Программирование на C++

Тема 4.1 GUI с использованием Qt

Qt («кьют») – это библиотека классов с инструментарием для создания межплатформенных приложений с графическим пользовательским интерфейсом (**Graphics User Interface, GUI**) преимущественно на языке C++.

- **Qt лежит в основе:**
  - среды рабочего стола KDE Unix-подобных ОС;
  - таких приложений, как Skype, VLC, Virtual Box и многих других.
- **Поддерживаются ОС:**
  - MS Windows, Mac OS X;
  - различные дистрибутивы Linux;
  - мобильные ОС Symbian S60 и Maemo.
- существует две не полностью совместимые ветви версий Qt – 3.x и 4.x.

- Qt – это библиотека классов с инструментарием для создания межплатформенных приложений с графическим пользовательским интерфейсом (Graphics User Interface, GUI) преимущественно на языке C++.
- Однако, в различное время были созданы интерфейсы, позволяющие вести разработку с использованием Qt и на других языках программирования, таких как: Python – PyQt, PySide; Ruby – QtRuby; Java – QtJambi; PHP – PHP-Qt и другие.
- Qt лежит в основе популярной среди пользователей Unix-подобных систем среды рабочего стола KDE, а также таких приложений, как Skype, VLC, Virtual Box и многих других.
- Использование API Qt вместо других, специфичных для платформы, программных интерфейсов, позволяет создавать приложения, которые, во многих случаях, без всяческих доработок будет компилироваться и исполняться на любой из ОС поддерживаемых Qt, а в большинстве других случаев требовать лишь незначительной доработки. Среди таких ОС, помимо MeeGo – Windows, Mac OS X, различные дистрибутивы Linux, Solaris, использующие оконную систему X11, Symbian, Windows CE.

Краткая история Qt :

- 1991 - начата разработка Qt Гаавардом Нордом и Айриком Шамбе-Ингом, основавшими компанию Trolltech;
- 1995 – первый релиз Qt. Qt включает в себя набор графических компонент для X11/Unix и Windows.
- 2001 – вышла Qt 3.0, в которой появилась поддержка Mac OS X.
- 2005 - вышла Qt 4.0.
- 2008 - компания Trolltech была приобретена компанией Nokia и переименована сперва в Qt Software, а впоследствии — в Qt Development Frameworks.
- 2009 - вышла Qt 4.5. В фреймворк была добавлена третья опция лицензирования — LGPL, что сделало возможным использование «бесплатной» версии Qt в проектах с закрытым кодом (при выполнении некоторых условий).

- Разработка Qt как графического toolkit (библиотеки графических компонентов) была начата в 1991 году Гаавардом Нордом и Айриком Шамбе-Ингом, основавшими впоследствии компанию Quasar Technologies, затем переименованную в Trolltech.
- Идея разработки кроссплатфоменного toolkit появилась во время работы над графическим приложением для медицинской индустрии, которое должно было работать в ОС Windows и Unix. Буква Q появилась в названии фреймворка, поскольку Гааварду очень нравилось ее начертание в шрифте, использовавшемся в редакторе Emacs. Буква t, за которой скрывается слово «toolkit», была добавлена по аналогии с Xt – X Toolkit, библиотекой для создания виджетов в оконной системе X.
- Несколько лет проект разрабатывался без представления на рынке. Первый релиз Qt был сделан в 1995 году. Он включил в себя набор графических компонент для X11/Unix и Windows. В версии 3.0, вышедшей в 2001 году, появилась также поддержка Mac OS X.
- В различное время фреймворк Qt распространялся под разными лицензиями. Если версия Qt для оконной системы X11 изначально выпускалась как под коммерческой, так и под бесплатной (хотя и не свободной) лицензией с открытым исходным кодом, то первые версии для Windows и Mac OS X существовали лишь в версии для коммерческого использования BSD.
- Особую остроту вопрос лицензирования технологии приобрел с ростом популярности оконной среды KDE среди пользователей Linux в конце 90-ых годов, когда стало очевидно, что одна из важнейших компонент наиболее популярной свободной ОС не является свободным ПО.

