



The complete software development framework

# Qt 6 Programming

Second Edition, Ver 2.0

Qt Korea Developer Community  
[www.qt-dev.com](http://www.qt-dev.com)

Qt is highly intuitive, produces highly readable, maintainable and reusable code with high runtime performance and a small footprint, and support cross-platform.

Jesus loves you.

# Qt6 Programming

**Document version** Version 2.0 (13 June, 2024)

**Homepage URL** [www.qt-dev.com](http://www.qt-dev.com)

## Preface

We distribute this book free of charge to anyone interested in Qt.

If there is one thing I pray for, it is that those who are still unbelievers will hear the gospel and turn to our Father.

In addition, Jesus, His only begotten Son, came to this earth and took our sins upon Himself. By the precious blood of Jesus, our sins have been forgiven, and God the Father is waiting for all unbelievers to return to Him out of love for His children.

If any of you who have read this book are still unbelievers, we pray and bless you with our earnest hope that you will accept Jesus as your Saviour and be born again as a child of faith. We also pray that God's good grace will be with you. Amen.

37. Jesus replied: " 'Love the Lord your God with all your heart and with all your soul and with all your mind.'

38. This is the first and greatest commandment.

39. And the second is like it: 'Love your neighbor as yourself.'

Matthew 22:37~39

## Table of Contents

---

1. Introduction and installation of Qt .....	1
2. Useful development tools provided by Qt .....	16
3. Building a project with CMake.....	20
3.1. Implementing Console Applications with CMake.....	22
3.2. Implementing GUI Application with CMake .....	30
4. Building a project with qmake .....	38
4.1. Implementing Console applications with qmake .....	39
4.2. Implementing GUI application with qmake.....	46
5. Qt GUI Widgets .....	55
6. Layout.....	103
7. Data types and classes provided by Qt.....	108
8. Container Classes .....	120
9. Signal and Slot.....	125
10. Designing a GUI with Qt Designer .....	130
11. Dialog.....	140
12. Implementing a GUI with QMainWindow.....	152
13. Stream.....	157
14. File input and output.....	161
15. Qt Property.....	170
16. Model and View.....	176
17. 2D Graphics with QPainter Class .....	188

18.	Implementing Chromakey image processing with QPainter .....	210
19.	Timer .....	217
20.	Thread programming .....	221
21.	XML .....	229
22.	JSON.....	240
23.	Creating library (CMake).....	250
	23.1. Creating and Using Shared Libraries .....	251
	23.2. Building with Library .....	264
24.	Creating library (qmake).....	267
	24.1. Shared and Using Shared Library .....	269
	24.2. Building with Library .....	281
25.	D-Pointer.....	283
26.	Database Programming .....	300
27.	Qt for Android .....	320
	27.1. Android development environment on MS Windows.....	321
	27.2. Android development environment on Linux.....	338
28.	Essential Network Programming .....	351
	28.1. TCP Protocol-Based Server/Client Implementation .....	354
	28.2. Synchronous and asynchronous implementation .....	362
	28.3. Implement communication based on the UDP.....	370
	28.4. Broadcast.....	377
	28.5. Multicast.....	383
	28.6. Implementing chatting server and client.....	390

29.	<b>Qt WebSocket</b> .....	405
30.	<b>Qt WebEngine</b> .....	417
31.	<b>Qt HTTP Server</b> .....	431
32.	<b>Deployment</b> .....	443
33.	<b>Qt Graphics View Framework</b> .....	455
34.	<b>Animation Framework and State Machine</b> .....	466
35.	<b>Qt Chart</b> .....	479
36.	<b>Qt Data Visualization</b> .....	491
37.	<b>Inter Process Communication</b> .....	502
	37.1. Unix Domain Socket and Named pipe .....	503
	37.2. QProcess .....	514
	37.3. Shared Memory .....	520
	37.4. Qt D-Bus .....	531
38.	<b>Multimedia</b> .....	552
	38.1. Audio .....	554
	38.2. Video .....	581
	38.3. Camera .....	593
39.	<b>Serial Communication</b> .....	602
40.	<b>Qt Positioning</b> .....	615
41.	<b>Qt PDF</b> .....	620
42.	<b>Qt Printer Support</b> .....	626
43.	<b>Qt Pprotobuf</b> .....	632

