
بسم الله الرحمن الرحيم

## مقام معظم رهبری:

علم برای یک ملّت مهم‌ترین ابزار آبرو و پیشرفت و اقتدار است.

۱۳۹۴/۰۸/۲۰


# Qt Training in C++

Lecturer: Ali Panahi

Autumn 1401

# Introduction to Qt

## What is Qt ?

Qt is cross-platform software for creating graphical user interfaces as well as cross-platform applications that run on various software and hardware platforms such as Linux, Windows, macOS, Android or embedded systems with little or no change in the underlying codebase while still being a native application with native capabilities and speed.

Qt is currently being developed by The Qt Company, a publicly listed company, and the Qt Project under open-source governance, involving individual developers and organizations working to advance Qt. Qt is available under both commercial licenses and open-source GPL 2.0, GPL 3.0, and LGPL 3.0 licenses.

# Introduction to Qt

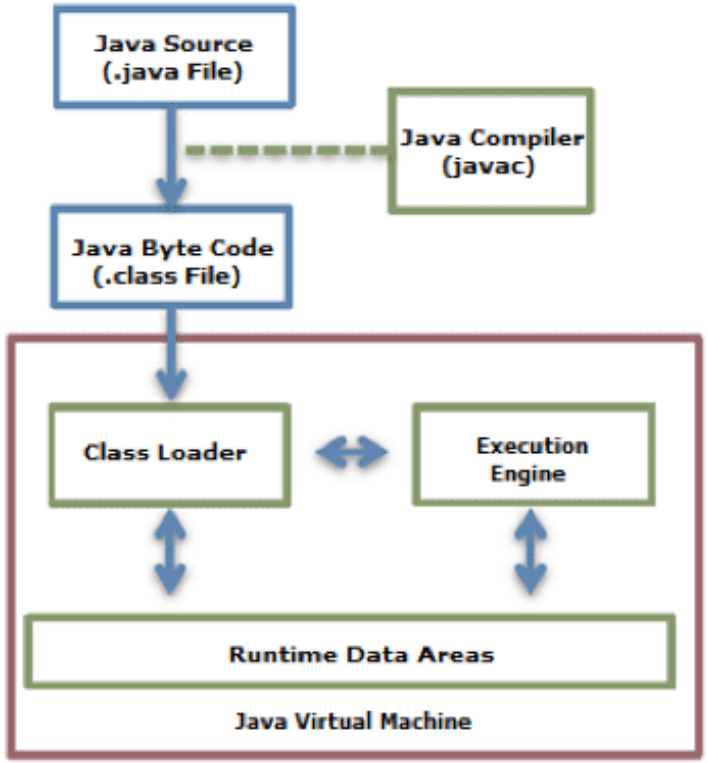
## Why use Qt ?

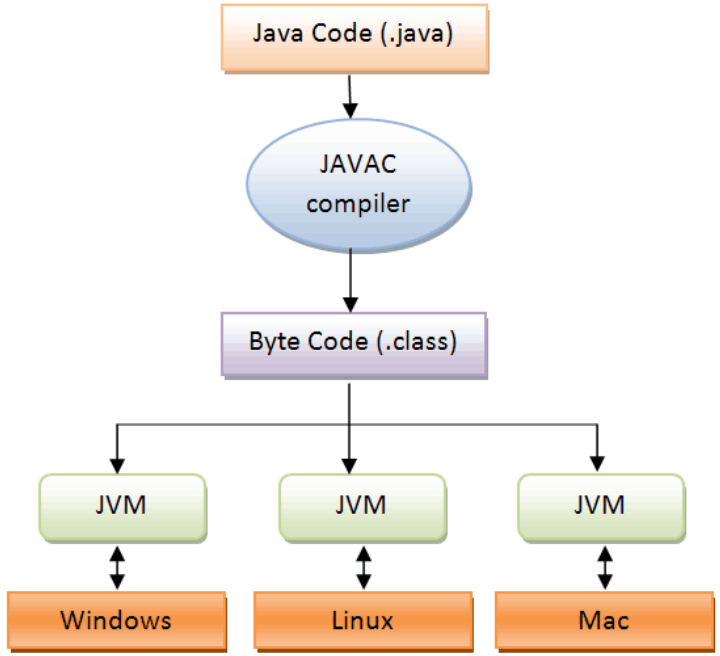
- Design and develop great user experiences
- Qt saves you money
- Get your products to market faster
- Performance, delivered
- One framework, fewer dependencies
- Develop for any platform
- We speak many languages
- Flexible. Reliable. Qt.
- Open source and future-proof

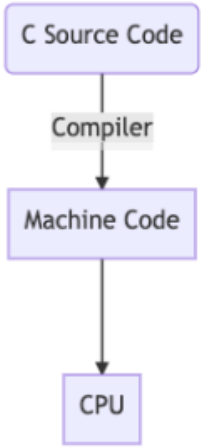
## Java vs .Net vs C++ vs QT vs .Net Core

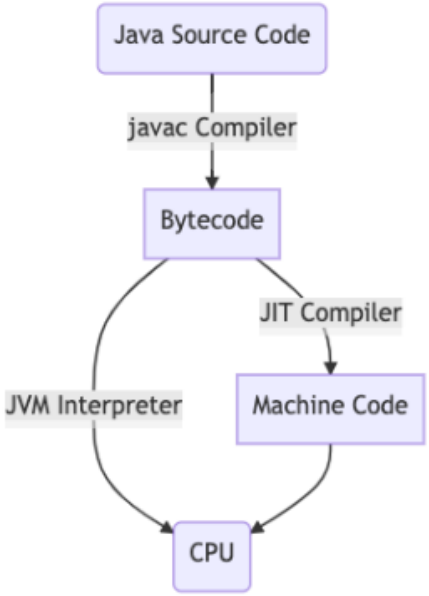
- Java
  - Java is object-oriented programming language
  - Java is a general-purpose
  - Java is cross-platform
  - Java is class-based
  - Java is managed
  - Java is free
  - Low execution speed
  - Disassembling object code
  - High development speed

# Introduction to Qt >> Java



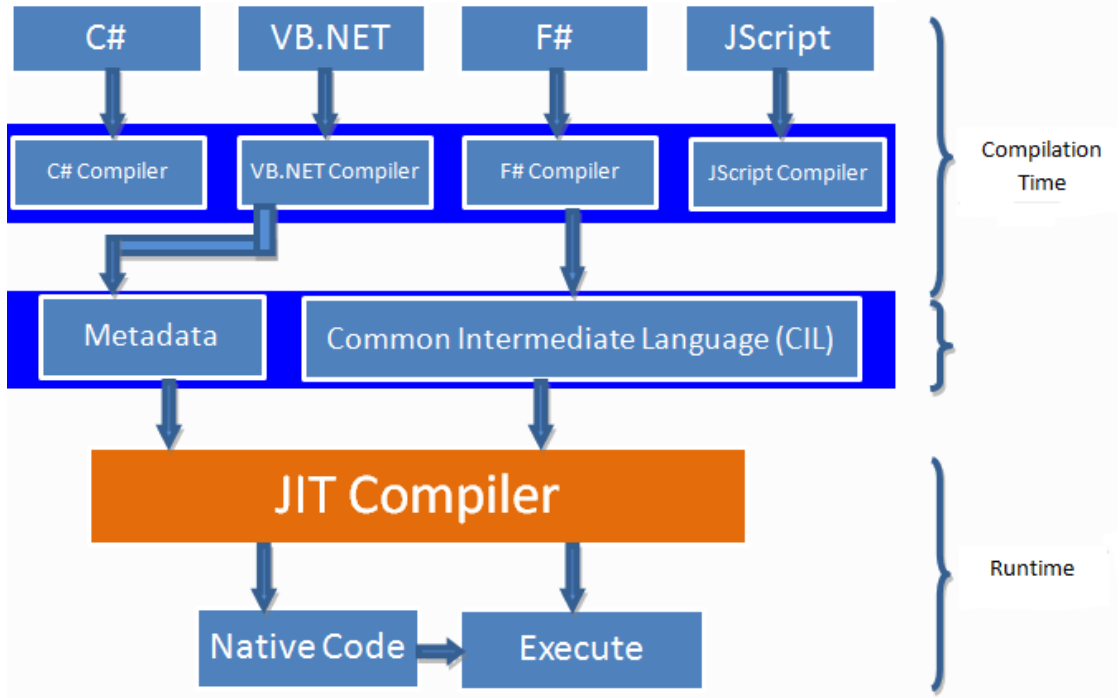



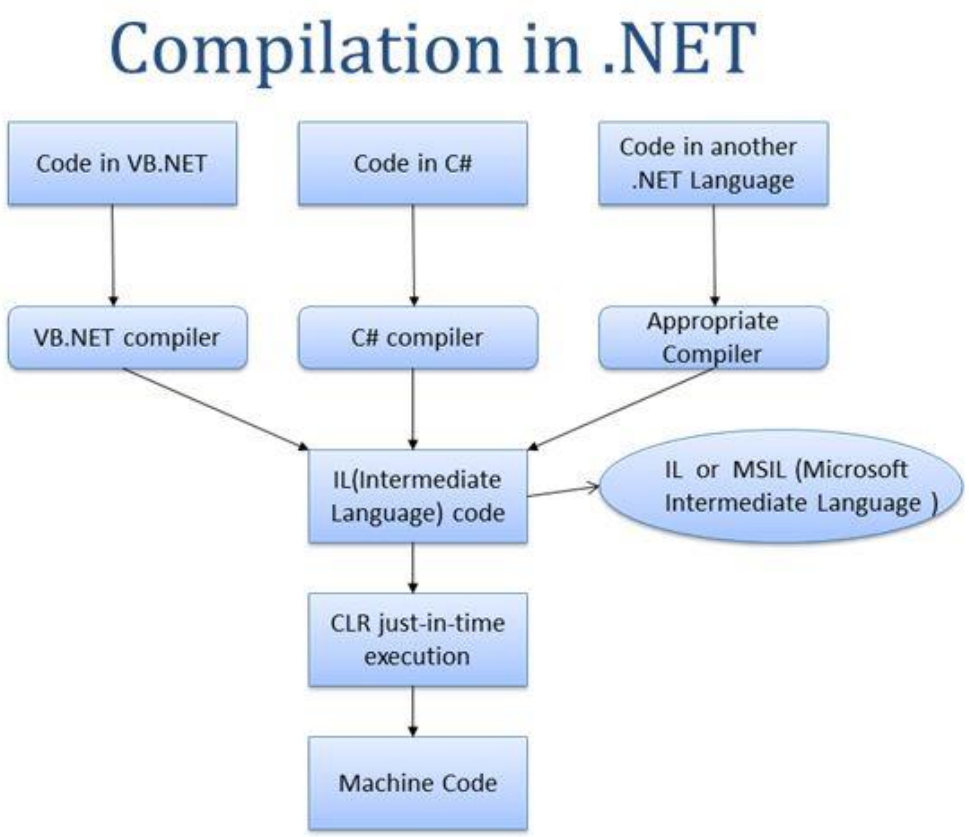



## Java vs .Net vs C++ vs QT vs .Net Core

- .Net
  - Multi-language Support
  - .Net is managed
  - Automatic resource management
  - .Net is commercial (Microsoft)
  - Low execution speed
  - Disassembling object code
  - Only for windows
  - High development speed

# Introduction to Qt >> .Net

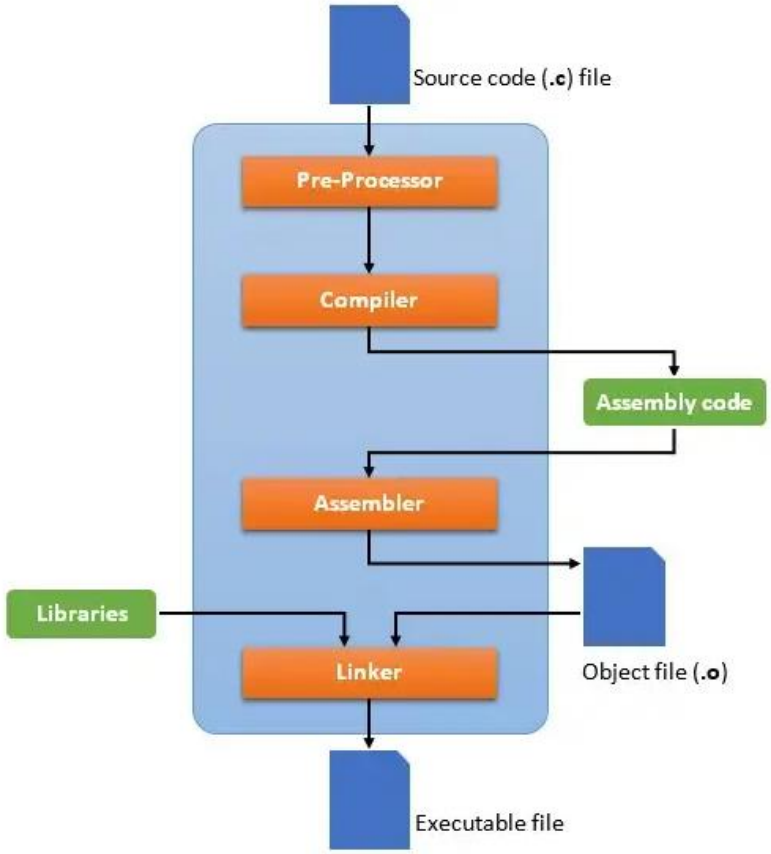


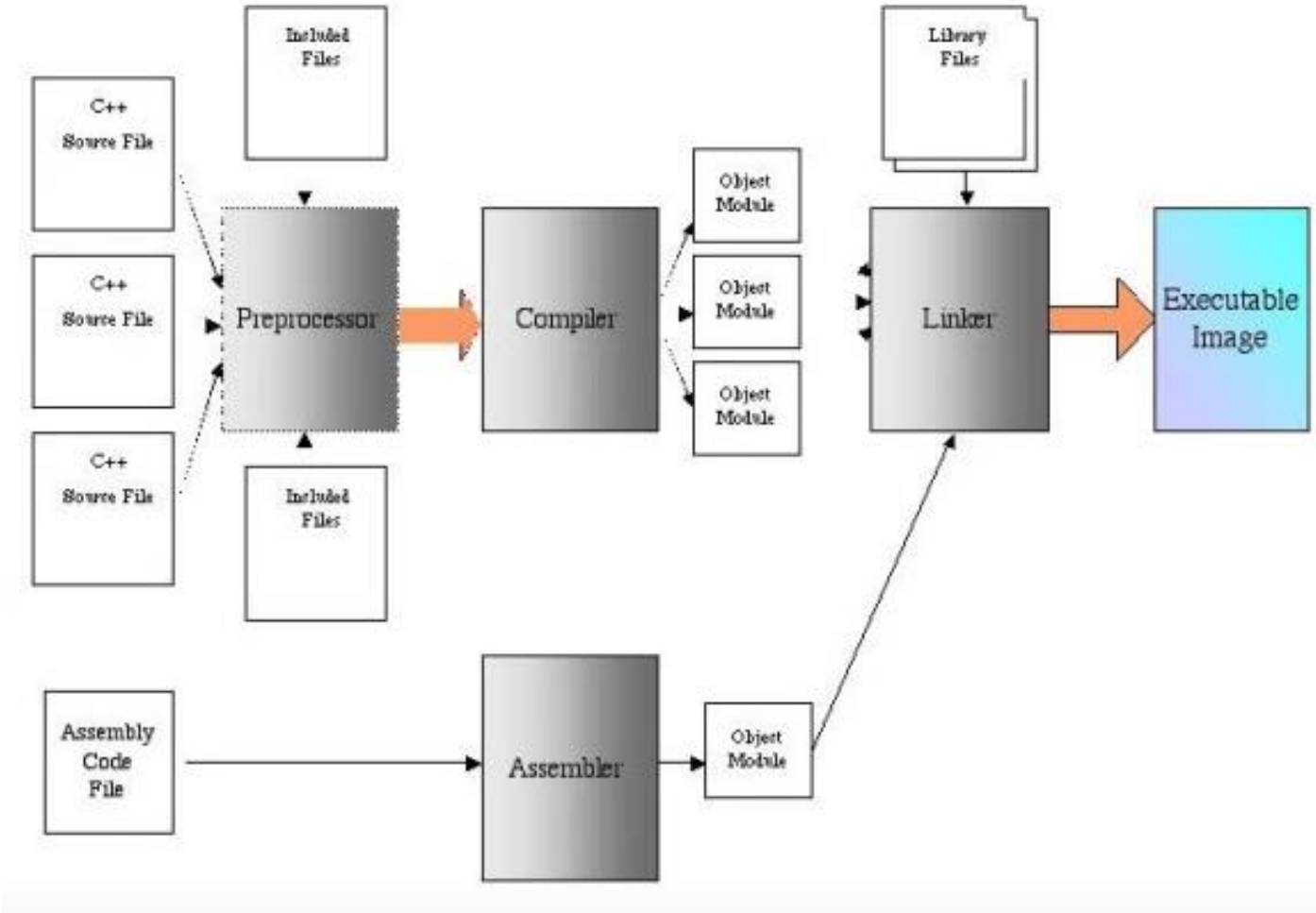



## Java vs .Net vs C++ vs QT vs .Net Core

- C/C++
  - It is native
  - Creating libraries that can be used in other languages
  - Powerful & fast
  - High security
  - Platform dependent
  - Irreversibility and reverse engineering of codes
  - It has no memory management
  - Low development speed

