кроссплатформенный фреймворк для разработки программного обеспечения на языке программирования C++

Проблемы, направления

- Промышленные/научные приложения (c/c++, аппаратура, embedded)
- Нативные приложения
- Разработка Gui MFC и тд.
- Веб должен умереть (проблемы с безопасностью https://habrahabr.ru/post/338880/)
- Кроссплатформенность

Qt представляет собой полный инструментарий для программирования, который состоит из отдельных модулей и предоставляет:

- поддержку двух- и трехмерной графики (фактически, являясь стандартом для платформонезависимого программирования на OpenGL);
- возможность интернационализации, которая позволяет значительно расширить рынок сбыта ваших программ;
- использование формата XML (extensible Markup Language);
- STL-совместимую библиотеку контейнеров;
- поддержку стандартных протоколов ввода/вывода;
- классы для работы с сетью;
- поддержку программирования баз данных, включая Oracle, Microsoft SQL Server, IBM DB2, MySQL, SQLite, Sybase, PostgreSQL;
- и многое другое.

«привязки» графического фреймворка Qt

- Python — PyQt, PySide
- Ruby — QtRuby
- Java — Qt Jambi
- PHP — PHP-Qt
- и другие.

PHP-Qt:

```
<?php
  $widget = new QWidget;
  $widget->show();
  qApp::exec();
```

Использование

Примеры продуктов

- Autodesk Maya,
- Skype
- Telegram
- Медиапроигрыватель VLC
- VirtualBox
- Google Планета Земля
- Mathematica
- 2GIS
- Viber
- KDE

Корпорации:

- European Space Agency
- DreamWorks
- Google
- HP
- Lucasfilm
- Panasonic
- Philips
- Samsung
- Siemens
- Volvo
- Walt Disney Animation Studios

Модуль *QtCore*

- контейнерные классы: **QList, QVector, QMap, QVariant, QString** и т. д.
- классы для ввода и вывода: **QiODevice, QTextStream, QFile**
- классы процесса **QProcess** и для программирования многопоточности: **QThread, QWaitCondition, QMutex**
- классы для работы с таймером: **QBasicTimer** и **QTimer**
- классы для работы с датой и временем: **QDate** и **QTime**
- класс **QObject,** являющийся *краеугольным камнем* объектной модели Qt
- базовый класс событий **QEvent**
- класс для сохранения настроек приложения **Qsettings**
- класс приложения **QCoreApplication,** из объекта которого, если требуется, можно запустить цикл событий
- классы поддержки анимации: **QAbstractAnimation, QVariantAnimation** и т. д.
- классы для машины состояний: **QStateMachine, QState** и т. д.
- классы моделей интервью: **QAbstractItemModel, QStringListModel, QAbstractProxyModel**
- модуль содержит так же механизмы поддержки файлов ресурсов

Объект класса `QCoreApplication` :

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

Модуль *QtGui*

- Предоставляет классы интеграции с оконной системой, с OpenGL и OpenGL ES.
- Содержит класс QWindow, который является элементарной областью с возможностью получения событий пользовательского ввода, изменения фокуса и размеров, а так же позволяющей производить графические операции и рисование на своей поверхности.
- Класс приложения этого модуля QGuiApplication. Этот класс содержит механизм цикла событий
- получения доступа к буферу обмена
- инициализации необходимых настроек приложения — например, палитры для расцветки элементов управления

Модуль *QtWidgets*

- класс Qwidget — это базовый класс для всех элементов управления библиотеки Qt
- классы для автоматического размещения элементов: QVBoxLayout,QHBoxLayout
- классы элементов отображения: QLabel,QLCDNumber
- классы кнопок: QPushButton,QCheckBox,QRadioButton
- классы элементов установок: QSlider,QScrollBar
- классы элементов ввода: QLineEdit,QSpinBox
- классы элементов выбора: QComboBox,QTooiBox
- классы меню: QMainwindow и QMenu
- классы окон сообщений и диалоговых окон: QMessageBox,Qdialog
- классы для рисования: QPainter,QBrush,QPen,QColor
- классы для растровых изображений: Qimage,QPixmap
- классы стилей отдельному элементу, так и всему приложению может быть присвоен определенный стиль, изменяющий их внешний облик;
- класс приложения QApplication,который предоставляет цикл событий.