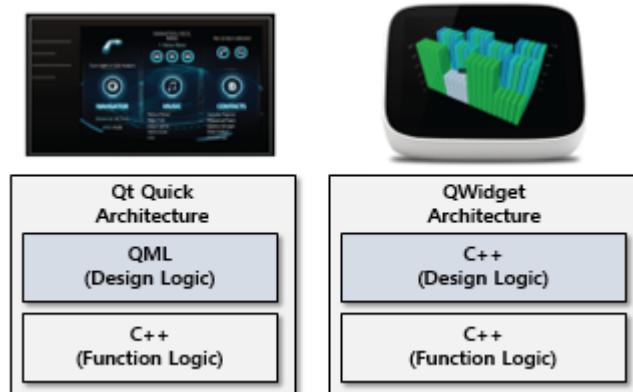
## 1. Introduction and installation of Qt

Qt provides the same development framework for developing applications on desktop-based operating systems such as MS Windows, Linux, and MacOS, which saves time and money in development.

The Qt framework also allows you to develop applications on Android (which uses the same kernel as Linux) and iOS mobile platforms using the same Qt framework.

In addition to desktop and mobile-based platforms, Qt can also be used to develop applications on embedded platforms, such as those embedded in small devices. The Qt framework can be used to develop applications on embedded Linux, QNX, and WinRT platforms using the Qt development framework.

Qt uses C++. (Python is also available, but Python is not covered here.) In addition to C++, the Qt framework provides Qt Quick (QML). Qt Quick uses an interpreter called QML. When developing an application with Qt, you can use QML to develop the GUI or UX. QML allows you to separate functional logic and design logic.



Design Logic refers to the GUI. Function Logic refers to the process or function of clicking a button on the GUI.

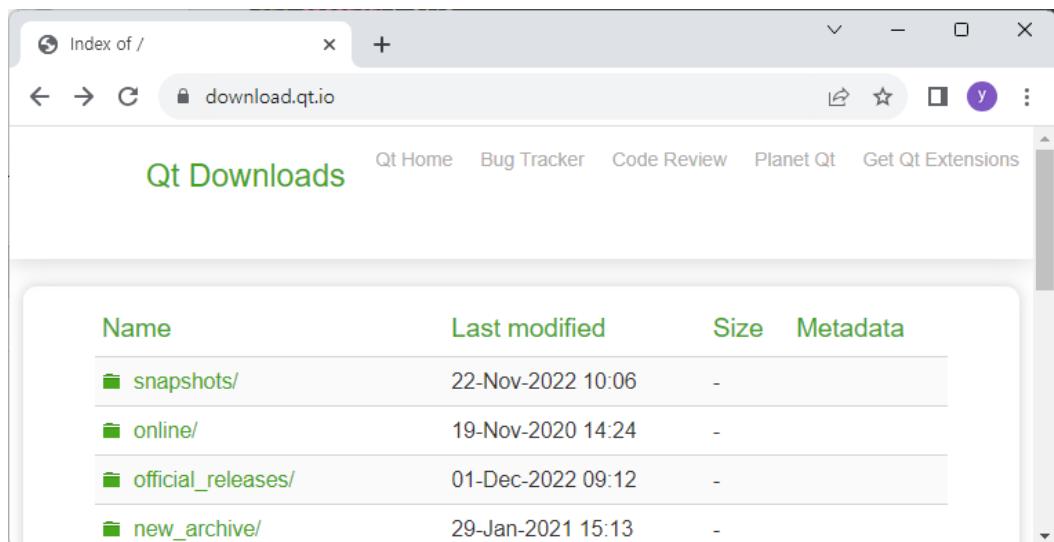
When developing an application with Qt, you can increase source code reusability by developing Function Logic in C++ and Design Logic in QML. For example, by developing the GUI screen in QML and the functions in C++, you can improve the reusability of the source code by separating the two logics.