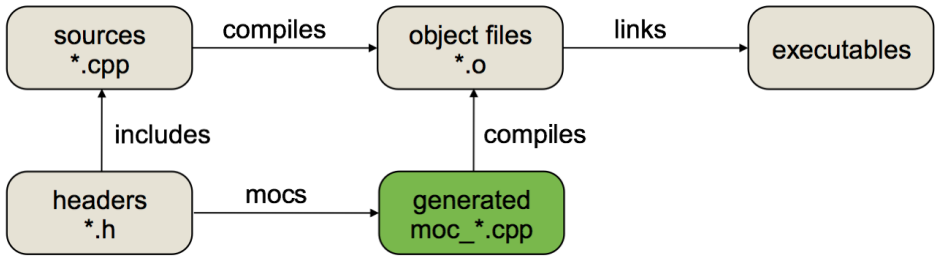
# Introduction to Qt >> C/C++

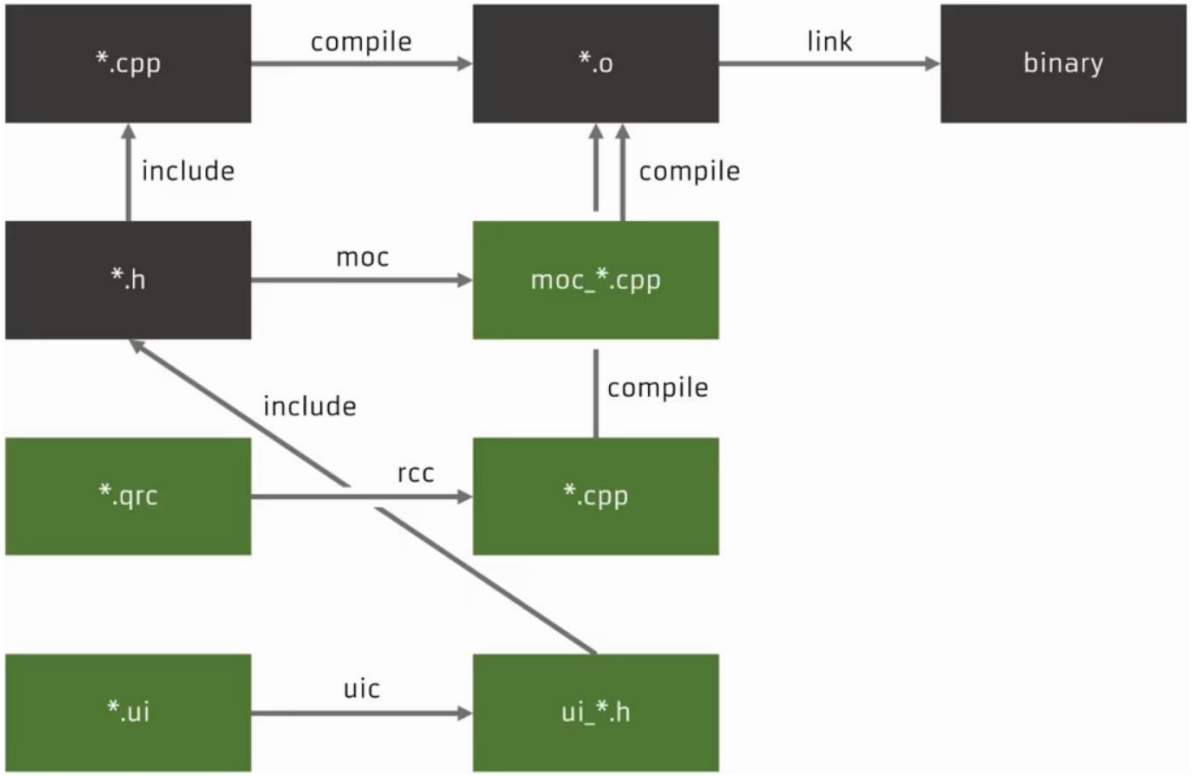


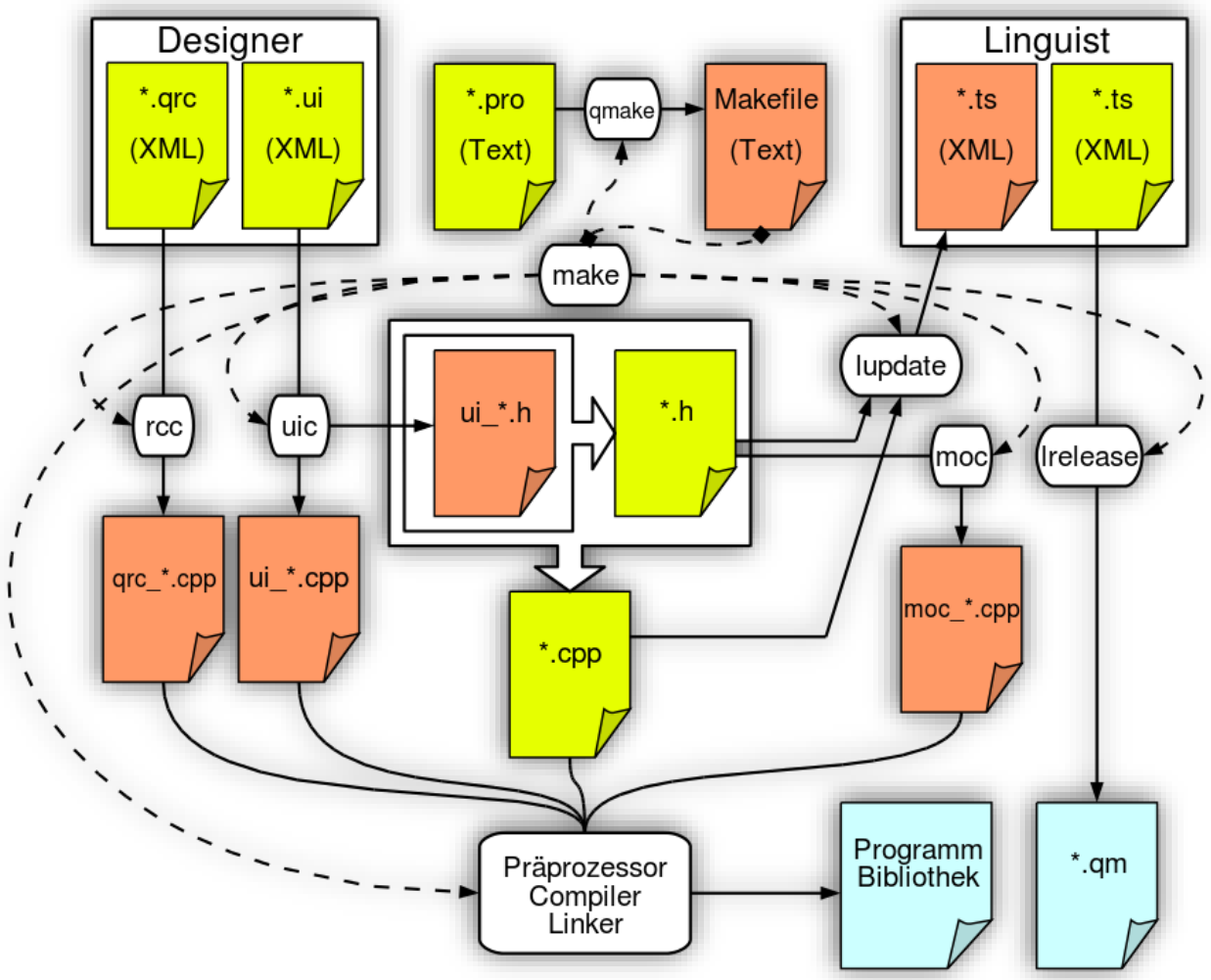



## Java vs .Net vs C++ vs QT vs .Net Core

- Qt
  - It is native
  - High security
  - Cross-platform (Windows, Linux, Mac)
  - Many possibilities
  - Free and open source


sources
*.cpp
compiles
object files
*.o
links
executables
includes
compiles
headers
*.h
mocs
generated
moc_*.cpp

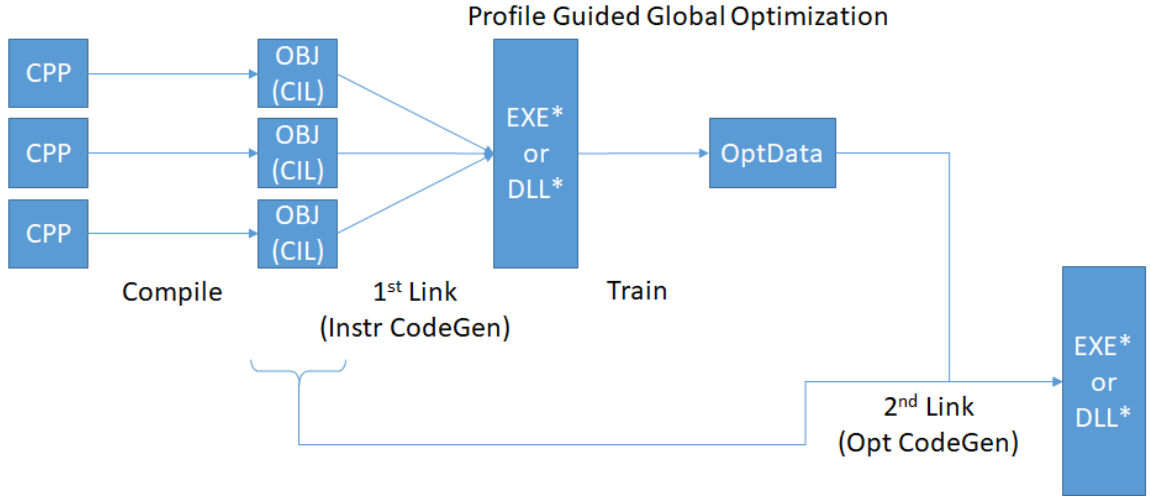

*.cpp
compile
*.o
link
binary
include
compile
*.h
moc
moc_*.cpp
include
compile
*.qrc
rcc
*.cpp
*.ui
uic
ui_*.h

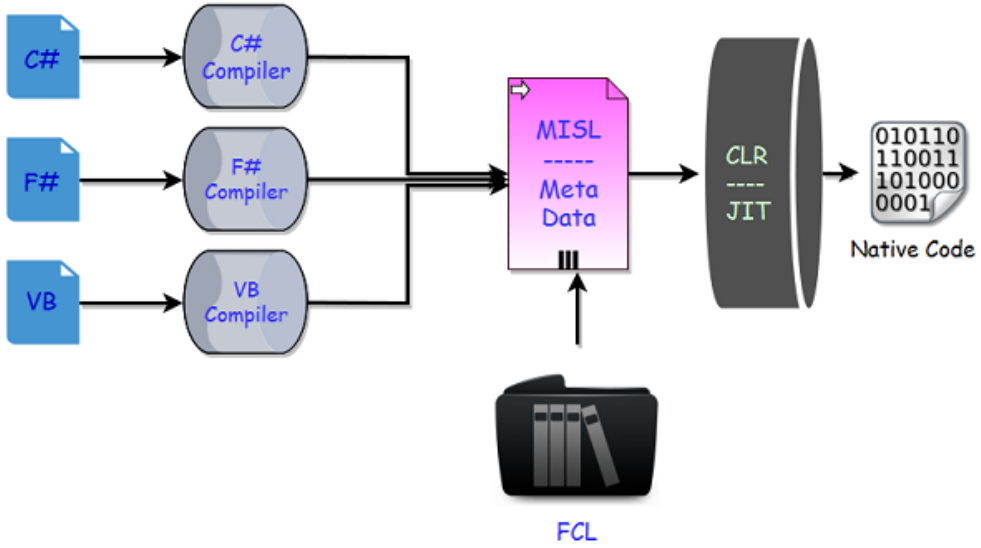

Designer
*.qrc
(XML)
*.ui
(XML)
*.pro
(Text)
qmake
Makefile
(Text)
Linguist
*.ts
(XML)
*.ts
(XML)
make
lupdate
rcc
uic
ui_*.h
*.h
moc
lrelease
qrc_*.cpp
ui_*.cpp
*.cpp
moc_*.cpp
Präprozessor
Compiler
Linker
Programm
Bibliothek
*.qm

## Java vs .Net vs C++ vs QT vs .Net Core

- .Net Core
  - Multi-language support
  - .Net core is managed
  - Automatic resource management
  - .Net core is free and open source (Microsoft)
  - Low execution speed
  - Disassembling object code
  - Cross-platform (Windows, Linux)
  - High development speed

# Introduction to Qt >> .Net Core



## Companies using Qt

Qt helps the best companies in the world deliver better user experiences faster.

Panasonic

In-flight entertainment systems

HARMAN

In-vehicle infotainment system

AMD

Graphics software

medec

Anesthesia & critical care medical devices

ZUKEN

EDA & CAD end-to-end engineering solutions

DENSO

Automotive mobility technology

# Introduction to Qt >> Qt Versions

| Version | Release date / Support until | Target (Windows) |
|---|---|---|
| Qt 0.90 | 1995 | - |
| Qt 1.0 | 1996 | - |
| Qt 2.0 | 1999 | - |
| Qt 3.0 | 2001 | - |
| Qt 4.0 | 2005 | - |
| Qt 4.8 LTS (4.8.7) | 2011 | XP, 7, 8.1, 10 |
| Qt 5.6 LTS (5.6.3) | 2016 / 2019 | - |
| Qt 5.9 LTS (5.9.9) | 2017 / 2020 | - |
| Qt 5.12 LTS (5.12.12) | 2018 / 2021 | - |
| Qt 5.15 LTS | 2020 / 2025 | 7, 8.1, 10, 11 |
| Qt 6.2 LTS (6.2.6) | 2021 / 2024 | 10, 11 |
| Qt 6.4 | 2022 / 2023 | 10, 11 |

# Introduction to Qt

## Languages that support the Qt library

Qt is developed with C++ language.

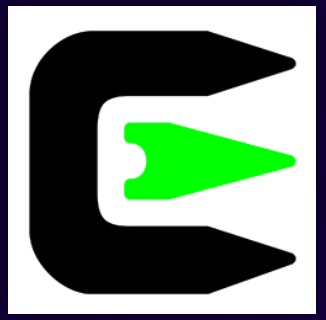
Qt can be used in several programming languages other than C++, such as Python, Javascript, C# and Rust via language bindings; many languages have bindings for Qt 5 and bindings for Qt 4.

# Introduction to Qt

## Qt compilers

- Windows
  - MSVC
  - MinGW (Gnu Based)
  - Cygwin (BSD Based)
- Linux
  - GCC/G++
  - CLang





# Introduction to Qt

## Qt development environments

- Qt Creator
  - Windows
  - Linux
- Visual Studio
  - Windows
- PyCharm (Only for PyQt)
  - Windows
  - Linux


## Installation methods

- From installer
  - Offline installer
  - Online installer
    - From Packages (Linux)
    - From Setup wizard
- From source

## Offline Installer

- It can be downloaded from the link below:
  - URL: https://www.qt.io/offline-installers
- It is precompiled
- Only available for limited editions
- No internet required
- The installer size is large

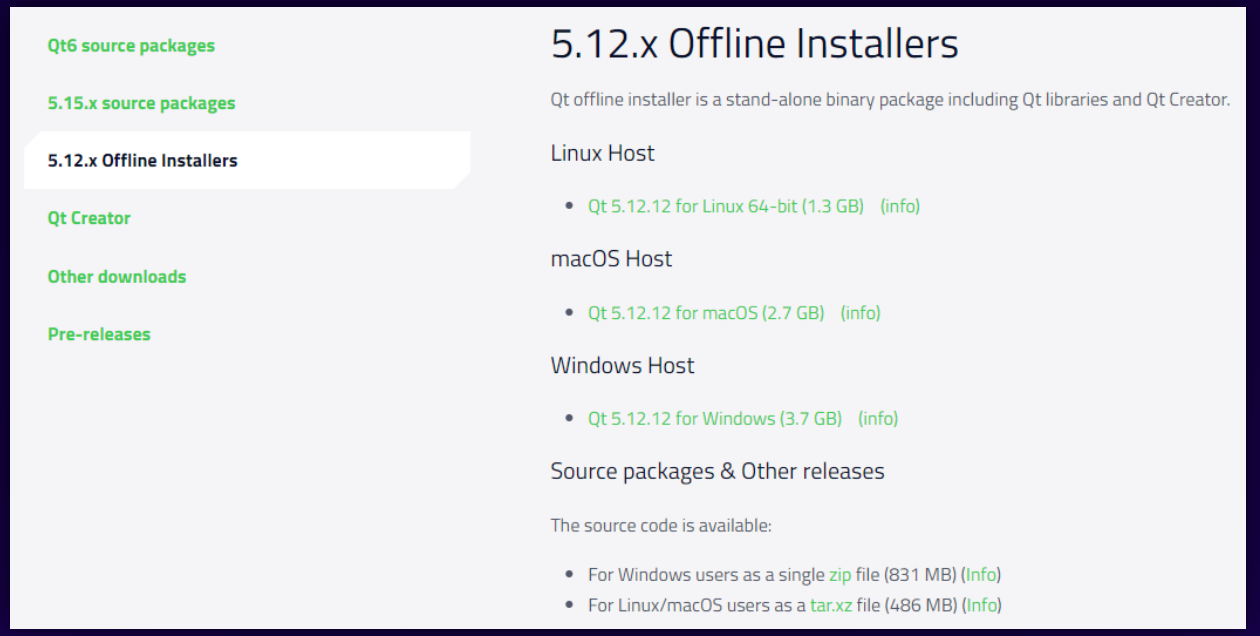
# Offline Installer

## On Windows

qt-opensource-windows-x86-%VERSION%.exe

## On Linux

qt-opensource-linux-x64-%VERSION%.run



## Online Installer

- It can be downloaded from the link below:
  - URL: https://download.qt.io/official_releases/online_installers/
- It is precompiled
- Only available for limited editions
- Internet required
- The installer size is small

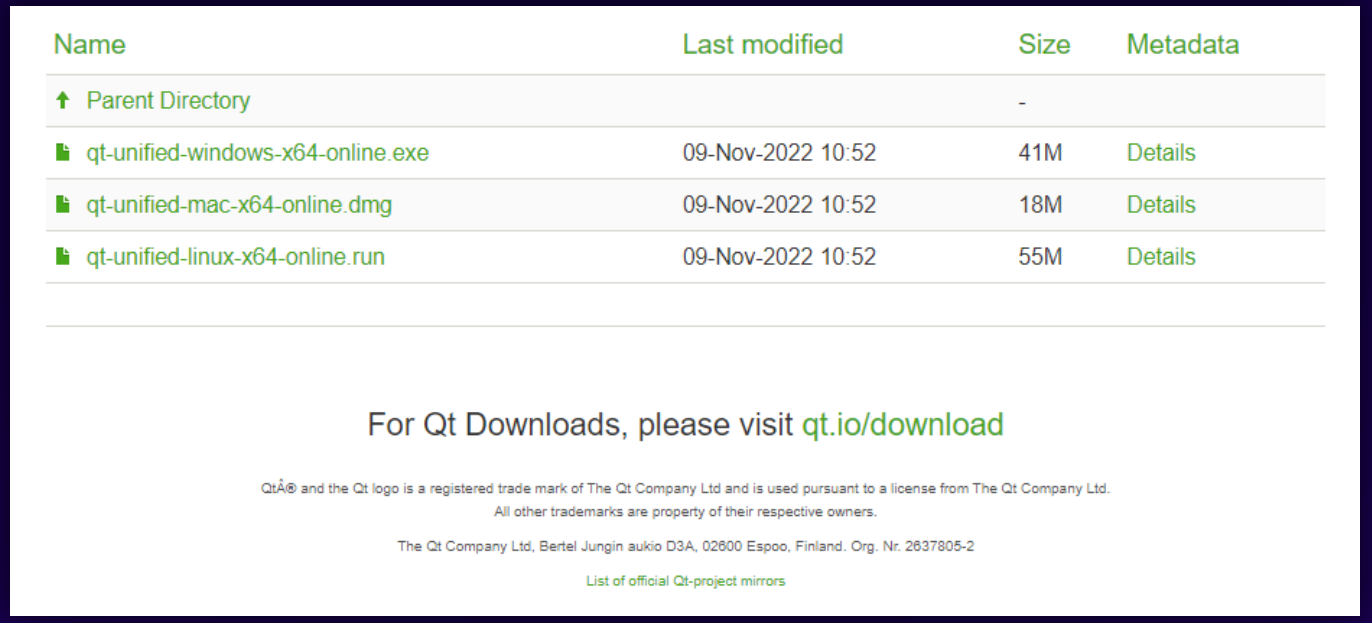
## Online Installer

### On Windows

qt-unified-windows-x64-%VERSION%-online.exe

### On Linux

qt-unified-linux-x64-%VERSION%-online.run



| Name | Last modified | Size | Metadata |
|---|---|---|---|
| Parent Directory | | - | |
| qt-unified-windows-x64-online.exe | 09-Nov-2022 10:52 | 41M | Details |
| qt-unified-mac-x64-online.dmg | 09-Nov-2022 10:52 | 18M | Details |
| qt-unified-linux-x64-online.run | 09-Nov-2022 10:52 | 55M | Details |