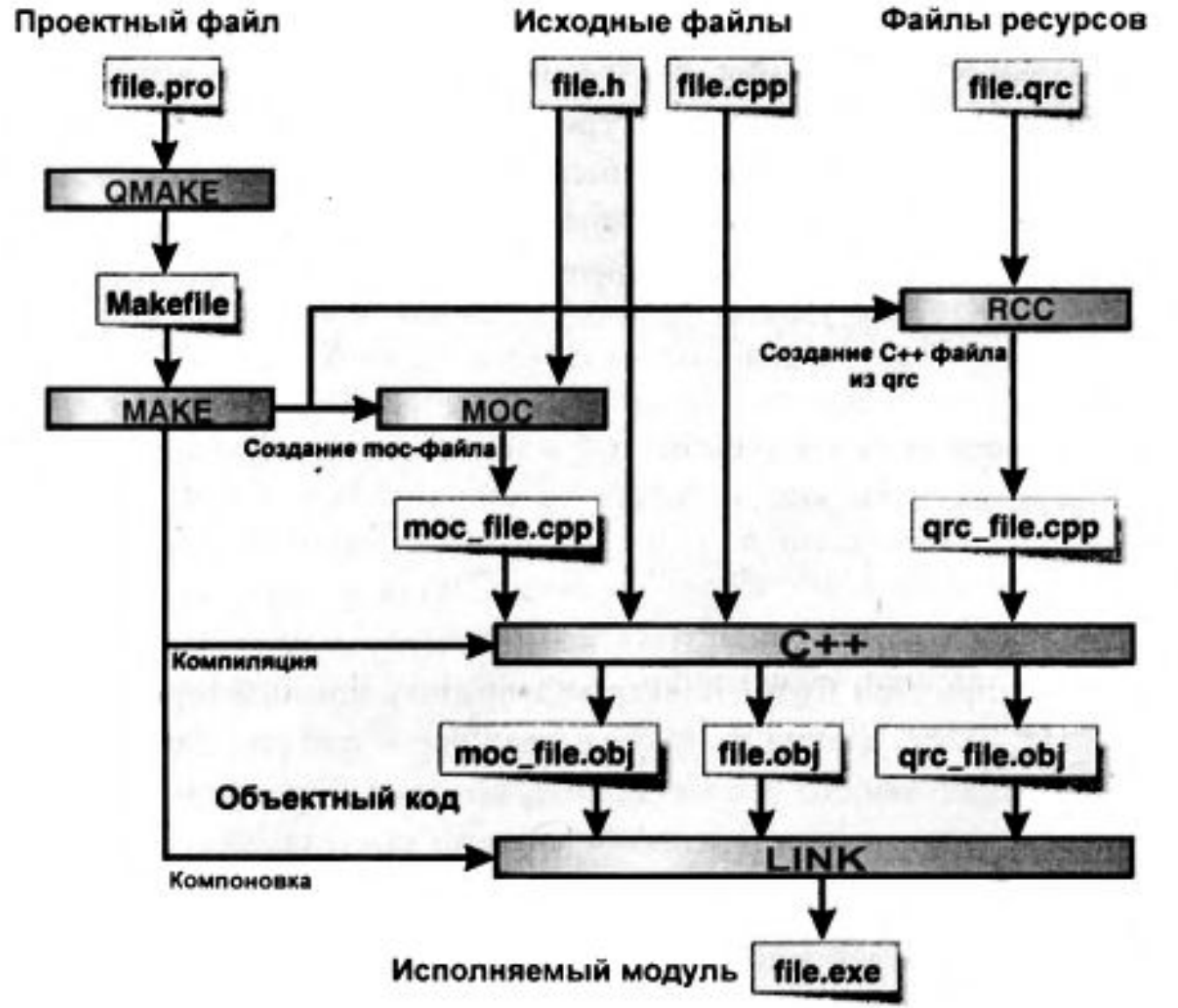
- Проблема лицензирования X11-версии была решена при помощи перехода на свободную лицензию QPL и основания KDE Free Qt Foundation — организации, гарантирующей, что в случае, если разработка свободной версии Qt будет приостановлено, последняя версия будет выпущена под лицензией типа BSD.
- Хотя к 2003 году версии Qt для OS X и для X11 выпускались под свободными лицензиями, версия для Windows по-прежнему выпускалась лишь под коммерческой лицензией. Это привело к тому, что в 2002 группа независимых разработчиков начала работу по портированию X11-версии фреймворка, выпущенной под лицензией GPL, на Windows. Работа эта, впрочем, не была завершена, поскольку в 2005 была выпущена версия фреймворка 4.0, в действие лицензии GPL было распространено на версии для всех поддерживаемых платформ. Добавленное позднее специальное исключение в лицензию, сделало возможным использование GPL-версии Qt в проектах, использующих одну из целого ряда свободных лицензий, таких, как BSD License, Eclipse Public License и других.
- В 2008 году компания Trolltech была приобретена компанией Nokia и переименована сперва в Qt Software, а впоследствии — в Qt Development Frameworks. Вскоре после этого была выпущена версия фреймворка для основной мобильной ОС, использующейся Nokia — Symbian S60. С развитием другой мобильной ОС, разрабатываемой Nokia — Maemo, в Qt была добавлена поддержка и этой платформы.
- В версии Qt 4.5, вышедшей 14 января 2009 г., в фреймворк была добавлена третья опция лицензирования — LGPL, что сделало возможным использование «бесплатной» версии Qt в проектах с закрытым кодом (при выполнении некоторых условий).

Инструменты разработки на Qt (Qt SDK)

- **Qt Creator** – кроссплатформенная IDE для работы с Qt.
- **QtDesigner** – инструмент для визуального дизайна графических интерфейсов.
- **QtAssistant** – система справки.
- **qmake** – система сборки .
- **moc** – метаобъектный компилятор , предварительная система обработки исходного кода.
- **uic** – компилятор графических интерфейсов, который получает на вход xml файл, сгенерированный QtDesigner, и по нему выдает код на C++.
- **rcc** – компилятор ресурсов.

- В пакете Qt SDK поставляется набор инструментов, которые облегчают разработку приложений с использованием фреймворка. Перечислим основные:
- Qt Creator – кроссплатформенная IDE для работы с фреймворком Qt, разработанная Qt Software.
- QtDesigner – инструмент для визуального дизайна графических интерфейсов. В результате работы QtDesigner создается xml файл, описывающий графический интерфейс.
- QtAssistant – система справки.
- qmake – система сборки.
- moc – метаобъектный компилятор , предварительная система обработки исходного кода. Позволяет использовать механизм слотов и сигналов. Утилита moc ищет в заголовочных файлах на C++ описания классов, содержащие макрос Q_OBJECT, и создаѐт дополнительный исходный файл на C++, содержащий реализацию дополнительных методов.
- uic – компилятор графических интерфейсов, который получает на вход xml файл, сгенерированный QtDesigner, и по нему выдает код на C++.
- rcc – компилятор ресурсов.

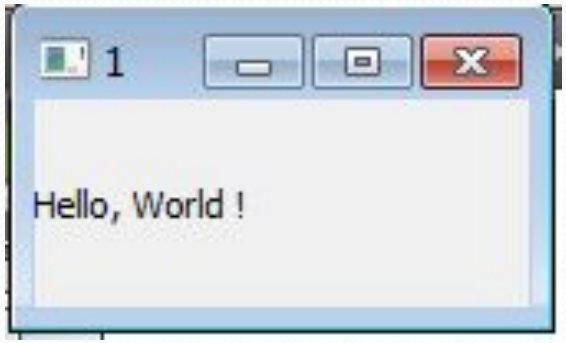
Структура Qt-проекта



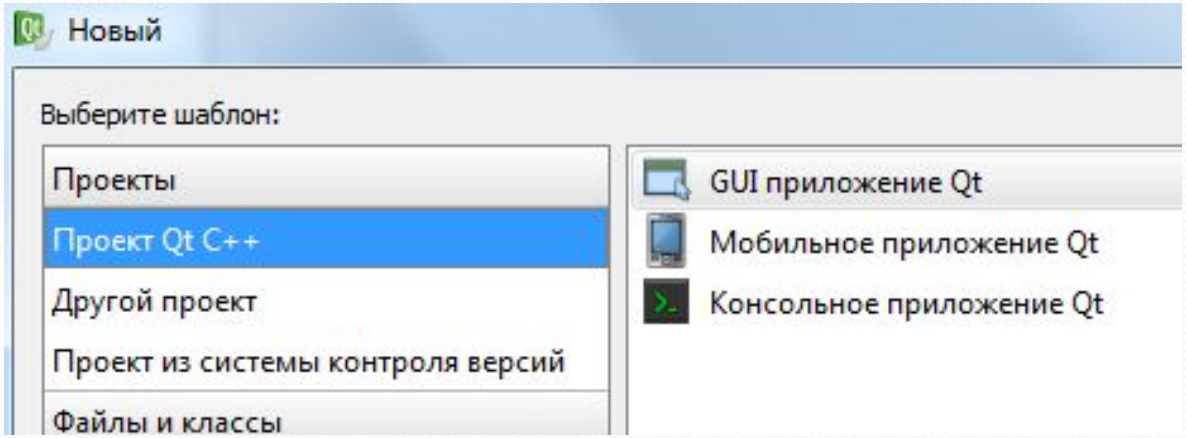
- Структура проекта Qt очень проста, помимо файлов исходного кода на C++ обычно имеется файл проекта с расширением pro. Из этого файла вызовом qmake создается make-файл. Этот make-файл содержит в себе все необходимые инструкции для создания готового исполняемого модуля.
- В make-файле содержится вызов MOC для создания дополнительного кода C++ и необходимых заголовочных файлов. Если проект содержит qrc-файл, то будет также создан файл C++, содержащий данные ресурсов. После этого все исходные файлы компилируются C++ компилятором в файлы объектного кода, которые объединяются компоновщиком link в готовый исполняемый модуль.