Модули Qt

Библиотека	Назначение
QtCore	Основополагающий модуль, состоящий из классов, не связанных с графическим интерфейсом
QtGui	Модуль базовых классов для программирования графического интерфейса
QtWidgets	Модуль, дополняющий QtGui «строительным материалом» для графического интерфейса в виде виджетов на С++
QtQuick	Модуль, содержащий описательный фреймворк для быстрого создания графического интерфейса
QtQML	Модуль, содержащий движок для языка QML и JavaScript
QtNetwork	Модуль для программирования сети
Qt0penGL	Модуль для программирования графики OpenGL
QtSql	Модуль для программирования баз данных
QtSvg	Модуль для работы с SVG (ScalaЬle Vector Graphics, масштабируемая векторная графика)

Библиотека	Назначение
QtXml	Модуль поддержки XM L, классы, относящиеся к SAX и DOM
QtXmlPatterns	Модуль поддержки XPath , Query, XSLT и XmlShemaValidator
QtScript (deprecated)	Модуль поддержки языка сценариев
QtScriptTools	Модуль дополнительных возможностей поддержки языка сценария. В настоящее время предоставляет отладчик
QtMultimedia	Модуль мультимедиа
QtMultimediaWidgets	Модуль с виджетами для QtMultimedia
QtWebKit	Модуль для создания WeЬ-приложений
QtWebKitWidgets	Модуль с виджетами для QtWebKit
QPrintSupport	Модуль для работы с принтером
QtTest	Модуль, содержащий классы для тестирования кода

Философия объектной модели

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.

Свойства

```
Q_PROPERTY(type name
    (READ getFunction [WRITE setFunction] |
     MEMBER memberName [(READ getFunction | WRITE setFunction)])
    [RESET resetFunction]
    [NOTIFY notifySignal]
    [REVISION int]
    [DESIGNABLE bool]
    [SCRIPTABLE bool]
    [STORED bool]
    [USER bool]
    [CONSTANT]
    [FINAL])
```

Условные обозначения:

() – обязательное поел

[] – опциональное поле

| – один из варинатов

```cpp
class Hero : public QObject
{
    Q_OBJECT
    Q_PROPERTY(int health READ healt WRITE sethealt) //alt+enter
private:
    int m_health;

public:
    explicit Hero(QObject *parent = nullptr);
    int healt() const
    {
        return m_health;
    }

public slots:
//особый метод для проверки
    void sethealt(int health)
    {
        m_health = health;
    }
};
```

Использование:

```cpp
Hero hr;
hr.sethealt(100);
hr.healt();
```

Если не нужен собственные «Геттер» и «Сеттер»

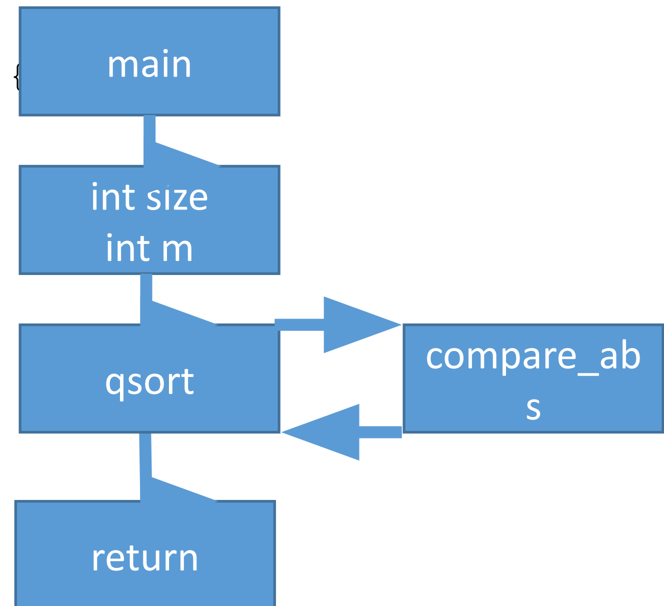
```
class Hero : public QObject
{
    Q_OBJECT
    Q_PROPERTY(int health MEMBER m_health)
public:
    explicit Hero(QObject *parent = nullptr);
private:
     int m_health;
};
```

```
Использование:
  Hero hr;
  hr.setProperty("health",100);
  hr.property("health");
```

Механизм сигналов и слотов

Что было до …

```
#include <stdlib.h>

int compare_abs(const void *a, const void *b) {
    int a1 = *(int*)a;
    int b1 = *(int*)b;
    return abs(a1) - abs(b1);
}
int main() {
    int size = 10;
    int m[size] = {1, -3, 5, 33, 44};
    // сортировка массива m по возрастанию
модулей
    qsort(m, size, sizeof(int), compare_abs);
    return 0;
}
```

