

بِسْمِ اللّٰهِ الرَّحْمٰنِ الرَّحِيْمِ



مقام معظم رهبری:

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

۱۳۹۴/۰۸/۲۰





Qt Training in C++

Lecturer: Ali Panahi

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.

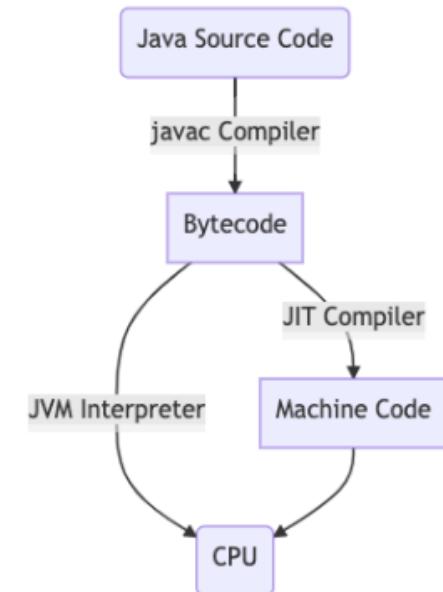
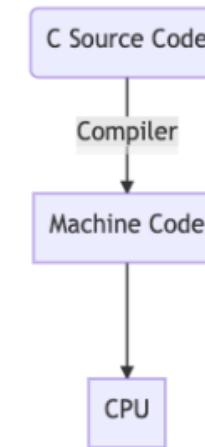
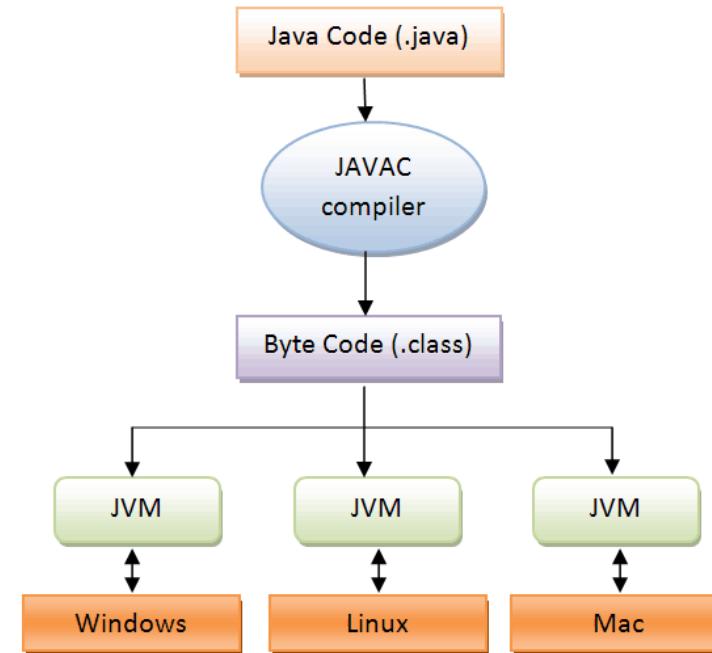
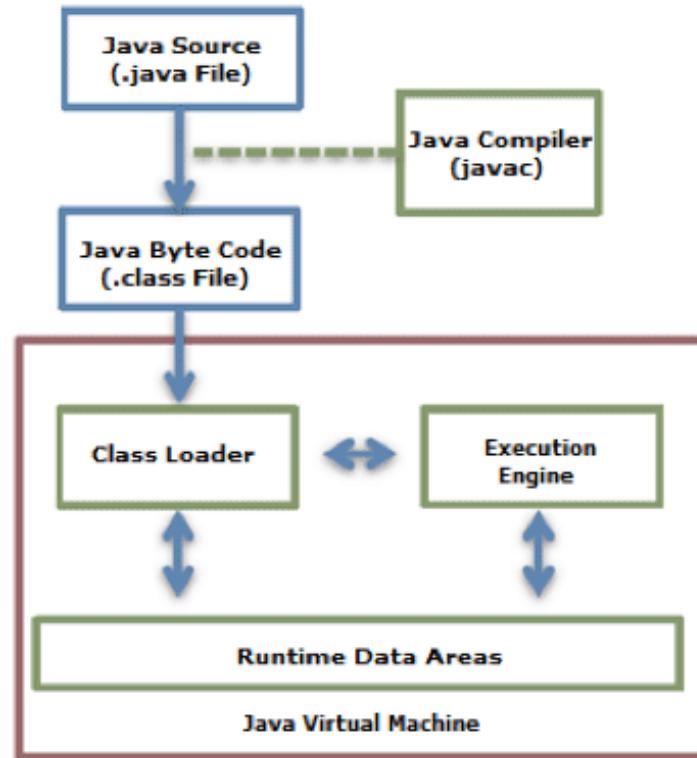
Why use Qt ?

- 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

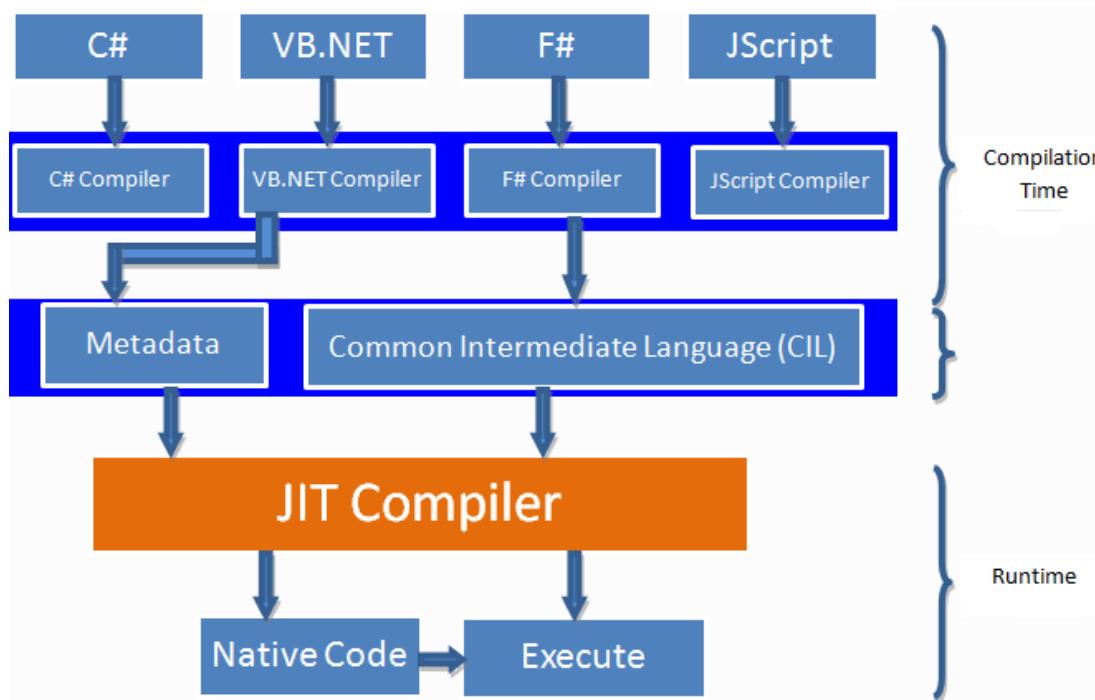




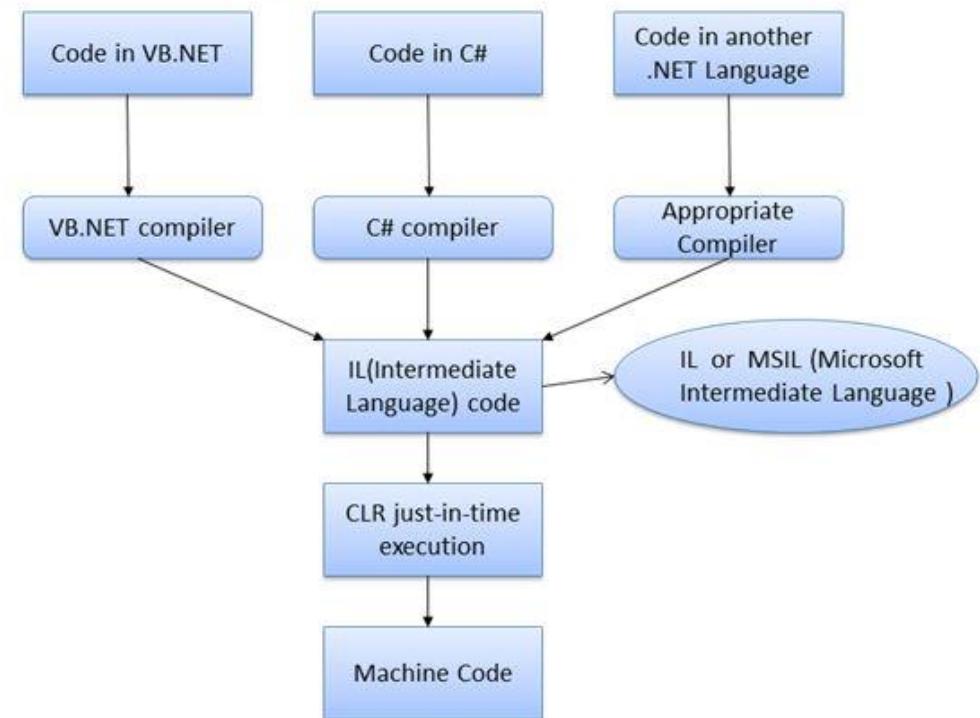
Java vs .Net vs C++ vs QT vs .Net Core

- .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



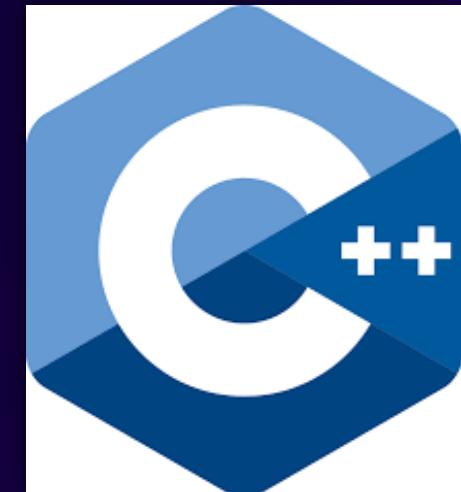


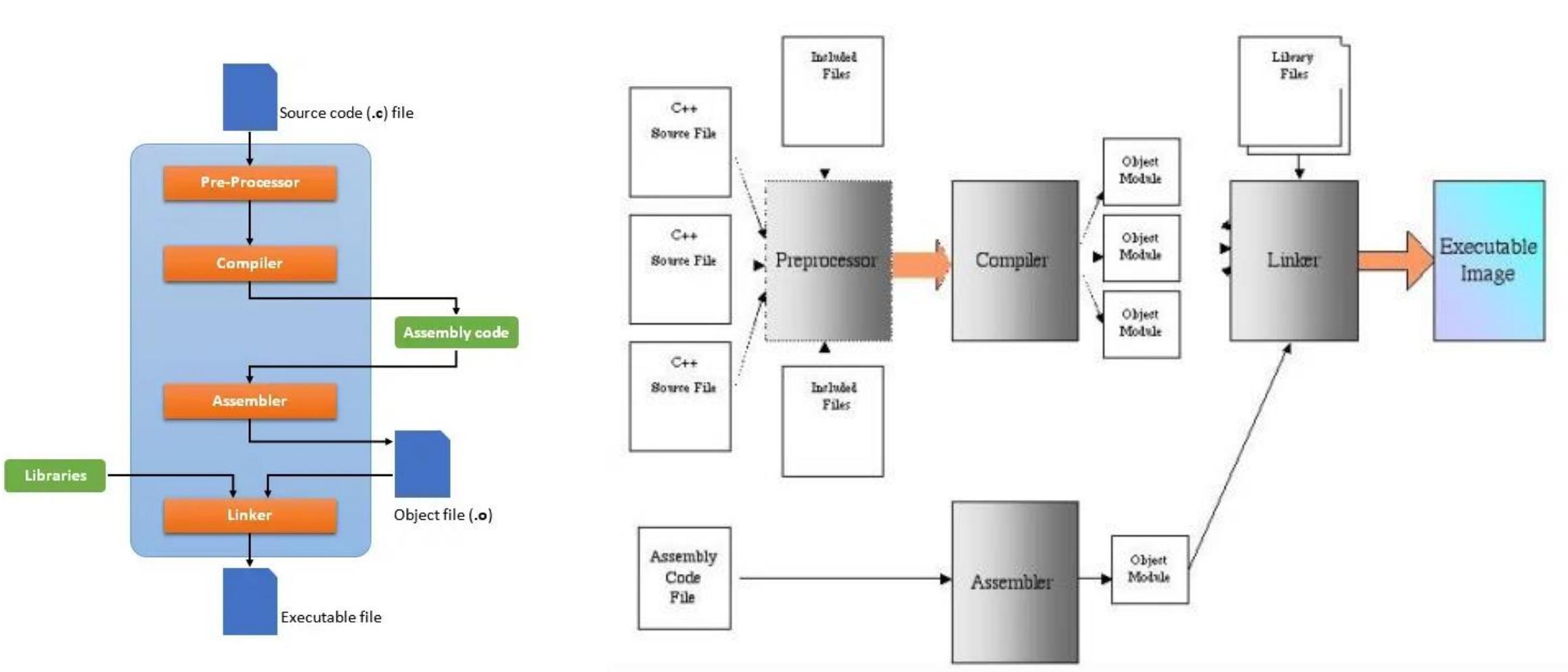
Compilation in .NET



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

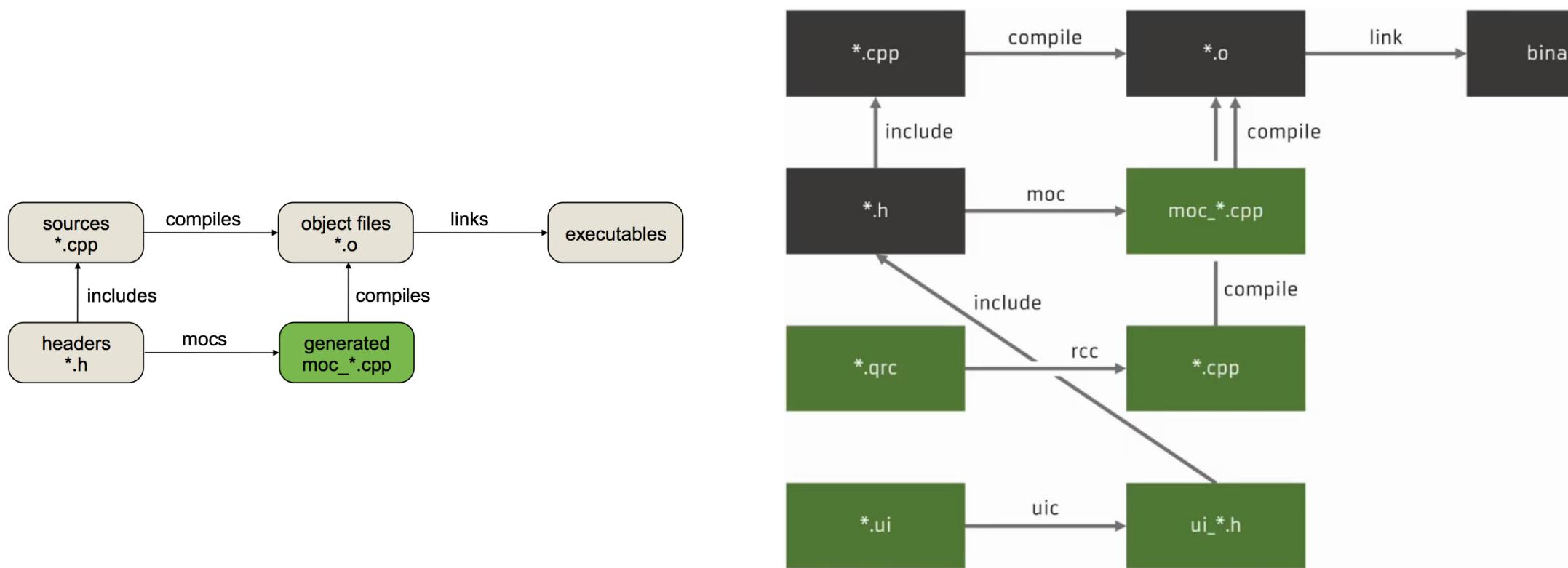


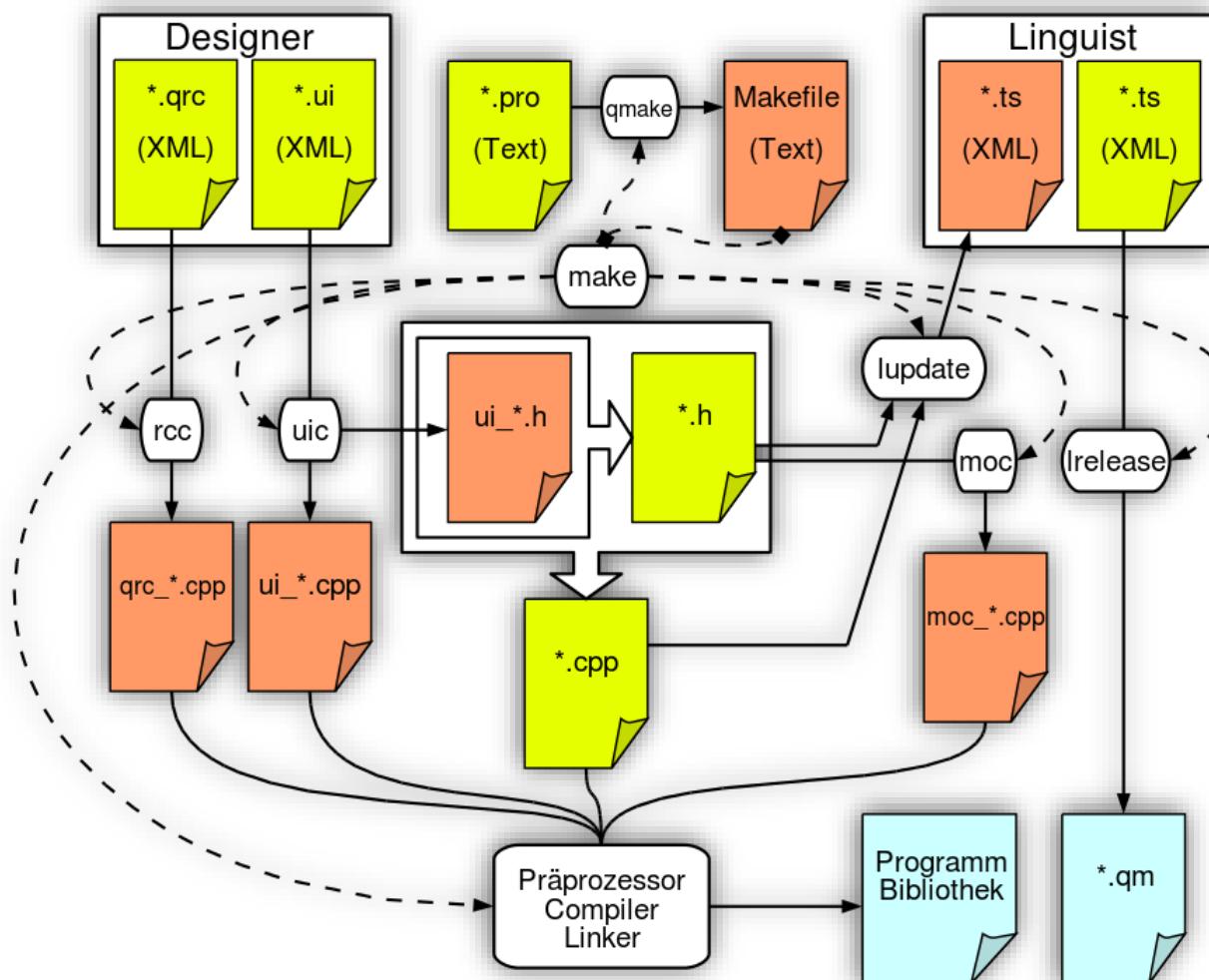


Java vs .Net vs C++ vs QT vs .Net Core

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

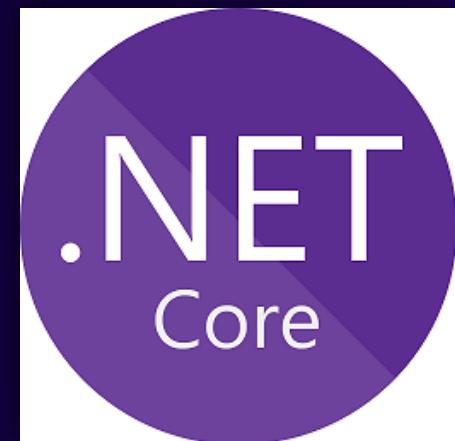






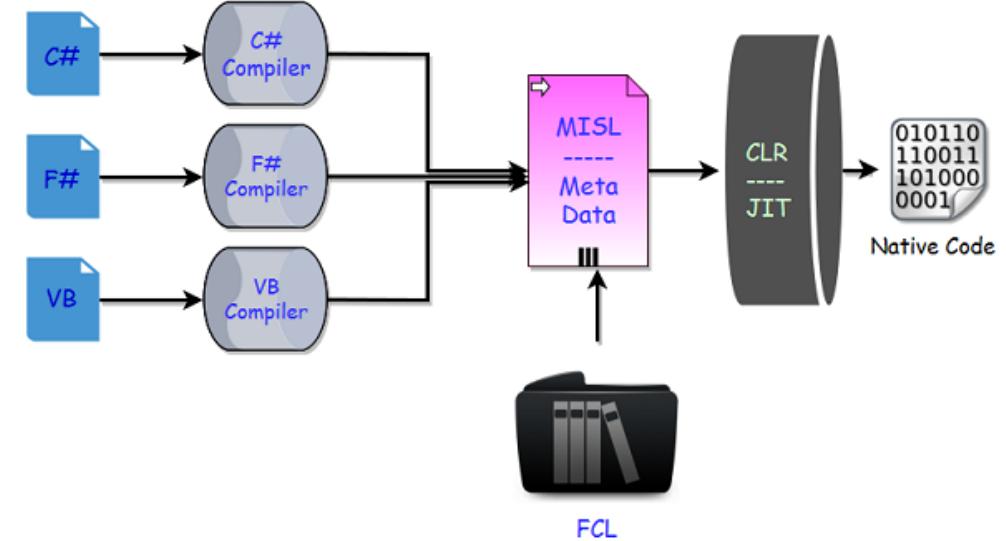
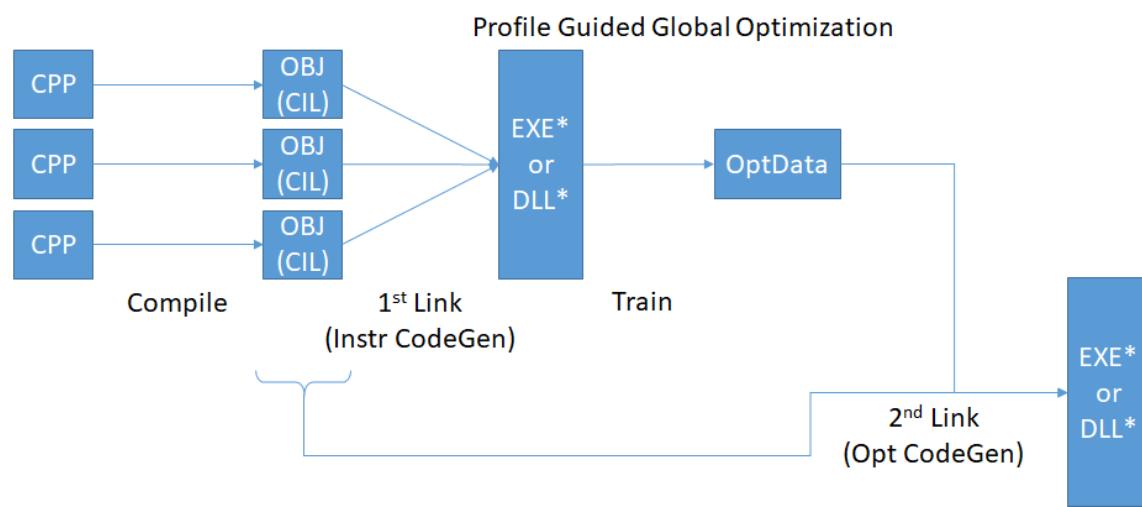
Java vs .Net vs C++ vs QT vs .Net Core

- .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

Qt



Companies using Qt

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



In-flight
entertainment
systems



In-vehicle
infotainment
system



Graphics software



Anesthesia &
critical care
medical devices



EDA & CAD end-
to-end
engineering
solutions



Automotive
mobility
technology

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

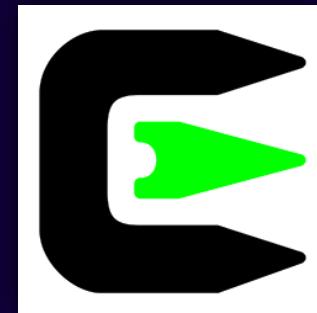
Languages that support the Qt library

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.

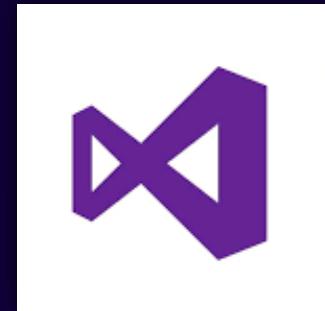
Qt compilers

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



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

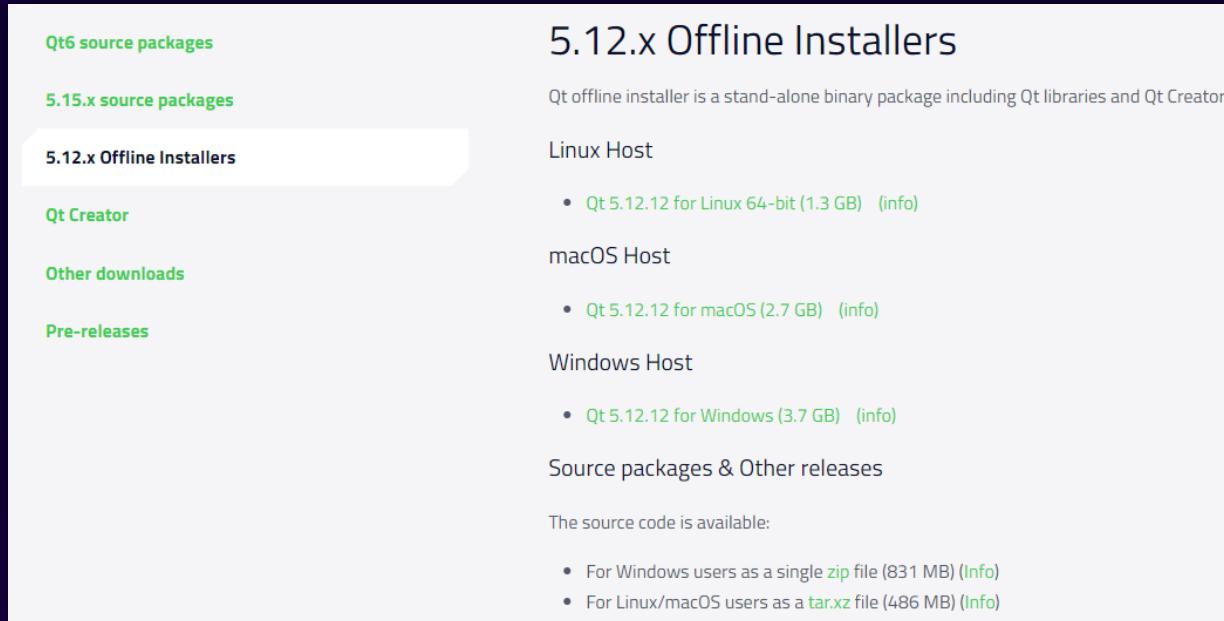
Offline Installer

On Windows

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

On Linux

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



5.12.x Offline Installers

Qt offline installer is a stand-alone binary package including Qt libraries and Qt Creator.

Linux Host

- [Qt 5.12.12 for Linux 64-bit \(1.3 GB\)](#) (info)

macOS Host

- [Qt 5.12.12 for macOS \(2.7 GB\)](#) (info)

Windows Host

- [Qt 5.12.12 for Windows \(3.7 GB\)](#) (info)

Source packages & Other releases

The source code is available:

- For Windows users as a single [zip](#) file (831 MB) ([Info](#))
- For Linux/macOS users as a [tar.xz](#) file (486 MB) ([Info](#))

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

Qt® and the Qt logo is a registered trade mark of The Qt Company Ltd and is used pursuant to a license from The Qt Company Ltd.
All other trademarks are property of their respective owners.

The Qt Company Ltd, Bertel Jungin aukio D3A, 02600 Espoo, Finland. Org. Nr. 2637805-2

[List of official Qt-project mirrors](#)

