

# Qt for Beginners

From Qt Wiki

Jump to: navigation, search

En Ar Bg De El Es Fa Fi Fr Hi Hu It Ja Kn Ko Ms Nl Pl Pt Ru Sq Th Tr Uk Zh

**Warning** : Be sure to have some knowledge of C++ before starting!

**Remark** : This tutorial series target mainly Qt4. Even if most of these tutorials are also valid for Qt5, the case of Qt5 is discussed in a separate part.

## C++ reminder

The **signature** of a method or function is simply its prototype. It completely describes a method or function. It contains the returned type, the name of the method/function (including the class name) and the parameters, including types.

```
Type MyObject::myFunction(  
    Type1 param1,  
    Type2 *param2,  
    const Type3 &param3  
);
```

New to Qt? Don't know how to start? Then this wiki page is for you! It is a step by step tutorial that presents all specificities and features of Qt. Want to learn more? Check out our C++ GUI Classes for Qt 5 (<https://doc.qt.io/qt-5/qtgui-module.html>) and C++ GUI Classes for Qt 6 (<https://doc.qt.io/qt-6/qtgui-module.html#details>).

## Introduction to Qt

Qt (pronounced as "cute", not "cu-tee") is a cross-platform framework that is usually used as a graphical toolkit, although it is also very helpful in creating CLI applications. It runs on the three major desktop OSes, as well as on mobile OSes, such as Symbian, Nokia Belle, Meego Harmattan, MeeGo or BB10, and on embedded devices. Ports for Android (Necessitas) and iOS are also in development.

## Contents

- 1 C++ reminder
- 2 Introduction to Qt
- 3 Installing Qt SDK
- 4 Qt Creator features
- 5 Our first window
- 6 How a Qt program is compiled
- 7 A pretty button
- 8 Qt class hierarchy
- 9 Parenting system
- 10 Subclassing QWidget
- 11 Further Reading
- 12 The observer pattern
- 13 Signals and slots
- 14 Transmitting information
- 15 Features of signals and slots
- 16 Examples
  - 16.1 Responding to an event
- 17 Transmitting information with signals and slots
- 18 Technical aspect
- 19 The Meta Object
- 20 Important macros
- 21 Creating custom signals and slots
  - 21.1 Creating custom slots
  - 21.2 Creating signals
  - 21.3 Example
- 22 Troubleshooting
- 23 Widgets
- 24 Signals and slots
- 25 Qt for beginners — Finding information in the documentation
- 26 Where to find the documentation
- 27 Important sections of the documentation
- 28 Browse the documentation of a class

Qt has an impressive collection of modules, including

- **QtCore**, a base library that provides containers, thread management, event management, and much more
- **QtGui** and **QtWidgets**, a GUI toolkit for Desktop, that provides a lot of graphical components to design applications.
- **QtNetwork**, that provides a useful set of classes to deal with network communications
- **QtWebkit**, the webkit engine, that enable the use of web pages and web apps in a Qt application.
- **QtSQL**, a full featured SQL RDBM abstraction layer extensible with own drivers, support for ODBC, SQLITE, MySQL and PostgreSQL is available out of the box
- **QtXML**, support for simple XML parsing (SAX) and DOM
- **QtXmlPatterns**, support for XSLT, XPath, XQuery and Schema validation

## Installing Qt SDK

To start writing Qt applications, you have to get Qt libraries, and, if you want, an IDE. They can be built from source, or better, be downloaded as an SDK from the download page (<http://www.qt.io/download/>).

This SDK includes a lot of features, like cross compilers for Symbian and the Nokia N9. You might choose not to install them by selecting "custom install". Be sure to keep these packages

- **QMake Documentation**
- **Qt Documentation**
- **Qt 4.8.1 (Desktop)**, assuming that Qt 4.8.1 is the latest version.
- **Qt Creator**

These packages can also be useful

- Qt Examples
- Qt Linguist

You can select other packages if you want to develop for Symbian / Maemo / Meego, or with older version of Qt.

**NB :** On linux, it is better to use the packages that your distribution provides. Qt Creator should be available in nearly all distributions, and installing it should install all dependencies, like libraries, compilers, and development headers.

***Note:** See the official *Getting Started with Qt Widgets* (<http://doc.qt.io/qt-5/gettingstartedqt.html>) page for an alternative tutorial.*

We are now ready to create our first window. And it will be as usual, a *hello world*.

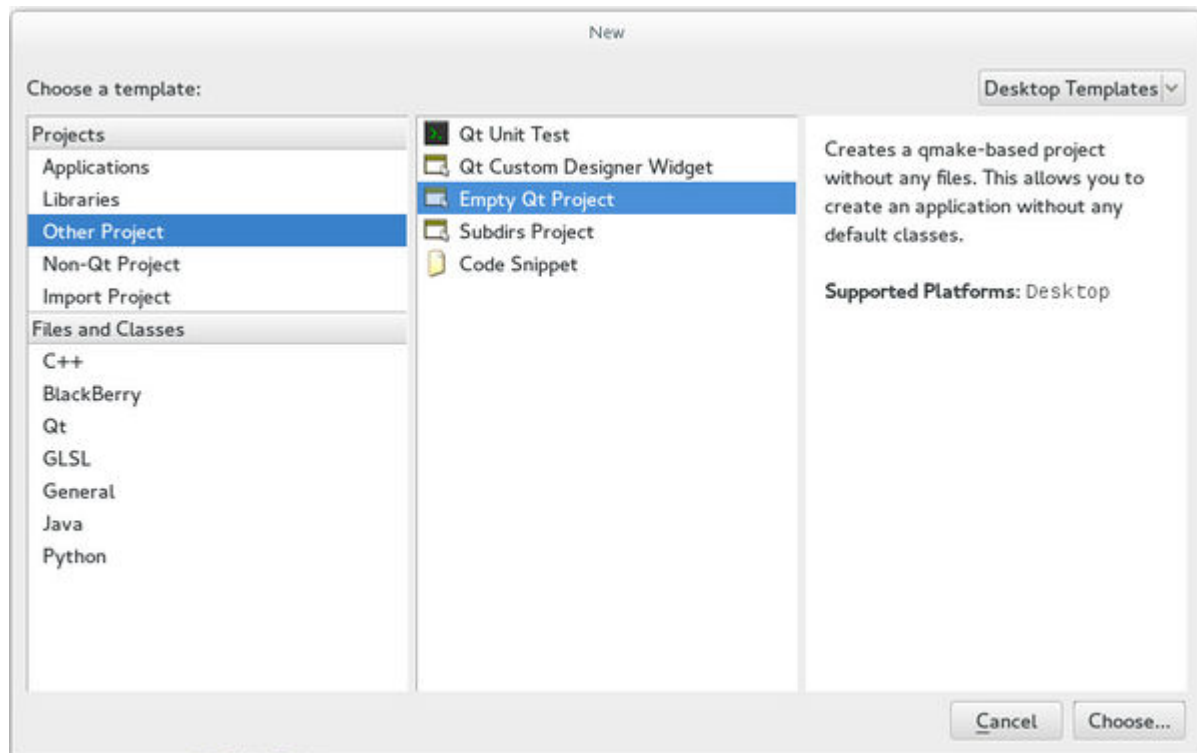
## Qt Creator features

Before writing our first GUI app, let's discover Qt Creator.

