Программирование в среде QT

Кроссплатформенный инструментарий разработки Qt появился впервые в 1995 году благодаря своим разработчикам Хаарварду Норду и Айрику Чеймб-Ингу. С самого начала Qt создавался как программный каркас для создания кроссплатформенных программ с графическим интерфейсом или фреймворк (программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта).

Первая версия библиотеки Qt вышла в сентябре 1995. Программы, разработанные с Qt, работали как под управлением операционных систем семейства Microsoft Windows™ так и под управлением Unix-подобных систем.

В декабре 2012 состоялся официальный выпуск Qt5. Эта версия кроссплатформенного средства разработки совместима с Qt4. Перенос кода с Qt4 на Qt5 не требует много усилий. В то же время, Qt5 отличается рядом особенностей, улучшений и большим количеством новых возможностей.

Лицензирование Qt

Qt распространяется по условиям трёх различных лицензий: GNU GPL v3, GNU LGPL v3 и по коммерческой лицензии компании Digia.

Лицензия GPL

Программа открыта, свободно распространяться, исходные тексты программы и все их изменения пребывают в свободном доступе.

Лицензия LGPL

Исходные тексты программы могут быть как открытыми так и закрытыми. С закрытой программой Qt связывается в виде динамических библиотек. Нельзя вставлять и использовать любые исходные тексты Qt в программе. Также любые изменения в исходных текстах Qt должны быть в свободном доступе.

Лицензия Commercial

В случае коммерческой лицензии, кроме возможности закрывать, модифицировать любым образом текст программы, модифицировать или закрывать изменения в коде Qt и произвольно выбирать лицензию и способ распространения программы, предоставляется также поддержка и консультации по использованию Qt.

Программирование в среде QT

Первая программа на Qt

Создание проекта: QT Creator -> Новый проект-> Приложение Qt Widgets

далее выбирать имя проекта и все предлагаемые настройки

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

#include <QtWidgets> -- подключение файла модуля Qt с заголовочными файлами для QApplication и QLabel. Программа создаёт объект класса QApplication - контроль и управление приложением. В конструктор этого класса необходимо передать два аргумента. 1-й представляет собой информацию о количестве аргументов в командной строке, а 2-й – это указатель на массив символьных строк, содержащих аргументы, по одному в строке. Программа Qt с графическим интерфейсом создаёт только один объект этого класса.

Затем создается объект класса Qlabel, при этом 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 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.

Программирование в среде QT

Библиотека Qt – это множество классов (более 500). Qt не является единым целым, она разбита на модули. Иерархия классов Qt имеет четкую внутреннюю структуру.
Любая Qt-программа с графическим интерфейсом так или иначе должна использовать хотя бы один трёх модулей: QtCore, QtGui или QtWidgets.
Для каждого модуля Qt предоставляет отдельный заголовочный файл, содержащий заголовочные файлы всех классов этого модуля.

Состав основных модулей Qt5

Qt Core – основной модуль, который содержит все базовые средства Qt. На его основе построены все другие модули. Каждая программа созданная с использованием Qt, использует этот модуль;
Qt Network – модуль для работы с сетевыми средствами;
Qt Gui – модуль поддержки графического вывода на экран. Qt5 виджеты вынесены в отдельный модуль;
Qt Widgets – модуль, который содержит набор виджетов для создания графического интерфейса пользователя;
Qt WebKit – средства работы с Web;
Qt WebKit Widgets – виджеты для работы с Web;
Qt Multimedia – средства работы с мультимедийными устройствами и файлами;
Qt Multimedia Widgets – виджеты для работы с мультимедийными устройствами и файлами;
Qt Sql – средства работы с базами данных;
Qt Qml – поддержка декларативной языка QML для разработки динамических визуальных интерфейсов;
Qt Quick – поддержка создания динамических визуальных интерфейсов;
Qt Quick Controls – использование технологии QtQuick для создания традиционного для рабочих столов графического интерфейса;
Qt Quick Layouts – компоновка для элементов QtQuick;
А также: QtXml, QtXmlPatterns, QtScript, QtScriptTools, QtWebKit, QtWebKitWidgets, QtPrintSupport и другие модули.