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
```

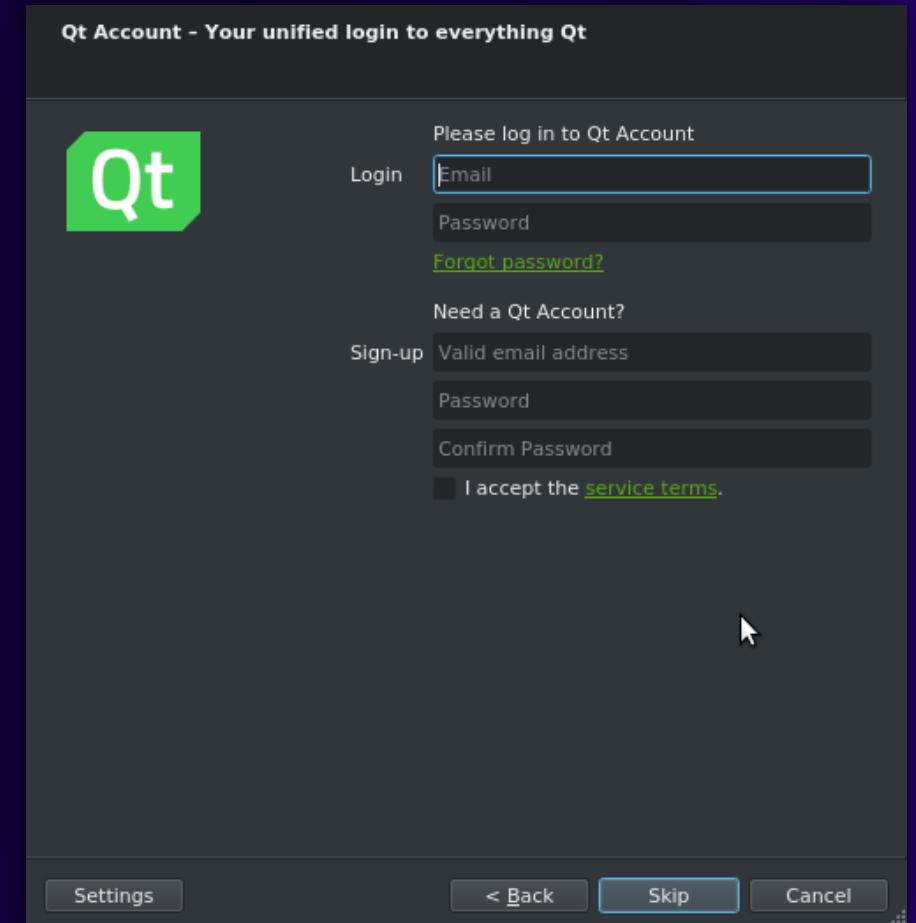
Offline/Online Installer

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.

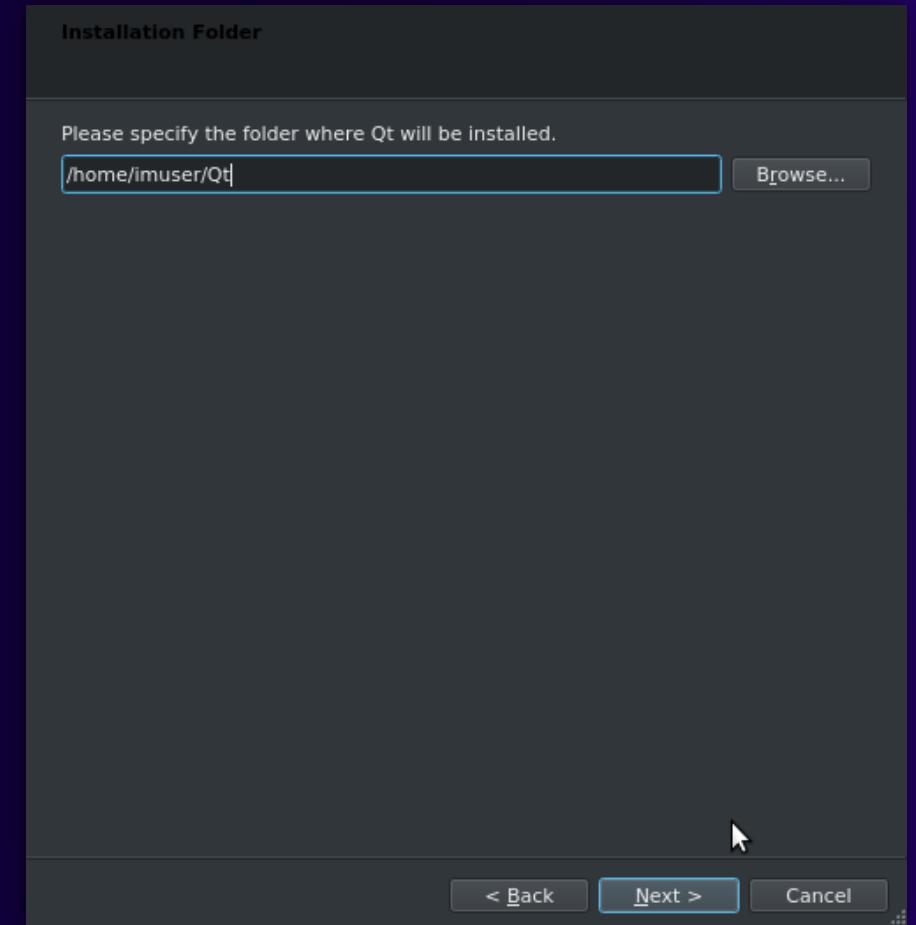
Login with a existing Qt account or create a new one:



Offline/Online Installer

Install Qt in any operative system

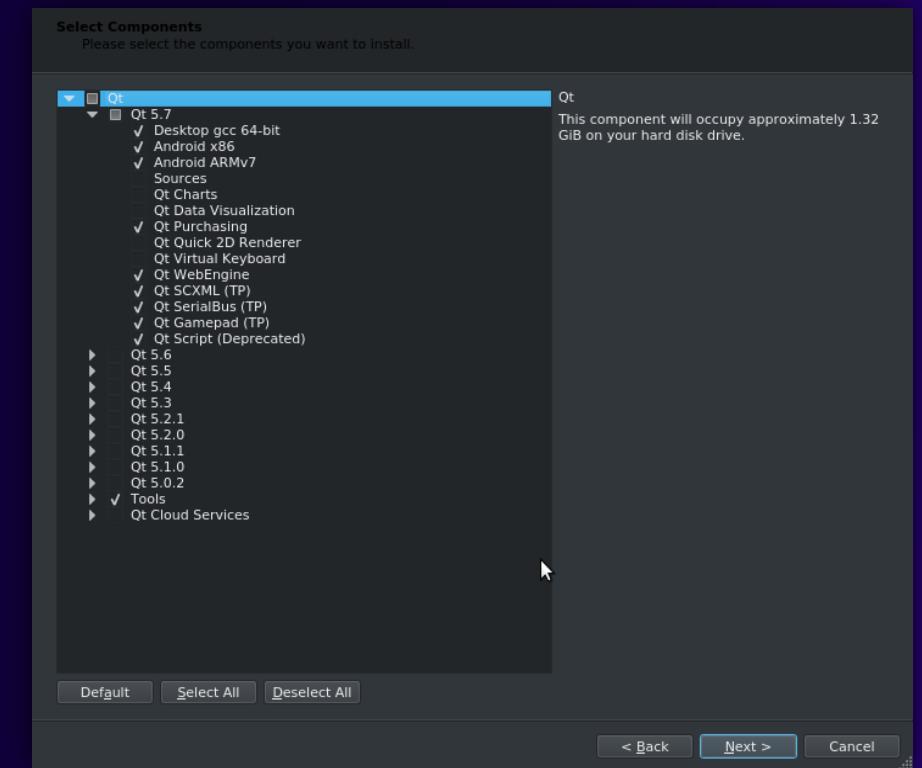
Select a path to install the Qt libraries and tools



Offline/Online Installer

Install Qt in any operative system

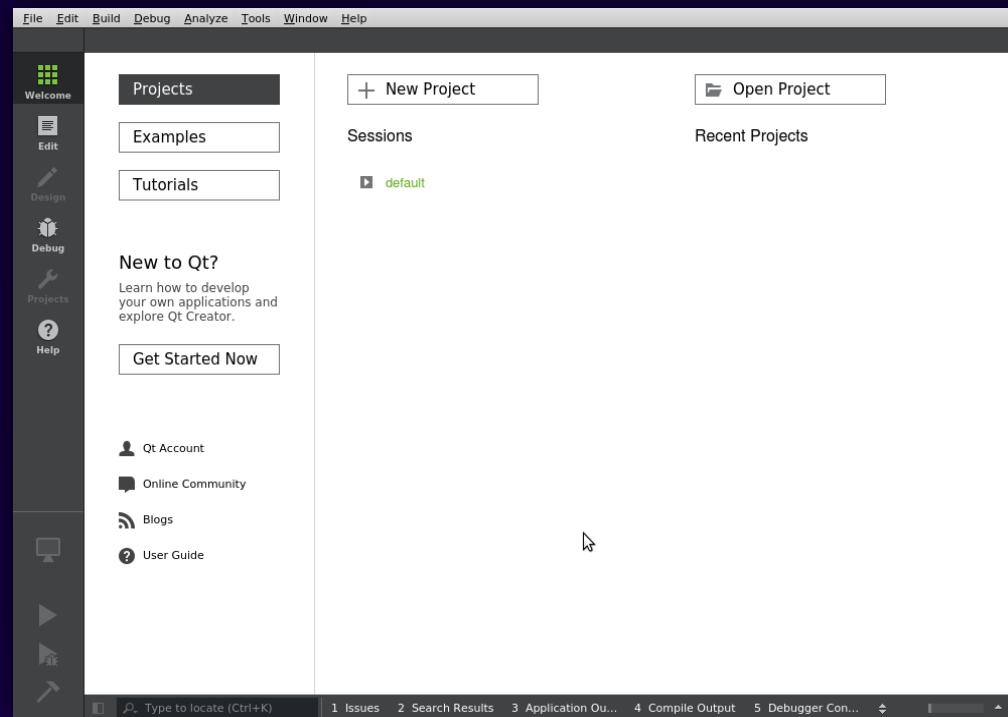
Select the library version and the features you want



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

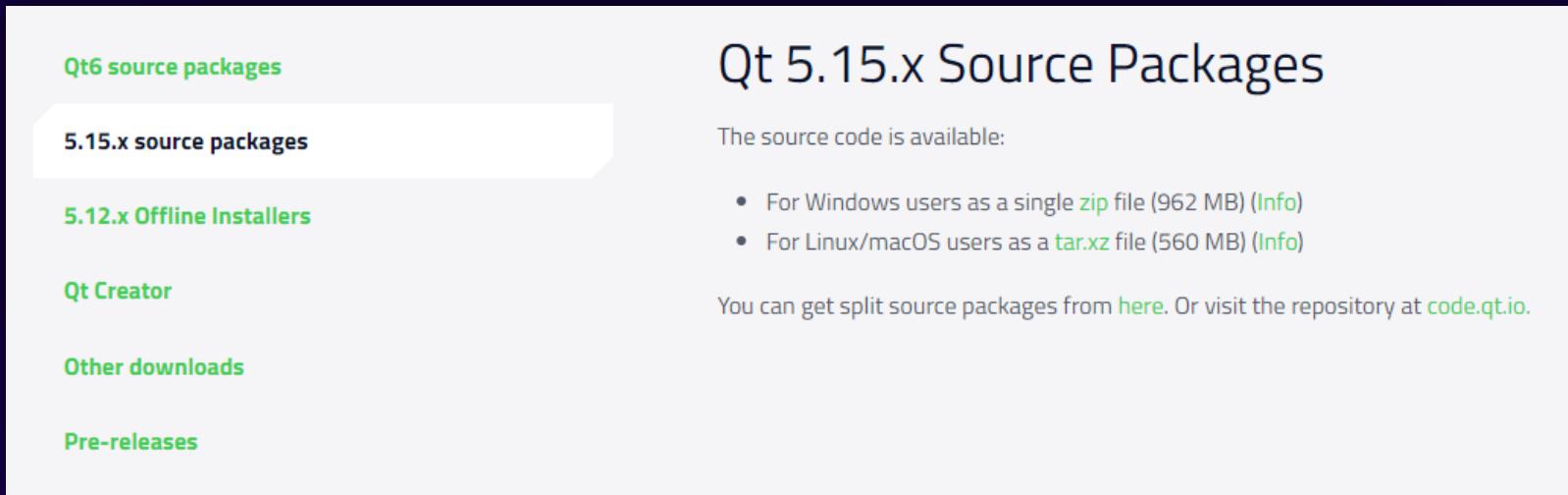
```
$ sudo apt-get update && sudo apt-get upgrade  
$ sudo apt install qtbase5-dev qt5-qmake qtbase5-dev-tools  
$ sudo apt-get install qtcreator  
$ qtcreator
```

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



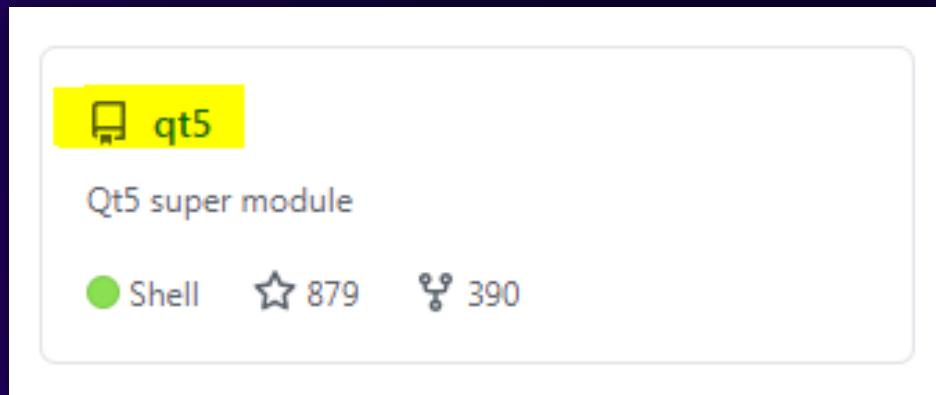
The screenshot shows the "Qt 5.15.x Source Packages" section of the Qt download page. On the left, there's a sidebar with links: "Qt6 source packages", "5.15.x source packages" (which is highlighted in blue), "5.12.x Offline Installers", "Qt Creator", "Other downloads", and "Pre-releases". The main content area has a heading "Qt 5.15.x Source Packages" and a sub-heading "The source code is available:". Below this, there are two bullet points:

- For Windows users as a single [zip](#) file (962 MB) ([Info](#))
- For Linux/macOS users as a [tar.xz](#) file (560 MB) ([Info](#))

At the bottom, it says "You can get split source packages from [here](#). Or visit the repository at [code.qt.io](#).

Build in Windows

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



System requirements

- CMake 3.18 or later
- Perl 5.8 or later
- Python 2.7 or later
- C++ compiler supporting the C++17 standard

It's recommended to have ninja 1.8 or later installed.

For other platform specific requirements, please see section "Setting up your machine" on:
http://wiki.qt.io/Get_The_Source

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

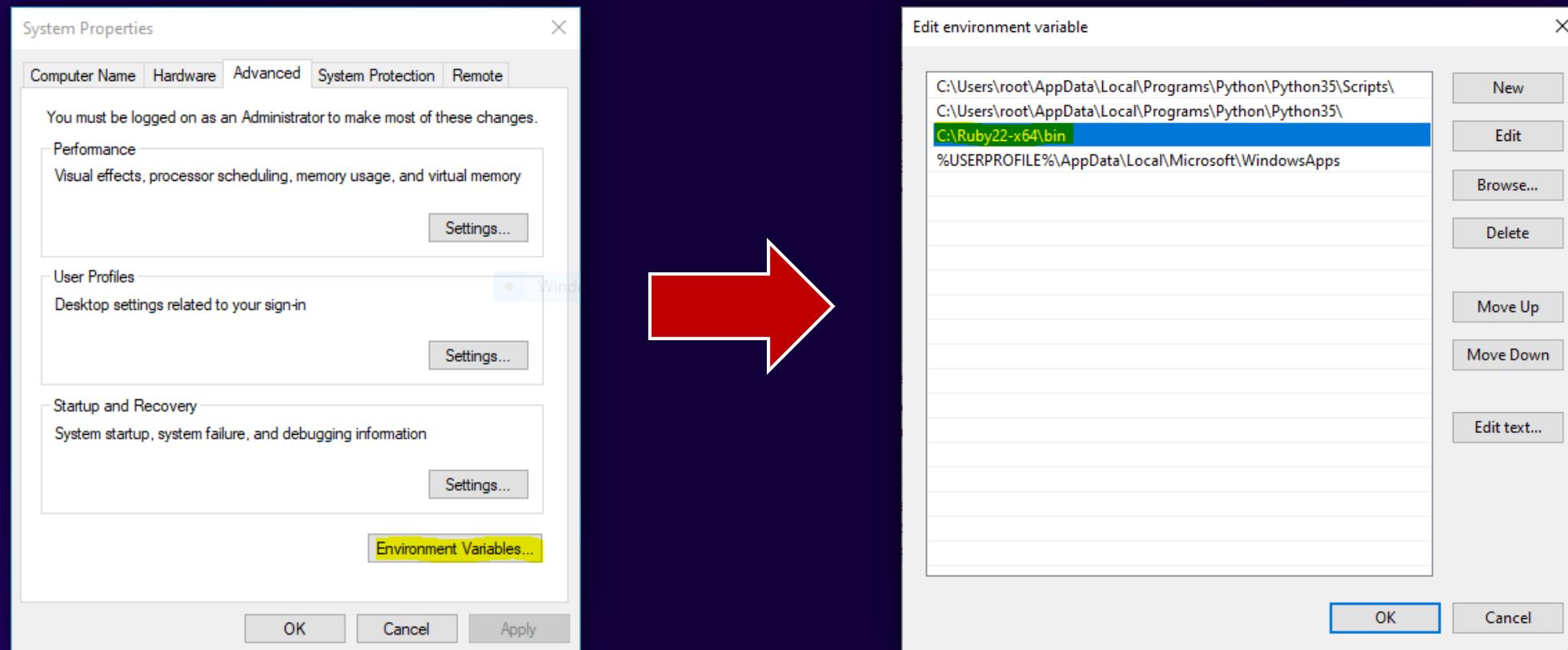
- Compiler:
 - Visual Studio 2019 or later
 - MinGW-builds gcc 8.1 or later
- Tools:
 - Perl version 5.12 or later [<http://www.activestate.com/activeperl/>]
 - Python version 2.7 or later [<http://www.activestate.com/activepython/>]
 - Ruby version 1.9.3 or later [<http://rubyinstaller.org/>]

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 Environment”



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

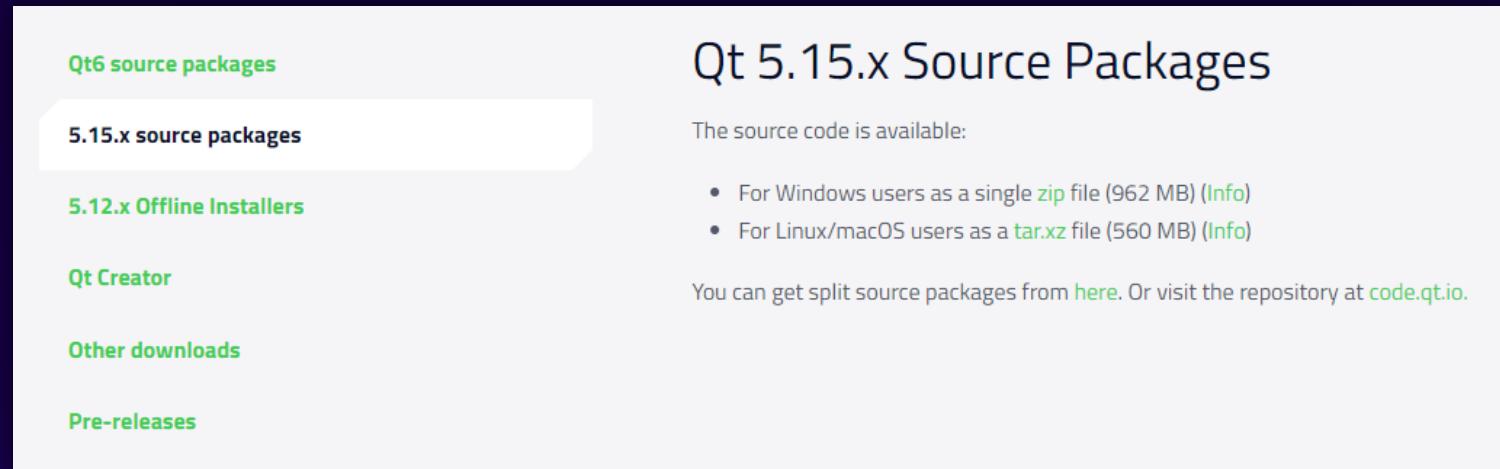
```
> mingw32-make clean
```

- 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-openssl110-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	<code>-qt-zlib</code>	<code>-system-zlib</code>
libjpeg	<code>-qt-libjpeg</code>	<code>-system-libjpeg</code>
libpng	<code>-qt-libpng</code>	<code>-system-libpng</code>
freetype	<code>-qt-freetype</code>	<code>-system-freetype</code>
PCRE	<code>-qt-pcre</code>	<code>-system-pcre</code>
<u>HarfBuzz-NG</u>	<code>-qt-harfbuzz</code>	<code>-system-harfbuzz</code>

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.

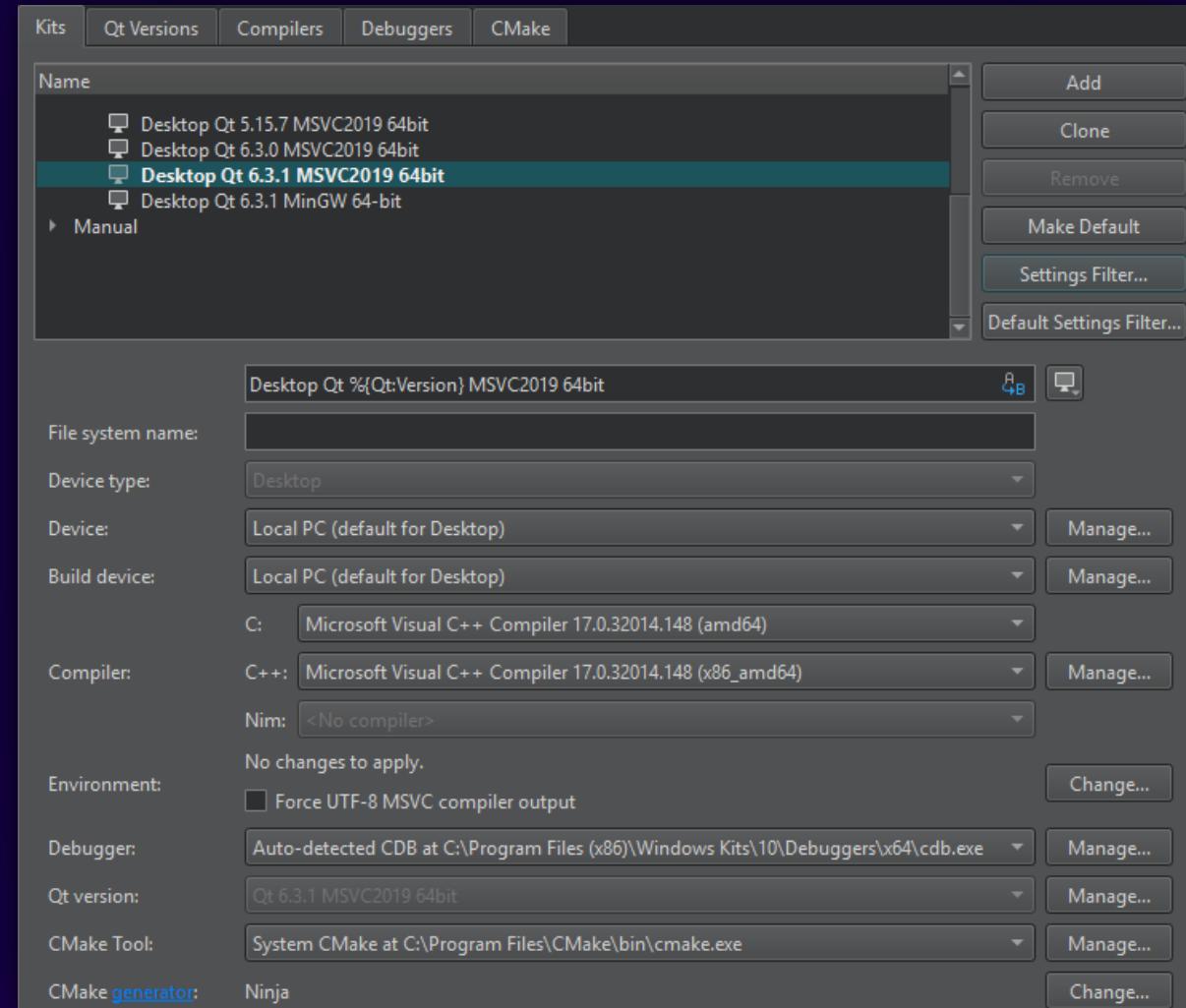
Kit Settings

- 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.

Kits:



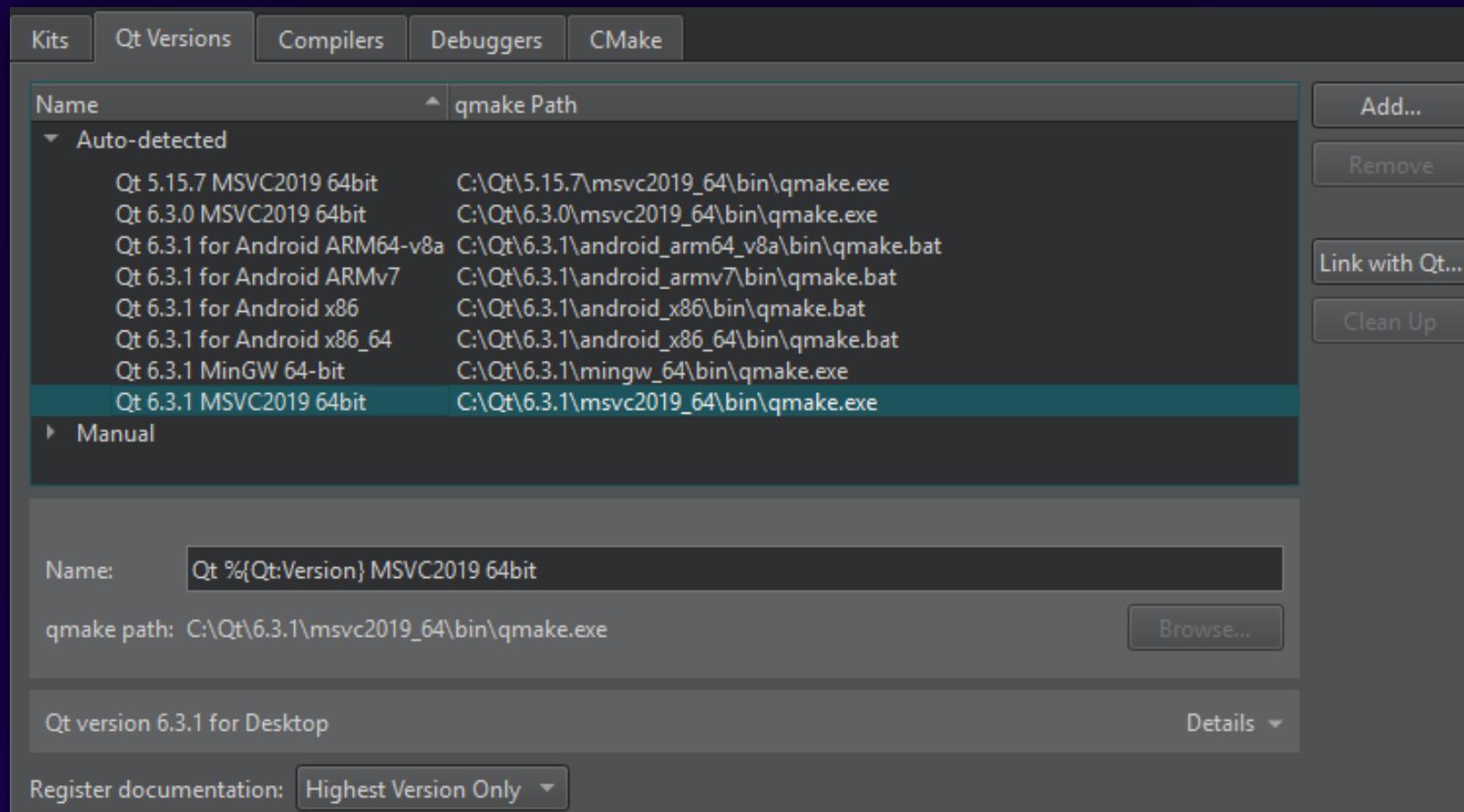
Adding Qt Versions

- 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.

Qt Versions:



Adding Compilers

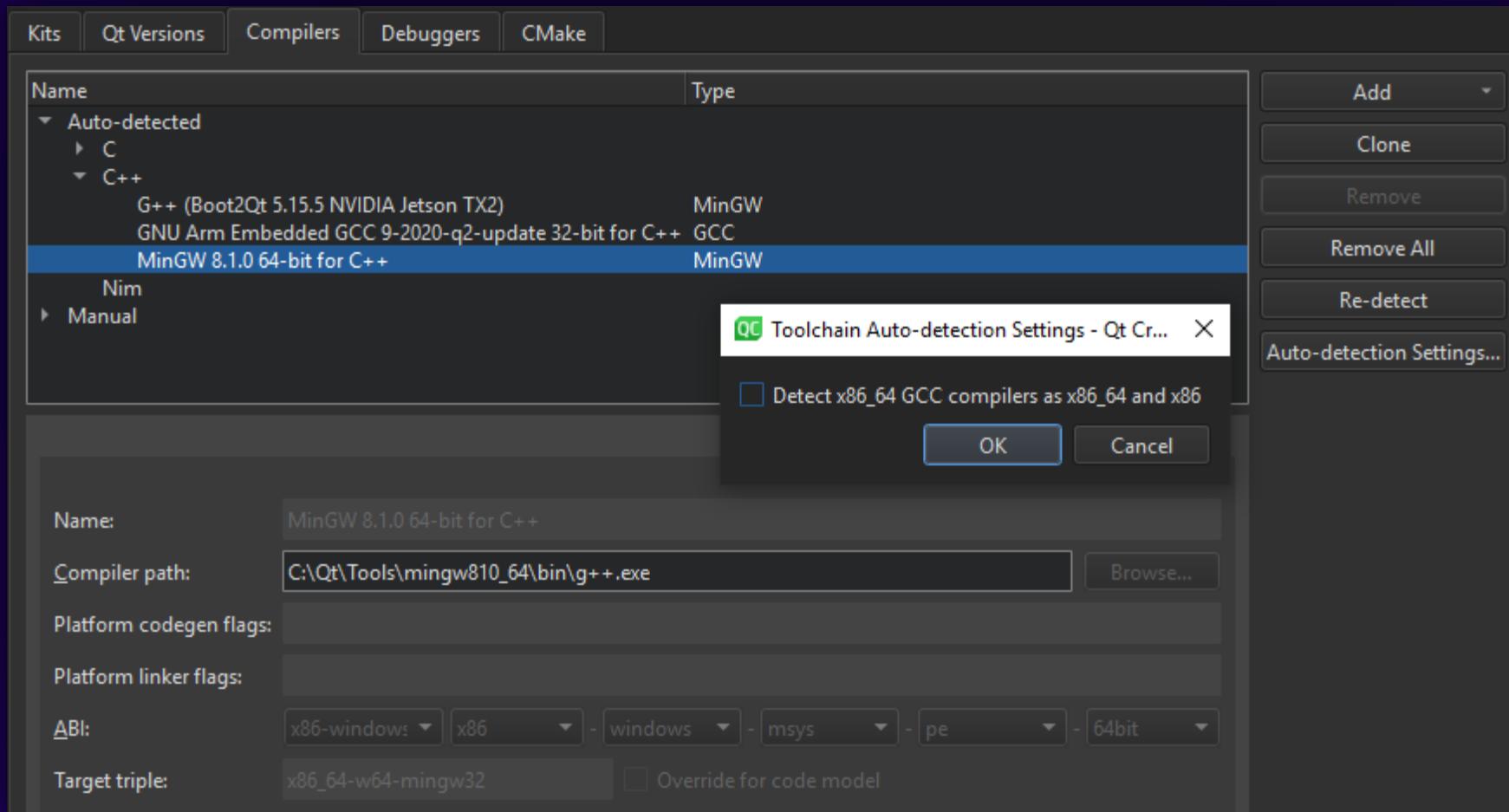
- Qt is supported on a variety of 32-bit and 64-bit platforms, and can usually be built on each platform with GCC, a vendor-supplied compiler, or a third party compiler.

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)

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)

Qt Debuggers:

The screenshot shows the 'Debuggers' tab of the Qt Creator settings. The interface includes a header with tabs: Kits, Qt Versions, Compilers, Debuggers (selected), and CMake. Below the tabs is a table with columns: Name, Location, and Type. The table lists several debuggers, both auto-detected and manual. A modal dialog is open at the bottom, allowing configuration for a selected debugger named 'CDB'. The dialog fields include: Name (CDB), Path (C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\cdb.exe), Type (CDB), ABIs (x86-windows-msvc2017-pe-32bit), Version (10.0.18362.1), and Working directory (empty). At the bottom right of the dialog are OK, Cancel, and Apply buttons.

Name	Location	Type	Add
Auto-detected			
GNU gdb 7.10.1 for MinGW 5.3.0 32bit	C:\Qt\Tools\mingw530_32\bin\gdb.exe	GDB	Clone
GNU gdb 8.1 for MinGW 7.3.0 64-bit	C:\Qt\Tools\mingw730_64\bin\gdb.exe	GDB	Remove
System LLDB at C:\Program Files\LLVM\bin\lldb.exe	C:\Program Files\LLVM\bin\lldb.exe	LLDB	
Auto-detected CDB at C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\cdb.exe	C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\cdb.exe	CDB	
Auto-detected CDB at C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\cdb.exe	C:\Program Files (x86)\Windows Kits\10\Debuggers\x64\cdb.exe	CDB	
Android Debugger (armeabi-v7a, NDK 20.1.5948944)	C:\Users\rimietti\AppData\Local\Android\Sdk\ndk-bundle\prebuilt\windows\bin\gdb.exe	GDB	
Android Debugger (armeabi-v7a, arm64-v8a, x86, x86_64, NDK 20.1.5948944)	C:\Users\rimietti\AppData\Local\Android\Sdk\ndk-bundle\prebuilt\windows\bin\gdb.exe	GDB	
Android Debugger (x86, NDK 20.1.5948944)	C:\Users\rimietti\AppData\Local\Android\Sdk\ndk-bundle\prebuilt\windows\bin\gdb.exe	GDB	
Manual			
CDB	C:\Program Files (x86)\Windows Kits\10\Debuggers\x86\cdb.exe	CDB	

Concepts

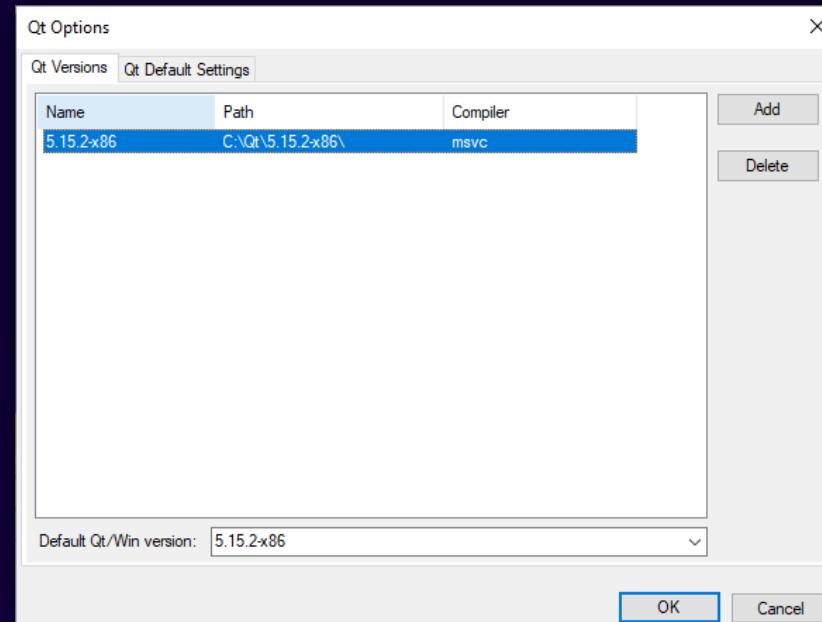
- **Shadow building** means building a project in a separate directory, the build directory. The build directory is different from the source directory. One of the benefits of shadow building is that it keeps your source directory clean, which makes it faster to switch between build configurations.
-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

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)



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
```

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
```

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".

```
$ export MY_ENV=/opt/... # echo $MYENV
$ cd my\dir\hello
$ qmake -project
$ qmake CONFIG+=debug|release ${qmake_options}
$ make ${make_options}
$ ./hello | ./hello &
$ make clean
```

g++

```
> set PATH=%PATH%;c:\qt\..\bin
> cd my\dir\hello
> qmake -project
> qmake
> nmake -f Makefile.Release | nmake -f Makefile.Debug
> hello.exe | start /wait hello.exe
> nmake clean
```

MSVC

Qt Console Application

- Add this line in *.pro file

```
QT -= gui  
  
CONFIG += c++11 console
```

- Show exit code:

```
$ echo $?
```

Linux

```
> echo %errorlevel%
```

Windows

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!".

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.

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.

- **-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.

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.

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.

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
```

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.

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  
}
```

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
```

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.
```

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 void bar_optimized(void) {  
2     foo = 0;  
3  
4     while (true);  
5 }  
6
```

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

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

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 referenced)
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:

```
1 #include <iostream>
2
3 void f(int a, int b, int c = 1);      // valid
4 void f(int a, int b = 1, int c);      // valid, add another default
5 void f(int a = 1, int b, int c);      // valid, add another default
6 void f(int a, int b, int c)
7 {
8     std::cout << a << ", " << b << ", " << c << std::endl;
9 }
10
11 int main()
12 {
13     f();
14     f(8);
15     f(8, 9);
16     f(8, 9, 10);
17
18     return 0;
19 }
```

- 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.

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*>(&stmt)) {  
2      ...  
3  }  
4  else if (ExprStm* e = dynamic_cast<ExprStm*>(&stmt)) {  
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.

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.



Sample 1

```
1 #include <gtk/gtk.h>
2
3 int main(int argc, char* argv[])
4 {
5     gtk_init(&argc, &argv);
6     GtkWidget* win = gtk_window_new(GTK_WINDOW_TOPLEVEL);
7     g_signal_connect(win, "delete_event", gtk_main_quit, NULL);
8     gtk_widget_show(win);
9     gtk_main();
10    return 0;
11 }
```

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

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`
```

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.
- Note: If the class is not in a separate file, include the file name with .moc extension before use class in that file. (#include “main.moc”).

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 alternative 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 from 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 its parent.

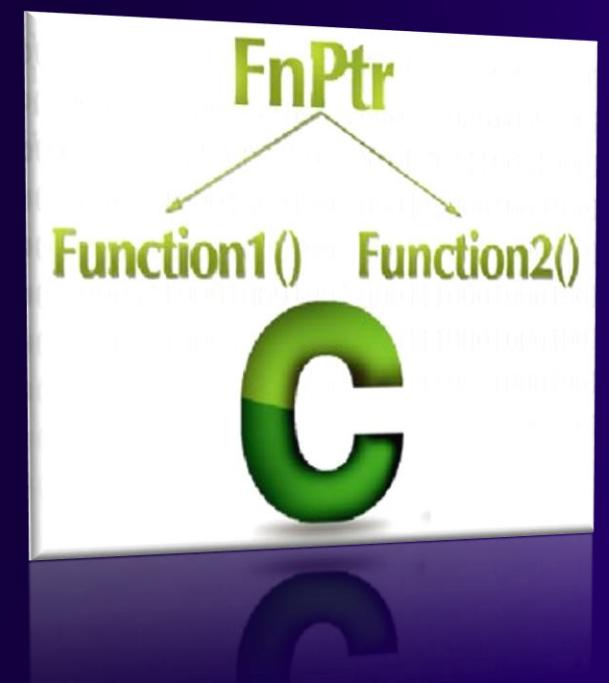
Note: You can manually call `setParent` with a reverse order of declaration which will break the automatic destruction.

Remarks

- Signals and slots are used for communication between objects.

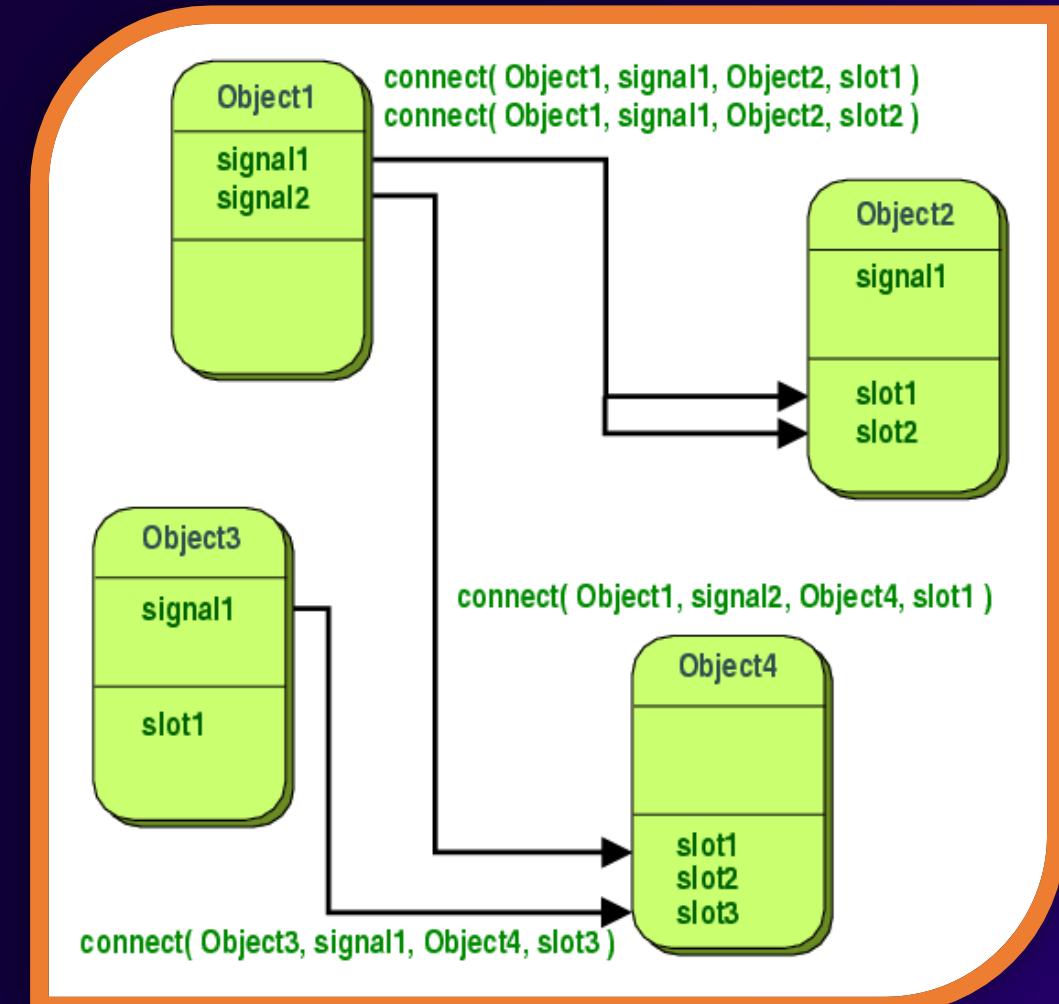
Introduction

- 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.

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.
- **Qt::BlockingQueuedConnection**
 - Same as Qt::QueuedConnection, except that the signalling thread blocks until the slot returns. This connection must not be used if the receiver lives in the signalling thread, or else the application will deadlock.

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.