For Qt Downloads, please visit qt.io/download

## Online Installer

### On Windows

Run “qt-unified-windows-x64-%VERSION%-online.exe” as administrator

### On Linux

This prerequisite must be installed in the Debian operating system

```
sudo apt-get install libxcb-xinerama0
```

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
```

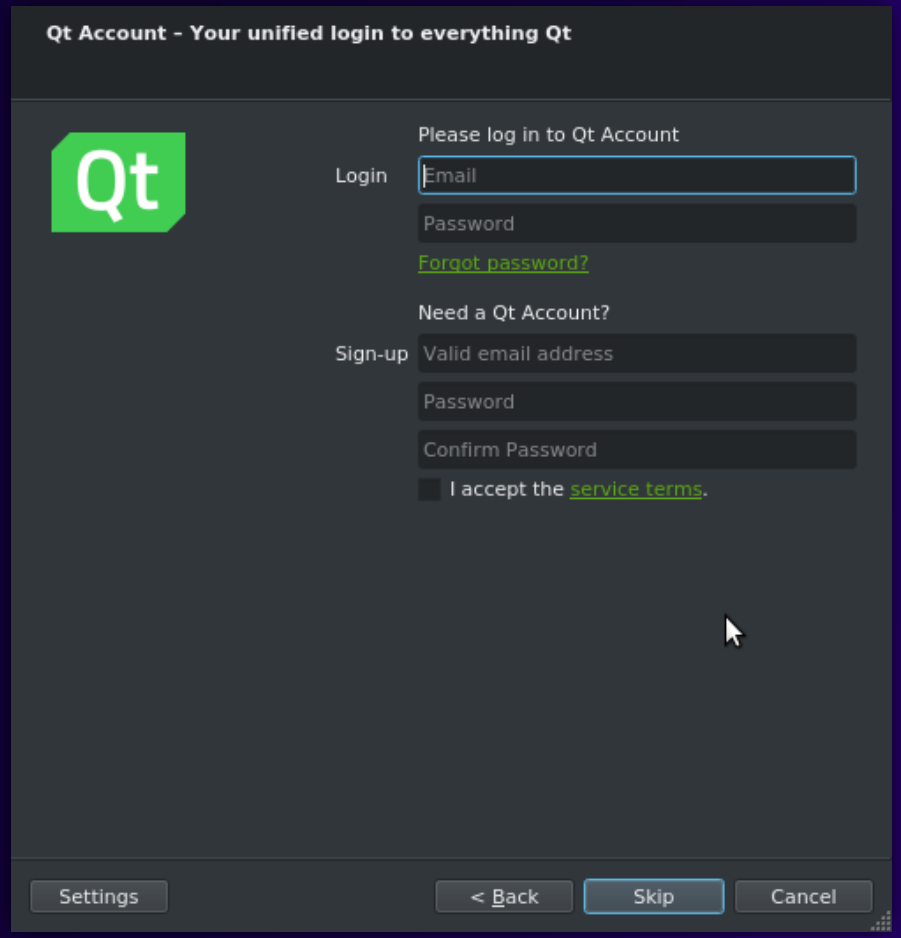
## Offline/Online Installer

### Install Qt in any operative system

Current sample is 4.5.0 version.

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.

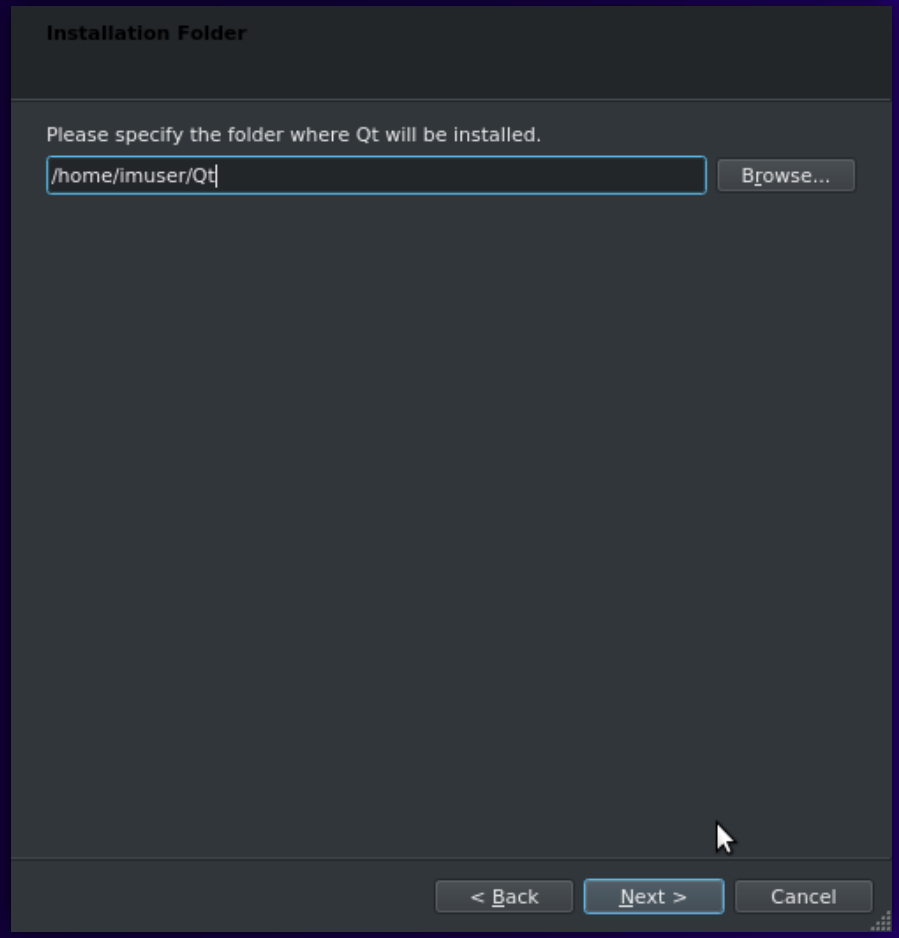
Login with a existing Qt account or create a new one:



## Offline/Online Installer

### Install Qt in any operative system

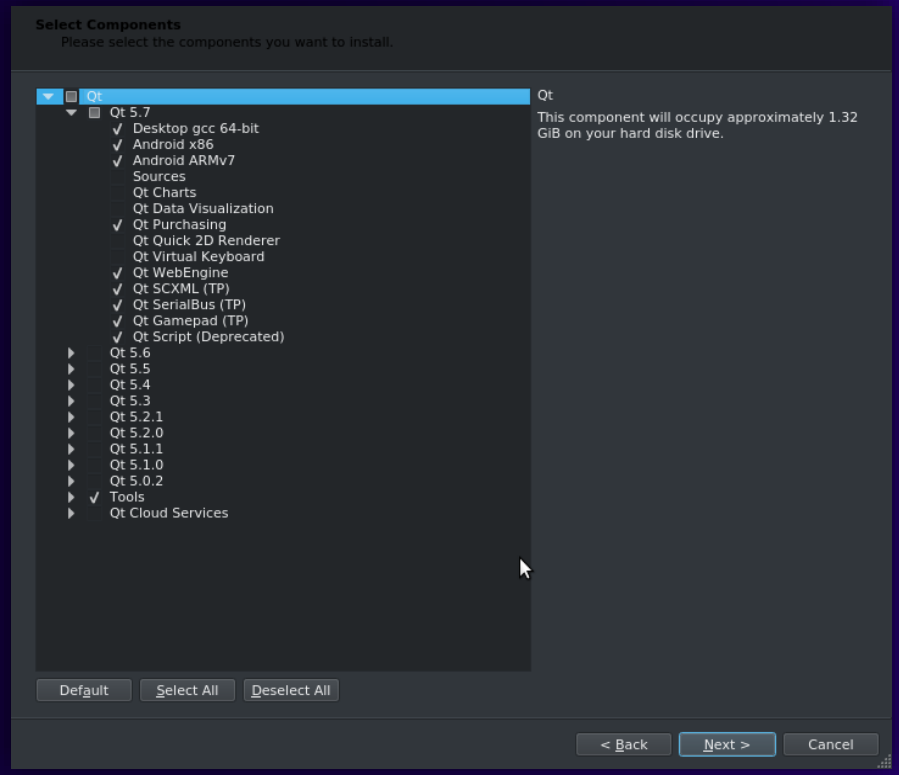
Select a path to install the Qt libraries and tools



## Offline/Online Installer

### Install Qt in any operative system

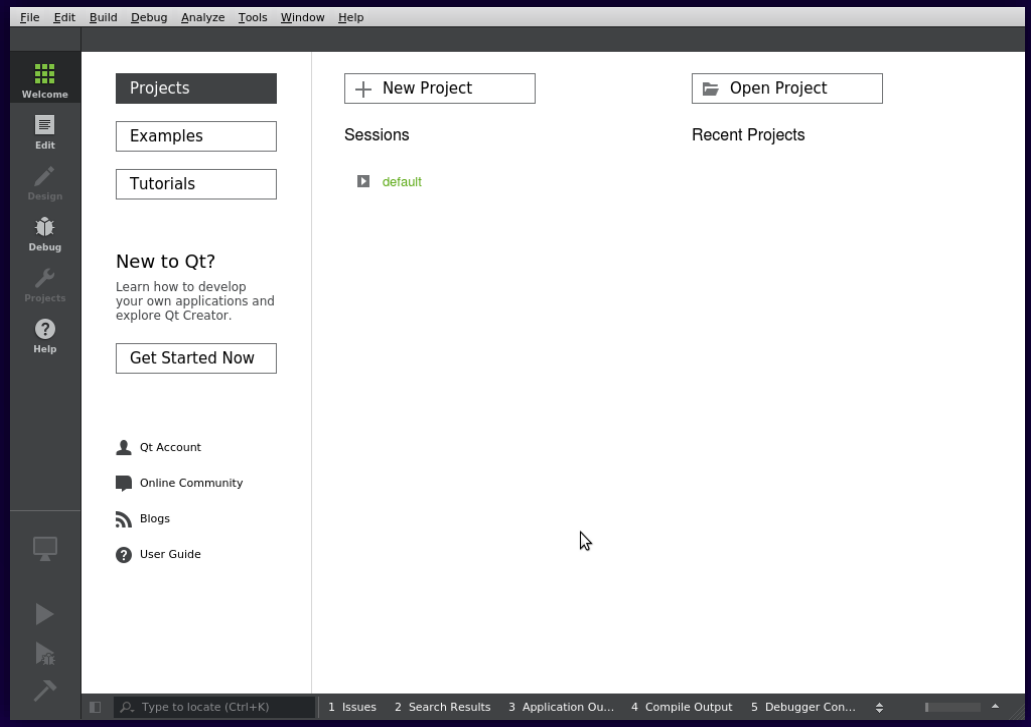
Select the library version and the features you want



# Offline/Online Installer

## Install Qt in any operative system

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.



## Installation From Packages (Linux)

Installation in Debin (11.x) distribution

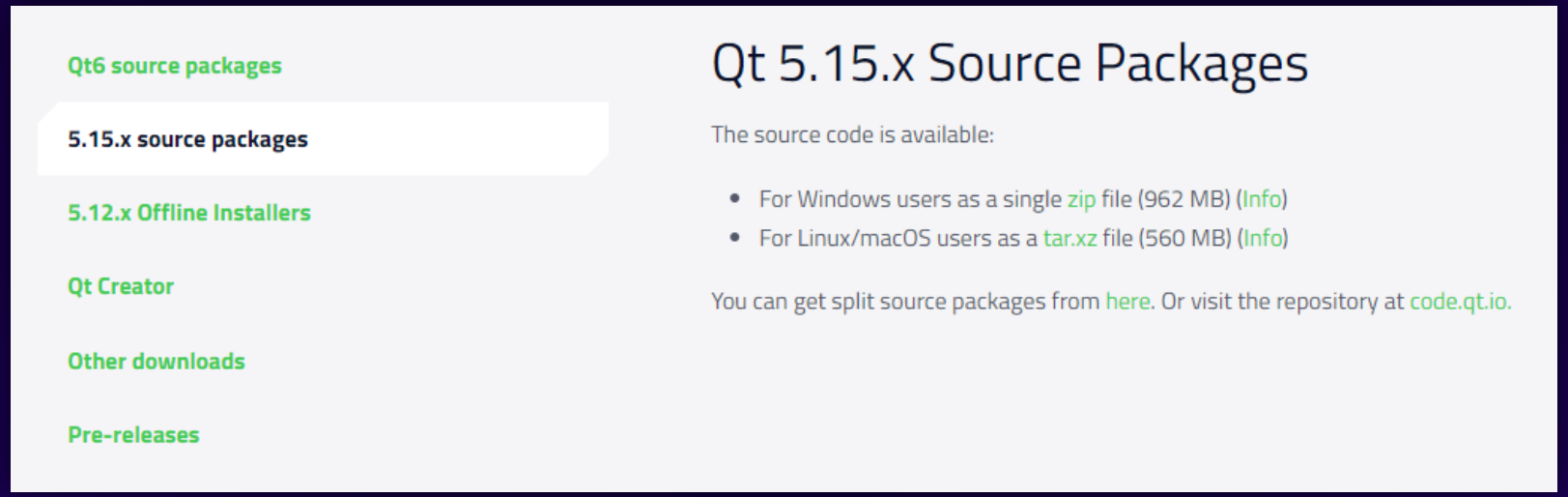
```
$ sudo apt-get update && sudo apt-get upgrade
$ sudo apt install qtbase5-dev qt5-qmake qtbase5-dev-tools
$ sudo apt-get install qtcreator
$ qtcreator
```

## Type of build

- Dynamic
  - Dynamic plug-in is basically a shared library which is loaded at runtime.
    - ✓ Ability to update and patch Qt libraries.
    - ✓ The Qt libraries should be included with the final executable file
- Static
  - Static plug-in is built into your executable (like a static lib).
    - ✓ The libraries are combined with the executable file and a final file is created

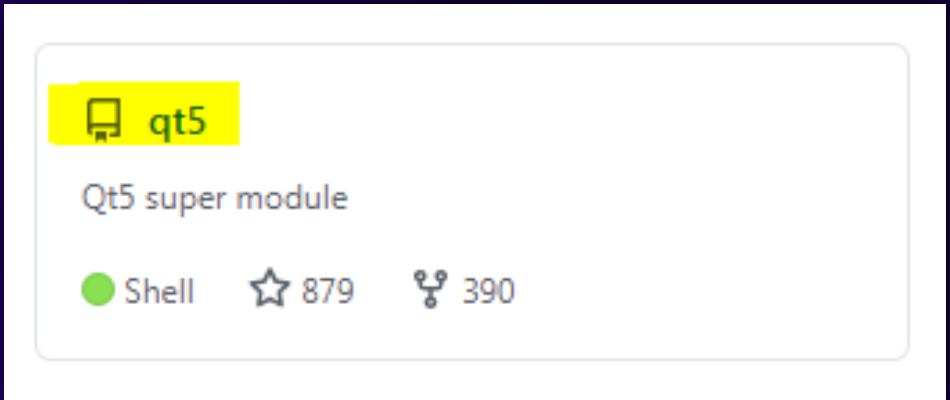
## Build in Windows

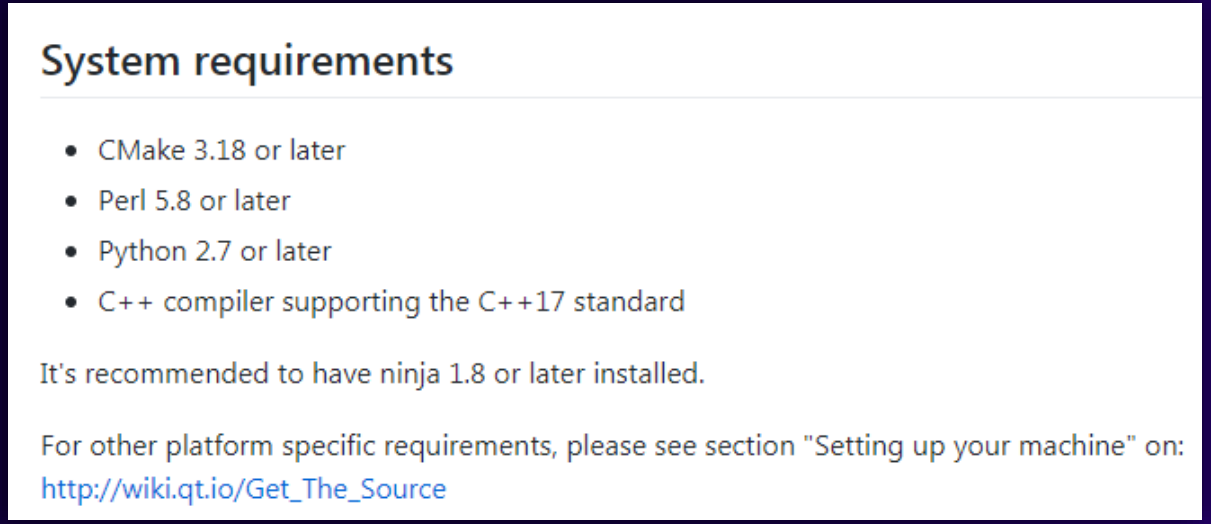
- It can be downloaded from the link below (windows version 5.15.2):
  - URL: https://www.qt.io/offline-installers
- Ability to customize modules
- The ability to change the source code
- Ability to compile statically



## Build in Windows

- Enter the following website (Qt requirements to compile):
  - URL: https://github.com/qt



## Build in Windows

- Prerequisites for compiling Qt on the windows platform

### Windows:

1. Open a command prompt.
2. Ensure that the following tools can be found in the path:

- Supported compiler (Visual Studio 2019 or later, or MinGW-builds gcc 8.1 or later)
- Perl version 5.12 or later [http://www.activestate.com/activeperl/]
- Python version 2.7 or later [http://www.activestate.com/activepython/]
- Ruby version 1.9.3 or later [http://rubyinstaller.org/]

