Лекция 8

БИБЛИОТЕКА QT

Лекция 8. Библиотека Qt

План

1. Введение в Qt.
2. Иерархия классов и объектная модель Qt.
3. Сигналы и слоты.
4. Файлы проектов Qt.
5. Базовые возможности библиотеки Qt.

1. Введение в Qt

Qt – это кроссплатформенная библиотека, написанная на языке программирования C++ и предназначенная для создания различных программ с высококлассным графическим интерфейсом пользователя, работающих в среде операционных систем семейства Windows, Linux и MaxOS.

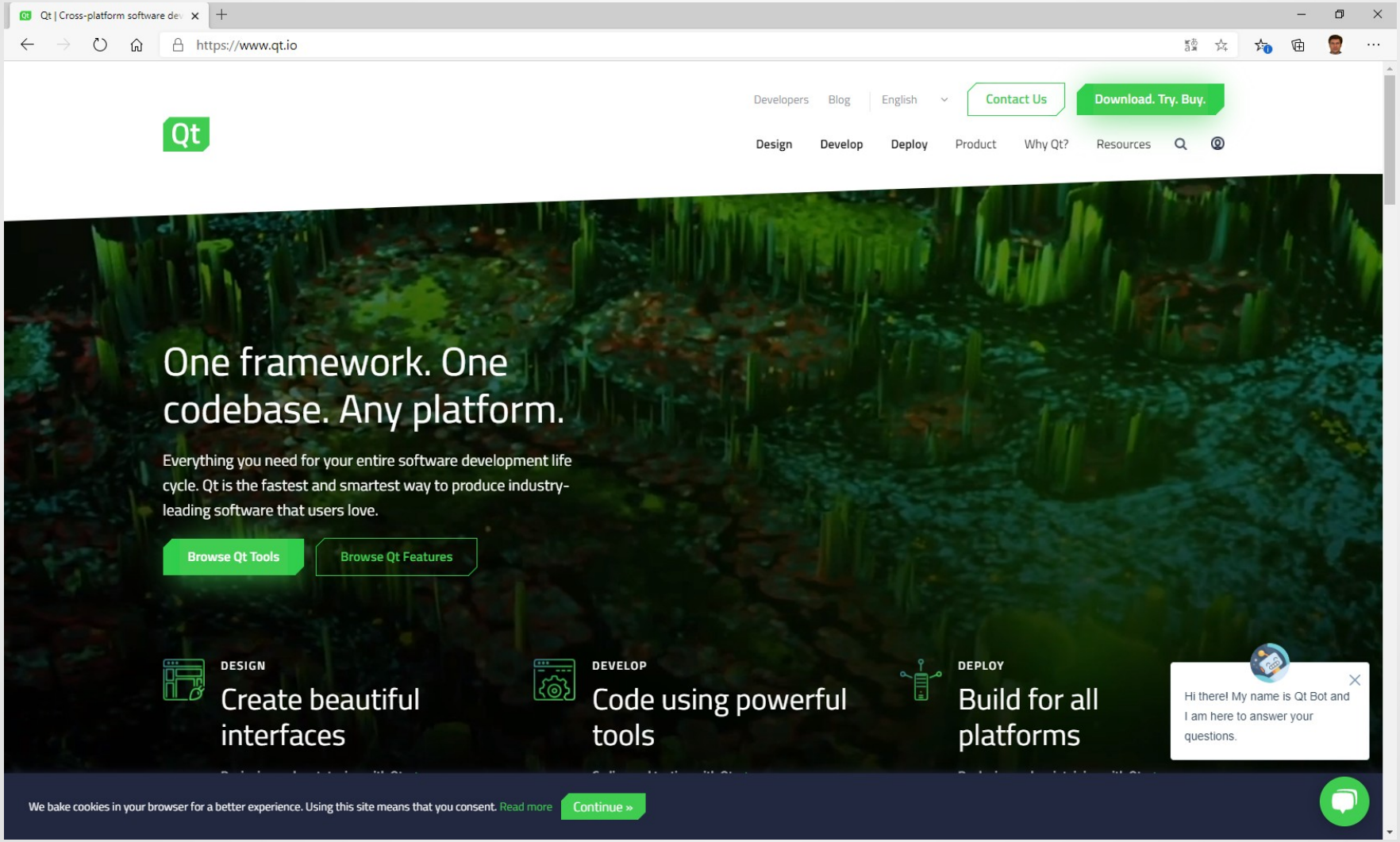
Библиотека Qt была разработана в 1996 году и с тех пор активно развивается. Ее используют: Autodesk, Google, Microsoft, Nokia, Panasonic, Siemens, Walt Disney Animation Studios и др.