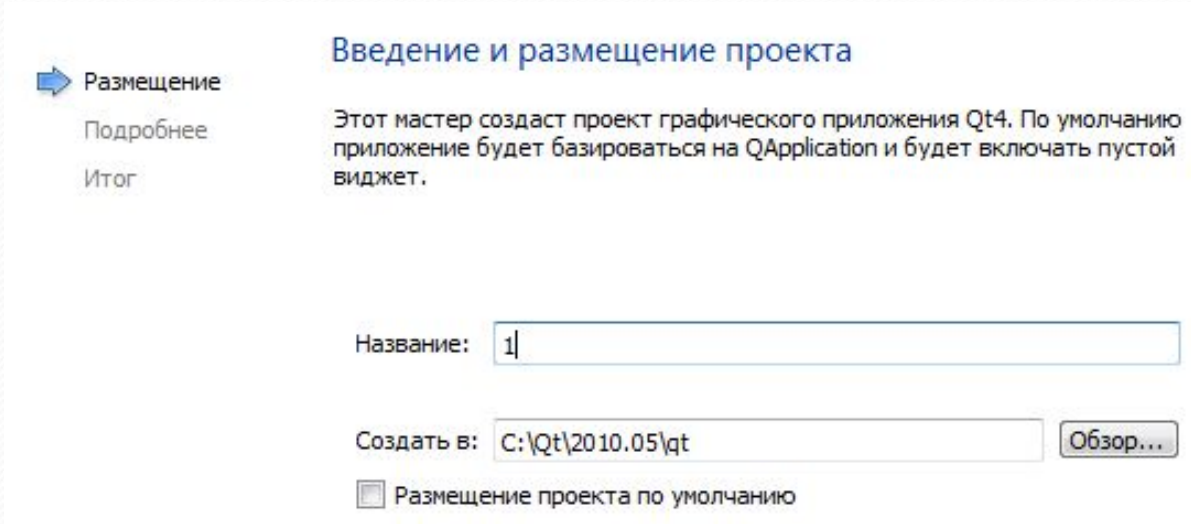
Первая программа с использованием Qt

```
// main.cpp
#include <QtGui>
int main(int argc, char *argv[])
 { QApplication app(argc, argv);
   QLabel lb1("Hello, World !");
   lb1.show();
   return app.exec();
 }
```

