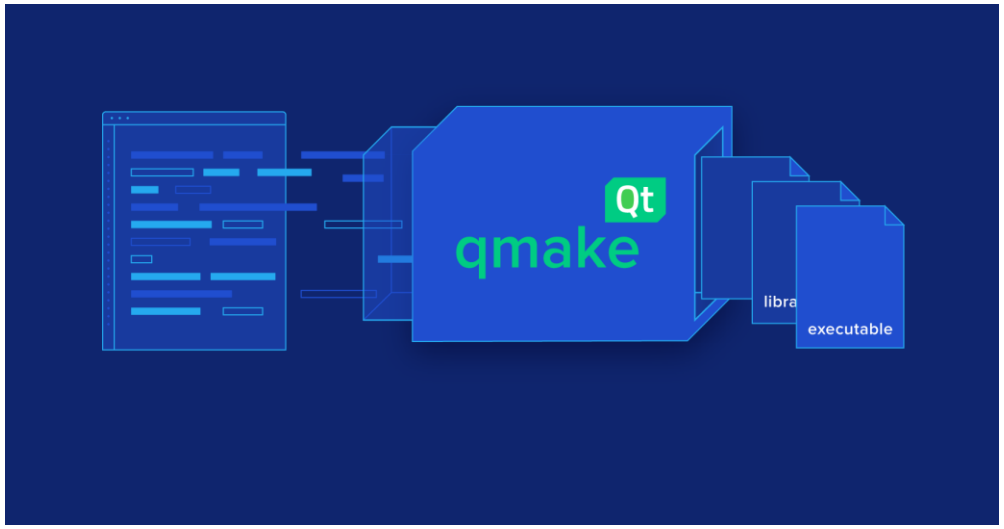
QMake. Структура проекта. MOC. Отладка ПО

Лекция + практика

Утилита qmake.

Программисты, как правило используют *make-файл* (*makefile*), который берет на себя всю работу по настройке компилятора и компоновщика.



Создание *make-файлов* вручную требует опыта и понимания процессов компоновки приложения, причем в зависимости от платформы вид этих файлов будет различаться. Сегодня появились специальные утилиты - генераторы, которые берут на себя работу по созданию *make-файлов*.
Утилита **qmake** вошла в поставку Qt, начиная с версии 3.0. Эта утилита так же хорошо переносима, как и сама Qt. Программа **qmake** при создании *make-файлов* интерпретирует файлы проектов, которые имеют расширение *.pro* и содержат различные параметры. Отметим, что утилита **qmake** способна **создавать** не только ***make-файлы***, но и **сами *pro-файлы***.

Утилита qmake.

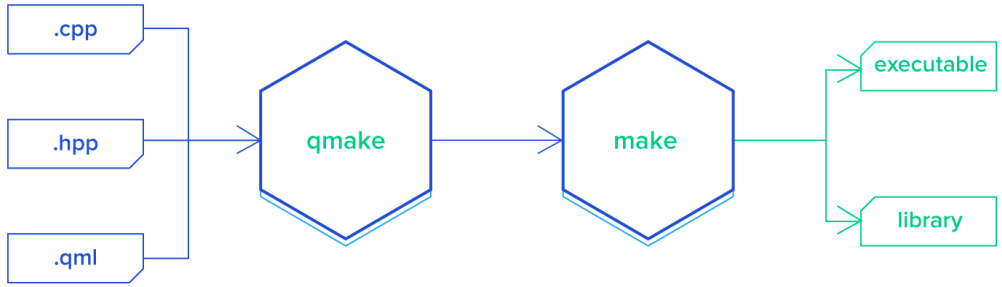
Допустим, вы указали, что в каталоге есть исходные файлы C++, выполнив следующую команду:

- **qmake -project**

В результате будет реализована попытка автоматического создания *pro-файла*. Это удобно, поскольку на первых порах вам не понадобится вникать во все тонкости создания *pro-файлов*. Также это может оказаться полезным в том случае, если вы обладаете большим количеством файлов, к которым требуется создать *pro-файл*, тогда у вас отпадет необходимость вносить их имена вручную. Создать из *pro-файла make-файл* нетрудно, для этого нужно просто выполнить команду:

- **qmake *file.pro* -o Makefile**

В этой команде, *“file.pro”* - это имя *pro-файла*, а *“Makefile”* - имя для создаваемого платформозависимого *make-файла*.