Программирование в среде QT

Модули QtCore

QtCore – базовый модуль Qt для приложений и не содержит классов, относящихся к интерфейсу пользователя. Для консольных приложений вполне можно 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 книги Шлее М. Qt 5.3. Профессиональное программирование на C++. - СПб.: БХВ-Петербург, 2015. - 928 с: ил.);**
- **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 основой 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: QAbstractltemModel, QStringListModel, QAbstractProxyModel (см. главу 12).**

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

Программирование в среде QT

Модули QCoreApplication и QApplication

Класс QCoreApplication является компонентой модуля QtCore и предоставляет обработку сообщений для консольного приложения Qt.
Этот класс используется для реализации цикла обработки сообщений в приложениях не предоставляющих GUI (graphical user interface). У каждого не-GUI приложения, использующего Qt, должен быть один объект QCoreApplication.
Объект класса приложения QCoreApplication выполняет следующие задачи:

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

Срок жизни объекта класса QCoreApplication соответствует продолжительности работы всего приложения, и он остается доступным в любой момент работы программы. В приложении может создаваться только один объект класса QCoreApplication.

В GUI-приложениях используется класс QApplication, который управляет главным потоком и основными настройками приложения GUI.
Класс содержит главный цикл обработки сообщений, где обрабатываются и пересылаются все сообщения посланные оконной системой и другими ресурсами.
Также в нём реализованы: инициализация, завершение приложения, управление сессией и возможности расширения системы и приложения.
Для любого приложения с GUI использующего Qt есть ровно один объект QApplication, независимо от того, имеет ли приложение 0, 1, 2 или больше окон в любой момент время.

Программирование в среде QT

Модуль QtGui

Модуль QtGui дополняет QtCore функциональностью GUI и предоставляет классы интеграции с оконной системой, с OpenGL и OpenGL ES. .

Для включения определений классов обоих этих модулей, используется директиву препроцессора: `#include <QtGui>`

Класс приложения этого модуля QGuiApplication.

Этот класс содержит механизм цикла событий и обладает так же возможностями:

- **получение доступа к буферу обмена (см. главу 29);**
- **инициализация необходимых настроек приложения – например, палитры для расцветки элементов управления (см. главу 13);**
- **управление формой курсора мыши.**

Программирование в среде QT

Модуль QWidgets

Модуль содержит в себе классы виджетов – инструментов для программирования графического интерфейса пользователя. В модуле 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 (см. главу 5);**
- **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: Qlmage, QPixmap (см. главу 19);**
- **классы стилей (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.**

Программирование в среде QT

Модуль QWidgets. Класс QApplication

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).

В случае, когда приложение неактивно, но надо обратить на себя внимание пользователя, класс QApplication предоставляет статический метод alert(). Его вызов приведет к подскакиванию значка(иконки) приложения на док-панели в ОС Mac OS X и пульсации иконки на панели задач в ОС Windows.

Программирование в среде QT

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) (см.главу 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 Sql

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).

Программирование в среде QT

Модули QtWebKit и QtWebKitWidgets

Модуль QtWebKit позволяет очень просто интегрировать в приложение возможости Web.
Модуль QtwebKitewidgets предоставляет готовые к интеграции в приложение элементы в виде виджетов с возможностью также расширять элементы Web своими собственными виджетами. (см. главу 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 (см.главу 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 (см. часть VIII).

Программирование в среде QT

Пространство имён Qt

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

```
using namespace Qt;
```

Объектная модель Qt

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.

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.

Программирование в среде QT

Объектная модель Qt

Сигналы и слоты – это средства, позволяющие эффективно производить обмен информацией о событиях, вырабатываемых объектами.

Таймер каждый из классов, унаследованных от класса Qobject может использовать объект таймера головного класса. 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, 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.