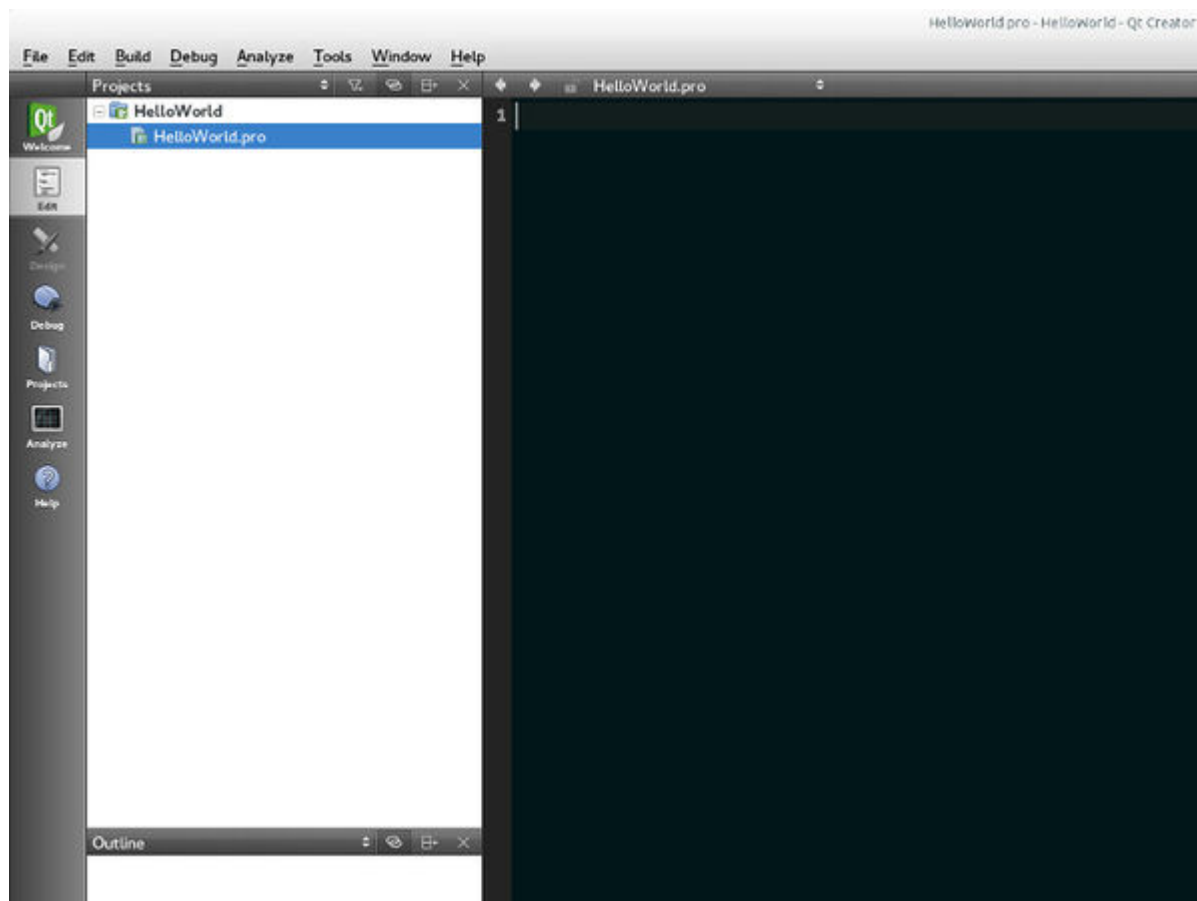
Qt Creator is yet another IDE for C++, but it is very well suited for coding Qt applications. It provides a doc browser and the "designer", which makes creation of windows easier, all wrapped in a well-designed user interface. It's also one of the fastest IDE's available.

## Our first window

Let's start by creating our first project. It will be an empty project, so we have to proceed with: File > New file or project > Other Projects > Empty Qt Project



Follow the wizard, and after selecting the project folder and name, and select the version of Qt to use, you should land on this page



This is the project file (extension .pro). Qt uses a command line tool that parses these project files in order to generate "makefiles", files that are used by compilers to build an application. This tool is called **qmake**. But, we shouldn't bother too much about qmake, since Qt Creator will do the job for us.

In a project file, there is some minimal code that should always be written :

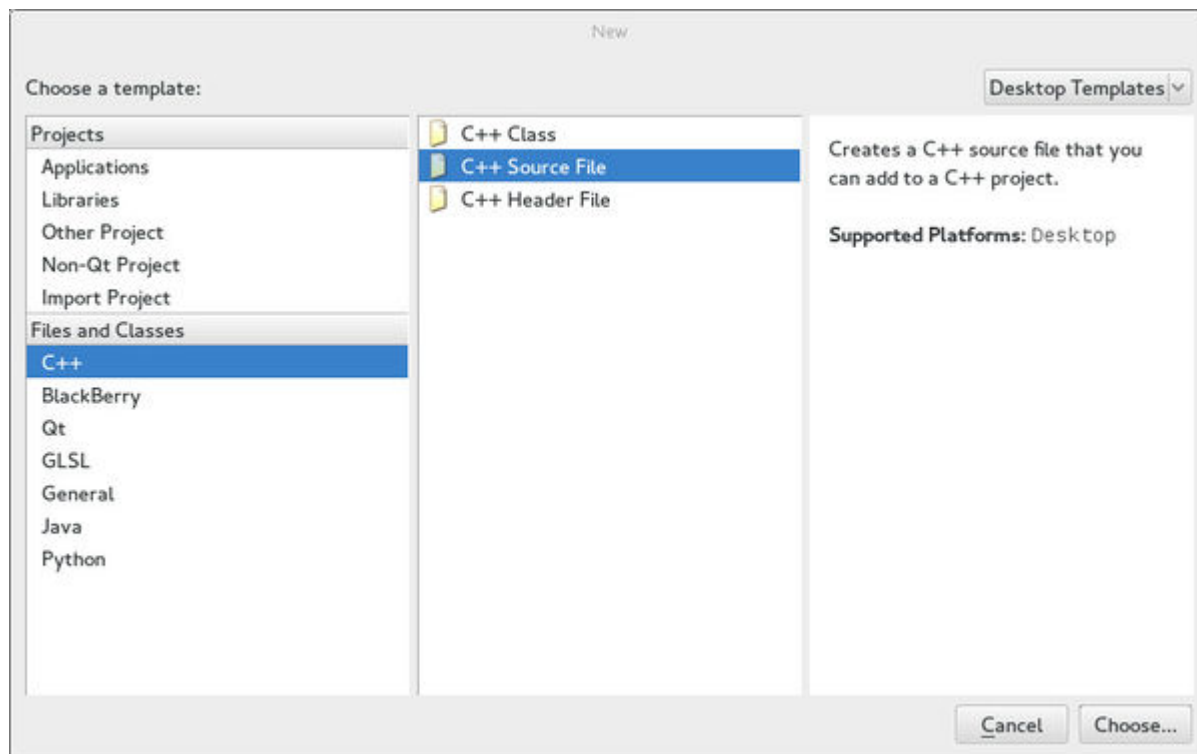
```
TEMPLATE = app
TARGET = name_of_the_app

QT = core gui

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

- *TEMPLATE* describes the type to build. It can be an application, a library, or simply subdirectories.
- *TARGET* is the name of the app or the library.
- *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.

Let's now add the entry point of our application. Using File > New file or project > C++ > C++ Source file should do the job.



Follow the wizard once again, naming the file "main", and you are done. You will notice that in the project file, a new line has been added automatically by Qt Creator :