Помимо создания GUI Qt позволяет:

- создавать многопоточные приложения;
- разрабатывать сетевые программы;
- разрабатывать программы с 2D- и 3D-графикой (в т. ч. и с использованием OpenGL);
- программировать базы данных (в т. ч. с использованием SQL);
- работать с XML;
- разрабатывать веб-приложения;
- работать с мультимедиа;
- взаимодействовать с ActiveX и COM (для разработчиков под Windows);
- и многое другое.

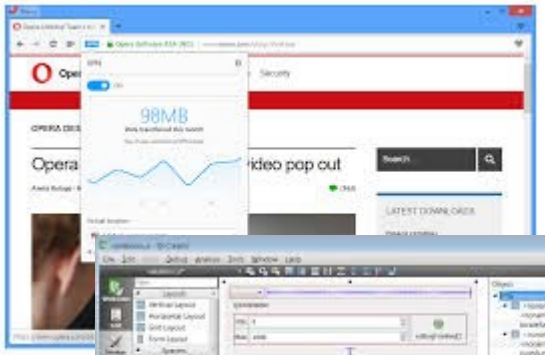
1. Введение в Qt

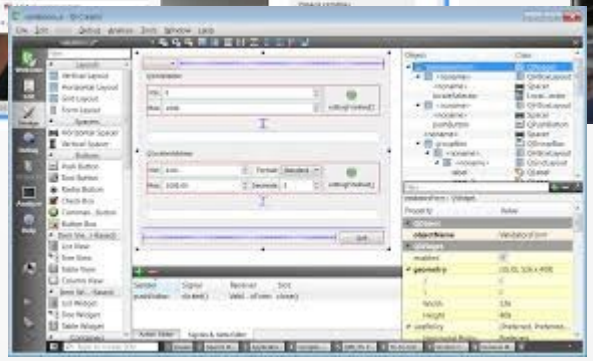
Официальный веб-сайт Qt: https://www.qt.io/

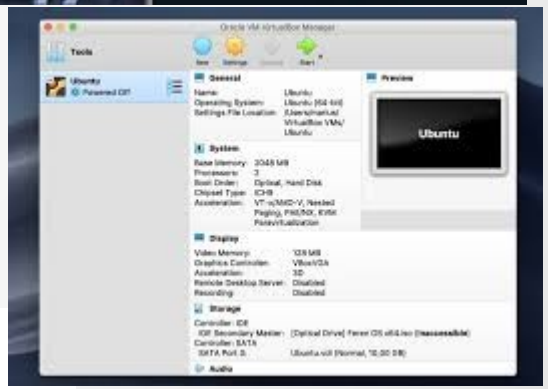


1. Введение в Qt

С помощью Qt написаны такие программы, как: KDE; Opera, Skype, Adobe Photoshop Album, Google Earth, VirtualBox, VCL media player и др.