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 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.

Программирование в среде QT

Объектная модель Qt. Механизм сигналов и слотов

Элементы графического интерфейса определенным образом реагируют на действия пользователя и посылают сообщения.

Расширение языка C++ дополнительными ключевыми словами для работы с сигналами и слотами решена в Qt с помощью специального препроцессора MOC (Meta Object Compiler, метаобъектный компилятор).

MOC анализирует классы на наличие специального макроса Q_OBJECT и внедряет в отдельный файл всю необходимую дополнительную информацию. Это происходит автоматически, без непосредственного участия программиста.

Макрос Q_OBJECT должен располагаться сразу на следующей строке после ключевого слова class с определением имени класса, после макроса не должна стоять точка 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.

Сигналы и слоты – краеугольный концепт программирования с использованием Qt, позволяющий соединить вместе несвязанные друг с другом объекты.

Каждый унаследованный от QObject класс способен отправлять и получать сигналы.

Программирование в среде QT

Объектная модель Qt. Механизм сигналов и слотов

Механизм сигналов и слотов Qt следующие преимущества:

- 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. Отправляемый 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. Более того, эта связь может выполняться между объектами, которые находятся в различных потоках;
- при уничтожении объекта происходит автоматическое разъединение всех сигнально-слотовых связей. Это гарантирует, что сигналы не будут отправляться к несуществующим объектам.

Программирование в среде QT

Объектная модель Qt. Механизм сигналов и слотов

Недостатки механизма сигналов и слотов Qt :

- сигналы и слоты не являются частью языка C++, поэтому требуется запуск дополнительного препроцессора перед компиляцией программы;
- отправка сигналов происходит немного медленнее, чем обычный вызов функции;
- 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 станет известно лишь тогда, когда приложение будет запущено в отладчике или на консоли.

Вся эта информация выводится на консоль, поэтому, для того чтобы увидеть ее в Windows, в проектном файле необходимо в секции config добавить опцию console (для Mac OS X и Linux никаких дополнительных изменений проектного файла не требуется).

Программирование в среде QT

Объектная модель Qt. Сигналы

Сигналы (signals) в Qt – это методы для пересылки сообщений.

Причиной для появления сигнала может быть сообщение об изменении состояния управляющего элемента – например, перемещение ползунка на экране.

На подобные изменения присоединенный объект, отслеживающий такие сигналы, может соответственно отреагировать (но не обязательно).

Это очень важный момент – соединяемые объекты могут быть абсолютно независимы и реализованы отдельно друг от друга, что позволяет объекту, отправляющему сигналы, не беспокоиться о том, что впоследствии будет происходить с этими сигналами. Принимают и обрабатывают сигналы другие объекты.

Благодаря такому разделению, можно разбить большой проект на компоненты, которые будут разрабатываться разными программистами по отдельности, а потом соединяться при помощи сигналов и слотов вместе. Это делает код очень гибким и легко расширяемым – если один из компонентов устареет или должен будет реализован иначе, то все другие компоненты, участвующие в коммуникации с этим компонентом, и сам проект в целом, не изменится. Новый компонент после разработки встанет на место старого и будет подключен к основной программе при помощи тех же самых сигналов и слотов.

Поэтому библиотека Qt особенно привлекательна при реализации компонентно-ориентированных приложений.

Однако не следует забывать, что большое количество взаимосвязей приводит к возникновению сильно связанных систем, в которых даже незначительные изменения могут привести к непредсказуемым последствиям.

Программирование в среде QT

Объектная модель Qt. Сигналы

Сигналы определяются в классе, как и обычные методы, только без реализации. С точки зрения программиста они являются лишь **прототипами методов, содержащихся в заголовочном файле определения класса.** Всю дальнейшую заботу о реализации кода для этих методов берет на себя MOC. Методы сигналов не должны возвращать каких-либо значений, и поэтому перед именем метода всегда должен стоять возвращаемый параметр void.