```
1 #include <QObject>
2
3 class Counter : public QObject
4 {
5     Q_OBJECT
6
7 public:
8     Counter() { m_value = 0; }
9     int value() const { return m_value; }
10
11 public slots:
12     void setValue(int value);
13
14 signals:
15     void valueChanged(int newValue);
16
17 private:
18     int m_value;
19 };
```

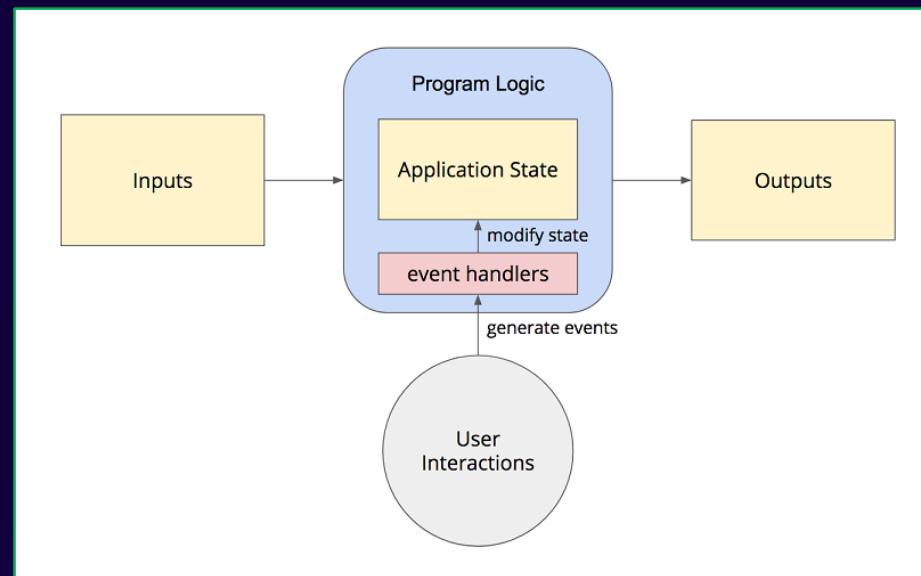
```
1 void Counter::setValue(int value)
2 {
3     if (value != m_value) {
4         m_value = value;
5         emit valueChanged(value);
6     }
7 }
8
9 Counter a, b;
10 QObject::connect(&a, &Counter::valueChanged,
11                  &b, &Counter::setValue);
12
13 a.setValue(12); // a.value() == 12, b.value() == 12
14 b.setValue(48); // a.value() == 12, b.value() == 48
```

Application Type

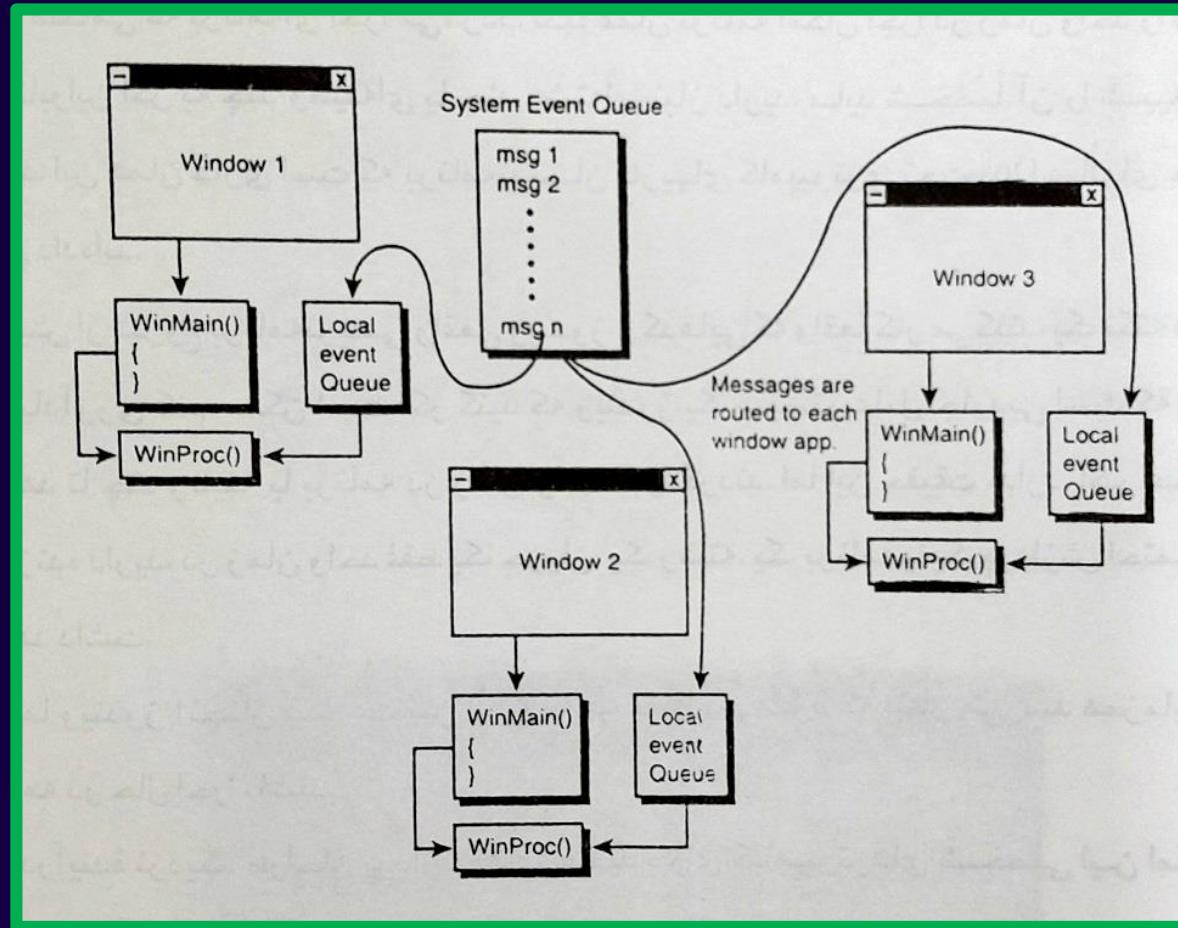
- Procedural
 - The program starts from a point and ends after doing the necessary tasks.
- Interactive (Event based)
 - The program always resides in the memory and waits for the user's commands.

Thread Type

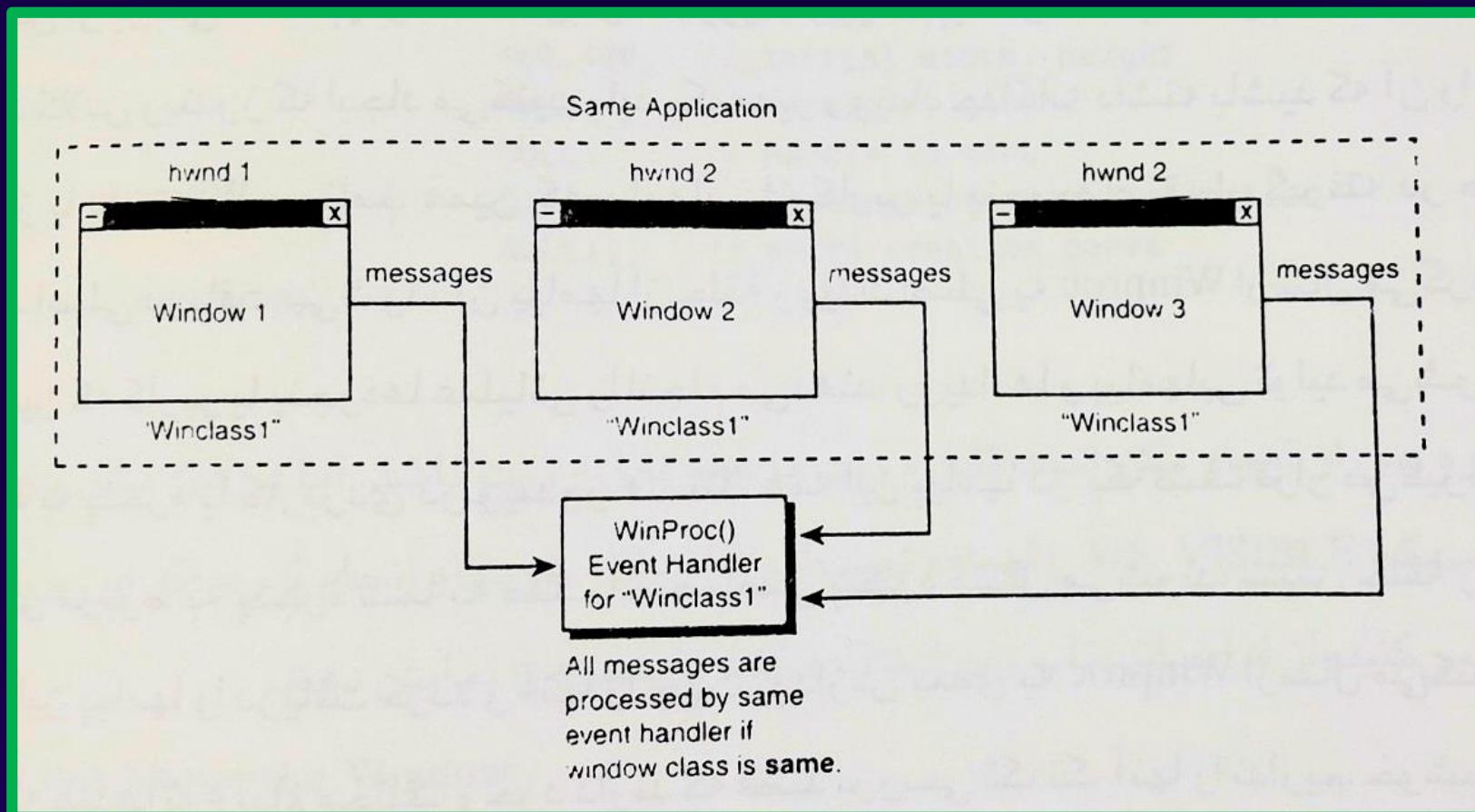
- Worker
- Ui



Windows Application



Windows Application



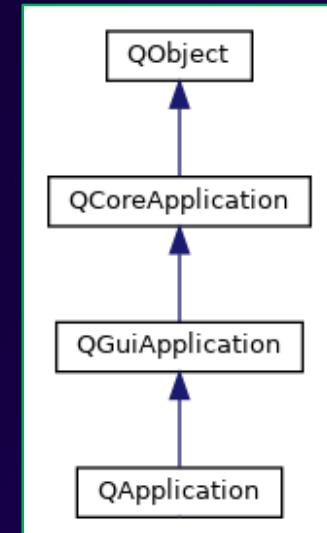
QCoreApplication

- 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

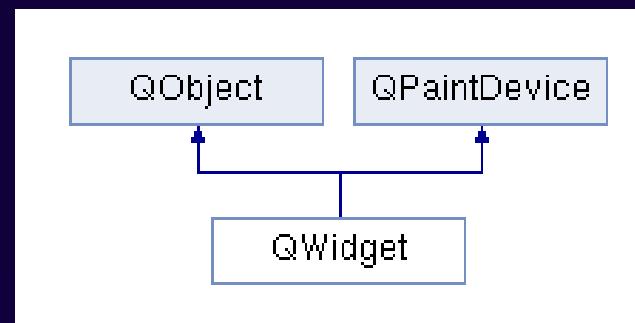
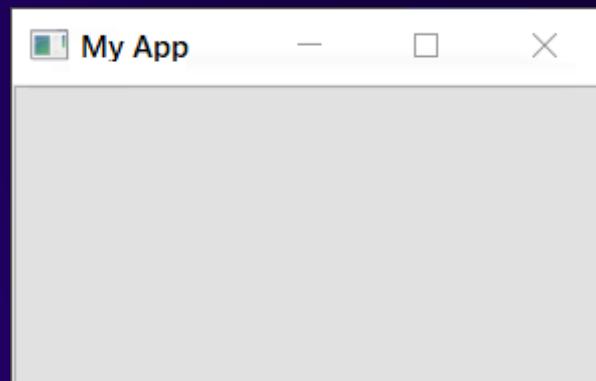
- `QString(const char *str);`
 - Constructs a string initialized with the 8-bit string str.
- `QString();`
 - Constructs a null string. Null strings are also empty.
- `int length() const;`
 - Returns the number of characters in this string. Equivalent to size().
- `void clear();`
 - Clears the contents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

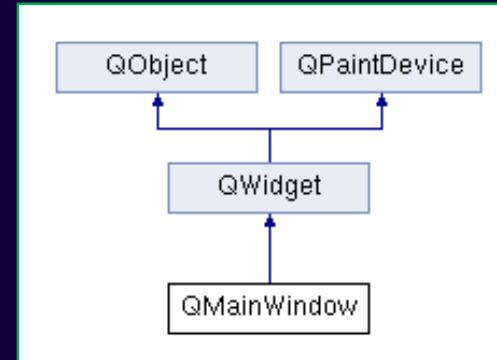


Header:	#include <QWidget>
qmake:	QT += widgets

- Note: Widget will be deleted automatically when closed. This behaviour can be modified using the following function: `widget.setAttribute(Qt::WA_DeleteOnClose, false);`

QMainWindow

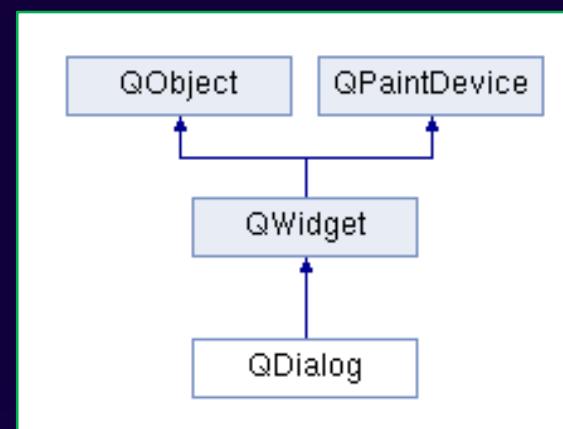
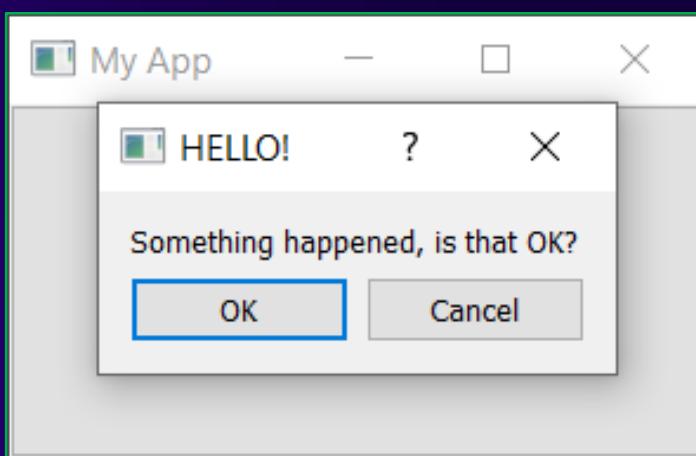
- The QMainWindow class provides a main application window.
- QMainWindow has its own layout to which you can add QToolBars, QDockWidgets, a QMenuBar, and a QStatusBar. The layout has a center area that can be occupied by any kind of widget.



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

QDialog

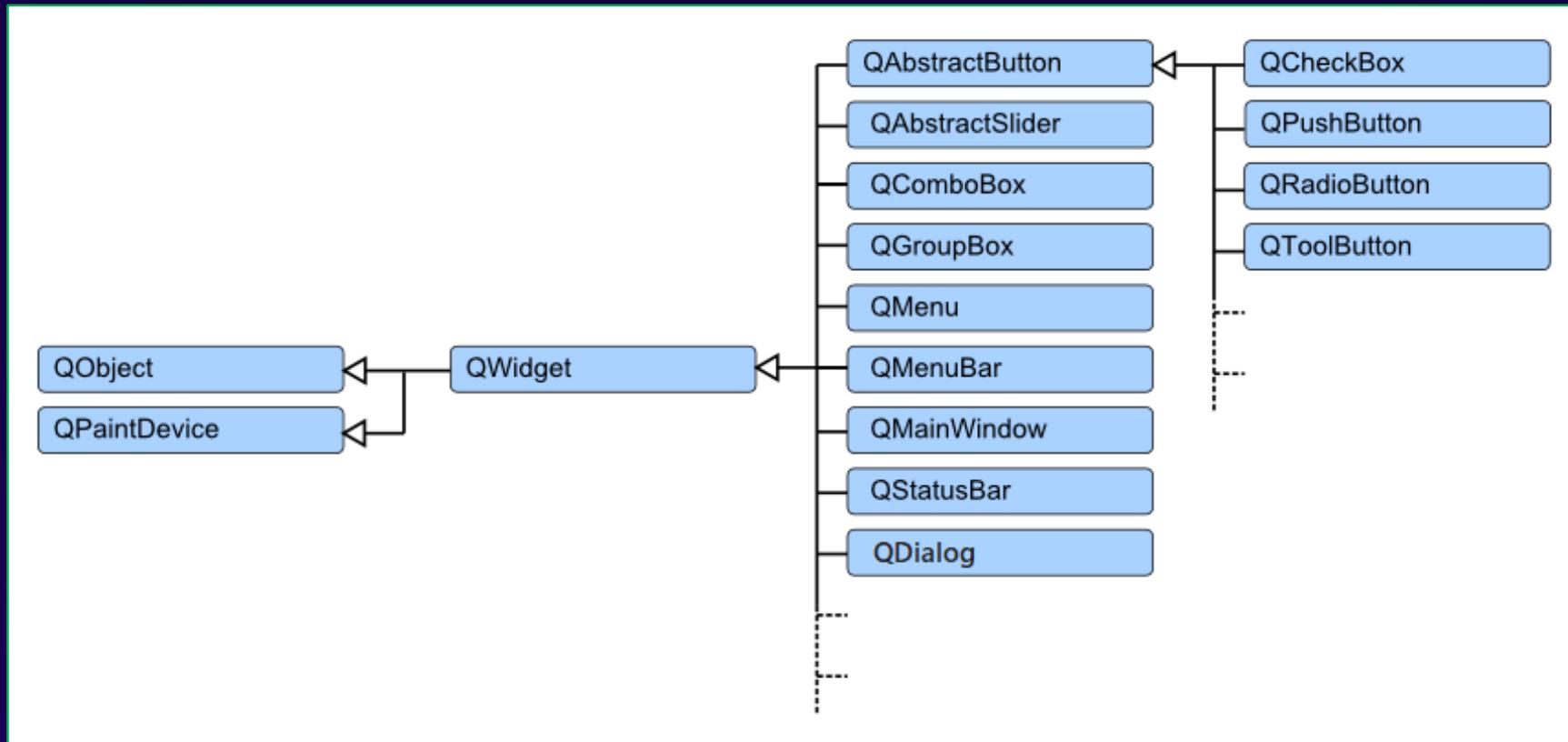
- The QDialog class is the base class of dialog windows.
- A dialog window is a top-level window mostly used for short-term tasks and brief communications with the user. QDialogs may be modal or modeless. QDialogs can provide a return value, and they can have default buttons.
- Modal Dialogs:
 - A modal 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

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

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

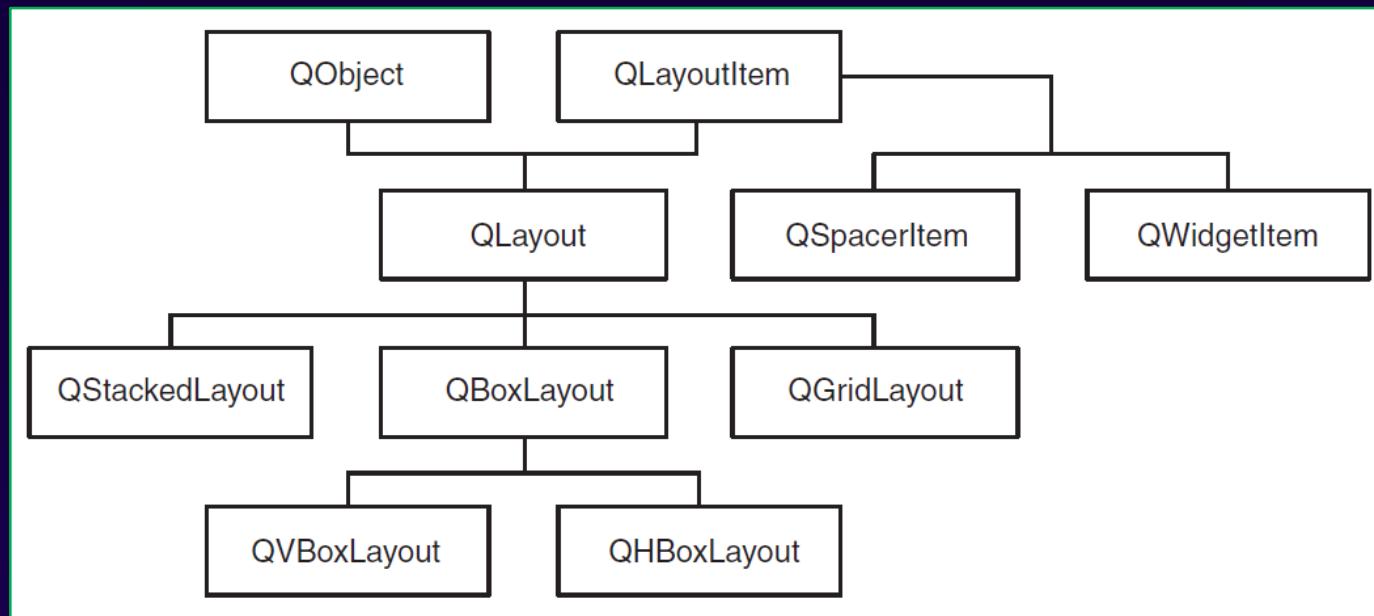
- `void textChanged(const QString &text);`
 - This signal is emitted whenever the text changes. The text argument is the new text.

Header: `#include <QLineEdit>`

qmake: `QT += widgets`

QLayout

- The Qt layout system provides a simple and powerful way of automatically arranging child widgets within a widget to ensure that they make good use of the available space.
- The QLayout class is the base class of geometry managers.



Methods

- `QLayout::QLayout(QWidget *parent = nullptr);`
 - Constructs a new top-level QLayout, with parent parent.
 - The layout is set directly as the top-level layout for parent. There can be only one top-level layout for a widget. It is returned by `QWidget::layout()`.
 - If parent is `nullptr`, then you must insert this layout into another layout, or set it as a widget's layout using `QWidget::setLayout()`.
 - Note: The constructor takes a parent of `QWidget` type to set its layout, but the `setParent` method takes an object (layer) to make the current layer its child.

How to use

- Correct

```
1  QVBoxLayout *verticalLayout;
2  QPushButton *pushButton;
3  QHBoxLayout *horizontalLayout;
4  QPushButton *pushButton_2;
5  QPushButton *pushButton_3;
6
7  verticalLayout = new QVBoxLayout(parent);
8  pushButton = new QPushButton(parent);
9  verticalLayout->addWidget(pushButton);
10
11 horizontalLayout = new QHBoxLayout();
12 pushButton_2 = new QPushButton(parent);
13 horizontalLayout->addWidget(pushButton_2);
14
15 pushButton_3 = new QPushButton(parent);
16 horizontalLayout->addWidget(pushButton_3);
17 verticalLayout->addLayout(horizontalLayout);
```

```
1  QVBoxLayout *verticalLayout;
2  QPushButton *pushButton;
3
4  verticalLayout = new QVBoxLayout();
5  pushButton = new QPushButton(this);
6
7  verticalLayout->addWidget(pushButton);
8  this->setLayout(verticalLayout);
```

```
1  QVBoxLayout *verticalLayout;
2  QPushButton *pushButton;
3
4  verticalLayout = new QVBoxLayout(this);
5  pushButton = new QPushButton(this);
6
7  verticalLayout->addWidget(pushButton);
```

How to use

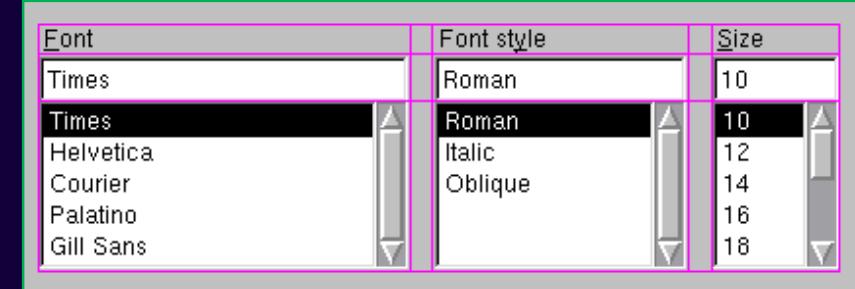
- Incorrect

```
1 QBoxLayout *verticalLayout;
2 QPushButton *pushButton;
3
4 verticalLayout = new QVBoxLayout;
5 pushButton = new QPushButton(this);
6
7 verticalLayout->addWidget(pushButton);
8 verticalLayout->setParent(this);
```

- Error:
 - `QLayout::parentWidget`: A layout can only have another layout as a parent.
 - **The program has unexpectedly finished.**

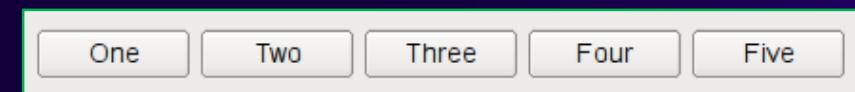
QGridLayout

- The QGridLayout class lays out widgets in a grid.



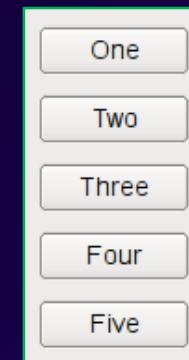
QHBoxLayout

- The QHBoxLayout class lines up widgets horizontally.



QVBoxLayout

- The QVBoxLayout class lines up widgets vertically.



QMessageBox

- The QMessageBox class provides a modal dialog for informing the user or for asking the user a question and receiving an answer.

Methods

- `QMessageBox(QMessageBox::Icon icon, const QString &title, const QString &text, QMessageBox::StandardButtons buttons = NoButton, QWidget *parent = nullptr, Qt::WindowFlags f = Qt::Dialog | Qt::MSWindowsFixedSizeDialogHint);`
 - Constructs a message box with the given icon, title, text, and standard buttons.
- `QMessageBox::QMessageBox(QWidget *parent = nullptr);`
 - Constructs a message box with no text and no buttons. parent is passed to the QDialog constructor.

Header: #include <QMessageBox>

qmake: QT += widgets

Methods

- `QMessageBox::Function(QWidget *parent, const QString &title, const QString &text, QMessageBox::StandardButtons buttons = Ok, QMessageBox::StandardButton defaultButton = NoButton);`
 - **Function:**
 - **critical:** Opens a critical message box with the given title and text in front of the specified parent widget.
 - **information:** Opens an information message box with the given title and text in front of the specified parent widget.
 - **warning:** Opens a warning message box with the given title and text in front of the specified parent widget.

Methods

- `QMessageBox::StandardButton QMessageBox::question(QWidget *parent, const QString &title, const QString &text, QMessageBox::StandardButtons buttons = StandardButtons(Yes | No), QMessageBox::StandardButton defaultButton = NoButton);`
 - Opens a question message box with the given title and text in front of the specified parent widget.
- `void QMessageBox::about(QWidget *parent, const QString &title, const QString &text)`
 - Displays a simple about box with title title and text text. The about box's parent is parent.

Separator

- Windows and Unix systems use different conventions for the file separator.
 - **"\" in Windows, "/" in Unix**
- Windows:
C:\Program Files\Java\jre6\bin;C:\MATLAB;C:\Some\Other\Path\bin;
- Unix:
/usr/bin:/usr/local/bin:/home/billybob/xmlbeans/bin

Absolute vs Relative Path

- You can do this by using relative path names instead. The relative path will find the file assuming you run the program directory.
- Absolute Path:
C:\Documents and Settings\billybob\My Documents\data\foo.txt
- Relative Path:
data/foo.txt

Case Sensitive Names

- Not all operating systems have case sensitive file systems. For example, Windows considers "FOO.txt" and "foo.txt" as the same. So avoid using file names that are case sensitive.
- Linux file system treats file and directory names as case-sensitive. FOO.txt and foo.txt will be treated as distinct files.

Special Characters

- The backslash (\) escape character turns special characters into string characters.

Escape character	Result	Description
\'	'	Single quote
\"	"	Double quote
\\	\	Backslash
\n	New Line	-
\t	Tab	-

- Example:
 - string txt = "We are the so-called \"Vikings\" from the north.;"
 - string txt = "It\'s alright.;"
 - string txt = "C:\\a\\b\\c.txt";

Line Breaks

- The simple fact is that it is different for all operating systems. There is no "universal" newline. The best you can do is be aware of the differences.
 - For Windows, it is CRLF
 - For UNIX, it is LF
 - For MAC (up through version 9) it was CR
 - For MAC OS X, it is LF
- The new line character (\n or endl) is translated into the equivalent character according to the type of operating system and compiler.

Character Name	Char	Decimal
Line Feed	LF	10
Carriage Return	CR	13

QDebug

- qDebug is used whenever the developer needs to write out debugging or tracing information to a device, file, string or console.

Methods

- `qDebug(const char *message, ...);`

Example

- `qDebug("Items in list: %d", myList.size());`
- `qDebug() << "Brush:" << myQBrush << "Other value:" << i;`

Header: #include <QDebug>

qmake: QT += core

File Types

- **Text File**

A text file stores data in the form of alphabets, digits and other special symbols by storing their ASCII values and are in a human readable format. For example, any file with a .txt, .c, etc extension.

- **Binary File**

A binary file contains a sequence or a collection of bytes which are not in a human readable format. For example, files with .exe, .mp3, etc extension. It represents custom data.

- There are three major differences between the two, i.e., Handling of newlines, storage of numbers, encoded and representation of EOF(End of File).

Binary vs Text files

Text Files vs Binary Files

- text files contain "text", usually in ASCII
- everything else is a binary file
- 1234 vs "1234"

byte 0	byte 1	byte 2	byte 3
00000000	00000000	00000100	11010010
0	0	4	210
49	50	51	52

QFile

- QFile is an I/O device for reading and writing text and binary files and resources. A QFile may be used by itself or, more conveniently, with a QTextStream or QDataStream.

Methods

- **QFile(const QString &name, QObject *parent);**
 - Constructs a new file object with the given parent to represent the file with the specified name.
- **QFile(QObject *parent);**
 - Constructs a new file object with the given parent.
- **QFile(const QString &name);**
 - Constructs a new file object to represent the file with the given name.
- **QFile();**
 - Constructs a QFile object.

Header: #include <QFile>

qmake: QT += core

Methods

- `bool copy(const QString &newName);`
 - Copies the file named `fileName()` to `newName`.
 - This file is closed before it is copied.
 - If the copied file is a symbolic link (`symlink`), the file it refers to is copied, not the link itself. With the exception of permissions, which are copied, no other file metadata is copied.
 - Constructs a new file object with the given parent to represent the file with the specified name.
- `bool exists() const;`
 - Returns true if the file specified by `fileName()` exists; otherwise returns false.
- `bool remove();`
 - Removes the file specified by `fileName()`. Returns true if successful; otherwise returns false.
 - The file is closed before it is removed.
- `bool rename(const QString &newName);`
 - Renames the file currently specified by `fileName()` to `newName`. Returns true if successful; otherwise returns false. The file is closed before it is renamed.

Methods

- `bool open(QIODevice::OpenMode mode);`
 - Opens the file using OpenMode mode, returning true if successful; otherwise false.
 - The mode must be QIODevice::ReadOnly, QIODevice::WriteOnly, or QIODevice::ReadWrite. It may also have additional flags, such as QIODevice::Text and QIODevice::Unbuffered.
- `qint64 write(const char *data, qint64 maxSize);`
 - Writes at most maxSize bytes of data from data to the device.
- `qint64 read(char *data, qint64 maxSize);`
 - Reads at most maxSize bytes from the device into data, and returns the number of bytes read.
- `qint64 readLine(char *data, qint64 maxSize);`
 - This function reads a line of ASCII characters from the device, up to a maximum of maxSize - 1 bytes, stores the characters in data, and returns the number of bytes read. If a line could not be read but no error occurred, this function returns 0. If an error occurs, this function returns the length of what could be read, or -1 if nothing was read.
- `void close();`
 - Calls QFileDevice::flush() and closes the file. Errors from flush are ignored.

QTextStream

- QTextStream can operate on a QIODevice, a QByteArray or a QString. Using QTextStream's streaming operators, you can conveniently read and write words, lines and numbers. For generating text, QTextStream supports formatting options for field padding and alignment, and formatting of numbers.

```
Header: #include <QTextStream>
qmake: QT += core
```

QDataStream

- A data stream is a binary stream of encoded information which is 100% independent of the host computer's operating system, CPU or byte order. For example, a data stream that is written by a PC under Windows can be read by a Sun SPARC running Solaris.