```
cd <path>\<source_package>
configure -prefix %CD%\qtbase
cmake --build .
```

## Build in Windows

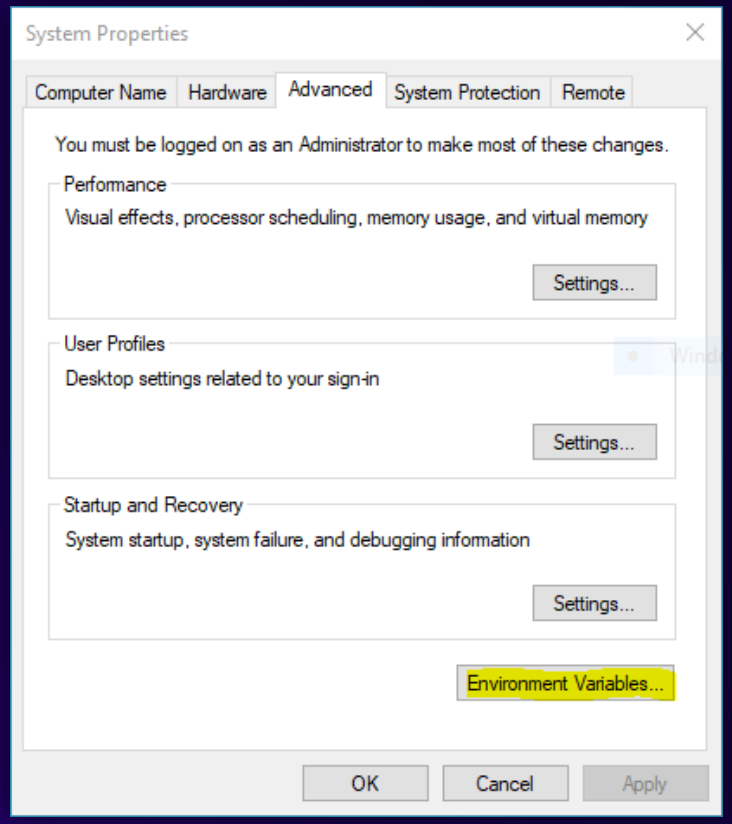
- Compiler:
  - Visual Studio 2019 or later
  - MinGW-builds gcc 8.1 or later
- Tools:
  - Perl version 5.12 or later [http://www.activestate.com/activeperl/]
  - Python version 2.7 or later [http://www.activestate.com/activepython/]
  - Ruby version 1.9.3 or later [http://rubyinstaller.org/]

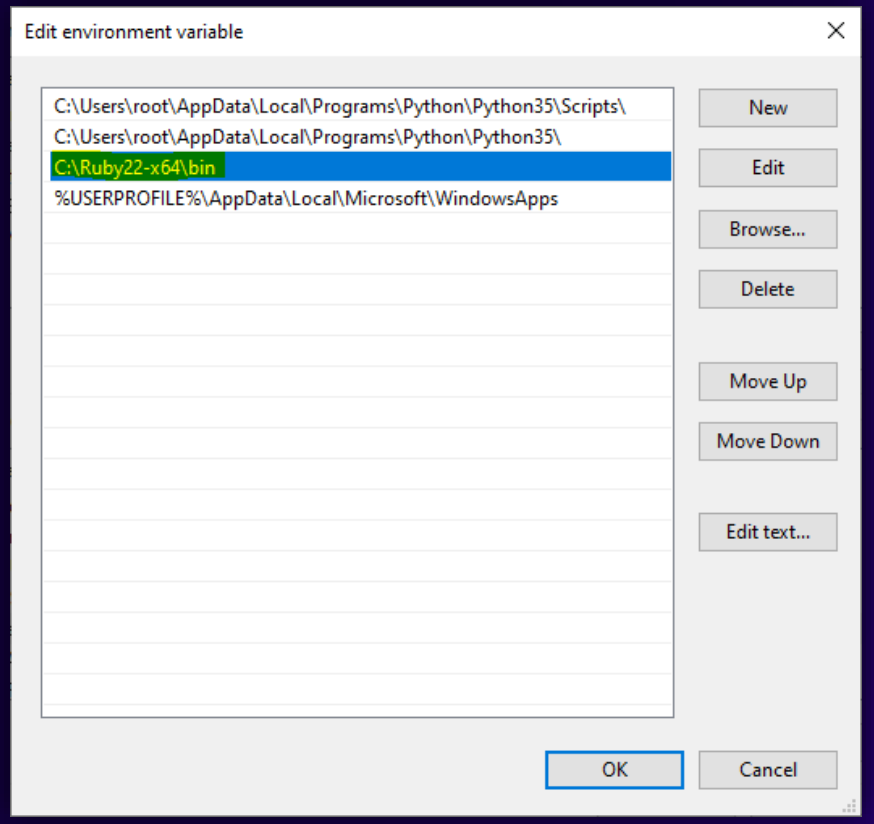
Note: “Python” and “Perl” installation requires internet.

Note: Compilation requires Windows version 7 or higher.

## Build in Windows

- After installing the tools (Perl, Python, Ruby), their path should be added to the “System Enviroment”



## Build in Windows

- First, download the source code
  - Example: qt-everywhere-src-5.15.2.zip
- Decompress the source code in the desired path, for example "c:\"

## Build in Windows >> Compile with MSVC

- Install Visual Studio 2019
- Depending on the required architecture (32-bit or 64-bit version), we run the Visual Studio command line from start menu:
  - x86 Native Tools Command Prompt for VS 2019
  - x64 Native Tools Command Prompt for VS 2019
- Enter the following commands in the command line:

```
> SET _ROOT=C:\Qt\qt-everywhere-src-5.15.2
> SET PATH=%_ROOT%\qtbase\bin;%_ROOT%\gnuwin32\bin;%PATH%
> SET PATH=%_ROOT%\qtrepotools\bin;%PATH%
```

## Build in Windows >> Compile with MSVC

- Enter the Qt source code path

```
> Cd C:\Qt\qt-everywhere-src-5.15.2
```

- Configure the compiler

```
> configure -debug-and-release -platform win32-msvc -developer-build
  -prefix "C:\Qt\5.15.2-x86" -nomake examples -nomake tests
  -skip qtwebengine -opensource -mp
```

- Start the compilation with the following command:
  - nmake
- install Qt with the following command
  - nmake install
- Clean source code
  - nmake clean

## Build in Windows >> Compile with MinGW

- Install MinGW 8.1 or later
  - URL: http://mingw-w64.org/
- Run mingw command line or run it in the "cmd" and set mingw path in system environment (Path variable).
- Enter the following commands in the command line:

```
> SET _ROOT=C:\Qt\qt-everywhere-src-5.15.2
> SET PATH=%_ROOT%\qtbase\bin;%_ROOT%\gnuwin32\bin;%PATH%
> SET PATH=%_ROOT%\qtrepotools\bin;%PATH%
```

- Enter the Qt source code path

```
> Cd C:\Qt\qt-everywhere-src-5.15.2
```

## Build in Windows >> Compile with MinGW

- Configure the compiler

```
> configure -debug-and-release -platform win32-g++ -developer-build
  -prefix "C:\Qt\5.15.2-mingw-x86" -nomake examples -nomake tests
  -skip qtwebengine -opensource -no-angle -mp
```

- Start the compilation with the following command:

```
> mingw32-make -jn (n is number of cpu for parallel build)
```

- Install Qt with the following command

```
> mingw32-make install
```

## Build in Windows

- Clean source code

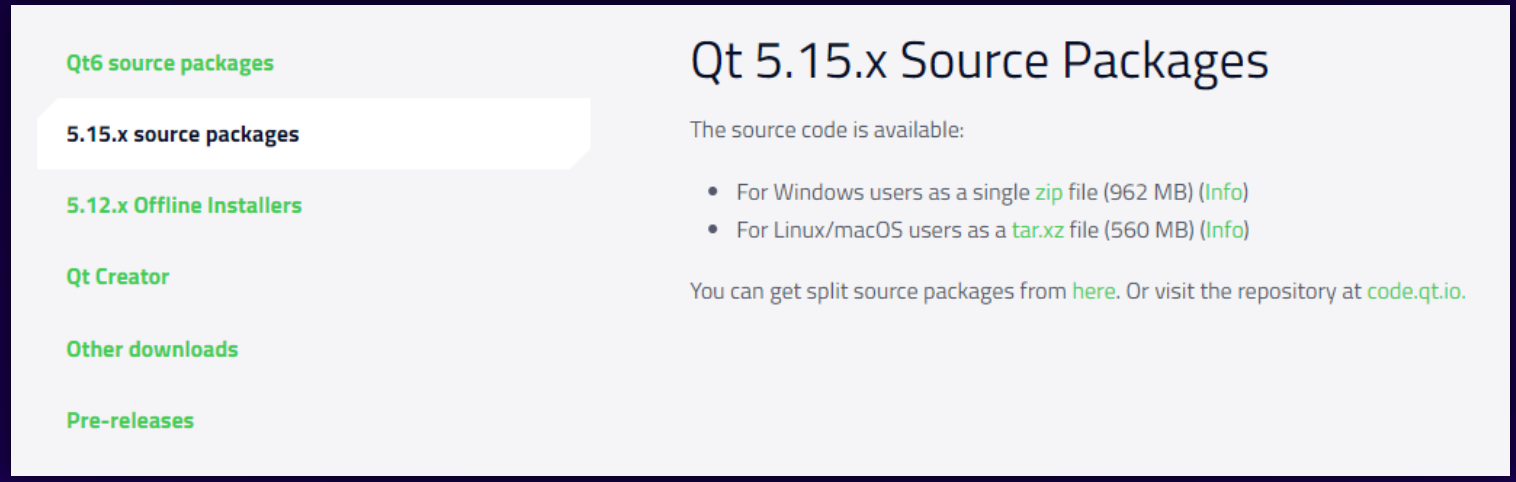
```
> mingw32-make clean
```

- Remove source code directory

```
> del /q "C:\Qt\qt-everywhere-src-5.15.2\*"
FOR /D %%p IN ("C:\Qt\qt-everywhere-src-5.15.2\*.*") DO rmdir "%%p" /s /q
```

## Build in Linux

- It can be downloaded from the link below (linux/mac version 5.15.2):
  - URL: https://www.qt.io/offline-installers
- Ability to customize modules
- The ability to change the source code
- Ability to compile statically



## Build in Linux

- Compiler:
  - GCC/g++
  - Clang
- Tools and library:
  - Perl (>=5.14)
  - Python (>=2.6.x)
  - build-essential
  - Libxcb

## Build in Linux / Dependency Packages

- Build Essentials

| Distribution | Packages |
|---|---|
| Ubuntu and/or Debian: | `$ sudo apt-get install build-essential perl python3 git` |
| Fedora 30: | `$ su - -c "dnf install perl-version git gcc-c++ compat-openssl10-devel harfbuzz-devel double-conversion-devel libzstd-devel at-spi2-atk-devel dbus-devel mesa-libGL-devel"` |
| OpenSUSE: | `$ sudo zypper install git-core gcc-c++ make` |

## Build in Linux / Dependency Packages

- Libxcb

| Distribution | Packages |
|---|---|
| Ubuntu and/or Debian: | `$ sudo apt-get install '^libxcb.*-dev' libx11-xcb-dev libglu1-mesa-dev libxrender-dev libxi-dev libxkbcommon-dev libxkbcommon-x11-dev` |
| Fedora 30: | `$ su - -c "dnf install libxcb libxcb-devel xcb-util xcb-util-devel xcb-util-*-devel libX11-devel libXrender-devel libxkbcommon-devel libxkbcommon-x11-devel libXi-devel libdrm-devel libXcursor-devel libXcomposite-devel"` |
| OpenSUSE 12+: | `$ sudo zypper in xorg-x11-libxcb-devel xcb-util-devel xcb-util-image-devel xcb-util-keysyms-devel xcb-util-renderutil-devel xcb-util-wm-devel xorg-x11-devel libxkbcommon-x11-devel libxkbcommon-devel libXi-devel` |
| Centos 7: | `$ yum install libxcb libxcb-devel xcb-util xcb-util-devel mesa-libGL-devel libxkbcommon-devel` |

## Build in Linux / Dependency Packages

- OpenGL support
  - For Qt Quick 2, a graphics driver with native OpenGL 2.0 support is highly recommended.
- Accessibility
  - It is recommended to build with accessibility enabled, install libatspi2.0-dev and libdbus-1-dev packages.

## Build in Linux / Dependency Packages

- Qt WebKit

| Distribution | Packages |
|---|---|
| Ubuntu and/or Debian: | `$ sudo apt-get install flex bison gperf libicu-dev libxslt-dev ruby` |
| Fedora 30: | `$ su - -c "dnf install flex bison gperf libicu-devel libxslt-devel ruby"` |
| OpenSUSE: | `$ sudo zypper install flex bison gperf libicu-devel ruby` |
| Mandriva/ROSA/Unity: | `$ urpmi gperf` |

## Build in Linux / Dependency Packages

- Qt WebEngine

| Distribution | Packages |
|---|---|
| Ubuntu and/or Debian: | `$ sudo apt-get install libxcursor-dev libxcomposite-dev libxdamage-dev libxrandr-dev libxtst-dev libxss-dev libdbus-1-dev libevent-dev libfontconfig1-dev libcap-dev libpulse-dev libudev-dev libpci-dev libnss3-dev libasound2-dev libegl1-mesa-dev gperf bison nodejs` |
| Fedora/RHEL: | `$ sudo dnf install freetype-devel fontconfig-devel pciutils-devel nss-devel nspr-devel ninja-build gperf cups-devel pulseaudio-libs-devel libcap-devel alsa-lib-devel bison libXrandr-devel libXcomposite-devel libXcursor-devel libXtst-devel dbus-devel fontconfig-devel alsa-lib-devel rh-nodejs12-nodejs rh-nodejs12-nodejs-devel` |
| OpenSUSE: | `$ sudo zypper install alsa-devel dbus-1-devel libXcomposite-devel libXcursor-devel libXrandr-devel libXtst-devel mozilla-nspr-devel mozilla-nss-devel gperf bison nodejs10 nodejs10-devel` |

## Build in Linux / Dependency Packages

- Qt Multimedia
  - You'll need at least alsa-lib (>= 1.0.15) and gstreamer (>=0.10.24) with the base-plugins package.

| Distribution | Packages |
|---|---|
| Ubuntu and/or Debian: | `$ sudo apt-get install libasound2-dev libgstreamer1.0-dev libgstreamer-plugins-base1.0-dev libgstreamer-plugins-good1.0-dev libgstreamer-plugins-bad1.0-dev` |
| Fedora 30: | `$ dnf install pulseaudio-libs-devel alsa-lib-devel gstreamer1-devel gstreamer1-plugins-base-devel wayland-devel` |

## Build in Linux / Dependency Packages

- QDoc Documentation Generator Tool

| Distribution | Packages |
| --- | --- |
| Ubuntu and/or Debian: | `$ sudo apt install clang libclang-dev` |
| Fedora 30: | `$ su -c 'dnf install llvm-devel'` |

## Build in Linux

- Enter the following website (Qt requirements to compile):
  - URL: https://doc.qt.io/qt-5/linux-building.html
  - URL: https://wiki.qt.io/Building_Qt_5_from_Git#Linux.2FX11
- First, download the source code
  - Example: qt-everywhere-src-5.15.2.tar.xz
- Installing the License File (Commercially Licensed Qt Only)
- Decompress the source code in the desired path, for example “/tmp“

```
$ cd /tmp
$ tar -xvf qt-everywhere-src-5.15.2.tar.xz
Or
$ cd /tmp
$ gunzip qt-everywhere-opensource-src-%VERSION%.tar.gz   # uncompress the archive
$ tar xvf qt-everywhere-opensource-src-%VERSION%.tar     # unpack it
```

## Build in Linux

- Installing the license file (Commercially licensed Qt only)
- Enter the Qt source code path

```
$ cd /tmp/qt-everywhere-opensource-src-%VERSION%
```

- Compile with GCC/g++
  - Configure the compiler

```
$ ./configure -platform linux-g++ -developer-build -prefix
"/opt/qt5.15.2-x64" -nomake examples -nomake tests -skip qtwebengine
-opensource -mp
```

  - Start the compilation with the following command:

```
$ gmake
```

## Build in Linux

- Compile with GCC/g++
  - Compile Qt Docs

```
$ make docs
```

  - Install Qt Docs

```
$ sudo gmake install_docs    # Need to root privileges
```

  - Remove source code directory

```
$ rm -rf /tmp/*
```

## Build in Linux

- Compile with GCC/g++
  - Install Qt with the following command

```
$ sudo gmake install   # Need to root privileges
```

  - Clean source code

```
$ gmake clean
```

WARNING: -debug-and-release is only supported on Darwin and Windows platforms. Qt can be built in release mode with separate debug information, so -debug-and-release is no longer necessary.

## Build Options

- Compiler Options: Compile with GCC/g++ (32 bit)

```
-platform linux-g++-32
```

- Compiler Options: Compile with Clang

```
-platform linux-clang
```

- Compiler Options: Cross-Compilation Options. To configure Qt for cross-platform development and deployment, the development toolchain for the target platform needs to be set up. This set up varies among the Supported Platforms.

```
-xplatform
```

## Build Options

- Install Directories

```
-prefix /opt/Qt
```

- Excluding Qt Modules. Configure's -skip option allows top-level source directories to be excluded from the Qt build.

```
./configure -skip qtconnectivity
```

- Including or Excluding Features. The -feature-<feature> and -no-feature-<feature> options include and exclude specific features, respectively.

```
./configure -no-feature-accessibility
```

## Build Options

- Third-Party Libraries. The Qt source packages include third-party libraries. To set whether Qt should use the system's versions of the libraries or to use the bundled version, pass either -system or -qt before the name of the library to configure.

```
./configure -no-zlib -qt-libjpeg -qt-libpng -system-xcb
```

| Library Name | Bundled in Qt | Installed in System |
|---|---|---|
| zlib | -qt-zlib | -system-zlib |
| libjpeg | -qt-libjpeg | -system-libjpeg |
| libpng | -qt-libpng | -system-libpng |
| freetype | -qt-freetype | -system-freetype |
| PCRE | -qt-pcre | -system-pcre |
| HarfBuzz-NG | -qt-harfbuzz | -system-harfbuzz |

## Build Options

- OpenGL Options for Windows
  - Dynamic: With the dynamic option, Qt will try to use native OpenGL first. If that fails, it will fall back to ANGLE and finally to software rendering in case of ANGLE failing as well.

```
configure.bat -opengl dynamic
```

  - Desktop: With the desktop option, Qt uses the OpenGL installed on Windows, requiring that the OpenGL in the target Windows machine is compatible with the application. The -opengl option accepts two versions of OpenGL ES, es2 for OpenGL ES 2.0 or es1 for OpenGL ES Common Profile.

```
configure.bat -opengl desktop
```

  - You can also use -opengl dynamic, which enable applications to dynamically switch between the available options at runtime. For more details about the benefits of using dynamic GL-switching, see Graphics Drivers.