Сигнал не обязательно соединять со слотом. Если соединения не произошло, то он просто не будет обрабатываться. Подобное разделение отправляющих и получающих объектов исключает возможность того, что один из подсоединенных слотов каким-то образом сможет помешать объекту, отправившему сигналы. Библиотека Qt предоставляет большое количество уже готовых сигналов для существующих элементов управления. В основном, для решения поставленных задач хватает этих сигналов, но есть возможность создавать новые сигналы в своих классах.

Пример: Определение сигнала

```
class MySignal
{
  Q_OBJECT
  signals:
  void doIt ();
   . . .
};
```

Метод сигнала doIt() не имеет реализации – эту работу принимает на себя MOC, обеспечивая примерно такой алгоритм:

```
void MySignal::doIt ()
{
  QMetaObject :: activate (this, &staticMetaObject, 0, 0);
}
```

Программирование в среде QT

Объектная модель Qt. Сигналы

Очевидно, что не имеет смысла определять сигналы в классе как private, protected или public, поскольку они играют роль вызывающих методов.

Выслать сигнал можно при помощи ключевого слова emit. Ввиду того, что сигналы играют роль вызывающих методов, конструкция отправки сигнала emit doIt () приведет к обычному вызову метода doIt ().

Сигналы могут отправляться из классов, которые их содержат. Например, в листинге на предыдущем слайде сигнал doIt () может отсылаться только объектами класса MySignal, и никакими другими.

Чтобы иметь возможность отослать сигнал программно из объекта этого класса, следует добавить метод sendSgnal (), вызов которого заставит объект класса MySignal отправлять сигнал doIt (), как это показано на листинге ниже (слева).

Реализация сигнала

```
class MySignal
{
  Q_OBJECT
 public:
  void sendSignal()
   {
    emit doIt ();
   }
 signals:  void dolt();
};
```

Сигналы также могут высылать информацию, передаваемую в параметре. Например, если возникла необходимость передать в сигнале строку текста, то можно реализовать это, как показано ниже:

Реализация сигнала с параметром

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

Программирование в среде QT

Объектная модель Qt. Слоты

Слоты (slots) – это методы, которые присоединяются к сигналам, т.е. алгоритмы реакции на сообщение сигнала

Слоты – это практически обычные методы класса. Самое большое их отличие состоит в возможности принимать сигналы. Кроме того в слотах нельзя использовать параметры по умолчанию, например: slotMethod(int n=8),- или определять слоты как static.

Как и обычные методы, они определяются в классе как public, private или protected.

Если необходимо сделать так, чтобы слот мог соединяться только с сигналами сторонних объектов, но не вызываться как обычный метод со стороны, то тогда слот нужно объявить как protected или private.

Во всех других случаях объявляйте их как public.

В объявлениях перед каждой группой слотов должно стоять соответственно: private slots:, protected slots: или public slots:.

Слоты могут быть и виртуальными, но у них медленное соединение с сигналом.

Классы библиотеки Qt содержат целый ряд уже реализованных слотов. Но определение слотов для своих классов – это частая процедура.

Пример реализации слота показан в листинге:

Реализация слота

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

Внутри слота вызовом метода sender () можно узнать, от какого объекта был выслан сигнал. Он возвращает указатель на объект типа QObject. Например, в этом случае на консоль будет выведено имя объекта, выславшего сигнал:

```
void slot ()  {  qDebug () << sender () ->
                    objectName()  };
```

Программирование в среде QT

Объектная модель Qt. Соединение объектов

Соединение объектов осуществляется при помощи статического метода **connect ()**, который определен в классе **QObject.**

В общем виде вызов метода **connect ()** выглядит следующим образом:

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

Ему передаются пять следующих параметров:

- **sender** – указатель на объект, отправляющий 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 специальный макрос **SLOT (method ())**;
- **type** – управляет режимом обработки.