```
Header: #include <QDataStream>
qmake: QT += core
```

QByteArray

- The QByteArray class provides an array of bytes.

Header: #include <QByteArray>

qmake: QT += core

QChar

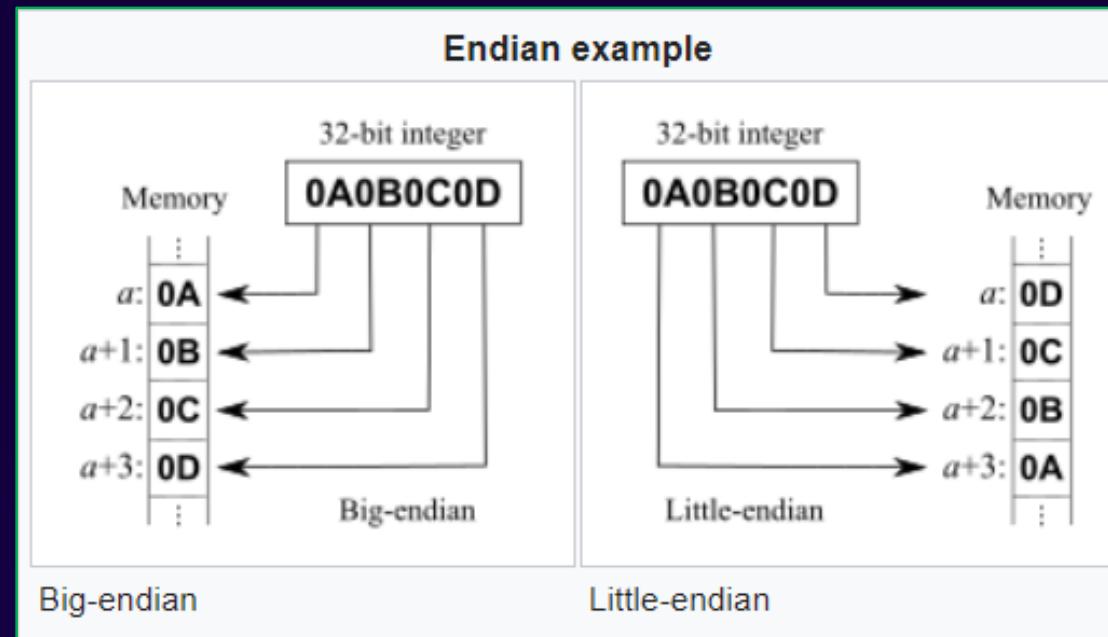
- The QChar class provides a 16-bit Unicode character.

Header: #include <QChar>

qmake: QT += core

LittleEndian vs. BigEndian architecture

- Computers store data in memory in binary. Specifically, little-endian is when the least significant bytes are stored before the more significant bytes, and big-endian is when the most significant bytes are stored before the less significant bytes.



LittleEndian vs. BigEndian architecture

Little endian memory: Data structure		Big endian memory: Data structure	
Data structure fields	Byte organization	Data structure fields	Byte organization
elem8_1	0x11	elem8_1	0x11
elem8_2	0x22	elem8_2	0x22
elem8_3	0x33	elem8_3	0x33
elem8_4	0x44	elem8_4	0x44
elem16_1	0x55	elem16_1	0x66
elem16_2	0x66	elem16_2	0x55
elem32	0x77	elem32	0x88
	0x88		0x77
	0x99		0xCC
	0xAA		0xBB
	0xBB		0xAA
	0xCC		0x99

Example

- 0x12345678 storing this 4-byte value into memory
 - little-endian: 78 56 34 12
 - big-endian: 12 34 56 78
- 0x123456789abcdef0 storing this 8-byte value into memory
 - little-endian (value is at memory address 0x00): f0 de bc 9a 78 56 34 12
 - big-endian (value is at memory address 0x00): 12 34 56 78 9a bc de f0
- `int a[] = {0x12345678, 0x9abcdef0};`
 - little-endian:
 - 0x00: 78 56 34 12
 - 0x04: f0 de bc 9a
 - big-endian: 12 34 56 78
 - 0x00: 12 34 56 78
 - 0x04: 9a bc de f0

Example

- `char s[] = {0x12, 0x34, 0x56, 0x78, 0x9a, 0xbc, 0xde, 0xf0};`
 - **little-endian:**
 - 0x00: 12
 - 0x01: 34
 - 0x02: 56
 - 0x03: 78
 - 0x04: 9a
 - 0x05: bc
 - 0x06: de
 - 0x07: f0
 - **big-endian:**
 - 0x00: 12 34 56 78 9a bc de f0

Example

```
1 QDataStream dataStream(header);
2 dataStream.setByteOrder(QDataStream::BigEndian);
3 quint16 low, high;
4 quint32 temp;
5 dataStream >> low >> high; temp = (high << 32) | low;
```

Example

```
1 #include <QtCore>
2
3 int main()
4 {
5     QFile file("file.txt");
6
7     if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
8     {
9         return -1;
10    }
11
12    QTextStream in(&file);
13
14    in.setCodec("UTF-8");
15
16    while(!in.atEnd())
17    {
18        QString line = in.readLine();
19        qDebug() << line;
20    }
21 }
```

QStringList

- The QStringList class provides a list of strings.

Methods

- QStringList(const QString &str);
 - Constructs a string list that contains the given string, str.
- QStringList();
 - Constructs an empty string list.
- Void sort(Qt::CaseSensitivity cs = Qt::CaseSensitive);
 - Sorts the list of strings in ascending order. If cs is Qt::CaseSensitive (the default), the string comparison is case sensitive; otherwise the comparison is case insensitive.
- Int removeDuplicates();
 - This function removes duplicate entries from a list. The entries do not have to be sorted. They will retain their original order.

Header: #include <QStringList>

qmake: QT += core

QLocale

- The QLocale class converts between numbers and their string representations in various languages.

Methods

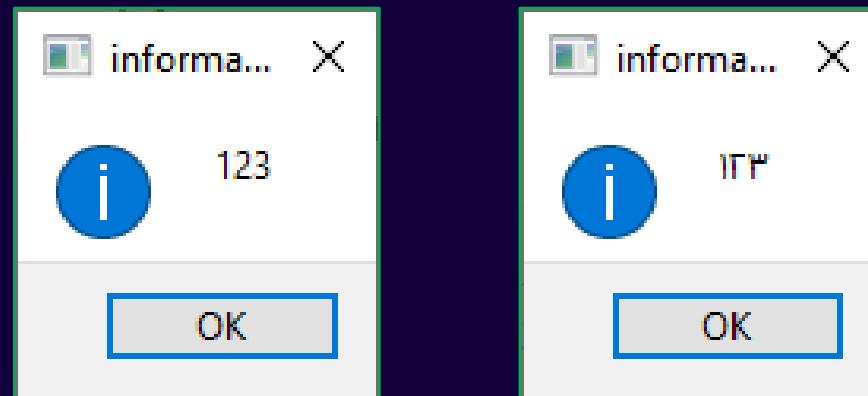
- QLocale(QLocale::Language language, QLocale::Script script, QLocale::Country country);**
 - Constructs a QLocale object with the specified language, script and country.
- QLocale(const QString &name);**
 - Constructs a QLocale object with the specified name, which has the format "language[_script][_country][.codeset][@modifier]" or "C".

Header: #include <QLocale>

qmake: QT += core

QLocale

```
1 //QLocale local(QLocale::Persian, QLocale::Iran);
2 QLocale local("en_US");
3 //QLocale local("fa_IR");
4 QMessageBox::information(nullptr, "information", local.toString(123));
```



QRegularExpression

- Regular expressions, or regexps, are a very powerful tool to handle strings and texts. This is useful in many contexts, e.g:
 - Validation
 - Searching
 - Search and Replace
 - String Splitting

Methods

- `QRegularExpression(const QString &pattern, QRegularExpression::PatternOptions options = NoPatternOption);`
 - Constructs a `QRegularExpression` object using the given pattern as pattern and the options as the pattern options.
- `QRegularExpressionMatch match(const QString &subject, int offset = 0, QRegularExpression::MatchType matchType = NormalMatch, QRegularExpression::MatchOptions matchOptions = NoMatchOption) const;`
 - Attempts to match the regular expression against the given subject string.

Header: `#include <QRegularExpression>`

qmake: `QT += core`

Supported syntax

\a	BELL
\A	beginning of input
\b inside a [set]	BACKSPACE
\b outside a [set]	on a word boundary
\B	not on a word boundary
\cX	ASCII control character X
\d	digit
\D	non digit
\e	ESCAPE
\E	end of \Q ... \E quoting
\f	FORM FEED
\G	end of previous match
\n	LINE FEED
\N{x}	UNICODE CHARACTER NAME x
\p{x}	UNICODE PROPERTY NAME x

\P{x}	UNICODE PROPERTY NAME not x
\Q	start of \Q ... \E quoting
\r	CARRIAGE RETURN
\s	white space
\S	non white space
\t	HORIZONTAL TAB
\uhhhh	U+hhhh (between U+0000 and U+FFFF)
\Uhhhhhhhh	U+hhhhhhhh (between U+00000000 and U+0010FFFF)
\v	VERTICAL TAB
\w	word character
\W	non word character
\x{hhhh}	U+hhhh
\xhhh	U+hhhh
\X	grapheme cluster
\Z	end of input (or before the final)

\z	end of input
\n	n-th backreference
\0ooo	ASCII/Latin-1 character 0ooo
.	any character but newlines
^	line beginning
\$	line end
\	quote the following symbol
[pattern]	set

Operators

*	match 0 or more times
+	match 1 or more times
?	match 0 or 1 times
{n}	match n times
{n,}	match n or more times
{n,m}	match between n and m times
*?	match 0 or more times, not greedy
?	match 1 or more times, not greedy
??	match 0 or 1 times, not greedy
{n}?	match n times
{n,}?	match n or more times, not greedy
{n,m}?	match between n and m times, not greedy
*+	match 0 or more times, possessive
+	match 1 or more times, possessive
?	match 0 or 1 times, possessive
{n}+	match n times

{n,}+	match n or more times, possessive
{n,m}+	match between n and m times, possessive
(...)	capturing group
(?: ...)	group
(?> ...)	atomic grouping
(?# ...)	comment
(?= ...)	look-ahead assertion
(?! ...)	negative look-ahead assertion
(?<= ...)	look-behind assertion
(?<! ...)	negative look-behind assertion
(?flags: ...)	flags change
(?flags)	flags change
(?P<name> ...)	named capturing group
(?<name> ...)	named capturing group
(?'name' ...)	named capturing group
...)	branch reset

Flags

i	case insensitive
/m	multi-line
/s	dot matches anything
/x	ignore whitespace and comments
/U	minimal match

QRegExp

- The QRegExp class provides pattern matching using regular expressions.

Difference between QRegularExpression and QRegExp?

- The QRegularExpression class introduced in Qt 5 is a big improvement upon QRegExp, in terms of APIs offered, supported pattern syntax and speed of execution. The biggest difference is that QRegularExpression simply holds a regular expression, and it's not modified when a match is requested. Instead, a QRegularExpressionMatch object is returned, in order to check the result of a match and extract the captured substring. The same applies with global matching and QRegularExpressionMatchIterator.

Header: #include <QRegExp>

qmake: QT += core

QRegularExpressionValidator

QRegExpValidator

- QRegularExpressionValidator uses a regular expression (regexp) to determine whether an input string is Acceptable, Intermediate, or Invalid.

Methods

- **QRegularExpressionValidator(const QRegularExpression &re, QObject *parent = nullptr);**
 - Constructs a validator with a parent object that accepts all strings that match the regular expression re.
- **virtual QValidator::State validate(QString &input, int &pos) const override;**
 - Returns Acceptable if input is matched by the regular expression for this validator, Intermediate if it has matched partially (i.e. could be a valid match if additional valid characters are added), and Invalid if input is not matched.

Header: #include <QRegularExpressionValidator>

qmake: QT += gui

QTest

- Qt Test is a framework for unit testing Qt based applications and libraries. Qt Test provides all the functionality commonly found in unit testing frameworks as well as extensions for testing graphical user interfaces.

Creating a Test

- To create a test, subclass QObject and add one or more private slots to it. Each private slot is a test function in your test.
- you can define the following private slots that are not treated as test functions. When present, they will be executed by the testing framework and can be used to initialize and clean up either the entire test or the current test function.
 - initTestCase() will be called before the first test function is executed.
 - initTestCase_data() will be called to create a global test data table.
 - cleanupTestCase() will be called after the last test function was executed.
 - init() will be called before each test function is executed.
 - cleanup() will be called after every test function.

Header: #include <QTest>

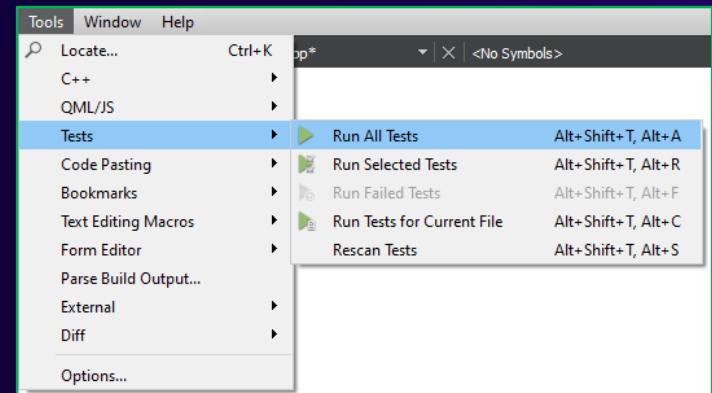
qmake: QT += testlib

Methods

- **QVERIFY(condition);**
 - The QVERIFY() macro checks whether the condition is true or not. If it is true, execution continues.
- **QVERIFY2(condition, message);**
 - The QVERIFY2() macro behaves exactly like QVERIFY(), except that it reports a message when condition is false. The message is a plain C string.
- **QCOMPARE(actual, expected);**
 - The QCOMPARE() macro compares an actual value to an expected value using the equality operator. If actual and expected match, execution continues.
- **int QTest::qExec(QObject *testObject, int argc = 0, char **argv = nullptr);**
 - Executes tests declared in testObject. In addition, the private slots initTestCase(), cleanupTestCase(), init() and cleanup() are executed if they exist;

Test execution methods

- Run the program (not debug)
- Using Qt Creator IDE
- Run the program through the command line



Test Results

Test summary: 5 passes, 1 fails.

● PASS	Executing test case TestFooClass
>	● PASS Executing test function initTestCase
>	● PASS Executing test function test_func_foo
>	● PASS Executing test function cleanupTestCase
● FAIL	Executing test case TestBarClass
>	● PASS Executing test function initTestCase
>	● FAIL Executing test function test_func_bar
>	● PASS Executing test function cleanupTestCase

main.cpp 47

main.cpp 51

```
C:\Users\...\Qt\Qt5.15.2\msvc2019\debug>TestUnit.exe
***** Start testing of TestFooClass *****
Config: Using QTest library 5.15.2, Qt 5.15.2 (i386-little_endian-ilp32 shared (dynamic) debug build; by MSVC 2019), windows 10
PASS : TestFooClass::initTestCase()
PASS : TestFooClass::test_func_foo()
PASS : TestFooClass::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped, 0 blacklisted, 2ms
***** Finished testing of TestFooClass *****
***** Start testing of TestBarClass *****
Config: Using QTest library 5.15.2, Qt 5.15.2 (i386-little_endian-ilp32 shared (dynamic) debug build; by MSVC 2019), windows 10
PASS : TestBarClass::initTestCase()
PASS : TestBarClass::test_func_bar()
PASS : TestBarClass::cleanupTestCase()
Totals: 3 passed, 0 failed, 0 skipped, 0 blacklisted, 1ms
***** Finished testing of TestBarClass *****
```

Data Driven Testing

- We will demonstrate how to execute a test multiple times with different test data. So far, we have hard coded the data we wanted to test into our test function. If we add more test data with Data Driven Testing.
- A test function's associated data function carries the same name, appended by `_data`. Our data function looks like this:
 - `void toUpper();`
 - `void toUpper_data();`
- First, we define the two elements of our test table using the `QTest::addColumn()` function: a test string, and the expected result of applying the `QString::toUpper()` function to that string.
- Then we add some data to the table using the `QTest::newRow()` function. Each set of data will become a separate row in the test table.

```
1 QCOMPARE(QString("hello").toUpper(), QString("HELLO"));
2 QCOMPARE(QString("Hello").toUpper(), QString("HELLO"));
3 QCOMPARE(QString("Hello").toUpper(), QString("HELLO"));
4 QCOMPARE(QString("HELLO").toUpper(), QString("HELLO"));
```

```
1 QTest::addColumn<QString>("string");
2 QTest::addColumn<QString>("result");
3
4 QTest::newRow("all lower") << "hello" << "HELLO";
6 QTest::newRow("mixed") << "Hello" << "HELLO";
7 QTest::newRow("all upper") << "HELLO" << "HELLO";
```

Simulating GUI Events

- Qt Test features some mechanisms to test graphical user interfaces. Instead of simulating native window system events, Qt Test sends internal Qt events. That means there are no side-effects on the machine the tests are running on.
- Simulate writing using the QTest::keyClicks() function.

Replaying GUI Events

- We will show how to simulate a GUI event, and how to store a series of GUI events as well as replay them on a widget.
- A QTestEventList can be populated with GUI events that can be stored as test data for later usage, or be replayed on any QWidget.

Writing a Benchmark

- To create a benchmark we extend a test function with a QBENCHMARK macro. A benchmark test function will then typically consist of setup code and a QBENCHMARK macro that contains the code to be measured.

Task

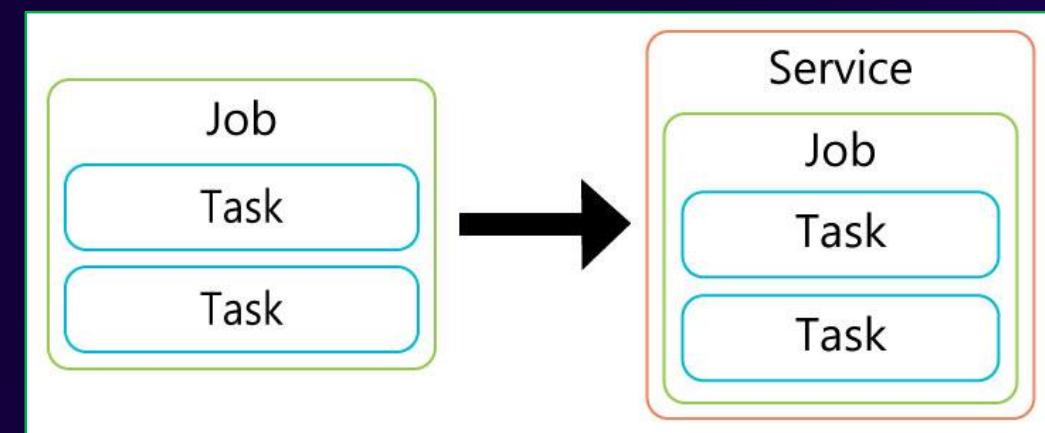
- Task is a unit of work being executed. Task in Operating System may be synonymous with process. A task is a subpart of a job. Tasks combine to form a job.

Job

- A job is a complete unit of work under execution. A job consists of many tasks which in turn, consist of many processes. A job is a series of tasks in a batch mode. Programs are written to execute a job.

Service

- A Service is a collection of jobs.

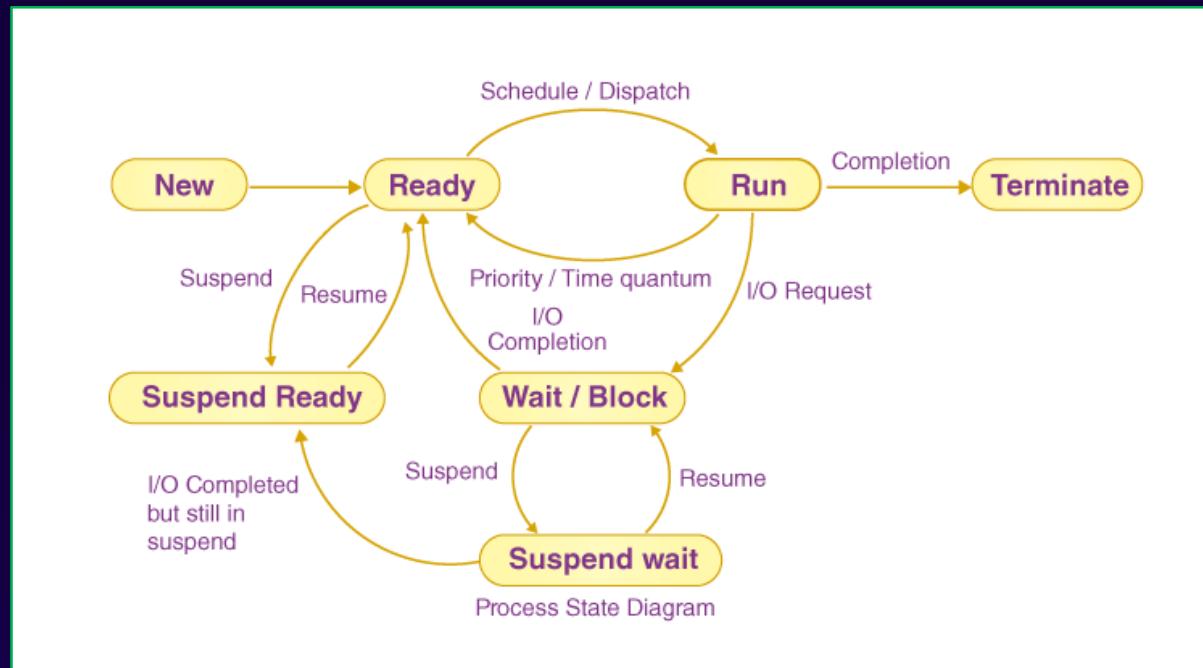


Process

- The process is a program under execution. A program can be defined as a set of instructions.

Process vs Program

- The program is a passive entity and the process is an active entity. When we execute a program, it remains on the hard drive of our system and when this program comes into the main memory it becomes a process. The process can be present on a hard drive, memory or CPU.



What are the Process States in Operating System?

- **New State**

When a program in secondary memory is started for execution, the process is said to be in a new state.

- **Ready State**

After being loaded into the main memory and ready for execution, a process transitions from a new to a ready state. The process will now be in the ready state, waiting for the processor to execute it. Many processes may be in the ready stage in a multiprogramming environment.

- **Run State**

After being allotted the CPU for execution, a process passes from the ready state to the run state.

- **Terminate State**

When a process's execution is finished, it goes from the run state to the terminate state. The operating system deletes the process control box (or PCB) after it enters the terminate state.

- **Block or Wait State**

If a process requires an Input/Output operation or a blocked resource during execution, it changes from run to block or the wait state.

What are the Process States in Operating System?

- **Suspend Ready State**

If a process with a higher priority needs to be executed while the main memory is full, the process goes from ready to suspend ready state. Moving a lower-priority process from the ready state to the suspend ready state frees up space in the ready state for a higher-priority process.

Until the main memory becomes available, the process stays in the suspend-ready state. The process is brought to its ready state when the main memory becomes accessible.

- **Suspend Wait State**

If a process with a higher priority needs to be executed while the main memory is full, the process goes from the wait state to the suspend wait state. Moving a lower-priority process from the wait state to the suspend wait state frees up space in the ready state for a higher-priority process.

The process gets moved to the suspend-ready state once the resource becomes accessible. The process is shifted to the ready state once the main memory is available.

Thread

- A thread is a path of execution within a process. A process can contain multiple threads.
- A thread is also known as lightweight process. The idea is to achieve parallelism by dividing a process into multiple threads. For example, in a browser, multiple tabs can be different threads. MS Word uses multiple threads: one thread to format the text, another thread to process inputs, etc.

Process vs Thread

- The primary difference is that threads within the same process run in a shared memory space, while processes run in separate memory spaces.
- Threads are not independent of one another like processes are, and as a result threads share with other threads their code section, data section, and OS resources (like open files and signals). But, like process, a thread has its own program counter (PC), register set, and stack space.

Advantages of Thread over Process

- Responsiveness: If the process is divided into multiple threads, if one thread completes its execution, then its output can be immediately returned.
- Faster context switch: Context switch time between threads is lower compared to process context switch. Process context switching requires more overhead from the CPU.
- Effective utilization of multiprocessor system: If we have multiple threads in a single process, then we can schedule multiple threads on multiple processor. This will make process execution faster.
- Resource sharing: Resources like code, data, and files can be shared among all threads within a process.

Note: stack and registers can't be shared among the threads. Each thread has its own stack and registers.

GUI Thread and Worker Thread (User Level thread (ULT) & Kernel Level Thread (KLT))

- Each program has one thread when it is started. This thread is called the "main thread" (also known as the "GUI thread" in Qt applications). The Qt GUI must run in this thread. All widgets and several related classes, for example [QPixmap](#), don't work in secondary threads. A secondary thread is commonly referred to as a "worker thread" because it is used to offload processing work from the main thread.

There are basically two use cases for threads:

- Make processing faster by making use of multicore processors.
- Keep the GUI thread or other time critical threads responsive by offloading long lasting processing or blocking calls to other threads.

Warning: Developers need to be very careful with threads. It is easy to start other threads, but very hard to ensure that all shared data remains consistent. Problems are often hard to find because they may only show up once in a while or only on specific hardware configurations. Before creating threads to solve certain problems, possible alternatives should be considered.

When to Use Alternatives to Threads

Alternatives	Comments
QEventLoop::processEvents()	Calling <code>QEventLoop::processEvents()</code> repeatedly during a time-consuming calculation prevents GUI blocking. However, this solution doesn't scale well because the call to <code>processEvents()</code> may occur too often, or not often enough, depending on hardware.
QTimer	Background processing can sometimes be done conveniently using a timer to schedule execution of a slot at some point in the future. A timer with an interval of 0 will time out as soon as there are no more events to process.
QSocketNotifier QNetworkAccessManager QIODevice::readyRead()	This is an alternative to having one or multiple threads, each with a blocking read on a slow network connection. As long as the calculation in response to a chunk of network data can be executed quickly, this reactive design is better than synchronous waiting in threads. Reactive design is less error prone and energy efficient than threading. In many cases there are also performance benefits.

Program Counter (PC)

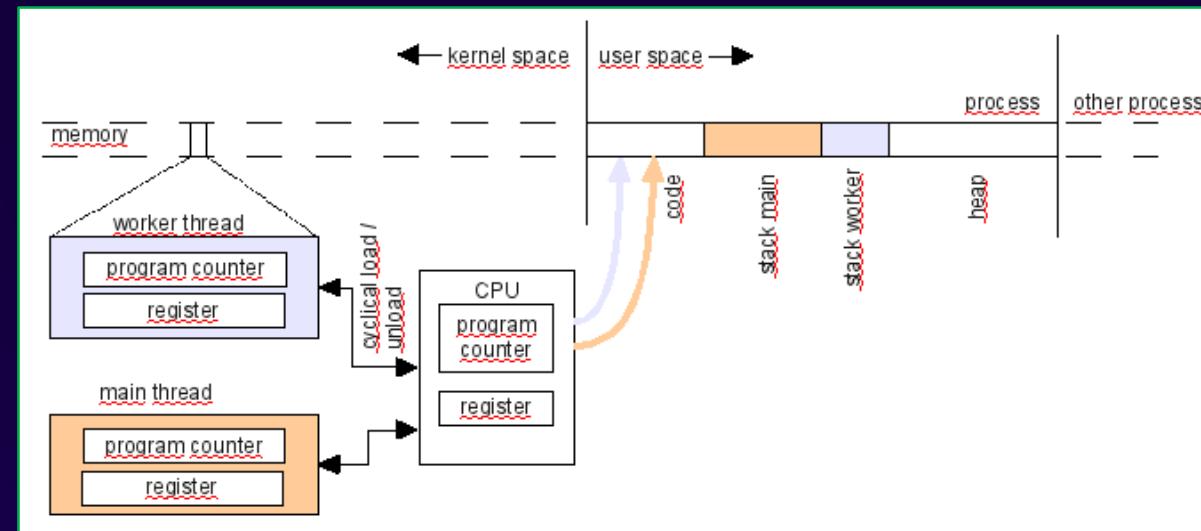
- A program counter (PC) is a CPU register in the computer processor which has the address of the next instruction to be executed from memory.

Registers

- The Registers are very fast computer memory which are used to execute programs and operations efficiently.

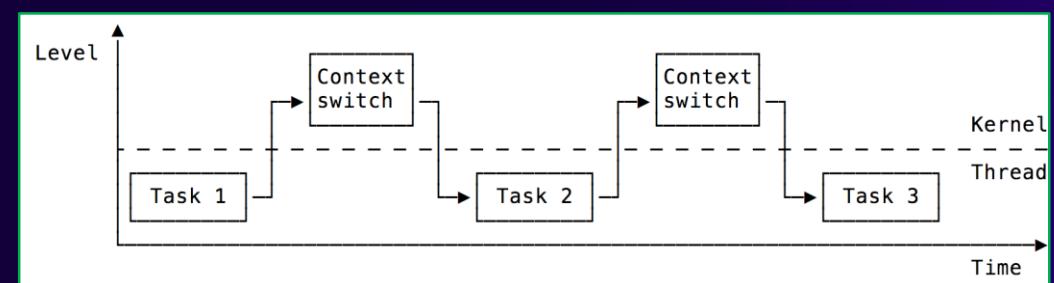
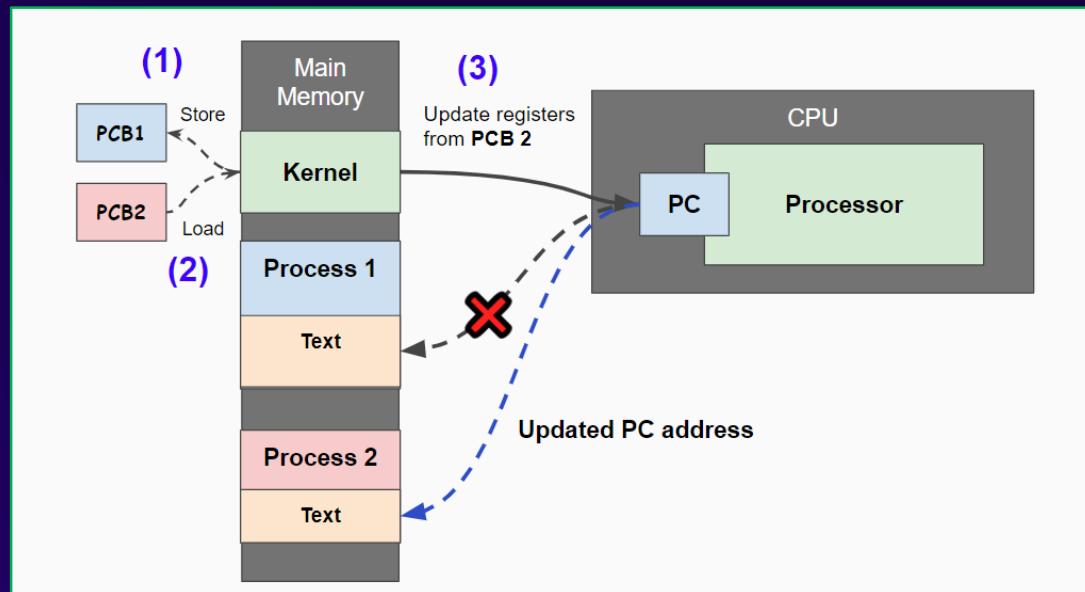
Each thread has its own stack, which means each thread has its own call history and local variables. Unlike processes, threads share the same address space. The following diagram shows how the building blocks of threads are located in memory. Program counter and registers of inactive threads are typically kept in kernel space. There is a shared copy of the code and a separate stack for each thread.

Warning: If two threads have a pointer to the same object, it is possible that both threads will access that object at the same time and this can potentially destroy the object's integrity.



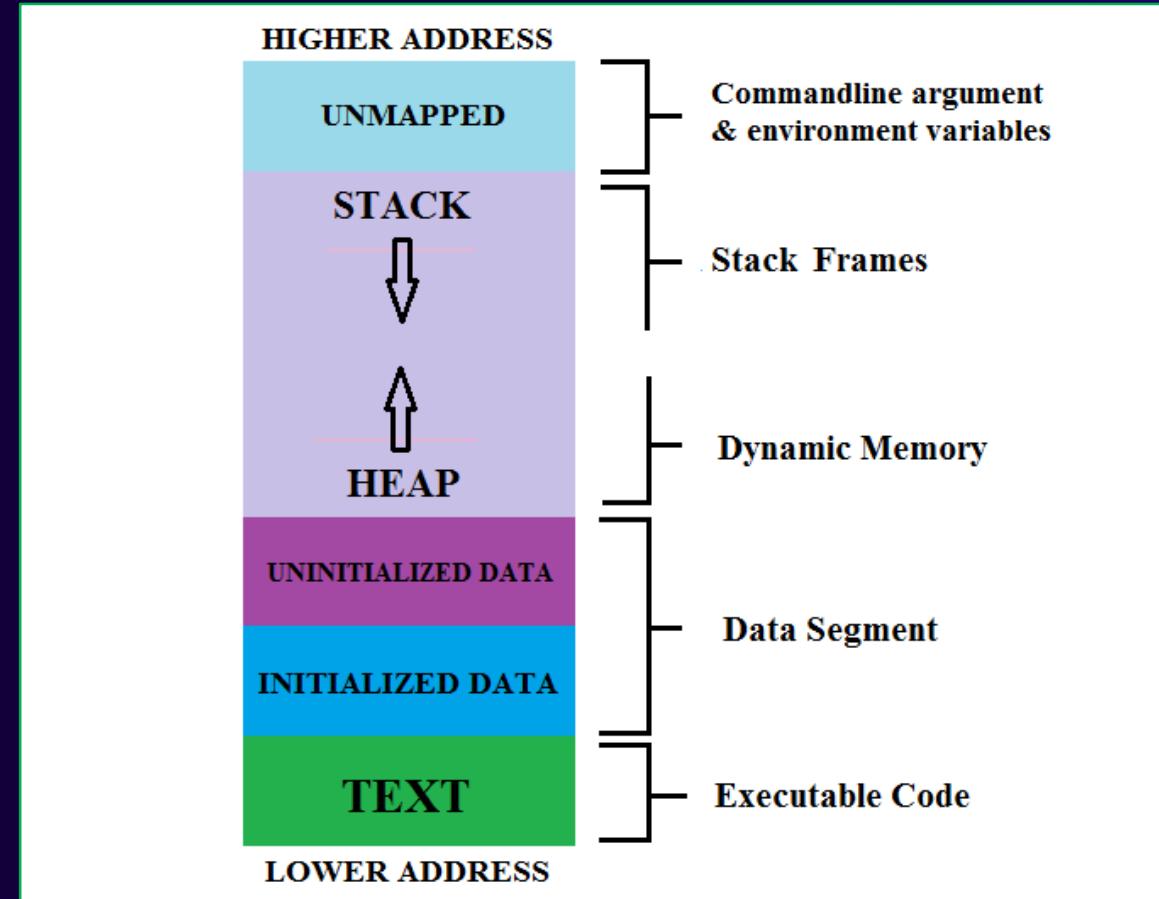
Context Switch

- context switch is the process of storing the state of a process or thread, so that it can be restored and resume execution at a later point, and then restoring a different, previously saved, state. This allows multiple processes to share a single central processing unit (CPU), and is an essential feature of a multitasking operating system.



Memory Layout of C Program

- **Text segment**
- **Data segment**
- **Heap segment**
- **Stack segment**
- **Unmapped or reserved**



Text Segment

- Text segment contain executable instructions of your C program, its also called code segment.

Data Segment

- There are two sub section of this segment called initialized & uninitialized data segment:
 - Initialized data:- It contains both static and global data that are initialized with non-zero values.
 - Uninitialized data segment (BSS) contains all global and static variables that are initialized to zero or do not have explicit initialization in source code.

Heap Segment

- The heap segment is area where dynamically allocated memory (allocated by malloc(), calloc(), realloc() and new for C++) resides.

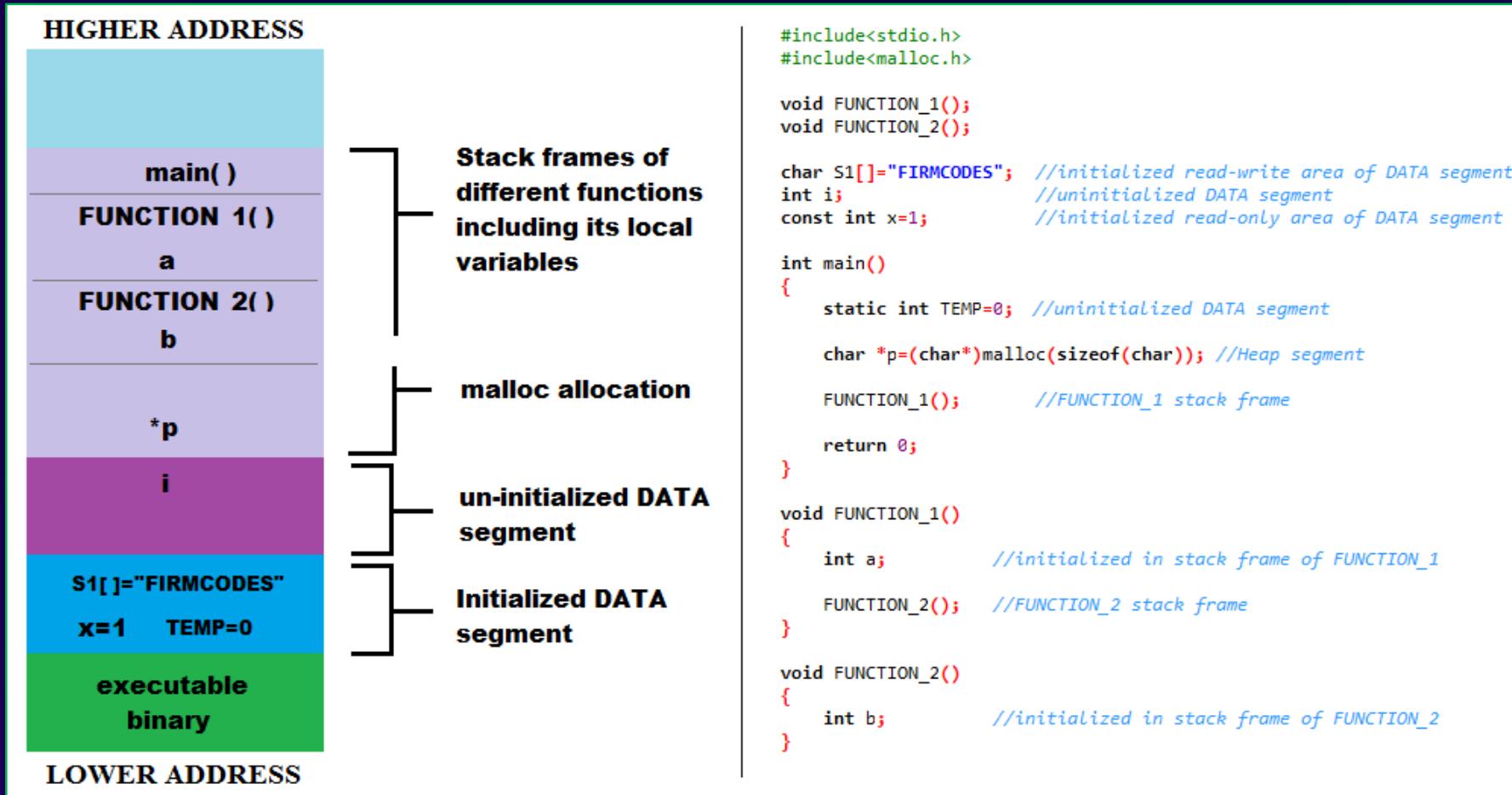
Stack Segment

- The stack segment is area where local variables are stored. By saying local variable means that all those variables which are declared in every function including main() in your C program.
- When we call any function, stack frame is created and when function returns, stack frame is destroyed including all local variables of that particular function.
- Stack frame contain some data like return address, arguments passed to it, local variables, and any other information needed by the invoked function.

Unmapped or reserved segment

- Unmapped or reserved segment contain command line arguments and other program related data like lower address-higher address of executable image, etc.

Understanding Practically



QThread: Low-Level API with Optional Event Loops

- QThread is the foundation of all thread control in Qt. Each QThread instance represents and controls one thread.
- QThread can either be instantiated directly or subclassed. Instantiating a QThread provides a parallel event loop, allowing QObject slots to be invoked in a secondary thread. Subclassing a QThread allows the application to initialize the new thread before starting its event loop, or to run parallel code without an event loop.

QThreadPool and QRunnable: Reusing Threads

- Creating and destroying threads frequently can be expensive. To reduce this overhead, existing threads can be reused for new tasks. QThreadPool is a collection of reusable QThreads.

Qt Concurrent: Using a High-level API

- The Qt Concurrent module provides high-level functions that deal with some common parallel computation patterns: map, filter, and reduce. Unlike using QThread and QRunnable, these functions never require the use of low-level threading primitives such as mutexes or semaphores.

void QObject::deleteLater()

- Schedules this object for deletion.
- The object will be deleted when control returns to the event loop. If the event loop is not running when this function is called (e.g. `deleteLater()` is called on an object before `QCoreApplication::exec()`), the object will be deleted once the event loop is started. If `deleteLater()` is called after the main event loop has stopped, the object will not be deleted. Since Qt 4.8, if `deleteLater()` is called on an object that lives in a thread with no running event loop, the object will be destroyed when the thread finishes.
- Note that entering and leaving a new event loop (e.g., by opening a modal dialog) will not perform the deferred deletion; for the object to be deleted, the control must return to the event loop from which `deleteLater()` was called. This does not apply to objects deleted while a previous, nested event loop was still running: the Qt event loop will delete those objects as soon as the new nested event loop starts.

Note: It is safe to call this function more than once; when the first deferred deletion event is delivered, any pending events for the object are removed from the event queue.

QMetaObject Struct

- The QMetaObject class contains meta-information about Qt objects.
- The Qt Meta-Object System in Qt is responsible for the signals and slots inter-object communication mechanism, runtime type information, and the Qt property system. A single QMetaObject instance is created for each QObject subclass that is used in an application, and this instance stores all the meta-information for the QObject subclass.

QMetaObject::invokeMethod

- As you haven't specified a Qt::ConnectionType, the method will be invoked as Qt::AutoConnection, which means that it will be invoked synchronously (like a normal function call) if the object's thread affinity is to the current thread, and asynchronously otherwise. "Asynchronously" means that a QEvent is constructed and pushed onto the message queue, and will be processed when the event loop reaches it.
- The reason to use QMetaObject::invokeMethod if the recipient object might be in another thread is that attempting to call a slot directly on an object in another thread can lead to corruption or worse if it accesses or modifies non-thread-safe data.

QMetaObject::invokeMethod

- Invokes the member (a signal or a slot name) on the object obj. Returns true if the member could be invoked. Returns false if there is no such member or the parameters did not match.
- The invocation can be either synchronous or asynchronous, depending on type:
 - If type is Qt::DirectConnection, the member will be invoked immediately.
 - If type is Qt::QueuedConnection, a QEvent will be sent and the member is invoked as soon as the application enters the main event loop.
 - If type is Qt::BlockingQueuedConnection, the method will be invoked in the same way as for Qt::QueuedConnection, except that the current thread will block until the event is delivered. Using this connection type to communicate between objects in the same thread will lead to deadlocks.
 - If type is Qt::AutoConnection, the member is invoked synchronously if obj lives in the same thread as the caller; otherwise it will invoke the member asynchronously.

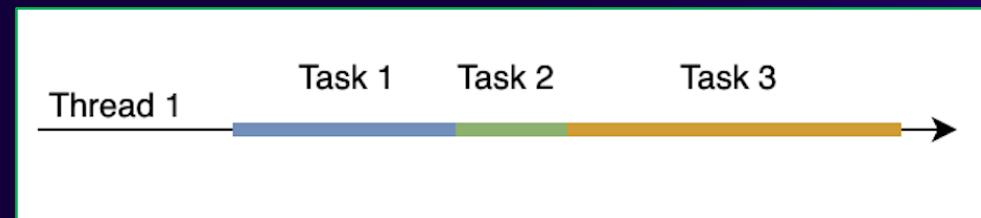
```
1 QMetaObject::invokeMethod(thread, "quit", Qt::QueuedConnection);
2
3 QString retVal;
4 QMetaObject::invokeMethod(obj, "compute", Qt::DirectConnection, Q_RETURN_ARG(QString, retVal), Q_ARG(QString,
5 "sqrt"), Q_ARG(int, 42), Q_ARG(double, 9.7));
```

Synchronous

- Implies that tasks will be executed one by one. A next task is started only after current task is finished. Task 3 is not started until Task 2 is finished.
 - **Single Thread + Sync - Sequential**

Usual execution. Pseudocode:

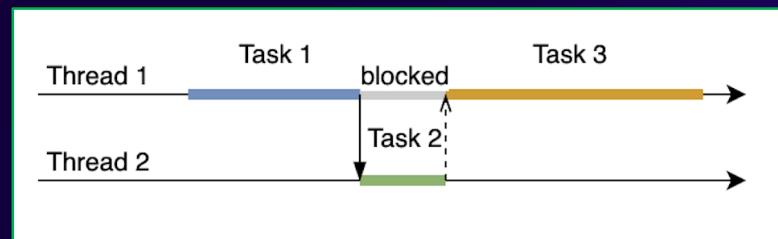
```
1 main() {  
2     task1()  
3     task2()  
4     task3()  
5 }
```



- **Multi Thread + Sync - Parallel**

Blocked. Blocked means that a thread is just waiting (although it could do something useful. e.g. Java ExecutorService and Future) Pseudocode:

```
1 main() {  
2     task1()  
3     Future future = ExecutorService.submit(task2())  
4     future.get() //<- blocked operation  
5     task3()  
6 }
```



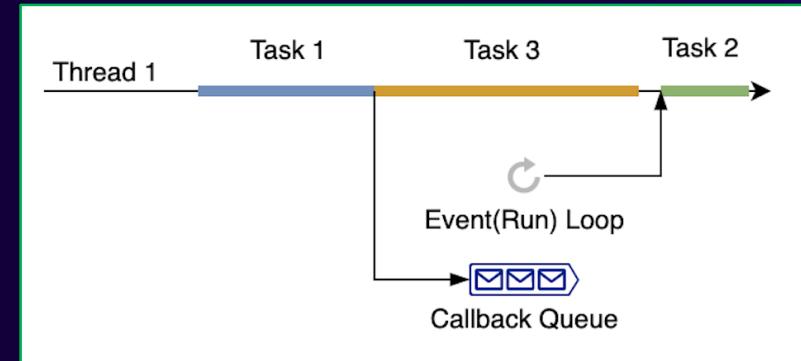
Asynchronous

- Implies that task returns control immediately with a promise to execute a code and notify about result later(e.g. callback, feature). Task 3 is executed even if Task 2 is not finished. async callback, completion handler.

- **Single Thread + Async - Concurrent**

Callback Queue (Message Queue) and Event Loop (Run Loop, Looper) are used. Event Loop checks if Thread Stack is empty and if it is true it pushes first item from the Callback Queue into Thread Stack and repeats these steps again. Simple examples are button click, post event...

