Создание простого оконного приложения в Qt Creator*

1 Создание проекта

Для того, чтобы создать простое оконное приложение, использующее фреймворк Qt, с помощью среды Qt Creator, достаточно выбрать `File - New File or Project - Applications - Qt Gui Application` и затем указать расположение файлов проекта, выбрать используемые тулчейны, название класса для главного окна, его базовый класс и имена генерируемых файлов; также будет предложено сгенерировать форму (чекбокс `Generate form`) и указать имя данного файла.

После выполнения этих действий в дереве текущего проекта появятся (если вы оставили все настройки и имена файлов без изменений):

- Файл проекта (`[название проекта].pro`)
- Заголовочный файл главного окна приложения (`mainwindow.h`)
- Реализация функции `main`, содержащей инициализацию приложения и создание главного окна (`main.cpp`)
- Реализация функций класса `MainWindow` (`mainwindow.cpp`)
- Описание формы (`mainwindow.ui`)

1.1 User Interface Compiler (`uic`)

Листинг 1: Пример qmake-проекта для приложения, содержащего формы с XML описанием

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

TARGET = gui_application
TEMPLATE = app


SOURCES += main.cpp\
        mainwindow.cpp

HEADERS  += mainwindow.h

FORMS    += \
        mainwindow.ui
```

Несложно заметить, что по сравнению с проектом консольного приложения, в файле проекта (листинг 1) появилось управление ещё одной переменной – `FORMS`. По этим файлам (которые содержат описание элементов пользовательского интерфейса в формате XML) с помощью `uic`[1] будут сгенерированы соответствующие заголовочные файлы.

С их содержанием можно ознакомиться либо сгенерировав их самостоятельно используя `uic`, либо просмотрев их после сборки в целевой директории.

*Руководство *должно* соответствовать Qt Creator 2.8.1; работа с другими версиями может несколько отличаться

[1]Более подробно ознакомиться с User Interface Compiler можно в документации: `http://qt-project.org/doc/qt-5.1/qtdoc/uic.html`

1.2 Meta Object Compiler (`moc`)

Листинг 2: Содержимое заголовочного файла, сгенерированного для главного окна приложения

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
        Q_OBJECT

public:
        explicit MainWindow(QWidget *parent = 0);
        ~MainWindow();

private:
        Ui::MainWindow *ui;
};

#endif // MAINWINDOW_H
```

В заголовочном файле, сгенерированном для главного окна приложения (`mainwindow.h`, листинг 2) используется макрос `Q_OBJECT`, и поэтому файл будет обработан на стадии сборки мета-объектным компилятором (`moc` [2]).

Листинг 3: Содержимое исходного файла, сгенерированного для главного окна приложения

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

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

MainWindow::~MainWindow()
{
        delete ui;
}
```

Файл `mainwindow.cpp` (листинг 3) использует заголовочный файл `ui_mainwindow.h`, который будет сгенерирован при сборке с помощью `uic` (см. раздел 1.1) на основе `mainwindow.ui`

[2] Утилита, позволяющая использовать расширения C++ от Qt в файлах, содержащих объявление классов, содержащих макрос `Q_OBJECT`; расширения C++ позволяют использовать систему сигналов и слотов (описанную в разделе 2), RTTI, системы свойств; в результате работы `moc` будет сгенерирован новый заголовочный файл; описание в документации: `http://qt-project.org/doc/qt-5.1/qtdoc/moc.html`

2 Сигналы и слоты[3]

Объекты в Qt связываются при помощи подхода сигналов и слотов, реализуя шаблон проектирования «наблюдатель»[4]; с использованием Qt реализация данного подхода упрощается за счёт использования `moc`.

Сигнал вырабатывается, когда происходит некоторое событие. Слот – это функция, вызываемая в ответ на связанные с ней сигналы.

2.1 Сигналы

Сигналы вырабатываются классами; только класс, определяющий сигнал и его потомки могут вырабатывать сигнал.

Когда сигнал вырабатывается (с помощью ключевого слова `emit`), в некотором порядке запускаются все связанные с ним слоты (выполнение кода, следующего после `emit` может выполнятся как после выполнения всех связанных слотов, так и сразу (при связях с отложенным исполнением), в зависимости от типов связей).

Сигналы определяются с помощью ключевого слова `signals` (листинг 4) (наличие макроса `Q_OBJECT` необходимо для того, чтобы файл был обработан `moc` (см. раздел 1.2)).

Листинг 4: Пример объявления сигнала