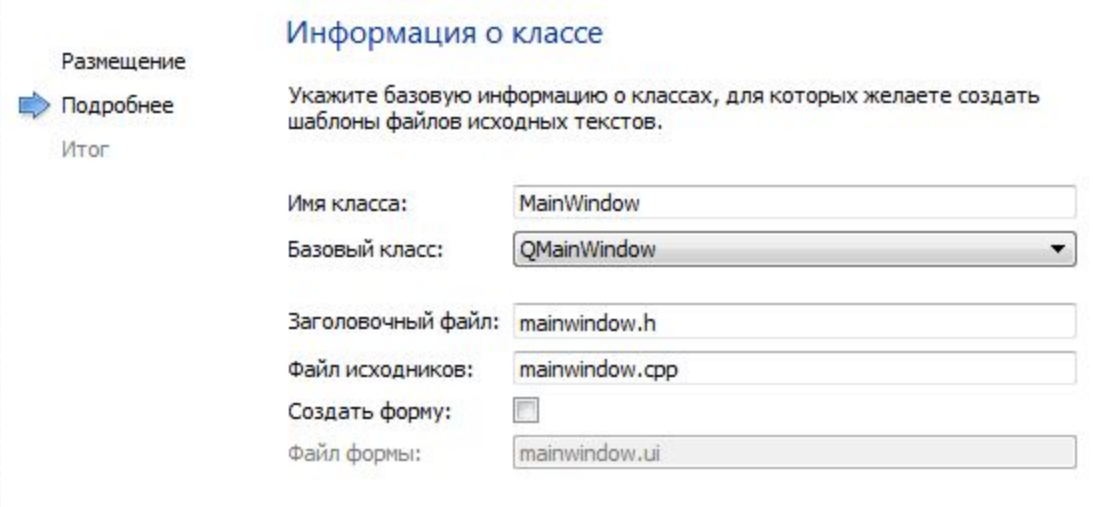


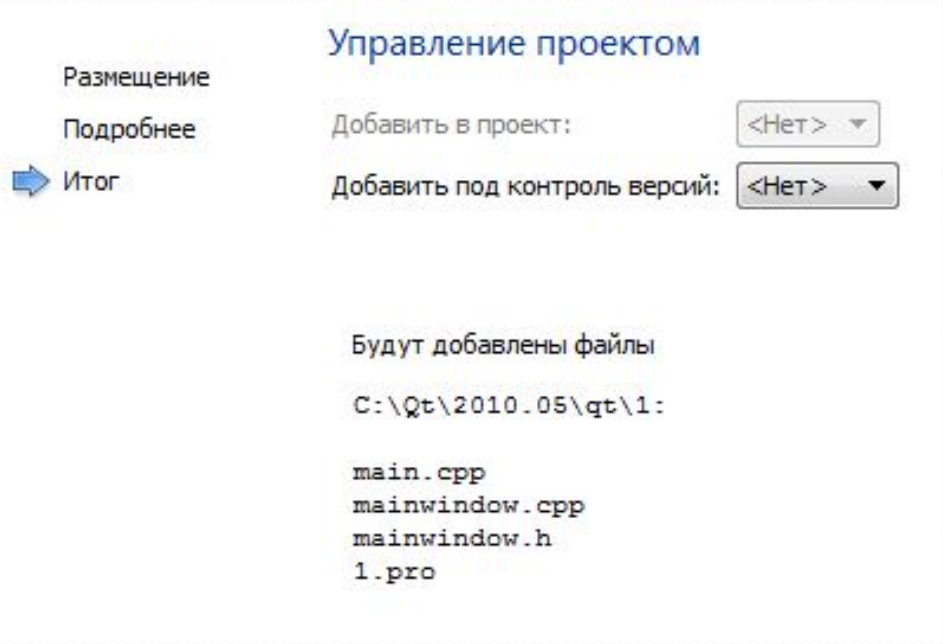
Первая программа на Qt



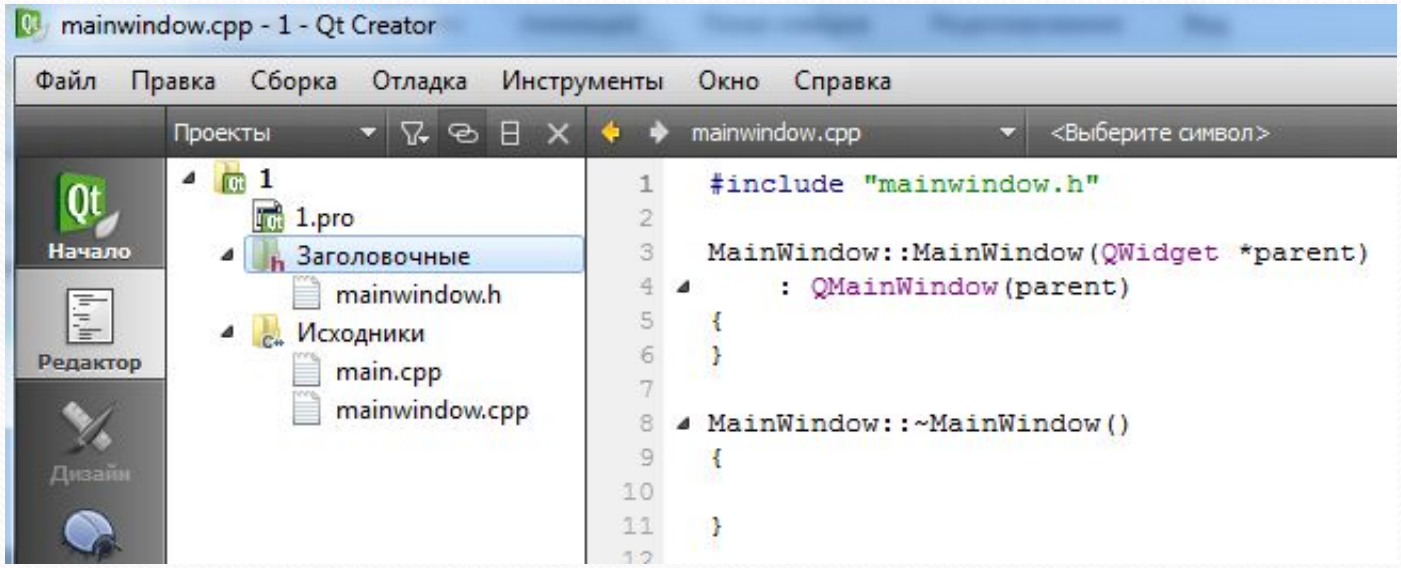


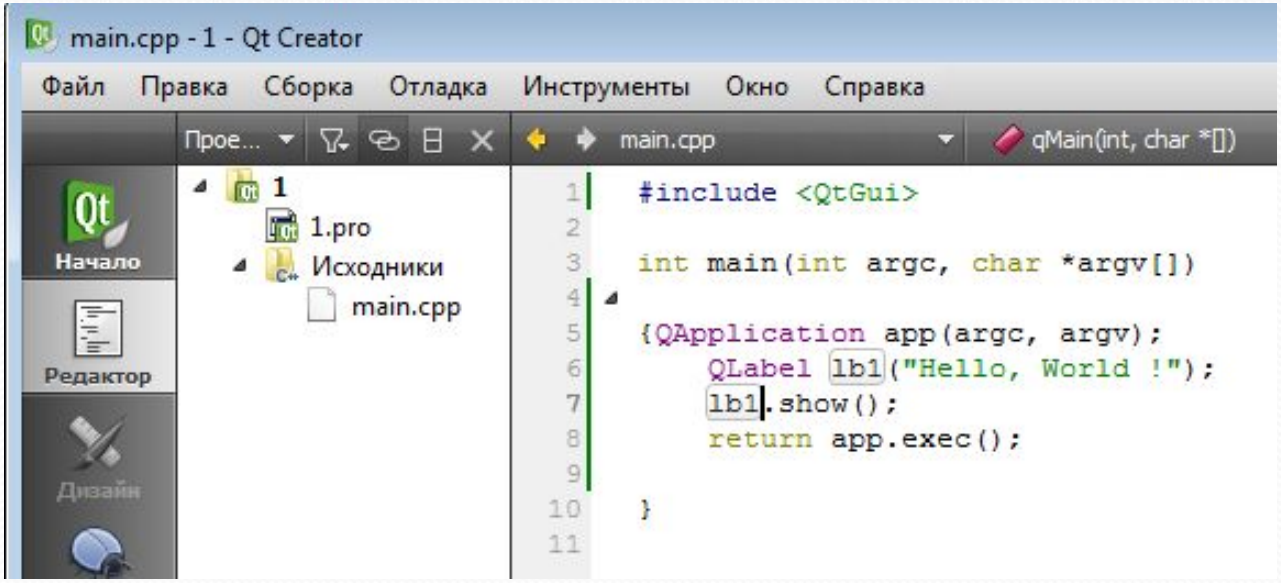
Первая программа на Qt





Первая программа на Qt

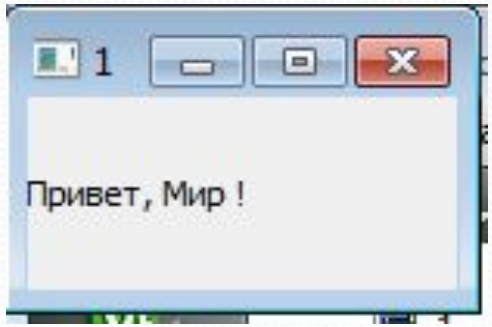




- В первой строке листинга подключается заголовочный файл QtGui, включающий в себя заголовочные файлы для используемых в программе классов: QAppiication и QLabel. Можно обойтись без модуля QtGui, если непосредственно подключить заголовочные файлы для поддержки классов QApplication и QLabel.
- Сначала 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 в созданном приложении имеется сразу несколько независимых друг от друга элементов управления, то при закрытии окна последнего такого элемента управления завершится и само приложение.
- В последней строке программы приложение запускается вызовом метода 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.