```
1 main() {  
2     task1()  
3     ThreadMain.handler.post(task2());  
4     task3()  
5 }
```



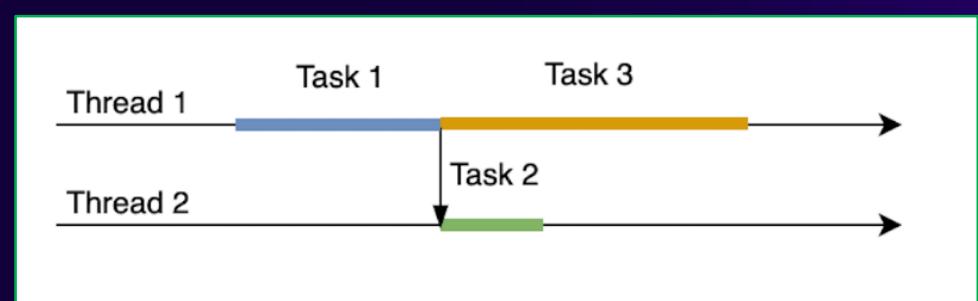
Asynchronous

- Implies that task returns control immediately with a promise to execute a code and notify about result later(e.g. callback, feature). Task 3 is executed even if Task 2 is not finished. async callback, completion handler.

- **Multi Thread + Async - Concurrent and Parallel**

Non-blocking. For example when you need to make some calculations on another thread without blocking. Pseudocode:

```
1 main() {  
2     task1()  
3  
4     new Thread(task2()).start();  
5     //or  
6     Future future = ExecutorService.submit(task2())  
7  
8     task3()  
9 }
```

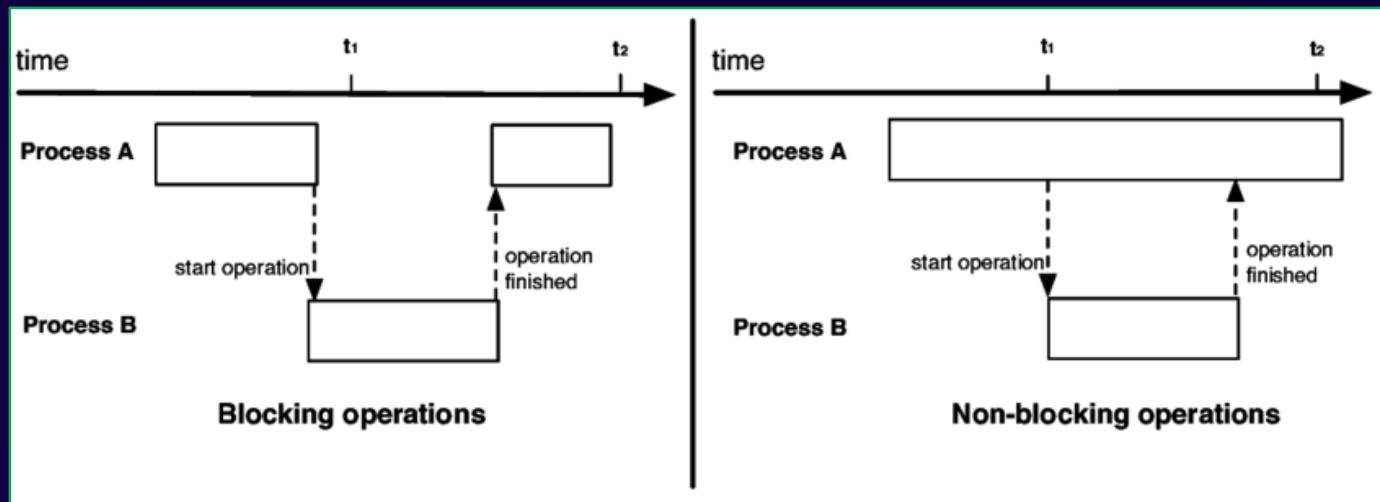


Synchronous vs Asynchronous

Synchronous	Asynchronous
<p>Students learn at the same time.</p> <ul style="list-style-type: none">• Communication is live• Promotes collaboration• Allows for real-time feedback and clarification 	<p>Students learn at different times.</p> <ul style="list-style-type: none">• Communication is not live• Deadlines can be flexible within a greater timeline• Materials are provided up-front 
<p>Examples:</p> <ul style="list-style-type: none">• video conferencing, live chat, live streamed videos	<p>Examples:</p> <ul style="list-style-type: none">• Email, screencasts, Flipgrid videos, blog posts/comments

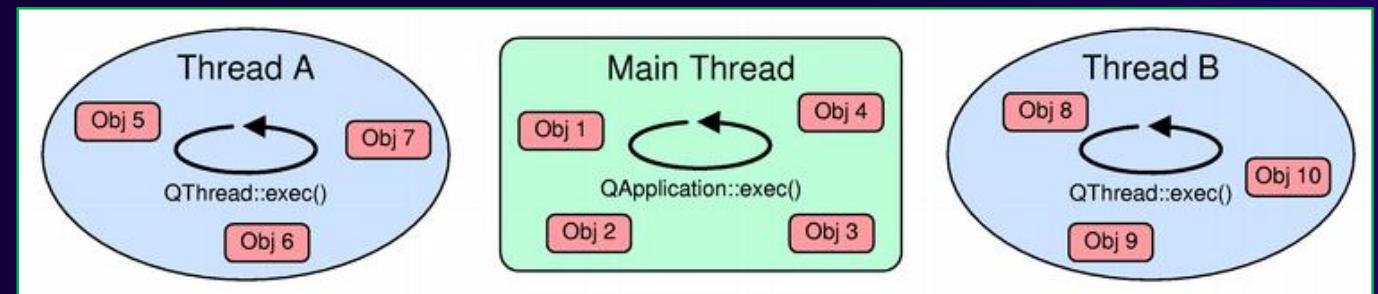
Blocking and Non-Blocking

- Blocking means that execution of your code (in that thread) will stop for the duration of the call. Essentially, the function call will not return until the blocking operation is complete.
- A blocking read will wait until there is data available (or a timeout, if any, expires), and then returns from the function call. A non-blocking read will (or at least should) always return immediately, but it might not return any data, if none is available at the moment.
- Note: Blocking and Non-Blocking are the same Synchronous and Asynchronous.



QThread

- QThread can either be instantiated directly or subclassed. Instantiating a QThread provides a parallel event loop, allowing QObject slots to be invoked in a secondary thread. Subclassing a QThread allows the application to initialize the new thread before starting its event loop, or to run parallel code without an event loop.



Methods

- `void start(QThread::Priority priority = InheritPriority);`
 - Begins execution of the thread by calling `run()`.

Header: `#include <QThread>`

qmake: `QT += core`

Methods

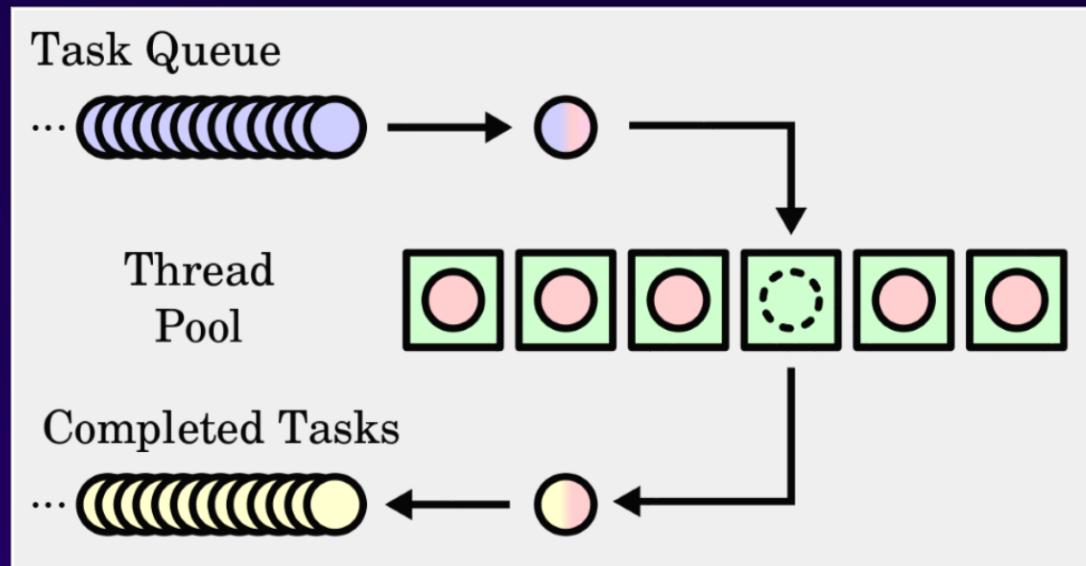
- `void terminate();`
 - Terminates the execution of the thread. The thread may or may not be terminated immediately, depending on the operating system's scheduling policies. Use `QThread::wait()` after `terminate()`, to be sure.
- Warning: This function is dangerous and its use is discouraged. The thread can be terminated at any point in its code path. Threads can be terminated while modifying data. There is no chance for the thread to clean up after itself, unlock any held mutexes, etc. In short, use this function only if absolutely necessary.*
- `void quit();`
 - Tells the thread's event loop to exit with return code 0 (success). Equivalent to calling `QThread::exit(0)`.
- `bool wait(QDeadlineTimer deadline = QDeadlineTimer(QDeadlineTimer::Forever));`
 - The thread associated with this `QThread` object has finished execution (i.e. when it returns from `run()`). This function will return true if the thread has finished. It also returns true if the thread has not been started yet.
 - The deadline is reached. This function will return false if the deadline is reached.

Methods

- **void moveToThread(QThread *targetThread);**
 - Changes the thread affinity for this object and its children. The object cannot be moved if it has a parent. Event processing will continue in the targetThread.
 - To move an object to the main thread, use QApplication::instance() to retrieve a pointer to the current application, and then use QApplication::thread() to retrieve the thread in which the application lives. For example: myObject->moveToThread(QApplication::instance()->thread());
- **void run(); [virtual protected]**
 - The starting point for the thread. After calling start(), the newly created thread calls this function. The default implementation simply calls exec().
 - You can reimplement this function to facilitate advanced thread management. Returning from this method will end the execution of the thread.

QThreadPool

- QThreadPool manages and recycles individual QThread objects to help reduce thread creation costs in programs that use threads. Each Qt application has one global QThreadPool object, which can be accessed by calling `globalInstance()`.
- To use one of the QThreadPool threads, subclass `QRunnable` and implement the `run()` virtual function. Then create an object of that class and pass it to `QThreadPool::start()`.



Header:	#include <QThreadPool> #include <QRunnable>
qmake:	QT += core

Methods

- **int activeThreadCount() const;**
 - This property holds the number of active threads in the thread pool.
- **int maxThreadCount() const;**
 - This property holds the maximum number of threads used by the thread pool.
Note: The thread pool will always use at least 1 thread, even if maxThreadCount limit is zero or negative.
- **void setMaxThreadCount(int maxThreadCount);**
 - Call maxThreadCount() to query the maximum number of threads to be used. If needed, you can change the limit with setMaxThreadCount();
- **bool waitForDone(int msecs = -1);**
 - Waits up to msecs milliseconds for all threads to exit and removes all threads from the thread pool. Returns true if all threads were removed; otherwise it returns false. If msecs is -1 (the default), the timeout is ignored (waits for the last thread to exit).

QList

- The QList class is a template class that provides lists

Methods

- `QList();`
 - Constructs an empty list.
- `void clear();`
 - Removes all items from the list.
- `void push_back(const T &value);`
 - This function is provided for STL compatibility. It is equivalent to `append(value)`.
- `void push_front(const T &value);`
 - This function is provided for STL compatibility. It is equivalent to `prepend(value)`.
- `int size() const;`
 - Returns the number of items in the list.

Header: `#include <QList>`

qmake: `QT += core`

QVector

- The QVector class is a template class that provides a dynamic array.

Methods

- QVector();**
 - Constructs an empty vector.
- QVector(int size);**
 - Constructs a vector with an initial size of size elements.
- void resize(int size);**
 - Sets the size of the vector to size. If size is greater than the current size, elements are added to the end; the new elements are initialized with a default-constructed value. If size is less than the current size, elements are removed from the end.
- void push_back(const T &value);**
 - Constructs an empty string list.
- void removeAt(int i);**
 - Removes the element at index position i.

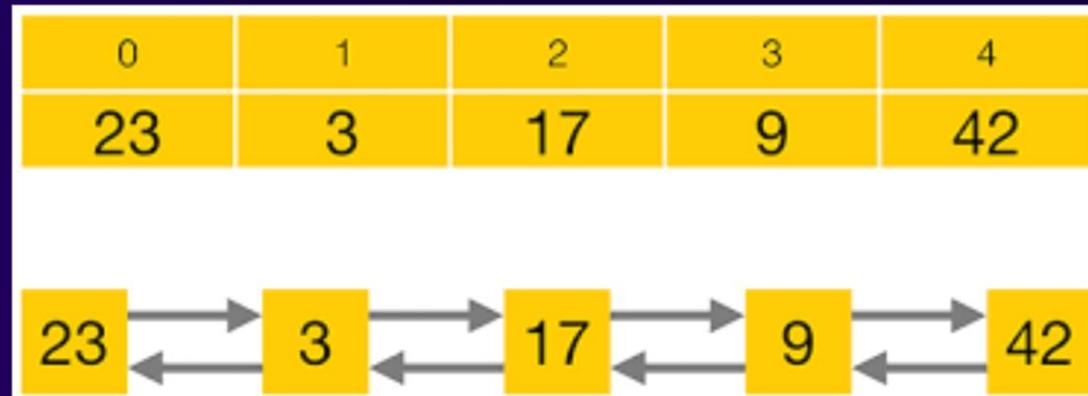
Header: #include <QVector>

qmake: QT += core

Difference Between Vector and List

Vector	List
It has contiguous memory.	While it has non-contiguous memory.
Vector may have a default size.	List does not have default size.
In vector, each element only requires the space for itself only.	In list, each element requires extra space for the node which holds the element, including pointers to the next and previous elements in the list.
Insertion at the end requires constant time but insertion elsewhere is costly.	Insertion is cheap no matter where in the list it occurs.
Deletion at the end of the vector needs constant time but for the rest it is $O(n)$.	Deletion is cheap no matter where in the list it occurs.
Random access of elements is possible.	Random access of elements is not possible.
Iterators become invalid if elements are added to or removed from the vector.	Iterators are valid if elements are added to or removed from the list.

Difference Between Vector and List



	ArrayList	LinkedList
get()	O(1)	O(n)
add()	O(1)	O(1) amortized
remove()	O(n)	O(n)

QDir

- The QDir class provides access to directory structures and their contents.

Methods

- `QDir(const QString &path = QString());`
 - Constructs a QDir pointing to the given directory path. If path is empty the program's working directory, ("."), is used.
- `bool exists() const;`
 - Returns true if the directory exists; otherwise returns false.
- `bool mkdir(const QString &dirName, QFile::Permissions permissions) const;`
 - Creates a sub-directory called dirName.
- `QStringList entryList(const QStringList &nameFilters, QDir::Filters filters = NoFilter, QDir::SortFlags sort = NoSort) const;`
 - Returns a list of the names of all the files and directories in the directory, ordered according to the name and attribute filters previously set with setNameFilters() and setFilter(), and sorted according to the flags set with setSorting().

Header: `#include <QDir>`

qmake: `QT += core`

QMap

- The QMap class is a template class that provides a red-black-tree-based dictionary.
- QMap<Key, T> is one of Qt's generic container classes. It stores (key, value) pairs and provides fast lookup of the value associated with a key.
- We can store multiple values per key by using insertMulti() instead of insert().
- If we want to navigate through all the (key, value) pairs stored in a QMap, we can use an iterator. QMap provides both Java-style iterators (QMapIterator and QMutableMapIterator) and STL-style iterators (QMap::const_iterator and QMap::iterator). The example above shows how to iterate over a QMap<QString, int> using a Java-style iterator.

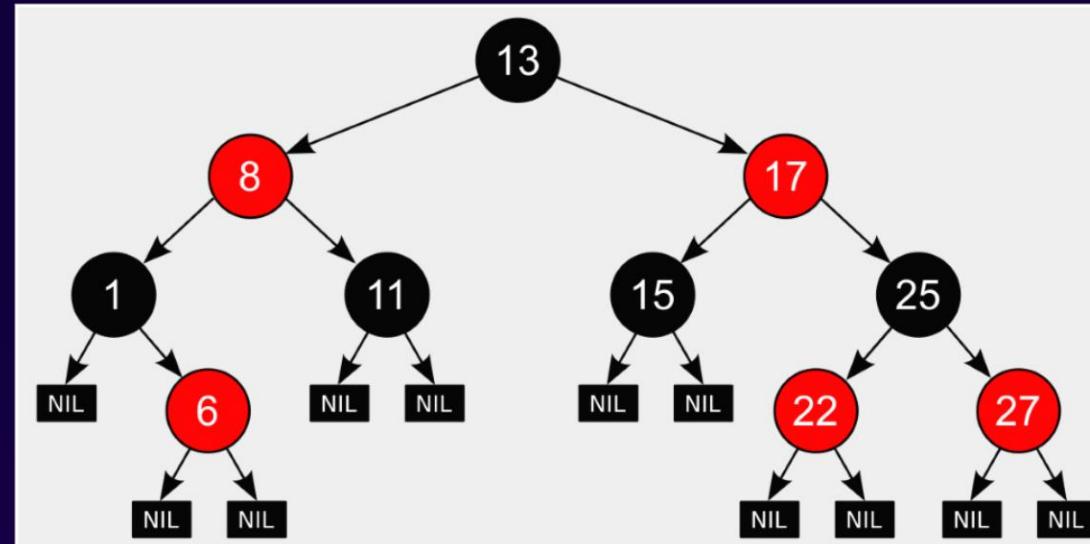
Header: #include <QMap>

qmake: QT += core

Methods

- `QMap(const QMap<Key, T> &other);`
 - Constructs a QMap.
- `QMap::iterator insert(const Key &key, const T &value);`
 - Inserts a new item with the key key and a value of value.
- `const T value(const Key &key, const T &defaultValue = T()) const;`
 - Returns the value associated with the key key as a modifiable reference.

red-black-tree-based dictionary



QtConcurrent

- The `QtConcurrent` namespace provides high-level APIs that make it possible to write multi-threaded programs without using low-level threading primitives such as mutexes, read-write locks, wait conditions, or semaphores. Programs written with `QtConcurrent` automatically adjust the number of threads used according to the number of processor cores available. This means that applications written today will continue to scale when deployed on multi-core systems in the future.

Header:	<code>#include <QtConcurrent></code>
qmake:	<code>QT += concurrent</code>

QtConcurrent

- QtConcurrent includes functional programming style APIs for parallel list processing, including a MapReduce and FilterReduce implementation for shared-memory (non-distributed) systems, and classes for managing asynchronous computations in GUI applications:
 - Concurrent Map and Map-Reduce
 - Concurrent Filter and Filter-Reduce
 - Concurrent Run
 - QFuture represents the result of an asynchronous computation.
 - QFutureIterator allows iterating through results available via QFuture.
 - QFutureWatcher allows monitoring a QFuture using signals-and-slots.
 - QFutureSynchronizer is a convenience class that automatically synchronizes several QFutures.

QFuture

- The QFuture class represents the result of an asynchronous computation. QFuture allows threads to be synchronized against one or more results which will be ready at a later point in time. The result can be of any type that has a default constructor and a copy constructor. If a result is not available at the time of calling the result(), resultAt(), or results() functions, QFuture will wait until the result becomes available. You can use the isResultReadyAt() function to determine if a result is ready or not. For QFuture objects that report more than one result, the resultCount() function returns the number of continuous results.
- QFuture also offers ways to interact with a running computation. For instance, the computation can be canceled with the cancel() function. To pause the computation, use the setPaused() function or one of the pause(), resume(), or togglePaused() convenience functions. Be aware that not all running asynchronous computations can be canceled or paused. For example, the future returned by QtConcurrent::run() cannot be canceled; but the future returned by QtConcurrent::mappedReduced() can.

Header:	#include <QFuture>
qmake:	QT += core

QFuture

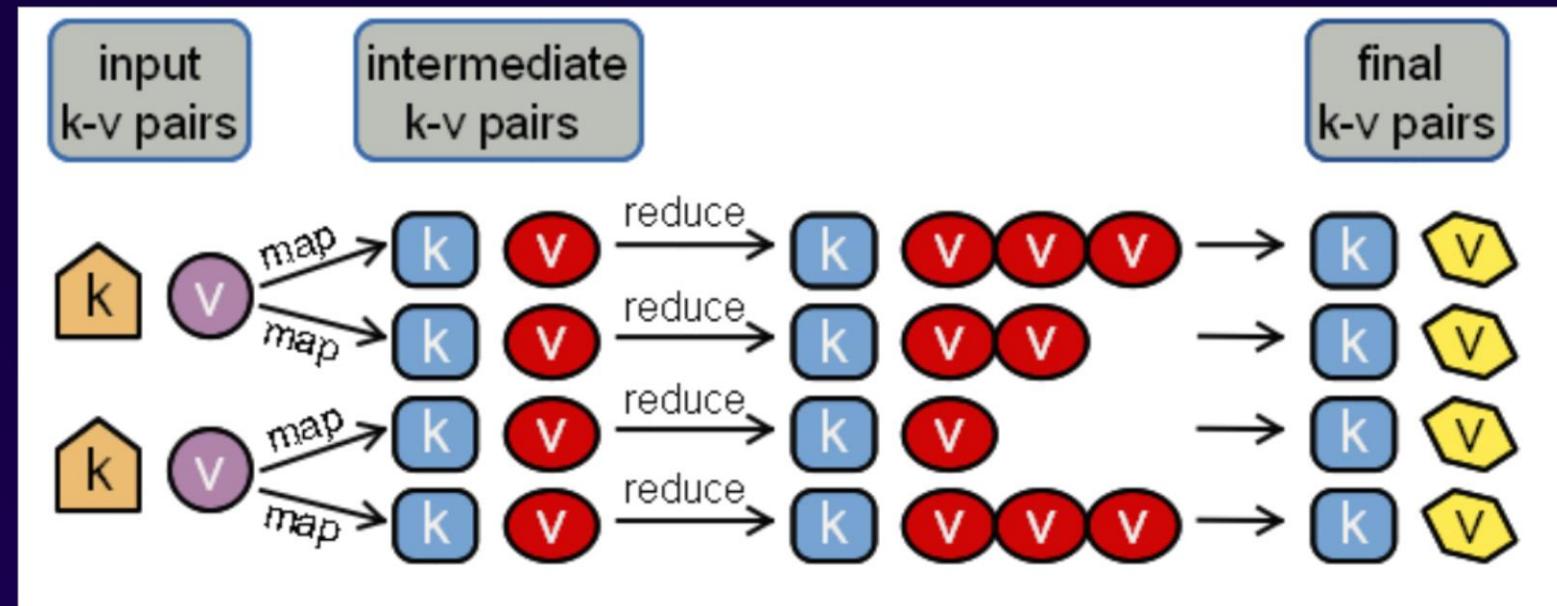
- Note: QFuture also offers ways to interact with a running computation. For instance, the computation can be canceled with the cancel() function. To pause the computation, use the setPaused() function or one of the pause(), resume(), or togglePaused() convenience functions. Be aware that not all running asynchronous computations can be canceled or paused. For example, the future returned by QtConcurrent::run() cannot be canceled; but the future returned by QtConcurrent::mappedReduced() can.

Introduction

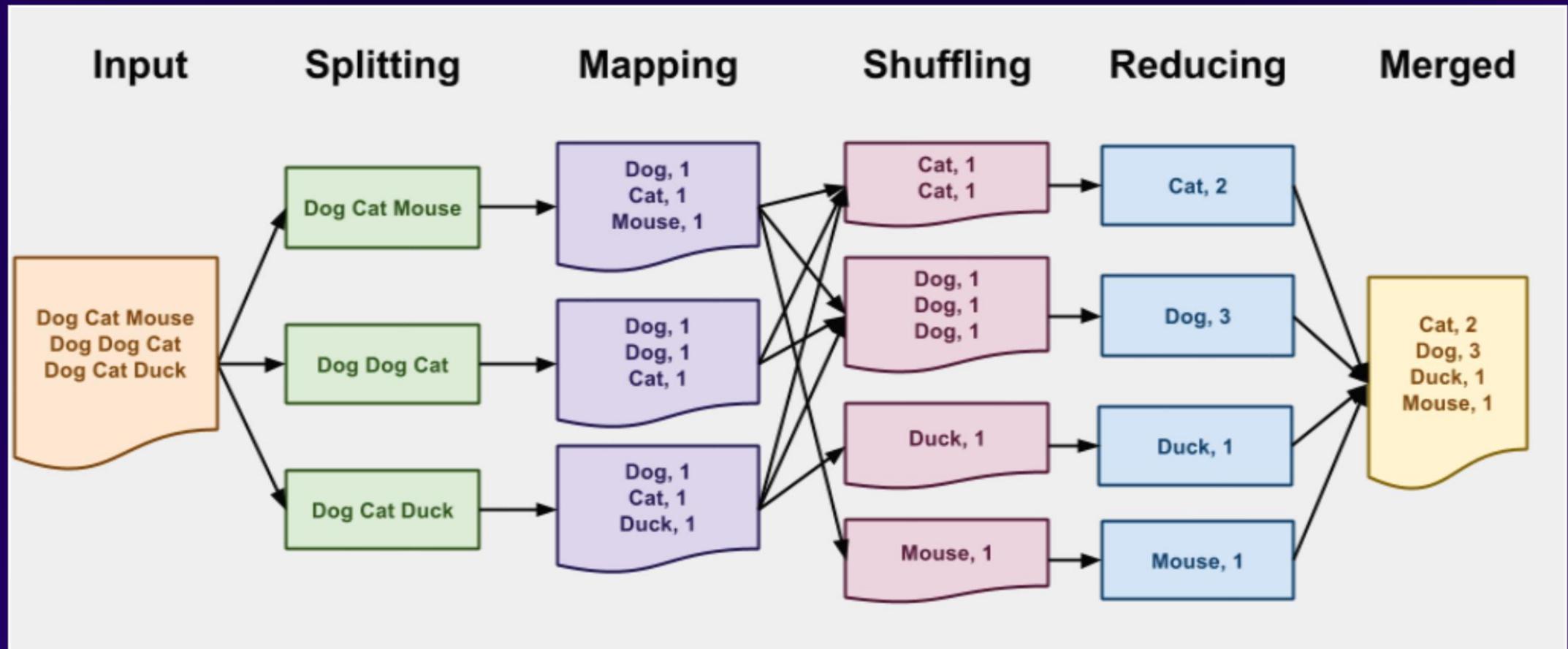
- MapReduce is a high-level programming model for processing large data sets in parallel, originally developed by Google, adapted from functional programming. The model is suitable for a range of problems such as matrix operations, relational algebra, statistical frequency counting, etc. To learn more about MapReduce, you can read Chapter 2 from the book Mining of Massive Datasets by Leskovec, Rajaraman & Ullman.

Introduction

- In general, a map function takes an input and produces an intermediate key-value pair or a list of intermediate key-value pairs. Reduce function performs a summary operation on the intermediate keys. You can see this below:



Introduction



Distribution of population age

- This example shows a census of a small state with 5 cities. We are going to process multiple files, each corresponding to a single city.
- **Variant 1**
 - Variant 1 follows the standard definition of MapReduce. A single map function reads a single city file and converts it to a QList<int>. The map function only converts a single city file into a different format. There is no grouping of values within a single city. It is the reduce function that performs the grouping of ages into one large state distribution.
- **Variant 2**
 - In Variant 2 a map function also reads a single city file. However, unlike in Variant 1, the map function also performs a local reduction , i.e. it groups the ages of citizens of a single city into age intervals. The reduce function then groups the already existing city distributions into one large state distribution.

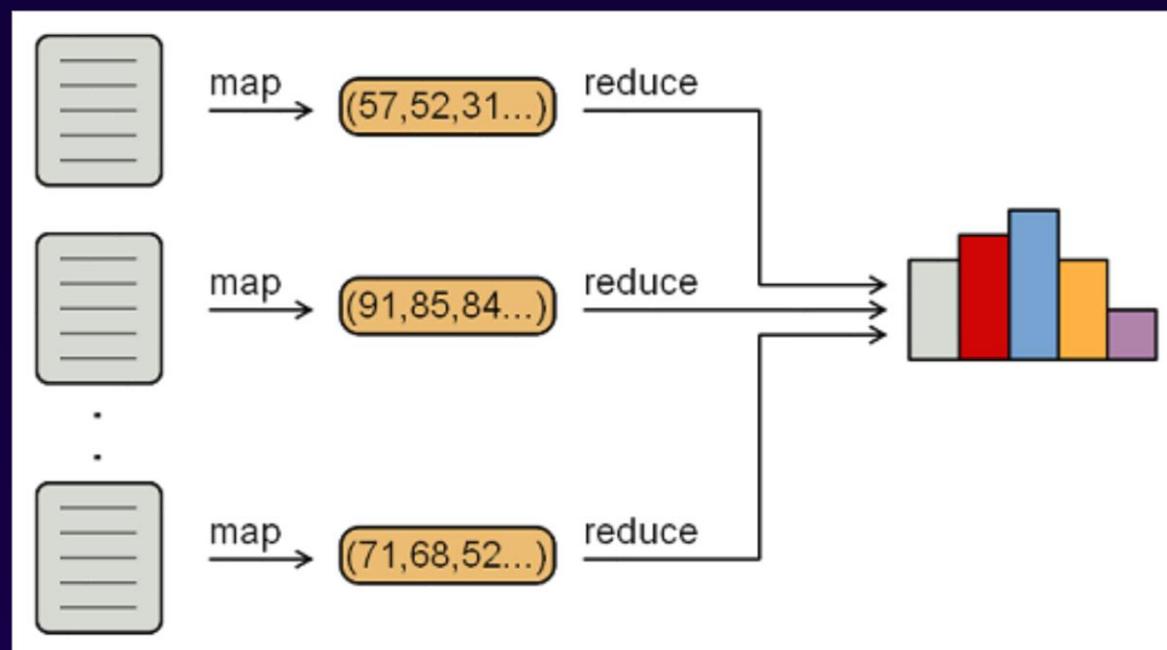
General implementation

- We firstly find all the files in the current folder that start with "city" (in our example this only works for the current folder. If you need an implementation that searches through subdirectories).
- After that, we create an instance of QFuture to represent the result of our asynchronous computation and call Qt::mappedReduced(). Qt's mappedReduced() requires three parameters - a sequence to which to apply the map function, a map function and a reduce function. The fourth parameter is optional and specifies the order in which results from the map function are passed to the reduce function (not relevant for our implementation). You can see the signature below:

```
QFuture<T> QtConcurrent::mappedReduced (const Sequence & sequence, MapFunction  
mapFunction, ReduceFunction reduceFunction, QtConcurrent::ReduceOptions reduceOptions =  
UnorderedReduce | SequentialReduce);
```

Variant 1 - standard implementation

- Variant 1 represents the standard implementation of MapReduce. The map function reads a single city file, checks the validity of the age values and converts it to a QList<int>. This really just results in a different representation of the file, with no grouping into age intervals.

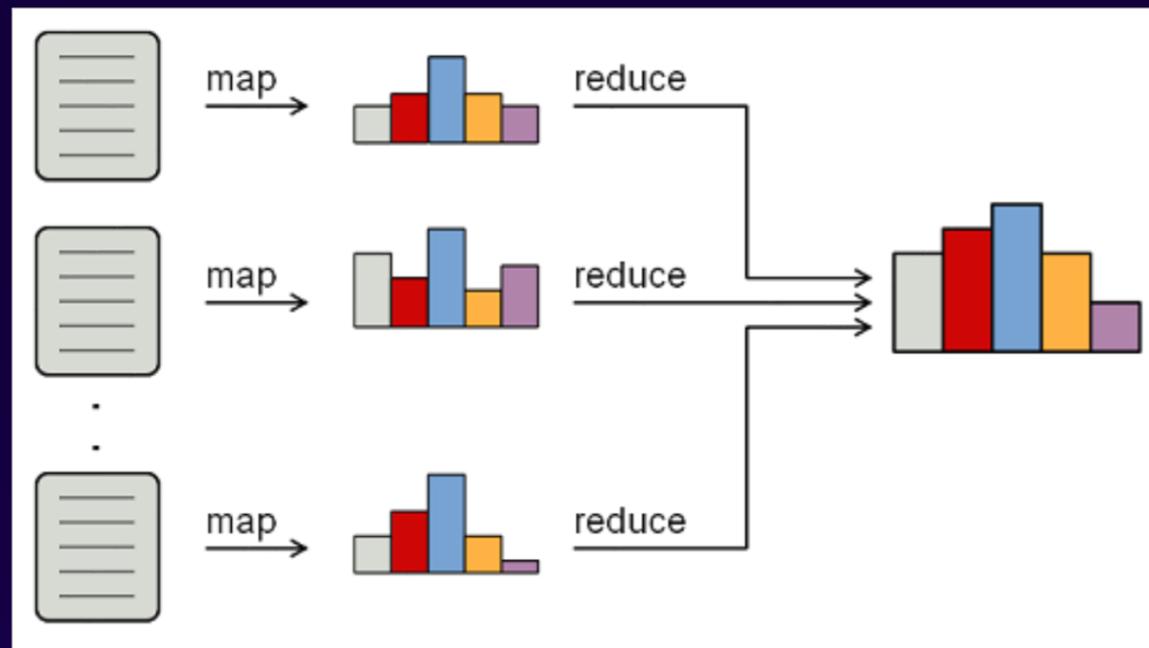


Variant 1 - Standard implementation

- In contrast, the reduce function takes the result of the map function (`QList<int>`) and groups it into age intervals. The resulting (state-level) distribution is stored in `QVector<int>` product. The size of the resulting vector (product), i.e. the number of intervals in the histogram, is derived from the `yearInterval` (number of years that one interval represents, in our case 5 years) and by the age of the oldest person in the state. The product vector is resized whenever it comes across an age that does not fit into any of the current bins.

Variant 2 - Local reduction in the map function

- In Variant 2, the map function, aside from reading a single 'city' file, also performs a local reduction of age values and returns the local age distribution (i.e. age distribution of one city).



Variant 2 - Standard implementation

- The reduce function then groups the existing city distributions into one large state distribution.

Matrix-vector multiplication

- Firstly, a short reminder of how matrix-vector multiplication works (figure below). To calculate a specific entry of the matrix-vector product we have to multiply corresponding row entries of the matrix by the components of the vector and sum up the values. To get the first component of the matrix-vector product, we have to multiply the elements in the first row of the matrix by the corresponding vector components and sum up the values. In matrix-vector multiplication, the number of columns of the matrix has to be equal to the number of components of the vector.

$$\begin{bmatrix} 3 & 2 & 0 \\ 0 & 4 & 1 \\ 2 & 0 & 1 \end{bmatrix} \begin{bmatrix} 4 \\ 3 \\ 1 \end{bmatrix} = \begin{bmatrix} 18 \\ 13 \\ 9 \end{bmatrix}$$

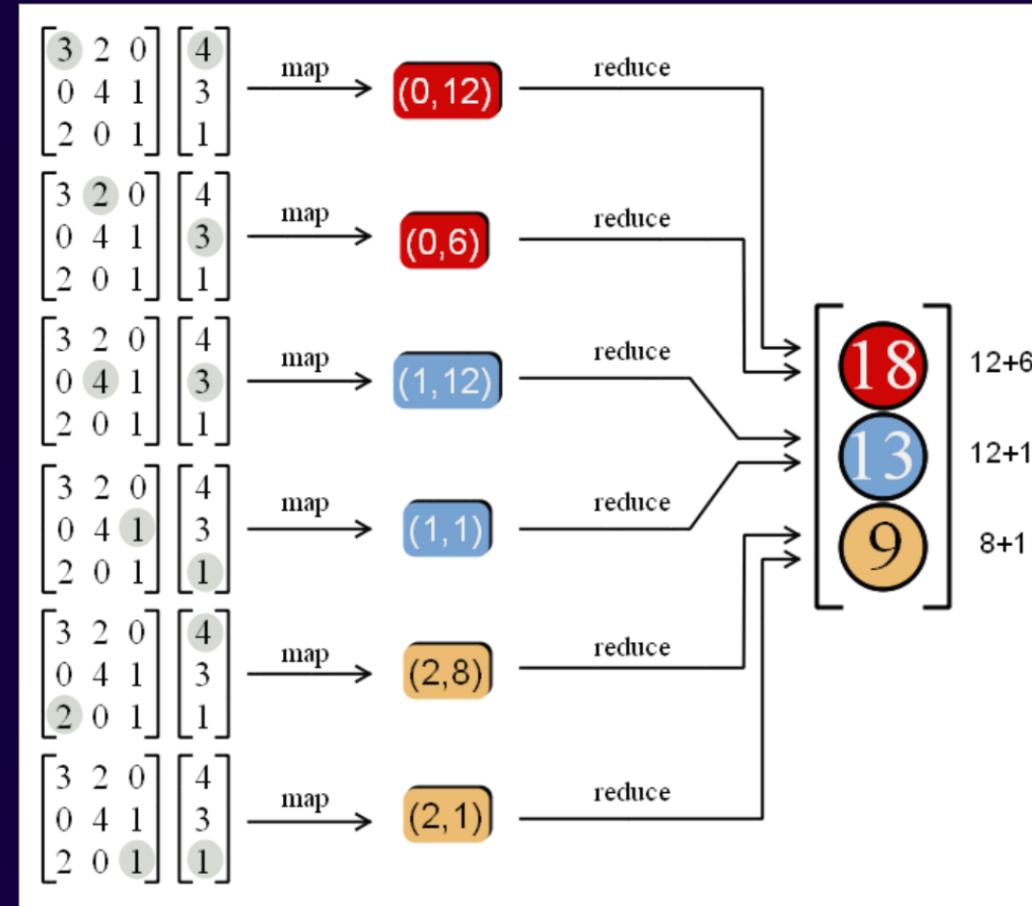
Matrix file:

- We store the sparse matrix (sparse matrices often appear in scientific and engineering problems) as a triple with explicit coordinates (i, j, a_{ij}). E.g. the value of the first entry of the matrix $(0,0)$ is 3. Similarly, the value of the second entry in $(0,1)$ is 2. We do not store the zero entries of the matrix.
- We store the vector in a dense format without explicit coordinates. You can see this below:

```
i, j, aij  
0,0,3  
0,1,2  
1,1,4  
1,2,1  
2,0,2  
2,2,1
```

```
4  
3  
1
```

Map-Reduce function



General implementation for both variants - without and with function object:

- **Variant 1 - without function object**

After populating the matrix and the vector we create an instance of QFuture that will represent the result of our calculation. After that, we call mappedReduced.

- **Variant 2 - with a function object**

In the second variant, we are going to use a map function object instead of a map function. This is so that we can pass the input vector by reference, rather than storing it in a global variable.

Concurrent Run

- The `QtConcurrent::run()` function runs a function in a separate thread. The return value of the function is made available through the `QFuture` API.
- Capabilities:
 - Running a Function in a Separate Thread
 - Passing Arguments to the Function
 - Returning Values from the Function
 - Using Lambda Functions

QFutureWatcher

- template <typename T> class QFutureWatcher
- The QFutureWatcher class allows monitoring a QFuture using signals and slots.
- Be aware that not all running asynchronous computations can be canceled or paused. For example, the future returned by QtConcurrent::run() cannot be canceled; but the future returned by QtConcurrent::mappedReduced() can.

Concept

- A race condition occurs when two or more threads can access shared data and they try to change it at the same time. Because the thread scheduling algorithm can swap between threads at any time, you don't know the order in which the threads will attempt to access the shared data. Therefore, the result of the change in data is dependent on the thread scheduling algorithm, i.e. both threads are "racing" to access/change the data.
- Problems often occur when one thread does a "check-then-act" (e.g. "check" if the value is X, then "act" to do something that depends on the value being X) and another thread does something to the value in between the "check" and the "act". E.g:

```
if (x == 5) // The "Check"
{
    y = x * 2; // The "Act"

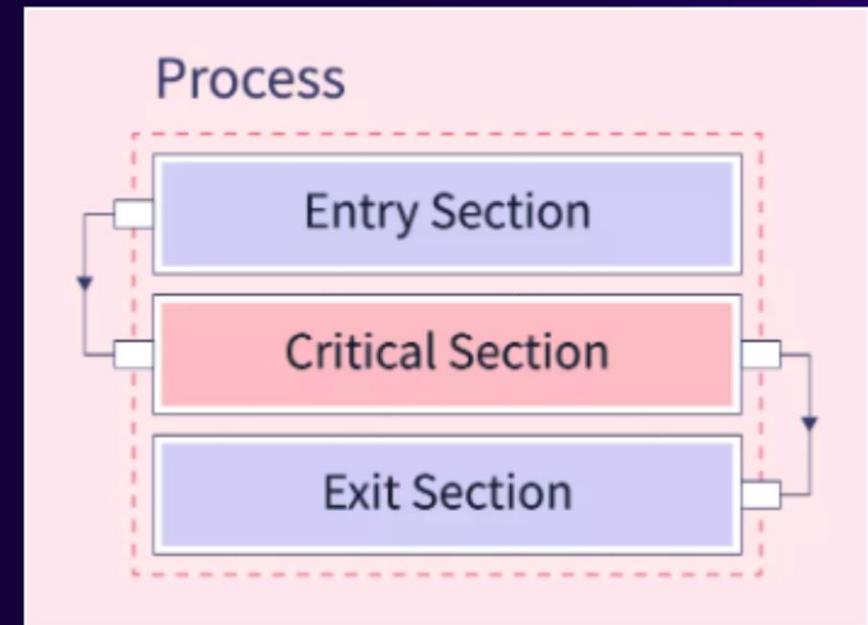
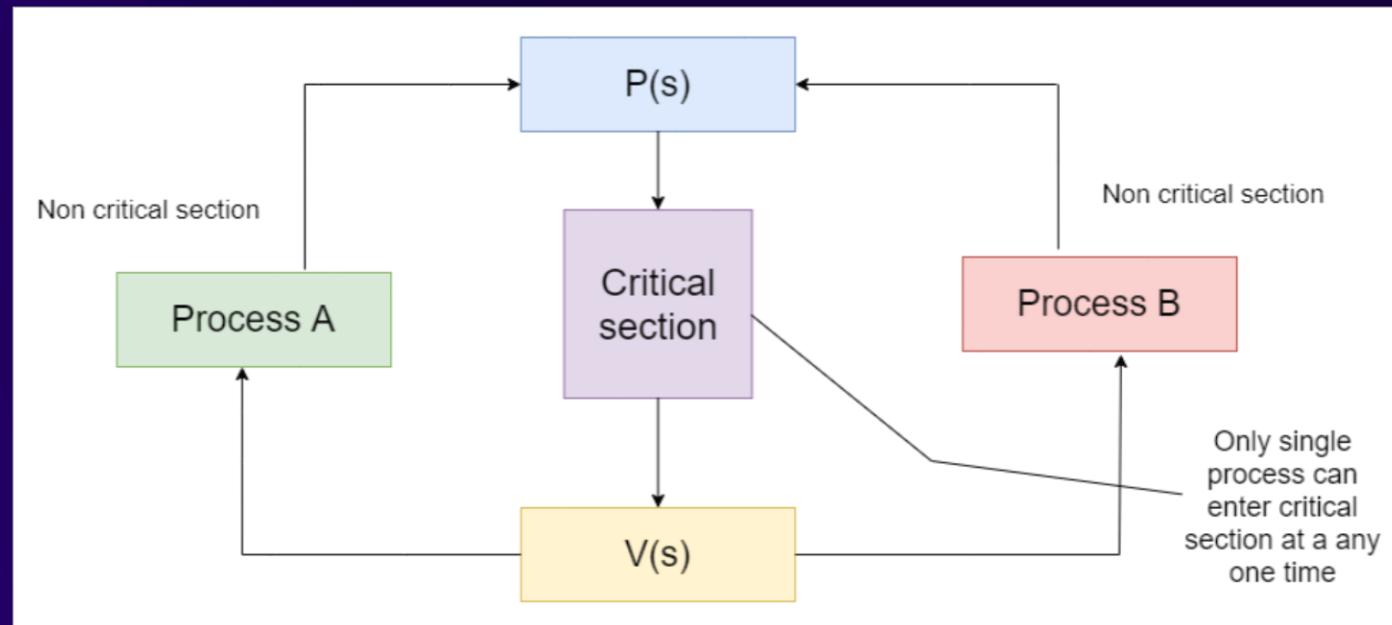
    // If another thread changed x in between "if (x == 5)" and "y = x * 2" above,
    // y will not be equal to 10.
}
```

Concept

- Concurrent accesses to shared resources can lead to unexpected or erroneous behavior, so parts of the program where the shared resource is accessed need to be protected in ways that avoid the concurrent access. One way to do so is known as a critical section or critical region. This protected section cannot be entered by more than one process or thread at a time; others are suspended until the first leaves the critical section.
- In order to prevent race conditions from occurring, you would typically put a lock around the shared data to ensure only one thread can access the data at a time. This would mean something like this:

```
// Obtain lock for x
if (x == 5)
{
    y = x * 2; // Now, nothing can change x until the lock is released.
                // Therefore y = 10
}
// release lock for x
```

Critical Section



Critical Section Conditions

- Any solution to the critical section problem must satisfy three requirements:
 - **Mutual Exclusion:** If a process is executing in its critical section, then no other process is allowed to execute in the critical section.
 - **Progress:** If no process is executing in the critical section and other processes are waiting outside the critical section, then only those processes that are not executing in their remainder section can participate in deciding which will enter in the critical section next, and the selection can not be postponed indefinitely.
 - **Bounded Waiting:** A bound must exist on the number of times that other processes are allowed to enter their critical sections after a process has made a request to enter its critical section and before that request is granted.

What is Atomic?

- An atomic instruction is a type of instruction in embedded programming that is indivisible, meaning it cannot be interrupted by another instruction.

```
count++;
```

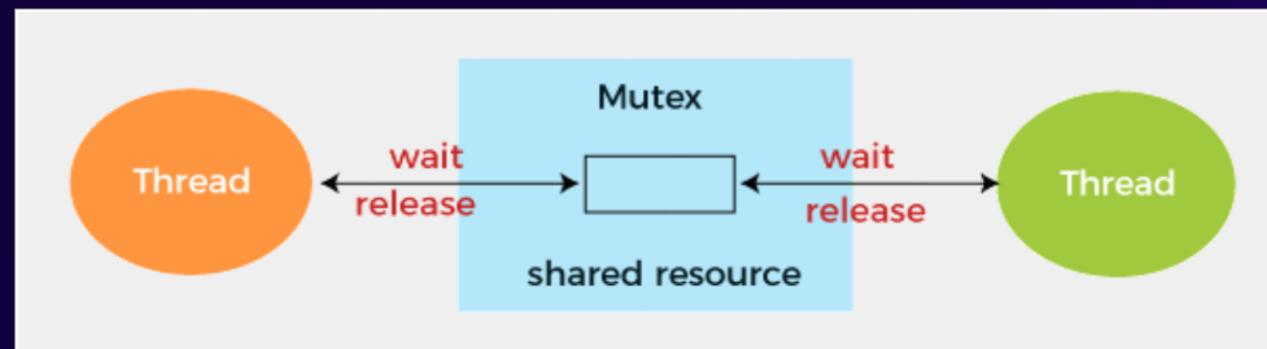
- The above statement can be decomposed into, atleast three operations.
 - Fetching count value
 - Incrementing count value
 - Storing the updated value

If a thread executing the function containing the above statement is fetching its value (say 2). It is possible that at this point of execution, the thread can be preempted and another thread may invoke the same function. Consequently, the value of count will be incremented to 3 by that thread. When the former thread is resumed, it still retains the previous value (2), instead of latest value (3), and ends up in writing back 3 again. Infact, the value of count should be 4 due to affect of both the threads.

What is Mutex?

- Mutex is a mutual exclusion object that synchronizes access to a resource. It is created with a unique name at the start of a program. The mutex locking mechanism ensures only one thread can acquire the mutex and enter the critical section. This thread only releases the mutex when it exits in the critical section.

```
wait (mutex);  
.....  
Critical Section  
.....  
signal (mutex);
```



Advantages of Mutex

- Mutex is just simple locks obtained before entering its critical section and then releasing it.
- Since only one thread is in its critical section at any given time, there are no race conditions, and data always remain consistent.

Disadvantages of Mutex

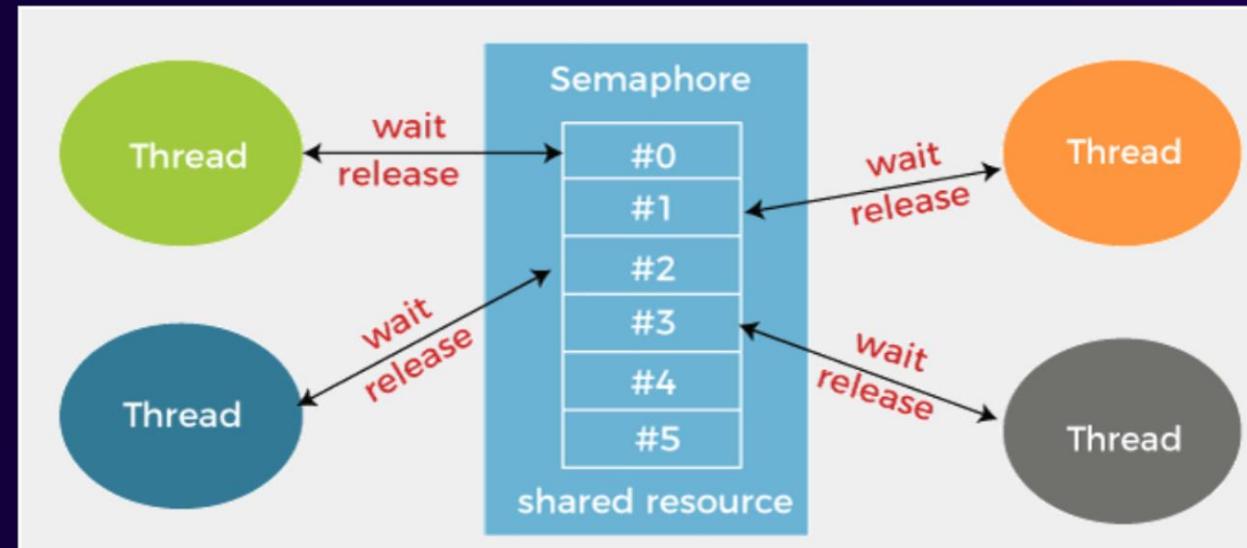
- If a thread obtains a lock and goes to sleep or is preempted, then the other thread may not move forward. This may lead to starvation.
- It can't be locked or unlocked from a different context than the one that acquired it.
- Only one thread should be allowed in the critical section at a time.
- The normal implementation may lead to a busy waiting state, which wastes CPU time.

What is Semaphore?

- Semaphore is simply a variable that is non-negative and shared between threads. A semaphore is a signaling mechanism, and another thread can signal a thread that is waiting on a semaphore.
- A semaphore uses two atomic operations.

```
wait(S)
{
    while (S<=0);
    S--;
}

signal(S)
{
    S++;
}
```

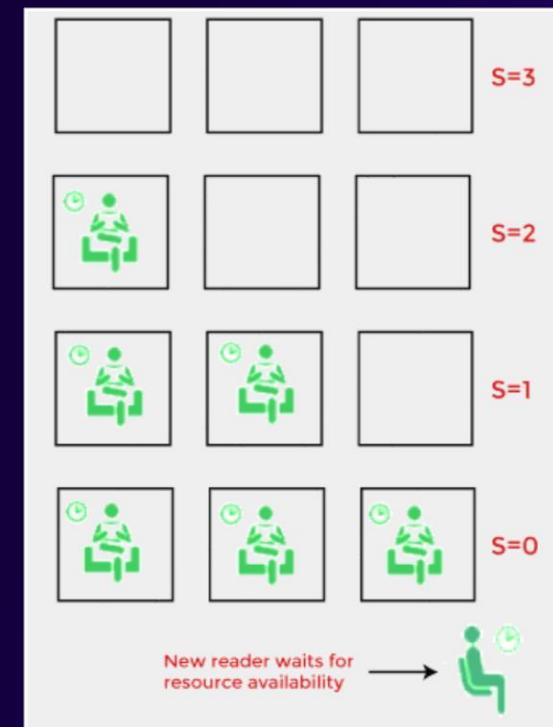
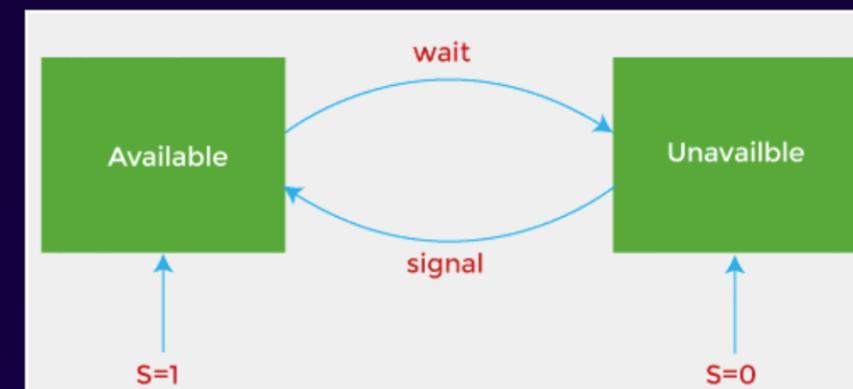


Wait and Signal

- 1. Wait: The wait operation decrements the value of its argument S if it is positive. If S is negative or zero, then no operation is performed.
- 2. Signal for the process synchronization: The signal operation increments the value of its argument S.

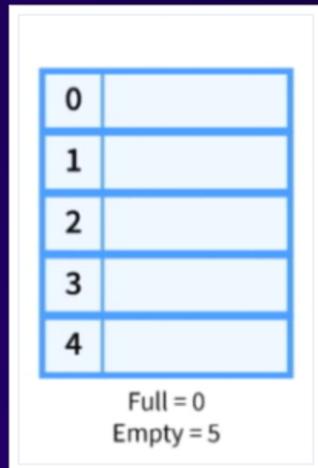
Types of Semaphore

- 1. Counting Semaphore
- 2. Binary semaphore



Concept

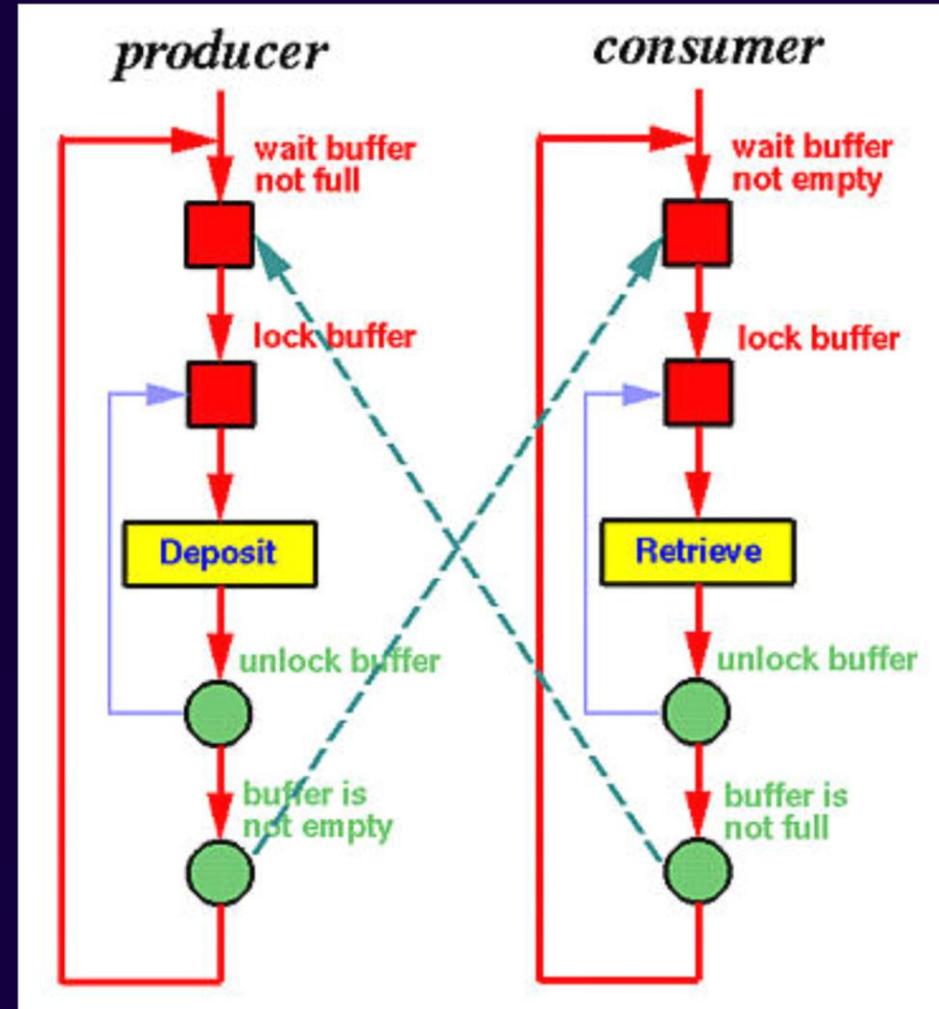
- A producer must wait until the buffer is not full, deposit its data, and then notify the consumers that the buffer is not empty.
- A consumer, on the other hand, must wait until the buffer is not empty, retrieve a data item, and then notify the producers that the buffer is not full.



```
1 | do{  
2 |     // producer produces an item  
3 |     wait(empty);  
4 |     wait(mutex);  
5 |     // put the item into the buffer  
6 |     signal(mutex);  
7 |     signal(full);  
8 | } while(true)
```

```
1 | do{  
2 |     wait(full);  
3 |     wait(mutex);  
4 |     // removal of the item from the buffer  
5 |     signal(mutex);  
6 |     signal(empty);  
7 |     // consumer now consumed the item  
8 | } while(true)
```

Concept



QMutex

- The QMutex class provides access serialization between threads.

Methods

- **QMutex();**
 - Constructs a new mutex. The mutex is created in an unlocked state.
- **void lock();**
 - Locks the mutex. If another thread has locked the mutex then this call will block until that thread has unlocked it.
- **void unlock();**
 - Unlocks the mutex. Attempting to unlock a mutex in a different thread to the one that locked it results in an error. Unlocking a mutex that is not locked results in undefined behavior.
 - **Preconditions: The calling thread owns the mutex.**

Warning: Destroying a locked mutex may result in undefined behavior.

Header:	#include <QMutex>
qmake:	QT += core

QSemaphore

- The QSemaphore class provides a general counting semaphore.

Methods

- `QSemaphore(int n = 0);`
 - Creates a new semaphore and initializes the number of resources it guards to n (by default, 0).
- `acquire(int n = 1);`
 - Tries to acquire n resources guarded by the semaphore. If n > available(), this call will block until enough resources are available.
- `available() const;`
 - Returns the number of resources currently available to the semaphore. This number can never be negative.
- `release(int n = 1);`
 - Releases n resources guarded by the semaphore.

Warning: Destroying a semaphore that is in use may result in undefined behavior.

Header: #include <QSemaphore>

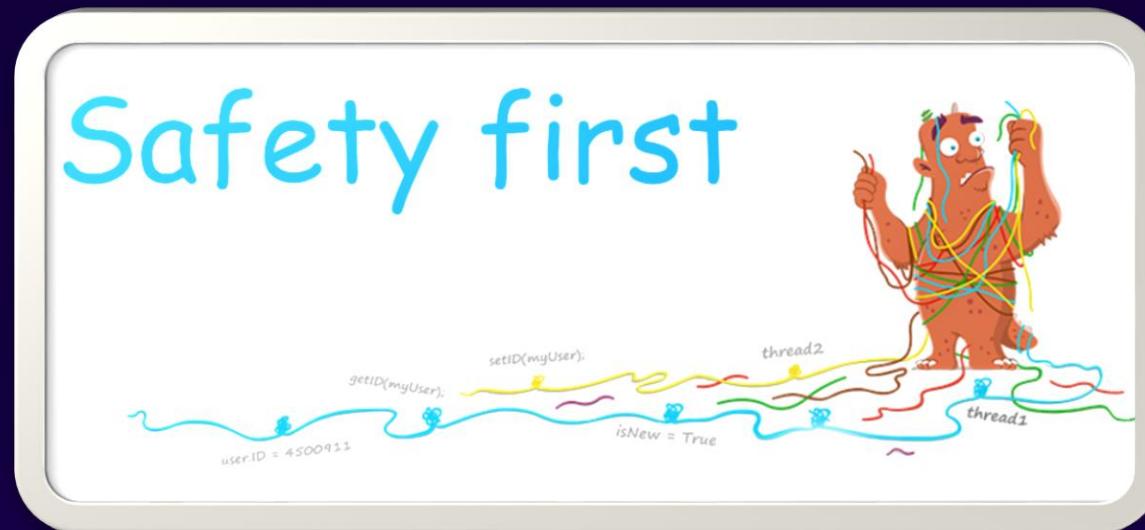
qmake: QT += core

Concept

- Thread-safe code only manipulates shared data structures in a manner that ensures that all threads behave properly and fulfill their design specifications without unintended interaction. There are various strategies for making thread-safe data structures.

Levels of thread safety

- Software libraries can provide certain thread-safety guarantees. For example, concurrent reads might be guaranteed to be thread-safe, but concurrent writes might not be.



Levels of thread safety

- Different vendors use slightly different terminology for thread-safety:
 - Thread safe: Implementation is guaranteed to be free of race conditions when accessed by multiple threads simultaneously.
 - Conditionally safe: Different threads can access different objects simultaneously, and access to shared data is protected from race conditions.
 - Not thread safe: Data structures should not be accessed simultaneously by different threads.
- Note: Thread safety guarantees usually also include design steps to prevent or limit the risk of different forms of deadlocks, as well as optimizations to maximize concurrent performance. However, deadlock-free guarantees cannot always be given, since deadlocks can be caused by callbacks and violation of architectural layering independent of the library itself.

Approaches for avoiding race conditions to achieve thread-safety:

- The **first** class of approaches focuses on avoiding shared state and includes:
 - **Re-entrancy**: Writing code in such a way that it can be partially executed by a thread, executed by the same thread. This requires the saving of state information in variables local to each execution.
 - **Thread-local storage**: Variables are localized so that each thread has its own private copy. These variables retain their values across subroutine and other code boundaries and are thread-safe.
 - **Immutable objects**: The state of an object cannot be changed after construction. This implies both that only read-only data is shared and that inherent thread safety is attained. Mutable (non-const) operations can then be implemented in such a way that they create new objects instead of modifying existing ones.

Approaches for avoiding race conditions to achieve thread-safety:

- The **second** class of approaches are synchronization-related, and are used in situations where shared state cannot be avoided:
 - **Mutual exclusion**: Access to shared data is serialized using mechanisms that ensure only one thread reads or writes to the shared data at any time.
 - **Atomic operations**: Shared data is accessed by using atomic operations which cannot be interrupted by other threads. This usually requires using special machine language instructions, which might be available in a runtime library.

Thread-safe in QT

- The class is thread-safe if its member functions can be called safely from multiple threads, even if all the threads use the same instance of the class. Note: Qt classes are only documented as thread-safe if they are intended to be used by multiple threads.

Reentrancy and Thread-Safety

- A thread-safe function can be called simultaneously from multiple threads, even when the invocations use shared data, because all references to the shared data are serialized.
- A reentrant function can also be called simultaneously from multiple threads, but only if each invocation uses its own data.

Reentrancy and Thread-Safety

QSettings Class

The QSettings class provides persistent platform-independent application settings. [More...](#)

Header:	#include <QSettings>
qmake:	QT += core
Inherits:	QObject

- › [List of all members, including inherited members](#)
- › [Obsolete members](#)

Note: All functions in this class are **reentrant**.

Note: These functions are also **thread-safe**:

Note: These functions are also **thread-safe**:

Note: All functions in this class are **reentrant**.

QQueue Class

template <typename T> class QQueue

The QQueue class is a generic container that provides a queue. [More...](#)

Header:	#include <QQueue>
qmake:	QT += core
Inherits:	QList

- › [List of all members, including inherited members](#)

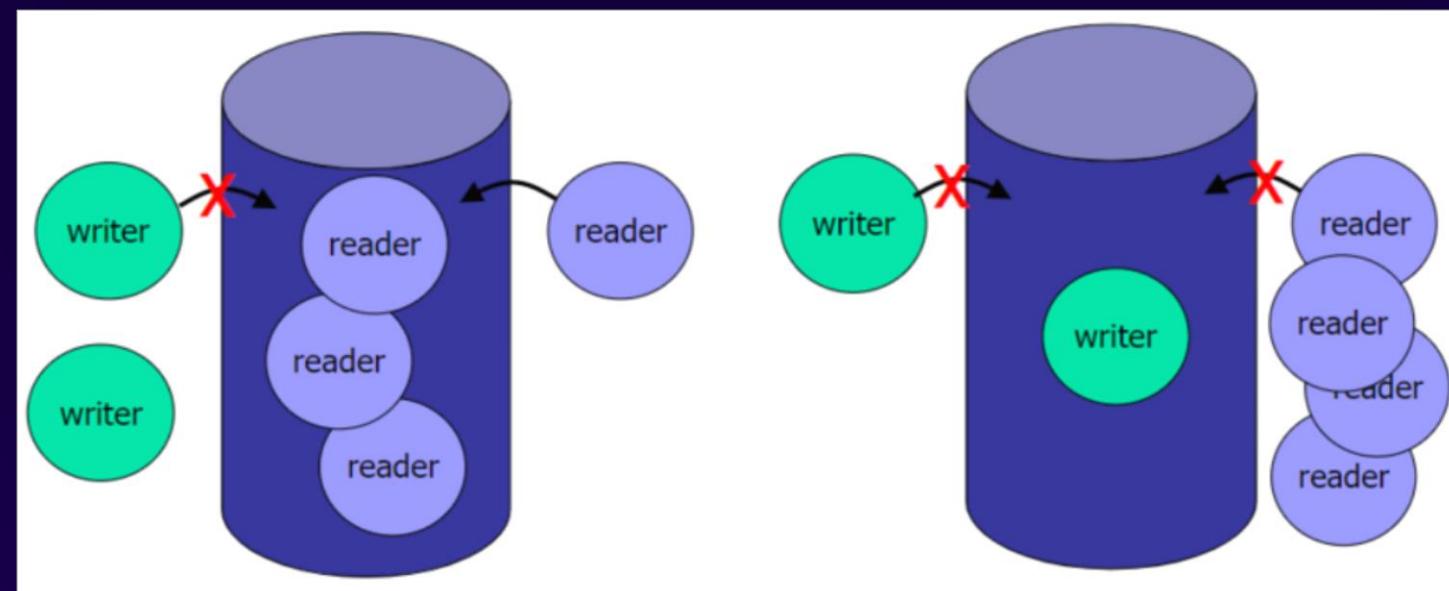
Note: All functions in this class are **reentrant**.

Note: All functions in this class are **reentrant**.

- › [List of all members, including inherited members](#)

Readers/Writers Problem

1. It is not possible to read and write at the same time.
2. Many readers can be at the same time.
3. Only one writer at a time.



Readers/Writers Problem:

- **Using a QMutex:**

- readers share data structures:

1. semaphore mutex, wrt; // initialized to 1
2. int readcount; // initialized to 0, controlled by mutex

- writers also share semaphore wrt.

```
1 Reader:
2
3     while(1)
4     {
5         wait(mutex);
6         readcount++;
7
8         if (readcount==1) wait(wrt);
9
10        signal(mutex);
11
12        // reading
13
14        wait(mutex);
15        readcount--;
16
17        if (readcount==0) signal(wrt);
18
19        signal(mutex);
20 }
```

```
1 Writer:
2
3     while(1)
4     {
5         wait(wrt);
6         // writing
7         signal(wrt);
8     }
```

Readers/Writers Problem:

- **Using a QReadWriteLock**

- A read-write lock is a synchronization tool for protecting resources that can be accessed for reading and writing. This type of lock is useful if you want to allow multiple threads to have simultaneous read-only access, but as soon as one thread wants to write to the resource, all other threads must be blocked until the writing is complete.