```
configure.bat -opengl es2
```

Note: For a full list of options, consult the help with configure -help.

## Checking Build and Run Settings

- The Qt Installer attempts to auto-detect the installed compilers and Qt versions. If it succeeds, the relevant kits will automatically become available in Qt Creator.

## Adding Kits

- Qt Creator groups settings used for building and running projects as kits to make cross-platform and cross-configuration development easier. Each kit consists of a set of values that define one environment, such as a device, compiler, Qt version, and debugger command to use, and some metadata, such as an icon and a name for the kit. Once you have defined kits, you can select them to build and run projects.

## Specifying Kit Settings

- Select Edit > Preferences > Kits > Add
- Specify kit settings. The settings to specify depend on the build system and device type.

## Kit Settings

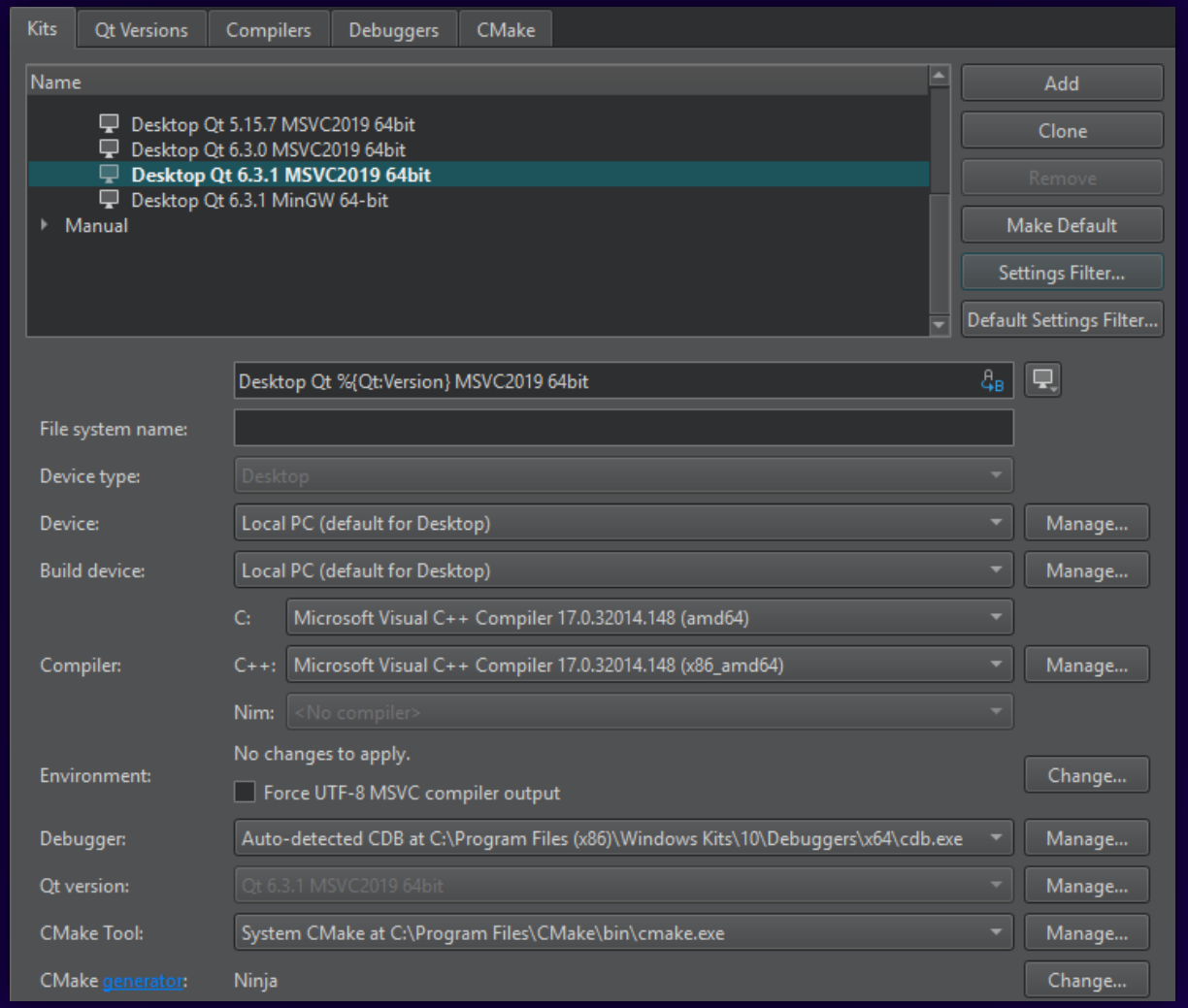
- Name:
  - Name of the kit.
- Device type:
  - The device to build applications on.
- Device:
  - The device to run applications on.
- Compiler:
  - C or C++ compiler that you use to build the project. You can add compilers to the list if they are installed on the development PC.

## Kit Settings

- Debugger:
  - Debugger to debug the project on the target platform. Qt Creator automatically detects available debuggers and displays a suitable debugger in the field. You can add debuggers to the list.
- Qt version
  - Qt version to use for building the project. You can add Qt versions to the list if they are installed on the development PC.

# Introduction to Qt Creator >> Managing Kits

## Kits:



## Adding Qt Versions

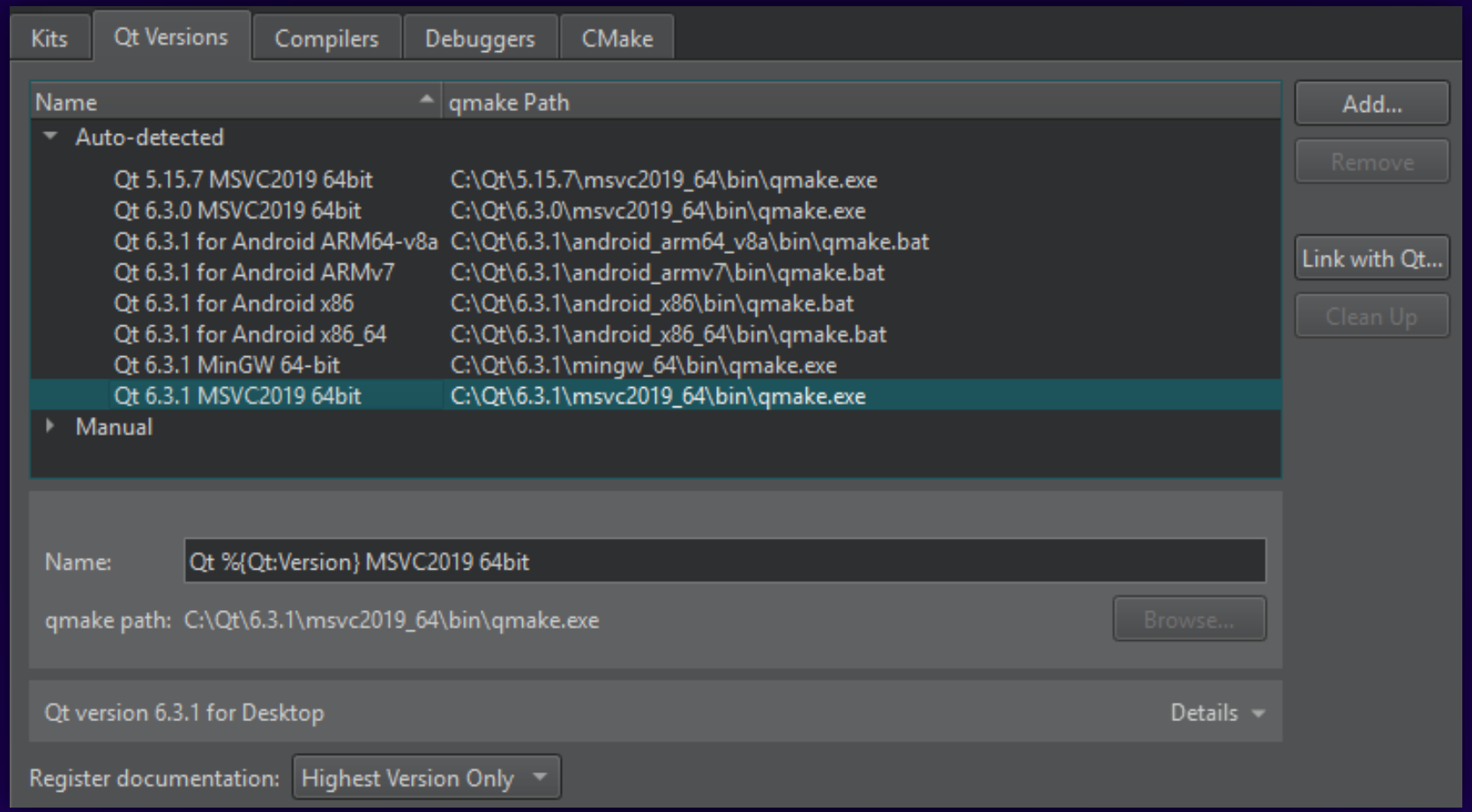
- Qt Creator allows you to have multiple versions of Qt installed on your development PC and use different versions to build your projects.

## Setting Up New Qt Versions

- Select Edit > Preferences > Kits > Qt Versions > Add.
- Select the qmake executable for the Qt version that you want to add.

# Introduction to Qt Creator >> Managing Kits

## Qt Versions:



## Adding Compilers

- Qt is supported on a variety of 32-bit and 64-bit platforms, and can usually be built on each platform with GCC, a vendor-supplied compiler, or a third party compiler.

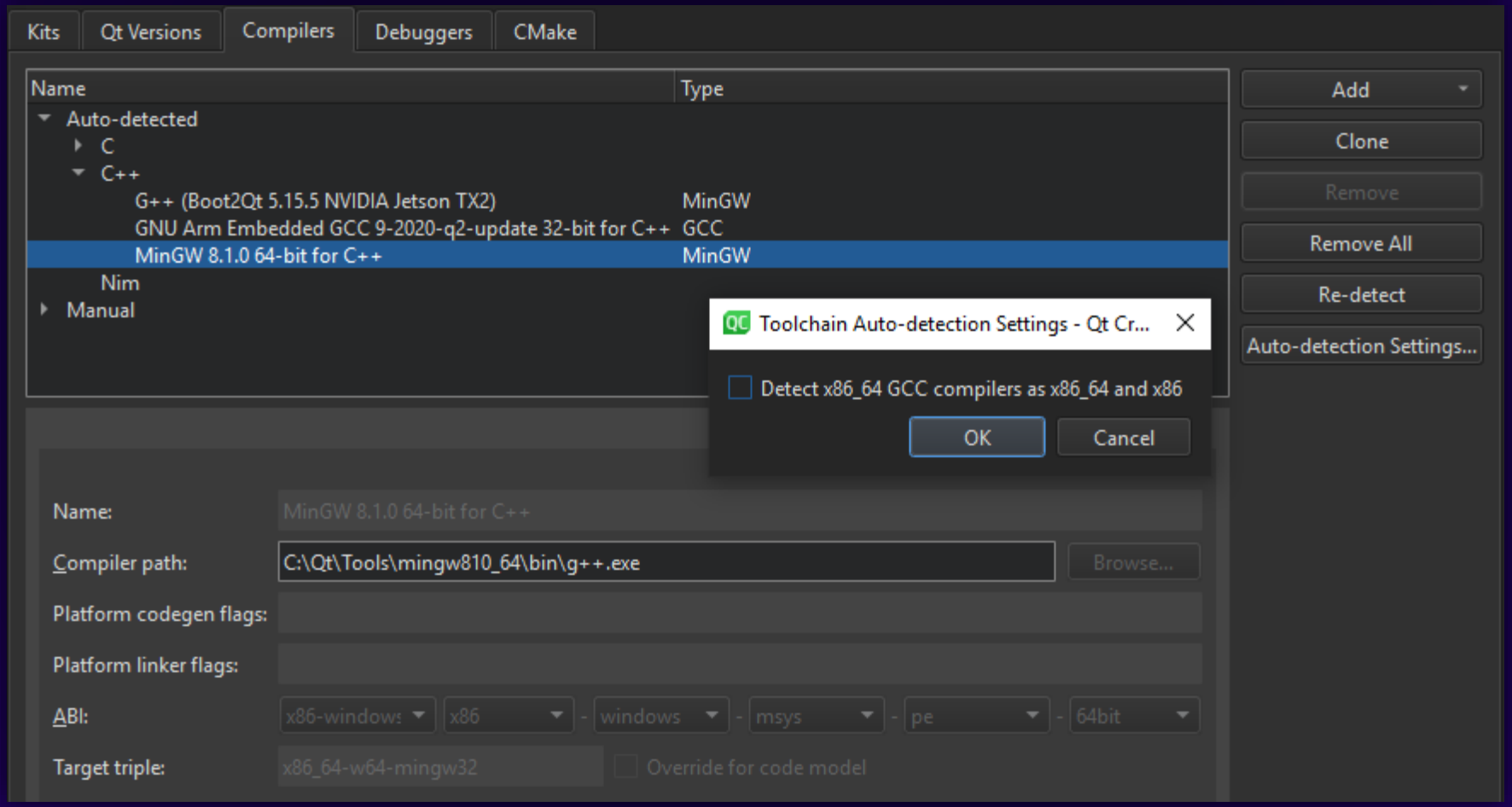
## You can add the following compilers to build applications

- Clang is a C, C++, Objective C, and Objective C++ front-end for the LLVM compiler for Windows, Linux, and macOS.
- clang-cl is an alternative command-line interface to Clang that is compatible with the Visual C++ compiler, cl.exe.
- GNU Compiler Collection (GCC) is a compiler for Linux and macOS.
- ICC (Intel C++ Compiler) is a group of C and C++ compilers. Only the GCC-compatible variant, available for Linux and macOS, is currently supported by Qt Creator.
- MinGW (Minimalist GNU for Windows) is a native software port of GCC and GNU Binutils for use in the development of native Microsoft Windows applications on Windows. MinGW is distributed together with Qt Creator and Qt for Windows.
- MSVC (Microsoft Visual C++ Compiler) is a C++ compiler that is installed with Microsoft Visual Studio.
- Nim is the Nim Compiler for Windows, Linux, and macOS.
- QCC is the interface for compiling C++ applications for QNX.

Note: MSVC compiler exists in Windows Software Development Kit (SDK)

# Introduction to Qt Creator >> Managing Kits

## Qt Compilers:



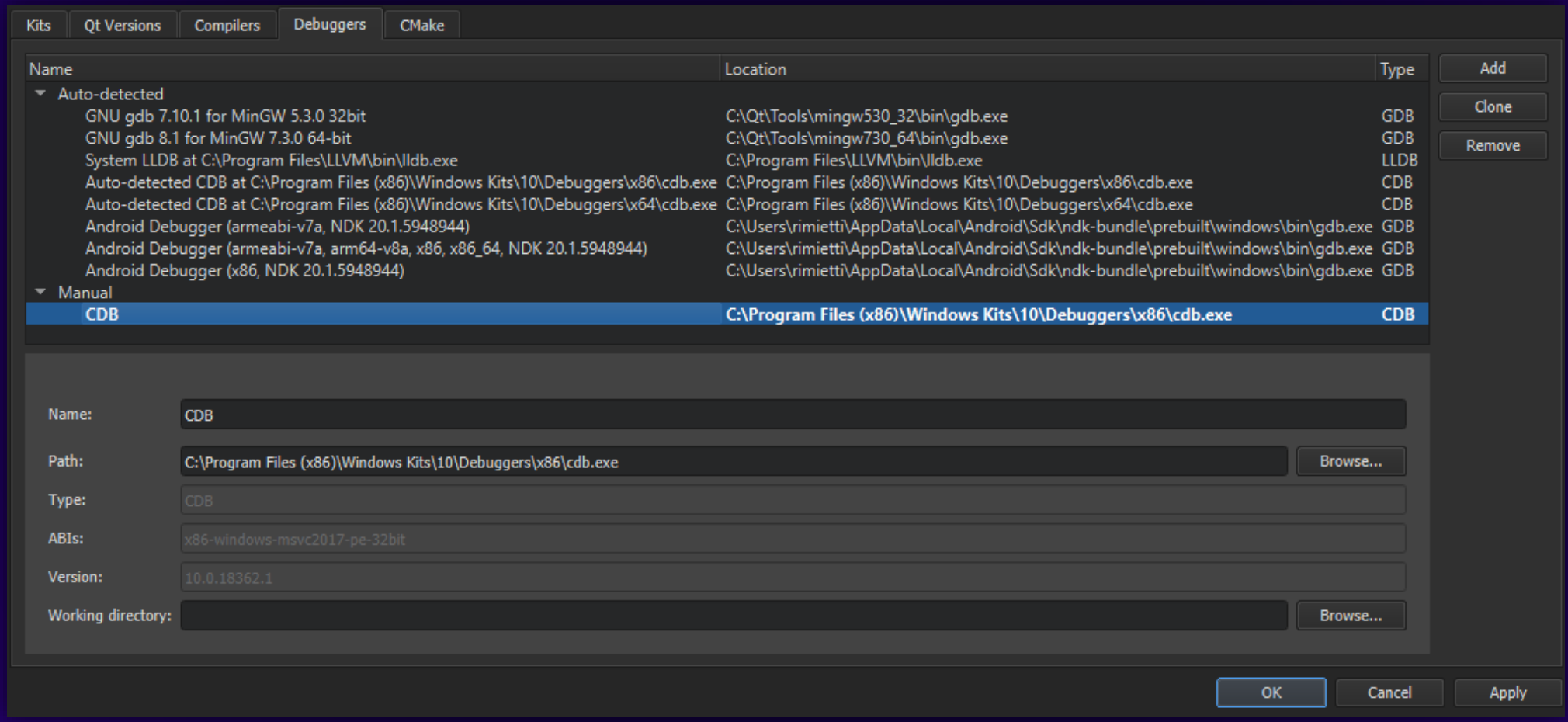
## Adding Debuggers

- The Qt Creator debugger plugin acts as an interface between the Qt Creator core and external native debuggers such as the GNU Symbolic Debugger (GDB), the Microsoft Console Debugger (CDB), a QML/JavaScript debugger, and the debugger of the low level virtual machine (LLVM) project, LLDB.
- Select Edit > Preferences > Kits > Debuggers > Add.
- In the Path field, specify the path to the debugger binary:
  - For CDB (Windows only), specify the path to the Windows Console Debugger executable.
  - For GDB, specify the path to the GDB executable. The executable must be built with Python scripting support enabled.
  - For LLDB (experimental), specify the path to the LLDB executable.

Note: CDB compiler exists in Windows Software Development Kit (SDK)

# Introduction to Qt Creator >> Managing Kits

## Qt Debuggers:



## Concepts

- Shadow building means building a project in a separate directory, the build directory. The build directory is different from the source directory. One of the benefits of shadow building is that it keeps your source directory clean, which makes it faster to switch between build configurations.
- Abi
- llvm

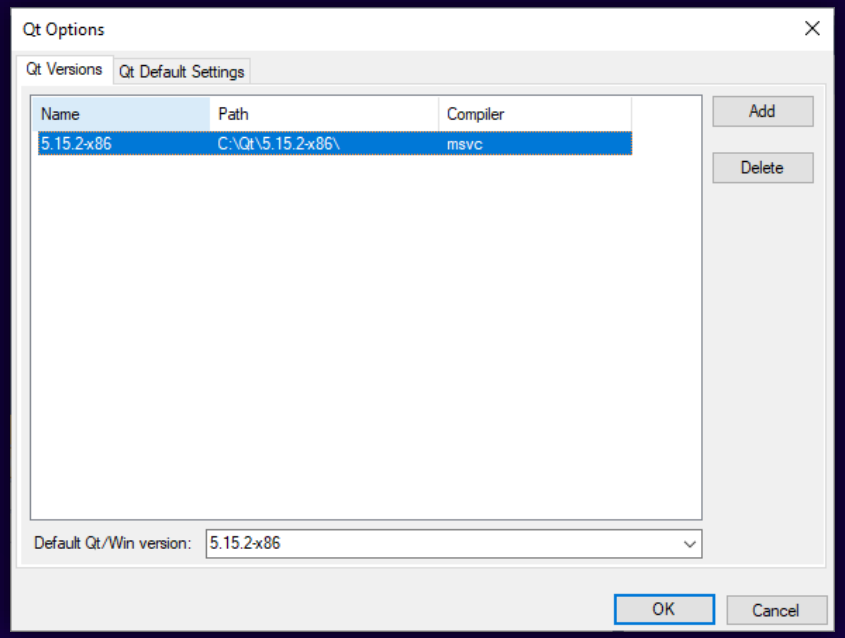
# Introduction to Visual Studio >> Manage Kits

## Prerequisite

- Install Visual Studio (Example: Visual Studio 2019)
- Install Visual Studio addin (Example: qt-vsaddin-msvc2019-2.6.0-rev.07.vsix)

## Adding Kits

- Select Extensions > Qt VS Tools > Qt Options > Add (qmake file from Compiled Qt for MSVC)



# Cross-Platform Development

## C Console Application

- Create the helloworld.c program using a Vim editor as shown below.

