

# Технология ООП

Санкт-Петербургский государственный политехнический университет

1 ноября 2011

qmake создает файл сборки, основываясь на информации в файле проекта.

qmake содержит дополнительные свойства для поддержки разработки с Qt, включая автоматическое создание правил для moc и uic.

qmake имеет два режима работы, в первом он генерирует проектные файлы (.pro), а во втором на основе проектных файлов генерирует правила сборки. Опции:

- -recursive
- -nodepend
- -nomoc
- -Wnone
- -Wall

В файле проекта, переменные используются для хранения списков строк. В простых проектах эти переменные информируют qmake о параметрах настройки, именах файлов и каталогах, которые используются в процессе сборки.

```
HEADERS = mywidget.h mylist.h \\  
         test.h
```

Оператор \$\$ используется для извлечения содержимого переменной и может быть использован для передачи значений между переменными или передачи их в функции:

```
EVERYTHING = $$SOURCES $$HEADERS
```

Специальный оператор `$$[...]` может быть использован для получения доступа к различным опциям конфигурирования, которые были установлены при сборке Qt:

```
message(Qt version: $$[QT_VERSION])
```

Для получения содержимого значения окружения при запуске `qmake` используйте оператор `$$(...)`:

```
DESTDIR = $$ (PWD) #выполняется на этапе обработки  
проекта
```

```
DESTDIR = $(PWD) #выполняется на этапе обработки  
makefile'a
```

Операторы:

- `variable = value`
- `variable += value`
- `variable -= value`
- `variable *= value`
- `variable = value`

`DEFINES += QT_DLL`

`CONFIG += gui`

Для добавления комментария в файл проекта используется символ `#`. Если необходимо вставить символ `#` как часть переменной, то нужно использовать переменную `LITERAL_HASH`.

```
urlPieces = http://qt.nokia.com/doc/4.0/qtextdocument.html  
pageCount  
message($$join(urlPieces, $$LITERAL_HASH))
```

Области видимости аналогичны операторам `if` в процедурных языках программирования. Если некоторое условие истинно, то объявления внутри области видимости обрабатываются.

---

```
<условие> {  
    <команда или определение>  
    ...  
}
```

---

Открывающая скобка должна находиться в той же строке, что и условие.

---

```
win32 {  
    SOURCES += paintwidget_win.cpp  
}
```

---

Области видимости допускают вложенность, а также сокращенную запись используя оператор «:». Значения, сохраненные в переменной CONFIG, специально обрабатываются для qmake. Например:

```
CONFIG += opengl
```

---

```
opengl {  
    TARGET = application-gl  
} else {  
    TARGET = application  
}
```

---

Простые циклы создаются с помощью перебора списка значений, используя встроенную функцию `for`. Следующий код добавляет каталоги в переменную `SUBDIRS`, но только в том случае, если они существуют:

---

```
EXTRAS = handlers tests docs
for(dir , EXTRAS) {
    exists($$dir) {
        SUBDIRS += $$dir
    }
}
```

---

Встроенные функции позволяют обрабатывать содержимое переменных. Эти функции обрабатывают переданные им аргументы и возвращают в качестве результата значение или список значений.

---

```
HEADERS = model.h
HEADERS += $$OTHER_HEADERS
HEADERS = $$unique(HEADERS)
win32:INCLUDEPATH += $$quote(C:/mylibs/extra
    headers)
```

---

Выполняются только при выполнении условия, например,  
`isEmpty(variablename):`

---

```
isEmpty( CONFIG ) {  
    CONFIG += qt warn_on debug  
}
```

---

# Пользовательские функции

```
defineReplace(headersAndSources) {  
  variable = $$1  
  names = $$eval($$variable)  
  headers =  
  sources =  
  
  for(name, names) {  
    header = ${name}.h  
    exists($$header) {  
      headers += $$header  
    }  
    source = ${name}.cpp  
    exists($$source) {  
      sources += $$source  
    }  
  }  
  return($$headers $$sources)  
}
```

Перечень переменных которые распознаются Qt:

- CONFIG
- DESTDIR
- FORMS
- HEADERS
- QT
- RESOURCES
- SOURCES
- TEMPLATE

Переменная `TEMPLATE` используется для определения типа проекта, который будет собран. Доступные типы проектов:

- `app`
- `lib`
- `subdirs`
- `vcapp`
- `vclib`
- `vcsubdirs`

Переменная CONFIG определяет параметры и возможности, которые должен использовать компилятор, и библиотеки, с которыми будет идти компоновка.

