

 **FREE eBook**

LEARNING

Qt

Free unaffiliated eBook created from
Stack Overflow contributors.

#qt

Table of Contents

About.....	1
Chapter 1: Getting started with Qt.....	2
Remarks.....	2
Versions.....	2
Examples.....	2
Installation and Setup on Windows and Linux.....	2
Hello World.....	8
Basic application with QtCreator and QtDesigner.....	9
Chapter 2: About using layouts, widget parenting.....	13
Introduction.....	13
Remarks.....	13
Examples.....	13
Basic Horizontal Layout.....	13
Basic Vertical Layout.....	14
Combining Layouts.....	15
Grid layout example.....	16
Chapter 3: Build QtWebEngine from source.....	19
Introduction.....	19
Examples.....	19
Build on Windows.....	19
Chapter 4: CMakeLists.txt for your Qt project.....	20
Examples.....	20
CMakeLists.txt for Qt 5.....	20
Chapter 5: Common Pitfalls.....	22
Examples.....	22
Using Qt::DirectConnection when receiver object doesn't receive signal.....	22
Chapter 6: Communication between QML and C++.....	24
Introduction.....	24
Examples.....	24
Call C++ in QML.....	24

Call QML in C++	25
Chapter 7: Deploying Qt applications.....	30
Examples.....	30
Deploying on windows.....	30
Integrating with CMake.....	30
Deploying on Mac.....	31
Deploying on linux.....	32
Chapter 8: Header on QListView.....	33
Introduction.....	33
Examples.....	33
Custom QListView declaration.....	33
Implementation of the custom QListView.....	34
Use case: MainWindow declaration.....	35
Use case: Implementation.....	35
Use case: Sample output.....	36
Chapter 9: Implicit sharing.....	38
Remarks.....	38
Examples.....	38
Basic Concept.....	38
Chapter 10: Model/View.....	40
Examples.....	40
A Simple Read-only Table to View Data from a Model.....	40
A simple tree model.....	43
Chapter 11: Multimedia.....	47
Remarks.....	47
Examples.....	47
Video Playback in Qt 5.....	47
Audio Playback in Qt5.....	47
Chapter 12: QDialogs.....	49
Remarks.....	49
Examples.....	49
MyCompareFileDialog.h.....	49

MyCompareFileDialogDialog.cpp.....	49
MainWindow.h.....	50
MainWindow.cpp.....	50
main.cpp.....	51
mainwindow.ui.....	51
Chapter 13: QGraphics.....	53
Examples.....	53
Pan, zoom, and rotate with QGraphicsView.....	53
Chapter 14: qmake.....	55
Examples.....	55
Default "pro" file.....	55
Preserving source directory structure in a build (undocumented "object_parallel_to_source".....	55
Simple Example (Linux).....	56
SUBDIRS example.....	57
Library example.....	59
Creating a project file from existing code.....	59
Chapter 15: QObject.....	61
Remarks.....	61
Examples.....	61
QObject example.....	61
qobject_cast.....	61
QObject Lifetime and Ownership.....	62
Chapter 16: Qt - Dealing with Databases.....	64
Remarks.....	64
Examples.....	64
Using a Database on Qt.....	64
Qt - Dealing with Sqlite Databases.....	65
Qt - Dealing with ODBC Databases.....	66
Qt - Dealing with in-memory Sqlite Databases.....	68
Remove Database connection correctly.....	69
Chapter 17: Qt Container Classes.....	71
Remarks.....	71

Examples.....	71
QStack usage.....	71
QVector usage.....	71
QLinkedList usage.....	72
QList.....	72
Chapter 18: Qt Network.....	75
Introduction.....	75
Examples.....	75
TCP Client.....	75
TCP Server.....	77
Chapter 19: Qt Resource System.....	81
Introduction.....	81
Examples.....	81
Referencing files within code.....	81
Chapter 20: QTimer.....	82
Remarks.....	82
Examples.....	82
Simple example.....	82
Singleshot Timer with Lambda function as slot.....	84
Using QTimer to run code on main thread.....	84
Basic Usage.....	84
QTimer::singleShot simple usage.....	85
Chapter 21: Signals and Slots.....	87
Introduction.....	87
Remarks.....	87
Examples.....	87
A Small Example.....	87
The new Qt5 connection syntax.....	88
Connecting overloaded signals/slots.....	90
Multi window signal slot connection.....	91
Chapter 22: SQL on Qt.....	94
Examples.....	94

Basic connection and query	94
Qt SQL query parameters	94
MS SQL Server Database Connection using QODBC	95
Chapter 23: Threading and Concurrency	98
Remarks	98
Examples	98
Basic usage of QThread	98
QtConcurrent Run	99
Invoking slots from other threads	100
Chapter 24: Using Style Sheets Effectively	102
Examples	102
Setting a UI widget's stylesheet	102
Credits	103

About

You can share this PDF with anyone you feel could benefit from it, downloaded the latest version from: [qt](#)

It is an unofficial and free Qt ebook created for educational purposes. All the content is extracted from [Stack Overflow Documentation](#), which is written by many hardworking individuals at Stack Overflow. It is neither affiliated with Stack Overflow nor official Qt.

The content is released under Creative Commons BY-SA, and the list of contributors to each chapter are provided in the credits section at the end of this book. Images may be copyright of their respective owners unless otherwise specified. All trademarks and registered trademarks are the property of their respective company owners.

Use the content presented in this book at your own risk; it is not guaranteed to be correct nor accurate, please send your feedback and corrections to info@zzzprojects.com

Chapter 1: Getting started with Qt

Remarks

As [official documentation](#) stated, Qt is a cross-platform application development framework for desktop, embedded and mobile. Supported Platforms include Linux, OS X, Windows, VxWorks, QNX, Android, iOS, BlackBerry, Sailfish OS and others.

This section provides an overview of what Qt is, and why a developer might want to use it.

It should also mention any large subjects within Qt, and link out to the related topics. Since the documentation for qt is new, you may need to create initial versions of those related topics.

Versions

Version	Release date
Qt 3.0	2001-10-16
Qt 3.3	2004-02-05
Qt 4.1	2005-12-20
Qt 4.8	2011-12-15
Qt 5.0	2012-12-19
Qt 5.6	2016-03-16
Qt 5.7	2016-06-16
Qt 5.8	2017-01-23
Qt 5.9	2017-05-31

Examples

Installation and Setup on Windows and Linux

Download Qt for Linux Open Source Version

Go to <https://www.qt.io/download-open-source/> and click on Download Now, make sure that you are downloading the Qt installer for Linux.

Recommended

We detected your operating system as: Linux
Recommended download: Qt Online Installer for Linux

Before you begin your download, please make sure you:

- › learn about the [obligations of the LGPL](#).
- › read the [FAQ](#) about developing with the LGPL.

[Download Now](#)

Qt online installer is a small executable which downloads content over internet based on your selections. It provides all Qt 5.x binary & source packages and latest Qt Creator.

For more information visit our [Developers page](#).
Not the download package you need? [View All Downloads](#)

A file with the name qt-unified-linux-x-online.run will be downloaded, then add exec permission

```
chmod +x qt-unified-linux-x-online.run
```

Remember to change 'x' for the actual version of the installer. Then run the installer

```
./qt-unified-linux-x-online.run
```

Download Qt for Windows Open Source Version

Go to <https://www.qt.io/download-open-source/>. The following screenshot shows the download page on Windows:

Your download

We detected your operating system as: Windows

Recommended download: Qt Online Installer for Windows

Before you begin your download, please make sure you:

- › learn about the [obligations of the LGPL](#).
- › read the [FAQ](#) about developing with the LGPL.

[Download Now](#)

Qt online installer is a small executable which downloads content over internet based on your selections. It provides all Qt 5.x binary & source packages and the latest Qt Creator.

For more information visit our [Developers page](#).

Not the download package you need? [View All Downloads](#)

What you should do now depends on which IDE you're going to use. If you're going to use Qt Creator, which is included in the installer program, just click on Download Now and run the executable.

If you're going to use Qt in Visual Studio, normally the Download Now button should also work. Make sure the file downloaded is called qt-opensource-windows-x86-msvc2015_64-x.x.x.exe or qt-opensource-windows-x86-msvc2015_32-x.x.x.exe (where x.x.x is the version of Qt, for example 5.7.0). If that's not the case, click on View All Downloads and select one of the first four options under Windows Host.

If you're going to use Qt in Code::Blocks, click on View All Downloads and select Qt x.x.x for Windows 32-bit (MinGW x.x.x, 1.2 GB) under Windows Host.

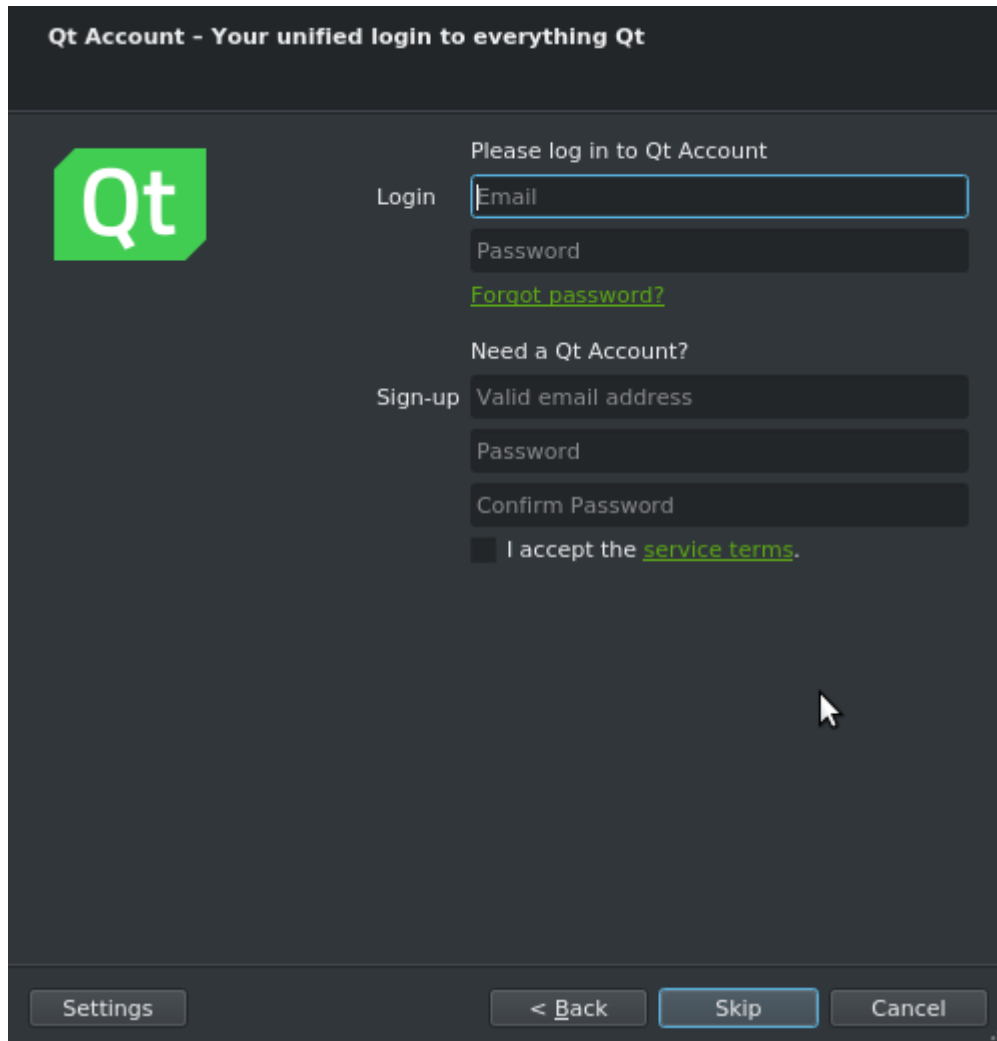
Once you've downloaded the appropriate installer file, run the executable and follow the instructions below. Note that you need to be an administrator to install Qt. If you're not an

administrator, you can find several alternative solutions [here](#).

Install Qt in any operative system

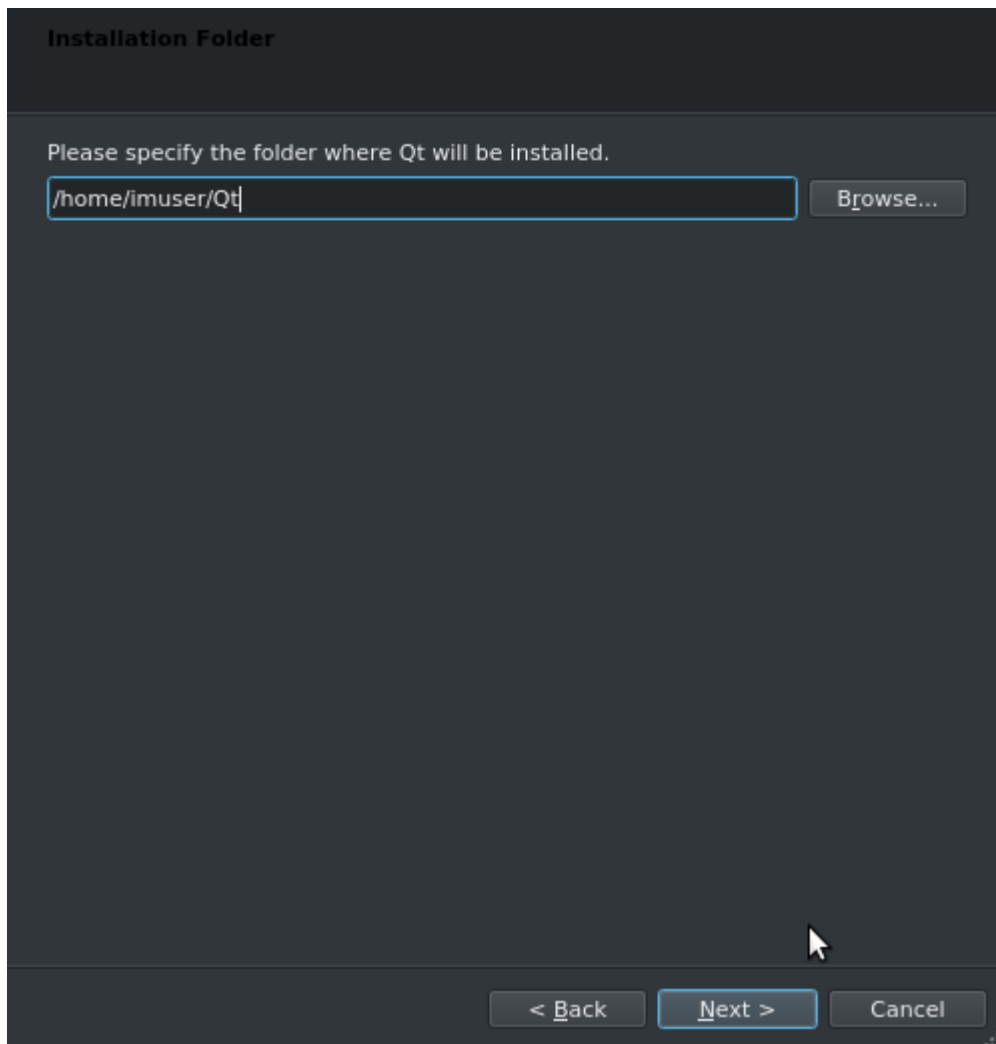
Once you've downloaded Qt and opened the installer program, the installation procedure is the same for all operative systems, although the screenshots might look a bit different. The screenshots provided here are from Linux.

Login with a existing Qt account or create a new one:

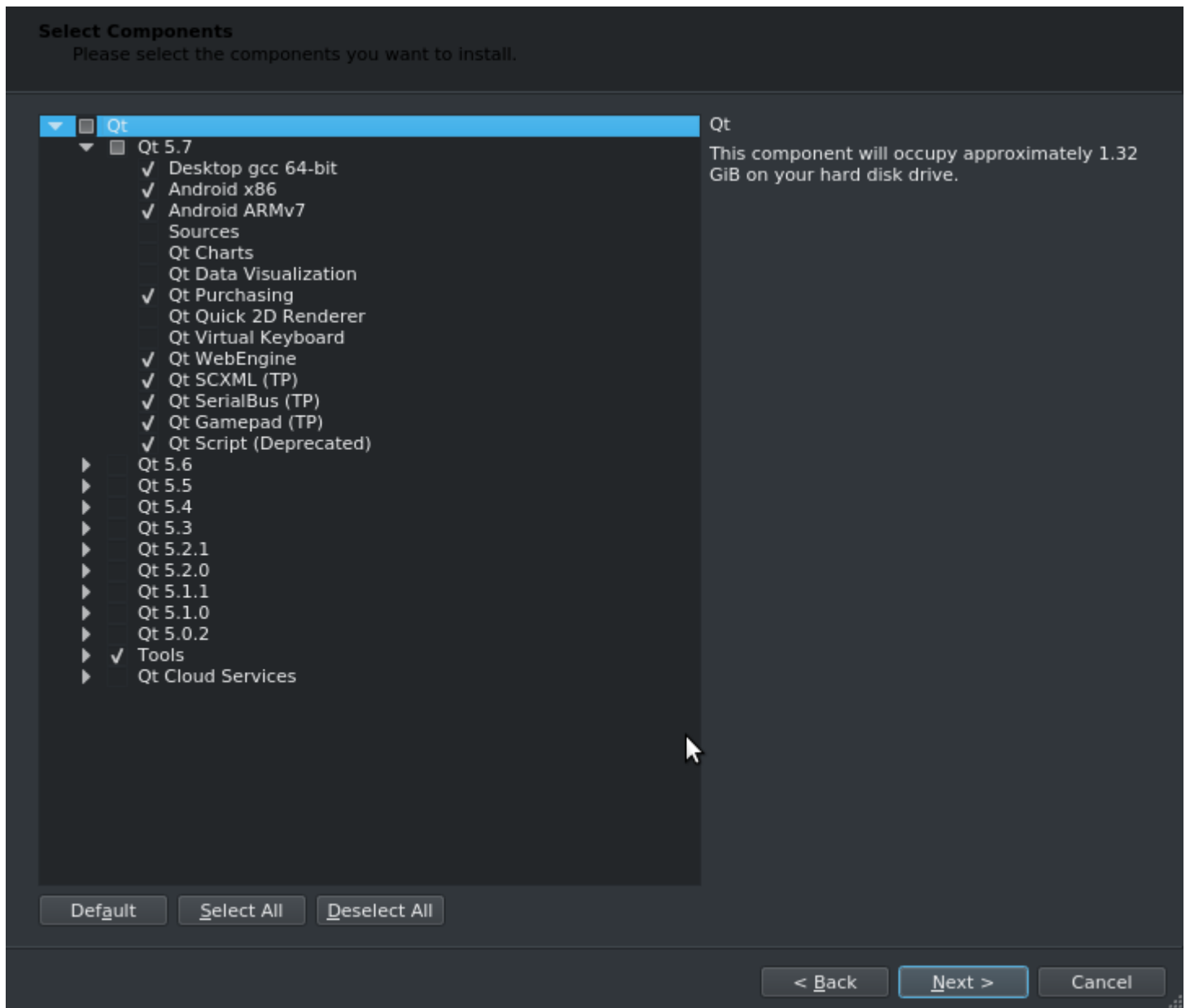


The image shows a Qt Account login and sign-up window. The title bar reads "Qt Account - Your unified login to everything Qt". On the left is the Qt logo. The main area is divided into two sections: "Login" and "Sign-up". The "Login" section has a heading "Please log in to Qt Account" and fields for "Email" and "Password", with a link for "Forgot password?". The "Sign-up" section has a heading "Need a Qt Account?" and fields for "Valid email address", "Password", and "Confirm Password", along with a checkbox for "I accept the service terms." At the bottom are buttons for "Settings", "< Back", "Skip", and "Cancel".