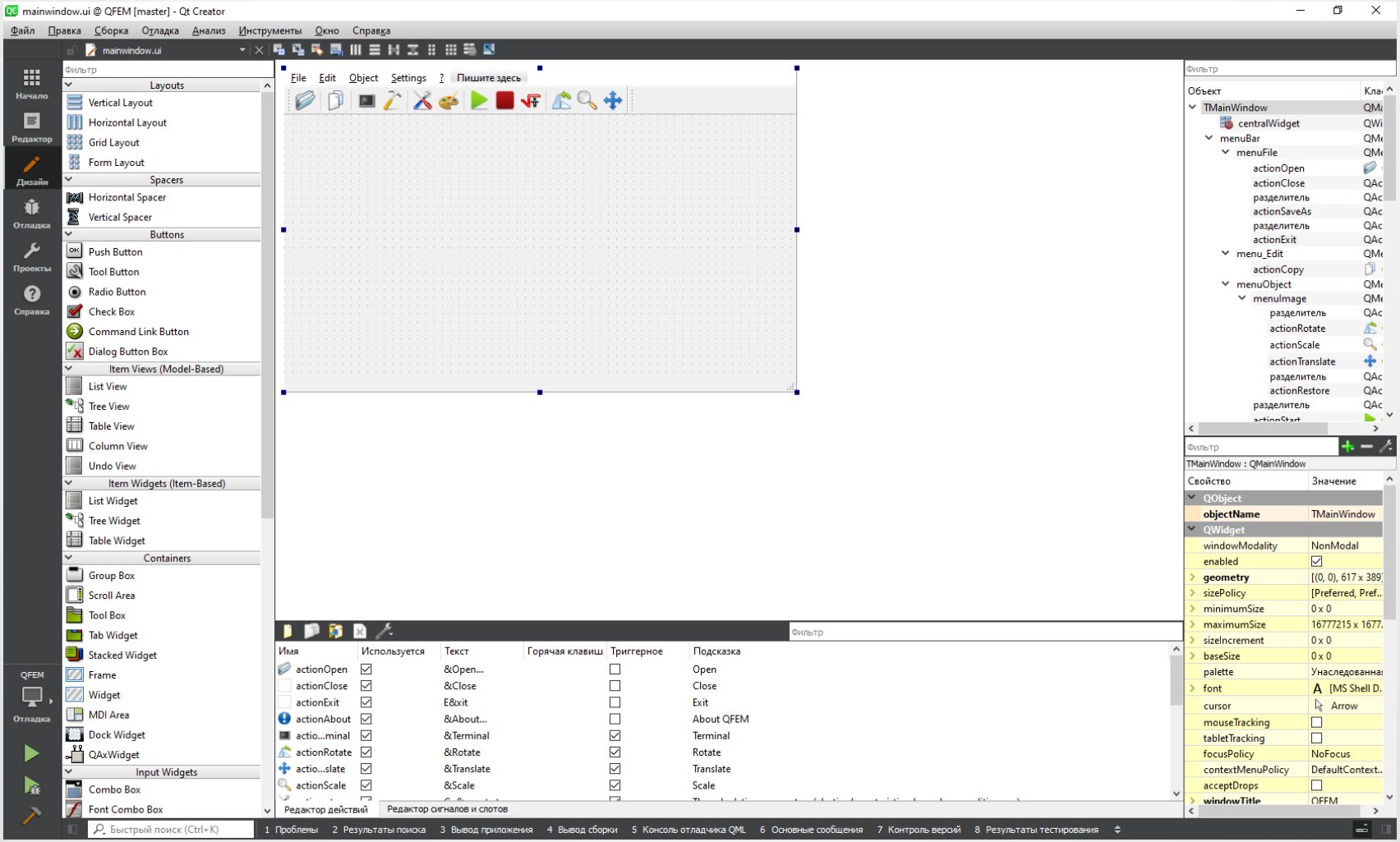






1. Введение в Qt

Широкой популярностью также пользуется кроссплатформенная IDE Qt Creator, основанная на использовании библиотеки Qt и поддерживающая технологию визуального программирования.