```
class MyClass : public QObject {
        Q_OBJECT
...
signals:
        void mySignal();
        void signalWithParam(int value);
}
```

Сигналы могут иметь любое количество аргументов; реализовывать эти функции не нужно.

2.2 Слоты

Слоты являются обычными функциями-членами класса. При вызове в качестве функции, используются обычные правила C++; но вызов в качестве слота – доступен для любого класса, независимо от уровня доступа.

Слоты определяются с помощью ключевого слова `slots` (листинг 5) (как и сигналы, они могут иметь любое количество аргументов).

Листинг 5: Пример объявления слота

```
class MyClass : pulic QObject {
        Q_OBJECT
...
public slots:
        void mySlot(int arg);
}
```

Реализовывать функции-слоты необходимо.

2.3 Связывание

Связывать можно:

- Слот с сигналом (при возникновении сигнала будет вызван связанный с ним слот)
- Сигнал с сигналом (при возникновении первого сигнала будет выработан второй)

[3] Более подробно механизм описан в документации: `http://qt-project.org/doc/qt-5.1/qtcore/signalsandslots.html`

[4] `http://en.wikipedia.org/wiki/Observer_pattern`

Все связи имеют характер «многие-ко-многим» (т.е. каждый сигнал может быть связан с несколькими слотами, и каждый слот – с несколькими сигналами).

Связывание осуществляется с помощью статического метода `connect` класса `QObject`. При этом доступны две[5][6] перегрузки метода `connect` (листинг 6)

Листинг 6: Два способа связывания сигнала со слотом

```
QObject::connect(object1, SIGNAL(valueChanged(int)), object2, SLOT(setValue(int));
QObject::connect(object1, &Class1::valueChanged, object2, &Class2::setValue);
```

В первом случае связывание происходит во время исполнения и во время компиляции ошибки, связанные с типами аргументов, наличием слотов и сигналов не проверяются (связывание производится по строковым сигнатурам (полученным с помощью макросов `SLOT` и `SIGNAL`) методов).

Во втором – проверяются (но в случае, если используемый вами компилятор не поддерживает C++11, в частности `variadic templates`, то число аргументов у сигналов и слотов при использовании такого синтаксиса ограничено шестью).

Уничтожить созданную ранее связь можно используя статический метод `disconnect` класса `QObject`.

3 Использование подхода модель-представление

Для использования подхода «модель-представление» предлагается использовать механизм слотов и сигналов для связи модели и представления (при проектировании следует помнить, что наличие циклов в графе связей между сигналами и слотами не контролируется).

Стандартные компоненты (`QListView`, `QTreeView`, `QTableView`,...) предполагают использование в качестве модели класса, унаследованного от `QAbstractItemModel`[7] (или его потомков:`QAbstractListModel`, `QAbstractTableModel`, ...). При этом, в зависимости от того, какое подмножество операций (чтение, запись, поддержка выделения, ...) должна обеспечивать модель, потребуется реализовать то или иное подмножество методов в классе-наследнике.

[5] Начиная с Qt 5.0

[6] На самом деле – чуть больше: `http://qt-project.org/doc/qt-5.1/qtcore/qobject.html#connect` , в частности – тип связи

[7] `http://qt-project.org/doc/qt-5.1/qtcore/qabstractitemmodel.html`

Приложения

В этой части будет рассмотрено простое приложение, в некоторой мере эксплуатирующее механизм слотов и сигналов для реализации пользовательского интерфейса и его связи с моделью данных.

A Проект

Создайте новый проект оконного приложения (`File - New File or Project - Applications - Qt Gui Application`)

B Модель

Реализуем модель, состояние которой описывается целым числом a, обладающую следующим поведением:

- позволяет инкрементировать и декрементировать a
- оповещает все связанные объекты при изменении состояния

Для этого реализуем:

- Слоты `increment` и `decrement`
- Сигнал `valueChanged`

B.1 Объявление

Добавьте новый класс в проект:

- `File - New File or Project`, или `Ctrl+N`, или `Add New` в контестном меню проекта в дереве проекта
- Выберите `C++ - C++ Class`
- Введите имя класса (`Model`), базовый класс (`QObject`); остальные поля можно оставить в значении по-умолчанию

После этих действий, вы должны получить файл `model.h` подобного содержания: 7

Листинг 7: Содержимое файла `model.h` после создания класса