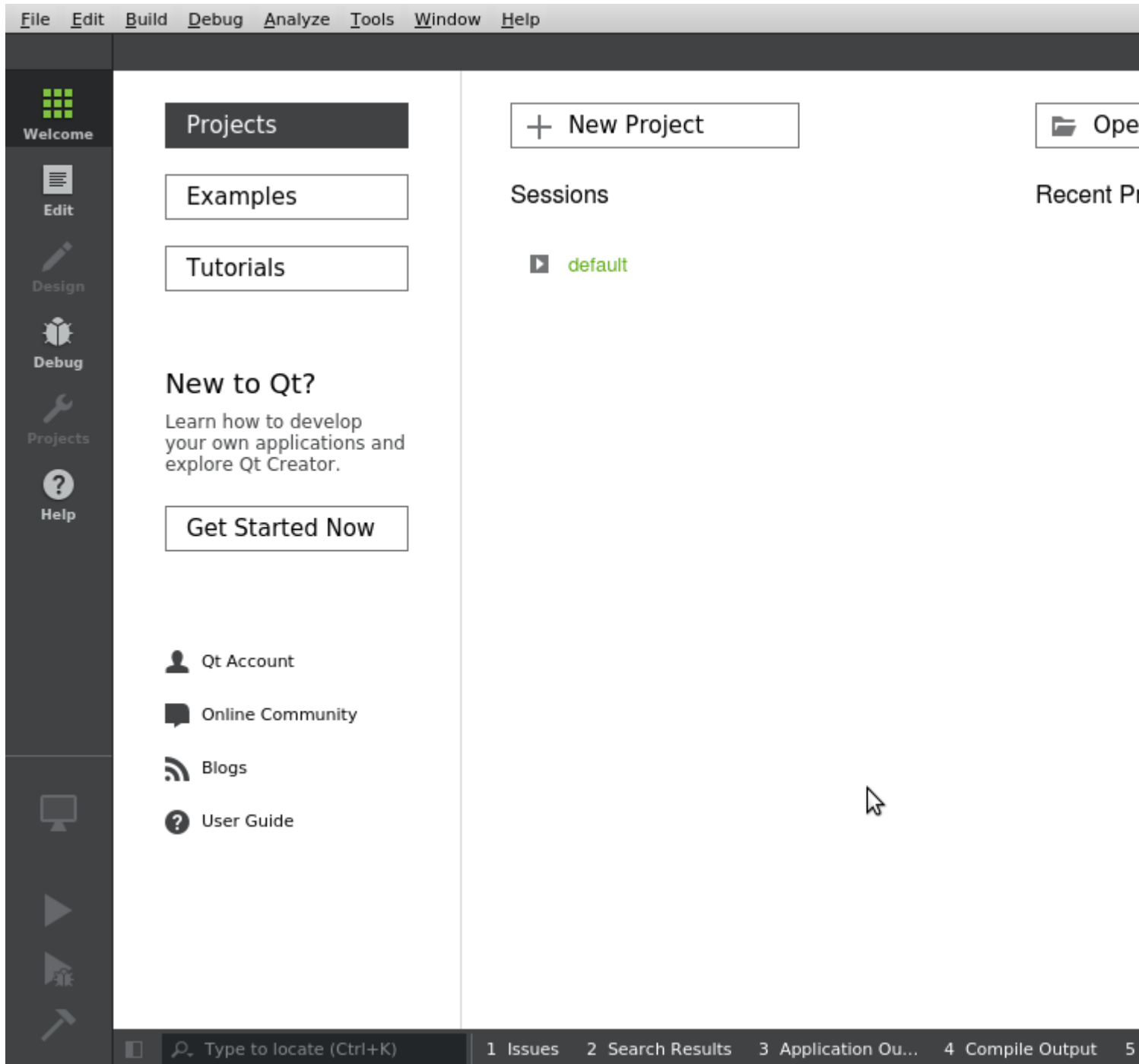
Select a path to install the Qt libraries and tools



Select the library version and the features you want



After downloading and the installation is finished, go to the Qt installation directory and launch Qt Creator or run it directly from the command line.



Hello World

In this example, we simply create and show a push button in a window frame on the desktop. The push button will have the label `Hello world!`

This represents the simplest possible Qt program.

First of all we need a project file:

helloworld.pro

```
QT       += core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets
```

```
TARGET = helloworld
TEMPLATE = app

SOURCES += main.cpp
```

- QT is used to indicate what libraries (Qt modules) are being used in this project. Since our first app is a small GUI, we will need QtCore and QtGui. As Qt5 separate QtWidgets from QtGui, we need add `greaterThan` line in order to compile it with Qt5.
- TARGET is the name of the app or the library.
- TEMPLATE describes the type to build. It can be an application (app), a library (lib), or simply subdirectories (subdirs).
- SOURCES is a list of source code files to be used when building the project.

We also need the main.cpp containing a Qt application:

main.cpp

```
#include <QApplication>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QPushButton button ("Hello world!");
    button.show();

    return a.exec(); // .exec starts QApplication and related GUI, this line starts 'event
loop'
}
```

- QApplication object. This object manages application-wide resources and is necessary to run any Qt program that has a GUI. It needs argv and args because Qt accepts a few command line arguments. When calling `a.exec()` the Qt event loop is launched.
- QPushButton object. The push button with the label `Hello world!`. The next line, `button.show()`, shows the push button on the screen in its own window frame.

Finally, to run the application, open a command prompt, and enter the directory in which you have the .cpp file of the program. Type the following shell commands to build the program.

```
qmake -project
qmake
make
```

Basic application with QtCreator and QtDesigner

QtCreator is, at the moment, the best tool to create a Qt application. In this example, we will see how to create a simple Qt application which manage a button and write text.

To create a new application click on File->New File or Project:

File

Edit

Build

Debug

Analyze

Tools

Win



New File or Project...

Ctrl+N



Open File or Project...

Ctrl+O

Open File With...

Recent Files

Recent Projects

Sessions

Session Manager...

Close Project

Close All Projects and Editors



Save

Ctrl+S

Save As...

Save All

Ctrl+Sh

Revert to Saved

Close

Ctrl+W

which is an awesome class which can convert many thing in many others things. So left add an int which increase when we push the button.

So the .h:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

public slots:
    void whenButtonIsClicked();

private slots:
    void on_pushButton_clicked();

private:
    Ui::MainWindow *ui;
    double _smallCounter;
};

#endif // MAINWINDOW_H
```

The .cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

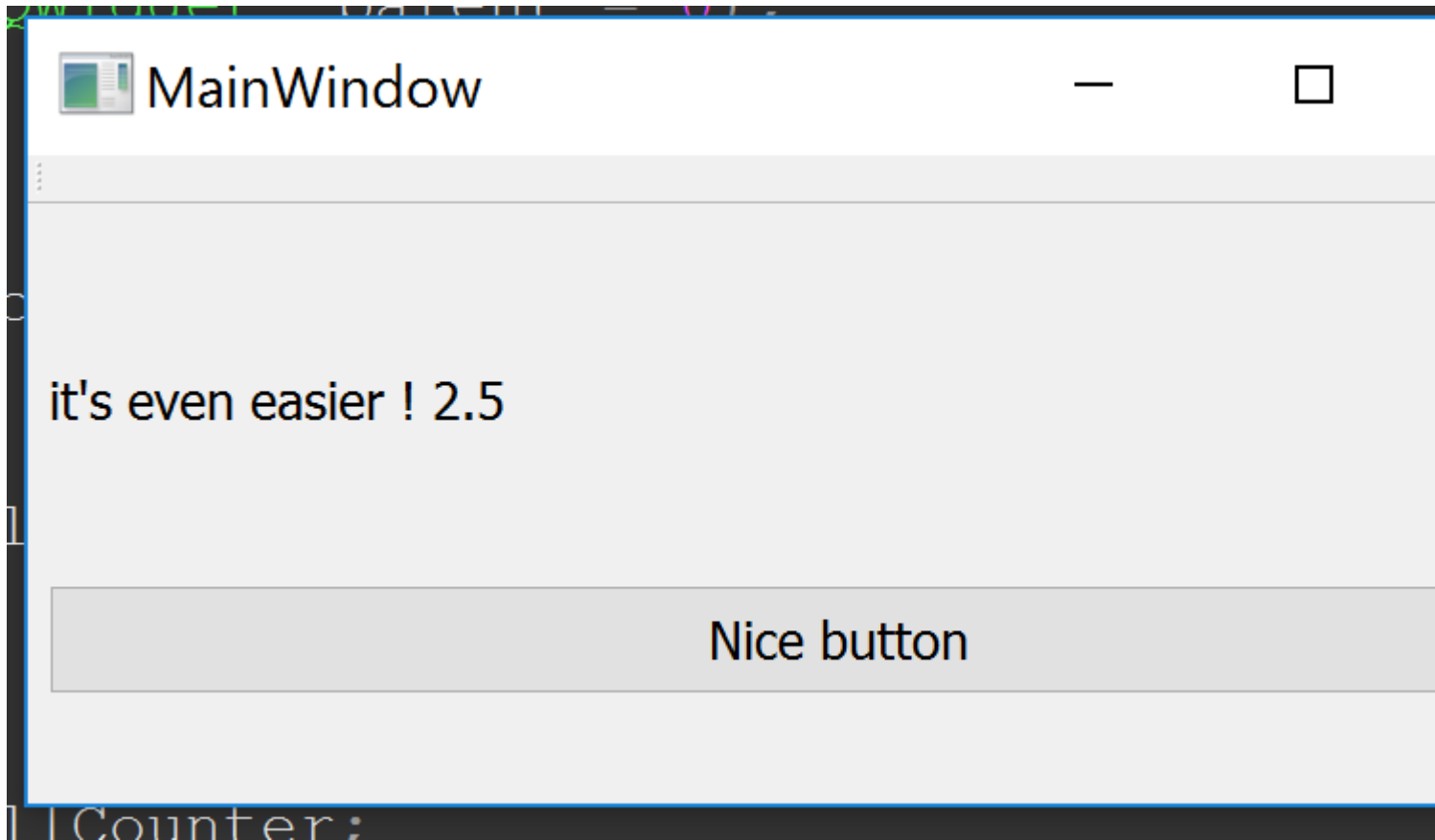
MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    // connect(ui->pushButton, SIGNAL(clicked(bool)), this, SLOT(whenButtonIsClicked()));
    _smallCounter = 0.0f;
}

MainWindow::~~MainWindow()
{
    delete ui;
}

void MainWindow::whenButtonIsClicked()
{
    ui->label->setText("the button has been clicked !");
}
```

```
void MainWindow::on_pushButton_clicked()
{
    _smallCounter += 0.5f;
    ui->label->setText("it's even easier ! " + QVariant(_smallCounter).toString());
}
```

And now, we can save and run again. Every time you click on the button, it show "it's even easier !" with the value of `_smallCounter`. So you should have something like:



This tutorial is done. If you want to learn more about Qt, let's see the other examples and documentation of Qt on [the StackOverflow Documentation](#) or [the Qt Documentation](#)

Read Getting started with Qt online: <https://riptutorial.com/qt/topic/902/getting-started-with-qt>

Chapter 2: About using layouts, widget parenting

Introduction

The layouts are a necessary in every Qt application. They manage the object, their position, their size, how they are resized.

Remarks

From [Qt layout documentation](#):

When you use a layout, you do not need to pass a parent when constructing the child widgets. The layout will automatically reparent the widgets (using `QWidget::setParent()`) so that they are children of the widget on which the layout is installed.

So do :

```
QGroupBox *box = new QGroupBox("Information:", widget);
layout->addWidget(box);
```

or do :

```
QGroupBox *box = new QGroupBox("Information:", nullptr);
layout->addWidget(box);
```

is exactly the same.

Examples

Basic Horizontal Layout

The horizontal layout set up the object inside it horizontally.

basic code:

```
#include <QApplication>

#include <QMainWindow>
#include <QWidget>
#include <QHBoxLayout>
#include <QPushButton>

int main(int argc, char *argv[])
{
```

```

QApplication a(argc, argv);

QMainWindow window;
QWidget *widget = new QWidget(&window);
QHBoxLayout *layout = new QHBoxLayout(widget);

window.setCentralWidget(widget);
widget->setLayout(layout);

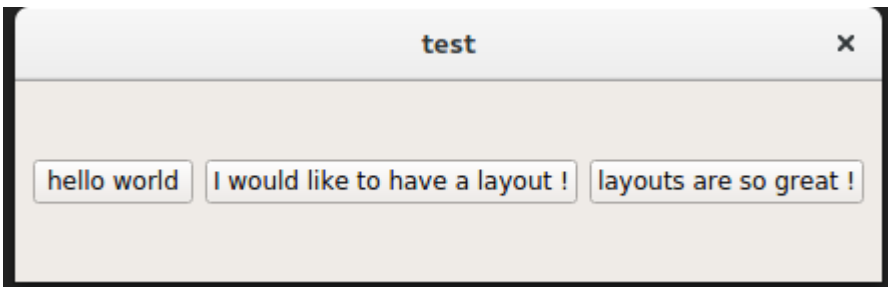
layout->addWidget(new QPushButton("hello world", widget));
layout->addWidget(new QPushButton("I would like to have a layout !", widget));
layout->addWidget(new QPushButton("layouts are so great !", widget));

window.show();

return a.exec();
}

```

this will output:



Basic Vertical Layout

The vertical layout set up the object inside it vertically.

```

#include "mainwindow.h"
#include <QApplication>

#include <QMainWindow>
#include <QWidget>
#include <QVBoxLayout>
#include <QPushButton>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QMainWindow window;
    QWidget *widget = new QWidget(&window);
    QVBoxLayout *layout = new QVBoxLayout(widget);

    window.setCentralWidget(widget);
    widget->setLayout(layout);

    layout->addWidget(new QPushButton("hello world", widget));
    layout->addWidget(new QPushButton("I would like to have a layout !", widget));
}

```

```

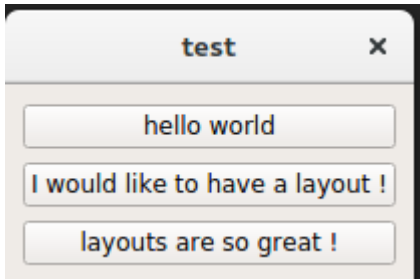
layout->addWidget(new QPushButton("layouts are so great !", widget));

window.show();

return a.exec();
}

```

output:



Combining Layouts

You can combine multiple layout thanks to other QWidgets in your main layout to do more specific effects like an information field: for example:

```

#include <QApplication>

#include <QMainWindow>
#include <QWidget>
#include <QVBoxLayout>
#include <QPushButton>
#include <QLabel>
#include <QLineEdit>
#include <QGroupBox>

#include <QTextEdit>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QMainWindow window;
    QWidget *widget = new QWidget(&window);
    QVBoxLayout *layout = new QVBoxLayout(widget);

    window.setCentralWidget(widget);
    widget->setLayout(layout);

    QGroupBox *box = new QGroupBox("Information:", widget);
    QVBoxLayout *boxLayout = new QVBoxLayout(box);

    layout->addWidget(box);

    QWidget* nameWidget = new QWidget(box);
    QWidget* ageWidget = new QWidget(box);
    QWidget* addressWidget = new QWidget(box);

    boxLayout->addWidget(nameWidget);
    boxLayout->addWidget(ageWidget);

```

```

boxLayout->addWidget(addressWidget);

QHBoxLayout *nameLayout = new QHBoxLayout(nameWidget);
nameLayout->addWidget(new QLabel("Name:"));
nameLayout->addWidget(new QLineEdit(nameWidget));

QHBoxLayout *ageLayout = new QHBoxLayout(ageWidget);
ageLayout->addWidget(new QLabel("Age:"));
ageLayout->addWidget(new QLineEdit(ageWidget));

QHBoxLayout *addressLayout = new QHBoxLayout(addressWidget);
addressLayout->addWidget(new QLabel("Address:"));
addressLayout->addWidget(new QLineEdit(addressWidget));

QWidget* validateWidget = new QWidget(widget);
QHBoxLayout *validateLayout = new QHBoxLayout(validateWidget);
validateLayout->addWidget(new QPushButton("Validate", validateWidget));
validateLayout->addWidget(new QPushButton("Reset", validateWidget));
validateLayout->addWidget(new QPushButton("Cancel", validateWidget));

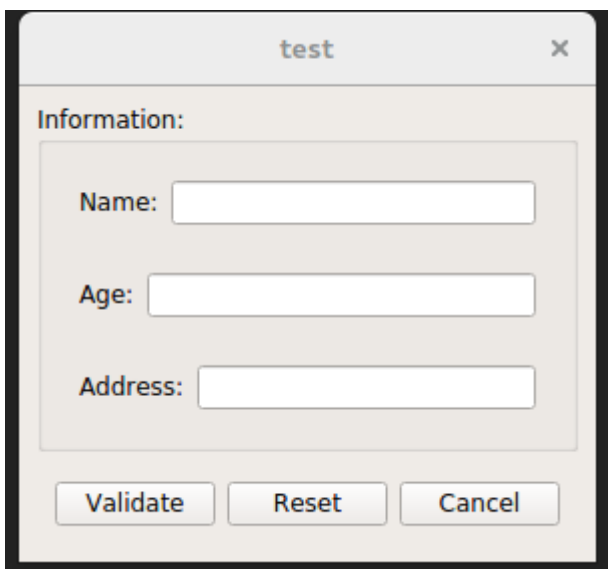
layout->addWidget(validateWidget);

window.show();

return a.exec();
}