Первая программа с использованием Qt

```
// main.cpp
#include <QtGui>
int main(int argc, char *argv[])
{ QApplication app(argc, argv);
  QTextCodec::setCodecForCStrings(QTextCodec::codecFor
  Name("Windows-1251"));
  QLabel lb1("Привет, Мир !");
  lb1.show();
    return app.exec();
 }
```

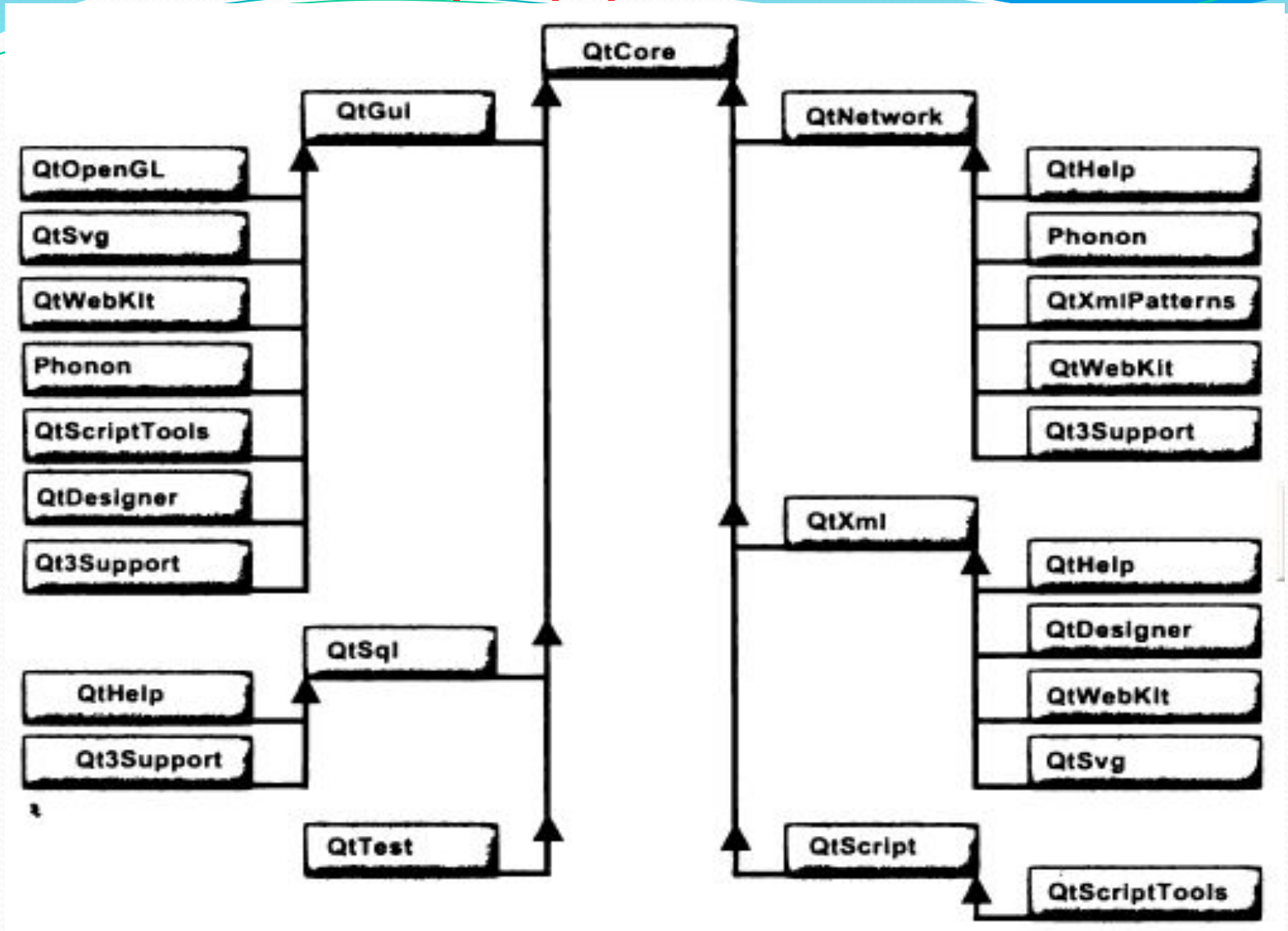


Основные классы Qt

Модуль библиотеки	Назначение
QtCore	Основополагающий модуль, состоящий из классов, не связанных с графическим интерфейсом
QtGui	Модуль для программирования графического интерфейса
QtNetwork	Модуль для программирования сети
QtOpenGL	Модуль для программирования графики OpenGL
QtSql	Модуль для программирования баз данных
QtXml	Модуль поддержки XML
QtScript	Модуль поддержки языка сценариев Модуль поддержки языка сценариев
QtWebKit	Модуль для создания веб-приложений
Phonon	Модуль мультимедиа

- Библиотека Qt — это множество классов (более 500).
- Qt разбита на модули. Любая Qt-программа так или иначе должна использовать хотя бы один из модулей, в большинстве случаев это QtCore и QtGui.
- Модуль Qt Mobility обеспечивает удобную разработку приложений для мобильных платформ, поддерживающих Qt и, в первую очередь, ОС MeeGo.
- Qt Mobility предоставляет интерфейс для функциональностей, специфичных для мобильных устройств, таких как, например:
  - сервисы (GSM-связь, Bluetooth) ;
  - записная книжка;
  - мгновенные сообщения;
  - органайзер;
  - устройства позиционирования;
  - сенсоры (акселерометр, датчик освещѐнности).

Обзор иерархии классов Qt



- Иерархия классов Qt имеет четкую внутреннюю структуру, которую важно понять, чтобы уметь хорошо и интуитивно ориентироваться в этой библиотеке.
- Наиболее значимый из перечисленных в табл. модулей — это QtCore, так как он является базовым для всех остальных модулей (рис.). 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>

Основные классы модуля QtCore

- контейнерные классы **QList, QVector**, **QMap** ;
- классы для ввода и вывода **QIODevice**, **QTextstream**, **QFile** ;
- классы процесса **QProcess** и для программирования многопоточности **QThread**, **QWaitCondition**, **QMutex** ;
- классы для работы с таймером **QBasicTimer** и **QTimer** ;
- классы для работы с датой и временем **QDate** и **QTime** ;
- класс **QObject,** являющийся краеугольным камнем объектной модели Qt ;
- базовый класс событий **QEvent** ;
- класс для сохранения настроек приложения **QSettings** ;
- класс приложения **QCoreAppiication,** из объекта которого, если требуется, можно запустить цикл событий.