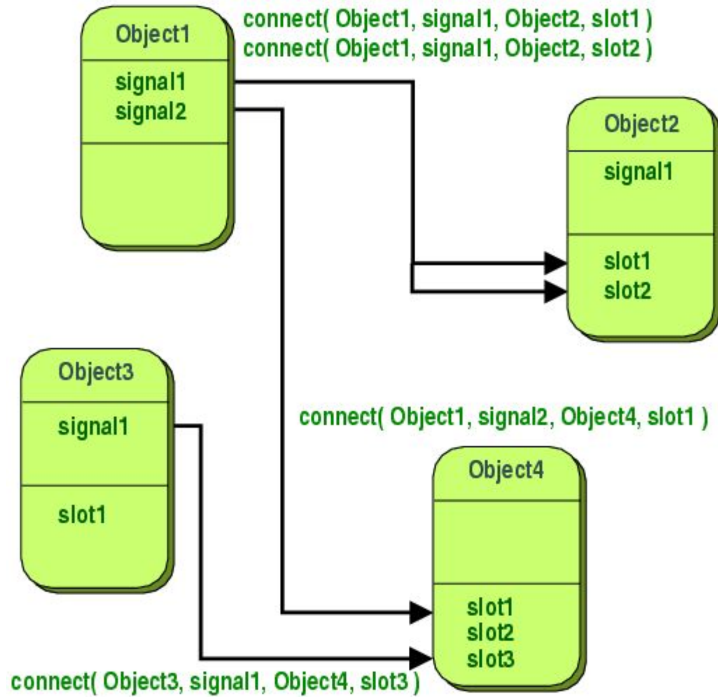


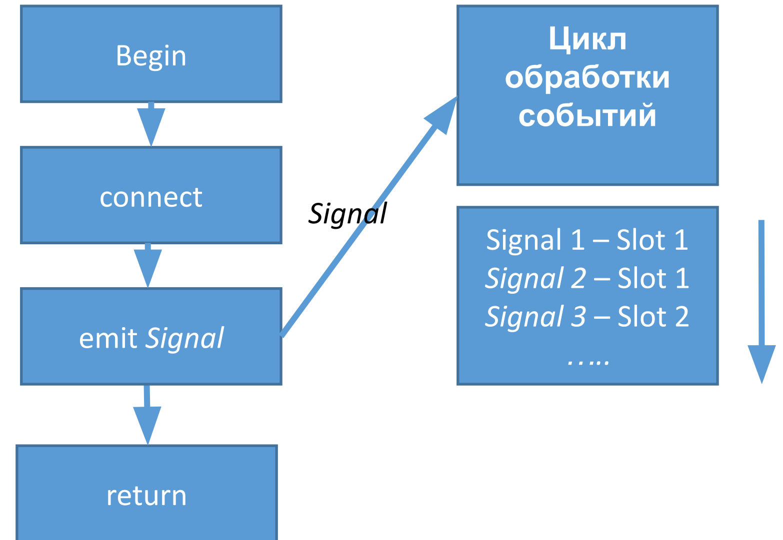
Сейчас в С++ ввели Lambda

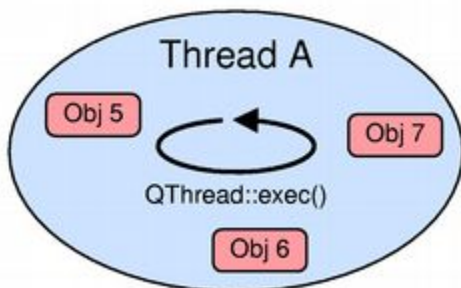
MFC. Карты сообщений

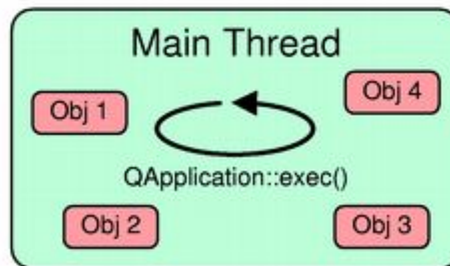
```
class CPhotoStylerApp : public CWinApp {
public:
    CPhotoStylerApp();
public:
    virtual BOOL InitInstance();
    afx_msg void OnAppAbout();
    afx_msg void OnFileNew));
    DECLARE_MESSAGE_MAP()
};

BEGIN_MESSAGE_MAP(CPhotoStylerApp, CWinApp)
    ON_COMMAND(ID_APP_ABOUT, OnAppAbout)
    ON_COMMAND(ID_FILE_NEW, OnFileNew)
    ON_COMMAND(ID_FILE_NEW, CWinApp::OnFileNew)
    ON_COMMAND(ID_FILE_OPEN, CWinApp::OnFileOpen)
    ON_COMMAND(ID_FILE_PRINT_SETUP, CWinApp::OnFilePrintSetup)
END_MESSAGE_MAP()
```