```
#ifndef MODEL_H
#define MODEL_H

#include <QObject>

class Model : public QObject
{
    Q_OBJECT
public:
    explicit Model(QObject *parent = 0);

signals:

public slots:

};

#endif // MODEL_H
```

В созданном файле `model.h`:

- Добавьте объявление `private` поля `a` (оно будет описывать состояние модели)
- Добавьте объявления слотов `increment` и `decrement` (в секции `public slots`, см. 5)
- Добавьте объявление сигнала `valueChanged` с аргументами типов `QString` (в нём будет передаваться значение `a` в текстовом виде) и `int` (в нём будет передаваться значение поля `a`) (его нужно объявить в секции `signals`, см.4)
- Добавьте объявления методов `getText` (он будет возвращать строковое представление значения `a`), `getValue` и `setValue` (он будет устанавливать и получения значения поля `a`)
- Удалите аргументы из объявления конструктора

В результате этих манипуляций должно получиться следующее объявление класса `Model`: 8

Листинг 8: Объявление класса модели

```
#include <QObject>
#include <QString>

class Model : public QObject
{
    Q_OBJECT
public:
    explicit Model();
    void setValue(int value);
    int getValue();
    QString getText();
signals:
    void valueChanged(QString displayValue, int value);
public slots:
    void increment();
    void decrement();
private:
    int a;
};
```

B.2 Реализация

Добавьте реализации всех объявленных методов (метод слота `valueChanged` реализовывать не нужно):

- `getText`: используйте статический метод `number` класса `QString`
- `setValue`: установите новое значение поля `a` и выработайте сигнал `valueChanged` (для аргумента текстого значения используйте `getText`)
- `getValue`: верните значение поля `a`
- `increment`, `decrement`:
  - инкрементируйте (декрементируйте) поле `a`
  - выработайте сигнал `valueChanged`
- Конструктор: установите значение поля `a` в 0 (вырабатывать сигнал нет необходимости, так как в момент вызова конструктора объект не мог быть связан ни с каким другим)

В результате должна получиться подобная реализация класса `Model`: 9

Листинг 9: Реализация модели

```
Model::Model()  :  a(0)  {
}

QString  Model::getText()  {
        return  QString::number(a);
}

void  Model::setValue(int  value)  {
        a  =  value;
        emit  valueChanged(getText(),  a);
}

int  Model::getValue()  {
        return  a;
}

void  Model::increment()  {
        ++a;
        emit  valueChanged(getText(),  a);
}

void  Model::decrement()  {
        --a;
        emit  valueChanged(getText(),  a);
}
```

C Пользовательский интерфейс и связь слотов с сигналами

C.1 Пользовательский интерфейс

Создадим пользовательский интерфейс и объеденим его с созданной моделью.

Откройте `mainwindow.ui` в дизайнере форм. Удалите меню (`menuBar`), панель инструментов (`mainToolBar`) и строку статуса (`statusBar`); для этого можно воспользоваться контекстным меню нужного объекта в древовидным представлением в окне `Object Inspector`, либо контекстным меню элемента на форме, отображаемой дизайнером форм. Если эти объекты нужно добавить – это можно сделать через контекстное меню формы, или древовидного представления с помощью пунктов контекстного меню (`Create Menu Bar`, `Add Tool Bar`, `Create Status Bar`).

Добавьте на форму две кнопки («Push Button»), перетащив их из списка элементов, в панели свойств задайте им имена `incrementButton` и `decrementButton` (`objectName`) и текст (`text`). Добавьте на форму виджет для отображения текста Label, и задайте ему имя `valueLabel`.

C.2 Связывание сигналов и слотов

Модифицируйте класс главного окна, добавив приватное поле `model` для указателя на модель и его инициализацию в конструкторе.

В конструкторе главного окна нужно связать:

- Сигнал `clicked` кнопки `incrementButton` со слотом `increment` модели
- Сигнал `clicked` кнопки `decrementButton` со слотом `decrement` модели
- Сигнал `valueChanged` модели со слотом `setText valueLabel`

Также имеет смысл добавить в конструктор инициализацию поля, отображающего состояние модели в её начальное значение.

В результате должна получиться подобная реализация класса `MainWindow`: 1011

Осталось изменить метод `main`, добавив в него создание модели, и пример можно будет запустить.

Листинг 10: Объявление класса главного окна

```
#include <QMainWindow>
#include "model.h"