- Модуль QtCore является базовым для приложений и не содержит классов, относящихся к интерфейсу пользователя. Для реализации консольных приложений можно ограничиться одним этим модулем. В модуль QtCore входят более 200 классов.
- Остановимся на классе QCoreAppiication. Объект класса приложения QCoreAppiication можно образно сравнить с сосудом, содержащим объекты, подсоединенные к контексту операционной системы.
- Срок жизни объекта класса QCoreAppiication соответствует продолжительности работы всего приложения, и он остается доступным в любой момент работы программы.
- Объект класса QCoreAppiication должен создаваться в приложении только один раз. К задачам этого объекта можно отнести:
  - управление событиями между приложением и операционной системой;
  - передача и предоставление аргументов командной строки.

Основные классы модуля QtGui

- класс **QWidget** — базовый класс для всех элементов управления библиотеки Qt;
- классы для автоматического размещения элементов **QVBoxLayout, QHBoxLayout**;
- классы элементов отображения **QLabel, QLCDNumber**;
- классы кнопок **QPushButton, QCheckBox, QRadioButton**;
- классы элементов установок **QSlider, QScroiiBar**;
- классы элементов ввода **QLineEdit, QSpinBox**;
- классы элементов выбора **QComboBox, qtoolBox**;
- классы меню **QMainWindow И QMenu**;
- классы окон сообщений и диалоговых окон **QMessageBox, QDialog**;
- классы для рисования **QPainter, QBrush, QPen, Qcolor**;
- классы для растровых изображений **Qimage, Qpixmap**;
- классы стилей **QMotif style, Qwindowsstyie** и другие;
- класс приложения **QAppiication,** который предоставляет цикл событий.

- Модуль QtGui содержит в себе классы, необходимые для программирования графического интерфейса пользователя. В этот модуль входят около 300 классов.
- Класс QWidget 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.
- Рассмотрим подробнее класс QApplication. Все, что было сказано ранее о классе QCoreApplication, относится также и к этому классу, так как он является прямым его наследником. 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 ;
  - получение указателя на объект рабочего стола (desktop);
  - получение доступа к буферу обмена;
  - 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 ;
  - обеспечение правильного завершения работающего приложения при завершении работы операционной системы ;
  - инициализация необходимых настроек приложения, например, палитры для расцветки элементов управления .

Объектная модель Qt

- **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.

- 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.
- Сигналы и слоты — это средства, позволяющие эффективно производить обмен информацией о событиях, вырабатываемых объектами.
- Поддержка таймера дает возможность каждому из классов, унаследованных от класса QObject, не создавать дополнительно объект таймера. Тем самым экономится время на разработку.
- Механизм объединения объектов в иерархические структуры позволяет резко сократить временные затраты при разработке приложений, не заботясь об освобождении памяти создаваемых объектов, так как 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 QObject, благодаря чему можно изменять реакцию объектов на происходящие события без изменения исходного кода класса.
- 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 , для классов, унаследованных от QObject.

Свойства

- **Определение свойства в общем виде :**

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

- Свойства — это поля, для которых обязательно должны существовать методы чтения. С их помощью можно получать доступ к атрибутам объектов извне, например из Qt Script. Свойства также широко используются в визуальной среде разработки пользовательского интерфейса Qt Designer. Этот механизм реализован в Qt при помощи директив препроцессора. Задается свойство при помощи макроса Q_PROPERTY.
- В определении свойства 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, то есть тем, будет ли свойство запоминаться во время сохранения объекта.

- **Пример. 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;  }
   }
                 pobj ->setProperty ("readonly", true);
 bool bReadOnly = pobj->property ("readonly").toBool();
```

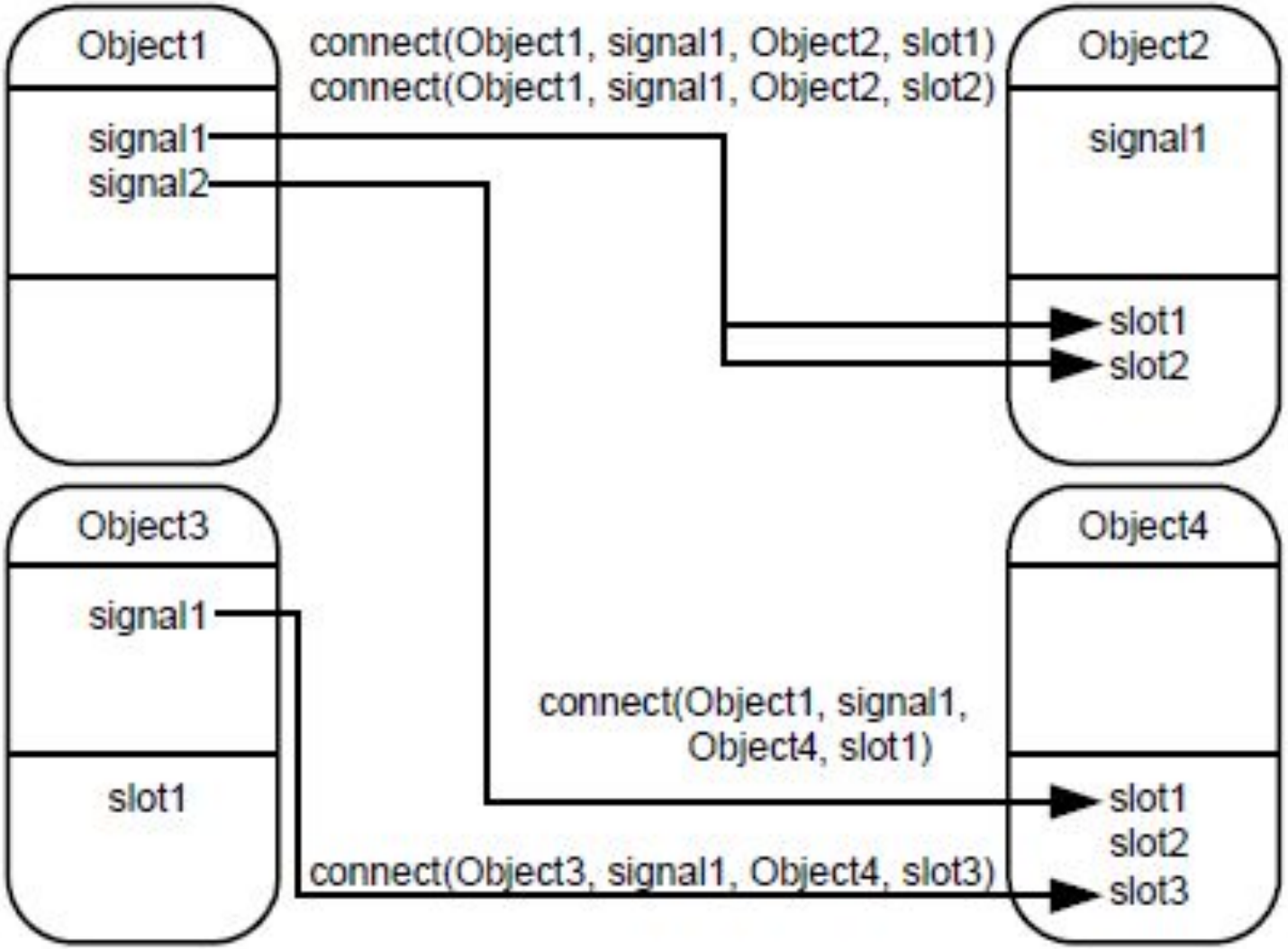
- Класс MyClass, показанный в листинге, наследуется от QObject. В классе определяется атрибут m_bReadOnly, в котором будут запоминаться значения состояния. Этот атрибут инициализируется в конструкторе значением false.
- Для получения и изменения значения атрибута в классе MyClass определены методы isReadOnly() 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



- Преимущества применения механизма сигналов и слотов:
  - каждый класс, унаследованный от QObject, может иметь любое количество сигналов и слотов;
  - сообщения, посылаемые посредством сигналов, могут иметь множество аргументов любого типа;
  - сигнал можно соединять с различным количеством слотов. Высылаемый сигнал поступит ко всем подсоединенным слотам;
  - слот может принимать сообщения от многих сигналов, принадлежащих разным объектам;
  - соединение сигналов и слотов можно производить в любой точке приложения;
  - сигналы и слоты являются механизмами, обеспечивающими связь между
  - объектами.
  - при уничтожении объекта происходит автоматическое разъединение всех сигнально - слотовых связей. Это гарантирует, что сигналы не будут высылаться к несуществующим объектам.
- Недостатки, связанные с применением сигналов и слотов:
  - сигналы и слоты не являются частью языка C++, поэтому требуется запуск дополнительного препроцессора перед компиляцией программы;
  - отсылка сигналов происходит немного медленнее, чем обычный вызов функции, который производится при использовании механизма функций обратного вызова;
  - существует необходимость в наследовании класса QObject;
  - в процессе компиляции не производится никаких проверок: имеется ли сигнал или слот в соответствующих классах или нет; совместимы ли сигнал и слот друг с другом и могут ли они быть соединены вместе. Об ошибке можно будет узнать лишь тогда, когда приложение будет запущено.

Сигналы

- **Пример.** Определение и реализация сигнала

