять элементы Web своими собственными виджетами. Более подробно с этими модулями можно ознакомиться в *главе 46*.

Модули *QtMultimedia* и *QtMultimediaWidgets*

Модуль `QtMultimedia` обладает всем необходимым для создания приложений с поддержкой мультимедиа. Он поддерживает как низкий уровень, необходимый для более детальной специализированной реализации, так и высокий уровень, делающий возможным проигрывать видео- и звуковые файлы при помощи всего нескольких строк программного кода. Модуль `QtMultimediaWidgets` содержит полезные элементы в виде виджетов, которые позволяют экономить время для реализации. Более подробно с этими модулями можно ознакомиться в *главе 27*.

Модули *QtScript* и *QtScriptTools*

Модуль `QtScript` предоставляет возможности расширения и изменения уже написанных приложений при помощи языка сценариев JavaScript. Модуль `QtScriptTools` обеспечивает средства отладки для программ сценариев. Эти модули подробно описывает *часть VII*.

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*).

Резюме

Библиотека Qt не является монолитной библиотекой, она разбита на отдельные модули: `QtCore`, `QtGui`, `QtWidgets`, `QtQuick`, `QtQML`, `QtScript`, `QtMultimedia`, `QtWebKit`, `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 необходимы модули `QtWidgets` или `QtQuick`. Классы `QGuiApplication` и `QApplication` являются стержнем для Qt-приложений с графическим интерфейсом. Объект одного из этих классов не должен создаваться в приложении больше одного раза.

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 его правильно распознать. 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.

Таймер 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 37*.

Механизм объединения объектов в иерархические структуры позволяет резко сократить временные затраты при разработке приложений, не заботясь об освобождении памяти создаваемых объектов, поскольку 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.

Приведение типов. 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 задействованы в визуальной среде разработки пользовательского интерфейса Qt Designer (см. *главу 44*), механизм которой реализован в Qt при помощи директив препроцессора. Задается свойство использованием 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, 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, то есть тем, будет ли свойство запоминаться во время сохранения объекта.

Итак, теперь, когда вы познакомились с понятием «свойство» (хотя в ближайшее время этот механизм нам и не понадобится), давайте все равно 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 у них наверняка появятся трудности.

Существуют, впрочем, специальные библиотеки классов языка C++, облегчающие программирование для ОС Windows. Одной из самых первых таких библиотек (и до сих пор находящихся в применении у целого ряда индивидуальных разработчиков и компаний) является Microsoft Foundation Classes (MFC). Назвать ее объектно-ориентированной можно лишь с большой натяжкой, поскольку она создавалась людьми, не подозревающими о существовании самых элементарных принципов объектно-ориентированного подхода. Одна из главных 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 применяются 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)
```