```

will output :



Grid layout example

The grid layout is a powerful layout with which you can do an horizontal and vertical layout a once.

example:

```

#include "mainwindow.h"
#include <QApplication>

```

```

#include <QMainWindow>
#include <QWidget>
#include <QVBoxLayout>
#include <QPushButton>
#include <QLabel>
#include <QLineEdit>
#include <QGroupBox>

#include <QTextEdit>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);

    QMainWindow window;
    QWidget *widget = new QWidget(&window);
    QGridLayout *layout = new QGridLayout(widget);

    window.setCentralWidget(widget);
    widget->setLayout(layout);

    QGroupBox *box = new QGroupBox("Information:", widget);
    layout->addWidget(box, 0, 0);

    QVBoxLayout *boxLayout = new QVBoxLayout(box);

    QWidget* nameWidget = new QWidget(box);
    QWidget* ageWidget = new QWidget(box);
    QWidget* addressWidget = new QWidget(box);

    boxLayout->addWidget(nameWidget);
    boxLayout->addWidget(ageWidget);
    boxLayout->addWidget(addressWidget);

    QHBoxLayout *nameLayout = new QHBoxLayout(nameWidget);
    nameLayout->addWidget(new QLabel("Name:"));
    nameLayout->addWidget(new QLineEdit(nameWidget));

    QHBoxLayout *ageLayout = new QHBoxLayout(ageWidget);
    ageLayout->addWidget(new QLabel("Age:"));
    ageLayout->addWidget(new QLineEdit(ageWidget));

    QHBoxLayout *addressLayout = new QHBoxLayout(addressWidget);
    addressLayout->addWidget(new QLabel("Address:"));
    addressLayout->addWidget(new QLineEdit(addressWidget));

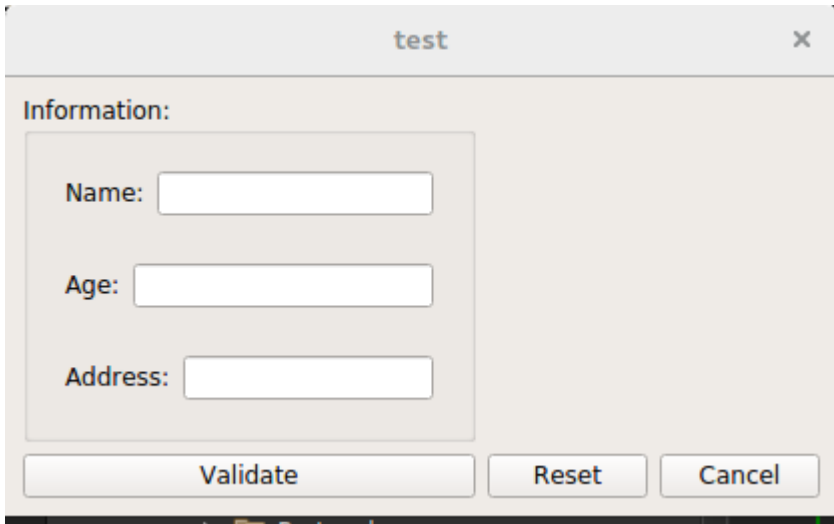
    layout->addWidget(new QPushButton("Validate", widget), 1, 0);
    layout->addWidget(new QPushButton("Reset", widget), 1, 1);
    layout->addWidget(new QPushButton("Cancel", widget), 1, 2);

    window.show();

    return a.exec();
}

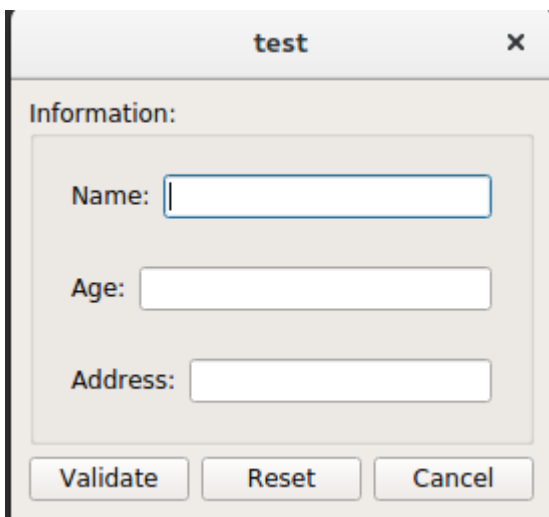
```

give :



so you can see that the group box is only in the first column and first row as the addWidget was `layout->addWidget(box, 0, 0);`

However, if you change it to `layout->addWidget(box, 0, 0, 1, 3);`, the new 0 and 3 represent how many line and column you want for your widget so it give :



exactly the same as you created a horizontal and then a vertical layout in a subwidget.

Read About using layouts, widget parenting online: <https://riptutorial.com/qt/topic/9380/about-using-layouts--widget-parenting>

Chapter 3: Build QtWebEngine from source

Introduction

Sometimes we need to build QtWebEngine from source for some reason, such as for mp3 support.

Examples

Build on Windows

Requirements

- Windows 10, please **set your system locale to English**, otherwise there may be errors
- Visual Studio 2013 or 2015
- QtWebEngine 5.7 source code (could be downloaded from [here](#))
- Qt 5.7 install version, install it and add `qmake.exe` folder to system path
- Python 2, add `python.exe` folder to system path
- Git, add `git.exe` folder to system path
- gperf, add `gperf.exe` folder to system path
- flex-bison, add `win_bison.exe` folder to system path, and rename it to `bison.exe`

Note: I didn't test for Visual Studio versions, all Qt versions.. Let's just take an example here, other versions should be about the same.

Steps to build

1. Decompress source code to a folder, let's call it `ROOT`
2. Open Developer Command Prompt for VS2013, and go to `ROOT` folder
3. Run `qmake WEBENGINE_CONFIG+=use_proprietary_codecs qtwebengine.pro`. We add this flag to enable mp3 support.
4. Run `nmake`

Note: Mp3 is not supported by QtWebEngine by default, due to license issue. Please make sure to get a license for the codec you added.

Read Build QtWebEngine from source online: <https://riptutorial.com/qt/topic/8718/build-qtwebengine-from-source>

Chapter 4: CMakeLists.txt for your Qt project

Examples

CMakeLists.txt for Qt 5

A minimal CMake project file that uses Qt5 can be:

```
cmake_minimum_required(VERSION 2.8.11)

project(myproject)

find_package(Qt5 5.7.0 REQUIRED COMPONENTS
    Core
)

set(CMAKE_AUTOMOC ON)

add_executable(${PROJECT_NAME}
    main.cpp
)

target_link_libraries(${PROJECT_NAME}
    Qt5::Core
)
```

`cmake_minimum_required` is called to set minimum required version for CMake. The minimum required version for this example to work is `2.8.11` -- previous versions of CMake need additional code for a target to use Qt.

`find_package` is called to search an installation of Qt5 with a given version -- `5.7.0` in the example -- and wanted components -- `Core` module in the example. For a list of available modules, see [Qt Documentation](#). Qt5 is marked as `REQUIRED` in this project. The path to the installation can be hinted by setting the variable `Qt5_DIR`.

`AUTOMOC` is a boolean specifying whether CMake will handle the Qt `moc` preprocessor automatically, i.e. without having to use the `QT5_WRAP_CPP()` macro.

Other "AUTOMOC-like" variables are:

- `AUTOUIIC`: a boolean specifying whether CMake will handle the Qt `uic` code generator automatically, i.e. without having to use the `QT5_WRAP_UI()` macro.
- `AUTORCC`: a boolean specifying whether CMake will handle the Qt `rcc` code generator automatically, i.e. without having to use the `QT5_ADD_RESOURCES()` macro.

`add_executable` is called to create an executable target from the given source files. The target is then linked to the listed Qt's modules with the command `target_link_libraries`. From CMake 2.8.11, `target_link_libraries` with Qt's imported targets handles linker parameters, as well as include directories and compiler options.

Read CMakeLists.txt for your Qt project online: <https://riptutorial.com/qt/topic/1991/cmakelists-txt-for-your-qt-project>

Chapter 5: Common Pitfalls

Examples

Using Qt::DirectConnection when receiver object doesn't receive signal

Some times you see a signal is emitted in sender thread but connected slot doesn't called (in other words it doesn't receive signal), you have asked about it and finally got that the connection type `Qt::DirectConnection` would fix it, so the problem found and everything is ok.

But generally this is bad idea to use `Qt::DirectConnection` until you really know what is this and there is no other way. Lets explain it more, Each thread created by Qt (including main thread and new threads created by `QThread`) have Event loop, the event loop is responsible for receiving signals and call appropriate slots in its thread. Generally executing a blocking operation inside an slot is bad practice, because it blocks the event loop of that threads so no other slots would be called.

If you block an event loop (by making very time consuming or blocking operation) you will not receive events on that thread until the event loop will be unblocked. If the blocking operation, blocks the event loop forever (such as busy while), the slots could never be called.

In this situation you may set the connection type in connect to `Qt::DirectConnection`, now the slots will be called even the event loop is blocked. so how this could make broke everything? In `Qt::DirectConnection` Slots will be called in emitter threads, and not receiver threads and it can broke data synchronizations and ran into other problems. So never use `Qt::DirectConnection` unless you know what are you doing. If your problem will be solved by using `Qt::DirectConnection`, you have to carefull and look at your code and finding out why your event loop is blocked. Its not a good idea to block the event loop and its not recommended in Qt.

Here is small example which shows the problem, as you can see the `nonBlockingSlot` would be called even the `blockingSlot` blocked event loop with `while(1)` which indicates bad coding

```
class TestReceiver : public QObject{
    Q_OBJECT
public:
    TestReceiver(){
        qDebug() << "TestReceiver Constructed in" << QThread::currentThreadId();
    }
public slots:
    void blockingSlot()
    {
        static bool firstInstance = false;
        qDebug() << "Blocking slot called in thread" << QThread::currentThreadId();
        if(!firstInstance){
            firstInstance = true;
            while(1);
        }
    }
    void nonBlockingSlot(){
        qDebug() << "Non-blocking slot called" << QThread::currentThreadId();
    }
}
```

```

    }
};

class TestSender : public QObject{
    Q_OBJECT
public:
    TestSender(TestReceiver * receiver){
        this->nonBlockingTimer.setInterval(100);
        this->blockingTimer.setInterval(100);

        connect(&this->blockingTimer, &QTimer::timeout, receiver,
&TestReceiver::blockingSlot);
        connect(&this->nonBlockingTimer, &QTimer::timeout, receiver,
&TestReceiver::nonBlockingSlot, Qt::DirectConnection);
        this->nonBlockingTimer.start();
        this->blockingTimer.start();
    }
private:
    QTimer nonBlockingTimer;
    QTimer blockingTimer;
};

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    TestReceiver TestReceiverInstance;
    TestSender testSenderInstance(&TestReceiverInstance);
    QThread receiverThread;
    TestReceiverInstance.moveToThread(&receiverThread);
    receiverThread.start();

    return a.exec();
}

```

Read Common Pitfalls online: <https://riptutorial.com/qt/topic/8238/common-pitfalls>

Chapter 6: Communication between QML and C++

Introduction

We may use QML to build hybrid applications, since it's much more easier than C++. So we should know how they communicate with each other.

Examples

Call C++ in QML

Register C++ classes in QML

At C++ side, imagine we have a class named `QmlCppBridge`, it implements a method called `printHello()`.

```
class QmlCppBridge : public QObject
{
    Q_OBJECT
public:
    Q_INVOKABLE static void printHello() {
        qDebug() << "Hello, QML!";
    }
};
```

We want to use it in QML side. We should register the class by calling `qmlRegisterType()`:

```
// Register C++ class as a QML module, 1 & 0 are the major and minor version of the QML module
qmlRegisterType<QmlCppBridge>("QmlCppBridge", 1, 0, "QmlCppBridge");
```

In QML, use following code to call it:

```
import QmlCppBridge 1.0    // Import this module, so we can use it in our QML script

QmlCppBridge {
    id: bridge
}
bridge.printHello();
```

Using `QQmlContext` to inject C++ classes or variables to QML

We still use the C++ class in previous example:

```
QQmlApplicationEngine engine;
QQmlContext *context = engine.rootContext();
```

```
// Inject C++ class to QML
context->setContextProperty(QStringLiteral("qmlCppBridge"), new QmlCppBridge(&engine));

// Inject C++ variable to QML
QString demoStr = QStringLiteral("demo");
context->setContextProperty(QStringLiteral("demoStr"), demoStr);
```

At QML side:

```
qmlCppBridge.printHello();    // Call to C++ function
str: demoStr                  // Fetch value of C++ variable
```

Note: This example is based on Qt 5.7. Not sure if it fits earlier Qt versions.

Call QML in C++

To call the QML classes in C++, you need to set the `objectName` property.

In your Qml:

```
import QtQuick.Controls 2.0

Button {
    objectName: "buttonTest"
}
```

Then, in your C++, you can get the object with `QObject::findChild<QObject*>(QString)`

Like that:

```
QQmlApplicationEngine engine;
QQmlComponent component(&engine, QUrl(QLatin1String("qrc:/main.qml")));

QObject *mainPage = component.create();
QObject* item = mainPage->findChild<QObject*>("buttonTest");
```

Now you have your QML object in your C++. But that could seems useless since we cannot really get the components of the object.

However, we can use it to send **signals** between the QML and the C++. To do that, you need to add a signal in your QML file like that: `signal buttonClicked(string str)`. Once you create this, you need to emit the signal. For example:

```
import QtQuick 2.0
import QtQuick.Controls 2.1

Button {
    id: buttonTest
    objectName: "buttonTest"

    signal clickedButton(string str)
    onClicked: {
        buttonTest.clickedButton("clicked !")
    }
}
```

```
}  
}
```

Here we have our qml button. When we click on it, it goes to the **onClicked** method (a base method for buttons which is called when you press the button). Then we use the id of the button and the name of the signal to emit the signal.

And in our cpp, we need to connect the signal with a slot. like that:

main.cpp

```
#include <QGuiApplication>  
#include <QQmlApplicationEngine>  
#include <QQmlComponent>  
  
#include "ButtonManager.h"  
  
int main(int argc, char *argv[])  
{  
    QCoreApplication::setAttribute(Qt::AA_EnableHighDpiScaling);  
    QGuiApplication app(argc, argv);  
  
    QQmlApplicationEngine engine;  
    QQmlComponent component(&engine, QUrl(QLatin1String("qrc:/main.qml")));  
  
    QObject *mainPage = component.create();  
    QObject* item = mainPage->findChild<QObject*>("buttonTest");  
  
    ButtonManager buttonManager(mainPage);  
    QObject::connect(item, SIGNAL(clickedButton(QString)), &buttonManager,  
        SLOT(onButtonClicked(QString)));  
  
    return app.exec();  
}
```

As you can see, we get our qml button with `findChild` as before and we connect the signal to a Button manager which is a class created and who look like that. ButtonManager.h

```
#ifndef BUTTONMANAGER_H  
#define BUTTONMANAGER_H  
  
#include <QObject>  
  
class ButtonManager : public QObject  
{  
    Q_OBJECT  
public:  
    ButtonManager(QObject* parent = nullptr);  
public slots:  
    void onButtonClicked(QString str);  
};  
  
#endif // BUTTONMANAGER_H
```

ButtonManager.cpp


```
#include "ButtonManager.h"
#include <QDebug>

ButtonManager::ButtonManager(QObject *parent)
    : QObject(parent)
{

}

void ButtonManager::onButtonClicked(QString str)
{
    qDebug() << "button: " << str;
}
```

So when the signal will be received, it will call the method `onButtonClicked` which will write "button: clicked !"

output:



Hello World

Application Output



AndroidTest



button: "clicked !"

button: "clicked !"

button: "clicked !"

button: "clicked !"

D:\Projects\build-Andr

Starting D:\Projects\k

QML debugging is enabl

button: "clicked !"

button: "clicked !"

button: "clicked !"

button: "clicked !"

button: "clicked !"

<https://riptutorial.com/qt/topic/8735/communication-between-qml-and-cplusplus>

Chapter 7: Deploying Qt applications

Examples

Deploying on windows

Qt provides a deployment tool for Windows: `windeployqt`. The tool inspects a Qt application executable for its dependencies to Qt modules and creates a deployment directory with the necessary Qt files to run the inspected executable. A possible script may look like:

```
set PATH=%PATH%;<qt_install_prefix>/bin
windeployqt --dir /path/to/deployment/dir /path/to/qt/application.exe
```

The `set` command is called to add Qt's `bin` directory to the `PATH` environment variable. `windeployqt` is then called:

- The path to the deployment directory is given an optional argument given with the parameter `--dir` (default is the path where `windeployqt` is called).
- The path to the executable to be inspected is given as last argument.

The deployment directory can then be bundled with the executable.

NOTE:

If you are using pre-compiled Qt5.7.0 with vs2013 on Windows (*not sure if all versions has this issue*), there is a chance, that you need to manually copy `<QTDIR>\5.7\msvc2015\qml` to your `bin` directory of your program. Otherwise the program will auto quit after start.

See also [Qt documentation](#).