2. Иерархия классов и объектная модель Qt

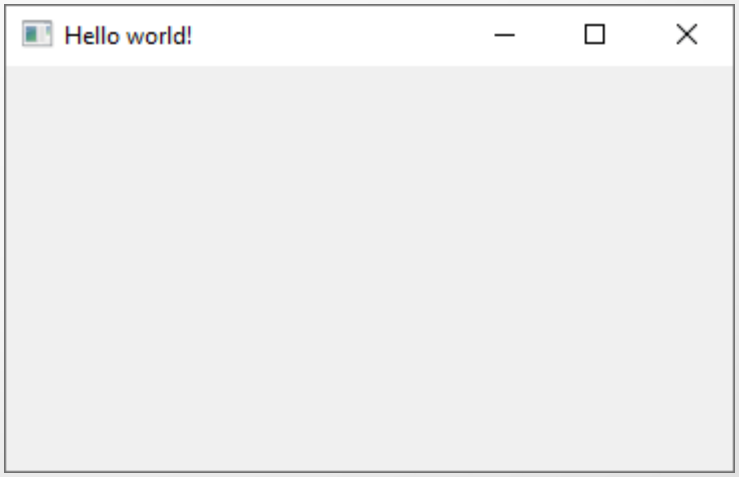
Простейшую программу на Qt с GUI можно, например, написать таким образом:

Исходный код

```
#include <QtWidgets>

int main(int argc, char **argv)
{
    QApplication app(argc, argv);
    QMainWindow mw;

    mw.setWindowTitle("Hello world!");
    mw.show();
    return app.exec();
}
```

Файл настроек компиляции

```
QT += gui widgets

CONFIG += c++17

SOURCES += \
        main.cpp
```



2. Иерархия классов и объектная модель Qt

Библиотека Qt состоит из более чем 500 классов, охватывающих большую часть функционала современных операционных систем. Для удобства программистов они объединены в модули. Основные среди них приведены в следующей таблице:

Модуль	Обозначение	Назначение
QtCore	core	Набор базовых классов, не связанных с GUI
QtGui	gui	Набор базовых классов для программирования GUI
QtWidgets	widgets	Модуль, содержащий графические виджеты для GUI
QtNetwork	network	Набор классов для программирования сети
QtOpenGL	opengl	Модуль для программирования графики OpenGL
QtSql	sql	Модуль для программирования баз данных
QtXml	xml	Набор классов для работы с XML
QtMultimedia	mulimedia	Классы для работы с мультимедиа
QtWebKit	webkit	Модуль для создания веб-приложений
QtPrintSupport	printsupport	Модуль для работы с принтером

3. Иерархия классов и объектная модель Qt

Большинство классов библиотеки Qt унаследованы от **QObject**. Класс QObject поддерживает:

- сигналы и слоты (signal/slot);
- таймер;
- механизмы объединения объектов в иерархии;
- события и механизмы их фильтрации;
- метаобъектную информацию;
- приведение типов;
- свойства.

Сигналы и **слоты** – это механизм обмена информацией о событиях, вырабатываемых объектами, а также способах их обработки, реализованный в Qt.

Использование **таймера**, определенного в QObject, позволяет производным классам использовать его в своих целях, экономя при этом время на разработку.

3. Иерархия классов и объектная модель Qt

Механизмы объединения объектов в иерархии позволяют сократить временные затраты на разработку программ, не заботясь при этом об освобождении памяти для создаваемых объектов, т. к. родительские объекты автоматически очистят память, занимаемую своими дочерними объектами.

Механизм фильтрации событий позволяет осуществлять их перехват, благодаря чему можно изменить реакцию объектов на происходящие события без изменения исходного кода класса.

Метаобъектная информация содержит данные о иерархии классов, а также позволяет узнать имя класса.