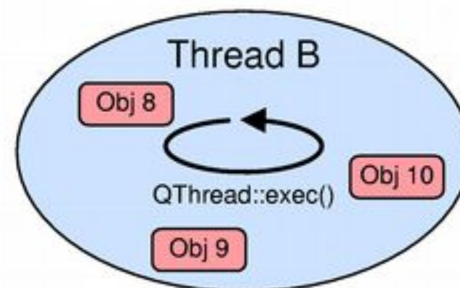
Механизм сигналов и слотов











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.

Недостатки

- 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, который осуществляется при использовании механизма функций обратного вызова;
- 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)

Сигналы

Описание:

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

```
class MySignal : public QObject {
    Q_OBJECT
public:
    void sendSignal()
    {
        if (выражение)
            emit sendString("Info");
    }
signals:
    void sendString(const QStrings);
};
```

Реализация сигнала генерируемая MOC

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

Слоты

```
class MySlot : public QObject {
    Q_OBJECT
public:
    MySlot();
public slots:
    void slot()
    {
        qDebugO « "I'm a slot";
        qDebug() « sender()->objectName( );
    }
};
```

- нельзя использовать параметры по Умолчанию — например: slotMethod(int п = 8)
- определять слоты как static
- слоты по умолчанию public. Если нужно ограничить их вызов как метода, то объявляем private\protected

Соединение объектов

Вид 1:
```
QObject::connect(const QObject* sender,
                 const char* signal,
                 const QObject* receiver,
                 const char* slot,
                 Qt::ConnectionType type = Qt:Autoconnection);
```

Пример:
```
QObject::connect(pSender, SIGNAL(signalMethod( )),pReceiver, SLOT(slotMethod( )));
```

Вид 2:
```
QObject::connect(const QObject* sender,
                 const QMetaMethods signal,
                 const QObject* receiver,
                 const QMetaMethods slot,
                 Qt::ConnectionType type = Qt:Autoconnection);
```

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

Метод `connect ( )` после вызова возвращает объект класса `Connection`

Вид 1 – выявление ошибок соединения на этапе выполнения

Вид 2 – выявление ошибок соединения на этапе компиляции

```
#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()) , scounter, SLOT (slotInc()));
    QObject: :connect (&counter, SIGNAL(counterChanged(int)),&lbl,SLOT(setNum(int)
    QObject: :connect (&counter, SIGNAL(goodbye()), &app, SLOT(quit()));
    return app.exec ( ); // запуск бесконечного цикла обработки сообщений
}
```

```
#include <QObject>
class Counter : public QObject {
    Q_OBJECT
private:
    int m_nValue;
public:
    Counter();
public slots:
    void slotlnc0;
signals:
    void goodbye ( );
    void counterChanged(int);
};
```

```
void Counter::slotlnc()
{
    emit counterChanged(++m_nValue);
    if (m_nValue == 5) {
        emit goodbye();
    }
}
```

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

- Автоматическое соединение (Auto Connection) (по умолчанию) Если сигнал испускается в потоке, с которым объект-получатель имеет родство, то поведение такое же, как и при прямом соединении. В противном случае, поведение аналогично соединению через очередь."
- Прямое соединение (Direct Connection) Слот вызывается немедленно при отправке сигнала. Слот выполняется в потоке отправителя, который не обязательно является потоком-получателем. (**аналогично**: object ->called_SLOT())
- Соединение через очередь (Queued Connection) Слот вызывается, когда управление возвращается в цикл обработки событий в потоке получателя. Слот выполняется в потоке получателя.
- Блокирующее соединение через очередь (Blocking Queued Connection) Слот вызывается так же, как и при соединении через очередь, за исключением того, что текущий поток блокируется до тех пор, пока слот не возвратит управление. **Замечание:** Использование этого типа подключения объектов в одном потоке приведет к взаимной блокировке.
- Уникальное соединение (Unique Connection) Поведение такое же, что и при автоматическом соединении, но соединение устанавливается только если оно не дублирует уже существующее соединение. т.е., если тот же сигнал уже соединён с тем же самым слотом для той же пары объектов, то соединение не будет установлено и connect() вернет false.