Integrating with CMake

It is possible to run `windeployqt` and `macdeployqt` from CMake, but first the path to the executables must be found:

```
# Retrieve the absolute path to qmake and then use that path to find
# the binaries
get_target_property(_qmake_executable Qt5::qmake IMPORTED_LOCATION)
get_filename_component(_qt_bin_dir "${_qmake_executable}" DIRECTORY)
find_program(WINDEPLOYQT_EXECUTABLE windeployqt HINTS "${_qt_bin_dir}")
find_program(MACDEPLOYQT_EXECUTABLE macdeployqt HINTS "${_qt_bin_dir}")
```

In order for `windeployqt` to find the Qt libraries in their installed location, the folder must be added to `%PATH%`. To do this for a target named `myapp` after being built:

```
add_custom_command(TARGET myapp POST_BUILD
    COMMAND "${CMAKE_COMMAND}" -E
    env PATH="${_qt_bin_dir}" "${WINDEPLOYQT_EXECUTABLE}"
```

```

    "$<TARGET_FILE:myapp>"
    COMMENT "Running windeployqt..."
)

```

For running `macdeployqt` on a bundle, it would be done this way:

```

add_custom_command(TARGET myapp POST_BUILD
    COMMAND "${MACDEPLOYQT_EXECUTABLE}"
        "$<TARGET_FILE_DIR:myapp>/../.."
        -always-overwrite
    COMMENT "Running macdeployqt..."
)

```

Deploying on Mac

Qt offers a deployment tool for Mac: The Mac Deployment Tool.

The Mac deployment tool can be found in `QTDIR/bin/macdeployqt`. It is designed to automate the process of creating a deployable application bundle that contains the Qt libraries as private frameworks.

The mac deployment tool also deploys the Qt plugins, according to the following rules (unless **-no-plugins option is used**):

- The platform plugin is always deployed.
- Debug versions of the plugins are not deployed.
- The designer plugins are not deployed.
- The image format plugins are always deployed.
- The print support plugin is always deployed.
- SQL driver plugins are deployed if the application uses the Qt SQL module.
- Script plugins are deployed if the application uses the Qt Script module.
- The SVG icon plugin is deployed if the application uses the Qt SVG module.
- The accessibility plugin is always deployed.

To include a 3rd party library in the application bundle, copy the library into the bundle manually, after the bundle is created.

To use `macdeployqt` tool you can open terminal and type:

```
$ QTDIR/bin/macdeployqt <path to app file generated by build>/appFile.app
```

The app file will now contain all the Qt Libraries used as private frameworks.

`macdeployqt` also supports the following options

Option	Description
<code>-verbose=<0-3></code>	0 = no output, 1 = error/warning (default), 2 = normal, 3 = debug
<code>-no-plugins</code>	Skip plugin deployment

Option	Description
-dmg	Create a .dmg disk image
-no-strip	Don't run 'strip' on the binaries
-use-debug-libs	Deploy with debug versions of frameworks and plugins (implies -no-strip)
-executable=	Let the given executable also use the deployed frameworks
-qmlidir=	Deploy imports used by .qml files in the given path

Detailed informations can be found on [Qt Documentation](#)

Deploying on linux

There is a deployment tool for linux on [GitHub](#). While not perfect, it is linked to from the Qt wiki. It's based conceptually on the Qt Mac Deployment Tool and functions similarly by providing an [AppImage](#).

Given that a desktop file should be provided with an AppImage, `linuxdeployqt` can use that to determine the parameters of the build.

```
linuxdeployqt ./path/to/appdir/usr/share/application_name.desktop
```

Where the [desktop file](#) specifies the executable to be run (with `EXEC=`), the name of the application, and an icon.

Read Deploying Qt applications online: <https://riptutorial.com/qt/topic/5857/deploying-qt-applications>

Chapter 8: Header on QListView

Introduction

The QListView widget is part of the Model/View programming mechanisms of Qt. Basically, it allows to display items stored in a Model under the form of a list. In this topic we will not get deep into the Model/View mechanisms of Qt, but rather focus on the graphical aspect of one View widget: the QListView, and especially how to add a header on top of this object through the use of the QPaintEvent object.

Examples

Custom QListView declaration

```
/*!
 * \class MainMenuListView
 * \brief The MainMenuListView class is a QListView with a header displayed
 *        on top.
 */
class MainMenuListView : public QListView
{
    Q_OBJECT

    /*!
     * \class Header
     * \brief The header class is a nested class used to display the header of a
     *        QListView. On each instance of the MainMenuListView, a header will
     *        be displayed.
     */
    class Header : public QWidget
    {
    public:
        /*!
         * \brief Constructor used to defined the parent/child relation
         *        between the Header and the QListView.
         * \param parent Parent of the widget.
         */
        Header(MainMenuListView* parent);

        /*!
         * \brief Overridden method which allows to get the recommended size
         *        for the Header object.
         * \return The recommended size for the Header widget.
         */
        QSize sizeHint() const;

    protected:
        /*!
         * \brief Overridden paint event which will allow us to design the
         *        Header widget area and draw some text.
         * \param event Paint event.
         */
        void paintEvent(QPaintEvent* event);
    };
};
```

```

private:
    MainMenuListView* menu;    /*!< The parent of the Header. */
};

public:
    /*!
    * \brief Constructor allowing to instanciate the customized QListView.
    * \param parent Parent widget.
    * \param header Text which has to be displayed in the header
    *           (Header by default)
    */
    MainMenuListView(QWidget* parent = nullptr, const QString& header = QString("Header"));

    /*!
    * \brief Catches the Header paint event and draws the header with
    *           the specified text in the constructor.
    * \param event Header paint event.
    */
    void headerAreaPaintEvent(QPaintEvent* event);

    /*!
    * \brief Gets the width of the List widget.
    *           This value will also determine the width of the Header.
    * \return The width of the custom QListView.
    */
    int headerAreaWidth();

protected:
    /*!
    * \brief Overridden method which allows to resize the Header.
    * \param event Resize event.
    */
    void resizeEvent(QResizeEvent* event);

private:
    QWidget*    headerArea;    /*!< Header widget. */
    QString     headerText;    /*!< Header title. */
};

```

Implementation of the custom QListView

```

QSize MainMenuListView::Header::sizeHint() const
{
    // fontmetrics() allows to get the default font size for the widget.
    return QSize(menu->headerAreaWidth(), fontMetrics().height());
}

void MainMenuListView::Header::paintEvent(QPaintEvent* event)
{
    // Catches the paint event in the parent.
    menu->headerAreaPaintEvent(event);
}

MainMenuListView::MainMenuListView(QWidget* parent, const QString& header) :
QListView(parent), headerText(header)
{
    headerArea = new Header(this);
}

```



```

    // Really important. The view port margins define where the content
    // of the widget begins.
    setViewportMargins(0, fontMetrics().height(), 0, 0);
}

void MainMenuListView::headerAreaPaintEvent(QPaintEvent* event)
{
    // Paints the background of the header in gray.
    QPainter painter(headerArea);
    painter.fillRect(event->rect(), Qt::lightGray);

    // Display the header title in black.
    painter.setPen(Qt::black);

    // Writes the header aligned on the center of the widget.
    painter.drawText(0, 0, headerArea->width(), fontMetrics().height(), Qt::AlignCenter,
headerText);
}

int MainMenuListView::headerAreaWidth()
{
    return width();
}

void MainMenuListView::resizeEvent(QResizeEvent* event)
{
    // Executes default behavior.
    QListView::resizeEvent(event);

    // Really important. Allows to fit the parent width.
    headerArea->adjustSize();
}

```

Use case: MainWindow declaration

```

class MainMenuListView;

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    MainWindow(QWidget* parent = 0);
    ~MainWindow();

private:
    MainMenuListView* menuA;
    MainMenuListView* menuB;
    MainMenuListView* menuC;
};

```

Use case: Implementation

```

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent)
{
    QWidget* w = new QWidget(this);

```

```

QHBoxLayout* hbox = new QHBoxLayout();

QVBoxLayout* vbox = new QVBoxLayout();
menuA = new MainMenuListView(w, "Images");
menuB = new MainMenuListView(w, "Videos");
menuC = new MainMenuListView(w, "Devices");
vbox->addWidget(menuA);
vbox->addWidget(menuB);
vbox->addWidget(menuC);
vbox->setSpacing(0);
hbox->addLayout(vbox);

QPlainTextEdit* textEdit = new QPlainTextEdit(w);
hbox->addWidget(textEdit);

w->setLayout(hbox);
setCentralWidget(w);

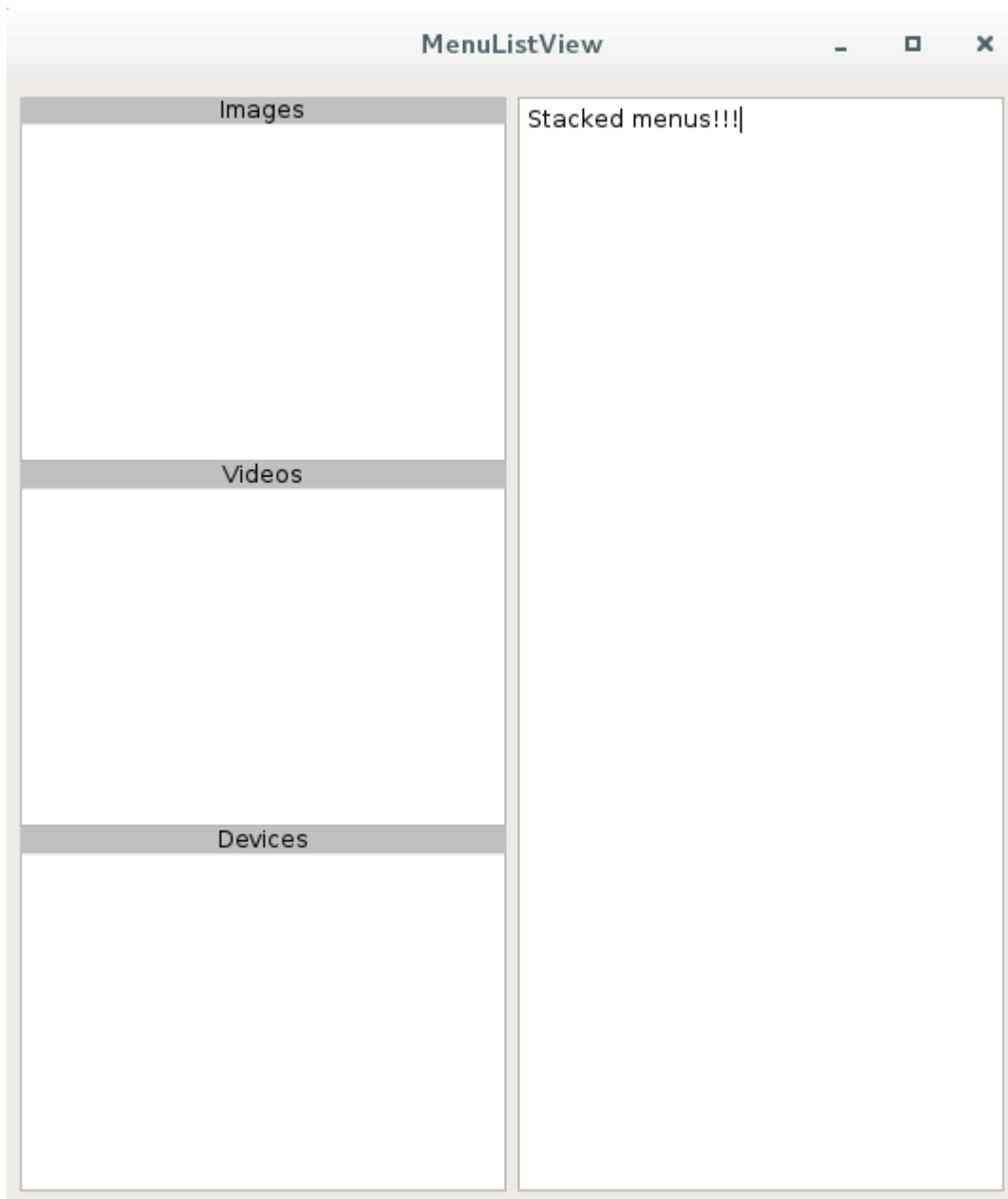
move((QApplication::desktop()->screenGeometry().width() / 2) - (size().width() / 2),
      (QApplication::desktop()->screenGeometry().height() / 2) - (size().height() / 2));
}

MainWindow::~MainWindow() {}

```

Use case: Sample output

Here is a sample output:



As you can see above, it can be useful for creating stacked menus. Note that this sample is trivial. The two widgets have the same size constraints.

Read Header on QListView online: <https://riptutorial.com/qt/topic/9382/header-on-qlistview>

Chapter 9: Implicit sharing

Remarks

STL style iterators on Qt Container can have some negative side effect due to the implicit-sharing. It is advised to avoid copying a Qt container while you have iterators active on them.

```
QVector<int> a,b; //2 vectors
a.resize(1000);
b = a; // b and a now point to the same memory internally

auto iter = a.begin(); //iter also points to the same memory a and b do
a[4] = 1; //a creates a new copy and points to different memory.
//Warning 1: b and iter point sill to the same even if iter was "a.begin()"

b.clear(); //delete b-memory
//Warning 2: iter only holds a pointer to the memory but does not increase ref-count.
//           so now the memory iter points to is invalid. UB!
```

Examples

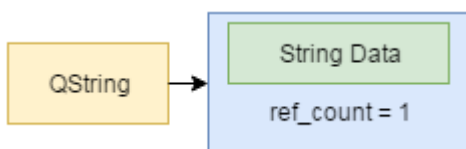
Basic Concept

Several Qt Objects and Containers use a concept calles **implicit sharing**, which can also be refered to as **copy-on-write**.

Implicit sharing means that the classes who use this concept share the same data on initialization.

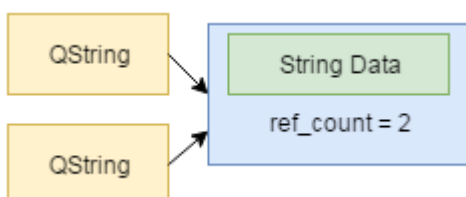
One of these classes to use the concept is QString.

```
QString s1("Hello World");
```



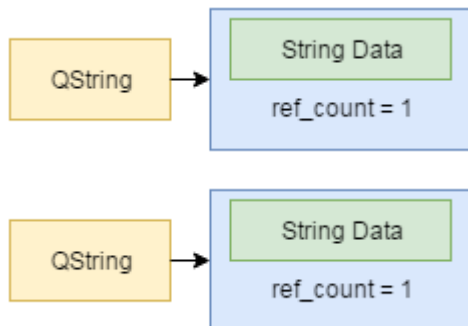
This is a simplified model of a QString. Internally it has a memory block, with the actual string data and and a reference counter.

```
QString s2 = s1;
```



If we now copy this `QString` both objects will internally point to the same content, thus avoiding unnecessary copy operations. Note how the reference count also got upped. So in case the first string gets deleted the shared-data still knows it is referenced by another `QString`.

```
s2 += " and all the other Worlds!"
```



Now when the `QString` is actually modified the object "detaches" itself from the memory block, copying its content and modifies the content.

Read Implicit sharing online: <https://riptutorial.com/qt/topic/6801/implicit-sharing>

Chapter 10: Model/View

Examples

A Simple Read-only Table to View Data from a Model

This is a simple example to display read-only data that is tabular in nature using Qt's [Model/View Framework](#). Specifically, the Qt Objects [QAbstractTableModel](#) (sub-classed in this example) and [QTableView](#) are used.

Implementations of the methods [rowCount\(\)](#), [columnCount\(\)](#), [data\(\)](#) and [headerData\(\)](#) are required to give the [QTableView](#) object a means to obtain information about the data contained in the [QAbstractTableModel](#) object.

The method `populateData()` was added to this example to provide a means to populate the [QAbstractTableModel](#) object with data from some arbitrary source.