```
/* Hello World C Program */

#include <stdio.h>

int main()
{
        printf("Hello World!\n");
        return 0;
}
```

- Compile the helloworld.c Program

```
$ gcc main.cpp
```

- Execute the C Program (a.out)

```
$ ./a.out
```

# Cross-Platform Development


## C++ Console Application

- Create a file titled main.cpp somewhere on your computer. Within the file, include the following:

```
#include <iostream>

int main() {
        std::cout << "Hello World\n";
        return 0;
}
```

- To compile our application we can execute the following from the command line:

```
$ g++ main.cpp -o app
```

- The app file can be ran by executing the following:

```
./app
```

# Cross-Platform Development


## Qt Console Application

- Let's begin with a simple C++ program. Open a text editor and enter the following source code. Create a directory "hello" and save the source code into a file hello.cpp residing in this directory.

```
#include <QTextStream>
int main() {
        QTextStream(stdout) << "Hello, world!" << endl;
        return 0;
}
```

- Now enter the "hello" directory and type: "qmake -project". This will create an qmake project file. Thereafter run just "qmake" without arguments. this will create a "Makefile" which contains the rules to build your application. Then run "make" (called nmake or gmake on some platforms), which will build your app according to the rules layed out in the "Makefile". Finally you can run the app "./hello".

```
$ cd my/dir/hello
$ qmake -project
$ qmake CONFIG+=debug|release ${qmake_options}
$ make ${make_options}
$ ./hello | ./hello &
$ make clean
```

g++

```
> cd my\dir\hello
> qmake -project
> qmake
> nmake -f Makefile.Release | nmake -f Makefile.Debug
> hello.exe | start /wait hello.exe
> nmake clean
```

MSVC

# Cross-Platform Development


## Qt Console Application

- Add this line in *.pro file

```
QT -= gui

CONFIG += c++11 console
```

- Show exit code:

```
$ echo $?
```
Linux

```
> echo %errorlevel%
```
Windows

# Cross-Platform Development


## Qt Desktop Application

- Simply replace the source code in hello.cpp by:

```
#include <QtGui>
#include <QApplication>
#include <QLabel>

int main(int argc, char **argv) {
 QApplication app(argc, argv);
 QLabel label("Hello, world!");
 label.show();
 return app.exec();
}
```

- Add the following lines to the .pro file after the include path:

```
...
INCLUDEPATH += .

QT += gui
QT += widgets
...
```

- Run "qmake" and then "make" again. If you launch the application should see a small window saying "Hello, world!".

# Cross-Platform Development >> qmake

## Running qmake

- The behavior of qmake can be customized when it is run by specifying various options on the command line.

## Command Syntax

- The syntax used to run qmake takes the following simple form:

```
qmake [mode] [options] files
```

Note: If you installed Qt via a package manager, the binary may be qmake6.

## Operating Modes

- qmake supports two different modes of operation. In the default mode, qmake uses the information in a project file to generate a Makefile, but it is also possible to use qmake to generate project files.
  - -makefile: qmake output will be a Makefile.
  - -project: qmake output will be a project file.

# Cross-Platform Development >> qmake

## Files

- The files argument represents a list of one or more project files, separated by spaces.

## General Options

- -help

  qmake will go over these features and give some useful help.

- -o file

  qmake output will be directed to file. If this option is not specified, qmake will try to use a suitable file name for its output, depending on the mode it is running in.

  If '-' is specified, output is directed to stdout.

- -d

  qmake will output debugging information. Adding -d more than once increases verbosity.

# Cross-Platform Development >> qmake

- -t tmpl

  qmake will override any set TEMPLATE variables with tmpl, but only after the .pro file has been processed.
- -tp prefix

  qmake will add prefix to the TEMPLATE variable.
- -Wall

  qmake will report all known warnings.
- -Wnone

  No warning information will be generated by qmake.
- -Wparser

  qmake will only generate parser warnings. This will alert you to common pitfalls and potential problems in the parsing of your project files.
- -Wlogic

  qmake will warn of common pitfalls and potential problems in your project file. For example, qmake will report multiple occurrences of files in lists and missing files.

# Cross-Platform Development >> qmake

## Makefile Mode Options

- In Makefile mode, qmake will generate a Makefile that is used to build the project.

```
qmake -makefile [options] files
```

- -after

  qmake will process assignments given on the command line after the specified files.
- -nocache

  qmake will ignore the .qmake.cache file.
- -nodepend

  qmake will not generate any dependency information.
- -cache file

  qmake will use file as the cache file, ignoring any other .qmake.cache files found.
- -spec spec

  qmake will use spec as a path to platform and compiler information, and ignore the value of QMAKESPEC.

Note: You may also pass qmake assignments on the command line. They are processed before all of the files specified. For example, the following command generates a Makefile from test.pro:

```
qmake -makefile -o Makefile "CONFIG+=test" test.pro
qmake "CONFIG+=test" test.pro
```

## Project Mode Options

- In project mode, qmake will generate a project file. Additionally, you may supply the following options in this mode:

```
qmake -project [options] files
```

- -r

  qmake will look through supplied directories recursively.
- -nopwd

  qmake will not look in your current working directory for source code. It will only use the specified files.

# Cross-Platform Development >> .pro file

## Project File Elements

- Project files contain all the information required by qmake to build your application, library, or plugin.

## Variables

- In a project file, variables are used to hold lists of strings. In the simplest projects, these variables inform qmake about the configuration options to use, or supply filenames and paths to use in the build process.

| Variable | Contents |
|---|---|
| CONFIG | General project configuration options. |
| DESTDIR | The directory in which the executable or binary file will be placed. |
| FORMS | A list of UI files to be processed by the user interface compiler (uic). |
| HEADERS | A list of filenames of header (.h) files used when building the project. |
| QT | A list of Qt modules used in the project. |
| RESOURCES | A list of resource (.qrc) files to be included in the final project. See the The Qt Resource System for more information about these files. |
| SOURCES | A list of source code files to be used when building the project. |
| TEMPLATE | The template to use for the project. This determines whether the output of the build process will be an application, a library, or a plugin. |

# Cross-Platform Development >> .pro file

## Variables

- The following snippet illustrates how lists of values are assigned to variables:

```
HEADERS = mainwindow.h paintwidget.h
```

- The list of values in a variable is extended in the following way:

```
SOURCES = main.cpp mainwindow.cpp \
          paintwidget.cpp
CONFIG += console
```

- The contents of a variable can be read by prepending the variable name with $$. This can be used to assign the contents of one variable to another:

```
TEMP_SOURCES = $$SOURCES
```

# Cross-Platform Development >> .pro file

## Whitespace

- Usually, whitespace separates values in variable assignments. To specify values that contain spaces, you must enclose the values in double quotes:

```
DEST = "Program Files"
```

## Comments

- You can add comments to project files. Comments begin with the # character and continue to the end of the same line. For example:

```
# Comments usually start at the beginning of a line, but they
# can also follow other content on the same line.
```

Note: To include the # character in variable assignments, it is necessary to use the contents of the built-in LITERAL_HASH variable.

# Cross-Platform Development >> .pro file

## Built-in Functions and Control Flow

- qmake provides a number of built-in functions to enable the contents of variables to be processed. The most commonly used function in simple project files is the include() function which takes a filename as an argument.

```
include(other.pro)
```

- Support for conditional structures is made available via scopes that behave like if statements in programming languages:

```
win32 {
    SOURCES += paintwidget_win.cpp
}
```

# Cross-Platform Development >> .pro file

## Project Templates

- The TEMPLATE variable is used to define the type of project that will be built.

| Template | qmake Output |
|---|---|
| app (default) | Makefile to build an application. |
| lib | Makefile to build a library. |
| subdirs | Makefile containing rules for the subdirectories specified using the SUBDIRS variable. Each subdirectory must contain its own project file. |

## General Configuration

- The CONFIG variable specifies the options and features that the project should be configured with. For example, if your application uses the Qt library and you want to build it in debug mode, your project file will contain the following line:

```
CONFIG += qt debug
```

# Cross-Platform Development >> .pro file

## Declaring Qt Libraries

- If the CONFIG variable contains the qt value, qmake's support for Qt applications is enabled. This makes it possible to fine-tune which of the Qt modules are used by your application. we can enable the XML and network modules in the following way:

```
QT += network xml
```

Note: QT includes the core and gui modules by default, so the above declaration adds the network and XML modules to this default list. The following assignment omits the default modules, and will lead to errors when the application's source code is being compiled:

```
QT = network xml # This will omit the core and gui modules.
```

- If you want to build a project without the gui module, you need to exclude it with the "-=" operator. By default, QT contains both core and gui, so the following line will result in a minimal Qt project being built:

```
QT -= gui # Only the core module is used.
```

# Cross-Platform Development >> .pro file

## Declaring Other Libraries

- The paths that qmake searches for libraries and the specific libraries to link against can be added to the list of values in the LIBS variable. For example, the following lines show how a library can be specified:

```
LIBS += -L/usr/local/lib -lmath
```

- The paths containing header files can also be specified in a similar way using the INCLUDEPATH variable. For example, to add several paths to be searched for header files:

```
INCLUDEPATH = c:/msdev/include d:/stl/include
```

## Volatile

- Volatile keyword indicates that a value may change between different accesses, even if it does not appear to be modified. This keyword prevents an optimizing compiler. Volatile values primarily arise in hardware access (memory-mapped I/O), where reading from or writing to memory is used to communicate with peripheral devices, and in threading, where a different thread may have modified a value.
  - Pointer variable to the register of a peripheral
  - The value of the variable is changed in the service routine of an interrupt
  - In Multithread systems that use two or more threads with this variable
- In this example, the code sets the value stored in foo to 0. It then starts to poll that value repeatedly until it changes to 255:

```
1  static int foo;
2
3  void bar(void) {
4      foo = 0;
5
6      while (foo != 255)
7          ;
8  }
```

## Volatile

- An optimizing compiler will notice that no other code can possibly change the value stored in foo, and will assume that it will remain equal to 0 at all times. The compiler will therefore replace the function body with an infinite loop similar to this:

```
1
2
3
4
5
6
void bar_optimized(void) {
    foo = 0;

    while (true);
}
```

- To prevent the compiler from optimizing code as above, the volatile keyword is used:

```
1
2
3
4
5
6
7
8
static volatile int foo;

void bar(void) {
    foo = 0;

    while (foo != 255);
}
```

## Overloading Function

- C++ lets you specify more than one function of the same name in the same scope. These functions are called overloaded functions, or overloads. Overloaded functions enable you to supply different semantics for a function, depending on the types and number of its arguments.
- Overloading Considerations:

| Function declaration element | Used for overloading |
|---|---|
| Function return type | No |
| Number of arguments | Yes |
| Type of arguments | Yes |
| Presence or absence of ellipsis | Yes |
| Use of **typedef** names | No |
| Unspecified array bounds | No |
| **const** or **volatile** | Yes, when applied to entire function (Must be refrenced) |
| Reference qualifiers (**&** and **&&**) | Yes |

## Overloading Function

- Example 1:

```
1  #include <iostream>
2  using namespace std;
3
4  void add(int a, int b)
5  {
6      cout << "sum = " << (a + b);
7  }
8
9  void add(int a, int b, int c)
10 {
11     cout << endl << "sum = " << (a + b + c);
12 }
13
14 // Driver code
15 int main()
16 {
17     add(10, 2);
18     add(5, 6, 4);
19
20     return 0;
21 }
```

- Output:

```
sum = 12
sum = 15
```

## Overloading Function

- Example 2:

```
1  #include <iostream>
2  using namespace std;
3
4  void print(int i)
5  {
6      cout << " Here is int " << i << endl;
7  }
8  void print(double f)
9  {
10     cout << " Here is float " << f << endl;
11 }
12 void print(char const* c)
13 {
14     cout << " Here is char* " << c << endl;
15 }
16
17 int main() {
18     print(10);
19     print(10.10);
20     print("ten");
21     return 0;
22 }
```

- Output:

```
Here is int 10
Here is float 10.1
Here is char* ten
```

## Optional Arguments/Default Arguments

- Allows a function to be called without providing one or more trailing arguments.

```
1  void f(int a, int b = 2, int c = 3);  // trailing defaults
2  void g(int a = 1, int b = 2, int c);  // error, leading defaults
3  void h(int a, int b = 3, int c);      // error, default in middle
```

- Once a default argument has been given in a declaration or definition, you cannot redefine that argument, even to the same value. However, you can add default arguments not given in previous declarations. For example, the last declaration below attempts to redefine the default values for a and b:

```
1  void f(int a, int b, int c = 1);      // valid
2  void f(int a, int b = 1, int c);      // valid, add another default
3  void f(int a = 1, int b, int c);      // valid, add another default
4  void f(int a = 1, int b = 1, int c = 1); // error, redefined defaults
```


## Optional Arguments/Default Arguments

- Example:

```cpp
#include <iostream>

void f(int a, int b, int c = 1);      // valid
void f(int a, int b = 1, int c);      // valid, add another default
void f(int a = 1, int b, int c);      // valid, add another default
void f(int a, int b, int c)
{
    std::cout << a << ", " << b << ", " << c << std::endl;
}

int main()
{
    f();
    f(8);
    f(8, 9);
    f(8, 9, 10);

    return 0;
}
```

- Output:

```
1, 1, 1
8, 1, 1
8, 9, 1
8, 9, 10
```

## static_cast

- static_cast is used for cases where you basically want to reverse an implicit conversion, with a few restrictions and additions. static_cast performs no runtime checks. This should be used if you know that you refer to an object of a specific type, and thus a check would be unnecessary. Example:

```
1   void func(void* data) {
2       // Conversion from MyClass* -> void* is implicit
3       MyClass* c = static_cast<MyClass*>(data);
4       ...
5   }
6
7   int main() {
8       MyClass c;
9       start_thread(&func, &c)  // func(&c) will be called
10          .join();
11  }
```

- In this example, you know that you passed a MyClass object, and thus there isn't any need for a runtime check to ensure this.

# Advanced Programming >> Cast

## dynamic_cast

- dynamic_cast is useful when you don't know what the dynamic type of the object is. It returns a null pointer if the object referred to doesn't contain the type casted to as a base class (when you cast to a reference, a **bad_cast** exception is thrown in that case).

```
1  if (JumpStm* j = dynamic_cast<JumpStm*>(&stm)) {
2      ...
3  }
4  else if (ExprStm* e = dynamic_cast<ExprStm*>(&stm)) {
5      ...
6  }
```

- You can not use dynamic_cast for downcast (casting to a derived class) if the argument type is not polymorphic. For example, the following code is not valid, because Base doesn't contain any virtual function:

```
1  struct Base { };
2  struct Derived : Base { };
3  int main() {
4      Derived d; Base* b = &d;
5      dynamic_cast<Derived*>(b); // Invalid
6  }
```

- An "up-cast" (cast to the base class) is always valid with both static_cast and dynamic_cast, and also without any cast, as an "up-cast" is an implicit conversion (assuming the base class is accessible, i.e. it's a public inheritance).

## reinterpret_cast

- To force the pointer conversion, in the same way as the C-style cast does in the background, the reinterpret cast would be used instead.

```
1  char c = 10;        // 1 byte
2  int* p = (int*)&c; // 4 bytes
3  int* q = static_cast<int*>(&c); // compile-time error
4  int* r = reinterpret_cast<int*>(8); // forced conversion
```

## Regular Cast

- Needless to say, this is much more powerful as it combines all of const_cast, static_cast and reinterpret_cast, but it's also unsafe, because it does not use dynamic_cast.

# Advanced Programming >> GTK+

## Remarks

- GTK is a free and open-source cross-platform widget toolkit for creating graphical user interfaces.
- Installation:

```
sudo apt-get install libgtk-3-dev
```

## GtkWidget

- GtkWidget is the base class that all widgets in GTK+ derive from. It manages the widget lifecycle, states, and style.


## gtk_init

- The gtk_init function initializes GTK+ and parses some standard command line options. This function must be called before using any other GTK+ functions.

## gtk_main

- The This code enters the GTK+ main loop. From this point, the application sits and waits for events to happen.

# Advanced Programming >> GTK+


## Sample 1

```c
#include <gtk/gtk.h>

int main(int argc, char* argv[])
{
    gtk_init(&argc, &argv);
    GtkWidget* win = gtk_window_new(GTK_WINDOW_TOPLEVEL);
    g_signal_connect(win, "delete_event", gtk_main_quit, NULL);
    gtk_widget_show(win);
    gtk_main();
    return 0;
}
```

```
$gcc -o appOut main.c `pkg-config --libs --cflags gtk+-3.0`
```

# Advanced Programming >> GTK+


## Sample 2

```
1  #include <gtk/gtk.h>
2
3  void button_clicked(GtkWidget * widget, gpointer data)
4  {
5      g_print("\tButton Clicked - %d was passed.\n", data);
6  }
7
8  int main(int argc, char* argv[])
9  {
10     gtk_init(&argc, &argv);
11
12     GtkWidget* win = gtk_window_new(GTK_WINDOW_TOPLEVEL);
13     gtk_window_set_title(GTK_WINDOW(win), "GTK+ Sample");
14     gtk_window_set_default_size(GTK_WINDOW(win), 230, 150);
15     gtk_window_set_position(GTK_WINDOW(win), GTK_WIN_POS_CENTER);
16     g_signal_connect(win, "delete_event", gtk_main_quit, NULL);
17
18     GtkWidget* btn = gtk_button_new_with_label("Button");
19     gtk_widget_set_tooltip_text(btn, "Button widget");
20     g_signal_connect(btn, "clicked", G_CALLBACK(button_clicked), (gpointer)10);
21     GtkWidget* halign = gtk_alignment_new(0, 0, 0, 0);
22     gtk_container_add(GTK_CONTAINER(halign), btn);
23     gtk_container_add(GTK_CONTAINER(win), halign);
24     gtk_widget_show_all(win);
25     gtk_main();
26
27     return 0;
28 }
```

```
$g++ -o appOut main.cc `pkg-config --libs --cflags gtk+-3.0`
```

# Advanced Programming >> QObject

## Remarks

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

## Description

- Q_OBJECT macro appears in private section of a class. Q_OBJECT 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 Q_OBJECT, it processes it and generates C++ source file containing meta object source code.

## QMetaObject

- QMetaObject class contains meta-information about Qt objects.

## Examples

- Here is the example of class header with Q_OBJECT and signal/slots:

```
1  #include <QObject>
2
3  class MyClass : public QObject
4  {
5      Q_OBJECT
6
7  public:
8  public slots:
9      void setNumber(double number);
10
11 signals:
12     void numberChanged(double number);
13
14 private:
15 };
```

## qobject_cast

- A functionality which is added by deriving from QObject and using the Q_OBJECT macro is the ability to use the qobject_cast. Example:

```
1  class myObject : public QObject
2  {
3      Q_OBJECT
4          //...
5  };
6
7  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.

```
1  myObject* my = qobject_cast<myObject*>(obj);
2
3  if (!myObject)
4  {
5      //wrong type
6  }
```

- 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 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.