Разъединение объектов

- disconnect
- Уничтожение связи при уничтожении объекта (связь это тоже **объект** принадлежащий 2м «родителям»)

Общий вид:

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

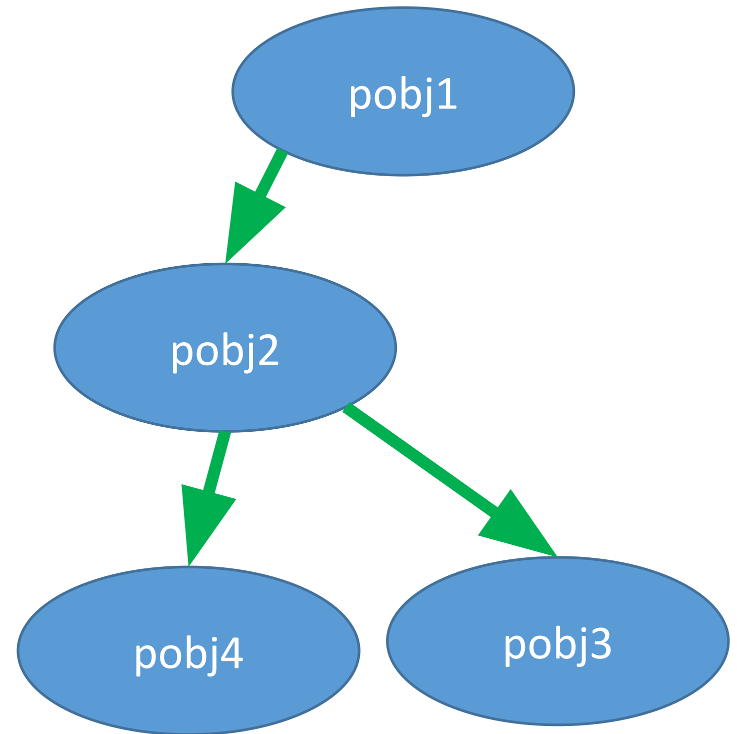
Сокращённые варианты:

- `disconnect(signal, receiver, slot) // сигнал в текущем объекте, слот в другом объекте(receiver)`
- `disconnect(receiver, slot) // сигнал и слот в текущем объекте`

Объектные иерархии

Задача: корректное уничтожение динамически создаваемых объектов без утечки памяти

```
QObject* pobj1 = new QObject;
QObject* pobj2 = new QObject (pobj1) ;
QObject* pobj4 = new QObject (pobj2) ;
QObject* pobj3 = new QObject (pobj2) ;

pobj2->setObjectName ("the first child of pobj1") ;
pobj3->setObjectName ("the first child of pobj2") ;
pobj4->setObjectName ("the second child of pobj2");
```

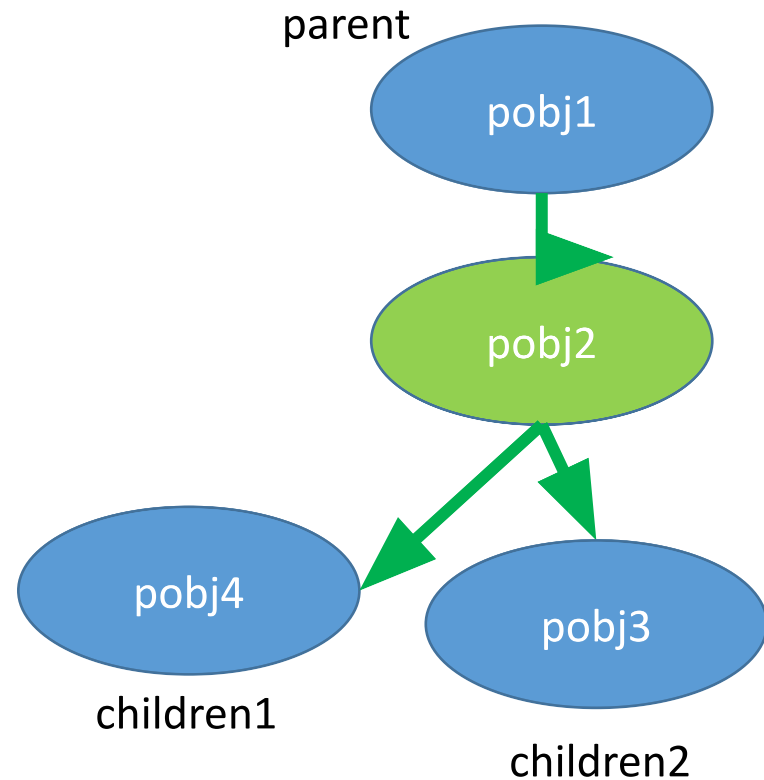


При уничтожении созданного объекта (при вызове его деструктора) **все** присоединенные к нему объекты-потомки **уничтожаются автоматически**.

