

GUI Applications with C++
Build GUI apps for linux with QT

Nathan Warner



**Northern Illinois
University**

Computer Science
Northern Illinois University
November 23, 2023
United States

Contents

| | |
|--|----|
| Preface | 2 |
| Source File (Simple GUI) | 3 |
| .pro file (Making project) | 4 |
| Building the application | 4 |
| Other options for pro file variables | 5 |
| Includes | 7 |
| Example Application | 7 |
| Q_OBJECT | 8 |
| Signals and Slots | 8 |
| Signals | 8 |
| Slots | 8 |
| Connecting Signals to Slots | 9 |
| The MyWindow Constructor and Widget Hierarchy | 9 |
| Main Semantics | 10 |
| QApplication Initialization: | 10 |
| Creating a Window | 10 |
| Starting the Event Loop: | 10 |
| main.moc | 11 |
| Defining our own window object | 11 |
| Creating a button | 12 |
| Include Necessary Headers | 12 |
| Creation | 12 |
| Making the button do something | 12 |
| Creating Labels | 14 |
| QStrings | 14 |
| Creating a QString | 14 |
| Converting to QString | 14 |
| Converting numeric types to QString | 15 |

Preface

Creating a simple GUI application with Qt in C++ involves several steps. First, ensure you have the Qt framework installed on your Linux system. You can download QT on arch linux using pacman.

```
1  pacman -Sy qt5
```

Source File (Simple GUI)

Here we make a simple GUI to show how the projects are built

```
1  #include <QApplication>
2  #include <QProcess>
3  #include <QWidget>
4  #include <QPushButton>
5  #include <iostream>
6
7  int main(int argc, char *argv[]) {
8      QApplication app(argc, argv);
9
10     QWidget window;
11
12     QPushButton *button = new QPushButton("Click me", &window);
13     button->setGeometry(50, 50, 80, 30);
14
15     window.resize(250, 150);
16     window.setWindowTitle("Button Example");
17     window.show();
18
19     return app.exec();
20 }
```

.pro file (Making project)

To build our projects, we need to assemble a .pro file. A sample .pro file for the file above would look something like

```

1  TEMPLATE = app
2  TARGET = your_app_name
3  CONFIG += console c++11
4  QT += widgets
5
6  # Input
7  SOURCES += main.cpp
8
9  # If you have additional source files, list them here
10 # SOURCES += source1.cpp source2.cpp
11
12 # If you have header files, list them here
13 # HEADERS += header1.h header2.h
14
15 # If you have UI files created using Qt Designer, list them here
16 # FORMS += mainwindow.ui

```

- **TEMPLATE=app** tells Qt's build system, qmake, that your project is an application. This means qmake will generate a Makefile to build an executable.
- **TARGET=you_app_name** specifies the name for the executable
- **CONFIG += console c++11** is used to define various configuration options for the build process
 - **console:** This option is used to specify that the application is a console application. This is particularly relevant on Windows, where it determines whether a console window is opened alongside your application. For GUI applications, you typically wouldn't include console, as GUI applications on Windows usually don't need a console window. On Linux and macOS, the distinction is less significant, as terminal-based and graphical applications are not as strictly separated as on Windows.
 - **c++11** tells the compiler to use the C++11 standard
- **QT += widgets** is used to add the widgets module from the list of modules to be included in the application
- **SOURCES += main.cpp** is used to add source files in the build process

Building the app

Once we have made the .pro file, we can begin the build process

```

1  qmake filename.pro # First
2  make # Then

```

Other options for pro file variables

- **TEMPLATE:**

- **app:** This is used for building an application. It will create an executable. When you're developing a typical GUI or console application, you use `TEMPLATE = app`.
- **lib:** This is used for building a library. If you're developing a library (either static or dynamic), you would use `TEMPLATE = lib`. When you choose this, qmake will generate a Makefile suitable for building a library instead of an executable.
- **subdirs:** This is used for a project that contains multiple subprojects. The subdirs template is useful when your project is large and split across multiple directories, each of which is a different project (either an application or a library). With `TEMPLATE = subdirs`, qmake will manage these subprojects according to the instructions you provide in the .pro file.
- **aux:** This template is used for auxiliary files that are not compiled but are included in the project. It's less commonly used compared to the other templates.
- **vcapp and vclib:** These are for Visual Studio integration on Windows.