```
TEMPLATE = app
TARGET = name_of_the_app

QT = core gui

greaterThan(QT_MAJOR_VERSION, 4): QT += widgets

SOURCES += main.cpp
```

Headers and sources files can be added manually with

```
HEADERS += first_file.h second_file.h
SOURCES += first_file.cpp second_file.cpp
```

or

```
HEADERS += first_file.h second_file.h
SOURCES += first_file.cpp second_file.cpp
```

If you use Qt Creator's wizards, this is done automatically.

The minimal source code of a Qt application is

```
#include <QApplication>

int main(int argc, char **argv)
{
    QApplication app (argc, argv);
    return app.exec();
}
```

**QApplication** is a very important class. It takes care of input arguments, but also a lot of other things, and most notably, the *event loop*. The event loop is a loop that waits for user input in GUI applications.

When calling `app.exec()` the event loop is launched.

Let's compile this application. By clicking on the green arrow on the bottom left, Qt Creator will compile and execute it. And what happened? The application seems to be launched and not responding. That is actually normal. The event loop is running and waiting for events, like mouse clicks on a GUI, but we did not provide any event to be processed, so it will run indefinitely.

Let's add something to be displayed.

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

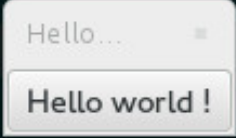
int main(int argc, char **argv)
{
    QApplication app (argc, argv);

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

    return app.exec();
}
```

Compile it, and ... here it is ! Our first window !

```
1  #include <QApplication>
2  #include <QPushButton>
3
4  int main(int argc, char **argv)
5  {
6      QApplication app (argc, argv);
7
8      QPushButton button ("Hello world !");
9      button.show();
10
11     return app.exec();
12 }
13
14
```



## How a Qt program is compiled

Qt Creator does the job of invoking the build system for us, but it might be interesting to know how Qt programs are compiled.

For small programs, it is easy to compile everything by hand, creating objects files, then linking them. But for bigger projects, the command line easily becomes hard to write. If you are familiar with Linux, you may know that all the programs are compiled using a makefile that describes all these command lines to execute. But for some projects, even writing a makefile can become tedious.

*qmake* is build system that comes with Qt, and it generates those makefiles for you (there are others that can be used, but we here give the example with *qmake*). With a simple syntax, it produces the makefile that is used to compile a Qt program. But that is not its only goal. Qt uses meta-objects to extend C++ functionalities, and *qmake* is responsible for preparing a makefile that contains this meta-object extraction phase. You will see this in another chapter.

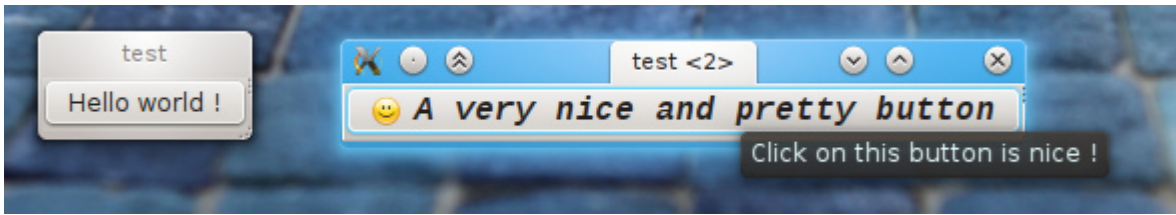
So, Qt apps are compiled in 3 steps

1. A *.pro* file is written to describe the project to compile
2. A makefile is generated using *qmake*
3. The program is built using *make* (or *nmake* or *jom* on windows)

## A pretty button

This chapter gives an overview of the widgets modules. It will cover widgets properties, the inheritance scheme that is used in widgets, and also the parenting system.

Now that we have our button, we may want to customize it a bit.



Qt objects have a lot of attributes that can be modified using getters and setters. In Qt, if an attribute is called *foo*, the associated getter and setter will have these signatures

```
T foo() const;  
void setFoo(const T);
```

In fact, Qt extends this system of attributes and getters and setters to something called *property*. A property is a value of any type that can be accessed, be modified or constant, and can notify a change. The property system is useful, especially in the third part (QML). For now, we will use "attribute" or "property" to do the same thing.

A QPushButton has plenty of properties :

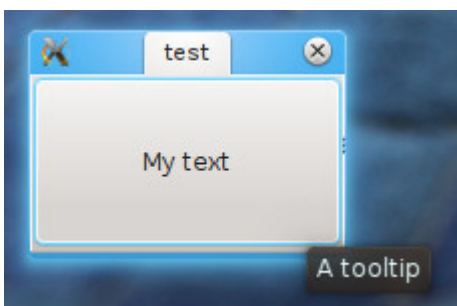
- text
- font
- tooltip
- icon
- ...

So we can use these to customize the button.

Let's first change the text and add a tooltip

```
#include <QApplication>  
#include <QPushButton>  
  
int main(int argc, char **argv)  
{  
    QApplication app (argc, argv);  
  
    QPushButton button;  
    button.setText("My text");  
    button.setTooltip("A tooltip");  
    button.show();  
  
    return app.exec();  
}
```

Here is the result:



We can also change the font. In Qt, a font is represented with the QFont (<http://doc.qt.io/qt-5/qfont.html#>) class. The documentation provides a lot of information. We are especially concerned here with one of the constructors of QFont.

```
QFont(const QString & family, int pointSize = -1, int weight = -1, bool italic = false)
```

In order to change the font, we have to instantiate a QFont class, and pass it to the QPushButton using setFont. The following snippet will change the font to Courier.

```
QFont font ("Courier");  
button.setFont(font);
```

You can try other parameters of QFont's constructor to reproduce the button that is represented in the first picture in this chapter.