Альтернативные решения: Smart pointers (Умные указатели)

Перемещение по иерархии объектов:

- children() – Список потомков
- parent() – 1 родитель



Вывод иерархии:

```
pobj1-> dumpObjectTree();
```

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

Поиск потомка:

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

Поиск потомков по шаблону:

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

Метаобъектная информация (moc)

MOC (Meta Object Compiler) генерирует код, инициализирующий мета-объект.

Мета-объект содержит:

- имя класса объекта
- имя класса родителя
- имена всех сигналов
- слотов
- указатели на их функции
- указатели и названия свойств

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

Альтернатива: Метаклассы в C++17 (отказ от MOC, C++/CLI и C++/CX)

Кодогеренация в C++17

```
$class interface {
    constexpr
    {
        compiler.require($interface.variables().empty(),
                         "Никаких данных-членов в интерфейсах!");
        for (auto f : $interface.functions())
        {
            compiler.require(!f.is_copy() && !f.is_move(),
          "Интерфейсы нельзя копировать или перемещать; используйте"
                             "virtual clone() вместо этого");
            if (!f.has_access())
                f.make_public(); // сделать все методы публичными!
            compiler.require(f.is_public(), "interface functions must be public");
               // проверить, что удалось
            f.make_pure_virtual();  // сделать метод чисто виртуальным
        }
    }
// наш интерфейс в терминах C++ будет просто базовым классом,
    // а значит ему нужен виртуальный деструктор
    virtual ~interface() noexcept { }
};
```

Зачем Qmake

Сборка простенькой программы из 2х файлов:

g++ -c -fno-keep-inline-dllexport -pipe -g -std=gnu++11 -Wextra -Wall -W -fexceptions -mthreads -DUNICODE -DQT_DEPRECATED_WARNINGS -DQT_QML_DEBUG -DQT_CORE_LIB -I../Test_QT_console -I. -IC:/Qt/5.9/mingw53_32/include -IC:/Qt/5.9/mingw53_32/include/QtCore -Idebug -IC:/Qt/5.9/mingw53_32/mkspecs/win32-g++ -o debug/main.o ../Test_QT_console/main.cpp

g++ -c -fno-keep-inline-dllexport -pipe -g -std=gnu++11 -Wextra -Wall -W -fexceptions -mthreads -DUNICODE -DQT_DEPRECATED_WARNINGS -DQT_QML_DEBUG -DQT_CORE_LIB -I../Test_QT_console -I. -IC:/Qt/5.9/mingw53_32/include -IC:/Qt/5.9/mingw53_32/include/QtCore -Idebug -IC:/Qt/5.9/mingw53_32/mkspecs/win32-g++ -o debug/test_op.o ../Test_QT_console/test_op.cpp

g++ -Wl,-subsystem,console -mthreads -o debug/Test_QT_console.exe debug/main.o debug/test_op.o -LC:/Qt/5.9/mingw53_32/lib C:/Qt/5.9/mingw53_32/lib/libQt5Cored.a

Идея: автоматизируем! –> make, makefile

Makefile

Элементарный makefile

```
hellomake: hellomake.c hellofunc.c
    g++ -o hellomake hellomake.c hellofunc.c -I.
```

Преимущества:

- Экономия времени на написание команд для сборки проекта (1 команда - make)
- Отслеживание и перекомпиляция только изменившихся объектов
- Кроссплатформенная сборка

Задачи:

- Универсальность
- Кроссплатформенность
- Нужно БОЛЬШЕ ~~(золота, рабов)~~ АВТОМАТИЗАЦИИ

Решение:
Генерация makefile для новых проектов исходя из настроек по умолчанию для текущего окружения/платформы

Qmake, qbs

«Универсальный» makefile в рамках конкретного окружения и НЕ сложного проекта:

```
IDIR =../include
CC=g++
CFLAGS=-I$(IDIR)
ODIR=obj
LDIR =../lib
LIBS=-lm

_DEPS = hellomake.h
DEPS = $(patsubst %,$(IDIR)/%,$(_DEPS))

_OBJ = hellomake.o hellofunc.o
OBJ = $(patsubst %,$(ODIR)/%,$(_OBJ))


$(ODIR)/%.o: %.c $(DEPS)
        $(CC) -c -o $@ $< $(CFLAGS)

hellomake: $(OBJ)
        gcc -o $@ $^ $(CFLAGS) $(LIBS)

.PHONY: clean

clean:
    rm -f $(ODIR)/*.o *~ core $(INCDIR)/*~
```

