

# Junior C++ developer

## Лекция 13

- qmake

# qmake

**Qmake** - утилита из состава **Qt**, которая помогает облегчить процесс сборки приложения на разных платформах. qmake автоматически генерирует **make-файлы**, основываясь на информации в файлах проекта (\*.pro).

Создан **Trolltech** специально для кроссплатформенного управления проектами с упором на **Qt**.

Название «qmake» несколько вводит в заблуждение, т.к. сам qmake ничего не билдит, он только подготавливает сборку под конкретный тулчейн. Идея заключается в том, чтобы на основании одного файла проекта qmake (\*.pro) можно было автоматически создать необходимый makefile или его аналог для каждого сочетания платформы и компилятора; дальше проект собирается уже средствами конкретного тулчейна.

# Qmake +

- очень хорошая интеграция в Qt Creator;
- можно управлять проектами, не используя Qt;
- с помощью скриптовых средств qmake можно реализовать весьма продвинутый функционал для управления проектами;
- проверен временем;
- легко пользоваться в случае простых проектов;
- существуют тонны исходников, использующих qmake для построения — так что qmake все равно нужен.

# Qmake -

- неполная и посредственно написанная документация;
- реализовать даже самый простенький, но нетривиальный функционал — действительно нетривиальная задача;
- закрученный и свой собственный синтаксис скриптовых средств, далекий от симпатичного и удобного.

# Переменные в qmake

- Имя переменной — обычный идентификатор, регистр имеет значение. Объявляются переменные неявно, с помощью присваивания значения.
- Значение переменной - это список значений, разделенных пробелами.
  - Количество пробелов между значениями в списке не важно.
  - Если значение содержит пробел, то его нужно заключить в двойные кавычки.
- В списке значений можно использовать значения других переменных с помощью оператора **\$\$**

```
VAR = value1 value2 # комментарий - от решетки до конца строки
VAR = value1 "value with spaces"
```

```
PREFIX = lib
VAR1 = value1 value2
VAR2 = v3 $$VAR1 ${PREFIX}v4 # v3 value1 value2 libv4
# {} нужны в том случае, если имя переменной без них
# будет неправильно определяться.
# $$PREFIXv4 молча даст пустую строку, т.к.
# если переменная не была определена ранее,
# ее значение полагается пустым.
```

# Операции с переменными

```
VAR = v1 v2 v3      # заменяет список:      v1 v2 v3
VAR += v3 v4         # добавляет к списку:    v1 v2 v3 v3 v4
VAR -= v3            # убирает из списка:     v1 v2 v4
VAR *= v4 v5 v5      # добавляет, если нет:  v1 v2 v4 v5
VAR += v5            # v1 v2 v4 v5 v5
VAR ~= s/v5/vvv     # заменяет по regexp:   v1 v2 v4 vvv v5
```

# Переменные исходных файлов

## HEADERS

Заголовочные файлы проекта (\*.h). Заголовочные файлы не компилируются сами по себе, но для них вызывается `mos`. Также для них генерируются зависимости в `Makefile`, так что если некий заголовочный файл изменится, то будут перекомпилированы все файлы, его включающие.

## SOURCES

Файлы, компилируемые `c/c++` компилятором (\*.c, \*.cpp, \*.cxx и т.п.). Естественно, список не включает в себя автоматически генерируемые файлы от `mos`, `uic` и т.п.

Рассмотрим простой проект состоящий из 3-х файлов:

- hello.cpp
- hello.h
- main.cpp

Заполним переменную **SOURCES** одним из способов:

1. `SOURCES += hello.cpp`  
`SOURCES += main.cpp`
2. `SOURCES = hello.cpp \`  
`main.cpp`

Заполним переменную **HEADERS**:

`HEADERS += hello.h`

Итоговый вариант:

```
HEADERS += hello.h
SOURCES += hello.cpp
SOURCES += main.cpp
```



# Debuggable

В релизной версии приложения удаляются все дебажные символы и отладочная информация.