- release
- debug
- debug\_and\_release
- ordered
- warn\_on
- warn\_off
- qt
- thread
- x11

Если переменная CONFIG содержит значение qt, то qmake поддерживает приложения Qt.

- core
- gui
- network
- opengl
- sql
- svg
- xml
- qt3support

---

```
CONFIG += qt  
QT += network xml
```

---

Если в проекте используются не только библиотеки Qt, то дополнительные библиотеки для линковки, а также пути поиска заголовочных файлов можно добавить:

---

```
LIBS += -Ld:/stl/lib -lmath  
INCLUDEPATH = c:/msdev/include d:/stl/include
```

---

Мета-объектный компилятор(МОС) — программа, которая обрабатывает расширения C++ от Qt. Каждый класс использующий возможности мета-объектной системы должен быть обработан мос.

Инструмент мос читает заголовочный файл C++. Если он находит одно или более объявлений классов, которые содержат макрос Q\_OBJECT, то он порождает файл исходного кода C++, содержащий мета-объектный код для этих классов.

Обычно moc используется с входным файлом, содержащим такое объявление класса:

---

```
class MyClass : public QObject
{
    Q_OBJECT
public :
    MyClass(QObject *parent = 0);
    ~MyClass();
};
```

---

После чего будет сгенерирован файл .cpp на основании файла, например, если файл с объявлением был myclass.h, то на выходе получится moc\_myclass.cpp.

Мос не обрабатывает весь с++ из-за этого классы-шаблоны не могут содержать сигналы и слоты.

Вызов препроцессора `moc` происходит то вызова препроцессора языка, а значит и до выполнения подстановки директив компилятора, таких как `define`

Классы, унаследованные от `QObject`, могут использовать механизм свойств, поддерживаемый библиотекой Qt.

Свойства — это поля класса, который определяются специальным образом и благодаря которым к атрибутам объекта класса можно получить доступ извне (например, из Qt Script).

Для определения свойств используются директивы препроцессора:

---

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

- WRITE — запись значения
- RESET — сброс значения
- DESIGNABLE — отображение в инспекторе свойств
- SCRIPTABLE — доступность в Qt Script
- STORED — поддержка сериализации

# Пример задания свойства

---

```
class MyExample : public QObject {  
Q_OBJECT  
Q_PROPERTY(int firstProperty READ getFirstPropValue  
          WRITE setFirstPropValue)  
private:  
int m_firstProp;  
public:  
int getFirstPropValue() { return m_firstProp; }  
void setFirstPropValue(int value) { m_firstProp =  
    value; }  
};
```

---

## Пример задания свойства

---

```
myExample->setProperty("firstProperty", 10);  
int propValue = myExample->property("firstProperty"  
    ).toInt();
```

---

Класс QString представляет собой строку символов Unicode. QString хранит строку 16-битных QChar, где каждый QChar хранит символ Unicode 4.0.

---

```
const QChar * constData () const  
bool isEmpty () const  
bool isNull () const  
int length () const  
QChar at ( int i ) const  
QByteArray toAscii () const  
...
```

---

---

```
QString str = "Hello";
QString str1;
if (str == "test1" || str == "Hello" )
{ ... }
```

```
static const QChar data[4] = { 0x0055, 0x006e, 0
    x10e3, 0x03a3 };
QString str(data, 4);
```

```
QString str = "and";
str.prepend("rock_");           // str == "rock and"
str.append("_roll");           // str == "rock and
    roll"
str.replace(5, 3, "&");         // str == "rock & roll"
```

---

Qt также предоставляет класс `QByteArray` для хранения произвольных байтов и 8-битных оканчивающихся на `'\0'` строк.

---

```
const char * constData () const  
bool isEmpty () const  
bool isNull () const  
int length () const  
char at ( int i ) const  
int indexOf ( char ch, int from = 0 ) const  
int lastIndexOf ( const QByteArray & ba, int from =  
    -1 ) const  
bool contains ( const QByteArray & ba ) const
```

---

Список строк, аналог конструкции `QList<QString>`.

---

```
QStringList fonts;  
    fonts << "Arial" << "Helvetica" << "Times" <<  
        "Courier";  
  
    for (int i = 0; i < fonts.size(); ++i)  
        cout << fonts.at(i).toLocal8Bit().  
            constData() << endl;  
  
    QStringList::const_iterator constIterator;  
    for (constIterator = fonts.constBegin();  
        constIterator != fonts.constEnd();  
        ++constIterator)  
        cout << (*constIterator).toLocal8Bit().  
            constData() << endl;  
  
    QString str = fonts.join(",");  
    QStringList list;  
    list = str.split(",");
```