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H
#include <QMainWindow>
#include <QAbstractTableModel>

namespace Ui {
    class MainWindow;
}

class TestModel : public QAbstractTableModel
{
    Q_OBJECT

public:
    TestModel(QObject *parent = 0);

    void populateData(const QList<QString> &contactName, const QList<QString> &contactPhone);

    int rowCount(const QModelIndex &parent = QModelIndex()) const Q_DECL_OVERRIDE;
    int columnCount(const QModelIndex &parent = QModelIndex()) const Q_DECL_OVERRIDE;

    QVariant data(const QModelIndex &index, int role = Qt::DisplayRole) const Q_DECL_OVERRIDE;
    QVariant headerData(int section, Qt::Orientation orientation, int role = Qt::DisplayRole)
const Q_DECL_OVERRIDE;

private:
    QList<QString> tm_contact_name;
    QList<QString> tm_contact_phone;
};

class MainWindow : public QMainWindow
{
    Q_OBJECT
```

```

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;

};

#endif // MAINWINDOW_H

```

mainwindow.cpp

```

#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    QList<QString> contactNames;
    QList<QString> contactPhoneNums;

    // Create some data that is tabular in nature:
    contactNames.append("Thomas");
    contactNames.append("Richard");
    contactNames.append("Harrison");
    contactPhoneNums.append("123-456-7890");
    contactPhoneNums.append("222-333-4444");
    contactPhoneNums.append("333-444-5555");

    // Create model:
    TestModel *PhoneBookModel = new TestModel(this);

    // Populate model with data:
    PhoneBookModel->populateData(contactNames, contactPhoneNums);

    // Connect model to table view:
    ui->tableView->setModel(PhoneBookModel);

    // Make table header visible and display table:
    ui->tableView->horizontalHeader()->setVisible(true);
    ui->tableView->show();
}

MainWindow::~MainWindow()
{
    delete ui;
}

TestModel::TestModel(QObject *parent) : QAbstractTableModel(parent)
{
}

// Create a method to populate the model with data:
void TestModel::populateData(const QList<QString> &contactName, const QList<QString>
&contactPhone)
{

```

```

    tm_contact_name.clear();
    tm_contact_name = contactName;
    tm_contact_phone.clear();
    tm_contact_phone = contactPhone;
    return;
}

int TestModel::rowCount(const QModelIndex &parent) const
{
    Q_UNUSED(parent);
    return tm_contact_name.length();
}

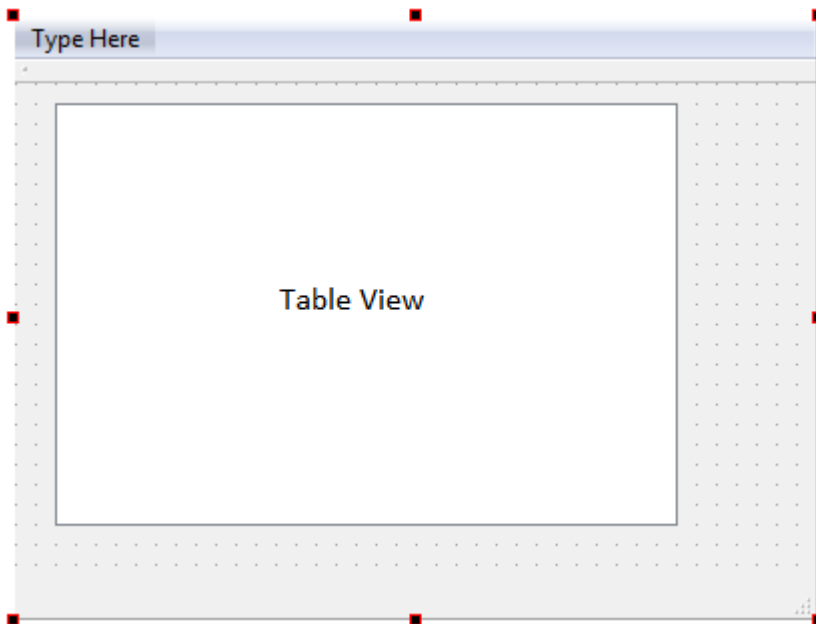
int TestModel::columnCount(const QModelIndex &parent) const
{
    Q_UNUSED(parent);
    return 2;
}

QVariant TestModel::data(const QModelIndex &index, int role) const
{
    if (!index.isValid() || role != Qt::DisplayRole) {
        return QVariant();
    }
    if (index.column() == 0) {
        return tm_contact_name[index.row()];
    } else if (index.column() == 1) {
        return tm_contact_phone[index.row()];
    }
    return QVariant();
}

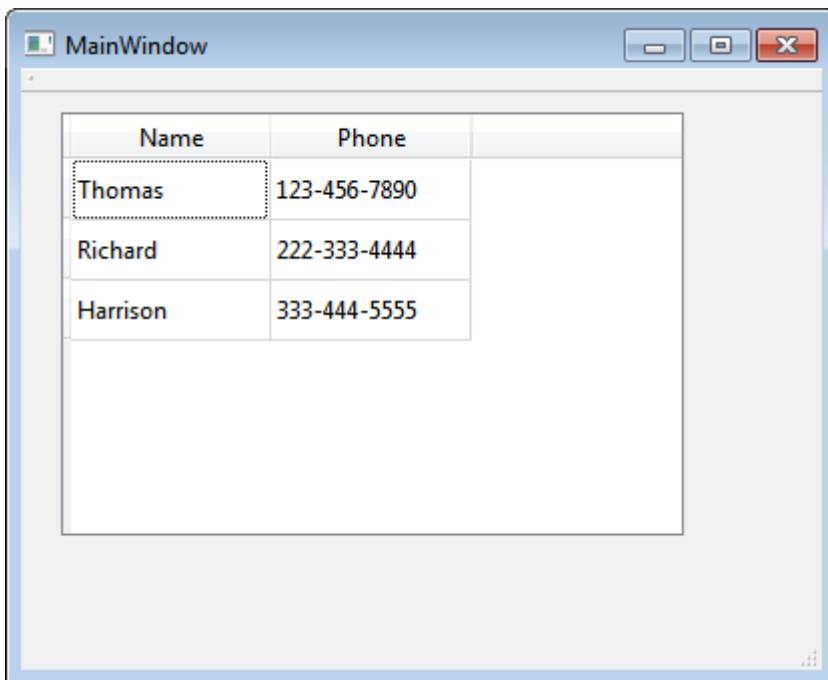
QVariant TestModel::headerData(int section, Qt::Orientation orientation, int role) const
{
    if (role == Qt::DisplayRole && orientation == Qt::Horizontal) {
        if (section == 0) {
            return QString("Name");
        } else if (section == 1) {
            return QString("Phone");
        }
    }
    return QVariant();
}

```

Using Qt Creator/Design, place a Table View object, named **tableView** in this example, in the **main window**:



The resulting program displays as:



A simple tree model

`QModelIndex` does not actually know about it's parent/child indexes, it only contains a **row**, a **column** and a **pointer**, and it is the models responsibility to use this data to provide information an index's relations. The model therefore needs to do a lot of conversions from the `void*` stored inside the `QModelIndex` to an internal data type and back.

TreeModel.h:

```
#pragma once

#include <QAbstractItemModel>

class TreeModel : public QAbstractItemModel
```

```

{
    Q_OBJECT
public:
    explicit TreeModel(QObject *parent = nullptr);

    // Reimplementation of QAbstractItemModel methods
    int rowCount(const QModelIndex &index) const override;
    int columnCount(const QModelIndex &index) const override;
    QModelIndex index(const int row, const int column,
        const QModelIndex &parent) const override;
    QModelIndex parent(const QModelIndex &childIndex) const override;
    QVariant data(const QModelIndex &index, const int role) const override;
    bool setData(const QModelIndex &index, const QVariant &value,
        const int role) override;
    Qt::ItemFlags flags(const QModelIndex &index) const override;

    void addRow(const QModelIndex &parent, const QVector<QVariant> &values);
    void removeRow(const QModelIndex &index);

private:
    struct Item
    {
        ~Item();

        // This could individual members, or maybe some other object that
        // contains the data we want to display/edit
        QVector<QVariant> values;

        // It is this information that the model needs to be able to answer
        // questions like "What's the parent QModelIndex of this QModelIndex?"
        QVector<Item *> children;
        Item *parent = nullptr;

        // Convenience method that's used in several places
        int rowInParent() const;
    };
    Item *m_root;
};

```

TreeModel.cpp:

```

#include "TreeModel.h"

// Adapt this to own needs
static constexpr int COLUMNS = 3;

TreeModel::Item::~~Item()
{
    qDeleteAll(children);
}

int TreeModel::Item::rowInParent() const
{
    if (parent) {
        return parent->children.indexOf(const_cast<Item *>(this));
    } else {
        return 0;
    }
}

TreeModel::TreeModel(QObject *parent)

```

```

: QAbstractItemModel(parent), m_root(new Item) {}

int TreeModel::rowCount(const QModelIndex &parent) const
{
    // Parent being invalid means we ask for how many rows the root of the
    // model has, thus we ask the root item
    // If parent is valid we access the Item from the pointer stored
    // inside the QModelIndex
    return parent.isValid()
        ? static_cast<Item *>(parent.internalPointer())->children.size()
        : m_root->children.size();
}

int TreeModel::columnCount(const QModelIndex &parent) const
{
    return COLUMNS;
}

QModelIndex TreeModel::index(const int row, const int column,
    const QModelIndex &parent) const
{
    // hasIndex checks if the values are in the valid ranges by using
    // rowCount and columnCount
    if (!hasIndex(row, column, parent)) {
        return QModelIndex();
    }

    // In order to create an index we first need to get a pointer to the Item
    // To get started we have either the parent index, which contains a pointer
    // to the parent item, or simply the root item

    Item *parentItem = parent.isValid()
        ? static_cast<Item *>(parent.internalPointer())
        : m_root;

    // We can now simply look up the item we want given the parent and the row
    Item *childItem = parentItem->children.at(row);

    // There is no public constructor in QModelIndex we can use, instead we need
    // to use createIndex, which does a little bit more, like setting the
    // model() in the QModelIndex to the model that calls createIndex
    return createIndex(row, column, childItem);
}

QModelIndex TreeModel::parent(const QModelIndex &childIndex) const
{
    if (!childIndex.isValid()) {
        return QModelIndex();
    }

    // Simply get the parent pointer and create an index for it
    Item *parentItem = static_cast<Item*>(childIndex.internalPointer())->parent;
    return parentItem == m_root
        ? QModelIndex() // the root doesn't have a parent
        : createIndex(parentItem->rowInParent(), 0, parentItem);
}

QVariant TreeModel::data(const QModelIndex &index, const int role) const
{
    // Usually there will be more stuff here, like type conversion from
    // QVariant, handling more roles etc.
    if (!index.isValid() || role != Qt::DisplayRole) {
        return QVariant();
    }

```

```

    }
    Item *item = static_cast<Item *>(index.internalPointer());
    return item->values.at(index.column());
}
bool TreeModel::setData(const QModelIndex &index, const QVariant &value,
    const int role)
{
    // As in data there will usually be more stuff here, like type conversion to
    // QVariant, checking values for validity etc.
    if (!index.isValid() || role != Qt::EditRole) {
        return false;
    }
    Item *item = static_cast<Item *>(index.internalPointer());
    item->values[index.column()] = value;
    emit dataChanged(index, index, QVector<int>() << role);
    return true;
}
Qt::ItemFlags TreeModel::flags(const QModelIndex &index) const
{
    if (index.isValid()) {
        return Qt::ItemIsEnabled | Qt::ItemIsSelectable | Qt::ItemIsEditable;
    } else {
        return Qt::NoItemFlags;
    }
}

// Simple add/remove functions to illustrate {begin,end}{Insert,Remove}Rows
// usage in a tree model
void TreeModel::addRow(const QModelIndex &parent,
    const QVector<QVariant> &values)
{
    Item *parentItem = parent.isValid()
        ? static_cast<Item *>(parent.internalPointer())
        : m_root;
    beginInsertRows(parent,
        parentItem->children.size(), parentItem->children.size());
    Item *item = new Item;
    item->values = values;
    item->parent = parentItem;
    parentItem->children.append(item);
    endInsertRows();
}
void TreeModel::removeRow(const QModelIndex &index)
{
    if (!index.isValid()) {
        return;
    }
    Item *item = static_cast<Item *>(index.internalPointer());
    Q_ASSERT(item != m_root);
    beginRemoveRows(index.parent(), item->rowInParent(), item->rowInParent());
    item->parent->children.removeOne(item);
    delete item;
    endRemoveRows();
}

```

Read Model/View online: <https://riptutorial.com/qt/topic/3938/model-view>

Chapter 11: Multimedia

Remarks

Qt Multimedia is a module providing handling of multimedia (audio, video) and also camera and radio functionality.

However, the supported files of QMediaPlayer depends on the platform. Indeed, on windows, QMediaPlayer uses DirectShow, on Linux, it uses GStreamer. So depending on the platform some files may work on Linux but not on Windows or the opposite.

Examples

Video Playback in Qt 5

Let's create very simple video player using QtMultimedia module of Qt 5.

In .pro file of your application you will need the following lines:

```
QT += multimedia multimediawidgets
```

Note that `multimediawidgets` is necessary for usage of `QVideoWidget`.

```
#include <QtMultimedia/QMediaPlayer>
#include <QtMultimedia/QMediaPlaylist>
#include <QtMultimediaWidgets/QVideoWidget>

QMediaPlayer *player;
QVideoWidget *videoWidget;
QMediaPlaylist *playlist;

player = new QMediaPlayer;

playlist = new QMediaPlaylist(player);
playlist->addMedia(QUrl::fromLocalFile("actualPathHere"));

videoWidget = new QVideoWidget;
player->setVideoOutput(videoWidget);

videoWidget->show();
player->play();
```

That's all - after launching the application (if necessary codecs are installed in the system), video file playback will be started.

The same way you can play video from URL in Internet, not just local file.

Audio Playback in Qt5

As this is an audio, we don't need a QVideoWidget. So we can do:

```
_player = new QMediaPlayer(this);
QUrl file = QUrl::fromLocalFile(QFileDialog::getOpenFileName(this, tr("Open Music"), "",
tr("")));
if (file.url() == "")
    return ;
_player->setMedia(file);
_player->setVolume(50);
_player->play();
```

in the .h:

```
QMediaPlayer *_player;
```

this will open a dialog where you can choose your music and it will play it.

Read Multimedia online: <https://riptutorial.com/qt/topic/7675/multimedia>

Chapter 12: QDialogs

Remarks

The QDialog class is the **base** class of dialog windows. A dialog window is a top-level window mostly used for short-term tasks and brief communications with the user. QDialogs may be **modal** or **modeless**.

Note that QDialog (and any other widget that has type Qt::Dialog) uses the parent widget slightly differently from other classes in Qt. A dialog is **always a top-level widget**, but if it has a parent, **its default location is centered on top of the parent's top-level widget** (if it is not top-level itself). It will also share the parent's taskbar entry.

A **modal** dialog is a dialog that blocks input to other visible windows in the same application. Dialogs that are used to request a file name from the user or that are used to set application preferences are usually modal. Dialogs can be **application modal** (the default) or **window modal**.

The most common way to display a modal dialog is to call its exec() function. When the user closes the dialog, exec() will provide a useful return value.

A **modeless** dialog is a dialog that operates independently of other windows in the same application. Modeless dialogs are displayed using show(), which returns control to the caller immediately.

Examples

MyCompareFileDialog.h

```
#ifndef MYCOMPAREFILEDIALOG_H
#define MYCOMPAREFILEDIALOG_H

#include <QtWidgets/QDialog>

class MyCompareFileDialog : public QDialog
{
    Q_OBJECT

public:
    MyCompareFileDialog(QWidget *parent = 0);
    ~MyCompareFileDialog();
};

#endif // MYCOMPAREFILEDIALOG_H
```

MyCompareFileDialogDialog.cpp

```
#include "MyCompareFileDialog.h"
```

```

#include <QLabel>

MyCompareFileDialog::MyCompareFileDialog(QWidget *parent)
: QDialog(parent)
{
    setWindowTitle("Compare Files");
    setWindowFlags(Qt::Dialog);
    setWindowModality(Qt::WindowModal);

    resize(300, 100);
    QSizePolicy sizePolicy(QSizePolicy::Preferred, QSizePolicy::Preferred);
    setSizePolicy(sizePolicy);
    setMinimumSize(QSize(300, 100));
    setMaximumSize(QSize(300, 100));

    QLabel* myLabel = new QLabel(this);
    myLabel->setText("My Dialog!");
}

MyCompareFileDialog::~MyCompareFileDialog()
{ }

```

MainWindow.h

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>

namespace Ui {
class MainWindow;
}

class MyCompareFileDialog;

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private:
    Ui::MainWindow *ui;
    MyCompareFileDialog* myDialog;
};

#endif // MAINWINDOW_H

```

MainWindow.cpp

```

#include "mainwindow.h"
#include "ui_mainwindow.h"
#include "mycomparefiledialog.h"

MainWindow::MainWindow(QWidget *parent) :

```



```

    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    myDialog = new MyCompareFileDialog(this);

    connect(ui->pushButton, SIGNAL(clicked()), myDialog, SLOT(exec()));
}

MainWindow::~MainWindow()
{
    delete ui;
}

```

main.cpp

```

#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}

```

mainwindow.ui

```

<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
    <class>MainWindow</class>
    <widget class="QMainWindow" name="MainWindow">
        <property name="geometry">
            <rect>
                <x>0</x>
                <y>0</y>
                <width>400</width>
                <height>300</height>
            </rect>
        </property>
        <property name="windowTitle">
            <string>MainWindow</string>
        </property>
        <widget class="QWidget" name="centralWidget">
            <widget class="QPushButton" name="pushButton">
                <property name="geometry">
                    <rect>
                        <x>140</x>
                        <y>80</y>
                        <width>111</width>
                        <height>23</height>
                    </rect>
                </property>
                <property name="text">

```

```

        <string>Show My Dialog</string>
    </property>
</widget>
</widget>
<widget class="QMenuBar" name="menuBar">
    <property name="geometry">
        <rect>
            <x>0</x>
            <y>0</y>
            <width>400</width>
            <height>21</height>
        </rect>
    </property>
</widget>
<widget class="QToolBar" name="mainToolBar">
    <attribute name="toolBarArea">
        <enum>TopToolBarArea</enum>
    </attribute>
    <attribute name="toolBarBreak">
        <bool>false</bool>
    </attribute>
</widget>
<widget class="QStatusBar" name="statusBar"/>
</widget>
<layoutdefault spacing="6" margin="11"/>
<resources/>
<connections/>
</ui>

```

Read QDialogs online: <https://riptutorial.com/qt/topic/7819/qdialogs>

Chapter 13: QGraphics

Examples

Pan, zoom, and rotate with QGraphicsView

`QGraphics` can be used to organize complicated scenes of visual objects into a framework that makes them easier to handle.

There are three major types of objects used in this framework `QGraphicsView`, `QGraphicsScene`, and `QGraphicsItems`. `QGraphicsItems` are the basic visual items that exist in the scene.

There are many types that are pre-built and can be used such as `Ellipses`, `Lines`, `Paths`, `Pixmap`s, `Polygons`, `Rectangles`, and `Text`.

You can also make your own items by inheriting `QGraphicsItem`. These items are then put into a `QGraphicsScene` which is basically the world you are planning to look at. The items can move within the scene which is like having them move in the world you are looking at. The items positioning and orientation is handled by transformation matrices called `QTransform`s. Qt has nice functions built in so you usually do not need to work with the `QTransform`s directly, instead you call functions such as `rotate` or `scale` which create the proper transforms for you. The scene is then viewed by the perspective defined in the `QGraphicsView` (again with `QTransform`s), which is the piece you would put into a widget in you UI.

In the following example there is a very simple scene with just one item (a pixmap), which is put into a scene and displayed in a view. By turning on the `DragMode` flag the scene can be panned around with the mouse and by using the `scale` and `rotate` functions it can be scaled in and out with the scroll on the mouse and rotated with the arrow keys.