Setting an icon is not very difficult either. An icon is represented with the QIcon (<http://doc.qt.io/qt-5/qicon.html#>) class. And you can create an icon provided that it has an absolute (or relative) path in the filesystem. I recommend providing the absolute path in this example. But for deployment considerations, you might use the relative path, or better, the resource system.

```
QIcon icon ("/path/to/my/icon/icon.png");  
button.setIcon(icon);
```

On Linux, and some other OS's, there is a convenient way to set an icon from an icon theme. It can be done by using the static method:

```
QIcon QIcon::fromTheme ( const QString &name, const QIcon &fallback = QIcon());
```

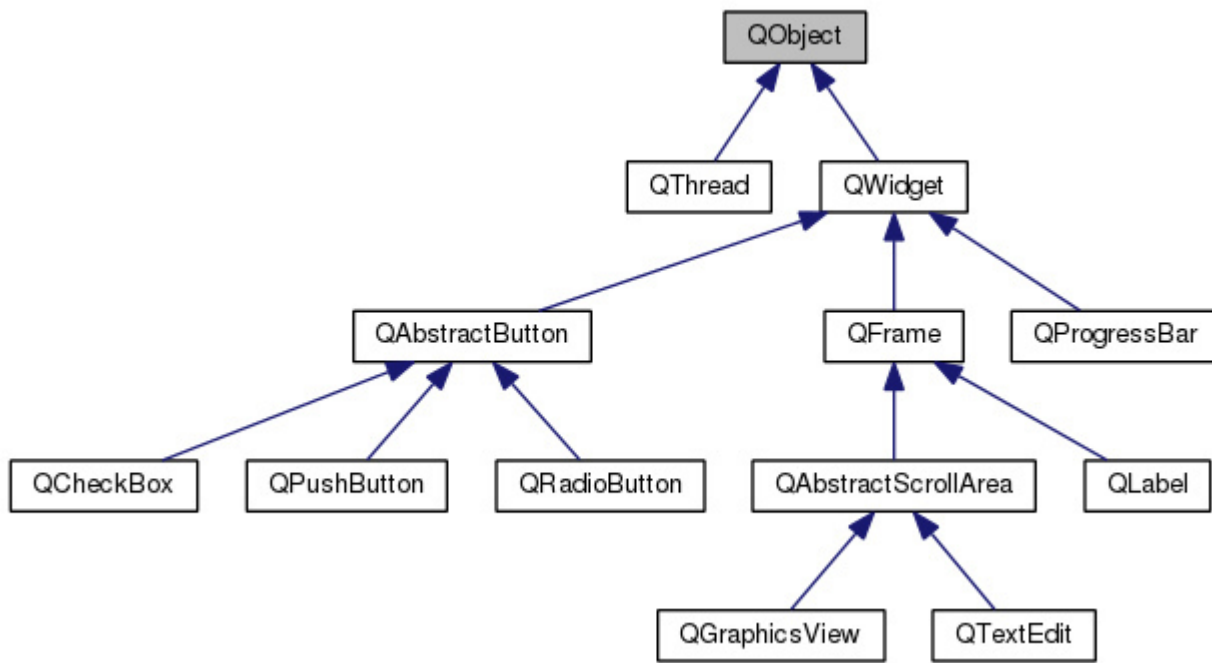
For example, in the screenshot at the beginning of this chapter, the smiley comes from the Oxygen KDE icon theme and was set by:

```
button.setIcon(QIcon::fromTheme("face-smile"));
```

## Qt class hierarchy

Qt widely uses inheritance, especially in the Widgets module. The following graph shows some of these inheritances:





QObject (<http://doc.qt.io/qt-5/qobject.html#>) is the most basic class in Qt. Most of classes in Qt inherit from this class. QObject provides some very powerful capabilities like:

- **object name** : you can set a name, as a string, to an object and search for objects by names.
- **parenting system** (described in the following section)
- **signals and slots** (described in the next chapter)
- **event management**

Widgets are able to respond to events and use parenting system and signals and slots mechanism. All widgets inherit from QObject. The most basic widget is the QWidget (<http://doc.qt.io/qt-5/qwidget.html#>). QWidget contains most properties that are used to describe a window, or a widget, like position and size, mouse cursor, tooltips, etc.

**Remark** : in Qt, a widget can also be a window. In the previous section, we displayed a button that is a widget, but it appears directly as a window. There is no need for a "QWindow" class.

Nearly all graphical elements inherit from QWidget. We can list for example:

QAbstractButton, a base class for all button types  
 QPushButton  
 QCheckBox  
 QRadioButton  
 QFrame, that displays a frame  
 QLabel, that displays text or picture

This inheritance is done in order to facilitate properties management. Shared properties like size and cursors can be used on other graphical components, and QAbstractButton (<http://doc.qt.io/qt-5/qabstractbutton.html#>) provides basic properties that are shared by all buttons.

## Parenting system

Parenting system is a convenient way of dealing with objects in Qt, especially widgets. Any object that inherits from QObject (<http://doc.qt.io/qt-5/qobject.html#>) can have a parent and children. This hierarchy tree makes many things convenient:

- When an object is destroyed, all of its children are destroyed as well. So, calling *delete* becomes optional in certain cases.
- All QObjects have *findChild* and *findChildren* methods that can be used to search for children of a given object.
- Child widgets in a QWidget (<http://doc.qt.io/qt-5/qwidget.html#>) automatically appear inside the parent widget.

The following snippet that creates a QPushButton (<http://doc.qt.io/qt-5/qpushbutton.html#>) inside a QPushButton:

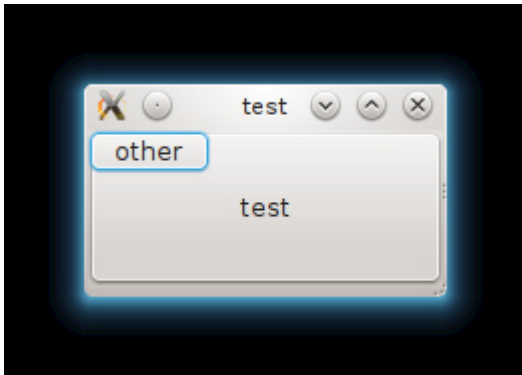
```
#include <QApplication>
#include <QPushButton>

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

    QPushButton button1 ("test");
    QPushButton button2 ("other", &button1);

    button1.show();

    return app.exec();
}
```



You can also note that when the application is closed, button1, which is allocated on the stack, is deallocated. Since button2 has button1 as a parent, it is deleted also. You can even test this in Qt Creator in the analyze section, by searching for a memory leak — there won't be any.

There is clearly no benefit in putting a button inside a button, but based on this idea, we might want to put buttons inside a container, that does not display anything. This container is simply the QWidget (<http://doc.qt.io/qt-5/qwidget.html#>).