Если бы мы выполнили команду без параметров, то утилита *qmake* попыталась бы найти в текущем каталоге *pro-файл* и, в случае успеха, автоматически создала бы *make-файл*. Таким образом, имея в распоряжении только исходные файлы на C++, можно создать исполняемую программу, выполнив всего лишь три команды:

> qmake -project
> qmake
> make

Опции файла проекта .pro

Опция	Назначение
HEADERS	Список созданных заголовочных файлов
SOURSES	Список созданных файлов реализации (с расширением cpp)
FORMS	Список файлов с расширением ui. Эти файлы создаются программой Qt Designer и содержат описание интерфейса пользователя в формате XML
LIBS	Задает список библиотек, которые должны быть подключены для создания исполняемого модуля
CONFIG	Задает опции, которые должен использовать компилятор
DESTDIR	Задает путь, куда будет помещен готовый исполняемый модуль
DEFINES	Здесь можно передать опции для компилятора. Например, это может быть опция помещения отладочной информации для отладчика debuger в исполняемый модуль
INCLUDPATH	Путь к каталогу, где содержатся заголовочные файлы. Этой опцией можно воспользоваться в том случае, если уже есть готовые заголовочные файлы, и вы хотите использовать их (подключить) в текущем проекте
DEPENDPATH	В этом разделе указываются зависимости, необходимые для компиляции
SUBDIRS TEMPLATE	Задает имена подкаталогов, которые содержат pro-файлы Задает разновидность проекта: app - приложение, lib - библиотека, subdirs - подкаталоги
TARGET	Имя приложения. Если это поле не заполнено, то название программы будет соответствовать имени проектного файла

Анатомия проектных файлов

```
TEMPLATE = app
HEADERS += filel.h \
           file2.h
SOURCES += main.cpp  \
           filel.cpp  \
           file2.cpp
TARGET file
CONFIG += qt warn_on release
```

В первой строке задается тип программы:
TEMPLATE = *app* (если бы нам нужно было создать библиотеку, то TEMPLATE имело бы значение *lib*).
Во второй строке (HEADERS) перечисляются все заголовочные файлы, принадлежащие проекту.
В опции SOURCES - все файлы реализации проекта. Строка TARGET - имя программы, а строка CONFIG - опции, которые должен использовать компилятор в соответствии с подсоединяемыми библиотеками.
В рассматриваемом случае:
+ qt указывает, что это Qt-приложение и используется библиотека Qt;
+ warn_on означает, что компилятор должен выдавать как можно больше предупреждающих сообщений;
+ release указывает, что приложение должно быть откомпилировано в окончательном варианте, без отладочной информации.

Напоминание при реализации классов

При реализации файлы классов следует разбивать на две части. Часть определения класса помещается в файл **.h*, а реализация класса в файл **.cpp*. Важно помнить, что в заголовочном файле с определением класса должна содержаться директива 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.

В начале определения класса содержится макрос **Q_OBJECT** для **MOC** - это необходимо, если ваш класс использует сигналы и слоты, а в других случаях, этим макросом можно пренебречь. Но учитывайте, что из-за отсутствия метаинформации нельзя будет использовать приведение типа:

qobject _ cast<T> (obj)

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

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

Эту конструкцию можно так же заменить на эквивалентную с использованием pragma, тогда код заголовочного файла будет смотреться более компактно:

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

Основная программа должна быть реализована в отдельном файле, который является «стартовой площадкой» приложения. Такому файлу принято давать имя ***main.cpp***. Это удобно еще и потому, что проект может состоять из сотен файлов, и если следовать указанному правилу, то найти отправную точку всего проекта не составит труда.

Метообъектный компилятор MOC

Метаобъектый компилятор (*MOC, Meta Object Compiler*), является не компилятором, а препроцессором, который исполняется в ходе компиляции приложения, создавая, в соответствии с определением класса, дополнительный код на языке C++. Это происходит из-за того, что определения сигналов и слотов в исходном коде программы недостаточно для компиляции. Сигнально-слотовый код должен быть преобразован в код, понятный для компилятора C++. Код сохраняется в файле с прототипом имени: ***moc_<filename>.cpp***.

What is MOC?