```
1  QReadWriteLock lock;
2
3  void ReaderThread::run()
4  {
5      ...
6      lock.lockForRead();
7      read_file();
8      lock.unlock();
9      ...
10 }
11
12 void WriterThread::run()
13 {
14     ...
15     lock.lockForWrite();
16     write_file();
17     lock.unlock();
18     ...
19 }
```

QMutexLocker

- The QMutexLocker class is a convenience class that simplifies locking and unlocking mutexes.
- QMutexLocker should be created within a function where a QMutex needs to be locked. The mutex is locked when QMutexLocker is created. You can unlock and relock the mutex with unlock() and relock(). If locked, the mutex will be unlocked when the QMutexLocker is destroyed.

QMutexLocker

- For example, this complex function locks a QMutex upon entering the function and unlocks the mutex at all the exit points:

```
1 int complexFunction(int flag)
2 {
3     mutex.lock();
4
5     int retVal = 0;
6
7     switch (flag) {
8     case 0:
9     case 1:
10        retVal = moreComplexFunction(flag);
11        break;
12    case 2:
13    {
14        int status = anotherFunction();
15        if (status < 0) {
16            mutex.unlock();
17            return -2;
18        }
19        retVal = status + flag;
20    }
21    break;
22    default:
23        if (flag > 10) {
24            mutex.unlock();
25            return -1;
26        }
27        break;
28    }
29
30    mutex.unlock();
31    return retVal;
32 }
```

```
1 int complexFunction(int flag)
2 {
3     QMutexLocker locker(&mutex);
4
5     int retVal = 0;
6
7     switch (flag) {
8     case 0:
9     case 1:
10        return moreComplexFunction(flag);
11    case 2:
12    {
13        int status = anotherFunction();
14        if (status < 0)
15            return -2;
16        retVal = status + flag;
17    }
18    break;
19    default:
20        if (flag > 10)
21            return -1;
22        break;
23    }
24
25    return retVal;
26 }
```

QWaitCondition

- The QWaitCondition class provides a condition variable for synchronizing threads.

Methods

- QWaitCondition();
 - Constructs a new wait condition object.
- Bool wait(QMutex *lockedMutex, QDeadlineTimer deadline = QDeadlineTimer(QDeadlineTimer::Forever));
 - Releases the lockedMutex and waits on the wait condition. The lockedMutex must be initially locked by the calling thread. If lockedMutex is not in a locked state, the behavior is undefined. If lockedMutex is a recursive mutex, this function returns immediately. The lockedMutex will be unlocked, and the calling thread will block until either of these conditions is met:
 - Another thread signals it using wakeOne() or wakeAll(). This function will return true in this case.
 - the deadline given by deadline is reached. If deadline is QDeadlineTimer::Forever (the default), then the wait will never timeout (the event must be signalled). This function will return false if the wait timed out.
 - The lockedMutex will be returned to the same locked state. This function is provided to allow the atomic transition from the locked state to the wait state.

Header: #include <QWaitCondition>

qmake: QT += core

Methods

- **void wakeAll();**
 - Wakes all threads waiting on the wait condition. The order in which the threads are woken up depends on the operating system's scheduling policies and cannot be controlled or predicted.
- **void wakeOne()**
 - Wakes one thread waiting on the wait condition. The thread that is woken up depends on the operating system's scheduling policies, and cannot be controlled or predicted.
 - If you want to wake up a specific thread, the solution is typically to use different wait conditions and have different threads wait on different conditions.

Example

- Let's suppose that we have three tasks that should be performed whenever the user presses a key. Each task could be split into a thread, each of which would have a run() body like this:

```
1 Forever {  
2     mutex.lock();  
3     keyPressed.wait(&mutex);  
4     do_something();  
5     mutex.unlock();  
6 }
```

- Here, the keyPressed variable is a global variable of type QWaitCondition.
- A fourth thread would read key presses and wake the other three threads up every time it receives one, like this:

```
1 Forever {  
2     getchar();  
3     keyPressed.wakeAll();  
4 }
```

- Note: The order in which the three threads are woken up is undefined.

Example

- Also, if some of the threads are still in `do_something()` when the key is pressed, they won't be woken up (since they're not waiting on the condition variable) and so the task will not be performed for that key press. This issue can be solved using a counter and a QMutex to guard it.

```
1 forever {
2     mutex.lock();
3     keyPressed.wait(&mutex);
4     ++count;
5     mutex.unlock();
6
7     do_something();
8
9     mutex.lock();
10    --count;
11    mutex.unlock();
12 }
```

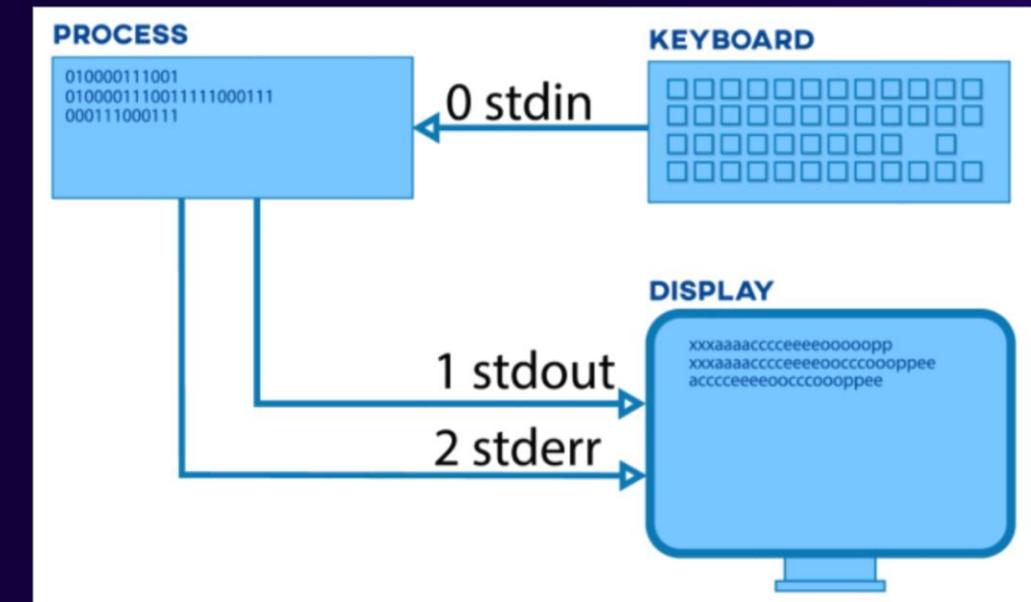
- Here's the code for the fourth thread:
- Note: The mutex is necessary because the results of two threads attempting to change the value of the same variable simultaneously are unpredictable.

```
1 forever {
2     getchar();
3
4     mutex.lock();
5     // Sleep until there are no busy worker threads
6     while (count > 0) {
7         mutex.unlock();
8         sleep(1);
9         mutex.lock();
10    }
11    keyPressed.wakeAll();
12    mutex.unlock();
13 }
14 }
```

Stream

- Every process In Windows/Linux produces three data streams, “stdin,” “stdout,” and “stderr”.
 - stdin: Takes input from the user via keyboard
 - stdout: Displays output on the screen
 - stderr: Shows error information on the screen
- Every data stream has a numeric id:

Numeric Id	Name
0	stdin
1	stdout
2	stderr



Linux

- **How to redirect Standard output and Standard error in Bash:**
 - To redirect the standard output of the command, we will use “1” with a redirection operator that is greater than the “>” sign:

```
$ls 1> stdout.txt # $ls > stdout.txt
```
 - The above command will create a file and place the standard output of the “ls” command in the “stdout.txt” file.
- **We can redirect standard error to a file as well by using the command:**
 - Make sure use “2” will greater than the “>” sign. Since there is no “myfile.txt” file in the directory, the “cat” command will give an error that will be appended in the “stderr.txt” file.

```
$cat myfile.txt 2> stderr.txt
```

Linux

- These standard outputs can be redirected with a single command also, use:
 - The output of the “ls” command will be written in the “stdout.txt” file, but the “stderr.txt” will remain empty because there would be no error.

```
$ ls 1> stdout.txt 2> stderr.txt
```

- Redirecting stdout and stderr to a single file:

```
cat sample.txt &> out
```

- Redirecting stderr to stdout:

```
$ ls > samplefile.txt 2>&1
```

Windows

- **Sending the STDERR and STDOUT to different files:**
 - Here the “File Not Found” message is the STDERR and the rest was for STDOUT.
- **Sending the STDERR and STDOUT to Same file:**
 - Here the 2>&1 instructs that the STDERR to be redirected to STDOUT which is in-turn writing out to alloutput.log file.

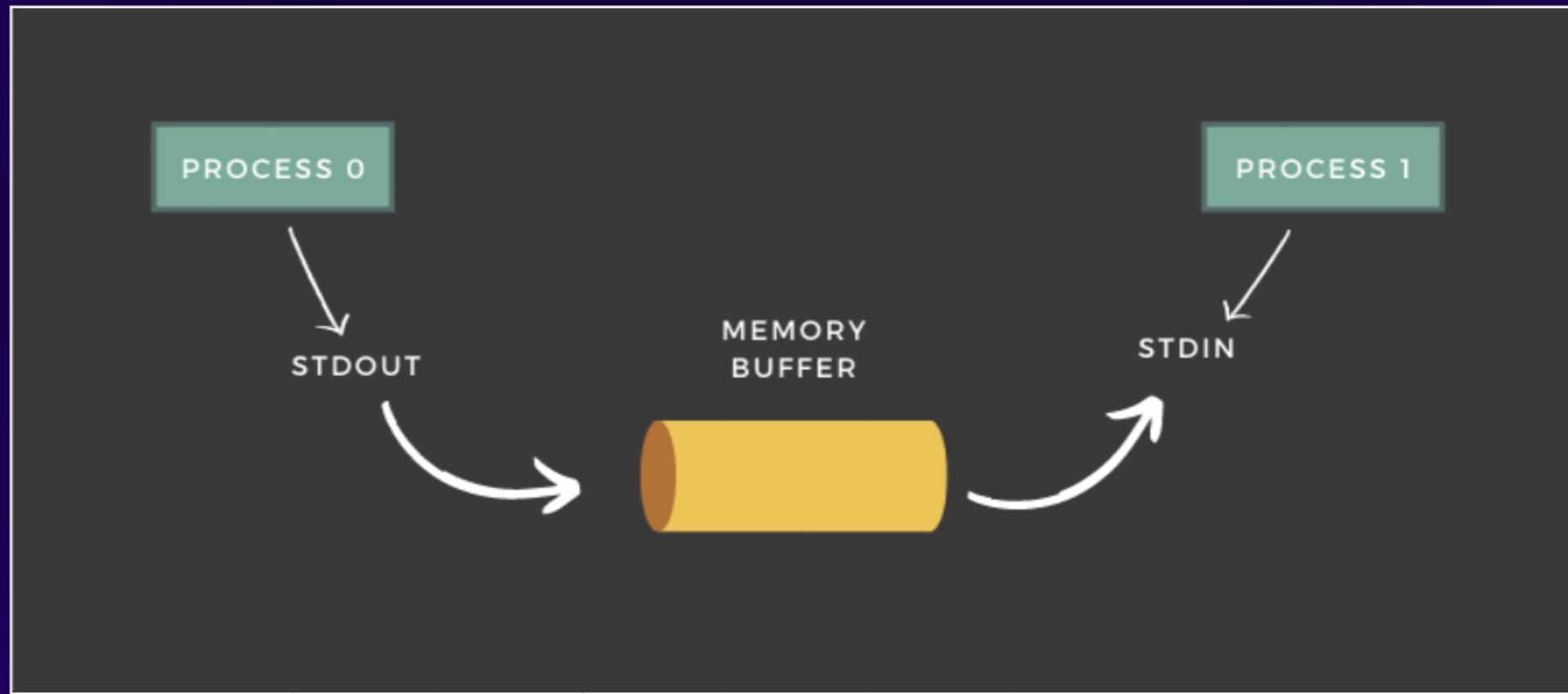
```
>dir nosuchfile.txt > out.log 2>error.log
```

```
>dir nosuchfile.txt > alloutput.log 2>&1
```

Pipe

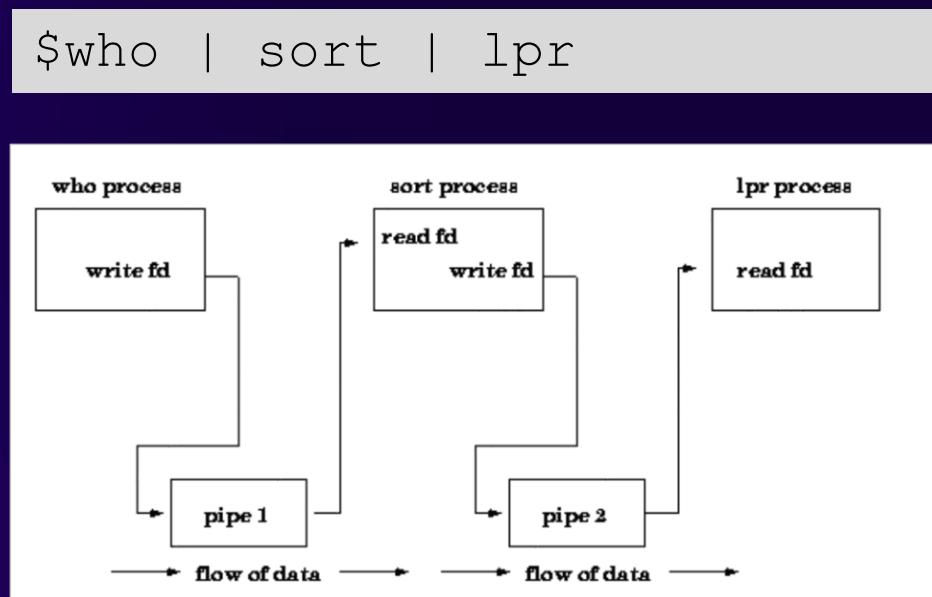
- Conceptually, a pipe is a connection between two processes, such that the standard output from one process becomes the standard input of the other process. In Operating System, Pipes are useful for communication between related processes(inter-process communication).
 - Pipe is one-way communication only i.e we can use a pipe such that One process write to the pipe, and the other process reads from the pipe. It opens a pipe, which is an area of main memory that is treated as a “virtual file”.
 - The pipe can be used by the creating process, as well as all its child processes, for reading and writing. One process can write to this “virtual file” or pipe and another related process can read from it.
 - If a process tries to read before something is written to the pipe, the process is suspended until something is written.
 - The pipe system call finds the first two available positions in the process's open file table and allocates them for the read and write ends of the pipe.

Pipe

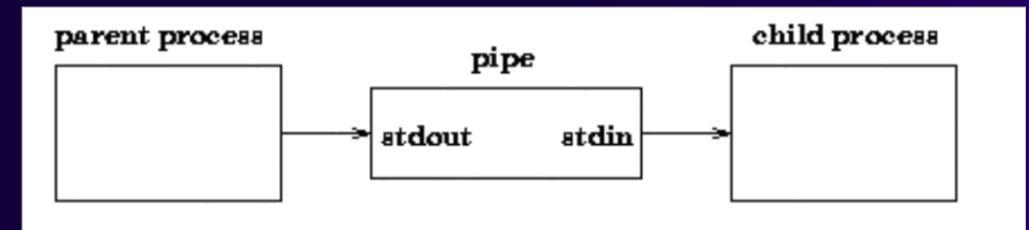


Pipe in Linux

- Then the Unix shell would create three processes with two pipes between them:



- When a pipe is used in a Unix command line, the first process is assumed to be writing to stdout and the second is assumed to be reading from stdin.



Pipe in Windows

- The pipe operator (|) takes the output (by default, STDOUT) of one command and directs it into the input (by default, STDIN) of another command. For example, the following command sorts the contents of the directory C:\

```
>dir C:\ | sort
```

- Combining Commands with Redirection Operators:

```
>dir C:\ | find "txt" > AllText.txt
```

- Using Multiple Pipe Commands:

```
>dir c:\ /s /b | find "TXT" | more
```

```
>tasklist | find "notepad"
```

QProcess

- The QProcess class is used to start external programs and to communicate with them.

Methods

- `QProcess(QObject *parent = nullptr);`
 - Constructs a QProcess object with the given parent.
- `void start(const QString &program, const QStringList &arguments, QIODevice::OpenMode mode = ReadWrite);`
 - Starts the given program in a new process, passing the command line arguments in arguments.
- `void close();`
 - Closes all communication with the process and kills it. After calling this function, QProcess will no longer emit readyRead(), and data can no longer be read or written.
- `bool waitForStarted(int msecs = 30000);`
 - Blocks until the process has started and the started() signal has been emitted, or until msecs milliseconds have passed.

Warning: Calling this function from the main (GUI) thread might cause your user interface to freeze.

Header:	#include <QProcess>
qmake:	QT += core

QProcessEnvironment

- The QProcessEnvironment class holds the environment variables that can be passed to a program.

Methods

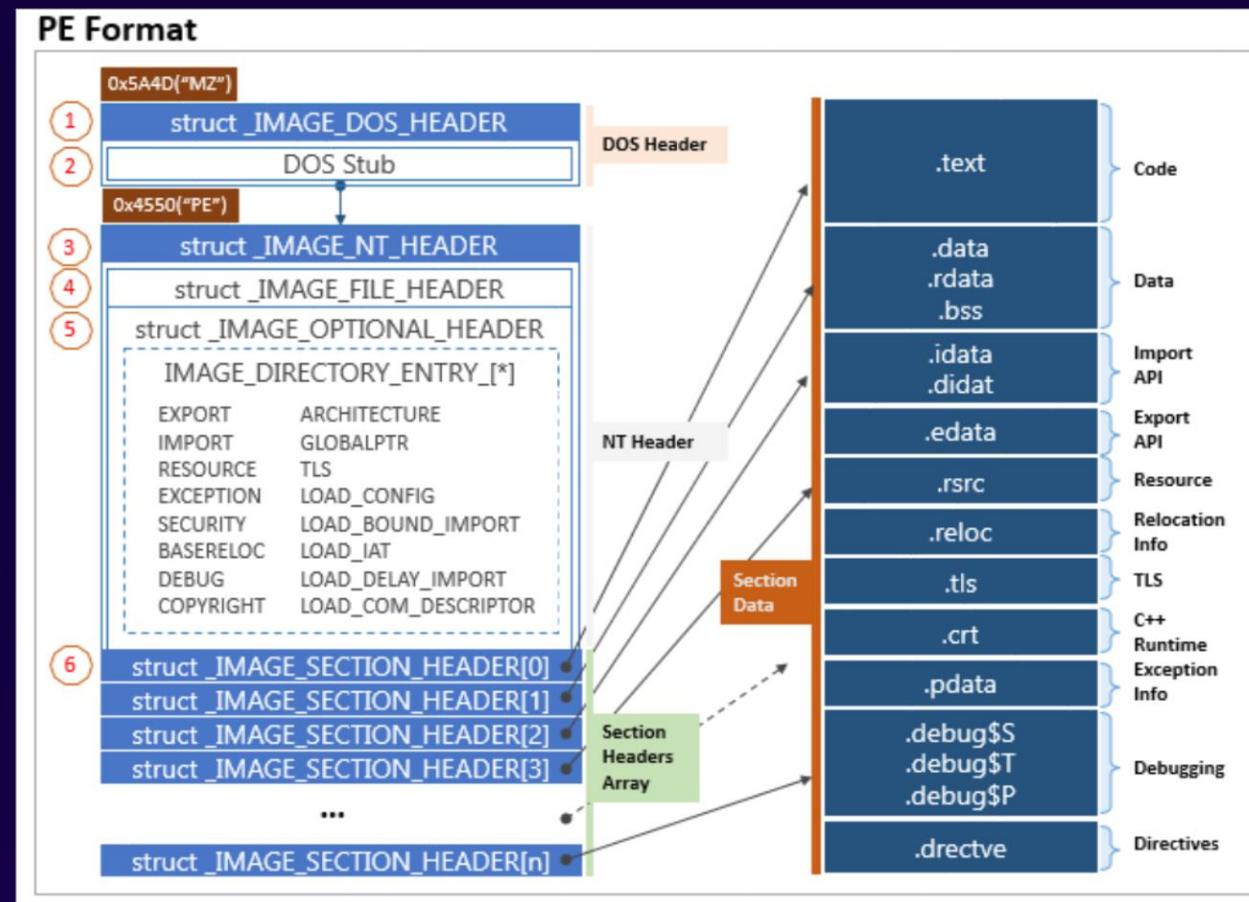
- **QProcessEnvironment(const QProcessEnvironment &other);**
 - Creates a QProcessEnvironment object that is a copy of other.
- **void insert(const QString &name, const QString &value);**
 - Inserts the environment variable of name name and contents value into this QProcessEnvironment object. If that variable already existed, it is replaced by the new value.
- **QProcessEnvironment systemEnvironment();**
 - The systemEnvironment function returns the environment of the calling process.

Header: #include <QProcessEnvironment>

qmake: QT += core

Resources

- Resources are part of executable files or libraries.



The Qt Resource System

- The Qt resource system is a platform-independent mechanism for storing binary files in the application's executable. This is useful if your application always needs a certain set of files (icons, translation files, etc.) and you don't want to run the risk of losing the files.

Resource Collection Files (.qrc)

- The resources associated with an application are specified in a .qrc file, an XML-based file format that lists files on the disk and optionally assigns them a resource name that the application must use to access the resource.

```
<!DOCTYPE RCC><RCC version="1.0">
<qresource>
    <file>images/copy.png</file>
    <file>images/cut.png</file>
    <file>images/new.png</file>
    <file>images/open.png</file>
    <file>images/paste.png</file>
    <file>images/save.png</file>
</qresource>
</RCC>
```

Resource Collection Files (.qrc)

- The resource files listed in the .qrc file are files that are part of the application's source tree.
- Resource data can either be compiled into the binary and thus accessed immediately in application code, or a binary resource can be created and at a later point in application code registered with the resource system.
- By default, resources are accessible in the application under the same file name as they have in the source tree, with a :/ prefix, or by a URL with a qrc scheme.

```
:/images/cut.png
```

- This can be changed using the file tag's alias attribute:

```
<file alias="cut-img.png">images/cut.png</file>
```

- The file is then accessible as :/cut-img.png from the application.
- It is also possible to specify a path prefix for all files in the .qrc file using the qresource tag's prefix attribute:

```
<qresource prefix="/myresources">
    <file alias="cut-img.png">images/cut.png</file>
</qresource>
```

- In this case, the file is accessible as :/myresources/cut-img.png.

Resource Collection Files (.qrc)

- Some resources need to change based on the user's locale, such as translation files or icons. This is done by adding a lang attribute to the qresource tag, specifying a suitable locale string. For example:

```
<qresource>
    <file>cut.jpg</file>
</qresource>
<qresource lang="fr">
    <file alias="cut.jpg">cut_fr.jpg</file>
</qresource>
```

- If the user's locale is French (i.e., `QLocale::system().name()` returns "fr_FR"), `:/cut.jpg` becomes a reference to the `cut_fr.jpg` image. For other locales, `cut.jpg` is used.

Resource Collection Files (.qrc)

- Some resources need to change based on the user's locale, such as translation files or icons. This is done by adding a lang attribute to the qresource tag, specifying a suitable locale string. For example:

```
<qresource>
    <file>cut.jpg</file>
</qresource>
<qresource lang="fr">
    <file alias="cut.jpg">cut_fr.jpg</file>
</qresource>
```

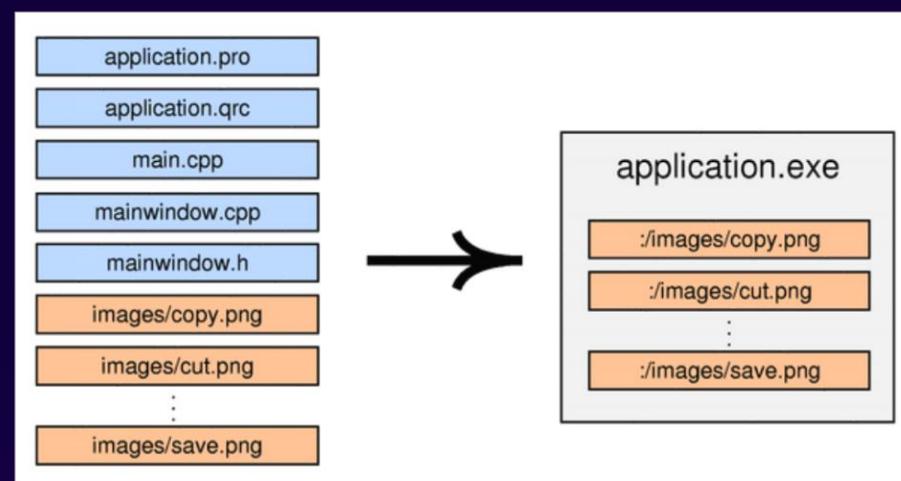
- If the user's locale is French (i.e., `QLocale::system().name()` returns "fr_FR"), `:/cut.jpg` becomes a reference to the `cut_fr.jpg` image. For other locales, `cut.jpg` is used.

Compiled-In Resources

- For a resource to be compiled into the binary the .qrc file must be mentioned in the application's .pro file so that qmake knows about it. For example:

```
RESOURCES = application.qrc
```

- qmake will produce make rules to generate a file called qrc_application.cpp that is linked into the application. This file contains all the data for the images and other resources as static C++ arrays of compressed binary data. The qrc_application.cpp file is automatically regenerated whenever the .qrc file changes or one of the files that it refers to changes. If you don't use .pro files, you can either invoke rcc manually or add build rules to your build system.



External Binary Resources

- For an external binary resource to be created you must create the resource data (commonly given the .rcc extension) by passing the -binary switch to rcc. Once the binary resource is created you can register the resource with the QResource API.

```
rcc -binary myresource.qrc -o myresource.rcc
```

- In the application, this resource would be registered with code like this:

```
QResource::registerResource("/path/to/myresource.rcc");
```

Header:	#include <QResource>
qmake:	QT += core

Compression

- rcc attempts to compress the content to optimize disk space usage in the final binaries. By default, it will perform a heuristic check to determine whether compressing is worth it and will store the content uncompressed if it fails to sufficiently compress. To control the threshold, you can use the `-threshold` option, which tells rcc the percentage of the original file size that must be gained for it to store the file in compressed form.
- The default value is "70", indicating that the compressed file must be 70% smaller than the original (no more than 30% of the original file size).

```
rcc -threshold 25 myresources.qrc
```

- It is possible to turn off compression, if desired. This can be useful if your resources already contain a compressed format, such as .png files, and you do not want to incur the CPU cost at build time to confirm that it can't be compressed. Another reason is if disk usage is not a problem and the application would prefer to keep the content as clean memory pages at runtime. You do this by giving the `-no-compress` command line argument.

```
rcc -no-compress myresources.qrc
```

Compression

- rcc also gives you some control over the compression level and compression algorithm, for example:

```
rcc -compress 2 -compress-algo zlib myresources.qrc
```

- It is also possible to use threshold, compress, and compress-algo as attributes in a .qrc file tag.

```
<qresource>
    <file compress="1" compress-algo="zstd">data.txt</file>
</qresource>
```

- The above will select the zstd algorithm with compression level 1.
- rcc supports the following compression algorithms and compression levels:
 - best
 - zstd
 - zlib

Using Resources in a Library

- If you have resources in a library, you need to force initialization of your resources by calling `Q_INIT_RESOURCE()` with the base name of the .qrc file. For example:

```
MyClass::MyClass() : BaseClass()
{
    Q_INIT_RESOURCE(resources);

    QFile file(":/myfile.dat");
    ...
}
```

- Similarly, if you must unload a set of resources explicitly (because a plugin is being unloaded or the resources are not valid any longer), you can force removal of your resources by calling `Q_CLEANUP_RESOURCE()` with the same base name as above.
- Note: The use of `Q_INIT_RESOURCE()` and `Q_CLEANUP_RESOURCE()` is not necessary when the resource is built as part of the application.

Linguist

- Qt provides excellent support for translating Qt C++ and Qt Quick applications into local languages. Release managers, translators, and developers can use Qt tools to accomplish their tasks.

Release Manager

- Two tools are provided for the release manager: lupdate and lrelease. These tools can process qmake project files, or operate directly on the file system.

Creating Translation Files

1. Run lupdate to generate the first set of translation source (TS) files with all the user-visible text but no translations.
2. Give the TS files to the translator who adds translations using Qt Linguist. Qt Linguist takes care of any changed or deleted source text.
3. Run lupdate to incorporate any new text added to the application. lupdate synchronizes the user-visible text from the application with the translations. It does not destroy any data.
4. To release the application, run lrelease to read the TS files and produce the QM files used by the application at runtime.

Add translations to the project

- In your qmake project file, the following variable TRANSLATIONS has to be added and must contain all language files you want to create initially.

```
TRANSLATIONS = languages/TranslationExample_en.ts  
languages/TranslationExample_fa.ts
```

Using lupdate

- The lupdate command line tool finds the translatable strings in the specified source, header and Qt Designer interface files, and produces or updates .ts translation files. The files to process and the files to update can be set at the command line, or provided in a .pro file specified as a command line argument. The developer creates the .pro file, as described in Qt Linguist Manual: Developers.

```
lupdate myproject.pro
```

Using lrelease

- The lrelease command line tool produces QM files out of TS files. The QM file format is a compact binary format that is used by the localized application. It provides extremely fast lookups for translations. The TS files lrelease processes can be specified at the command line, or given indirectly by a Qt .pro project file.

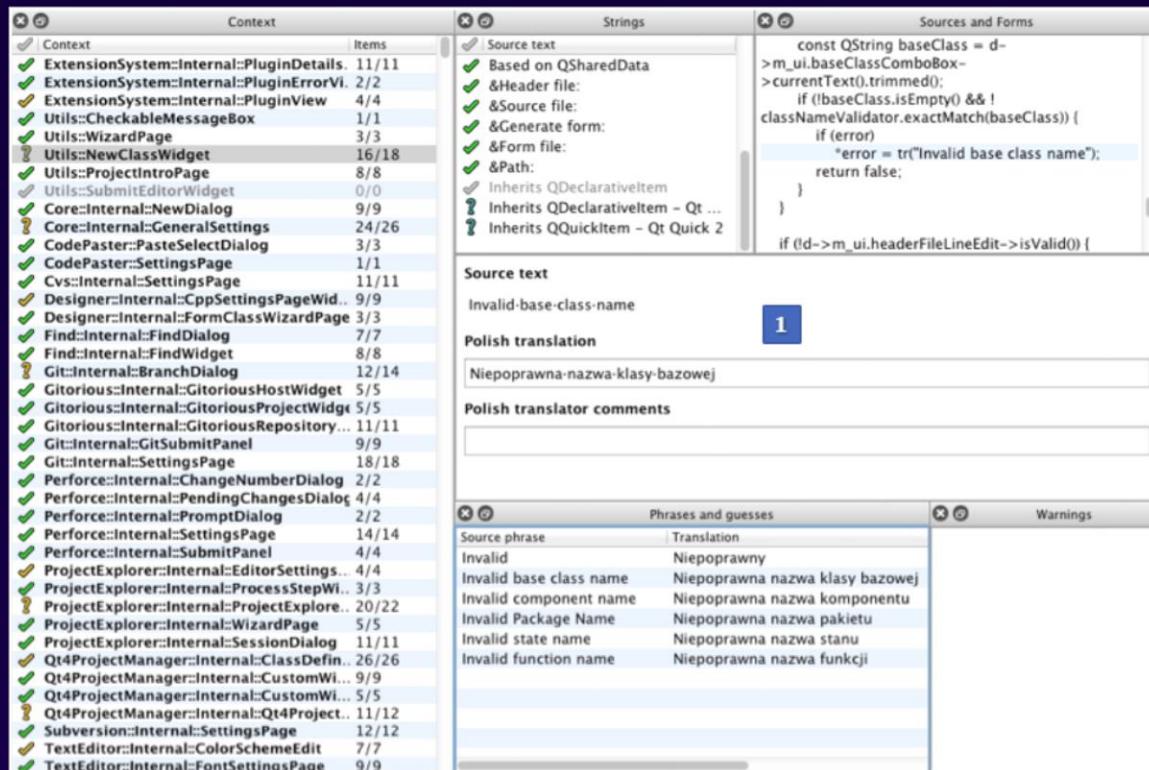
```
lrelease myproject.pro
```

- lrelease can be also be run without specifying a .pro file:

```
lrelease.exe main_en.ts languages\main_fr.ts
```

Translators (Qt Linguist)

- Qt Linguist is a tool for adding translations to Qt applications. Once you have installed Qt, you can start Qt Linguist in the same way as any other application on the development host.



Translators (Qt Linguist)

- The Qt Linguist main window contains a menu bar and the following views:
 1. Context (F6) for selecting from a list of contexts in which strings to be translated appear.
 2. Strings (F7) for viewing translatable strings found in a context.
 3. Sources and Forms (F9) for viewing the context where the current string is used if the source code for the context is accessible.
 4. Translation area for translating strings.
 5. Phrases and Guesses (F10) for viewing possible translations for the current string.
 6. Warnings (F8) for viewing translated strings that fail validation tests.

Translators (Qt Linguist)

- Changing the Target Locale
 - You can set the locale information explicitly in Edit > Translation File Settings. If the target language and country are not explicitly set when you open a translation source file, Qt Linguist attempts to deduct them from the translation source file name. This requires that the translation files adhere to the following file name convention: appname_language[_country].ts



Translators (Qt Linguist)

- Selecting Context to Translate
 - The Context view lists the contexts in which strings to be translated appear.

State	Icon	Description
Accepted/Correct	✓	All strings in the context have been translated, and all the translations passed the validation tests.
Accepted/Warnings	✓	All strings in the context have been translated or marked as translated, but at least one translation failed the validation tests. In the Strings view, you can see which string failed the test.
Not Accepted	?	At least one string in the context has not been translated or is not marked as translated.
Obsolete	✓	None of the translated strings appears in the context any more. This usually means the context itself no longer exists in the application.

Translators (Qt Linguist)

- Selecting String to Translate
 - The Strings view lists all the translatable strings found in the current context and their translation acceptance state. Selecting a string makes that string the current string in the translation area.

State	Icon	Description
Accepted/Correct	✓	The source string has a translation (possibly empty). The user has accepted the translation, and the translation passes all the validation tests. If the translation is empty, the user has chosen to leave it empty. Click the icon to revoke acceptance of the translation and decrement the number of accepted translations in the Items column of the gui Context view by 1. The state is reset to Not Accepted if the string has a translation, or to No Translation if the string's translation is empty. If 1 update changes the contents of a string, its acceptance state is automatically reset to Not Accepted.
Accepted/Warnings	⚠	The user has accepted the translation, but the translation does not pass all the validation tests. The validation test failures are shown in the Warnings view. Click the icon to revoke acceptance of the translation. The state is reset to Validation Failures, and the number of accepted translations in the Items column of the Context view is decremented by 1.
Not Accepted	?	The string has a translation that passes all the validation tests, but the user has not yet accepted the translation. Click the icon or press Ctrl+Enter to accept the translation. The state is reset to Accepted/Correct, and the number of accepted translations in the Items column of the Context view is incremented by 1.
No Translation	?	The string does not have a translation. Click the icon to accept the empty translation anyway. The state is reset to Accepted/Correct, and the number of accepted translations in the Items column of the Context view is incremented by 1.
Validation Failures	!	The string has a translation, but the translation does not pass all the validation tests. Validation test failures are shown in the Warnings view. Click on the icon or press Ctrl+Return to accept the translation even with validation failures. The state is reset to Accepted/Warnings. We recommended editing the translation to fix the causes of the validation failures. The state will reset automatically to Not Accepted, when all the failures have been fixed.
Obsolete	🔗	The string is obsolete. It is no longer used in the context. See the Release Manager for instructions on how to remove obsolete messages from the file.

Developers

- Specifying Translation Sources in Qt Project Files
 - To enable release managers to use lupdate and lrelease, specify a .pro Qt project file. There must be an entry in the TRANSLATIONS section of the project file for each language that is additional to the native language. A typical entry looks like this:

```
TRANSLATIONS = arrowpad_fr.ts \
               arrowpad_nl.ts
```

Developers

- Specifying Translation Sources in Qt Project Files
 - The lupdate tool extracts user interface strings from your application. It reads the application .pro file to identify which source files contain text to be translated. This means your source files must be listed in the SOURCES or HEADERS entry in the .pro file, or in resource files listed in the RESOURCE entry. If your files are not listed, the text in them will not be found.