If you would like to run this example create a instance of `View` that will be displayed and create a [resource](#) file with the prefix `/images` containing a image `my_image.png`.

```
#include <QGraphicsView>
#include <QGraphicsScene>
#include <QGraphicsPixmapItem>
#include <QWheelEvent>
#include <QKeyEvent>

class View : public QGraphicsView
{
    Q_OBJECT
public:
    explicit View(QWidget *parent = 0) :
        QGraphicsView(parent)
    {
        setDragMode(QGraphicsView::ScrollHandDrag);

        QGraphicsPixmapItem *pixmapItem = new
        QGraphicsPixmapItem(QPixmap(":/images/my_image.png"));
        pixmapItem->setTransformationMode(Qt::SmoothTransformation);
    }
};
```

```
    QGraphicsScene *scene = new QGraphicsScene();
    scene->addItem(pixmapItem);
    setScene(scene);
}

protected Q_SLOTS:
void wheelEvent(QWheelEvent *event)
{
    if(event->delta() > 0)
        scale(1.25, 1.25);
    else
        scale(0.8, 0.8);
}

void keyPressEvent(QKeyEvent *event)
{
    if(event->key() == Qt::Key_Left)
        rotate(1);
    else if(event->key() == Qt::Key_Right)
        rotate(-1);
}
};
```

Read QGraphics online: <https://riptutorial.com/qt/topic/7539/qgraphics>

Chapter 14: qmake

Examples

Default "pro" file.

qmake is a build automation tool, which is shipped with *Qt* framework. It does similar job to tools such as *CMake* or *GNU Autotools*, but it is designed to be used specifically with *Qt*. As such it is well integrated with *Qt* ecosystem, notably *Qt Creator* IDE.

If you start *Qt Creator* and select `File -> New File or Project -> Application -> Qt Widgets` application, *Qt Creator* will generate a project skeleton for you along with a "pro" file. The "pro" file is processed by *qmake* in order to generate files, which are in turn processed by underlying build systems (for example *GNU Make* or *nmake*).

If you named your project "myapp", then "myapp.pro" file will appear. Here's how such default file looks like, with comments, that describe each *qmake* variable, added.

```
# Tells build system that project uses Qt Core and Qt GUI modules.
QT      += core gui

# Prior to Qt 5 widgets were part of Qt GUI module. In Qt 5 we need to add Qt Widgets module.
greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

# Specifies name of the binary.
TARGET = myapp

# Denotes that project is an application.
TEMPLATE = app

# List of source files (note: Qt Creator will take care about this list, you don't need to
update is manually).
SOURCES += main.cpp\
           mainwindow.cpp

# List of header files (note: Qt Creator will take care about this list).
HEADERS  += mainwindow.h

# List of "ui" files for a tool called Qt Designer, which is embedded into Qt Creator in newer
versions of IDE (note: Qt Creator will take care about this list).
FORMS    += mainwindow.ui
```

Preserving source directory structure in a build (undocumented "object_parallel_to_source" option).

If you like to organize your project by keeping source files in different subdirectories, you should know that during a build *qmake* will not preserve this directory structure and it will keep all the ".o" files in a single build directory. This can be a problem if you had conflicting file names in different directories like following.

```
src/file1.cpp
src/plugin/file1.cpp
```

Now *qmake* will decide to create two "file1.o" files in a build directory, causing one of them to be overwritten by another. The build will fail. To prevent this you can add `CONFIG += object_parallel_to_source` configuration option to your "pro" file. This will tell *qmake* to generate build files that preserve your source directory structure. This way your build directory will reflect source directory structure and object files will be created in separate subdirectories.

```
src/file1.o
src/plugin/file1.o
```

Complete example.

```
QT += core
TARGET = myapp
TEMPLATE = app

CONFIG += object_parallel_to_source

SOURCES += src/file1.cpp \
           src/plugin/file1.cpp

HEADERS += src/plugin/file1.h
```

Note that `object_parallel_to_source` `CONFIG` option is **not officially documented**.

Simple Example (Linux)

Window.h

```
#include <QWidget>

class Window : public QWidget
{
    Q_OBJECT
public:
    Window(QWidget *parent = Q_NULLPTR) : QWidget(parent) {}
}
```

main.cpp

```
#include <QApplication>
#include "Window.h"

int main()
{
    QApplication app;
    Window window;
    window.show();
    return app.exec();
}
```

example.pro

```
# The QT variable controls what modules are included in compilation.
# Note that the 'core' and 'gui' modules are included by default.
# For widget-based GUI applications, the 'widgets' module needs to be added.
QT += widgets

HEADERS = Window.h # Everything added to the HEADER variable will be checked
                  # to see if moc needs to run on it, and it will be run if
                  # so.

SOURCES = main.cpp # Everything added to the SOURCES variable will be compiled
                  # and (in the simple example) added to the resulting
                  # executable.
```

Command Line

```
# Assuming you are in a folder that contains the above files.
> qmake          # You can also add the example.pro file if needed
> make           # qmake creates a Makefile, this runs make on it.
> ./example      # The name of the executable defaults to the .pro file name.
```

SUBDIRS example

The SUBDIRS ability of qmake can be used to compile a set of libraries, each of which depend on another. The example below is slightly convoluted to show variations with the SUBDIRS ability.

Directory Structure

Some of the following files will be omitted in the interest of brevity. They can be assumed to be the format as non-subdir examples.

```
project_dir/
-project.pro
-common.pri
-build.pro
-main.cpp
-logic/
----logic.pro
----some logic files
-gui/
----gui.pro
----gui files
```

project.pro

This is the main file that enables the example. This is also the file that would be called with qmake on the command line (see below).

```
TEMPLATE = subdirs # This changes to the subdirs function. You can't combine
                  # compiling code and the subdirs function in the same .pro
                  # file.

# By default, you assign a directory to the SUBDIRS variable, and qmake looks
```

```
# inside that directory for a <dirname>.pro file.
SUBDIRS = logic

# You can append as many items as desired.  You can also specify the .pro file
# directly if need be.
SUBDIRS += gui/gui.pro

# You can also create a target that isn't a subdirectory, or that refers to a
# different file(*).
SUBDIRS += build
build.file = build.pro # This specifies the .pro file to use
# You can also use this to specify dependencies.  In this case, we don't want
# the build target to run until after the logic and gui targets are complete.
build.depends = logic gui/gui.pro
```

(*) See [The reference documentation](#) for the other options for a subdirs target.

common.pri

```
#Includes common configuration for all subdirectory .pro files.
INCLUDEPATH += . ..
WARNINGS += -Wall

TEMPLATE = lib

# The following keeps the generated files at least somewhat separate
# from the source files.
UI_DIR = uics
MOC_DIR = mocs
OBJECTS_DIR = objs
```

logic/logic.pro

```
# Check if the config file exists
! include( ../common.pri ) {
    error( "Couldn't find the common.pri file!" )
}

HEADERS += logic.h
SOURCES += logic.cpp

# By default, TARGET is the same as the directory, so it will make
# liblogic.so (in linux).  Uncomment to override.
# TARGET = target
```

gui/gui.pro

```
! include( ../common.pri ) {
    error( "Couldn't find the common.pri file!" )
}

FORMS += gui.ui
HEADERS += gui.h
SOURCES += gui.cpp

# By default, TARGET is the same as the directory, so it will make
# libgui.so (in linux).  Uncomment to override.
```



```
# TARGET = target
```

build.pro

```
TEMPLATE = app

SOURCES += main.cpp

LIBS += -Llogic -Lgui -llogic -lgui

# This renames the resulting executable
TARGET = project
```

Command Line

```
# Assumes you are in the project_dir directory
> qmake project.pro # specific the .pro file since there are multiple here.
> make -n2 # This makes logic and gui concurrently, then the build Makefile.
> ./project # Run the resulting executable.
```

Library example

A simple example to make a library (rather than an executable, which is the default). `TEMPLATE` variable specifies type of the project you are making. `lib` option allows makefile to build a library.

library.pro

```
HEADERS += library.h
SOURCES += library.cpp

TEMPLATE = lib

# By default, qmake will make a shared library. Uncomment to make the library
# static.
# CONFIG += staticlib

# By default, TARGET is the same as the directory, so it will make
# liblibrary.so or liblibrary.a (in linux). Uncomment to override.
# TARGET = target
```

When you are building a library, you can add options `dll` (default), `staticlib` or `plugin` to `CONFIG`.

Creating a project file from existing code

If you have a directory with existing source files, you can use `qmake` with the `-project` option to create a project file.

Let's assume, the folder *MyProgram* contains the following files:

- `main.cpp`
- `foo.h`
- `foo.cpp`

- bar.h
- bar.cpp
- subdir/foobar.h
- subdir/foobar.cpp

Then by calling

```
qmake -project
```

a file *MyProgram.pro* is created with the following content:

```
#####
# Automatically generated by qmake (3.0) Mi. Sep. 7 23:36:56 2016
#####

TEMPLATE = app
TARGET = MyProgram
INCLUDEPATH += .

# Input
HEADERS += bar.h foo.h subdir/foobar.h
SOURCES += bar.cpp foo.cpp main.cpp subdir/foobar.cpp
```

The code can then be built as described in [this simple example](#).

Read qmake online: <https://riptutorial.com/qt/topic/4438/qmake>

Chapter 15: QObject

Remarks

`QObject` class is the base class for all Qt objects.

Examples

QObject example

`QObject` macro appears in private section of a class. `QObject` requires the class to be subclass of `QObject`. This macro is necessary for the class to declare its signals/slots and to use Qt meta-object system.

If Meta Object Compiler (MOC) finds class with `QObject`, it processes it and generates C++ source file containing meta object source code.

Here is the example of class header with `QObject` and signal/slots:

```
#include <QObject>

class MyClass : public QObject
{
    Q_OBJECT

public:

public slots:
    void setNumber(double number);

signals:
    void numberChanged(double number);

private:
}
```

qobject_cast

```
T qobject_cast(QObject *object)
```

A functionality which is added by deriving from `QObject` and using the `QObject` macro is the ability to use the `qobject_cast`.

Example:

```
class myObject : public QObject
{
    Q_OBJECT
    //...
```

```
};

QObject* obj = new myObject();
```

To check whether `obj` is a `myObject`-type and to cast it to such in C++ you can generally use a `dynamic_cast`. This is dependent on having RTTI enabled during compilation.

The `Q_OBJECT` macro on the other hands generates the conversion-checks and code which can be used in the `qobject_cast`.

```
myObject* my = qobject_cast<myObject*>(obj);
if(!myObject)
{
    //wrong type
}
```

This is not reliant of RTTI. And also allows you to cast across dynamic library boundaries (via Qt interfaces/plugins).

QObject Lifetime and Ownership

QObjects come with their own alternative lifetime concept compared to native C++'s raw, unique or shared pointers.

QObjects have the possibility to build an objecttree by declaring parent/child relationships.

The simplest way to declare this relationship is by passing the parent object in the constructor. As an lternative you can manually set the parent of a `QObject` by calling `setParent`. This is the only direction to declare this relationship. You cannot add a child to a parents class but only the other way round.

```
QObject parent;
QObject child* = new QObject(&parent);
```

When `parent` now gets deleted in stack-unwind `child` will also be deleted.

When we delete a `QObject` it will "unregister" itself form the parent object;

```
QObject parent;
QObject child* = new QObject(&parent);
delete child; //this causes no problem.
```

The same applies for stack variables:

```
QObject parent;
QObject child(&parent);
```

`child` will get deleted before `parent` during stack-unwind and unregister itself from it's parent.

Note: You can manually call `setParent` with a reverse order of declaration which **will** break the

automatic destruction.

Read QObject online: <https://riptutorial.com/qt/topic/6304/qobject>

Chapter 16: Qt - Dealing with Databases

Remarks

- You will need the Qt SQL plugin corresponding to the type given to `QSqlDatabase::addDatabase`
- If you don't have the required SQL plugin, Qt will warn you that it can't find the requested driver
- If you don't have the required SQL plugin you will have to compile them from the Qt source

Examples

Using a Database on Qt

In the Project.pro file we add :

```
CONFIG += sql
```

in MainWindow.h we write :

```
#include <QMainWindow>
#include <QSql>
#include <QDebug>

namespace Ui
{
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:

private:
    Ui::MainWindow *ui;
    QSqlDatabase db;
};
```

Now in MainWindow.cpp :

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
```

```

{
    ui->setupUi(this);

    db = QSqlDatabase::addDatabase("QT SQL DRIVER" , "CONNECTION NAME");
    db.setDatabaseName("DATABASE NAME");
    if(!db.open())
    {
        qDebug() << "Can't Connect to DB !";
    }
    else
    {
        qDebug() << "Connected Successfully to DB !";
        QSqlQuery query;
        query.prepare("QUERY TO BE SENT TO THE DB");
        if(!query.exec())
        {
            qDebug() << "Can't Execute Query !";
        }
        else
        {
            qDebug() << "Query Executed Successfully !";
        }
    }
}

MainWindow::~MainWindow()
{
    delete ui;
}

```

Qt - Dealing with Sqlite Databases

In the Project.pro file we add : CONFIG += sql

in MainWindow.h we write :

```

#include <QMainWindow>
#include <QSql>
#include <QDebug>

namespace Ui
{
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:

private:
    Ui::MainWindow *ui;
    QSqlDatabase db;
};

```

Now in MainWindow.cpp :

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    db = QSqlDatabase::addDatabase("QSQLITE" , "CONNECTION NAME");
    db.setDatabaseName("C:\\sqlite_db_file.sqlite");
    if(!db.open())
    {
        qDebug() << "Can't Connect to DB !";
    }
    else
    {
        qDebug() << "Connected Successfully to DB !";
        QSqlQuery query;
        query.prepare("SELECT name , phone , address FROM employees WHERE ID = 201");
        if(!query.exec())
        {
            qDebug() << "Can't Execute Query !";
        }
        else
        {
            qDebug() << "Query Executed Successfully !";
            while(query.next())
            {
                qDebug() << "Employee Name : " << query.value(0).toString();
                qDebug() << "Employee Phone Number : " << query.value(1).toString();
                qDebug() << "Employee Address : " << query.value(1).toString();
            }
        }
    }
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

Qt - Dealing with ODBC Databases

In the Project.pro file we add : CONFIG += sql

in MainWindow.h we write :

```
#include <QMainWindow>
#include <QSql>
#include <QDebug>

namespace Ui
{
    class MainWindow;
}
```



```

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:

private:
    Ui::MainWindow *ui;
    QSqlDatabase db;
};

```

Now in MainWindow.cpp :

```

#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    db = QSqlDatabase::addDatabase("QODBC" , "CONNECTION NAME");
    db.setDatabaseName("DRIVER={SQL Server};SERVER=localhost;DATABASE=WorkDatabase"); //
    "WorkDatabase" is the name of the database we want
    db.setUserName("sa"); // Set Login Username
    db.setPassword(""); // Set Password if required
    if(!db.open())
    {
        qDebug() << "Can't Connect to DB !";
    }
    else
    {
        qDebug() << "Connected Successfully to DB !";
        QSqlQuery query;
        query.prepare("SELECT name , phone , address FROM employees WHERE ID = 201");
        if(!query.exec())
        {
            qDebug() << "Can't Execute Query !";
        }
        else
        {
            qDebug() << "Query Executed Successfully !";
            while(query.next())
            {
                qDebug() << "Employee Name : " << query.value(0).toString();
                qDebug() << "Employee Phone Number : " << query.value(1).toString();
                qDebug() << "Employee Address : " << query.value(1).toString();
            }
        }
    }
}

MainWindow::~MainWindow()
{
}

```

```
    delete ui;
}
```

Qt - Dealing with in-memory Sqlite Databases

In the Project.pro file we add : CONFIG += sql

in MainWindow.h we write :

```
#include <QMainWindow>
#include <QSql>
#include <QDebug>

namespace Ui
{
    class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

private slots:

private:
    Ui::MainWindow *ui;
    QSqlDatabase db;
};
```

Now in MainWindow.cpp :

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);

    db = QSqlDatabase::addDatabase("QSQLITE" , "CONNECTION NAME");
    db.setDatabaseName(":memory:");
    if(!db.open())
    {
        qDebug() << "Can't create in-memory Database!";
    }
    else
    {
        qDebug() << "In-memory Successfully created!";
        QSqlQuery query;

        if (!query.exec("CREATE TABLE employees (ID INTEGER, name TEXT, phone TEXT, address TEXT)"))
    }
```

```

    {
        qDebug() << "Can't create table!";
        return;
    }
    if (!query.exec("INSERT INTO employees (ID, name, phone, address) VALUES (201, 'Bob',
'5555-5555', 'Antarctica')"))
    {
        qDebug() << "Can't insert record!";
        return;
    }

    qDebug() << "Database filling completed!";
    if(!query.exec("SELECT name , phone , address FROM employees WHERE ID = 201"))
    {
        qDebug() << "Can't Execute Query !";
        return;
    }
    qDebug() << "Query Executed Successfully !";
    while(query.next())
    {
        qDebug() << "Employee Name : " << query.value(0).toString();
        qDebug() << "Employee Phone Number : " << query.value(1).toString();
        qDebug() << "Employee Address : " << query.value(1).toString();
    }
}

MainWindow::~MainWindow()
{
    delete ui;
}

```

Remove Database connection correctly

If we want to remove some database connection from the list of database connections. we need to use `QSqlDatabase::removeDatabase()`, however it's a static function and the way it work is a little wired.

```

// WRONG WAY
QSqlDatabase db = QSqlDatabase::database("sales");
QSqlQuery query("SELECT NAME, DOB FROM EMPLOYEES", db);
QSqlDatabase::removeDatabase("sales"); // will output a warning

// "db" is now a dangling invalid database connection,
// "query" contains an invalid result set

```

The correct way that Qt Document suggest us is below.

```

{
    QSqlDatabase db = QSqlDatabase::database("sales");
    QSqlQuery query("SELECT NAME, DOB FROM EMPLOYEES", db);
}
// Both "db" and "query" are destroyed because they are out of scope
QSqlDatabase::removeDatabase("sales"); // correct

```

Read Qt - Dealing with Databases online: <https://riptutorial.com/qt/topic/1993/qt---dealing-with->

Chapter 17: Qt Container Classes

Remarks

Qt provides its own template container classes. They are all implicitly shared. They provide two kinds of iterators (Java style and STL style.)

Qt sequential containers include: QVector, QList, QLinkedList, QStack, QQueue.

Qt associative containers include: QMap, QMultiMap, QHash, QMultiHash, QSet.