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

- To 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;

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

- The same applies for stack variables:

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

Note: 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.

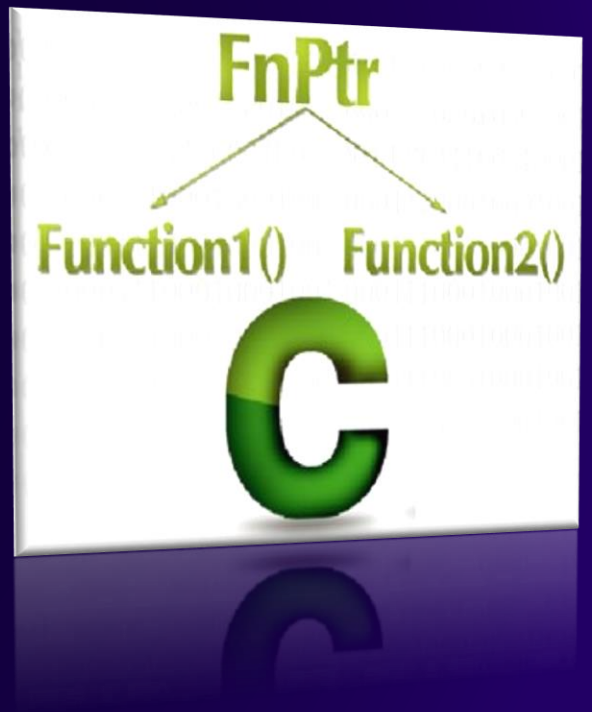
# Advanced Programming >> Signals & Slots

## Remarks

- Signals and slots are used for communication between objects.

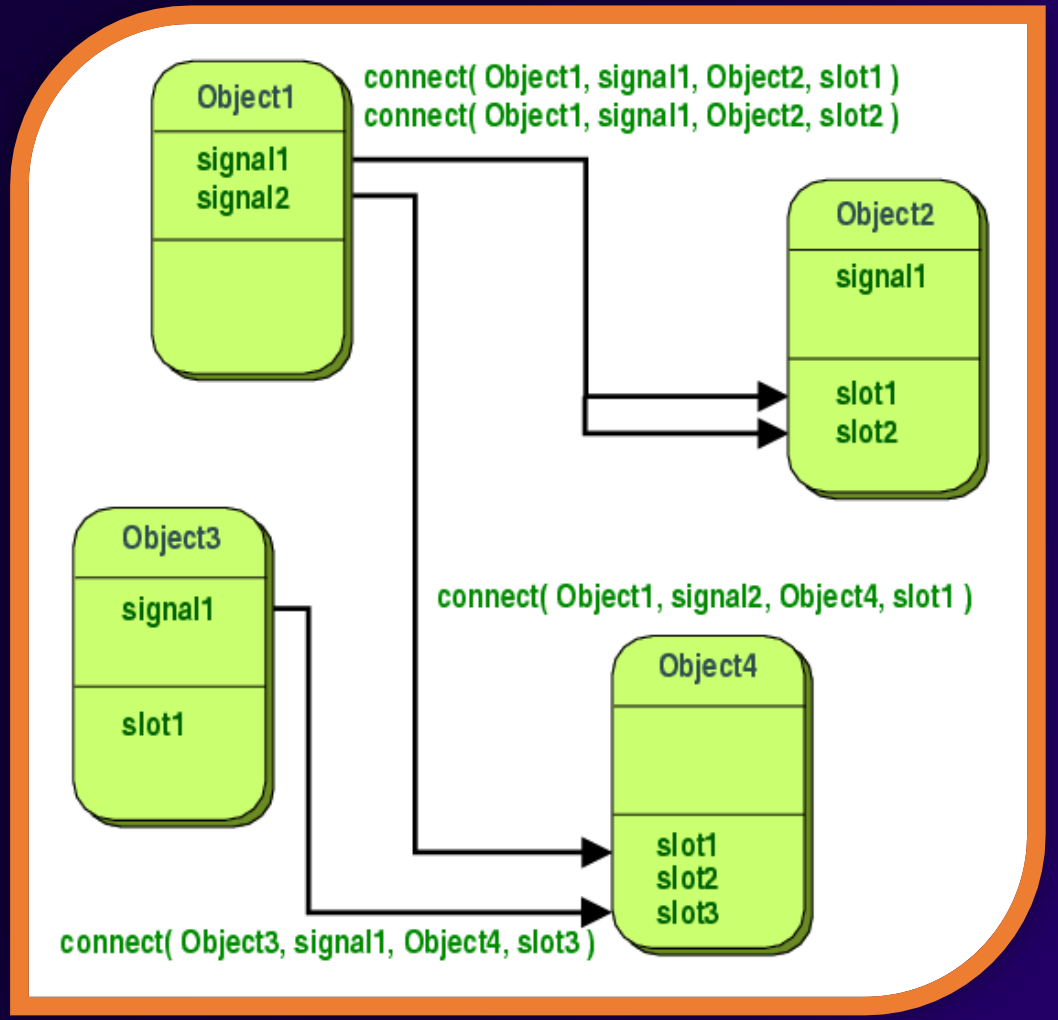
## Introduction

- 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.
- Other toolkits achieve this kind of communication using callbacks. A callback is a pointer to a function, so if you want a processing function to notify you about some event you pass a pointer to another function (the callback) to the processing function. The processing function then calls the callback when appropriate.



## Description

- In Qt, we have an alternative to the callback technique: We use signals and slots. A signal is emitted when a particular event occurs. A slot is a function that is called in response to a particular signal.
- The signals and slots mechanism is type safe: The signature of a signal must match the signature of the receiving slot.



## Signals

- Signals are public access functions and can be emitted from anywhere, but we recommend to only emit them from the class that defines the signal and its subclasses.
- When a signal is emitted, the slots connected to it are usually executed immediately, just like a normal function call. When this happens, the signals and slots mechanism is totally independent of any GUI event loop. Execution of the code following the emit statement will occur once all slots have returned. The situation is slightly different when using queued connections; in such a case, **emit** keyword will continue immediately, and the slots will be executed later.
- A signal can be connected to many slots and signals. Many signals can be connected to one slot.
- If a signal is connected to several slots, the slots are activated in the same order in which the connections were made, when the signal is emitted.

# Advanced Programming >> Signals & Slots

## Slots

- A slot is called when a signal connected to it is emitted. Slots are normal C++ functions and can be called normally; their only special feature is that signals can be connected to them.
- You can also define slots to be virtual, which we have found quite useful in practice.
- Compared to callbacks, signals and slots are slightly slower because of the increased flexibility they provide, although the difference for real applications is insignificant.

## Connect

- Creates a connection of the given type from the signal in the sender object to the method in the receiver object. Returns a handle to the connection that can be used to disconnect it later.

## Disconnect

- Disconnects signal in object sender from method in object receiver. Returns true if the connection is successfully broken; otherwise returns false.


## Connect

- SIGNAL() and SLOT() macros:

```
1  QLabel* label = new QLabel;
2  QScrollBar* scrollBar = new QScrollBar;
3  QObject::connect(scrollBar, SIGNAL(valueChanged(int)), label, SLOT(setNum(int)));
```

- A signal can also be connected to another signal:

```
1  class MyWidget : public QWidget
2  {
3      Q_OBJECT
4  public:
5      MyWidget() {
6          myButton = new QPushButton(this);
7          connect(myButton, SIGNAL(clicked()), this, SIGNAL(buttonClicked()));
8      }
9  signals:
10     void buttonClicked();
11 private:
12     QPushButton* myButton;
13 };
```


## Connect

- Creates a connection of the given type from the signal in the sender object to the method in the receiver object.

```
1  QLabel* label = new QLabel;
2  QLineEdit* lineEdit = new QLineEdit;
3  QObject::connect(lineEdit, &QLineEdit::textChanged, label, &QLabel::setText);
```

- Creates a connection from signal in sender object to functor, and returns a handle to the connection.

```
1  void someFunction();
2  QPushButton* button = new QPushButton;
3  QObject::connect(button, &QPushButton::clicked, someFunction);
```

- You can also connect to functors or C++11 lambdas:

```
1  connect(sender, &QObject::destroyed, this, [=]() { this->m_objects.remove(sender); });
```

## Connection Type

- Qt::AutoConnection:
  - (Default) If the receiver lives in the thread that emits the signal, Qt::DirectConnection is used. Otherwise, Qt::QueuedConnection is used. The connection type is determined when the signal is emitted.
- Qt::DirectConnection:
  - The slot is invoked immediately when the signal is emitted. The slot is executed in the signalling thread.
- Qt::QueuedConnection:
  - The slot is invoked when control returns to the event loop of the receiver's thread. The slot is executed in the receiver's thread.

## Disconnect

- Disconnect everything connected to an object's signals:

```
1  disconnect(myObject, nullptr, nullptr, nullptr);
2  // equivalent to the non - static overloaded function
3  myObject->disconnect();
```

- Disconnect everything connected to a specific signal:

```
1  disconnect(myObject, SIGNAL(mySignal()), nullptr, nullptr);
2  // equivalent to the non-static overloaded function
3  myObject->disconnect(SIGNAL(mySignal()));
```

- Disconnect a specific receiver:

```
1  disconnect(myObject, nullptr, myReceiver, nullptr);
2  // equivalent to the non - static overloaded function
3  myObject->disconnect(myReceiver);
```

- Disconnect a connection from one specific signal to a specific slot:

```
1  QObject::disconnect(lineEdit, &QLineEdit::textChanged, label, &QLabel::setText);
```

## Sample

Note: When x is used, the class should be written in a separate file.

```
#include <QObject>

class Counter : public QObject
{
    Q_OBJECT

public:
    Counter() { m_value = 0; }
    int value() const { return m_value; }

public slots:
    void setValue(int value);

signals:
    void valueChanged(int newValue);

private:
    int m_value;
};
```

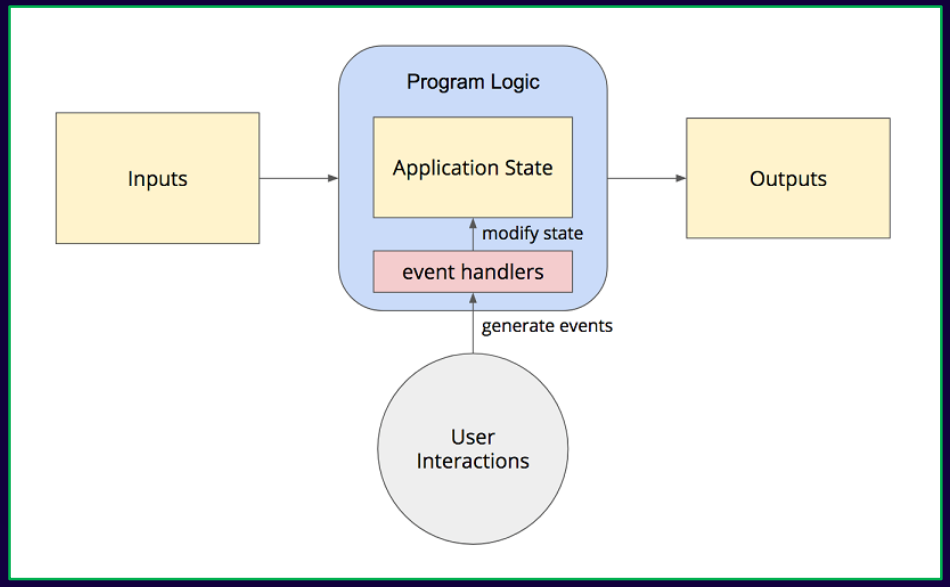
```
void Counter::setValue(int value)
{
    if (value != m_value) {
        m_value = value;
        emit valueChanged(value);
    }
}

Counter a, b;
QObject::connect(&a, &Counter::valueChanged,
                 &b, &Counter::setValue);

a.setValue(12); // a.value() == 12, b.value() == 12
b.setValue(48); // a.value() == 12, b.value() == 48
```

## Application Type

- Procedural
  - The program starts from a point and ends after doing the necessary tasks.
- Interactive (Event based)
  - The program always resides in the memory and waits for the user's commands.