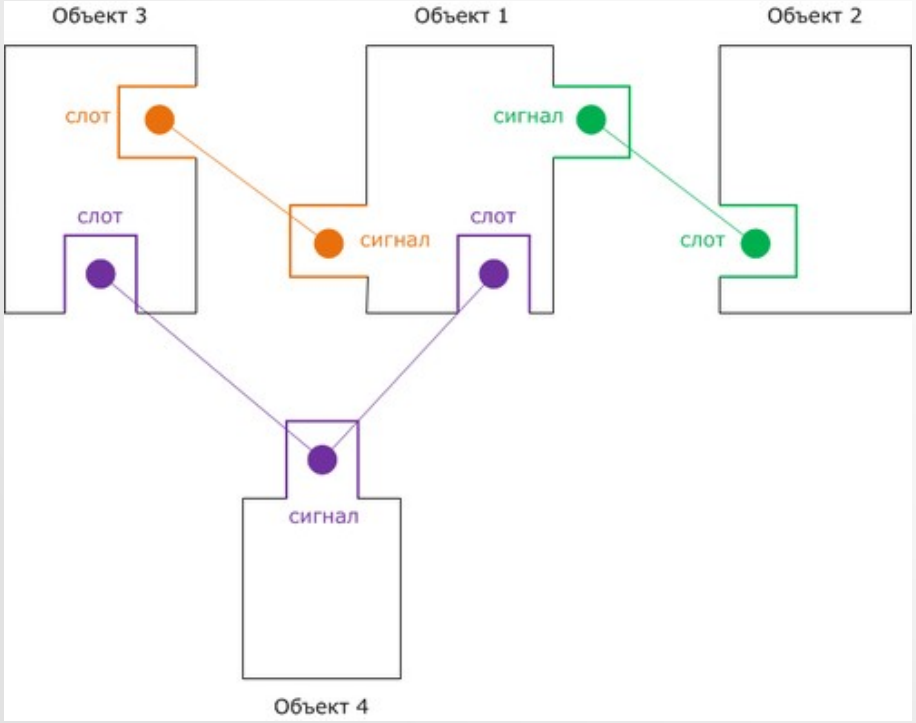
Приведение типов позволяет на базе метаобъектной информации осуществлять преобразование между классами-наследниками от QObject.

Свойства – специальные поля, для доступа к которым в классе должны существовать специальные методы чтения. Свойства широко применяются в визуальной среде разработки GUI – **Qt Designer**.

3. Сигналы и слоты

Классическим подходом к программной реализации GUI является использование **функций обратного вызова** (**callback function**), которые вызываются для обработки действий пользователя с элементами графического интерфейса.

Такой подход по своей сути не является объектно-ориентированным и его использование затрудняет чтение исходного кода.

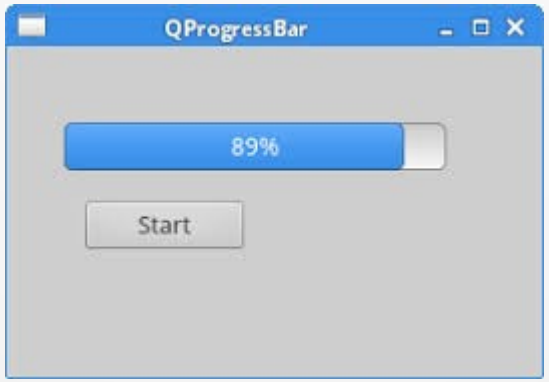


В Qt для обмена сообщениями между объектами и их обработки используется полностью объектно-ориентированный механизм **сигналов** и **слотов**. С его помощью можно соединять несвязанные изначально друг с другом объекты.

3. Сигналы и слоты

Механизм сигналов и слотов – основной **концепт** программирования с использованием Qt. Каждый производный от QObject класс может как принимать, так и отправлять сигналы.

Например, таймер генерирует и отправляет сообщение (сигнал) о том, что время обновилось. Пользовательский виджет, получив это сообщение, вызывает соответствующий метод (слот) для его обработки (обновления своего содержимого).



3. Сигналы и слоты

Сигналы в Qt – это специальные методы классов, способные осуществлять пересылку сообщений.

Например:

```
class TDrawMesh : public QGLWidget
{
    Q_OBJECT      // Директива метаобъектного компилятора
    // …
public:
    void rotateX(int angle)
    {
        emit xRotate(angle);      // Посылка сигнала
    }
    // …
signals:
    void xRotation(int);  // Сигнал, отправляемый при вращении вокруг оси X
    void yRotation(int);  // … Y
    void zRotation(int);  // … Z
};
```

3. Сигналы и слоты

Слоты в Qt – это специальные методы, автоматически вызываемые при получении сигнала (слот при этом должен быть предварительно соединен с сигналом).