The following code is used to display a button inside a widget:

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

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

    QWidget window;
    window.setFixedSize(100, 50);

    QPushButton *button = new QPushButton("Hello World", &window);
    button->setGeometry(10, 10, 80, 30);

    window.show();
}
```

```
return app.exec();  
}
```

Note that we create a fixed size widget (that acts as a window) using *setFixedSize*. This method has the following signature:

```
void QWidget::setFixedSize(int width, int height);
```

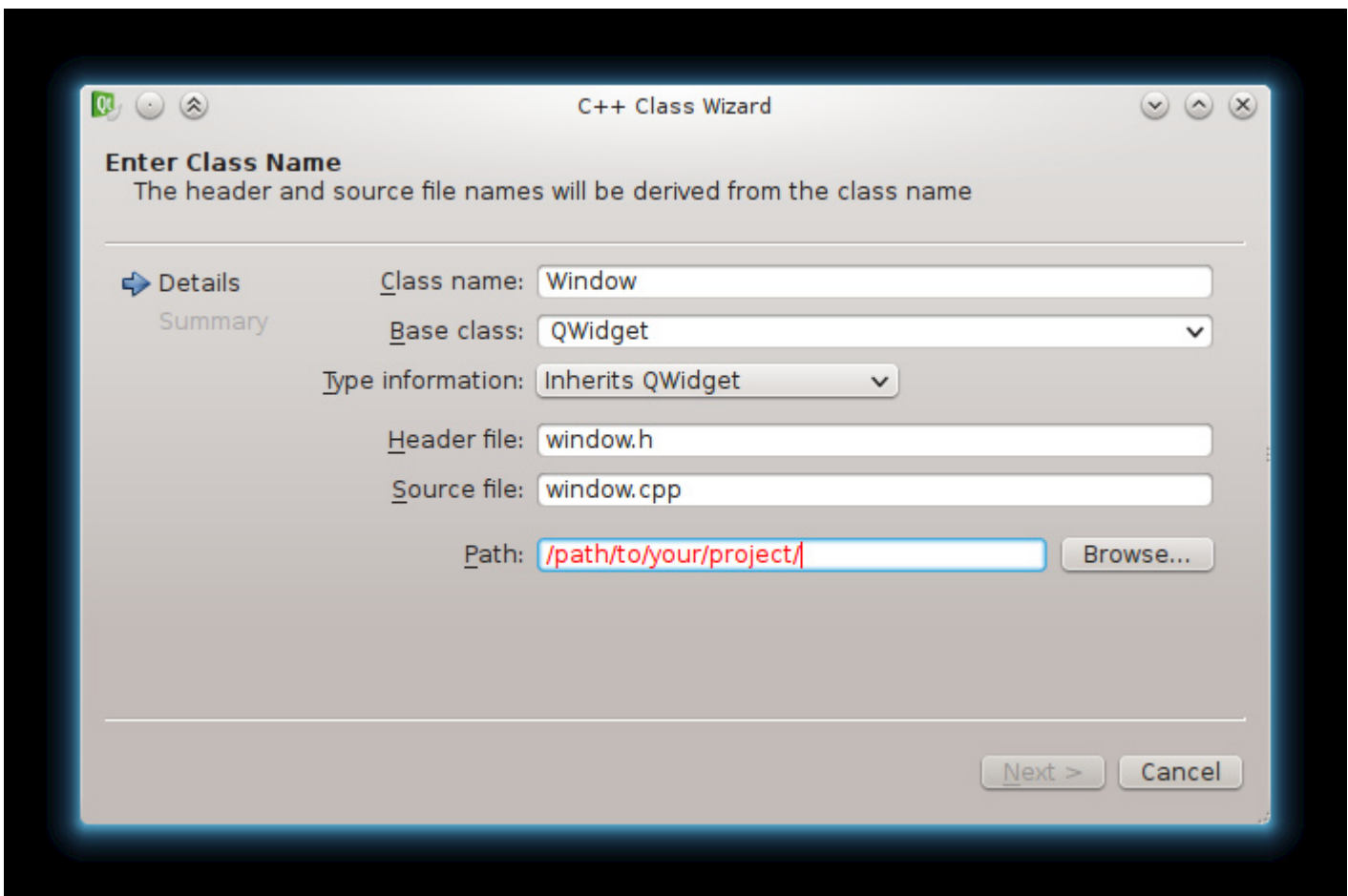
We also positioned the button using *setGeometry*. This method has the following signature:

```
void QWidget::setGeometry(int x, int y, int width, int height);
```

## Subclassing QWidget

Until now, we have put all of our code in the *main* function. This was not a problem for our simple examples, but for more and more complex applications we might want to split our code into different classes. What is often done is to create a class that is used to display a window, and implement all the widgets that are contained in this window as attributes of this class.

Inside Qt Creator, you can automatically create a new class with File > New file or project > C++ > C++ Class



Make the class inherit from *QWidget*, and you should obtain code similar to below

*Header*

```

#ifndef WINDOW_H
#define WINDOW_H

#include <QWidget>

class Window : public QWidget
{
    Q_OBJECT
public:
    explicit Window(QWidget *parent = 0);

    signals:
    public slots:
};

#endif // WINDOW_H

```

## Source

```

#include "window.h"

Window::Window(QWidget *parent) :
    QWidget(parent) {}

```

You can see that Qt Creator automatically generates a class template. Notice that there are some new elements in the header :

- The **Q\_OBJECT** macro.
- A new category of methods : **signals**
- A new category of methods : **public slots**

All these elements will be explained in the next chapter, and none of them are needed now. Implementing the window is done in the constructor. We can declare the size of the window, as well as the widgets that this window contains and their positions. For example, implementing the previous window that contains a button can be done in this way :

## *main.cpp*

```

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

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

    Window window;
    window.show();

    return app.exec();
}

```

## *window.h*

```

#ifndef WINDOW_H
#define WINDOW_H

#include <QWidget>

class QPushButton;
class Window : public QWidget
{

```

```

public:
    explicit Window(QWidget *parent = 0);
private:
    QPushButton *m_button;
};

#endif // WINDOW_H

```

*window.cpp*

```

#include "window.h"
#include <QPushButton>

Window::Window(QWidget *parent) :
    QWidget(parent)
{
    // Set size of the window
    setFixedSize(100, 50);

    // Create and position the button
    m_button = new QPushButton("Hello World", this);
    m_button->setGeometry(10, 10, 80, 30);
}

```

Please note that there is no need for writing a destructor for deleting `m_button`. With the parenting system, when the `Window` instance is out of the stack, the `m_button` is automatically deleted.

## Further Reading

A better overview of `QPushButton` (<http://doc.qt.io/qt-5/qpushbutton.html#>) is given in this wiki page [How to Use QPushButton](#)

## The observer pattern

Nearly all UI toolkits have a mechanism to detect a user action, and respond to this action. Some of them use *callbacks*, others use *listeners*, but basically, all of them are inspired by the observer pattern ([http://en.wikipedia.org/wiki/Observer\\_pattern](http://en.wikipedia.org/wiki/Observer_pattern)).

Observer pattern is used when an *observable* object wants to notify other *observers* objects about a state change. Here are some concrete examples:

- A user has clicked on a button, and a menu should be displayed.
- A web page just finished loading, and a process should extract some information from this loaded page.
- An user is scrolling through a list of items (in an app store for example), and reached the end, so other items should be loaded.

Observer pattern is used everywhere in GUI applications, and often leads to some boilerplate code ([http://en.wikipedia.org/wiki/Boilerplate\\_code](http://en.wikipedia.org/wiki/Boilerplate_code)). Qt was created with the idea of removing this boilerplate code and providing a nice and clean syntax, and the signal and slots mechanism is the answer.

## Signals and slots

Instead of having observable objects and observers, and registering them, Qt provides two high level concepts: **signals** and **slots**.

- A **signal** is a message that an object can send, most of the time to inform of a status change.
- A **slot** is a function that is used to accept and respond to a signal.

Here are some examples of signals and slots from our well known QPushButton (<http://doc.qt.io/qt-5/qpushbutton.html#>) class.

- clicked
- pressed
- released

As you can see, their names are quite explicit. These signals are sent when the user clicked (pressed then released), pressed or released the button.

Here are some slots, from different classes

- QApplication::quit
- QWidget::setEnabled
- QPushButton::setText

In order to respond to a signal, a slot must be *connected* to a signal. Qt provides the method QObject::connect. It is used this way, with the two macros **SIGNAL** and **SLOT**