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
        Q_OBJECT

public:
        explicit MainWindow(Model *model, QWidget *parent = 0);
        ~MainWindow();

private:
        Ui::MainWindow *ui;
        Model *model;
};
```

Листинг 11: Реализация класса главного окна

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(Model *m, QWidget *parent) :
        QMainWindow(parent),
        ui(new Ui::MainWindow),
        model(m) {

        Q_ASSERT(m);

        ui->setupUi(this);

        ui->valueLabel->setText(model->getText());

        QObject::connect(ui->incrementButton, &QPushButton::clicked,
                         model, &Model::increment);
        QObject::connect(ui->decrementButton, &QPushButton::clicked,
                         model, &Model::decrement);
        QObject::connect(model, &Model::valueChanged,
                         ui->valueLabel, &QLabel::setText);
}

MainWindow::~MainWindow() {
        delete ui;
}
```

Один экземпляр реализованной подобным образом модели можно использовать в любом количестве окон, тогда на изменение модели будут реагировать все связанные окна; проверьте это, добавив в метод `main` создание ещё одного окна (конечная реализация приведена в 12)

Листинг 12: Реализация метода `main` с иллюстрацией использования одной модели в нескольких окнах

```
#include "mainwindow.h"
#include "model.h"
#include <QApplication>

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

        Model m;

        MainWindow w(&m), w2(&m);

        w.setWindowTitle("First window");
        w2.setWindowTitle("Second window");

        QRect g = w.geometry();
        w.setGeometry(50, 50, g.width(), g.height());
        w2.setGeometry(150, 150, g.width(), g.height());

        w.show();
        w2.show();

        return a.exec();
}
```

D Работа с меню

Работа с меню окна осуществляется в дизайнере форм[8].

Добавьте (если оно отсутствует) главное меню к форме с помощью пункта `Create Menu Bar` в контекстном меню формы.

Теперь можно последовательно добавлять элементы (щёлкнув в `Type here` и написав имя элемента) в меню; для указания клавиши, соответствующей пункту меню используйте символ `&`. Элементы меню бывают двух типов:

- `QMenu` – пункт меню, содержащий подменю
- `QAction` – пункт меню, не содержащий подменю

Создайте меню `Model` с пунктами `Increment` и `Decrement`, и добавьте в конструктор главного окна связывание сигналов `triggered` пунктов меню со слотами модели (код для связывания приведён в 13).

Листинг 13: Связывание пунктов меню и модели

```
QObject::connect(ui->actionIncrement, &QAction::triggered, model, &Model::increment);
QObject::connect(ui->actionDecrement, &QAction::triggered, model, &Model::decrement);
```

С помощью свойства `enabled` можно управлять доступностью меню, подменю и пунктов меню; с помощью свойства `checkable` пункта меню можно получить пункт меню, который можно отметить

[8]Конечно же, можно написать необходимый код самостоятельно, так как дизайнер форм лишь создаёт описание, по которому потом (с помощью `uic`) генерируется код

(состояние отметки можно передать в слот с подходящей сигнатурой); с помощью свойства `shortcut` можно задать горячую клавишу, соответствующую пункту меню.

Аналогичным образом осуществляется работа с панелью инструментов. Создайте её с помощью пункта `Add Tool Bar` контекстного меню формы. Теперь перейдите в окно `Action Editor`, в нём есть `actionIncrement` и `actionDecrement` (возникшие в процессе создания меню, новые `QAction` можно создать в окне `Action Editor` с помощью кнопки `New`). Перетащите их на созданную панель инструментов, после чего достаточно будет пересобрать приложение, так как связь необходимых сигналов уже была осуществлена при создании меню.

Добавить разделитель между пунктами меню и панели инструментов можно с помощью их контекстного меню (пункты `Insert separator`, `Append separator`).

E Задание минимального размера окна

По-умолчанию минимальный размер окна равен 0×0. Изменить его можно с помощью свойства `minimumSize` (либо в редакторе форм, либо в коде методом `setMinimumSize`).

F Заголовок окна

Текст заголовка окна соответствует значению поля `windowTitle`. Изменить его можно в редакторе форм (свойство `windowTitle`) и в коде методом `setWindowTitle`.

G Рисование в пределах пользовательского интерфейса

Для рисования в пределах элемента пользовательского интерфейса необходимо переопределить метод `paintEvent` в дочернем классе.