- **CONFIG:**

- **debug:** Builds the application with debugging symbols. Useful for debugging the application.
- **release:** Builds the application in release mode, which typically includes optimizations and lacks debugging symbols.
- **qml_debug:** Enables debugging of QML code. Useful if you are using QML for your application's UI.
- **qt:** Ensures that Qt-specific build steps are executed, like running the Meta-Object Compiler (MOC) on classes that use Qt's signal and slot mechanism.
- **c++11, c++14, c++17, c++20:** Specifies the C++ standard to be used. You should match this with the version of C++ your code is written in.
- **warn_on:** Enables all compiler warnings. This is useful for ensuring that potential code issues are highlighted during the build process.
- **static:** When building a library, this option specifies that the library should be static rather than dynamic.
- **shared:** Opposite of static, this is used for building shared libraries.
- **testcase:** Used when building a project as a test case using Qt Test.
- **lex yacc:** Includes support for Lex and Yacc if you are using these tools in your project.
- **thread:** Enables multi-threading support in your application.
- **exceptions:** Enables support for C++ exceptions.
- **no_keywords:** Disables the use of Qt-specific keywords like slots, signals, and emit to avoid conflicts with third-party libraries.
- **opengl:** Enables the use of OpenGL in the application.
- **widgets:** Includes the QtWidgets module, necessary for applications using Qt Widgets.
- **network:** Includes the QtNetwork module for network programming.
- **sql:** Includes the Qt SQL module for database operations.
- **xml:** Includes the Qt XML module for XML processing.
- **link_pkgconfig:** Allows the use of pkg-config to find libraries.
- **precompile_header:** Enables the use of precompiled headers to speed up compilation.

- **QT: (List of modules we can add or remove)**

- **core:** This is included by default and provides core non-GUI functionality. It's essential for any Qt application.
- **gui:** Also included by default in most cases, this module is necessary for any application that uses Qt's graphical user interface elements.
- **widgets:** For applications that use QWidget-based user interfaces. This is a key module for traditional desktop GUI applications.
- **network:** Adds network communication capabilities, like handling TCP/IP connections, for your application.
- **sql:** If your application needs to interact with SQL databases, this module provides the necessary functionality.
- **qml:** Necessary for applications that use the QML language for designing user interfaces, especially in combination with the Qt Quick module.
- **quick:** Used in conjunction with QML to create fluid, dynamic user interfaces. It's a part of the Qt Quick framework.
- **multimedia:** For applications that need to handle audio, video, radio, and camera functionality.
- **bluetooth:** Provides classes for writing Bluetooth applications.
- **concurrent:** Enables easier use of multi-threading in applications.
- **printsupport:** For applications that require printing capabilities.
- **webkit or webengine:** For applications that need to embed web content using WebKit or Qt WebEngine.
- **xml:** Adds support for reading and writing XML data.
- **opengl:** For applications that use OpenGL for rendering graphics.
- **testlib:** For writing unit tests.
- **positioning:** For applications that require location and positioning functionality.
- **sensors:** Access to various hardware sensors like accelerometers, gyroscopes, etc.
- **serialport:** For applications that communicate with devices over serial ports.
- **svg:** Support for Scalable Vector Graphics (SVG) files.
- **dbus:** For applications that communicate with other applications using the D-Bus protocol (mostly relevant on Linux).

Includes

The main include we need to build our GUI apps is `<QApplication>`, this class manages the GUI application's control flow and main settings

The `<QWidget>` class is the base class of all user interface objects.

The `<QProcess>` allows us to launch external processes

Example Application

Here is a sample application which creates a clickable button that takes us to a certain directory in a new terminal window

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <QPushButton>
4  #include <QProcess>
5  #include <iostream>
6
7  class MyWindow : public QWidget {
8      Q_OBJECT
9
10     public:
11         MyWindow(QWidget *parent = nullptr) : QWidget(parent) {
12             QPushButton *button = new QPushButton("Open Terminal", this);
13             button->setGeometry(50, 50, 120, 30);
14             connect(button, &QPushButton::clicked, this, &MyWindow::onButtonClicked);
15         }
16
17     public slots:
18         void onButtonClicked() {
19             QProcess::startDetached("kitty --working-directory=/home/datura/tmp/cpp");
20         }
21 };
22
23 int main(int argc, char *argv[]) {
24     QApplication app(argc, argv);
25
26     MyWindow window;
27     window.resize(250, 150);
28     window.setWindowTitle("Button Example");
29     window.show();
30
31     return app.exec();
32 }
33
34 #include "main.moc"
```


Q_OBJECT

The `Q_OBJECT` macro is essential in Qt applications for any class that defines signals or slots, and it enables several key features provided by Qt's meta-object system.

Signals and Slots

Signals and slots are fundamental aspects of Qt and form the basis of its event communication system. They are used for communication between objects and are an integral part of Qt's programming model, especially in GUI applications. Here's a breakdown of what they are and how they work:

Signals

Definition: A signal is a message sent by an object to indicate that some event has occurred or some state has changed. In Qt, signals are declared in a class, but they are not implemented in the class. They are just emitted (or "fired") when the event they represent occurs.

Usage: Signals are used to broadcast information. Any number of slots can listen and react to a particular signal.

Syntax: In Qt, signals are declared with the `signals` keyword.