Например:

```
class TGLFunction : public TGLMesh
{
    Q_OBJECT

    // …

public slots:
    void mouseDoubleClickEvent(QMouseEvent*);

    // …
};
```

3. Сигналы и слоты

Соединение сигналов и слотов происходит с помощью специального метода **connect()** класса Qobject.

```
QMetaObject::Connection
Qobject::connect(const QObject *sender,
                 const char *signal,
                 const QObject *receiver,
                 const char *method,
                 Qt::ConnectionType type = Qt::AutoConnection);
```

где: **sender** – указатель на объект, отправляющий сигнал;
signal – сигнал, с которым осуществляется соединение;
receiver – указатель на объект объект, содержащий слот;
method – вызываемый слот;
type – режим обработки (по-умолчанию – автоматический).

Например:

```
QLabel *label = new QLabel();
QScrollBar *scrollBar = new QScrollBar();

QObject::connect(scrollBar, SIGNAL(valueChanged(int)), label, SLOT(setNum(int)));
```

3. Сигналы и слоты

Отсоединение сигнала от слота происходит автоматически при удалении соответствующего объекта или с помощью специального метода **disconnect()** класса Qobject.

```
bool Qobject::disconnect( const char *signal = nullptr,
                          const QObject *receiver = nullptr,
                          const char *method = nullptr) const
```

Например:

```
// Отключение всех сигналов
disconnect(myObject, nullptr, nullptr, nullptr);
```

4. Файлы проектов Qt

При работе с библиотекой Qt на практике используют специальные **make-файлы** (**makefile**), содержащие различные параметры компиляции программы. Они как правило создаются с помощью специальной утилиты **qmake**. Проект можно создать автоматически при использовании Qt Creator.

Чаще всего в файлах проектов используются следующие параметры:

Опция	Назначение
HEADERS	Список заголовочных файлов проекта
SOURCES	Список исходных файлов проекта
TARGET	Имя результирующего файла
TEMPLATE	Тип результирующего файла (app – программа)
INCLUDEPATH	Путь к заголовочным файлам
LIBS	Путь к библиотекам
CONFIG	Параметры компилятора
FORMS	Файлы, описывающие экранные формы
TRANSLATIONS	Задает файлы переводов для проекта

4. Файлы проектов Qt

Пример файла-проекта:

```
#-------------------------------------------------
# Project created by QtCreator 2010-12-30T09:27:42
#-------------------------------------------------
QT += core gui opengl widgets

TARGET = QFEM
TEMPLATE = app

INCLUDEPATH += ../core \
               ../../../eigen

msvc:QMAKE_CXXFLAGS += /permissive-

win32 {
    INCLUDEPATH += ../../../intel/compilers_and_libraries_2019.5.281/windows/mkl/include/
    LIBS += -L$$PWD/../../../intel/compilers_and_libraries_2019.5.281/windows/mkl/lib/intel64_win/
}

unix {
    INCLUDEPATH +=../../../intel/mkl/include/
    LIBS += -L$$PWD/../../../intel/mkl/lib/intel64/ -lmkl_intel_lp64 -lmkl_sequential -lmkl_core
}

SOURCES += main.cpp mainwindow.cpp
HEADERS  += mainwindow.h
FORMS    += mainwindow.ui
TRANSLATIONS += QFEM_RU.ts
```

5. Базовые возможности библиотеки Qt

Одной из наиболее распространенных задач в программировании является обработка групп различным способом организованных данных.

Для этих целей Qt предоставляет специальную **библиотеку контейнеров Tulip**. Она является частью ядра Qt и по функционалу подобна STL – стандартной библиотеке шаблонов C++ (и при этом является совместимой с ней).

Классы, образующие Tulip, расположены в модуле QtCore. Они поддерживает:
контейнеры;
итераторы;
алгоритмы.