There are advantages and disadvantages to developing GUIs in QML. If your GUI uses a touch screen and requires a lot of graphical effects, such as Android or iOS, QML is a good choice. However, if you're presenting a lot of information to the user, such as on a desktop, it's sometimes (but not always) better to develop your GUI in C++.

And if it's an embedded system with limited memory or low CPU (or GPU) performance, using C++ may have a performance advantage over using QML. Therefore, it is recommended that you first look at the characteristics of the software you are developing and decide whether you want to develop the GUI in C++ or use QML.

- Downloading the Qt online installation files

Qt is a desktop-based platform (or operating system) that supports MS Windows, Linux, and MacOS platforms. Qt can be downloaded for each platform from the Qt official website.



Click on the "official\_releases" item, as shown in the screen above. Next, click on the "online\_installers" item.

Jesus loves you.

A screenshot of a web browser window titled "Index of /official\_releases". The address bar shows "download.qt.io/official\_releases/". The page displays a list of files and directories:

Name	Last modified	Size	Metadata
qt/	03-Apr-2023 07:37	-	
qt-installer-framework/	07-Jun-2023 12:51	-	
qbs/	03-Aug-2023 17:32	-	
pyside/	30-Nov-2015 13:39	-	
online_installers/	07-Jun-2023 13:52	-	
jom/	12-Dec-2018 15:13	-	
gdb/	17-Nov-2014 13:42	-	
additional_libraries/	03-Mar-2021 10:18	-	
QtForPython/	12-Apr-2022 15:00	-	
timestamp.txt	09-Aug-2023 08:00	11	<a href="#">Details</a>

Click "online\_installers" to download platform-specific online installation files, as shown in the screen below.

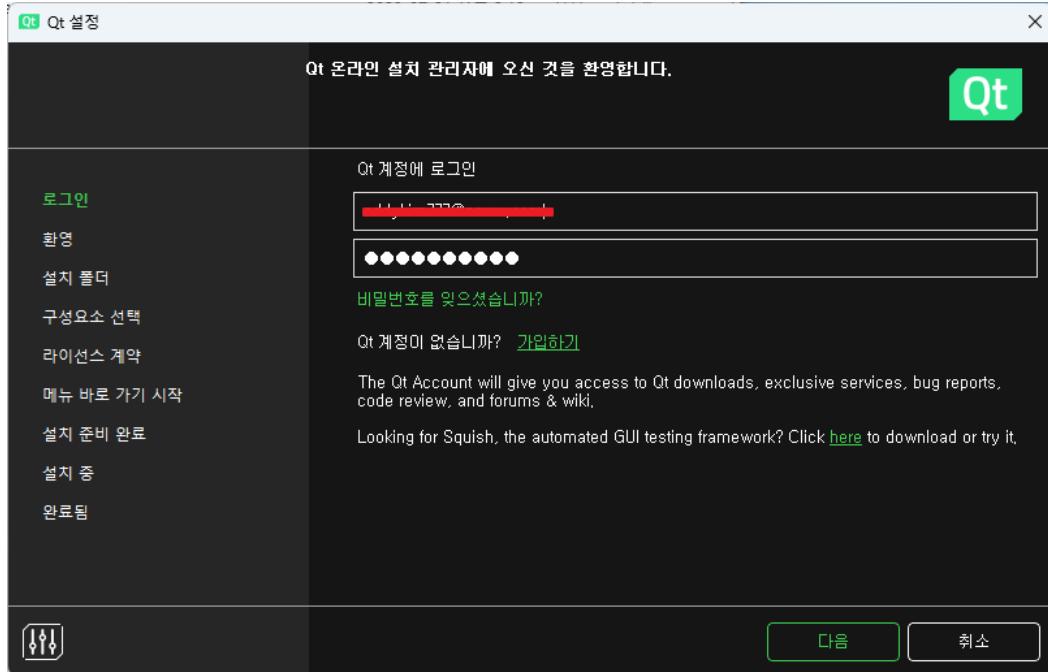
A screenshot of the "Qt Downloads" page from "download.qt.io". The URL in the address bar is "download.qt.io/official\_releases/online\_installers/". The page title is "Qt Downloads". The top navigation menu includes "Qt Home", "Bug Tracker", "Code Review", "Planet Qt", and "Get Qt Extensions". The main content area shows a table of files:

Name	Last modified	Size	Metadata
Parent Directory	-	-	
qt-unified-windows-x64-online.exe	07-Jun-2023 12:49	39M	<a href="#">Details</a>
qt-unified-mac-x64-online.dmg	07-Jun-2023 12:49	15M	<a href="#">Details</a>
qt-unified-linux-x64-online.run	07-Jun-2023 12:49	52M	<a href="#">Details</a>

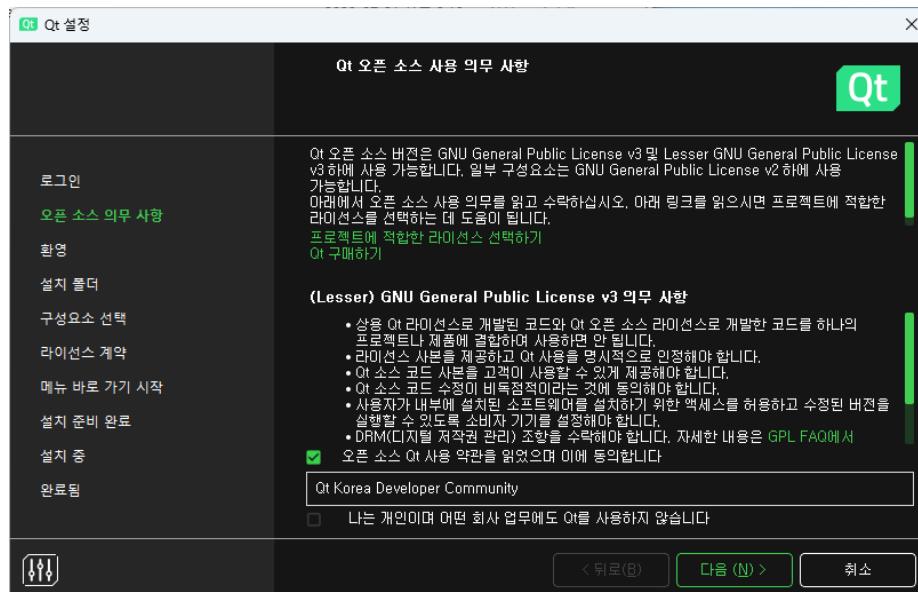
Files with the exe extension are online installation files for installation on MS Windows. Files with the dmg extension are for Mac OS. Files with the run extension are online installers for installation on Linux.