Загляните в: **build-MyFirstUnit-Desktop_Qt_5_9_0_MinGW_32bit\Makefile**

Qmake

Этапы сборки:

1. Автоматическое создание файла настроек проекта (NameOfProject.pro**)**
qmake -project

2. Автоматическое создание Makefile
qmake NameOfProject.pro -o Makefile

3. Компиляция проекта/программы
make

Содержание NameOfProject.pro

```
QT       += testlib
QT       -= gui

TARGET = tst_myfirstunittest
CONFIG   += console
CONFIG   -= app_bundle

TEMPLATE = app

SOURCES +=
tst_myfirstunittest.cpp \
    myclass.cpp

HEADERS += \
    myclass.h
```

Опция	Назначение
HEADERS	Список созданных заголовочных файлов
SOURCES	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
TARGET	Имя приложения. Если это поле не заполнено, то название программы будет соответствовать имени проектного файла
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 debugger в исполняемый модуль
INCLUDEPATH	Путь к каталогу, где содержатся заголовочные файлы. Этой опцией можно воспользоваться в 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	Задает имена подкаталогов , которые содержат pro-файлы
TEMPLATE	Задает разновидность проекта. Например: app - приложение , lib - библиотека, subdirs - подкаталоги

Критика make

- Традиционная для UNIX система сборки make выдает непозволительно долгое время сборки для больших проектов (особенно для инкрементальных сборок);
- Использование рекурсивого вызова make из корневой директории приводит к нарушениям зависимостей между модулями, для устранения которого приходится запускать сборку два или три раза (и увеличение времени сборки, естественно);
- Портируемость make сильно раздута, так как make работает не сам по себе, в makefile могут быть указаны десятки других утилит, которые специфичны для платформы;
- Неочевидные и откровенно ущербные элементы языка (чего стоят только ifeq, ifneq, ifdef, и ifndef)
- Проблемы типов (работа только со строками)
- Плохое разрешение зависимостей в случае изменения опций командной строки
- Проблемы с одновременным запуском и распараллеливанием
- Легко «убить» дерево сборки внезапной остановкой программы.

Критика Qmake

- Свой, самобытный синтаксис языка.
- Отсутствие сборки как таковой в qmake. Qmake не собирает ваш проект. Название обманчиво. Он лишь генерирует Makefile, а сборку осуществляет одна из утилит Make (*nix — gnu make, win-nmake, jom или mingw-make).
- Отсутствие прозрачной поддержки кросс-компиляции, сборки пакетов и деплоймента.

```
DEFINES += QT_NO_CAST_TO_ASCII
!macx:DEFINES += QT_USE_FAST_OPERATOR_PLUS QT_USE_FAST_CONCATENATION

unix {
    CONFIG(debug, debug|release):OBJECTS_DIR = $${OUT_PWD}/.obj/debug-shared
    CONFIG(release, debug|release):OBJECTS_DIR = $${OUT_PWD}/.obj/release-shared
    CONFIG(debug, debug|release):MOC_DIR = $${OUT_PWD}/.moc/debug-shared
    CONFIG(release, debug|release):MOC_DIR = $${OUT_PWD}/.moc/release-shared
    RCC_DIR = $${OUT_PWD}/.rcc
    UI_DIR = $${OUT_PWD}/.uic
}
```

Qbs(Qt Build System)

- Декларативность — используется QML/JavaScript синтаксис;
- Расширяемость — есть возможность писать свои модули для сборки, либо кодогенераторы и т. п.
- Скорость;
- Сборка напрямую — это не qmake, это не «посредник». Qbs сама вызывает компиляторы, компоновщик, и что там еще понадобится.

Файл сборки простейшего приложения

```
import qbs
Product {
    type: ["application"]
    Depends { name: "cpp" }
    files: ["main.cpp"]
}
```

Использование JavaScript

```
---helpers.js---
function planetsCorrectlyAligned()
{
    // implementation
    return true;
}

---myproject.qbs---
import qbs 1.0
import "helpers.js" as Helpers

Product {
    name: "myproject"
    Group {
        condition:
Helpers.planetsCorrectlyAligned()
        file: "magic_hack.cpp"
    }
}
```

- Метаобъектный компилятор
    - Препроцессор
    - Кодогенерация(автоматизирует создание кода из макросов) Пример: сигналы/слоты