После этого, используя класс `QPainter` можно рисовать различные примитивы, настроив перо (`QPen`, задаёт стиль границы рисуемого примитива), кисть (`QBrush`, задаёт метод закраски внутренности) и другие атрибуты. Пример использования класса `QPainter` приведён в 14.

H Диалоги

В Qt диалоговые окна являются наследниками класса `QDialog` и могут быть модальными (не позволяющими продолжить взаимодействие с программой до завершения диалога) и немодальными (позволяющими продолжать работу с приложением во время работы с диалогом).

Работа с диалогом происходит следующим образом:

1. Инициализация
   - Модальный диалог: может быть объявлен либо в месте вызова (но тогда установленные пользователем значения не сохранятся при следующем вызове, что может оказаться нарушением ожидаемого поведения), либо как член класса, тогда при следующем вызове состояние будет совпадать с предыдущим
   - Немодальный диалог: должен быть объявлен как член класса, иначе при выходе из области видимости переменная будет уничтожена и диалог исчезнет с экрана; требуется связать необходимые сигналы диалога с соответствующими слотами
2. Вызов
   - Модальный диалог: показ и получение статуса с помощью метода `exec`, получение результата через соответствующие методы и поля класса[9]
   - Немодальный диалог: показ с помощью метода `show` (логично отобразить диалог поверх других окон с помощью метода `raise` и установить правильный фокус ввода с помощью `activateWindow`), получение результата через сигналы

[9]На самом деле, получать результат завершения модального диалога можно так же, как и в случае немодального диалога (через сигналы), но это приводит к излишнему количеству кода

Листинг 14: Пример использования класса `QPainter`

```
void MainWindow::paintEvent(QPaintEvent *) {
        QPen pen;
        QBrush brush(Qt::blue);
        QPainter painter(this);

        pen.setWidth(2);
        brush.setStyle(Qt::BDiagPattern);

        painter.setBrush(brush);
        painter.setPen(pen);
        painter.drawRoundedRect(QRect(65, 85, 100, 120), 5.0, 5.0);

        QLinearGradient grad(115.0, 115.0, 150.0, 150.0);
        grad.setColorAt(0.0, Qt::blue);
        grad.setColorAt(1.0, Qt::white);
        QBrush gradientBrush(grad);
        pen.setColor(Qt::red);
        pen.setStyle(Qt::DotLine);

        painter.setBrush(gradientBrush);
        painter.setPen(pen);
        painter.drawEllipse(115, 115, 45, 60);

        pen.setColor(Qt::yellow);

        painter.setPen(pen);
        painter.drawText(95, 95, "QPainter");
}
```

H.1 Стандартные диалоги

В фреймворке Qt реализованно несколько стандартных диалогов для часто встречающихся применений:

- Выбор цвета (`QColorDialog`)
- Выбор файлов и каталогов (`QFileDialog`)
- Выбор шрифта (`QFontDialog`)
- Ввод одного значения (строки, числа), выбор строки из списка пользователем (`QInputDialog`)
- Информирование пользователя о событии или получение ответа на вопрос в форме «Да»/«Нет» (`QMessageBox`) (модальный диалог)
- Конфигурация параметров страницы для принтера (`QPageSetupDialog`)
- Конфигурация принтера (`QPrintDialog`)
- Предпросмотр печати (`QPrintPreviewDialog`)
- Информирование пользователя о прогрессе длительной задачи (`QProgressDialog`)

Пример использования стандартного диалога приведён в листинге 15 (вызывается диалог выбора файла, и в случае выбора пользователем файла – показывается MessageBox с его именем, в случае отмены – сообщение об отмене)

Листинг 15: Пример использования стандартного диалога

```
QFileDialog file(this);
file.setFileMode(QFileDialog::AnyFile);
QMessageBox mbox;
if(QDialog::Accepted == file.exec()) {
        mbox.setWindowTitle("User accepted dialog");
        mbox.setInformativeText(file.selectedFiles()[0]);
} else {
        mbox.setWindowTitle("User rejected dialog");
        mbox.setInformativeText("User closed dialog or pressed \"Cancel\" button");
}
mbox.exec();
```

H.2 Создание собственных диалогов

Создание собственных диалогов аналогично созданию форм. Для создания шаблона диалога выберите `File - New File or Project - Qt - Qt Designer Form` и выберите желаемый шаблон для диалога. После этого можно будет реализовать необходимый пользовательский интерфейс и требуемое поведение точно так же, как было показано выше на примере формы.