Если вы работаете с файлами проекта, то о существовании *MOC* можете и не догадываться, ведь в этом случае управление *MOC* автоматизировано. Для создания *moc-файла* вручную можно воспользоваться следующей командой:

>moc -o proc.moc proc.h

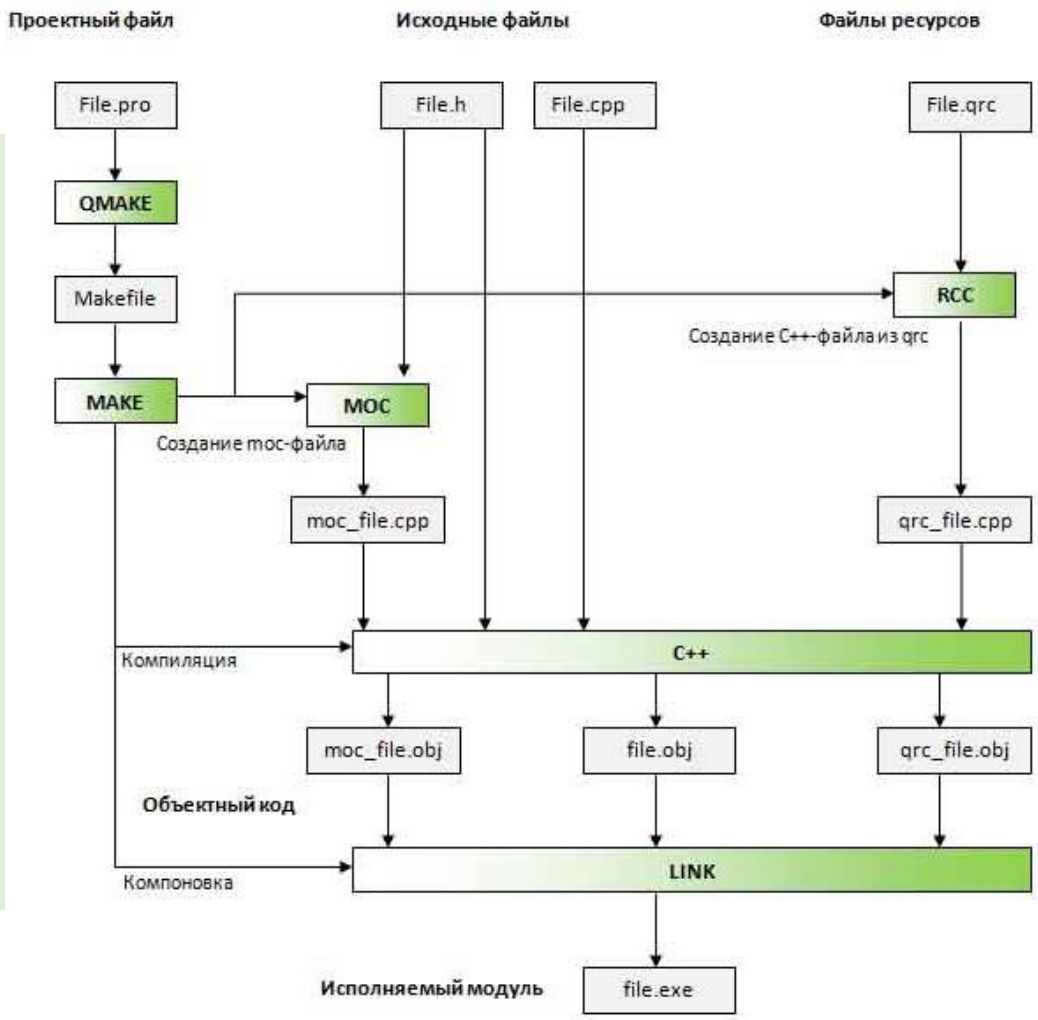
После ее исполнения *MOC* создаст дополнительный файл *proc.moc*

Для каждого класса, унаследованного от *QObject*, *MOC* предоставляет объект класса, унаследованного от ***QMetaObject***. Объект этого класса содержит информацию о структуре объекта - например, сигнально-слотовые соединения, имя класса и структуру наследования.

Структура Qt-проекта

Структура проекта Qt следующая - помимо *файлов исходного кода* на языке C++ обычно имеется *файл проекта* с расширением ****.pro***. Из него вызовом утилиты ***qmake*** и 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***.