Имеется три возможных значения type:

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 автоматический режим, который действует следующим образом:
если отсылающий сигнал объект находится в одном потоке с принимающим его объектом, то устанавливается режим Qt::DirectConnection,
в противном случае – режим Qt::QueuedConnection.
Этот режим (Qt::Autoconnection) определен в методе connection () по умолчанию.
Вряд ли придется изменять режимы "вручную", но такая возможность есть.

Программирование в среде QT

Объектная модель Qt. Соединение объектов

Существует альтернативный вариант метода connect (), преимущество которого заключается в том, что все ошибки соединения сигналов со слотами выявляются на этапе компиляции программы, а не при ее исполнении, как это происходит в классическом варианте метода connect ().

Прототип альтернативного метода выглядит так:

```
QObject :: connect ( const QObject* sender,
                     QMetaMethod& signal,
const QObject* receiver,
QMetaMethod& slot,
Qt::ConnectionType type = Qt:AutoConnection);
```

Параметры этого метода полностью аналогичны предыдущему за исключением тех параметров, которые были объявлены в предыдущем методе как const char*. Вместо них используются указатели на методы сигналов и слотов классов напрямую. Благодаря при ошибке с названием сигнала или слота она будет выявлена сразу в процессе компиляции программы.

Недостаток альтернативного метода: при каждом соединении нужно явно указывать имена классов для сигнала и слота и следить за совпадением их параметров.

Пример демонстрирует работу первого метода connect ():

```
void main()
{ QObject::connect ( pSender, SIGNAL(signalMethod()),
                     pReceiver, SLOT(slotMethod()));  }
```

А вот пример соединение при помощи альтернативного метода connect ():

```
Qobject::connect ( pSender,  &SenderClass::signalMethod,
                   pReceiver,  &ReceiverClass::slotMethod );
```

Программирование в среде QT

Объектная модель Qt. Соединение объектов

Метод connect () после вызова возвращает объект класса Connection, с помощью которого можно определить, произошло ли соединение успешно. В случае, если в методе будет допущена какая-либо ошибка, то аварийно завершить программу вызовом макроса Q_ASSERT ().

Класс Connection содержит оператор неявного преобразования к типу bool, поэтому в примере используется переменная bOk этого типа.

```
bool  bOk = true;
bOk &= connect(pcmd1, SIGNAL(clicked()), pobjReceive1,
       SLOT(slotButton1Clicked()));
bOk &= connect(pcmd2, SIGNAL(clicked()), pobjReceiver2,
       SLOT(slotButton2Clicked()));
Q_ASSERT(bOk);
```

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. Отправляемый сигнал должен содержаться в определении класса:

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

Отправку сигналов можно на некоторое время заблокировать, вызвав метод blocksignals () с параметром true. Объект будет "молчать", пока блокировку не снимут тем же методом blockSignals () с параметром false. При помощи метода signalsBlocked() можно узнать текущее состояние блокировки сигналов.

Слот, не имеющий параметров, можно соединить с сигналом, имеющим параметры. Это удобно, когда сигналы поставляют больше информации, чем требуется для объекта, получающего сигнал. В этом случае в слоте можно не указывать параметры:

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

Программирование в среде QT

Объектная модель Qt. Пример Счётчик (механизм слотов и сигналов)

В этом примере создается приложение, в первом окне которого (рис. справа) находится кнопка нажатия ADD, а во втором (рис. слева) – виджет надписи. При щелчке в правом окне на кнопке ADD (Добавить) происходит увеличение отображаемого в левом окне значения на единицу. При значении пять – выход.


```
// main.cpp -Счетчик. Демонстрация работы механизма сигналов и слотов
#include <QtWidgets>
#include "Counter.h"
int main (int argc, char** argv)
{  QApplication app (argc, argv);
    QLabel     lbl ("0");
    QPushButton cmd ("ADD");
    Counter     counter;
     lbl.show ();
     cmd.show ();
   QObject::connect (&cmd, SIGNAL (clicked()),
                     &counter, SLOT (slotInc())  );
   QObject::connect (&counter,
                SIGNAL (counterChanged(int)),  &lbl,
                     SLOT(setNum(int))    );
   QObject::connect (&counter,
        SIGNAL(goodbye()),  &app, SLOT(quit())
);
    return app.exec();
}
```

В main.cpp создается объект надписи lbl, нажимающаяся кнопка cmd и объект счетчика counter (описание которого приведено в листингах Counter.cpp и Counter.h).