5. Базовые возможности библиотеки Qt

Контейнеры – это специальные классы, предназначенные для хранения элементов различных типов. Они могут при этом как просто хранить какие-то объекты, так и реализовать функционал специальных структур данных: **списков**, **очередей**, **стеков** и т. п.

В Qt все контейнеры реализованы как шаблонные классы – они могут хранить данные любых типов. Все контейнеры делятся на две группы: **последовательные** и **ассоциативные**.

Последовательные контейнеры
(упорядоченные коллекции, где каждый элемент занимает определенную позицию):

QVector<T> – вектор;
QList<T> – список;
QLinkedList<T> – двусвязный список;
QStack<T> – стек;
QQueue<T> – очередь.

Ассоциативные контейнеры
(коллекции, где позиция элемента зависит от его значения):

QSet<T> – множество;
QMap<T> – словарь;
QMultiMap<T> – мультисловарь;
QHash<T> – хэш;
QMultiHash<T> – мультихэш.

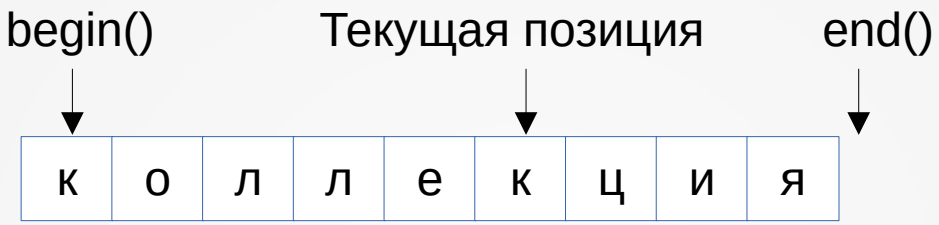
5. Базовые возможности библиотеки Qt

Для всех контейнеров доступны следующие операции:

Метод/оператор	Описание
==, !=	Операторы сравнения
=	Оператор присваивания
[ ]	Оператор индексации (не доступен для QSet<T> и QLinkedList<T>)
begin(), constBegin()	Возвращает итератор на первый элемент
end(), constEnd()	Возвращает итератор, указывающий на конец последовательности элементов в коллекции
clear()	Удаление всех элементов
insert()	Вставка элемента
remove()	Удаление элемента из коллекции
size(), count()	Возвращает количество элементов в контейнере
value()	Возвращает значение элемента (в QSet<T> не определен)
empty(), isEmpty()	Возвращает true, если контейнер пусть

5. Базовые возможности библиотеки Qt

Для перемещения по элементам контейнеров, абстрагируясь от их природы, используют **итераторы**.



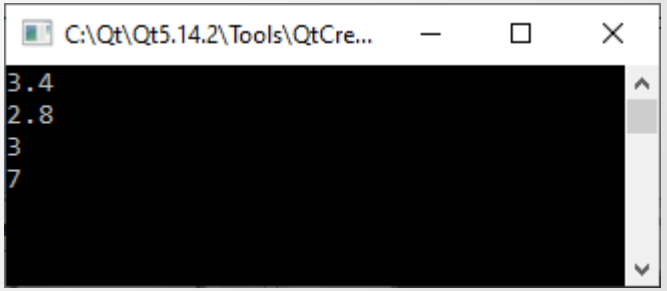
Вызов метода begin() возвращает итератор, указывающий на начало контейнера. Соответственно вызов end() возвращает итератор, указывающий на конец контейнера (**не на последний элемент!**).

Операторы **++** и **--** объекта итератора производят перемещения вперед и назад по коллекции (соответственно);

5. Базовые возможности библиотеки Qt

Пример программы, выводящей на экран положительные значения числовой коллекции.