#include <QtGlobal>

Qt содержит в заголовочном файле *QtGlobal* некоторые макросы и функции, которые могут быть очень полезны при написании программ. Шаблонные функции **qMax (a, b)** и **qMin (a, b)** используются для определения максимального и минимального из двух переданных значений:

```
int n = qMax<int>(3, 5); // n = 5
int n = qMin<int>(3, 5); // n = 3
```

Функция **qAbs (a)** возвращает абсолютное значение:

```
int n = qAbs(-5); // n = 5
```

Функция **qRound ()** округляет передаваемое число до целого:

```
int n = qRound(5.2); // n = 5
int n = qRound(-5.2); // n = -5
```

Функция **qBound ()** возвращает значение, находящееся между минимумом и максимумом:

```
int n = qBound(2, 12, 7); // n = 7
```

Функция для сравнение двух значений с плавающей точкой на точное равенство: **qFuzzyCompare()** берет в этом случае всю ответственность за правильное сравнение на себя. Она принимает два значения типа *double* или *float* и возвращает логическое значение *true*, если переменные считаются равными, в противном случае она возвращает значение *false*. Само сравнение осуществляется в относительной манере, когда точность для сравнения увеличивается с уменьшением численных значений сравниваемых величин. Поэтому единственное значение, которое представляет сложность для этой функции, - это нулевое значение. Но есть решение и для этой задачи. Нужно просто сделать так, чтобы сравниваемые значения были либо равны 1, либо больше 1.

```
double dVal1 = 0.0;
double dVal2 = myFunction();
if (qFuzzyCompare(1 + dVal1, 1 + dVal2)) {
    //Значения равны
}
```

Список типов qt

Здесь приведен список типов Qt, которые можно использовать при программировании.

Тип Qt	Эквивалент C++	Размер (биты)
qint8	signed char	8
quint8	unsigned char	8
qint16	short	16
quint16	unsigned short	16
qint32	int	32
quint32	unsigned int	32
qint64	int64 или long long	64
quint64	unsigned int64 или unsigned long long	64
qlonglong	То же самое, что и qint64	64
qulonglong	То же самое, что и quint64	64

Методы отладки ПО

Самый простой способ операции вывода в Qt - использование объекта класса **QDebug**. Этот объект очень напоминает стандартный объект потока вывода в C++ **cout**. Например, вывести сообщение в отладчике или на консоли с помощью функции *qDebug()* можно следующим образом:

qDebug() << "Test";

Важно понимать, что вывод информации с помощью функции *qDebug()* происходит при отладочных и релизных компоновках.

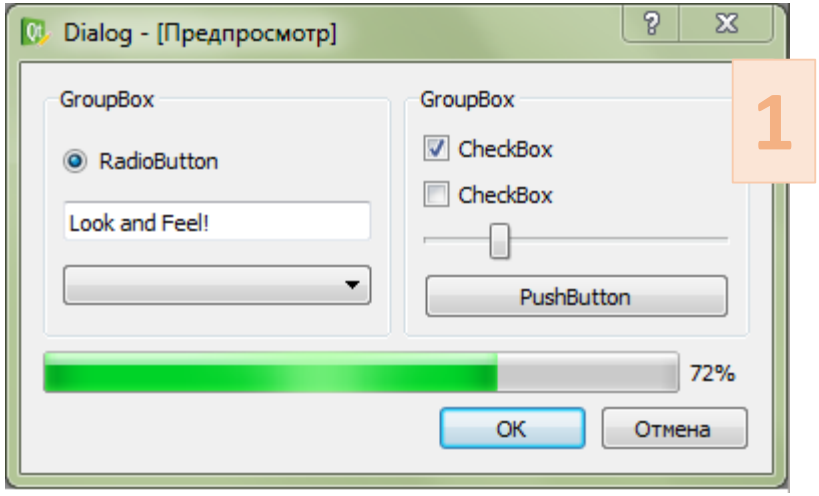
F5 – начать отладку;
Shift+F5 - закончить отладку
F10 - Строка кода может быть исполнена как одно целое
F11 - выполнить функцию или подфункцию