Далее сигнал clicked() соединяется со слотом slotInc(). При каждом нажатии на кнопку вызывается метод slotInc(), увеличивающий значение счетчика на 1.

Метод slotInc() должен сообщать о подобных изменениях, чтобы элемент надписи отображал всегда только действующее значение. Для этого сигнал counterChanged(int), передающий в параметре актуальное значение счетчика, соединяется со слотом setNum(int), способным принимать это значение.

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.

Программирование в среде QT

Объектная модель Qt. Пример Счётчик

```
//============================
// Counter.h
//============================
#pragma once
#include <QObject>
class Counter : public QObject
 {        Q_OBJECT
  private:
     int m_nValue;
  public:
    Counter();
  public slots:
     void slotInc();
  signals:
     void goodbye  ( );
     void counterChanged(int);
 };
```

```
//=======================================
// Counter.cpp
//=======================================

#include "Counter.h"

Counter::Counter() : QObject(), m_nValue(0)  { }
// ---------------------------------
void Counter::slotInc()
   {  emit counterChanged(++m_nValue);
      if (m_nValue == 5)
       {
         emit goodbye();
       }
    }
```

Как видно из листинга Counter.h, в определении класса счетчика содержатся два сигнала: goodbye(), сообщающий о конце работы счетчика, и counterChanged(int), передающий актуальное значение счетчика, а также слот slotInc(), увеличивающий значение счетчика на единицу.

В листинге Counter.cpp метод слота slotInc() отправляет два сигнала: counterChanged() и goodbye(). Сигнал goodbye() отправляется при значении атрибута m_nValue, равном 5.

Программирование в среде QT

Объектная модель Qt. Разъединение объектов

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 connect(). В общем виде этот метод выглядит таким образом:

```
Qobject : disconnect (sender, signal, receiver, slot);
```

Пример демонстрирует, как может быть выполнено разъединение объектов в программе:

```
void main()
 {
   Qobject : :disconnect (pSender, SIGNAL (signalMethod()),
                          pReceiver, SLOT (slotMethod())
                         );
 }
```

Существуют два сокращенных, не статических варианта:
disconnect (signal, receiver, slot)
и
disconnect(receiver, slot).

Программирование в среде QT

Объектная модель Qt. Переопределение сигналов

Если надо сократить в классе количество слот-методов и выполнить действия на разные сигналы в одном слоте, то следует воспользоваться классом QSignalMapper. С его помощью можно переопределить сигналы и сделать так, чтобы в слот отправлялись значения типов int, QString или QWidget.

Рассмотрим этот механизм на примере использования значений типа QString.

В программе есть две кнопки: при нажатии на первую кнопку нам нужно отобразить сообщение Button1 Action, а при нажатии на вторую – Button2 Action. Можно было бы реализовать в классе два разных слота, которые были бы соединены с сигналами clicked () каждой из двух кнопок и выводили бы каждый свое сообщение.

Но в примере показано как воспользоваться классом QSignalMapper:

```
MyClass::MyClass(QWidget* pwgt)
{  QSignalMapper* psigMapper =
                    new QSignalMapper(this);
  Connect (psigMapper, SIGNAL(mapped(const QString&)),
        this, SLOT(slotShowAction(const QString&))  ) ;
  QPushButton* pcmdl = new QPushButton("Button1");
 connect (pcmdl, SIGNAL (clicked ()), psigMapper,
                              SLOT(map()));
  psigMapper->setMapping(pcmdl, "Button1 Action");
  QPushButton* pcmd2 = new QPushButton("Button2");
 connect (pcmd2, SIGNAL (clicked ()) , psigMapper,
                              SLOT(map()));
  psigMapper->setMapping(pcmdl, "Button2 Action");
}

void MyClass::slotShowAction(const QString& str)
  { qDebugO << str;  }
```

Создаётся объект класса QSignalMapper, соединяется его сигнал mapped() с единственным слотом slotShowAction(), который принимает объекты QString.