```
HEADERS      = main-dlg.h \
               options-dlg.h
SOURCES      = main-dlg.cpp \
               options-dlg.cpp \
               main.cpp
FORMS        = search-dlg.ui
TRANSLATIONS = superapp_dk.ts \
               superapp_fi.ts \
               superapp_no.ts \
               superapp_se.ts
```

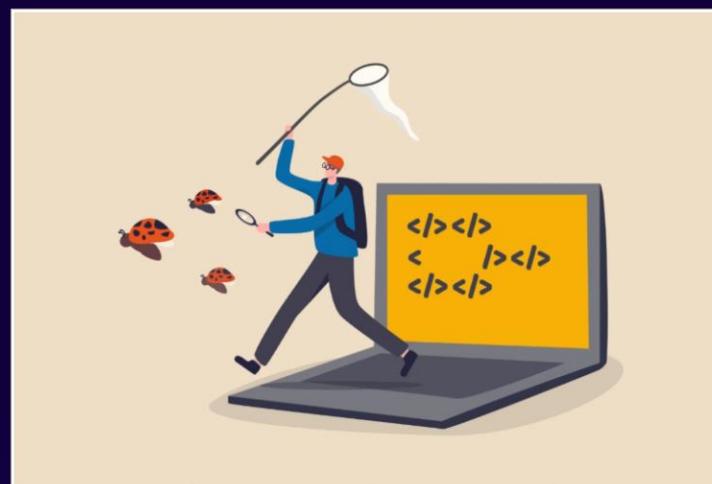
Developers

- Translation of messages
- Deploying Translations
 - The .qm files required for the application should be placed in a location where the loader code using QTranslator can locate them. Typically, this is done by specifying a path relative to QCOREAPPLICATION::applicationDirPath().
- Use Translation

```
QTranslator translator;  
  
translator.load("fa_IR.qm");  
app.installTranslator(&translator);
```

What is Debug and Release build

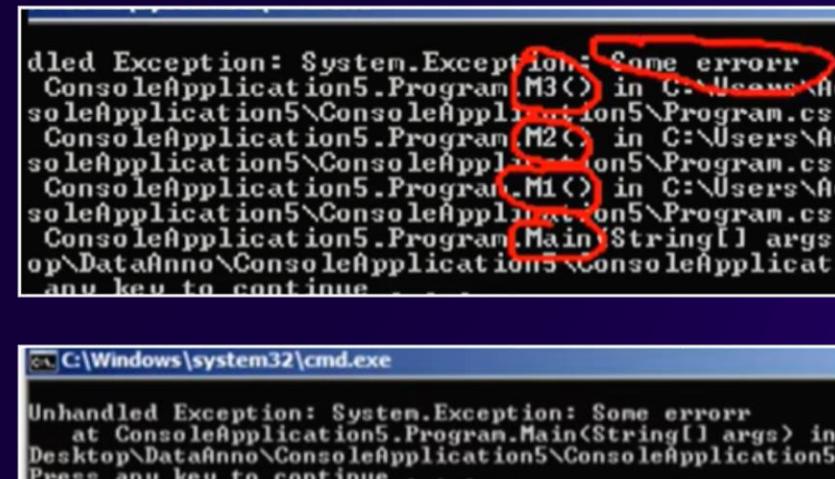
- Debug mode and Release mode are different configurations for building your project. Programmers generally use the Debug mode for debugging step by step their project and select the Release mode for the final build of Assembly file (.dll or .exe).
- The Debug mode does not optimize the binary it produces because the relationship between source code and generated instructions is more complex. This allows breakpoints to be set accurately and allows a programmer to step through the code one line at a time. The Debug configuration of your program is compiled with full symbolic debug information which help the debugger figure out where it is in the source code.



Difference between a Debug and Release build

- **Priority of the build mode:**
 - Debug Mode: When we are developing the application.
 - Release Mode: When we are going to production mode or deploying the application to the server.
- **Code optimization:**
 - Debug Mode: The debug mode code is not optimized.
 - Release Mode: The release mode code is optimized.
 - Run this program in debug mode and see your output window, In release mode, we cannot see the complete stack trace.

```
1 class program
2 {
3     static void main (string[] args) { M1(); }
4     static void M1() { M2(); }
5     static void M2() { M3(); }
6     static void M3() {
7         throw new Exception("some error");
8     }
9 }
```



The screenshot shows a Windows command prompt window titled 'C:\Windows\system32\cmd.exe'. It displays a stack trace for a 'System.Exception' exception. The stack trace is circled in red and shows multiple frames from 'ConsoleApplication5.Program'. The text in the window is as follows:

```
dled Exception: System.Exception: Some error
  ConsoleApplication5.Program.M3() in C:\Users\Ad
  soleApplication5\ConsoleApplication5\Program.cs
  ConsoleApplication5.Program.M2() in C:\Users\Ad
  soleApplication5\ConsoleApplication5\Program.cs
  ConsoleApplication5.Program.M1() in C:\Users\Ad
  soleApplication5\ConsoleApplication5\Program.cs
  ConsoleApplication5.Program.Main<String[] args>
op\Data\anno\ConsoleApplication5\ConsoleApplicati
anu key to continue . . .
```

Below the window, the command prompt prompt is visible: 'C:\Windows\system32\cmd.exe'.

Unhandled Exception: System.Exception: Some error
 at ConsoleApplication5.Program.Main<String[] args> in
Desktop\Data\anno\ConsoleApplication5\ConsoleApplication5'

Difference between a Debug and Release build

- **Performance:**
 - Debug Mode: In debug mode the application will be slow.
 - Release Mode: In release mode the application will be faster.
- **Debug Symbols:**
 - Debug Mode: In the debug mode code, which is under the debug, symbols will be executed.
 - Release Mode: In release mode code, which is under the debug, symbols will not be executed.

Debug Symbol

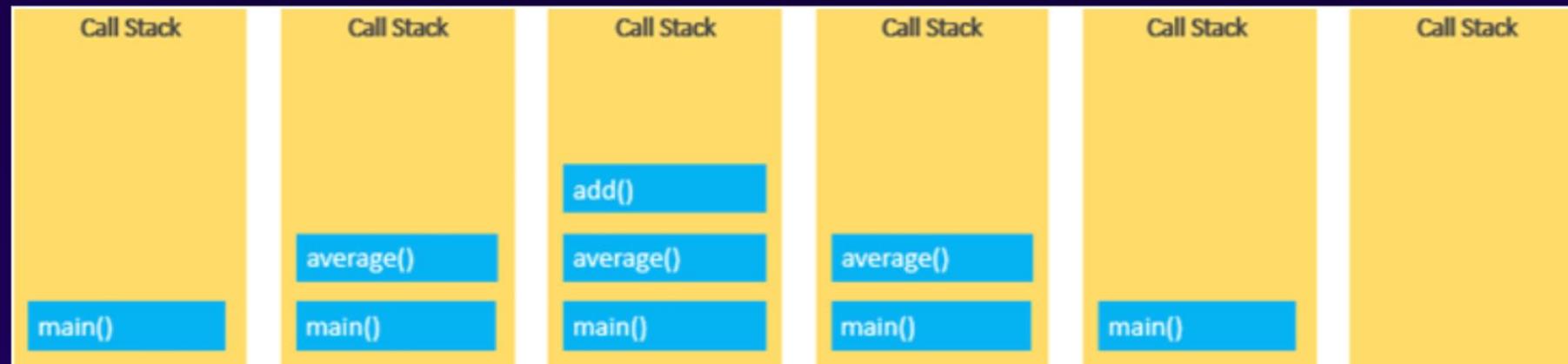
- A debug symbol is a special kind of symbol that attaches additional information to the symbol table of an object file, such as a shared library or an executable. This information allows a symbolic debugger to gain access to information from the source code of the binary, such as the names of identifiers, including variables and routines.
- The symbolic information may be compiled together with the module's binary file, or distributed in a separate file, or simply discarded during the compilation and/or linking.
- This information can be helpful while trying to investigate and fix a crashing application or any other fault.

Call Stack

- Call stack is a stack data structure that stores information about the active subroutines of a computer program. This kind of stack is also known as an execution stack, program stack, control stack, run-time stack, or machine stack, and is often shortened to just "the stack". Although maintenance of the call stack is important for the proper functioning of most software, the details are normally hidden and automatic in high-level programming languages. Many computer instruction sets provide special instructions for manipulating stacks.
- A call stack is used for several related purposes, but the main reason for having one is to keep track of the point to which each active subroutine should return control when it finishes executing. An active subroutine is one that has been called, but is yet to complete execution, after which control should be handed back to the point of call. Such activations of subroutines may be nested to any level (recursive as a special case), hence the stack structure. For example, if a subroutine DrawSquare calls a subroutine DrawLine from four different places, DrawLine must know where to return when its execution completes. To accomplish this, the address following the instruction that jumps to DrawLine, the return address, is pushed onto the top of the call stack with each call.

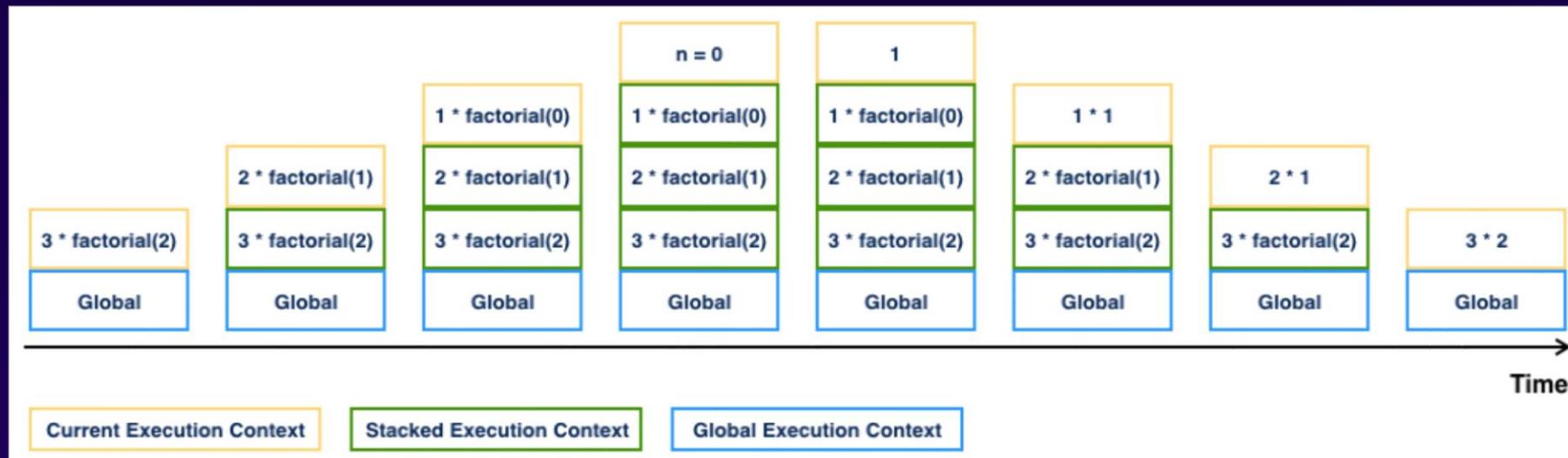
Call Stack

```
1 function add(a, b) {  
2     return a + b;  
3 }  
4  
5 function average(a, b) {  
6     return add(a, b) / 2;  
7 }  
8  
9 Function main() {  
10    let x = average(10, 20);  
11 }
```



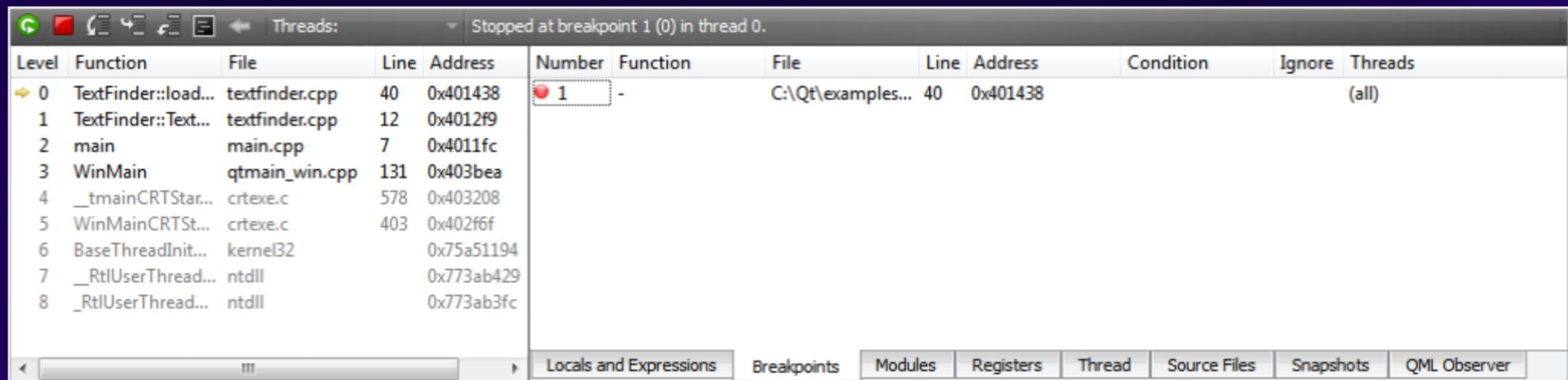
Call Stack

```
1 function factorial(n) {  
2     return n === 0 ? 1 : n * factorial(n - 1);  
3 }  
4  
5 factorial(10);
```



Stack Trace

- A stack trace is a report that provides information about program subroutines. It is commonly used for certain kinds of debugging, where a stack trace can help software engineers figure out where a problem lies or how various subroutines work together during execution.

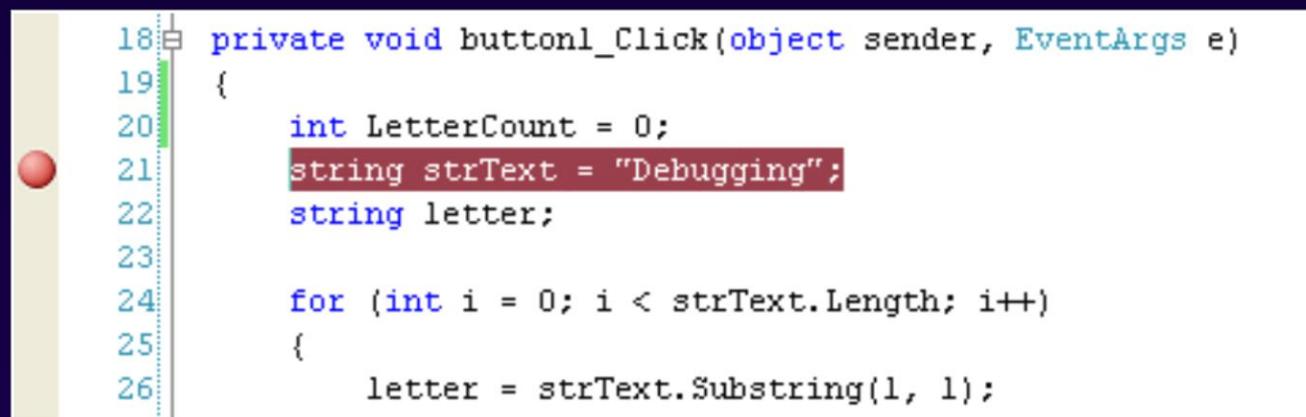


The screenshot shows the Qt Creator debugger interface with a stack trace window. The title bar says "Threads: Stopped at breakpoint 1 (0) in thread 0.". The main window displays two tables: one for the current thread (thread 0) and one for the current stack frame (frame 1). The left table lists the call stack from level 0 to 8, showing function names, file paths, line numbers, and addresses. The right table shows the current stack frame details. At the bottom, there are tabs for Locals and Expressions, Breakpoints, Modules, Registers, Thread, Source Files, Snapshots, and QML Observer, with "Breakpoints" currently selected.

Level	Function	File	Line	Address	Number	Function	File	Line	Address	Condition	Ignore	Threads
0	TextFinder::load...	textfinder.cpp	40	0x401438	1	-	C:\Qt\examples...	40	0x401438	(all)		
1	TextFinder::Text...	textfinder.cpp	12	0x4012f9								
2	main	main.cpp	7	0x4011fc								
3	WinMain	qtmain_win.cpp	131	0x403bea								
4	__tmainCRTStar...	crtexe.c	578	0x403208								
5	WinMainCRTSt...	crtexe.c	403	0x402f6f								
6	BaseThreadInit...	kernel32		0x75a51194								
7	__RtlUserThread...	ntdll		0x773ab429								
8	__RtlUserThread...	ntdll		0x773ab3fc								

Break Points

- In software development, a breakpoint is an intentional stopping or pausing place in a program, put in place for debugging purposes. It is also sometimes simply referred to as a pause.
- More generally, a breakpoint is a means of acquiring knowledge about a program during its execution. During the interruption, the programmer inspects the test environment (general purpose registers, memory, logs, files, etc.) to find out whether the program is functioning as expected. In practice, a breakpoint consists of one or more conditions that determine when a program's execution should be interrupted.



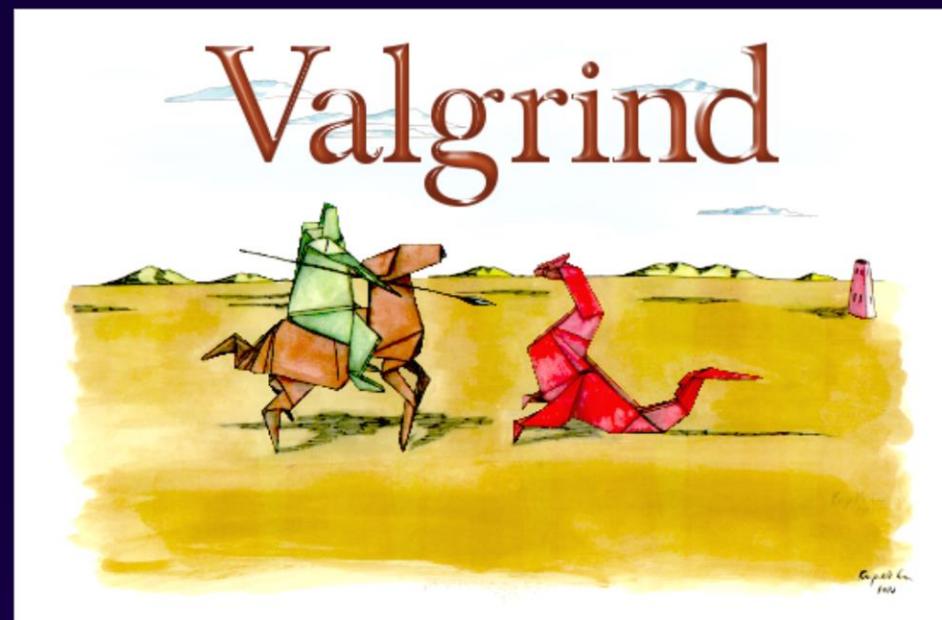
```
18:  private void button1_Click(object sender, EventArgs e)
19:  {
20:      int LetterCount = 0;
21:      string strText = "Debugging";
22:      string letter;
23:
24:      for (int i = 0; i < strText.Length; i++)
25:      {
26:          letter = strText.Substring(i, 1);
```

Debugging in Qt Creator

- Step Over
- Step Into
- Step Out
- Run to Line
- Jump to Line
- Local and Expressions
- Memory View
- Instruction Mode
- Toggle Breakpoint
- Custom Breakpoint

Introduction

- The Valgrind tool suite provides a number of debugging and profiling tools that help you make your programs faster and more correct. The most popular of these tools is called Memcheck. It can detect many memory-related errors that are common in C and C++ programs and that can lead to crashes and unpredictable behaviour.
- Valgrind is a runtime debugging tool.



How to install in Linux

- Online

```
$sudo apt-get install --download-only valgrind
```

- Offline

```
$sudo dpkg -i *.deb
```

Preparing your program

- Compile your program with -g to include debugging information so that Memcheck's error messages include exact line numbers. Using -O0 is also a good idea, if you can tolerate the slowdown. With -O1 line numbers in error messages can be inaccurate, although generally speaking running Memcheck on code compiled at -O1 works fairly well, and the speed improvement compared to running -O0 is quite significant. Use of -O2 and above is not recommended as Memcheck occasionally reports uninitialised-value errors which don't really exist.

Running your program under Memcheck

- If you normally run your program like this:

```
$myprog arg1 arg2
```

- Memcheck is the default tool. The --leak-check option turns on the detailed memory leak detector.
- Your program will run much slower (eg. 20 to 30 times) than normal, and use a lot more memory. Memcheck will issue messages about memory errors and leaks that it detects.

```
$valgrind --leak-check=yes myprog arg1 arg2
```

Serial Transmission

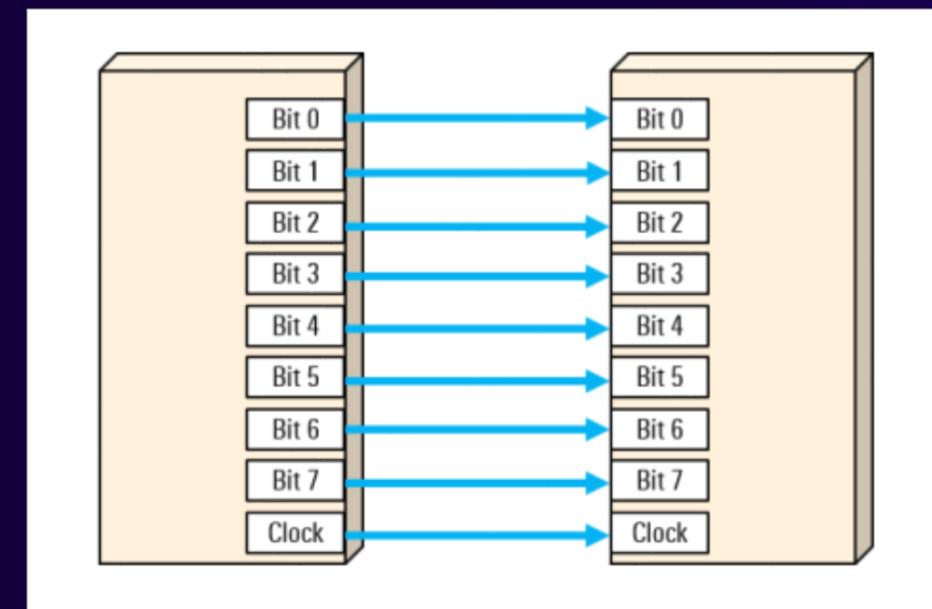
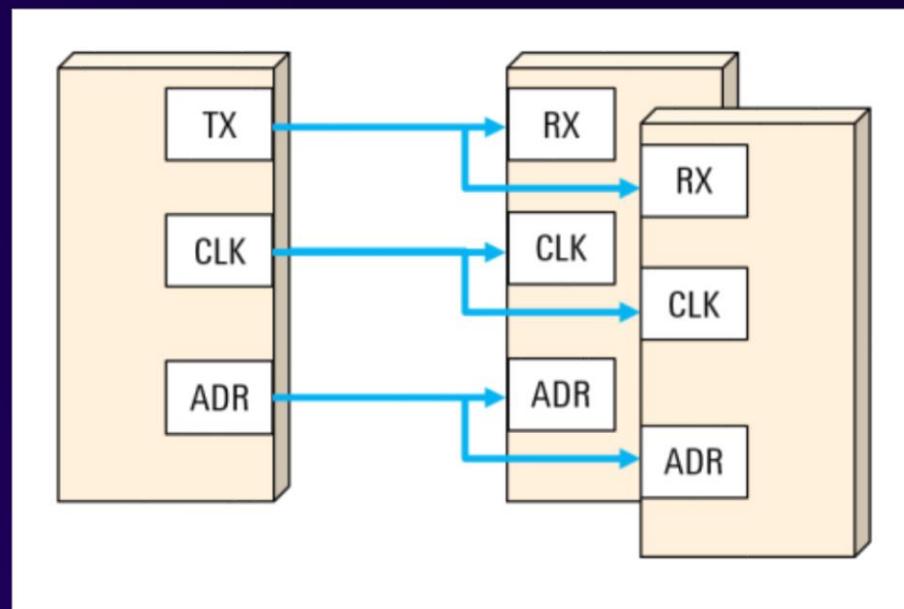
- As the name implies, serial transmission sends one bit at a time, with all the bits being sent over the same conductor. Serial transmission works well for longer-distance applications, applications needing higher throughput, and applications where there are multiple nodes. All of this comes at a cost, with serial transmission being more complex and harder to analyze. While it is true that data bits are being sent over a single conductor or “wire,” most serial protocols use multiple conductors.

Parallel Transmission

- Parallel transmission moves multiple bits simultaneously between transmitter and receiver, usually with a separate conductor per bit. Parallel connections work well for short-distance and/or point-to-point connections. They have simple timing and are relatively easy to analyze. But as popular as parallel transmission once was, it's now largely being replaced by serial transmission.

Serial vs Parallel

- In addition to the wire for the data bits, many protocols also add a clock signal, some type of control or addressing function for multiple nodes.



Serial Communication Protocols

- Serial protocols are used in a very wide variety of applications. The three main serial protocols used for generic applications are UART/RS232, I²C/SPI.
- A special category of serial protocols are those used in the automotive industry, such as CAN/LIN and FlexRay™.

Generic applications

- UART (Universal Asynchronous Receiver/Transmitter)
 - Classic serial protocol
 - easy to implement
 - has been used in PC serial and COM ports for decades
- I²C (Inter-IC)
 - Communication between integrated circuits (and more)
- SPI (Serial Peripheral Interface)

Automotive applications

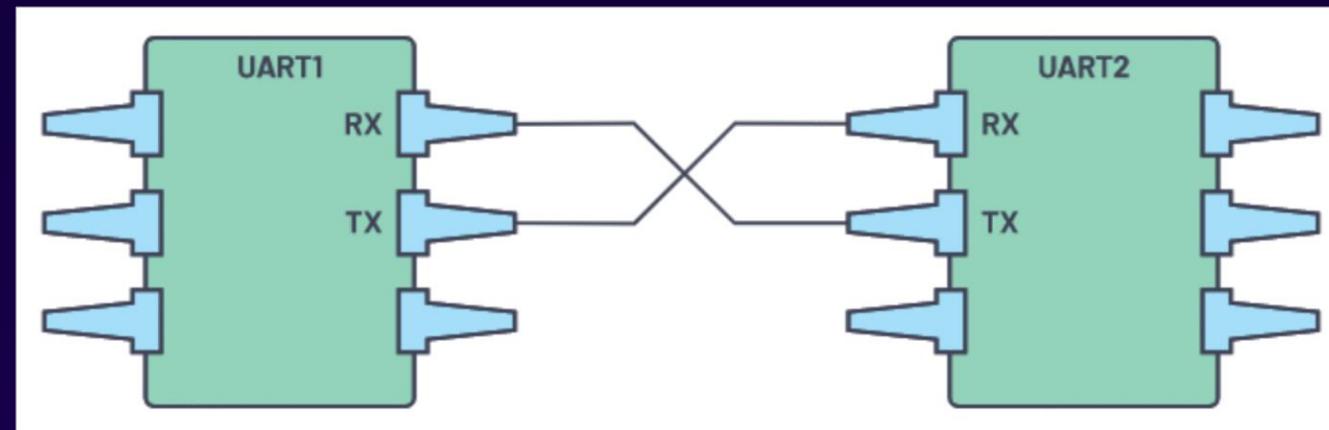
- CAN (Controller Area Network)
 - High Speed
 - often used with sensors
- LIN (Local Interconnect Network)
 - Lower speed
 - used with accessories (e.g. windows, mirrors)
- FlexRay
 - Higher speed with redundancy
 - Higher Speed than I²C
 - uses more wires and is generally more complex

Characteristics of serial protocols

- Although implementation details differ between protocols, all serial protocols have four basic characteristics, which are also important for the analysis and decode of serial data:
 - Levels: how voltages are used to represent zeros or ones.
 - Timing: how often bits are sent (bit time).
 - Framing: how bits are organized into groups and the role of each bit or group of bits.
 - Protocol: which messages are exchanged under which circumstances.

UART

- A Hardware Communication Protocol Understanding Universal Asynchronous Receiver/Transmitter.
- UART is a hardware communication protocol that uses asynchronous serial communication with configurable speed. Asynchronous means there is no clock signal to synchronize the output bits from the transmitting device going to the receiving end.
- Embedded systems, microcontrollers, and computers mostly use UART as a form of device-to-device hardware communication protocol. Among the available communication protocols, UART uses only two wires for its transmitting and receiving ends.



UART

- The two signals of each UART device are named:
 - Transmitter (Tx)
 - Receiver (Rx)
- The main purpose of a transmitter and receiver line for each device is to transmit and receive serial data intended for serial communication.

Baud Rate

- For UART and most serial communications, the baud rate needs to be set the same on both the transmitting and receiving device. The baud rate is the rate at which information is transferred to a communication channel. In the serial port context, the set baud rate will serve as the maximum number of bits per second to be transferred.
- Speed 9600, 19200, 38400, 57600, 115200, 230400, 460800, 921600, 1000000, 1500000
- The UART interface does not use a clock signal to synchronize the transmitter and receiver devices; it transmits data asynchronously. Instead of a clock signal, the transmitter generates a bitstream based on its clock signal while the receiver is using its internal clock signal to sample the incoming data. The point of synchronization is managed by having the same baud rate on both devices. Failure to do so may affect the timing of sending and receiving data that can cause discrepancies during data handling. The allowable difference of baud rate is up to 10% before the timing of bits gets too far off.

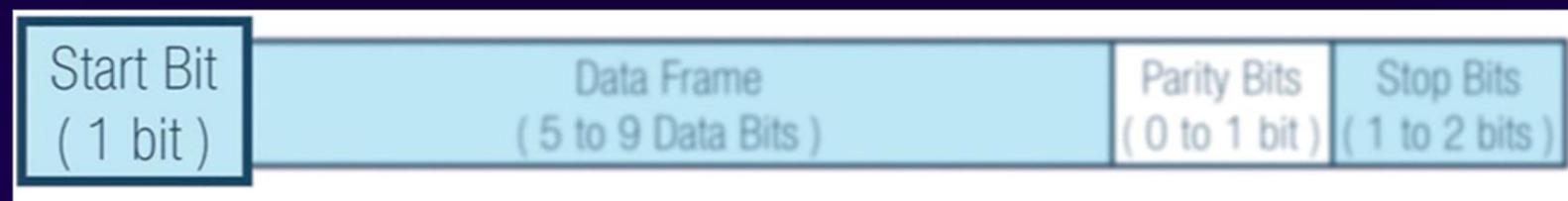
Uart Packet

- In UART, the mode of transmission is in the form of a packet. A packet consists of a start bit, data frame, a parity bit, and stop bits.



Uart Packet

- Start Bit
 - The UART data transmission line is normally held at a high voltage level when it's not transmitting data. To start the transfer of data, the transmitting UART pulls the transmission line from high to low for one (1) clock cycle. When the receiving UART detects the high to low voltage transition, it begins reading the bits in the data frame at the frequency of the baud rate.



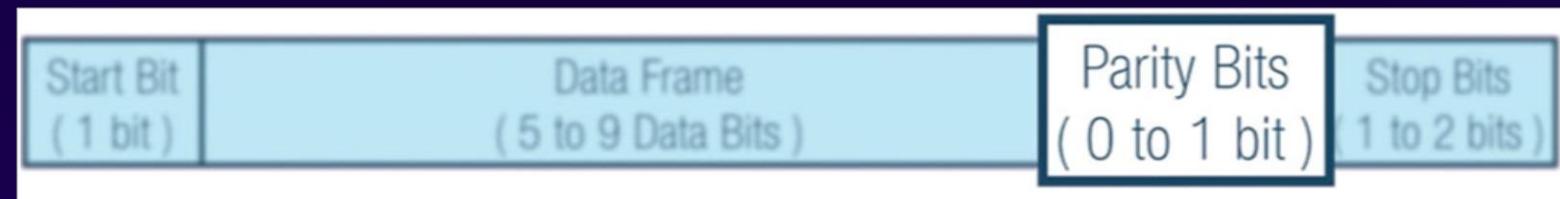
Uart Packet

- Data Frame
 - The data frame contains the actual data being transferred. It can be five (5) bits up to eight (8) bits long if a parity bit is used. If no parity bit is used, the data frame can be nine (9) bits long. In most cases, the data is sent with the least significant bit first.



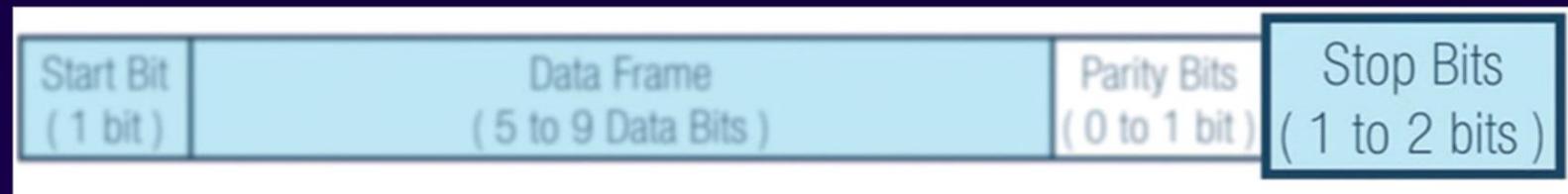
Uart Packet

- Parity
 - Parity describes the evenness or oddness of a number. The parity bit is a way for the receiving UART to tell if any data has changed during transmission. Bits can be changed by electromagnetic radiation, mismatched baud rates, or long-distance data transfers.
 - After the receiving UART reads the data frame, it counts the number of bits with a value of 1 and checks if the total is an even or odd number. If the parity bit is a 0 (even parity), the 1 or logic-high bit in the data frame should total to an even number. If the parity bit is a 1 (odd parity), the 1 bit or logic highs in the data frame should total to an odd number.
 - When the parity bit matches the data, the UART knows that the transmission was free of errors. But if the parity bit is a 0, and the total is odd, or the parity bit is a 1, and the total is even, the UART knows that bits in the data frame have changed.



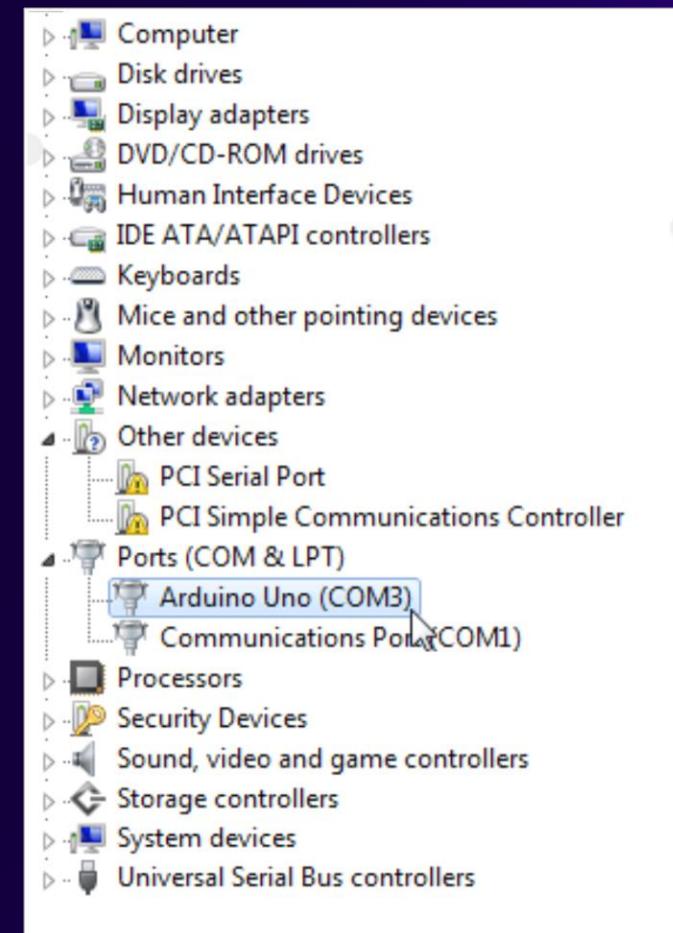
Uart Packet

- Stop Bits
 - To signal the end of the data packet, the sending UART drives the data transmission line from a low voltage to a high voltage for one (1) to two (2) bit(s) duration.



Serial port names in Windows and Linux

- Linux offers various tools and commands to access serial ports. Linux uses ttySx for a serial port device name. For example, COM1 (DOS/Windows name) is ttyS0, COM2 is ttyS1, and so on. USB based serial ports might use a name such as ttySUSB0. All these devices are located under /dev/ directory.
- Find the right serial port name Windows
 - Open Windows Device Manager.
 - Find "Ports (COM & LPT)" in the list.
 - Expand "Ports (COM & LPT)" to see the names of all serial ports.
- Find Port Number on Linux
 - Open terminal and type: ls /dev/tty*.
 - Note the port number listed for /dev/ttyUSB* or /dev/ttyACM*. The port number is represented with * here.



QSerialPort

- Qt Serial Port provides the basic functionality, which includes configuring, I/O operations, getting and setting the control signals of the RS-232 pinouts.
- Notes:
 - Serial protocol is hardware and does not have layering like network.
 - Only one program can open a particular serial port for sending and receiving at a time.
- Warning: The method `waitForReadyRead()` should be used before each `read()` call for the blocking approach, because it processes all the I/O routines instead of Qt event-loop.
- Warning: The method `waitForBytesWritten()` should be used after each `write()` call for the blocking approach, because it processes all the I/O routines instead of Qt event-loop.

Header:	#include <QSerialPort>
qmake:	QT += serialport

QSerialPortInfo

- Provides information about existing serial ports such as Port name, system location, description and manufacturer.

```
Header: #include <QSerialPortInfo>
```

```
qmake: QT += serialport
```

The Windows Deployment Tool

- The Windows deployment tool `windeployqt` is designed to automate the process of creating a deployable folder containing the Qt-related dependencies (libraries, QML imports, plugins, and translations) required to run the application from that folder. It creates an installation tree for Windows desktop applications, which can be easily bundled into an installation package.

`windeployqt`

- The tool can be found in `QTDIR/bin/windeployqt`. It needs to be run within the build environment in order to function correctly. When using Qt Installer, the script `QTDIR/bin/qtenv2.bat` should be used to set it up.
- `windeployqt` takes an `.exe` file or a directory that contains an `.exe` file as an argument, and scans the executable for dependencies. If a directory is passed with the `--qmldir` argument, `windeployqt` uses the `qmlimportscanner` tool to scan QML files inside the directory for QML import dependencies. Identified dependencies are then copied to the executable's directory.

windeployqt

```
C:\Windows\system32\cmd.exe
Microsoft Windows [Version 6.1.7601]
(c) Корпорация Майкрософт (Microsoft Corp.), 2009. Все права защищены.

C:\Users\Dekadent>d:
D:>cd AndroidQT\QT\5.5\mingw492_32\bin
D:\AndroidQT\QT\5.5\mingw492_32\bin>windeployqt.exe D:\EColor\EColor.exe
```

Thank You