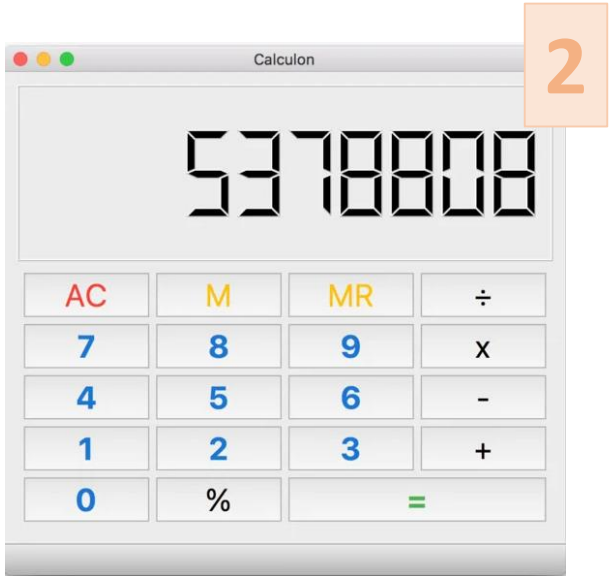
Точки остановки
Точки останова представляют место или набор мест в коде, которые при выполнении прервут отлаживаемую программу и передадут управление пользователю. Пользователь может просмотреть состояние прерванной программы или продолжить выполнение построчно или непрерывно.
Вы можете установить точки останова:
В конкретной строке на которой вы хотите остановить программу -- щёлкните на поле слева или нажмите F9.
На функции, в которой вы хотите прерывать программу - введите имя функции в “Установить точку останова” на функцию... в меню “Отладка”.
Вы можете удалить точку останова:
Щёлкнув на маркер точки останова в текстовом редакторе.
Выбрав точку останова в виде точек останова и нажав *Delete*.
Выбрав “Удалить точку останова” из контекстного меню точки останова в виде “Точки останова”.
Точки останова могут быть установлены и удалены перед тем как программа будет запущена или во время её работы под отладчиком. Также точки останова могут быть сохранены вместе с сессией.

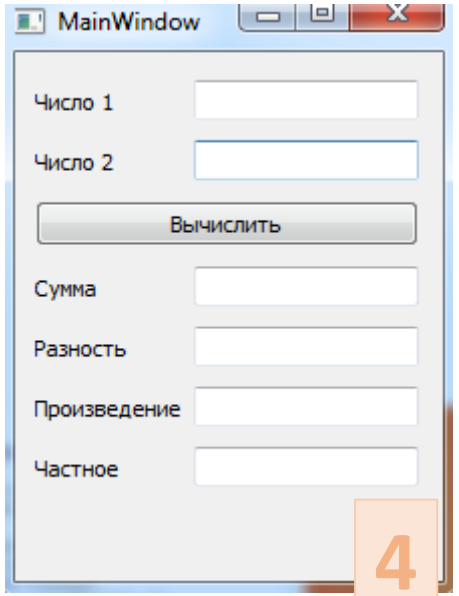
Практика

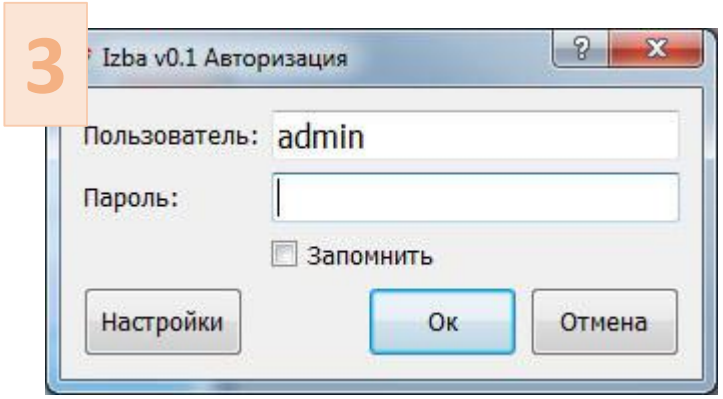
- 1. Создание проектного и исполняемого файла при помощь команд : *qmake, make*
- 2. Отладка ранее созданного ПО при помощи «точки останова» или вывода отладочной информации с использованием ф-ции: qDebug().
- 3. В проекте «SettingsWidgets» с прошлого занятия элементы управления поместить в объект QGridLayout.

Домашняя работа # 19









Задание:

Программно описать следующие GUI с использованием наследников класса QLayout.