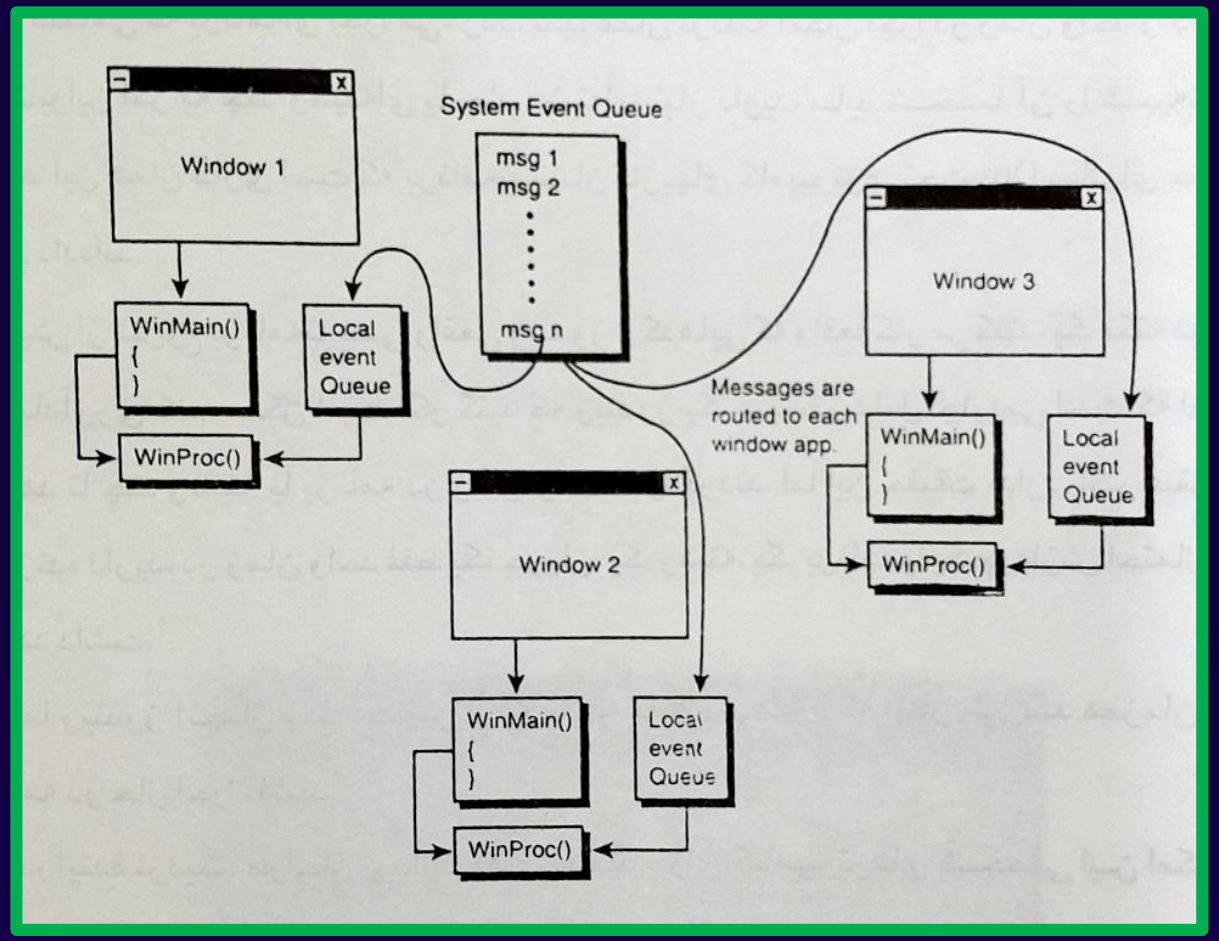
## Thread Type

- Worker
- Ui

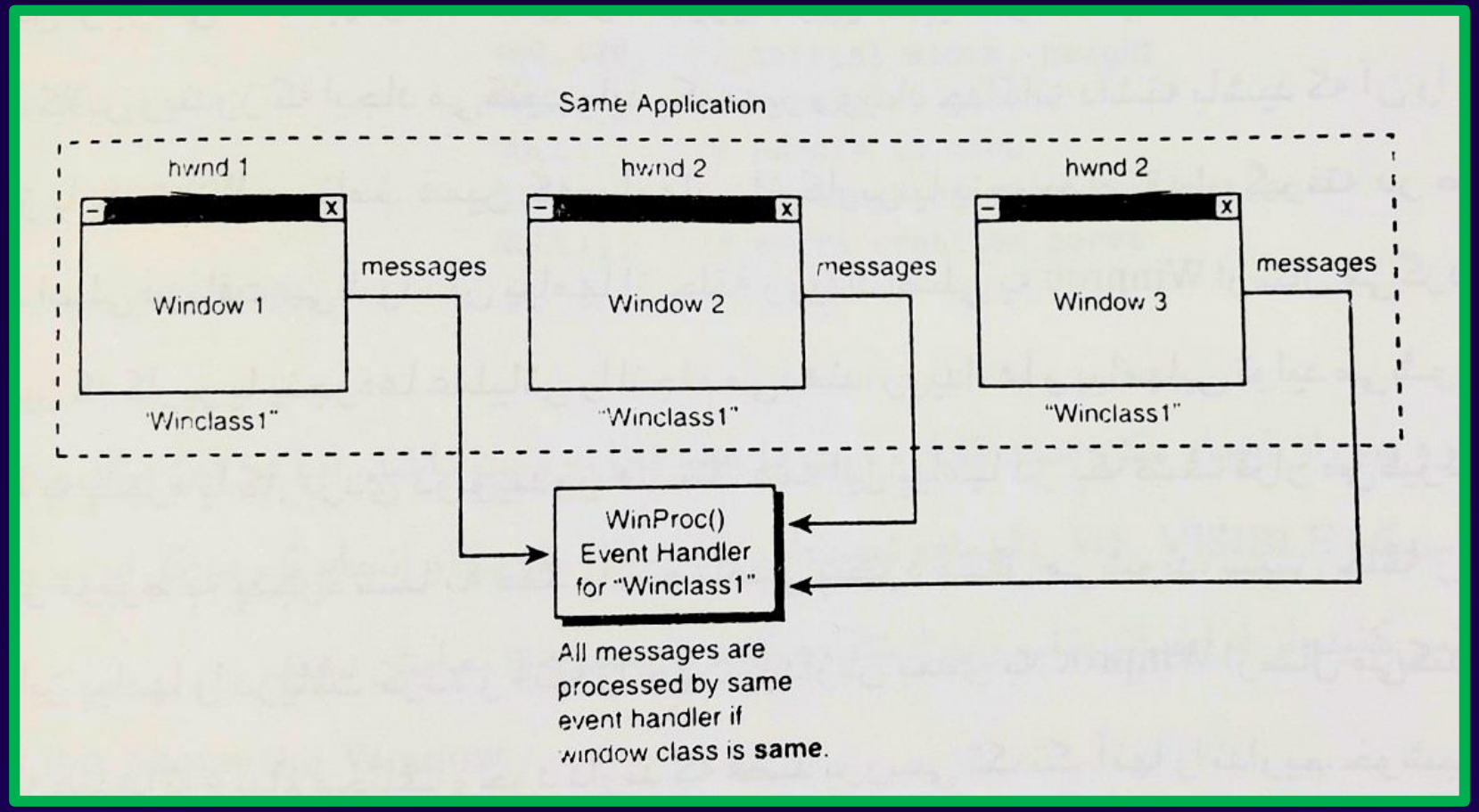


# Advanced Programming >> Message Loop

## Windows Application



## Windows Application



# Advanced Programming >> Application

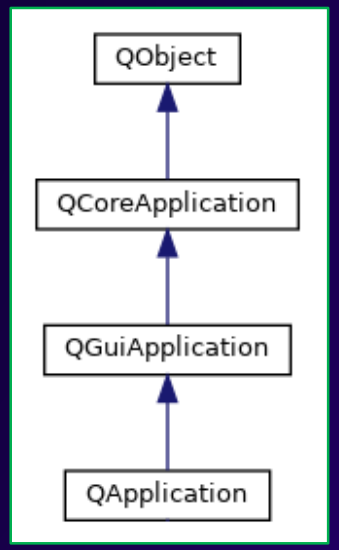
## QCoreApplication

- The QCoreApplication class provides an event loop for Qt applications without UI

## QApplication

- The QApplication class manages the GUI application's control flow and main settings.

```
1  #include <QtWidgets/QApplication>
2  #include <QWidget>
3
4  int main(int argc, char *argv[])
5  {
6      QApplication app(argc, argv);
7      QWidget w;
8
9      w.show();
10     return app.exec();
11 }
```



## QCoreApplication

| | |
|---|---|
| Header: | #include <QCoreApplication> |
| qmake: | QT += core |
| Inherits: | QObject |

## QGuiApplication

| | |
|---|---|
| Header: | #include <QGuiApplication> |
| qmake: | QT += gui |
| Inherits: | QCoreApplication |
| Inherited By: | QApplication |

## QApplication

| | |
|---|---|
| Header: | #include <QApplication> |
| qmake: | QT += widgets |
| Inherits: | QGuiApplication |

## QString

- The QString class provides a Unicode character string.

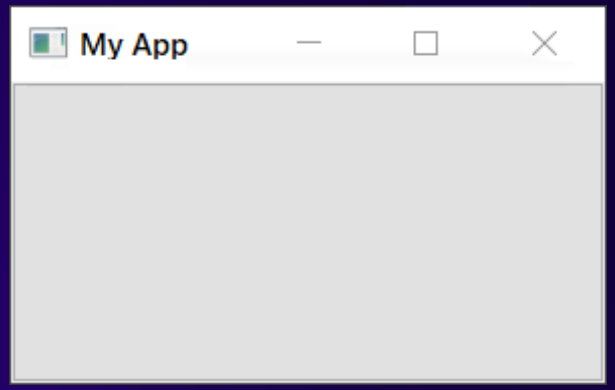
## Methods

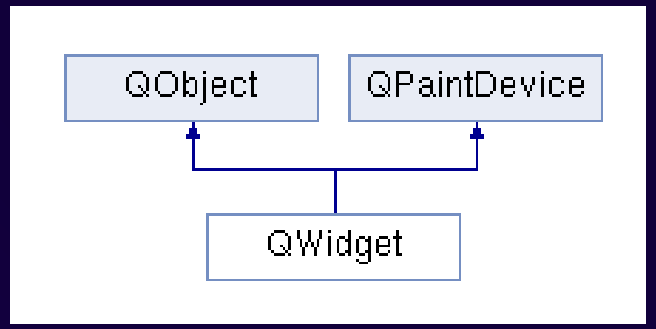
- QString(const char *str);
  - Constructs a string initialized with the 8-bit string str.
- QString();
  - Constructs a null string. Null strings are also empty.
- int length() const;
  - Returns the number of characters in this string. Equivalent to size().
- void clear();
  - Clears the contents of the string and makes it null.

| | |
|---|---|
| Header: | #include <QString> |
| qmake: | QT += core |

## QWidget

- The QWidget class is the base class of all user interface objects.
- The widget is the atom of the user interface: it receives mouse, keyboard and other events from the window system, and paints a representation of itself on the screen. Every widget is rectangular, and they are sorted in a Z-order. A widget is clipped by its parent and by the widgets in front of it.
- A widget that is not embedded in a parent widget is called a window. Usually, windows have a frame and a title bar, although it is also possible to create windows without such decoration using suitable window flags

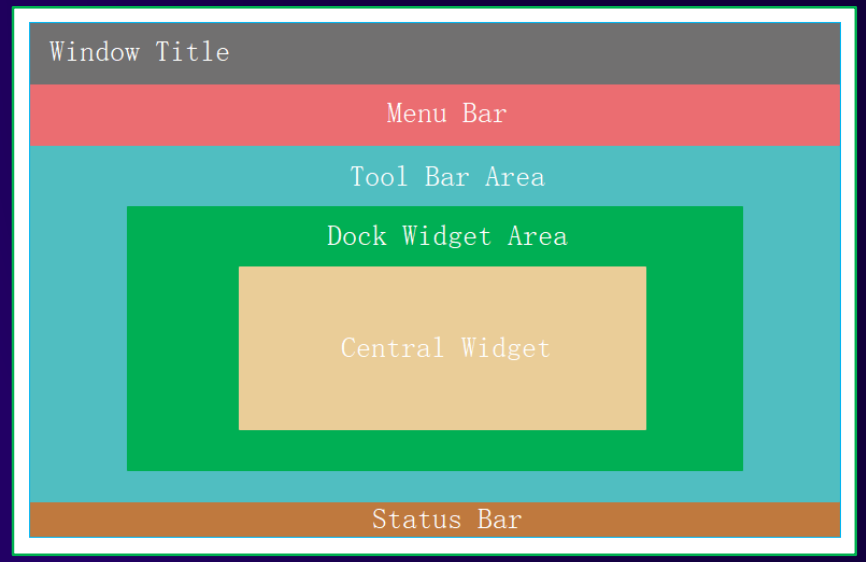


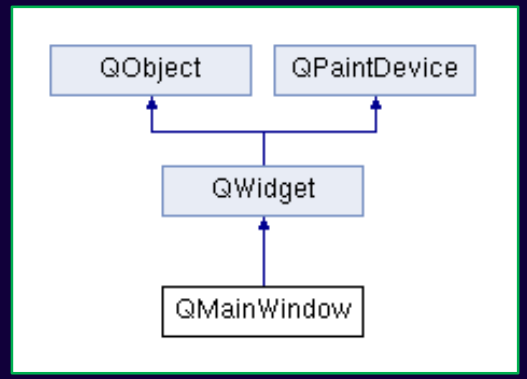




```
Header:   #include <QWidget>
qmake:    QT += widgets
```

# QMainWindow

- The QMainWindow class provides a main application window.
- QMainWindow has its own layout to which you can add QToolBars, QDockWidgets, a QMenuBar, and a QStatusBar. The layout has a center area that can be occupied by any kind of widget.

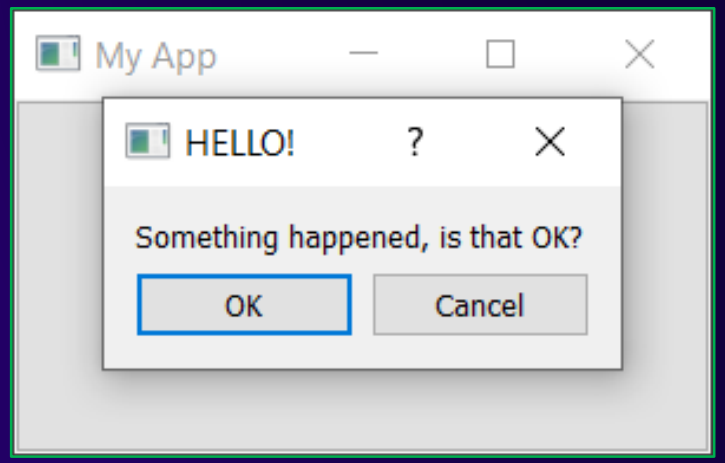


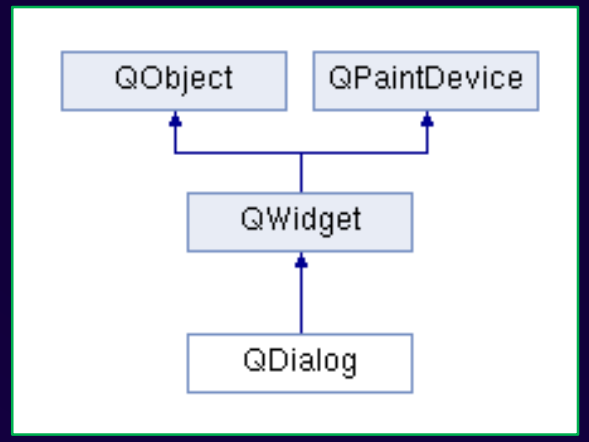




```
Header:  #include <QMainWindow>
qmake:   QT += widgets
```

## QDialog

- 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. QDialogs can provide a return value, and they can have default buttons.
- Modal Dialogs:
  - A modal dialog is a dialog that blocks input to other visible windows in the same application.

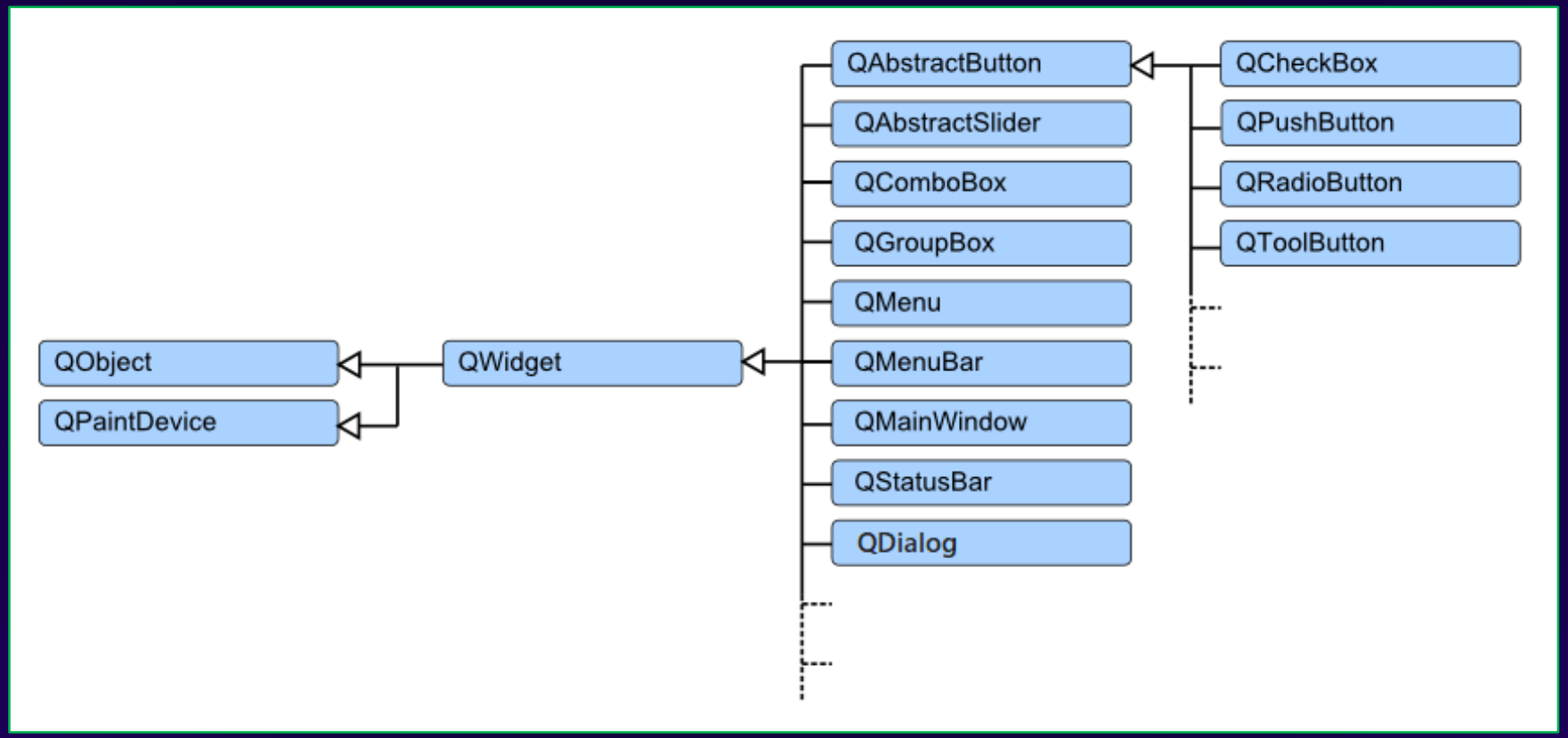



| | |
|---|---|
| Header: | #include <QDialog> |
| qmake: | QT += widgets |

## Controls

- The Qt Controls module provides a set of controls that can be used to build complete interfaces in Qt.



## QLabel

- The QLabel widget provides a text or image display. No user interaction functionality is provided.

## Methods

- QLabel(const QString &text, QWidget *parent = nullptr, Qt::WindowFlags f = Qt::WindowFlags());
  - Constructs a label that displays the text, text.
- QLabel(QWidget *parent = nullptr, Qt::WindowFlags f = Qt::WindowFlags());
  - Constructs an empty label.
- QString text() const;
- void setText(const QString &);
- void setNum(double num);
  - This is an overloaded function.
  - Sets the label contents to plain text containing the textual representation of double num. Any previous content is cleared.
- void show();
  - Shows the widget and its child widgets.

| | |
|---|---|
| Header: | #include <QLabel> |
| qmake: | QT += widgets |

# Advanced Programming >> Controls

## QPushButton

- The QPushButton widget provides a command button. Push (click) a button to command the computer to perform some action.

## Methods

- QPushButton(const QIcon &icon, const QString &text, QWidget *parent = nullptr);
  - Constructs a push button with an icon and a text, and a parent.
- QPushButton(const QString &text, QWidget *parent = nullptr);
  - Constructs a push button with the parent parent and the text text.
- QPushButton(QWidget *parent = nullptr);
  - Constructs a push button with no text and a parent.

## Signals

- void clicked(bool checked = false);
  - This signal is emitted when the button is activated (i.e., pressed down then released while the mouse cursor is inside the button), when the shortcut key is typed, or when click() or animateClick() is called.

| | |
|---|---|
| Header: | #include <QPushButton> |
| qmake: | QT += widgets |

# Advanced Programming >> Controls

## QLineEdit

- A line edit allows the user to enter and edit a single line of plain text.

## Methods

- QLineEdit(const QString &contents, QWidget *parent = nullptr);
  - Constructs a line edit containing the text contents.
- QLineEdit(QWidget *parent = nullptr);
  - Constructs a line edit with no text.
- Qstring text() const;
- void setText(const QString &);

## Signals

- void textChanged(const QString &text);
  - This signal is emitted whenever the text changes. The text argument is the new text.

| | |
|---|---|
| Header: | #include <QLineEdit> |
| qmake: | QT += widgets |

# Advanced Programming >> Controls

## Qlayout

- The QLayout class is the base class of geometry managers.

## QBoxLayout

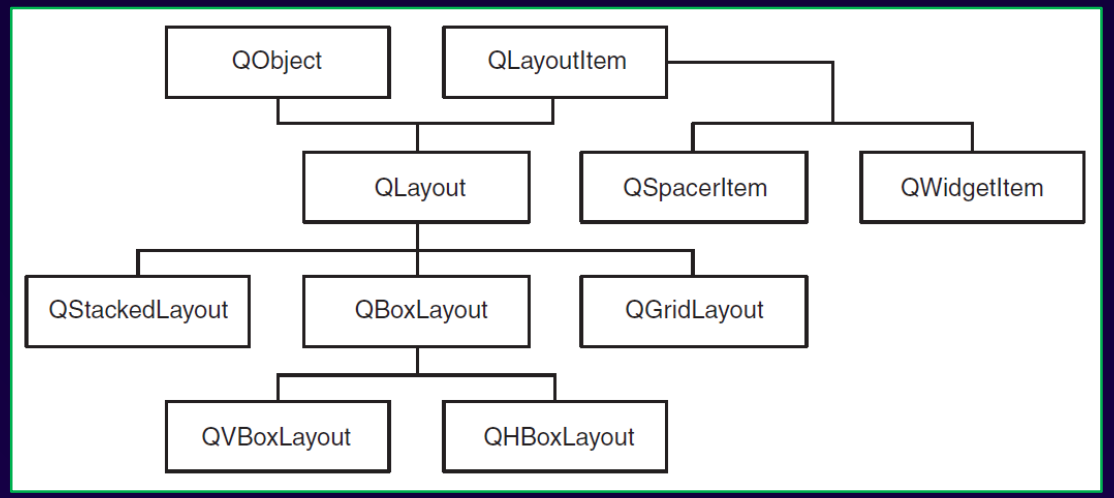
- The QBoxLayout class lines up child widgets horizontally or vertically.

## QHBoxLayout

- The QHBoxLayout class lines up widgets horizontally.

## QVBoxLayout

- The QVBoxLayout class lines up widgets vertically.



# Thank You

By Ali Panahi