Examples

QStack usage

`QStack<T>` is a template Qt class providing stack. Its analogue in STL is `std::stack`. It is last in, first out structure (LIFO).

```
QStack<QString> stack;
stack.push("First");
stack.push("Second");
stack.push("Third");
while (!stack.isEmpty())
{
    cout << stack.pop() << endl;
}
```

It will output: Third, Second, First.

`QStack` inherits from `QVector` so its implementation is quite different from STL. In STL `std::stack` is implemented as a wrapper to type passed as a template argument (deque by default). Still main operations are the same for `QStack` and for `std::stack`.

QVector usage

`QVector<T>` provides dynamic array template class. It provides better performance in most cases than `QList<T>` so it should be first choice.

It can be initialized in various ways:

```
QVector<int> vect;
vect << 1 << 2 << 3;

QVector<int> v {1, 2, 3, 4};
```

The latest involves initialization list.

```
QVector<QString> stringsVector;
```

```
stringsVector.append("First");
stringsVector.append("Second");
```

You can get *i*-th element of vector this way:

`v[i]` **or** `at[i]`

Make sure that *i* is valid position, even `at(i)` doesn't make a check, this is a difference from `std::vector`.

QLinkedList usage

In Qt you should use `QLinkedList` in case you need to implement [linked list](#).

It is fast to append, prepend, insert elements into `QLinkedList` - $O(1)$, but index lookup is slower than in `QList` or `QVector` - $O(n)$. This is normal taking into attention you have to iterate through nodes to find something in linked list.

Full algorithmic complexity table can be found [here](#).

Just to insert some elements into `QLinkedList` you can use operator `<<()`:

```
QLinkedList<QString> list;
list << "string1" << "string2" << "string3";
```

To insert elements in the middle of `QLinkedList` or modify all or some of its elements you can use Java style or STL style iterators. Here is a simple example how we multiply all the elements of `QLinkedList` by 2:

```
QLinkedList<int> integerList {1, 2, 3};
QLinkedList<int>::iterator it;
for (it = integerList.begin(); it != integerList.end(); ++it)
{
    *it *= 2;
}
```

QList

The `QList` class is a template class that provides lists. It stores items in a list that provides fast index-based access and index-based insertions and removals.

To insert items into the list, you can use `operator<<()`, `insert()`, `append()` or `prepend()`. For example:

operator<<()

```
QList<QString> list;
list << "one" << "two" << "three";
```

insert()

```
QList<QString> list;
list << "alpha" << "beta" << "delta";
list.insert(2, "gamma");
```

append()

```
QList<QString> list;
list.append("one");
list.append("two");
list.append("three");
```

prepend()

```
QList<QString> list;
list.prepend("one");
list.prepend("two");
list.prepend("three");
```

To access the item at a particular index position, you can use `operator[]()` or `at()`. `at()` may be faster than `operator[]()`, it never causes deep copy of container and should work in constant-time. Neither of them does argument-check. Examples:

```
if (list[0] == "mystring")
    cout << "mystring found" << endl;
```

Or

```
if (list.at(i) == "mystring")
    cout << "mystring found at position " << i << endl;
```

To remove items, there are functions such as `removeAt()`, `takeAt()`, `takeFirst()`, `takeLast()`, `removeFirst()`, `removeLast()`, or `removeOne()`. Examples:

takeFirst()

```
// takeFirst() removes the first item in the list and returns it
QList<QWidget *> list;
...
while (!list.isEmpty())
    delete list.takeFirst();
```

removeOne()

```
// removeOne() removes the first occurrence of value in the list
QList<QString> list;
list << "sun" << "cloud" << "sun" << "rain";
list.removeOne("sun");
```

To find all occurrences of a particular value in a list, you can use `indexOf()` or `lastIndexOf()`. Example:

indexOf()

```
int i = list.indexOf("mystring");  
if (i != -1)  
    cout << "First occurrence of mystring is at position " << i << endl;
```

Read Qt Container Classes online: <https://riptutorial.com/qt/topic/6303/qt-container-classes>

Chapter 18: Qt Network

Introduction

Qt Network provide tools to easily use many network protocols in your application.

Examples

TCP Client

To create a **TCP** connection in Qt, we will use [QTcpSocket](#). First, we need to connect with `connectToHost`.

So for example, to connect to a local tcp server: `_socket.connectToHost(QHostAddress("127.0.0.1"), 4242);`

Then, if we need to read datas from the server, we need to connect the signal `readyRead` with a slot. Like that:

```
connect(&_socket, SIGNAL(readyRead()), this, SLOT(onReadyRead()));
```

and finally, we can read the datas like that:

```
void MainWindow::onReadyRead()
{
    QByteArray datas = _socket.readAll();
    qDebug() << datas;
}
```

To write datas, you can use the `write(QByteArray)` method:

```
_socket.write(QByteArray("ok !\n"));
```

So a basic TCP Client can look like that:

main.cpp:

```
#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}
```

mainwindow.h:

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QTcpSocket>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

public slots:
    void onReadyRead();

private:
    Ui::MainWindow *ui;
    QTcpSocket _socket;
};

#endif // MAINWINDOW_H
```

mainwindow.cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

#include <QDebug>
#include <QHostAddress>

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow),
    _socket(this)
{
    ui->setupUi(this);
    _socket.connectToHost(QHostAddress("127.0.0.1"), 4242);
    connect(&_amp;socket, SIGNAL(readyRead()), this, SLOT(onReadyRead()));
}

MainWindow::~~MainWindow()
{
    delete ui;
}

void MainWindow::onReadyRead()
{
    QByteArray datas = _socket.readAll();
    qDebug() << datas;
    _socket.write(QByteArray("ok !\n"));
}
```

mainwindow.ui: (empty here)

```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
  <class>MainWindow</class>
  <widget class="QMainWindow" name="MainWindow">
    <property name="geometry">
      <rect>
        <x>0</x>
        <y>0</y>
        <width>400</width>
        <height>300</height>
      </rect>
    </property>
    <property name="windowTitle">
      <string>MainWindow</string>
    </property>
    <widget class="QWidget" name="centralWidget"/>
    <widget class="QMenuBar" name="menuBar">
      <property name="geometry">
        <rect>
          <x>0</x>
          <y>0</y>
          <width>400</width>
          <height>25</height>
        </rect>
      </property>
    </widget>
    <widget class="QToolBar" name="mainToolBar">
      <attribute name="toolBarArea">
        <enum>TopToolBarArea</enum>
      </attribute>
      <attribute name="toolBarBreak">
        <bool>false</bool>
      </attribute>
    </widget>
    <widget class="QStatusBar" name="statusBar"/>
  </widget>
  <layoutdefault spacing="6" margin="11"/>
  <resources/>
  <connections/>
</ui>
```

TCP Server

Create a **TCP server** in Qt is also very easy, indeed, the class [QTcpServer](#) already provide all we need to do the server.

First, we need to listen to any ip, a random port and do something when a client is connected. like that:

```
_server.listen(QHostAddress::Any, 4242);
connect(&_server, SIGNAL(newConnection()), this, SLOT(onNewConnection()));
```

Then, When here is a new connection, we can add it to the client list and prepare to read/write on the socket. Like that:

```

QTcpSocket *clientSocket = _server.nextPendingConnection();
connect(clientSocket, SIGNAL(readyRead()), this, SLOT(onReadyRead()));
connect(clientSocket, SIGNAL(stateChanged(QAbstractSocket::SocketState)), this,
SLOT(onSocketStateChanged(QAbstractSocket::SocketState)));
_sockets.push_back(clientSocket);

```

The `stateChanged(QAbstractSocket::SocketState)` allow us to remove the socket to our list when the client is disconnected.

So here a basic chat server:

main.cpp:

```

#include "mainwindow.h"
#include <QApplication>

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    MainWindow w;
    w.show();

    return a.exec();
}

```

mainwindow.h:

```

#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QTcpServer>
#include <QTcpSocket>

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

public slots:
    void onNewConnection();
    void onSocketStateChanged(QAbstractSocket::SocketState socketState);
    void onReadyRead();
private:
    Ui::MainWindow *ui;
    QTcpServer _server;
    QList<QTcpSocket*> _sockets;
};

#endif // MAINWINDOW_H

```

mainwindow.cpp:

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

#include <QDebug>
#include <QHostAddress>
#include <QAbstractSocket>

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow),
    _server(this)
{
    ui->setupUi(this);
    _server.listen(QHostAddress::Any, 4242);
    connect(&_server, SIGNAL(newConnection()), this, SLOT(onNewConnection()));
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::onNewConnection()
{
    QTcpSocket *clientSocket = _server.nextPendingConnection();
    connect(clientSocket, SIGNAL(readyRead()), this, SLOT(onReadyRead()));
    connect(clientSocket, SIGNAL(stateChanged(QAbstractSocket::SocketState)), this,
    SLOT(onSocketStateChanged(QAbstractSocket::SocketState)));

    _sockets.push_back(clientSocket);
    for (QTcpSocket* socket : _sockets) {
        socket->write(QByteArray::fromStdString(clientSocket->
peerAddress().toString().toStdString() + " connected to server !\n"));
    }
}

void MainWindow::onSocketStateChanged(QAbstractSocket::SocketState socketState)
{
    if (socketState == QAbstractSocket::UnconnectedState)
    {
        QTcpSocket* sender = static_cast<QTcpSocket*>(QObject::sender());
        _sockets.removeOne(sender);
    }
}

void MainWindow::onReadyRead()
{
    QTcpSocket* sender = static_cast<QTcpSocket*>(QObject::sender());
    QByteArray datas = sender->readAll();
    for (QTcpSocket* socket : _sockets) {
        if (socket != sender)
            socket->write(QByteArray::fromStdString(sender->
peerAddress().toString().toStdString() + ": " + datas.toStdString()));
    }
}
```

(use the same mainwindow.ui that the previous example)

Read Qt Network online: <https://riptutorial.com/qt/topic/9683/qt-network>

Chapter 19: Qt Resource System

Introduction

The Qt Resource system is a way to embed files within your project. Each resource file can have one or more *prefixes* and each *prefix* can have files in it.

Each file in the resources is a link to a file on the file system. When the executable is built, the files are bundled into the executable, so the original file does not need to be distributed with the binary.

Examples

Referencing files within code

Let's say that inside a resources file, you had a file called `/icons/ok.png`

The full url of this file within code is `qrc:/icons/ok.png`. In most cases, this can be shortened to `:/icons/ok.png`

For example, if you wanted to create a `QIcon` and set it as the icon of a button from that file, you could use

```
QIcon icon(":/icons/ok.png"); //Alternatively use qrc:/icons/ok.png
ui->pushButton->setIcon(icon);
```

Read Qt Resource System online: <https://riptutorial.com/qt/topic/8776/qt-resource-system>

Chapter 20: QTimer

Remarks

QTimer can also be used to request a function to run as soon as the event loop has processed all the other pending events. To do this, use an interval of 0 ms.

```
// option 1: Set the interval to 0 explicitly.
QTimer *timer = new QTimer;
timer->setInterval( 0 );
timer->start();

// option 2: Passing 0 with the start call will set the interval as well.
QTimer *timer = new QTimer;
timer->start( 0 );

// option 3: use QTimer::singleShot with interval 0
QTimer::singleShot(0, [](){
    // do something
});
```

Examples

Simple example

The following example shows how to use a `QTimer` to call a slot every 1 second.

In the example, we use a `QProgressBar` to update its value and check the timer is working properly.

main.cpp

```
#include <QApplication>

#include "timer.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    Timer timer;
    timer.show();

    return app.exec();
}
```

timer.h

```
#ifndef TIMER_H
#define TIMER_H

#include <QWidget>
```



```

class QProgressBar;

class Timer : public QWidget
{
    Q_OBJECT

public:
    Timer(QWidget *parent = 0);

public slots:
    void updateProgress();

private:
    QProgressBar *progressBar;
};

#endif

```

timer.cpp

```

#include <QLayout>
#include <QProgressBar>
#include <QTimer>

#include "timer.h"

Timer::Timer(QWidget *parent)
    : QWidget(parent)
{
    QHBoxLayout *layout = new QHBoxLayout();

    progressBar = new QProgressBar();
    progressBar->setMinimum(0);
    progressBar->setMaximum(100);

    layout->addWidget(progressBar);
    setLayout(layout);

    QTimer *timer = new QTimer(this);
    connect(timer, &QTimer::timeout, this, &Timer::updateProgress);
    timer->start(1000);

    setWindowTitle(tr("Timer"));
    resize(200, 200);
}

void Timer::updateProgress()
{
    progressBar->setValue(progressBar->value()+1);
}

```

timer.pro

```

QT += widgets

HEADERS = \
    timer.h
SOURCES = \

```

```
main.cpp \
timer.cpp
```

Singleshot Timer with Lambda function as slot

If a singleshot timer is required, it is quiet handy to have the slot as lambda function right in the place where the timer is declared:

```
QTimer::singleShot(1000, []() { /*Code here*/ } );
```

Due to [this Bug \(QTBUG-26406\)](#), this is way is only possible since Qt5.4.

In earlier Qt5 versions it has to be done with more boiler plate code:

```
QTimer *timer = new QTimer(this);
timer->setSingleShot(true);

connect(timer, &QTimer::timeout, [=]() {
    /*Code here*/
    timer->deleteLater();
} );
```

Using QTimer to run code on main thread

```
void DispatchToMainThread(std::function<void()> callback)
{
    // any thread
    QTimer* timer = new QTimer();
    timer->moveToThread(qApp->thread());
    timer->setSingleShot(true);
    QObject::connect(timer, &QTimer::timeout, [=]()
    {
        // main thread
        callback();
        timer->deleteLater();
    });
    QMetaObject::invokeMethod(timer, "start", Qt::QueuedConnection, Q_ARG(int, 0));
}
```

This is useful when you need to update a UI element from a thread. Keep in mind lifetime of anything the callback references.

```
DispatchToMainThread([]
{
    // main thread
    // do UI work here
});
```

Same code could be adapted to run code on any thread that runs Qt event loop, thus implementing a simple dispatch mechanism.

Basic Usage

`QTimer` add the functionality to have a specific function/slot called after a certain interval (repeatedly or just once).

The `QTimer` thus allows a GUI application to "check" things regularly or handle timeouts **without** having to manually start an extra thread for this and be careful about race conditions, because the timer will be handled in the main-event loop.

A timer can simply be used like this:

```
QTimer* timer = new QTimer(parent); //create timer with optional parent object
connect(timer,&QTimer::timeout,[this]() { checkProgress(); }); //some function to check
something
timer->start(1000); //start with a 1s interval
```

The timer triggers the `timeout` signal when the time is over and this will be called in the main-event loop.

`QTimer::singleShot` simple usage

The `QTimer::singleShot` is used to call a slot/lambda **asynchronously** after n ms.

The basic syntax is :

```
QTimer::singleShot(myTime, myObject, SLOT(myMethodInMyObject()));
```

with **myTime** the time in ms, **myObject** the object which contain the method and **myMethodInMyObject** the slot to call

So for example if you want to have a timer who write a debug line "hello !" every 5 seconds:

.cpp

```
void MyObject::startHelloWave()
{
    QTimer::singleShot(5 * 1000, this, SLOT(helloWave()));
}

void MyObject::helloWave()
{
    qDebug() << "hello !";
    QTimer::singleShot(5 * 1000, this, SLOT(helloWave()));
}
```

.hh

```
class MyObject : public QObject {
    Q_OBJECT
    ...
    void startHelloWave();

private slots:
    void helloWave();
}
```

```
...  
};
```

Read QTimer online: <https://riptutorial.com/qt/topic/4309/qtimer>

Chapter 21: Signals and Slots

Introduction

Signals and slots are used for communication between objects. The signals and slots mechanism is a central feature of Qt. In GUI programming, when we change one widget, we often want another widget to be notified. More generally, we want objects of any kind to be able to communicate with one another. Signals are emitted by objects when they change their state in a way that may be interesting to other objects. Slots can be used for receiving signals, but they are also normal member functions.

Remarks

Official documentation on this topic can be found [here](#).

Examples

A Small Example

Signals and slots are used for communication between objects. The signals and slots mechanism is a central feature of Qt and probably the part that differs most from the features provided by other frameworks.

The minimal example requires a class with one signal, one slot and one connection:

counter.h

```
#ifndef COUNTER_H
#define COUNTER_H

#include <QWidget>
#include <QDebug>

class Counter : public QWidget
{
    /*
     * All classes that contain signals or slots must mention Q_OBJECT
     * at the top of their declaration.
     * They must also derive (directly or indirectly) from QObject.
     */
    Q_OBJECT

public:
    Counter (QWidget *parent = 0): QWidget(parent)
    {
        m_value = 0;

        /*
         * The most important line: connect the signal to the slot.
         */
    }
};
```

```

        connect(this, &Counter::valueChanged, this, &Counter::printvalue);
    }

    void setValue(int value)
    {
        if (value != m_value) {
            m_value = value;
            /*
             * The emit line emits the signal valueChanged() from
             * the object, with the new value as argument.
             */
            emit valueChanged(m_value);
        }
    }

public slots:
    void printValue(int value)
    {
        qDebug() << "new value: " << value;
    }

signals:
    void valueChanged(int newValue);

private:
    int m_value;

};

#endif

```

The `main` sets a new value. We can check how the slot is called, printing the value.