```
1  signals:
2      void mySignal();
```

Slots

Definition: A slot is a function that is used to receive and respond to a signal. Slots can be normal member functions of a class.

Usage: Slots are used to perform actions in response to the occurrence of the event signified by a signal. A slot does not know if it has received a signal from a signal or directly as a regular function call.

Syntax: Slots are declared with the `slots` keyword or can be just regular member functions. They can also be private or protected.

```
1  public slots:
2      void mySlot();
```

Connecting Signals to Slots

Mechanism: In Qt, objects communicate with each other via signals and slots. When a signal is emitted, all connected slots are called.

Dynamic Connection: The connection between signals and slots can be made at runtime using the `QObject::connect()` function.

```
1 connect(sender, SIGNAL(mySignal()), receiver, SLOT(mySlot()));
```

The MyWindow Constructor and Widget Hierarchy

In Qt, the concept of parent-child relationships is pivotal for managing widgets (the graphical user interface elements). When you create a custom widget like `MyWindow`, which inherits from `QWidget`, you often include a constructor that allows for specifying a parent widget. This relationship is crucial for several aspects of a widget's behavior and lifecycle.

```
1 MyWindow(QWidget *parent = nullptr) : QWidget(parent) {  
2     // Constructor body  
3 }
```

- **Optional Parent Argument:** The constructor of `MyWindow` takes an optional parameter `parent`, which is a pointer to a `QWidget`. The default value of this parameter is `nullptr`.
- **Delegating to QWidget Constructor:** Inside the constructor, we use an initialization list to call the base class `QWidget`'s constructor with the `parent` argument. This is a common C++ technique for initializing base class members.
- **Behavior Based on parent Argument:**
 - **Top-Level Window:** If `nullptr` is passed to `MyWindow` (or no argument is provided), it implies that `MyWindow` does not have a parent widget. In this case, `MyWindow` acts as a top-level window. This means it can be a standalone window with its own window decorations (like a title bar, minimize/maximize buttons, etc.).
 - **Child Widget:** If a valid `QWidget` pointer is passed as the `parent`, `MyWindow` becomes a child widget of the specified parent widget. As a child widget, it will be contained within the parent widget's window and subject to its geometry and visibility. Additionally, the parent widget will manage the lifetime of `MyWindow`, automatically deleting it when the parent widget is destroyed.

This constructor design allows `MyWindow` to be versatile: it can either be an independent window or part of a larger interface, depending on how it's instantiated. This flexibility is a key feature of Qt's approach to building user interfaces, where the composition of widgets can be dynamically arranged.

Main Semantics

In a Qt application, the main function serves as the entry point and is responsible for setting up and running the application's main event loop

QApplication Initialization:

```
1  QApplication app(argc, argv);
```

- QApplication is a class that manages application-wide resources and is necessary for any Qt GUI application.
- It needs to be instantiated at the beginning of main.
- argc and argv are passed to QApplication to handle any command-line arguments that are relevant to Qt applications (like GUI style, plugin paths, etc.).

Creating a Window

If no custom window class is made, we can create a window with

```
1  QWidget window;  
2  window.resize(250, 150);  
3  window.setWindowTitle("Title Text");  
4  Window.show();
```

Starting the Event Loop:

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

- app.exec() starts the application's event loop. This is a crucial call; it enters the main loop where events (like mouse clicks, keypresses, or custom events) are received and dispatched to the appropriate widgets.
- The event loop continues to run until exit() is called, usually as a response to events like closing the main window.

main.moc

- **What is main.moc?:** The .moc files are generated by the Meta-Object Compiler (MOC) in Qt. MOC is a tool that processes Qt's specific extensions, like the Q_OBJECT macro, signals, and slots. It generates standard C++ code from these extensions, enabling features like signal-slot connections and introspection.
- **Role of main.moc:** Typically, for classes that include the Q_OBJECT macro or define signals and slots, a corresponding .moc file is generated. However, it's unusual to have a main.moc. In standard Qt applications, the main function usually doesn't define a new class and doesn't include Q_OBJECT. If your main.cpp includes definitions of such Qt classes, then main.moc might be generated and should be included at the end of the main.cpp file. This is not common practice and usually indicates that the application structure could be improved by moving Qt class definitions out of main.cpp.

Defining our own window object