```
class MySignal : public QObject
 { Q_OBJECT
     public:  void sendsignal()  {  emit doIt(); }
     signals:  void doIt();
 };

class MySignal : public QObject
{ Q_OBJECT
     public: void sendSignal()
                { emit sendString("Information"); }
     signals:
         void sendString(const QStringb);
};
```

- Сигналы определяются в классе, как и обычные методы, только без реализации. Всю дальнейшую заботу о реализации кода для этих методов берет на себя МОС. Методы сигналов не должны возвращать каких-либо значений, и поэтому перед именем метода всегда должно стоять void.
- Сигнал не обязательно соединять со слотом. Если соединения не произошло, то он просто не будет обрабатываться. Подобное разделение высылающих и получающих объектов исключает возможность того, что один из подсоединенных слотов каким-то образом сможет помешать объекту, выславшему сигналы.
- Библиотека предоставляет большое количество уже готовых сигналов для существующих элементов управления. В основном, для решения поставленных задач хватает этих сигналов, но иногда возникает необходимость реализации новых сигналов в своих классах.
- Не имеет смысла определять сигналы как private, protected или public, так как они играют роль вызывающих методов.
- Выслать сигнал можно при помощи ключевого слова emit. Ввиду того, что сигналы играют роль вызывающих методов, конструкция высылки сигнала emit doit () приведет к обычному вызову метода doit (). Сигналы могут высылаться из классов, которые их содержат.
- Чтобы иметь возможность отослать сигнал программно из объекта этого класса, следует добавить метод sendsignal (), вызов которого заставит объект класса MySignal высылать сигнал doit ().
- Сигналы также имеют возможность высылать информацию, передаваемую в параметре. Например, если возникла необходимость передать в сигнале строку текста,

Слоты

- **Пример. Реализация слота**

```
class MySlot : public QObject
{ Q_OBJECT
      public: MySlot() ;
      public slots:
        void slot() { qDebug() << "I'm a slot"; }
 };
```

- Слоты (slots) — это методы, которые присоединяются к сигналам. По сути, они являются обычными методами. Самое большое их отличие состоит в возможности принимать сигналы. Как и обычные методы, они определяются в классе как public, private или protected. Соответственно, перед каждой группой слотов должно стоять: private slots:, protected slots: или public slots:. Слоты могут быть и виртуальными.
- *Примечание .*По данным фирмы Nokia, соединение сигнала с виртуальным слотом примерно в десять раз медленнее, чем с невиртуальным. Поэтому не стоит делать слоты виртуальными, если нет особой необходимости.
- Есть небольшие ограничения, отличающие обычные методы от слотов. В слотах нельзя использовать параметры по умолчанию, например slotMethod(int n = 8) , или определять слоты как static.
- Классы библиотеки содержат целый ряд уже реализованных слотов. Но определение слотов для своих классов — это частая процедура.

Соединение объектов

- **Метод connect () :**

```
QObject::connect(const QObject* sender,
                 const char* signal,
                 const QObject* receiver,
                 const char* slot,
                 Qt::ConnectionType type =
  Qt::Autoconnection
);
```

- **Пример соединения объектов в программе.**

```
QObject:: connect (pSender, SIGNAL (signalMethod()) ,
  pReceiver, SLOT(slotMethod())
   );
```

- Параметры метода connect:
  - **sender** — указатель на объект, высылающий сигнал;
  - **signal** — это сигнал, с которым осуществляется соединение. Прототип (имя и аргументы) метода сигнала должен быть заключен в специальный Макрос SIGNAL (method ());
  - **receiver** — указатель на объект, который имеет слот для обработки сигнала;
  - **slot** — слот, который вызывается при получении сигнала. Прототип слота должен быть заключен в специальном макросе SLOT (method ());
  - **type** — управляет режимом обработки. Имеется три возможных значения: Qt: :Directconnection — сигнал обрабатывается сразу вызовом соответствующего метода слота, Qt: :Queuedconnection— сигнал преобразуется в событие и ставится в общую очередь для обработки, Qt: Autoconnection — это следующим образом: если высылающий сигнал объект находится в одном потоке с принимающим его объектом, то устанавливается режим Qt: :QueuedConnection, В противном случае— режим Qt: :DirectConnection. Этот режим(Qt: :Autoconnection) определен в методе connection ( ) по умолчанию.
- В случае, если слот содержится в классе, из которого производится соединение, то можно воспользоваться сокращенной формой метода connect(), опустив третий параметр (pReceiver), указывающий на объект-получатель. Другими словами, если в качестве объекта-получателя должен стоять указатель this, его можно просто не указывать.

Соединение объектов

- **Передача сигнала:**