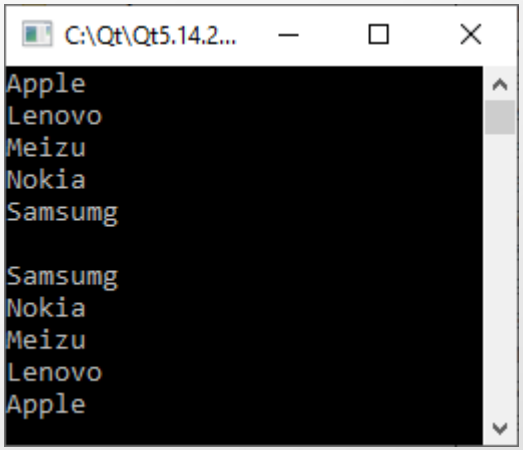
```
#include <QCoreApplication>
#include <QTextStream>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QTextStream stream(stdout);
    QVector<double> vec = {-1.5, 3.4, 2.8, -9.0, 3.0, 7.0 };

    for (auto it = vec.begin(); it != vec.end(); it++)
        if (*it > 0)
            stream << *it << endl;
    return a.exec();
}
```



5. Базовые возможности библиотеки Qt

В Qt для итерирования по коллекциям предоставляется специальный оператор **foreach**. Это разновидность цикла для перебора всех элементов контейнера. Например:

```
#include <QCoreApplication>
#include <QTextStream>
#include <QStack>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QTextStream stream(stdout);
    QStack<QString> stack;

    stack.push("Apple");
    stack.push("Lenovo");
    stack.push("Meizu");
    stack.push("Nokia");
    stack.push("Samsumg");

    foreach (auto it, stack)
        stream << it << endl;

    stream << endl;
    while (!stack.isEmpty())
        stream << stack.pop() << endl;
    return a.exec();
}
```



5. Базовые возможности библиотеки Qt

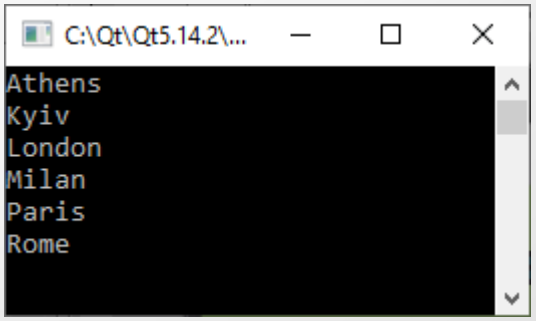
Для работы с контейнерами в библиотеке Qt реализованы такие **алгоритмы** (приведены в следующей таблице):

Алгоритм	Описание
qBinaryFind()	Двоичный поиск заданного значения
qCopy()	Копирование элементов, начиная с начала коллекции
qCopyBackward()	Копирование элементов, начиная с конца коллекции
qCount()	Подсчет элементов коллекции
qDeleteAll()	Удаление всех элементов
qEqual()	Сравнение коллекций
qFill()	Присвоение всем элементам коллекции заданного значения
qFind()	Поиск заданного значения
qLowerBound()	Нахождение элемента, больше или равного заданному значению
qUpperBound()	Нахождение элемента, меньшего заданному значению
qSort()	Сортировка
qStableSort()	Сортировка с сохранением позиции одинаковых элементов
qSwap()	Обмен местами двух значений

5. Базовые возможности библиотеки Qt

Например, сортировка списка может выглядеть таким образом:

```
#include <QCoreApplication>
#include <QTextStream>
#include <QList>
#include <QtAlgorithms>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QTextStream stream(stdout);
    QList<QString> list = { "Milan", "Rome", "London", "Paris", "Kyiv", "Athens" };

    qStableSort(list);

    foreach (auto it, list)
        stream << it << endl;
    return a.exec();
}
```



5. Базовые возможности библиотеки Qt

Поиск в коллекции может выглядеть таким образом:

```
#include <QCoreApplication>
#include <QTextStream>
#include <QList>
#include <QtAlgorithms>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    QTextStream stream(stdout);
    QList<QString> list = { "Milan", "Rome", "London", "Paris", "Kyiv", "Athens" };
    auto it = qFind(list.begin(), list.end(), "London");

    if (it == list.end())
        stream << "Unable find value in list" << endl;
    else
        stream << "Found: " << *it << endl;
    return a.exec();
}
```