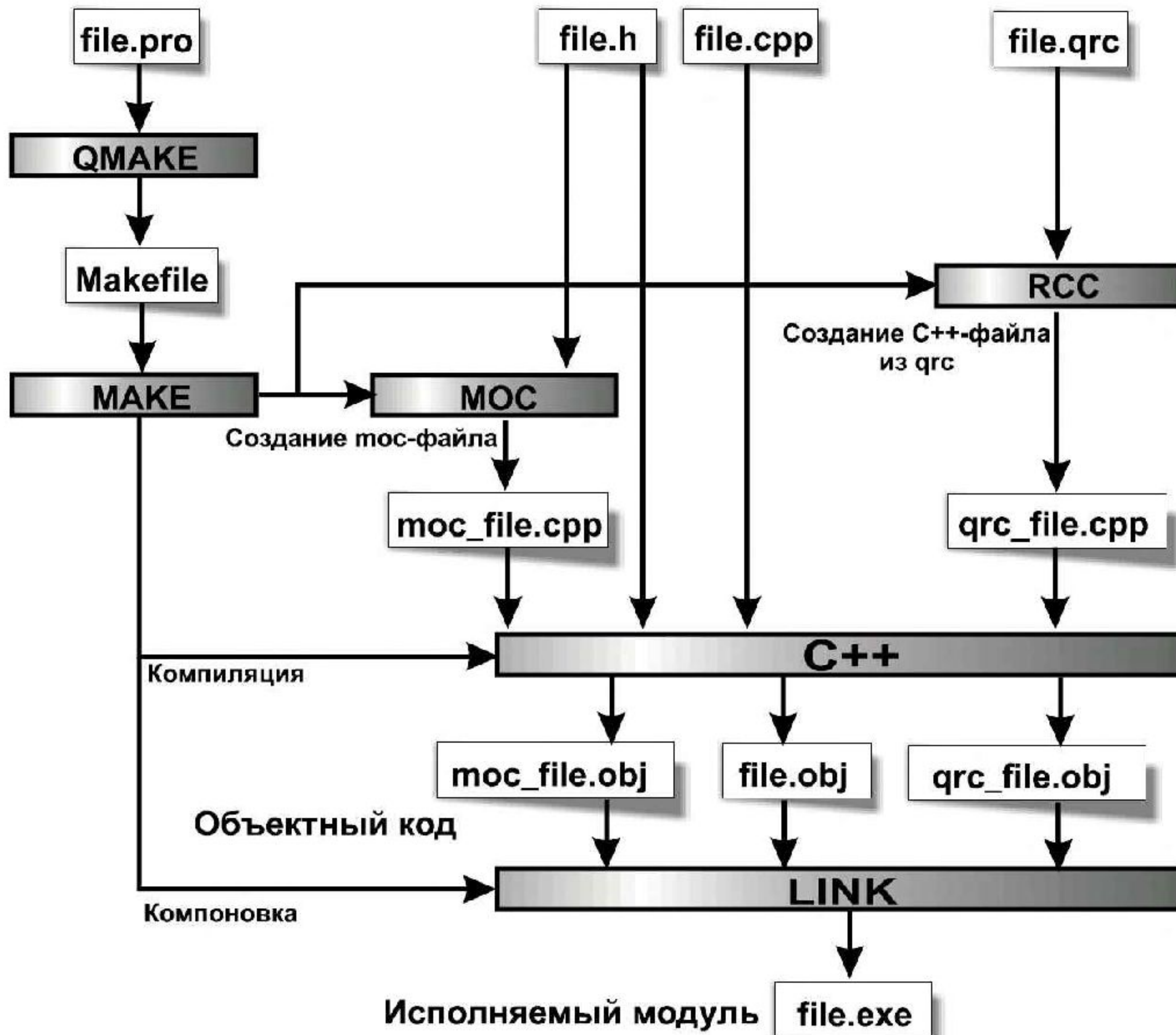
- Компилятор ресурсов
    - Внедрение ресурсов (изображений и тд.) в исполняемую программу

Файл qrc (Qt Resource Collection)

```
< ! DOCTYPE RCCXRCC version="1.0">
<qresource>
<file>images/open.png</file>
<file>images/quit .png</file>
</qresource>
</RCC>
```

Использование:

```
plbl->setPixmap (QPixmap ( " : /images/open.png" ) ) ;
```



Проектный файл
Исходные файлы
Файлы ресурсов
file.pro
file.h
file.cpp
file.qrc
QMAKE
Makefile
RCC
Создание C++-файла
из qrc
MAKE
MOC
Создание moc-файла
moc_file.cpp
qrc_file.cpp
C++
Компиляция
moc_file.obj
file.obj
qrc_file.obj
Объектный код
LINK
Компоновка
Исполняемый модуль
file.exe