```
MyClass::MyClass() : QObject()
{
connect(pSender, SIGNAL(signalMethod()),
  SIGNAL(mySignal()));
}
```

- Иногда возникают ситуации, когда объект не обрабатывает сигнал, а просто передает его дальше. Для этого необязательно определять слот, который в ответ на получение сигнала (при помощи emit) отсылает свой собственный. Можно просто соединить сигналы друг с другом. Высылаемый сигнал должен содержаться в определении класса.
- Высылку сигналов заблокировать можно на некоторое время, вызвав метод blocksignais() с параметром true. Объект будет "молчать", пока блокировка не будет снята тем же методом blocksignals () с параметром false.
- При помощи метода signalsblocked() можно узнать текущее состояние блокировки сигналов.

Механизм сигналов и слотов

- **Пример:** Приложение , в первом окне которого находится кнопка нажатия, а во втором — виджет надписи. При щелчке на кнопке ADD (Добавить) происходит увеличение отображаемого значения на 1. Как только значение будет равно пяти, произойдет выход из приложения.

```
//main.cpp
 #include <QtGui>
 #include "Counter.h"
 int main(int argc, char *argv[])
 { QApplication app(argc, argv);
   QLabel lb1("0");
   QPushButton cmd("ADD");
   Counter counter;
   lb1.show(); cmd.show();
   QObject::connect(&cmd, SIGNAL(clicked()), &counter,
SLOT(slotInc()));
   QObject::connect(&counter,
SIGNAL(counterChanged(int)),&lb1, SLOT(setNum(int)));
   QObject::connect(&counter, SIGNAL(goodbye()),&app,
SLOT(quit()));
   return app.exec();
 }
```

- В основной программе приложения создается объект надписи **lb1**, кнопка **cmd** и объект счетчика **counter** (описание которого приведено
- в файлах **Counter.h** и **Counter.cpp**). Далее сигнал **clicked()** соединяется со слотом **slotInc()**. При каждом нажатии на кнопку вызывается метод **slotInc()**, увеличивая значение счетчика на 1. Он должен быть в состоянии сообщать о подобных изменениях, чтобы элемент надписи отображал всегда только актуальное значение. Для этого сигнал **counterChanged(int)**, передающий в параметре актуальное значение счетчика, соединяется со слотом **setNum (int)**, способным принимать это значение.
- *Примечание.* При соединении сигналов со слотами, передающими значения, важно следить за совпадением их типов. Например, сигнал, передающий в параметре значение int, не должен соединяться со слотом, принимающим QString.
- Сигнал **goodbye()**, символизирующий конец работы счетчика, соединяется со слотом объекта приложения **quit()**, который осуществляет завершение работы приложения, после нажатия кнопки ADD в пятый раз.
- Приложение состоит из двух окон, и после закрытия последнего окна его работа автоматически завершится.

```
// Counter.h

 #ifndef COUNTER_H
 #define COUNTER_H
 #include <QObject>
 class Counter : public QObject
  { Q_OBJECT
    private: int m_nValue;
    public: Counter();
    public slots: void slotInc();
    signals: void goodbye();
             void counterChanged(int);
  };
  #endif // COUNTER_H
```

```
// Counter.cpp


 #include "Counter.h"
 Counter::Counter() : QObject(), m_nValue(0)
  {}
 void Counter::slotInc()
  { emit counterChanged(++m_nValue);
    if (m_nValue == 5)
        { emit goodbye();}
  }
```

- В листинге метод слота **`slotInc()`** высылает два сигнала:
- **`counterChanged()`** и **`goodbye ()`**. Сигнал goodbye () высылается при значении атрибута **`m_nValue`**, равном 5.
- Если есть возможность соединения объектов, то должна существовать и возможность их разъединения. В Qt, при уничтожении объекта, все связанные с ним соединения уничтожаются автоматически, но в редких случаях может возникнуть необходимость в уничтожении этих соединений "вручную".
- Для этого существует статический метод disconnect(), параметры которого аналогичны параметрам статического метода connect().

Организация объектных иерархий

- **Конструктор класса QObject :**

```
QObject (QObject* pobj = 0);
```

- **Пример создания объектной иерархии:**

```
QObject* pobj1 = new QObject;
QObject* pobj2 = new QObject(pobj1);
QObject* pobj4 = new QObject(pobj2);
QObject* pobj3 = new QObject(pobj1);
pobj2->setObjectName("the first child of pobj1");
pobj3->setObjectName("the second child of pobj1");
pobj4->setObjectName("the first child of pobj2");
```

- Организация объектов в иерархии снимает с разработчика необходимость самому заботиться об освобождении памяти от созданных объектов.
  - В конструктор класса QObject передается указатель на другой объект класса QObject или унаследованного от него класса. Благодаря этому параметру существует возможность создания объектов-иерархий. Он представляет собой указатель на объект-предок. Если в первом параметре передается значение равное нулю или ничего не передается, то это значит, что у создаваемого объекта нет предка, и он будет являться объектом верхнего уровня и находиться на вершине объектной иерархии. Объект-предок задается в конструкторе при создании объекта, но впоследствии его можно в любой момент исполнения программы изменить на другой при помощи метода setParent ( ).
- Созданные объекты по умолчанию не имеют имени. При помощи метода setObjectName() можно присвоить объекту имя. Имя объекта не имеет особого значения, но может быть полезно при отладке программы. Для того чтобы узнать имя объекта, можно вызвать метод objectName( ).
- В первой строке листинга создается объект верхнего уровня (объект без предка). При создании объекта pobj2 в его конструктор передается, в качестве предка, указатель на объект pobj1. Объект pobj3 имеет в качестве предка pobj1, а объект pobj4 имеет предка pobj2.

Контрольные вопросы

1) Понятие и краткая история Qt.
1) Структура Qt-проекта. Понятие и назначение qmake, moc , uic и rcc.
2) Структура простой программы, использующей Qt.
3) Обзор иерархии классов Qt. Основные классы модуля QtCore и QtGui.
4) Характеристика механизма сигналов и слотов объектной модели Qt.
5) Характеристика механизма свойств объектной модели Qt.