```

#include <QtGui>
#include "counter.h"

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    Counter counter;
    counter.setValue(10);
    counter.show();

    return app.exec();
}

```

Finally, our project file:

```

SOURCES    = \
            main.cpp
HEADERS    = \
            counter.h

```

The new Qt5 connection syntax

The conventional `connect` syntax that uses `SIGNAL` and `SLOT` macros works entirely at runtime, which

has two drawbacks: it has some runtime overhead (resulting also in binary size overhead), and there's no compile-time correctness checking. The new syntax addresses both issues. Before checking the syntax in an example, we'd better know what happens in particular.

Let's say we are building a house and we want to connect the cables. This is exactly what connect function does. Signals and slots are the ones needing this connection. The point is if you do one connection, you need to be careful about the further overlapping connections. Whenever you connect a signal to a slot, you are trying to tell the compiler that whenever the signal was emitted, simply invoke the slot function. This is what exactly happens.

Here's a sample **main.cpp**:

```
#include <QApplication>
#include <QDebug>
#include <QTimer>

inline void onTick()
{
    qDebug() << "onTick() ";
}

struct OnTimerTickListener {
    void onTimerTick()
    {
        qDebug() << "OnTimerTickListener::onTimerTick() ";
    }
};

int main(int argc, char *argv[])
{
    QApplication app(argc, argv);

    OnTimerTickListener listenerObject;

    QTimer timer;
    // Connecting to a non-member function
    QObject::connect(&timer, &QTimer::timeout, onTick);
    // Connecting to an object member method
    QObject::connect(&timer, &QTimer::timeout, &listenerObject,
&OnTimerTickListener::onTimerTick);
    // Connecting to a lambda
    QObject::connect(&timer, &QTimer::timeout, []() {
        qDebug() << "lambda-onTick";
    });

    return app.exec();
}
```

Hint: the old syntax (`SIGNAL/SLOT` macros) requires that the Qt metacompiler (MOC) is run for any class that has either slots or signals. From the coding standpoint that means that such classes need to have the `Q_OBJECT` macro (which indicates the necessity to run MOC on this class).

The new syntax, on the other hand, still requires MOC for signals to work, but **not** for slots. If a class only has slots and no signals, it need not have the `Q_OBJECT` macro and hence may not invoke the MOC, which not only reduces the final binary size but also reduces compilation time (no MOC

call and no subsequent compiler call for the generated *_moc.cpp file).

Connecting overloaded signals/slots

While being better in many regards, the new connection syntax in Qt5 has one big weakness: Connecting overloaded signals and slots. In order to let the compiler resolve the overloads we need to use `static_cast`s to member function pointers, or (starting in Qt 5.7) `qOverload` and friends:

```
#include <QObject>

class MyObject : public QObject
{
    Q_OBJECT
public:
    explicit MyObject(QObject *parent = nullptr) : QObject(parent) {}

public slots:
    void slot(const QString &string) {}
    void slot(const int integer) {}

signals:
    void signal(const QString &string) {}
    void signal(const int integer) {}
};

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

    // using pointers to make connect calls just a little simpler
    MyObject *a = new MyObject;
    MyObject *b = new MyObject;

    // COMPILER ERROR! the compiler does not know which overloads to pick :(
    QObject::connect(a, &MyObject::signal, b, &MyObject::slot);

    // this works, now the compiler knows which overload to pick, it is very ugly and hard to
    remember though...
    QObject::connect(
        a,
        static_cast<void (MyObject::*) (int)>(&MyObject::signal),
        b,
        static_cast<void (MyObject::*) (int)>(&MyObject::slot));

    // ...so starting in Qt 5.7 we can use qOverload and friends:
    // this requires C++14 enabled:
    QObject::connect(
        a,
        qOverload<int>(&MyObject::signal),
        b,
        qOverload<int>(&MyObject::slot));

    // this is slightly longer, but works in C++11:
    QObject::connect(
        a,
        QOverload<int>::of(&MyObject::signal),
        b,
        QOverload<int>::of(&MyObject::slot));
```



```
// there are also qConstOverload/qNonConstOverload and QConstOverload/QNonConstOverload,
the names should be self-explanatory
}
```

Multi window signal slot connection

A simple multiwindow example using signals and slots.

There is a MainWindow class that controls the Main Window view. A second window controlled by Website class.

The two classes are connected so that when you click a button on the Website window something happens in the MainWindow (a text label is changed).

I made a simple example that is also on [GitHub](#):

mainwindow.h

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include "website.h"

namespace Ui {
class MainWindow;
}

class MainWindow : public QMainWindow
{
    Q_OBJECT

public:
    explicit MainWindow(QWidget *parent = 0);
    ~MainWindow();

public slots:
    void changeText();

private slots:
    void on_openButton_clicked();

private:
    Ui::MainWindow *ui;

    //You want to keep a pointer to a new Website window
    Website* webWindow;
};

#endif // MAINWINDOW_H
```

mainwindow.cpp

```
#include "mainwindow.h"
#include "ui_mainwindow.h"
```

```

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
}

MainWindow::~MainWindow()
{
    delete ui;
}

void MainWindow::changeText()
{
    ui->text->setText("New Text");
    delete webWindow;
}

void MainWindow::on_openButton_clicked()
{
    webWindow = new Website();
    QObject::connect(webWindow, SIGNAL(buttonPressed()), this, SLOT(changeText()));
    webWindow->show();
}

```

website.h

```

#ifndef WEBSITE_H
#define WEBSITE_H

#include <QDialog>

namespace Ui {
class Website;
}

class Website : public QDialog
{
    Q_OBJECT

public:
    explicit Website(QWidget *parent = 0);
    ~Website();

signals:
    void buttonPressed();

private slots:
    void on_changeButton_clicked();

private:
    Ui::Website *ui;
};

#endif // WEBSITE_H

```

website.cpp

```

#include "website.h"

```

```

#include "ui_website.h"

Website::Website(QWidget *parent) :
    QDialog(parent),
    ui(new Ui::Website)
{
    ui->setupUi(this);
}

Website::~Website()
{
    delete ui;
}

void Website::on_changeButton_clicked()
{
    emit buttonPressed();
}

```

Project composition:

```

SOURCES += main.cpp \
           mainwindow.cpp \
           website.cpp

HEADERS += mainwindow.h \
           website.h

FORMS    += mainwindow.ui \
           website.ui

```

Consider the Uis to be composed:

- Main Window: a label called "text" and a button called "openButton"
- Website Window: a button called "changeButton"

So the keypoints are the connections between signals and slots and the management of windows pointers or references.

Read Signals and Slots online: <https://riptutorial.com/qt/topic/2136/signals-and-slots>

Chapter 22: SQL on Qt

Examples

Basic connection and query

The `QSqlDatabase` class provides an interface for accessing a database through a connection. An instance of `QSqlDatabase` represents the connection. The connection provides access to the database via one of the supported database drivers. Make sure to Add

```
QT += SQL
```

in the .pro file. Assume an SQL DB named TestDB with a countryTable that contains the next column:

```
| country |  
-----  
| USA    |
```

In order to query and get sql data from TestDB:

```
#include <QtGui>  
#include <QtSql>  
  
int main(int argc, char *argv[])  
{  
    QCoreApplication app(argc, argv);  
  
    QSqlDatabase db = QSqlDatabase::addDatabase("QPSQL"); // Will use the driver referred to  
by "QPSQL" (PostgreSQL Driver)  
    db.setHostName("TestHost");  
    db.setDatabaseName("TestDB");  
    db.setUserName("Foo");  
    db.setPassword("FooPass");  
  
    bool ok = db.open();  
    if(ok)  
    {  
        QSqlQuery query("SELECT country FROM countryTable");  
        while (query.next())  
        {  
            QString country = query.value(0).toString();  
            qWarning() << country; // Prints "USA"  
        }  
    }  
  
    return app.exec();  
}
```

Qt SQL query parameters

It's often convenient to separate the SQL query from the actual values. This can be done using placeholders. Qt supports two placeholder syntaxes: named binding and positional binding.

named binding:

```
QSqlQuery query;
query.prepare("INSERT INTO employee (id, name, salary) VALUES (:id, :name, :salary)");
query.bindValue(":id", 1001);
query.bindValue(":name", "Thad Beaumont");
query.bindValue(":salary", 65000);
query.exec();
```

positional binding:

```
QSqlQuery query;
query.prepare("INSERT INTO employee (id, name, salary) VALUES (?, ?, ?)");
query.addBindValue(1001);
query.addBindValue("Thad Beaumont");
query.addBindValue(65000);
query.exec();
```

Note that before calling `bindValue()` or `addBindValue()` you need to call `QSqlQuery::prepare()` once.

MS SQL Server Database Connection using QODBC

When trying to open a Database Connection with QODBC please ensure

- You have QODBC driver available
- Your server has an ODBC interface and is enabled to (this depends on your ODBC driver installations)
- use shared memory access, TCP/IP connections or named pipe connection.

All connections only require the DatabaseName to be set by calling `QSqlDatabase::setDatabaseName`.

Open Connection using shared memory access

For this option to work you will need to have access to memory of the machine and must have permissions to access shared memory. For using a shared memory connection it is required to set `lpc:` in front of the Server string. Connection using the SQL Server Native Client 11 is made using these steps:

```
QString connectString = "Driver={SQL Server Native Client 11.0}"; //
Driver is now {SQL Server Native Client 11.0}
connectString.append("Server=lpc:"+QHostInfo::localHostName()+"\\SQLINSTANCENAME;"); //
Hostname,SQL-Server Instance
connectString.append("Database=SQLDBSCHEMA;"); // Schema
connectString.append("Uid=SQLUSER;"); // User
connectString.append("Pwd=SQLPASS;"); // Pass
db.setDatabaseName(connectString);
```

```

if(db.open())
{
    ui->statusBar->showMessage("Connected");
}
else
{
    ui->statusBar->showMessage("Not Connected");
}

```

Open Connection using Named Pipe

This option requires your ODBC Connection to have a full DSN. The Server string is setup by using the Windows Computername and the Instancename of the SQL Server. The example connection will be opened using SQL Server Native Client 10.0

```

QString connectString = "Driver={SQL Server Native Client 10.0};"; // Driver can also be {SQL
Server Native Client 11.0}
connectString.append("Server=SERVERHOSTNAME\\SQLINSTANCENAME;"); // Hostname,SQL-Server
Instance
connectString.append("Database=SQLDBSCHEMA;"); // Schema
connectString.append("Uid=SQLUSER;"); // User
connectString.append("Pwd=SQLPASS;"); // Pass
db.setDatabaseName(connectString);

if(db.open())
{
    ui->statusBar->showMessage("Connected");
}
else
{
    ui->statusBar->showMessage("Not Connected");
}

```

Open Connection using TCP/IP

For opening a TCP/IP connection the server should be configured to allow connections on a fixed port, otherwise you will first have to query for the currently active port. In this example we have a fixed port at 5171. You can find an example for setting up the server to allow connections on a fixed port at [1](#). For open a connection using TCP/IP use a tuple of the servers IP and Port:

```

QString connectString = "Driver={SQL Server};"; // Driver is now {SQL Server}
connectString.append("Server=10.1.1.15,5171;"); // IP,Port
connectString.append("Database=SQLDBSCHEMA;"); // Schema
connectString.append("Uid=SQLUSER;"); // User
connectString.append("Pwd=SQLPASS;"); // Pass
db.setDatabaseName(connectString);

if(db.open())
{
    ui->statusBar->showMessage("Connected");
}
else
{
    ui->statusBar->showMessage("Not Connected");
}

```

Read SQL on Qt online: <https://riptutorial.com/qt/topic/10628/sql-on-qt>

Chapter 23: Threading and Concurrency

Remarks

A few notes that are already mentioned in the official docs [here](#) and [here](#):

- If an object has a parent, it has to be in the same thread as the parent, i.e. it cannot be moved to a new thread, nor can you set a parent to an object if the parent and the object live in different threads
- When an object is moved to a new thread, all of its children are also moved to the new thread
- You can only *push* objects to a new thread. You cannot *pull* them to a new thread, i.e. you can only call `moveToThread` from the thread where the object is currently living in

Examples

Basic usage of QThread

`QThread` is a handle to a platform thread. It lets you manage the thread by monitoring its lifetime, and requesting that it finishes its work.

In most cases inhering from the class is not recommended. The default `run` method starts an event loop that can dispatch events to objects living in the class. Cross-thread signal-slot connections are implemented by dispatching a `QMetaCallEvent` to the target object.

A `QObject` instance can be moved to a thread, where it will process its events, such as timer events or slot/method calls.

To do work on a thread, first create your own worker class that derives from `QObject`. Then move it to the thread. The object can run its own code automatically e.g. by using

`QMetaObject::invokeMethod()`.

```
#include <QObject>

class MyWorker : public QObject
{
    Q_OBJECT
public:
    Q_SLOT void doWork() {
        qDebug() << "doWork()" << QThread::currentThread();
        // and do some long operation here
    }
    MyWorker(QObject * parent = nullptr) : QObject{parent} {}
};

class MyController : public QObject
{
    Q_OBJECT
    Worker worker;
```



```

    QThread workerThread;
public:
    MyController() {
        worker.moveToThread(&workerThread);
        // provide meaningful debug output
        workerThread.setObjectName("workerThread");
        workerThread.start();
        // the thread starts the event loop and blocks waiting for events
    }
    ~MyController() {
        workerThread.quit();
        workerThread.wait();
    }
    void operate() {
        // Qt::QueuedConnection ensures that the slot is invoked in its own thread
        QMetaObject::invokeMethod(&worker, "doWork", Qt::QueuedConnection);
    }
};

```

If your worker should be ephemeral and only exist while its work is being done, it's best to submit a functor or a thread-safe method for execution in the thread pool via `QtConcurrent::run`.

QtConcurrent Run

If you find managing QThreads and low-level primitives like mutexes or semaphores too complex, Qt Concurrent namespace is what you are looking for. It includes classes which allow more high-level thread management.

Let's look at Concurrent Run. `QtConcurrent::run()` allows to run function in a new thread. When would you like to use it? When you have some long operation and you don't want to create thread manually.

Now the code:

```

#include <qtconcurrentrun.h>

void longOperationFunction(string parameter)
{
    // we are already in another thread
    // long stuff here
}

void mainThreadFunction()
{
    QFuture<void> f = run(longOperationFunction, "argToPass");
    f.waitForFinished();
}

```

So things are simple: when we need to run another function in another thread, just call `QtConcurrent::run`, pass function and its parameters and that's it!

`QFuture` presents the result of our asynchronous computation. In case of `QtConcurrent::run` we can't cancel the function execution.

Invoking slots from other threads

When a Qt event loop is used to perform operations and a non-Qt-savvy user needs to interact with that event loop, writing the slot to handle regular invocations from another thread can simplify things for other users.

main.cpp:

```
#include "OperationExecutioner.h"
#include <QCoreApplication>
#include <QThread>

int main(int argc, char** argv)
{
    QCoreApplication app(argc, argv);

    QThread thrd;
    thrd.setObjectName("thrd");
    thrd.start();
    while(!thrd.isRunning())
        QThread::msleep(10);

    OperationExecutioner* oe = new OperationExecutioner;
    oe->moveToThread(&thrd);
    oe->doIt1(123, 'A');
    oe->deleteLater();
    thrd.quit();
    while(!thrd.isFinished())
        QThread::msleep(10);

    return 0;
}
```

OperationExecutioner.h:

```
#ifndef OPERATION_EXECUTIONER_H
#define OPERATION_EXECUTIONER_H

#include <QObject>

class OperationExecutioner : public QObject
{
    Q_OBJECT
public slots:
    void doIt1(int argi, char argc);
};

#endif // OPERATION_EXECUTIONER_H
```

OperationExecutioner.cpp:

```
#include "OperationExecutioner.h"
#include <QMetaObject>
#include <QThread>
#include <QDebug>
```

```

void OperationExecutioner::doIt1(int argi, char argc)
{
    if (QThread::currentThread() != thread()) {
        qInfo() << "Called from thread" << QThread::currentThread();
        QMetaObject::invokeMethod(this, "doIt1", Qt::QueuedConnection,
                                   Q_ARG(int, argi), Q_ARG(char, argc));

        return;
    }

    qInfo() << "Called from thread" << QThread::currentThread()
            << "with args" << argi << argc;
}

```

OperationExecutioner.pro:

```

HEADERS += OperationExecutioner.h
SOURCES += main.cpp OperationExecutioner.cpp
QT -= gui

```

Read Threading and Concurrency online: <https://riptutorial.com/qt/topic/5022/threading-and-concurrency>

Chapter 24: Using Style Sheets Effectively

Examples

Setting a UI widget's stylesheet

You can set the desired UI widget's stylesheet using any valid CSS. The example below will set a QLabel's text color a border around it.

```
#include "mainwindow.h"
#include "ui_mainwindow.h"

MainWindow::MainWindow(QWidget *parent) :
    QMainWindow(parent),
    ui(new Ui::MainWindow)
{
    ui->setupUi(this);
    QString style = "color: blue; border: solid black 5px;";
    ui->myLabel->setStyleSheet(style); //This can use colors RGB, HSL, HEX, etc.
}

MainWindow::~MainWindow()
{
    delete ui;
}
```

Read Using Style Sheets Effectively online: <https://riptutorial.com/qt/topic/5931/using-style-sheets-effectively>

Credits

S. No	Chapters	Contributors
1	Getting started with Qt	agilob , Christopher Aldama , Community , demonplus , devbean , Dmitriy , Donald Duck , fat , Gabriel de Grimouard , Kamalpreet Grewal , Maxito , Tarod , thiagofalcao
2	About using layouts, widget parenting	Gabriel de Grimouard
3	Build QtWebEngine from source	Martin Zhai
4	CMakeLists.txt for your Qt project	Athena , demonplus , Robert , Velkan , wasthishelpful
5	Common Pitfalls	e.jahandar
6	Communication between QML and C++	Gabriel de Grimouard , Martin Zhai
7	Deploying Qt applications	Luca Angioloni , Martin Zhai , Nathan Osman , TriskaIJM , wasthishelpful
8	Header on QListView	Papipone
9	Implicit sharing	Hayt
10	Model/View	Jan , KernelPanic , Tim D
11	Multimedia	demonplus , Gabriel de Grimouard
12	QDialogs	Wilmort
13	QGraphics	Chris , demonplus
14	qmake	Caleb Huitt - cjhuitt , demonplus , doc , Gregor , Jon Harper
15	QObject	demonplus , Hayt
16	Qt - Dealing with Databases	Jan , Rinat , Shihe Zhang , Zylva
17	Qt Container Classes	demonplus , Tarod

18	Qt Network	Gabriel de Grimouard
19	Qt Resource System	Victor Tran
20	QTimer	avb , Caleb Huitt - cjhuitt , Eugene , Gabriel de Grimouard , Hayt , Rinat , Tarod , thuga , tpr , Victor Tran
21	Signals and Slots	Athena , devbean , fat , immerhart , Jan , Luca Angioloni , Robert , Tarod , Violet Giraffe
22	SQL on Qt	Noam M
23	Threading and Concurrency	demonplus , gmabey , Kuba Ober , Nathan Osman , RamenChef , thuga
24	Using Style Sheets Effectively	Nicholas Johnson