Для включения поддержки отладки необходимо в .pro файл добавить

```
CONFIG += debug
```

# Платформа-зависимые файлы

```
CONFIG += debug
```

```
HEADERS += hello.h
```

```
SOURCES += hello.cpp
```

```
SOURCES += main.cpp
```

```
win32 {
```

```
    SOURCES += hellowin.cpp #файл подключиться в сборке для Windows
```


```
}
```

```
unix {
```

```
    SOURCES += hellounix.cpp #файл подключиться в сборке для Unix-подобных ОС
```

```
}
```

Открывающиеся скобки обязательно должны быть на строке с условием



# Существование файла

```
!exists( main.cpp ) {  
    error( "No main.cpp file found" )  
}
```

Если файла не существует,  
то сборка остановится

# Вложенные условия

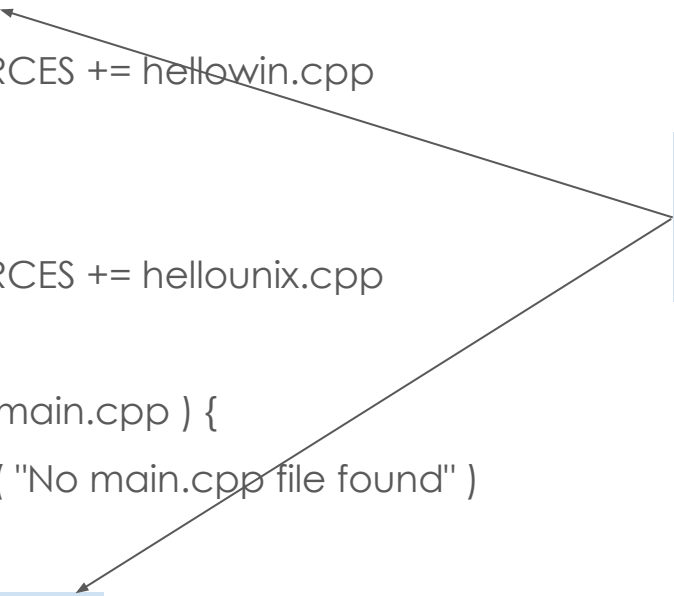
```
win32 {  
    debug {  
        CONFIG += console  
    }  
}
```

# Все вместе

```
CONFIG += debug  
HEADERS += hello.h  
SOURCES = hello.cpp main.cpp
```

```
win32 {  
    SOURCES += hellowin.cpp  
}  
unix {  
    SOURCES += hellounix.cpp  
}  
!exists( main.cpp ) {  
    error( "No main.cpp file found" )  
}  
win32:debug {  
    CONFIG += console  
}
```

Ещё один способ  
организовать вложенные  
условия



# Подключение библиотек

Для того чтобы подключить библиотеку необходимо :

- указать путь до библиотеки

```
LIBS += -L/usr/local/lib -lmath
```

- указать пути, содержащие заголовочные файлы библиотеки.

```
INCLUDEPATH = c:/msdev/include d:/stl/include
```

# QDebug

Класс **QDebug** предоставляет выходной поток для отладочной информации.

**QDebug** используется когда разработчику необходимо записать отладочную или трассировочную информацию на устройство, файл, в строку или консоль.

## ... QWarning(), QCritical(), QFatal

```
qWarning() << "Warning";  
qCritical() << "Critical";  
qFatal("Fatal");
```

С помощью этих событий, можно разделить ошибки по уровням значимости и применить фильтры, для разделения того, какие ошибки нужно выводить, а какие нет.



# Перенаправление

Можно перенаправить поток вывода сообщений об ошибках в текстовый файл и получить лог программы.

Для перенаправления сообщений об ошибках в текстовый файл, вам необходимо установить **CallBack-функцию** обработчик в приложение. Для этого используется функция ***qInstallMessageHandler*** .