Класс QSignalMapper предоставляет слот map(), с которым должен быть соединен каждый объект, сигнал которого должен быть переопределен.

При помощи метода QSignalMapper::setMapping() устанавливается конкретное значение, которое должно быть направлено в слот при получении сигнала, – в нашем случае сигнала clicked ().

Программирование в среде QT

Объектная модель Qt. Организация объектных иерархий

Организация объектов в иерархии снимает с разработчика необходимость самому заботиться об освобождении памяти от созданных объектов.
Конструктор класса QObject выглядит следующим образом:

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

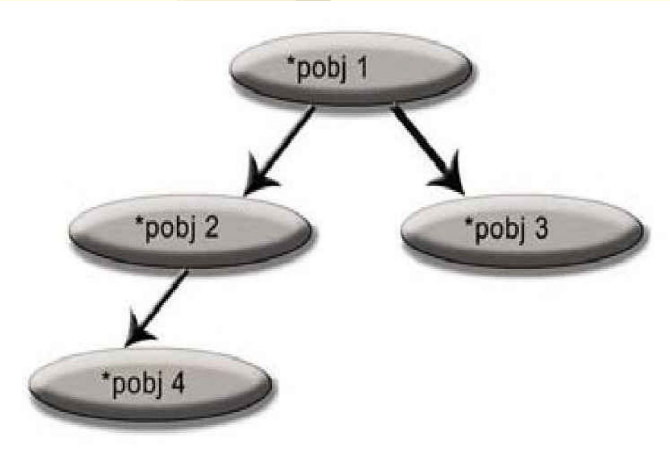
В его параметре передается указатель на другой объект класса QObject или унаследованного от него класса. Благодаря этому параметру существует возможность создания объектов-иерархий. Он представляет собой указатель на объект-предок. Если в первом параметре передается значение, равное нулю, или ничего 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 (), как показано в листинге ниже:

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

Схема получившейся объектной иерархии



При уничтожении созданного объекта (при вызове его деструктора) все присоединенные к нему объекты-потомки в куче уничтожаются автоматически.

Программирование в среде QT

Объектная модель Qt. Организация объектных иерархий

При программировании с Qt важно помнить, что все объекты должны создаваться в памяти динамически, с помощью оператора new. Исключение могут составлять только объекты, не имеющие предков.

Для получения информации об объектной иерархии существуют методы: parent() и children().

С помощью метода parent() можно определить объект-предок. Вызов pobj2->parent() вернет указатель на объект obj1. Для объектов верхнего уровня этот метод вернет значение 0. Чтобы вывести на консоль всю цепь имен объектов-предков какого-либо из объектов, можно поступить так, как показано ниже в листинге для объекта pobj4:

```
for (QObject* pobj = pobj4; pobj; pobj = pobj->parent())
    { qDebug() << pobj->obj ectName(); }
```

И на экране появиться:

the first child of pobj2
the first child of pobj1

Метод children () возвращает константный указатель на список объектов-потомков.

Метода findchlid() осуществляет поиск нужного объекта-потомка. В параметре метода передаётся имя искомого объекта. Например:

```
QObject* pobj = pobj1->findChild<QObject*>("the first child of pobj2");
```

Метод расширенного поиска findChildren(), возвращающий список указателей на объекты. Все параметры метода не обязательны: может передаваться либо строка имени, либо регулярное выражение, а вызов метода без параметров приведет к тому, что он вернет список указателей на все объекты-потомки. Пример ниже возвратит список указателей на объекты с именем которых начинается с букв th:

```
QList<QObject*> plist = pobjl->findChildren<QObject*>(QRegExp("th*"));
```

Для того чтобы возвратить все объекты потомков указанного типа, независимо от их имени, нужно просто не указывать аргумент:

```
QList<QObject*> plist = pobj1->findChildren<QObject*>();
```

Будут возвращены указатели на объекты pobj2, pobj3 и pobj4.