```
1  #include <QApplication>
2  #include <QWidget>
3  #include <iostream>
4  #include <iomanip>
5
6  class MainWindow : public QWidget {
7  private:
8      Q_OBJECT;
9
10 public:
11     MainWindow(QWidget* parent=nullptr) : QWidget(parent) {
12
13     }
14
15 };
16
17 int main(int argc, char* argv[]) {
18     QApplication app(argc, argv);
19
20     MainWindow window;
21     window.resize(1080,700);
22     window.setWindowTitle("Title Name");
23     window.show();
24
25     return app.exec();
26 }
27 #include "main.moc"
```

Creating a button

Now that we have most of the jargon out of the way, lets create some stuff.

Include Necessary Headers

```
1 #include <QPushButton>
```

Creation

In our custom window class,

```
1 MainWindow(QWidget *parent = nullptr) : QWidget(parent) {  
2     QPushButton *button = new QPushButton("Do Something", this);  
3     button->setGeometry(50, 50, 120, 30);  
4 }
```

- **QPushButton *button = new QPushButton("Open Terminal", this):** creates a new instance of QPushButton. The text "Open Terminal" is set as the button's label. The this pointer is passed as the parent of the button, which means the button is a child widget of MyWindow. As a child widget, it will be displayed within MyWindow and will be deleted when MyWindow is deleted (automatic memory management by Qt).
- **button->setGeometry(50, 50, 120, 30):** sets the position and size of the button within its parent widget (MyWindow). The button is placed at coordinates (50, 50) with a width of 120 pixels and a height of 30 pixels.

Making the button do something

First, lets create a member function for our window class that provides functionality for our button

```
1 #include <QProcess>
```

```
1 public slots:  
2     void onButtonClick() {  
3         QProcess::startDetached("kitty --working-directory=$HOME/tmp/cpp");  
4     }  
5 }
```

Then in the constructor we can add the connection

```
1 connect(object1, signal1, object2, slot2);  
2 connect(button, &QPushButton::clicked, this, &MainWindow::onButtonClicked);
```

- `connect(button, QPushButton::clicked, this, MainWindow::onButtonClicked);` establishes a connection between the clicked signal of the button and the `onButtonClicked` slot method of `MainWindow`. This is using the signal-slot mechanism in Qt.
- When the button is clicked, the clicked signal is emitted. Because of the connection established by `connect`, this will trigger the `onButtonClicked` method in the `MainWindow` class.

Args

- **First Argument - button:**
 - This is the source object that emits the signal. In your case, it's the pointer to the `QPushButton` instance you've created. The signal will be emitted from this button.
- **Second Argument - `QPushButton::clicked`:**
 - This specifies the signal you want to connect from the source object. The `QPushButton::clicked` is a pointer to the clicked signal of the `QPushButton` class. This signal is emitted by Qt when the button is clicked.
- **Third Argument - `this`:**
 - This is the receiver object, which is the object that owns the slot method you want to call. In this case, `this` refers to the current instance of the `MainWindow` class, indicating that the slot method belongs to this instance.
- **Fourth Argument - `MainWindow::onButtonClicked`:**
 - This specifies the slot, which is the method that will be called in response to the signal. `MainWindow::onButtonClicked` is a pointer to the `onButtonClicked` method of the `Wainwindow` class. This method should be defined in `MainWindow` and will be executed when the button is clicked.

Creating Labels

First, we need to make sure that we have `<QLabel>` included

```
1 #include <QLabel>
```

Then, in our windows constructor, we can create the label

```
1 label = new QLabel("Hello, World!", this);
2 label->setAlignment(Qt::AlignCenter);
3 label->setGeometry(0, 75, 250, 50); // Adjust geometry as needed
4 label->hide();
```

QStrings

QString is a fundamental part of the Qt framework, designed to represent strings of text in a way that is optimized for performance and flexibility, especially in the context of internationalization. It's a powerful alternative to standard C++ string types like `std::string` and C-style strings (`char*`).

We must use QStrings instead of `std::string` in our applications to avoid getting errors.

Creating a QString

Before we create any QStrings, we must include the header.

```
1 #include <QString>
```

Then we can create our QString objects

```
1 QString a = "My QString";
```

Converting to QString

```
1 std::string a = "String"
2 QString b = QString::fromStdString(a); // QString::toStdString() for the reverse
```

Converting numeric types to QString

```
1  QString a = QString::number(20);
```