```
FooObjectA *fooA = new FooObjectA();
FooObjectB *fooB = new FooObjectB();

QObject::connect(fooA, SIGNAL (bared()), fooB, SLOT (baz()));
```

assuming that FooObjectA have a bared signal, and FooObjectB have a baz slot.

You have to write the signature of the signal and the slot inside the two macros *SIGNAL* and *SLOT*. If you want to get some information about what these macros do, please read the last section of this chapter.

**Remark** : Basically, signals and slots are methods, that might or might not have arguments, but that never return anything. While the notion of a signal as a method is unusual, a slot is actually a real method, and can be called as usual in other methods, or whilst responding to a signal.

## Transmitting information

The signals and slots mechanism is useful to respond to buttons clicks, but it can do much more than that. For example, It can also be used to communicate information. Let's say while playing a song, a progress bar is needed to show how much time remains before the song is over. A media player might have a class that is used to check the progress of the media. An instance of this class might periodically send a *tick* signal, with the progress value. This signal can be connected to a QProgressBar (<http://doc.qt.io/qt-5/qprogressbar.html#>), that can be used to display the progress.

The hypothetical class used to check the progress might have a signal that have this signature :

```
void MediaProgressManager::tick(int milliseconds);
```

and we know from the documentation, that the QProgressBar has this slot:

```
void QProgressBar::setValue(int value);
```

You can see that the signal and the slot have the same kind of parameters, especially the type. If you connect a signal to a slot that does not share the same kind of parameters, when the connection is done (at run-time) you will get a warning like:

```
QObject::connect: Incompatible sender/receiver arguments
```

This is because the signal transmits the information to the slot using the parameters. The first parameter of the signal is passed to the first one of the slot, and the same for second, third, and so forth.

The code for the connection will look like this:

```
MediaProgressManager *manager = new MediaProgressManager();
QProgressBar *progress = new QProgressBar(window);

QObject::connect(manager, SIGNAL (tick(int)), progress, SLOT (setValue(int)));
```

You can see that you have to provide a signature inside the *SIGNAL* and *SLOT* macro, providing the type of the values that are passed through the signals. You may also provide the name of the variable if you want. (It is actually even better).

## Features of signals and slots

- A signal can be connected to several slots
- Many signals can be connected to a slot
- A signal can be connected to a signal: it is signal relaying. The second signal is sent if the first signal is sent.

## Examples

### Responding to an event

Remember our button app? Let's try to actually make something with this app, like being able to close it while clicking on the button. We already know that QPushButton (<http://doc.qt.io/qt-5/qpushbutton.html#>) provides the *clicked* signal. We also have to know that QApplication (<http://doc.qt.io/qt-5/qapplication.html#>) provides the *quit* slot, that closes the application.

In order to make a click on a button close the app, we have to connect the signal *clicked* of the button to the *quit* slot of QApplication instance. We can modify the code from the previous chapter to do this, but before that, you might wonder how to access to the QApplication instance while you are in another class. Actually, it is pretty simple, since there exists a static function in QApplication (<http://doc.qt.io/qt-5/qapplication.html#>), with the following signature, that is used to get it:

```
QApplication * QApplication::instance()
```

This leads to the following modification of our previous code:

*window.cpp*

```

#include "window.h"

#include <QApplication>
#include <QPushButton>

Window::Window(QWidget *parent) :
    QWidget(parent)
{
    // Set size of the window
    setFixedSize(100, 50);

    // Create and position the button
    m_button = new QPushButton("Hello World", this);
    m_button->setGeometry(10, 10, 80, 30);

    // NEW : Do the connection
    connect(m_button, SIGNAL (clicked()), QApplication::instance(), SLOT (quit()));
}

```

While clicking on the button inside of the window, the application should quit.

## Transmitting information with signals and slots

Here is a simpler example for information transmission. It only displays a progress bar and a slider (created by `QSlider` (<http://doc.qt.io/qt-5/qslider.html#>)) inside a window, and while the slider is moved, the value of the progress bar is synced with a very simple connection.

The interesting signals and slots are:

```

void QSlider::valueChanged(int value);
void QProgressBar::setValue(int value);

```

`QSlider` automatically emits the signal `valueChanged` with the new value passed as a parameter when the value is changed, and the method `setValue` of `QProgressBar`, is used, as we have seen, to set the value of the progress bar.

This leads to the following code:

```

#include <QApplication>
#include <QProgressBar>
#include <QSlider>

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

    // Create a container window
    QWidget window;
    window.setFixedSize(200, 80);

    // Create a progress bar
    // with the range between 0 and 100, and a starting value of 0
    QProgressBar *progressBar = new QProgressBar(&window);
    progressBar->setRange(0, 100);
    progressBar->setValue(0);
    progressBar->setGeometry(10, 10, 180, 30);

    // Create a horizontal slider
    // with the range between 0 and 100, and a starting value of 0
    QSlider *slider = new QSlider(&window);
    slider->setOrientation(Qt::Horizontal);
    slider->setRange(0, 100);
    slider->setValue(0);
    slider->setGeometry(10, 40, 180, 30);
}

```



```

window.show();

// Connection
// This connection set the value of the progress bar
// while the slider's value changes
QObject::connect(slider, SIGNAL (valueChanged(int)), progressBar, SLOT (setValue(int)));

return app.exec();
}

```

## Technical aspect

This section can be skipped for now if you only want to program with Qt. Just know that you need to put **SIGNAL** and **SLOT** around the signals and slots while calling connect. If you want to know how Qt works, it is better to read this.

## The Meta Object

Qt provides a *meta-object* system. Meta-object (literally "over the object") is a way to achieve some programming paradigms that are normally impossible to achieve with pure C++ like:

- **Introspection** : capability of examining a type at run-time
- **Asynchronous function calls**

To use such meta-object capabilities in an application, one can subclass QObject (<http://doc.qt.io/qt-5/qobject.html>) and mark it so that the meta-object compiler (moc) can interpret and translate it.

Code produced by moc includes signals and slots signatures, methods that are used to retrieve meta-information from those marked classes, properties handling... All this information can be accessed using the following method:

```

const QMetaObject * QObject::metaObject () const

```

QMetaObject (<http://doc.qt.io/qt-5/qmetaobject.html>) class contains all the methods that deal with meta-objects.

## Important macros

The most important macro is **Q\_OBJECT**. Signal-Slot connections and their syntax cannot be interpreted by a regular C++ compiler. The moc is provided to translate the QT syntax like "connect", "signals", "slots", etc into regular C++ syntax. This is done by specifying the **Q\_OBJECT** macro in the header containing class definitions that use such syntax.

*mywidget.h*

```

class MyWidget : public QWidget
{
    Q_OBJECT
public:
    MyWidget(QWidget *parent = 0);
}

```

Others marker macros for moc are

- **signals**
- **public** / **protected** / **private slots**

that mark the different methods that need to be extended.

**SIGNAL** and **SLOT** are also two very important and useful macros. When a signal is emitted, the meta-object system is used to compare the signature of the signal, to check the connection, and to find the slot using its signature. These macros are actually used to convert the provided method signature into a string that matches the one stored in the meta-object.

## Creating custom signals and slots

This chapter covers the second part of signals and slots: implementing custom signals and slots.

Creating custom slots and signals is really simple. Slots are like normal methods, but with small decorations around, while signals need little to no implementation at all.

Creating custom signals and slots is very simple. It is described by the following checklist:

- add **Q\_OBJECT** macro
- add **signals** section, and write signals prototypes
- add **public slots** or **protected slots** or **private slots** sections, and write slots prototypes
- implement slots as normal methods
- establish connections

### Creating custom slots

In order to implement a slot, we first need to make the class be able to send signals and have slots (see the previous chapter). This is done by setting the **Q\_OBJECT** macro in the class declaration (often in the header).

After that, a slot should be declared in the corresponding section, and implemented as a normal method.

Finally, slots are connected to signals.

### Creating signals

As for slots, we first need to add the **Q\_OBJECT** macro.

Signals should also be declared in the *signals* section, and there is no need for them to be implemented.

They are emitted using the **emit** keyword:

```
emit mySignal();
```

Note that in order to send signals that have parameters, you have to pass them in the signal emission:

```
emit mySignal(firstParameter, secondParameter ...);
```

### Example

## Creating custom slots

Let's start with our window with the button:

*window.h*

```
#ifndef WINDOW_H
#define WINDOW_H

#include <QWidget>

class QPushButton;
class Window : public QWidget
{
public:
    explicit Window(QWidget *parent = 0);
private:
    QPushButton *m_button;
};

#endif // WINDOW_H
```

*window.cpp*

```
#include "window.h"
#include <QPushButton>

Window::Window(QWidget *parent) :
    QWidget(parent)
{
    // Set size of the window
    setFixedSize(100, 50);

    // Create and position the button
    m_button = new QPushButton("Hello World", this);
    m_button->setGeometry(10, 10, 80, 30);
}
```

We might want to remove our previous connection that makes the application quit while clicking the button. Now, we want that, when clicking on the button, the text is changed. More precisely, we want that the button can be *checked*, and that, when checked, it displays "checked", and when unchecked, it restores "Hello World".

QPushButton does not implement such a specific slot, so we have to implement it on our own. As stated previously, we have to add the **Q\_OBJECT** macro.

```
class Window : public QWidget
{
    Q_OBJECT
public:
    explicit Window(QWidget *parent = 0);
private:
    QPushButton *m_button;
};
```

We also add our custom slot. Since we are trying to react from the button being checked, and since the corresponding signal is

```
void QPushButton::clicked(bool checked)
```

we might implement a slot that has this signature :

```
void Window::slotButtonClicked(bool checked);
```

Most of the time, by convention, we implement private and protected slots by prefixing them with "slot". Here, we are not interested in exposing this slot as a public function, we can make it private. The new header is then

*window.h*

```
#ifndef WINDOW_H
#define WINDOW_H

#include <QWidget>

class QPushButton;
class Window : public QWidget
{
    Q_OBJECT
public:
    explicit Window(QWidget *parent = 0);
private slots:
    void slotButtonClicked(bool checked);
private:
    QPushButton *m_button;
};

#endif // WINDOW_H
```

The implementation of this slot is

```
void Window::slotButtonClicked(bool checked)
{
    if (checked) {
        m_button->setText("Checked");
    } else {
        m_button->setText("Hello World");
    }
}
```

We need to make the button checkable, and establish the connection, we have to add this code in the constructor:

```
m_button->setCheckable(true);

connect(m_button, SIGNAL (clicked(bool)), this, SLOT (slotButtonClicked(bool)));
```

The resulting code is then:

*window.cpp*

```
#include "window.h"

#include <QPushButton>

Window::Window(QWidget *parent) :
    QWidget(parent)
{
    // Set size of the window
    setFixedSize(100, 50);

    // Create and position the button
```

```

m_button = new QPushButton("Hello World", this);
m_button->setGeometry(10, 10, 80, 30);
m_button->setCheckable(true);

connect(m_button, SIGNAL (clicked(bool)), this, SLOT (slotButtonClicked(bool)));
}

void Window::slotButtonClicked(bool checked)
{
    if (checked) {
        m_button->setText("Checked");
    } else {
        m_button->setText("Hello World");
    }
}
}

```

## Emitting custom signals

Based on the previous example, we want to close the application if the button is clicked (checked or unchecked) 10 times. We first need to implement a counter that will count the number of clicks. These modifications implement it:

```

class Window : public QWidget
{
    Q_OBJECT
public:
    explicit Window(QWidget *parent = 0);
private slots:
    void slotButtonClicked(bool checked);
private:
    int m_counter;
    QPushButton *m_button;
};

```

and

```

Window::Window(QWidget *parent) :
    QWidget(parent)
{
    // Set size of the window
    setFixedSize(100, 50);

    // Create and position the button
    m_button = new QPushButton("Hello World", this);
    m_button->setGeometry(10, 10, 80, 30);
    m_button->setCheckable(true);

    m_counter = 0;

    connect(m_button, SIGNAL (clicked(bool)), this, SLOT (slotButtonClicked(bool)));
}

void Window::slotButtonClicked(bool checked)
{
    if (checked) {
        m_button->setText("Checked");
    } else {
        m_button->setText("Hello World");
    }

    m_counter ++;
}

```

Now, we have to create a custom signal that is used to notify other components, that the counter has reached 10. In order to declare a signal, we have to add a

```
signals
```

section in the header. We might also declare a signal with the following signature:

```
void Window::counterReached()
```

The header class is then declared as followed:

```
class Window : public QWidget
{
    Q_OBJECT
public:
    explicit Window(QWidget *parent = 0);
signals:
    void counterReached();
private slots:
    void slotButtonClicked(bool checked);
private:
    int m_counter;
    QPushButton *m_button;
};
```

Even if the signal is declared as a method, there is no need to implement it. The meta-object compiler is used to do this.

Now we need to emit the signal when the counter reaches 10. It is simply done in the slot:

```
void Window::slotButtonClicked(bool checked)
{
    if (checked) {
        m_button->setText("Checked");
    } else {
        m_button->setText("Hello World");
    }

    m_counter++;
    if (m_counter == 10) {
        emit counterReached();
    }
}
```

We need to write the keyword **emit** to send the signal.

Connecting the newly created signal to the quit slot is done as usual:

```
connect(this, SIGNAL (counterReached()), QApplication::instance(), SLOT (quit()));
```

The final code is:

*window.h*

```
#ifndef WINDOW_H
#define WINDOW_H

#include <QWidget>

class QPushButton;
```

```

class Window : public QWidget
{
    Q_OBJECT
public:
    explicit Window(QWidget *parent = 0);
signals:
    void counterReached();
private slots:
    void slotButtonClicked(bool checked);
private:
    int m_counter;
    QPushButton *m_button;
};

#endif // WINDOW_H

```

*window.cpp*

```

#include "window.h"

#include <QPushButton>
#include <QApplication>

Window::Window(QWidget *parent) :
    QWidget(parent)
{
    // Set size of the window
    setFixedSize(100, 50);

    // Create and position the button
    m_button = new QPushButton("Hello World", this);
    m_button->setGeometry(10, 10, 80, 30);
    m_button->setCheckable(true);

    m_counter = 0;

    connect(m_button, SIGNAL (clicked(bool)), this, SLOT (slotButtonClicked(bool)));
    connect(this, SIGNAL (counterReached()), QApplication::instance(), SLOT (quit()));
}

void Window::slotButtonClicked(bool checked)
{
    if (checked) {
        m_button->setText("Checked");
    } else {
        m_button->setText("Hello World");
    }

    m_counter ++;
    if (m_counter == 10) {
        emit counterReached();
    }
}

```

And you can try and check that after clicking the button ten times, the application will quit.

## Troubleshooting

While compiling your program, especially when you are adding the macro `Q_OBJECT`, you might have this compilation error.

```

main.cpp:(.text._ZN6WindowD2Ev[_ZN6WindowD5Ev]+0x3): undefined reference to `vtable for Window'

```

This is because of the meta-object compiler not being run on a class that should have meta-object. You should **rerun qmake**, by doing `Build > Run qmake`.

## Widgets

Radio button is a standard GUI component. It is often used to make a unique choice from a list. In Qt, the `QRadioButton` (<http://doc.qt.io/qt-5/qradiobutton.html#>) is used to create radio buttons.

Thanks to a nice heritage, a `QRadioButton` behaves just like a `QPushButton`. All properties of the `QPushButton` are also the same in the `QRadioButton`, and everything that was learned in the second chapter can be reused here.

- text
- icon
- tooltip
- ...

By default, `QRadioButton`s are not grouped, so many of them can be checked at the same time. In order to have the "exclusive" behaviour of many radio buttons, we need to use `QButtonGroup` (<http://doc.qt.io/qt-5/qbuttongroup.html#>). This class can be used like this: We allocate a new button group and attach it to the parent object. Note that the parent object might be the mainwindow, or "this":

```
QButtonGroup *buttonGroup = new QButtonGroup(object);  
  
// Add buttons in the button group  
buttonGroup->addButton(button1);  
buttonGroup->addButton(button2);  
buttonGroup->addButton(button3);  
...
```

What we want is to create a menu picker. In a window, a list of yummy plates should be displayed with radio buttons, and a push button that is used to select the chosen plate should be displayed.

Obviously, nothing will happen (now) when the buttons are clicked.

## Signals and slots

Here is an example about signals and slots. We are going to write an application with two buttons. The first button should display information about Qt.

We provide you the following code to complete :

```
#include <QApplication>  
#include <QPushButton>  
  
int main(int argc, char **argv)  
{  
    QApplication app (argc, argv);  
  
    QWidget window;  
    window.setFixedSize(100, 80);  
  
    QPushButton *buttonInfo = new QPushButton("Info", &window);  
    buttonInfo->setGeometry(10, 10, 80, 30);  
  
    QPushButton *buttonQuit = new QPushButton("Quit", &window);  
    buttonQuit->setGeometry(10, 40, 80, 30);  
}
```



```
window.show();  
// Add your code here  
return app.exec();  
}
```

In order to display the information about Qt, you should use the following method

```
void QApplication::aboutQt();
```

You can also add icons on the buttons, or resize them. Obviously, the "Quit" button should be more important, so why not make it bigger?

But we really recommend you try and figure it out by yourself how to solve these exercises.

## Qt for beginners — Finding information in the documentation

Qt documentation is a very valuable piece of information. It is the place to find *everything* related to Qt. But, Qt documentation is not a tutorial on how to use Qt. It is a collection of all information related to classes, as well as some examples. The goal of this chapter is to introduce you to the documentation as a basis for programming with Qt.

### Where to find the documentation

The best source of documentation is on the internet, in this developer network :

Qt documentation on developer network (<http://doc.qt.io/>)

It provides the full doc, as well as some DocNotes, that users can add. These DocNotes give more examples and highlight some tricky points. The online documentation also has a quite powerful search engine and contains also all the documentation for all versions of Qt.

While the online version requires an internet connection, the DocNotes are still available. If the QtSDK was installed correctly, the documentation that matches the current version of Qt should have been installed, and the *Help* section of QtCreator should not be empty. You can also use *Qt Assistant*, that is a standalone doc browser.

### Important sections of the documentation

If you are running the offline documentation viewer, in either Qt Creator, or Qt Assistant, you will find in the summary that there are documentations for different components of the Qt SDK.

- Qt Assistant documentation
- Qt Designer documentation
- Qt Linguist documentation
- QMake documentation
- Qt reference documentation

The most important component is, of course, the Qt reference documentation.

Qt documentation provides a nice introduction of many components, and also the documentation for all the classes in Qt. This list is listed in the page All classes (<http://doc.qt.io/qt-4.8/classes.html>). Another interesting page is the page that lists All modules (<http://doc.qt.io/qt-4.8/modules.html>). This page provides information about the different components in Qt.

In this tutorial, we will mostly use these modules

- Qt Core (<http://doc.qt.io/qt-4.8/qtcore.html>)
- Qt GUI (<http://doc.qt.io/qt-4.8/qtgui.html>)

The search function is also quite important. If you know the class to use, and want to find the documentation, you can either type the name of this class in the search field (online), or in the filter in the index (offline). You can also search for methods and enumerations in these fields.

## Browse the documentation of a class

Classes documentation is always organized in the same fashion :

- Name and short description of the class
- Inheritance
- Enumerations
- Properties
- Public methods
- Public slots
- Signals
- Protected methods

Let's take the QTextEdit (<http://doc.qt.io/qt-5/qtextedit.html#>) class as an example.

Retrieved from "[https://wiki.qt.io/index.php?title=Qt\\_for\\_Beginners&oldid=37987](https://wiki.qt.io/index.php?title=Qt_for_Beginners&oldid=37987)"

- 
- This page was last edited on 13 April 2021, at 07:18.