Для отладки программы полезен метод dumpObjectInfo(), который показывает следующую информацию, относящуюся к объекту:

- имя объекта;
- класс, от которого был создан объект;
- сигнально-слотовые соединения.

Вся эта информация поступает в стандартный поток вывода stdout.

При отладке можно воспользоваться и методом dumpObjectTree(), предназначенным для отображения объектов-потомков в виде иерархии. Например, вызов dumpObjectTree() для нашего объекта pobj1 (из листинга на слайде Переопределение сигналов) покажет:

```
QObject::
QObject::the first child of pobj1
QObject::the first child of pobj2
QObject::the second child of pobj1
```

Программирование в среде QT

Объектная модель Qt. Метаобъектная информация

Каждый объект, созданный от класса QObject или от унаследованного от него класса, располагает структурой данных, называемой метаобъектной информацией (класс QMetaObject). В ней хранится информация о сигналах, слотах (включая указатели на них), о самом классе и о наследовании. Получить доступ к этой информации можно посредством метода QObject::metaObject().

Чтобы узнать имя класса объекта, от которого он был создан надо:

```
qDebug() << pobj1->metaObject()->className();
```

Чтобы сравнить имя класса с известным:

```
if (pobj1->metaObject()->className() == "MyClass")
{ // Выполнить какие-либо действия }
```

Для получения информации о наследовании классов существует метод inherits (const char*), который определен непосредственно в классе QObject и возвращает значение true, если класс объекта унаследован от указанного в этом методе класса либо создан от данного класса, иначе метод возвращает значение false.

```
if(pobj->inherits("QWidget"))
{   QWidget* pwgt = static_cast<QWidget*>(pobj);
    // Выполнить какие-либо действия с pwgt    }
```

Метаобъектную информацию использует и операция приведения типов qobject_cast<T>. Таким образом, при помощи метода inherits () пример можно изменить:

```
QWidget* pwgt = qobject_cast<QWidget*> (pobj ) ;
if(pwgt)      {   // Выполнить какие-либо действия с pwgt   }
```

К метаобъектной информации относится также и метод tr(), предназначенный для программ интернационализации (интернационализация – возможность выбора языка интерфейса приложения Qt).

Программирование в среде QT

Рекомендации для проекта с Qt

При реализации файлы классов лучше всего разбивать на две отдельные части. Часть определения класса помещается в заголовочный файл с расширением h, а реализация класса (члены-методы) – в файл с расширением cpp.

В заголовочном файле с определением класса важно размещать директиву препроцессора #ifndef. Смысл этой директивы состоит в том, чтобы избежать конфликтов в случае, когда один и тот же заголовочный файл будет включаться в исходные файлы более одного раза.

```
#ifndef _MyClass_h_
#define _MyClass_h_
  class MyClass { . . . };
#endif  //_MyClass_h_
```

Эту конструкцию можно заменить на эквивалентную с использованием прагмы, тогда код заголовочного файла будет более компактным:

```
#pragma once class
MyClass {  };
```

По традиции заголовочный файл, как правило, носит имя содержащегося в нем класса. В заголовочных файлах, в целях более быстрой компиляции, для указателей на типы данных используется предварительное объявление для типа данных, а не прямое включение посредством директивы #include.

В начале определения класса содержится макрос Q_OBJECT для MOC – это необходимо, если класс использует сигналы и слоты, а в других случаях, если нет нужды в метаинформации, этим макросом можно пренебречь. Но нужно учитывать то обстоятельство, что из-за отсутствия метаинформации нельзя будет использовать приведение типа qobject_cast<T> (obj).

```
class MyClass : public QObject {
 Q_OBJECT
public:
   MyClass();
};
```