- Installing Qt on MS Windows

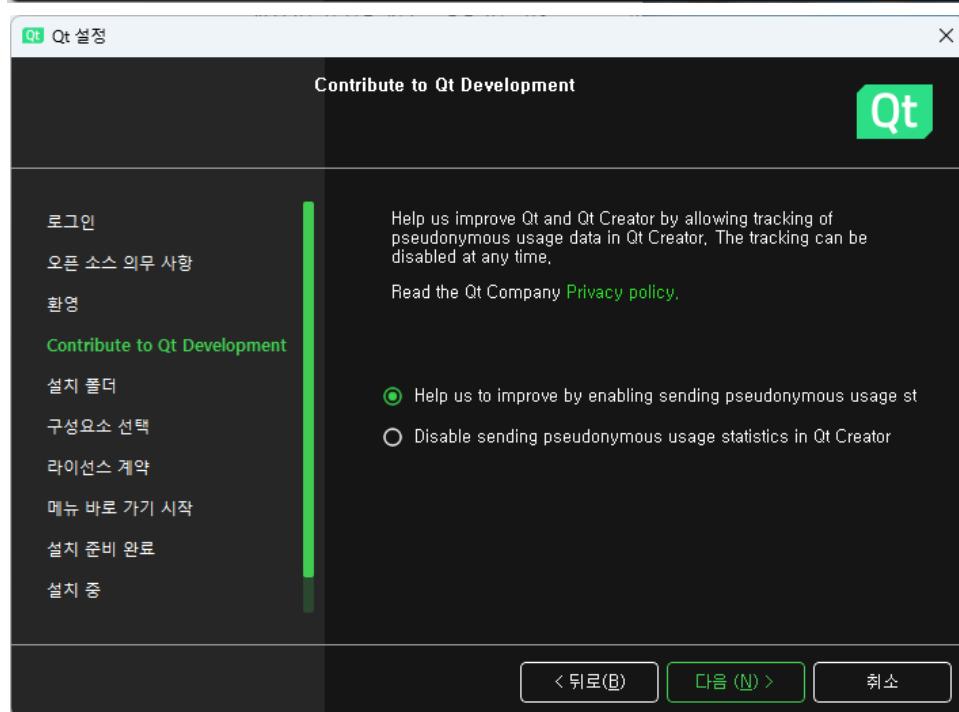
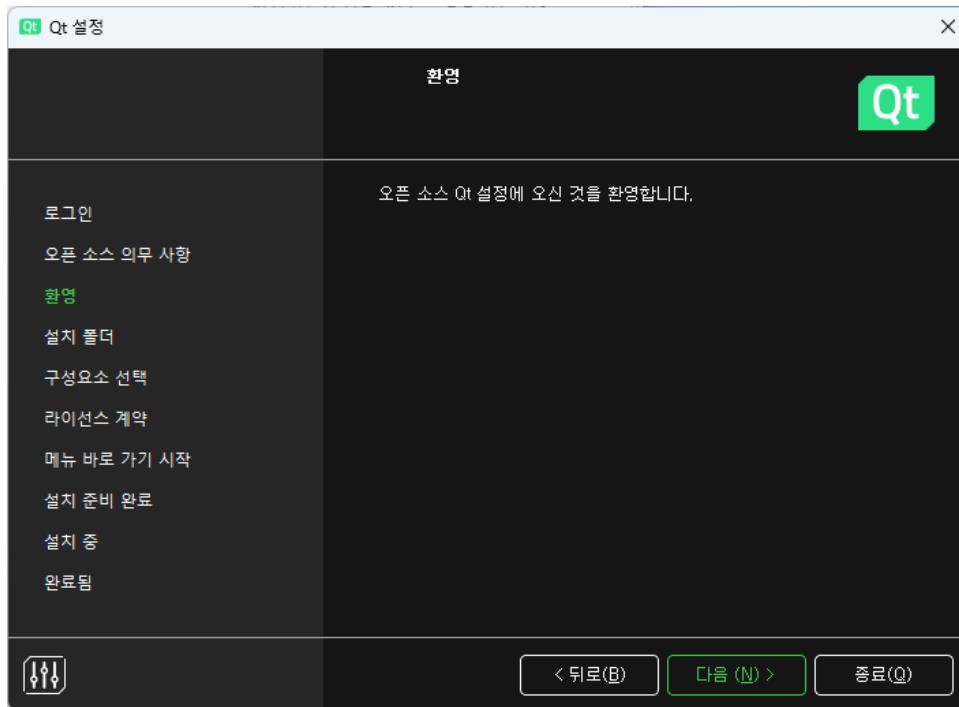
Download and run the exe file extension.



Enter your username and password as shown in the dialogue above. If you do not have a username, you can create an account by clicking [Sign up] on the dialogue screen above, and then enter your account information. Once you have completed entering your account information (username and password), click the [Next] button.

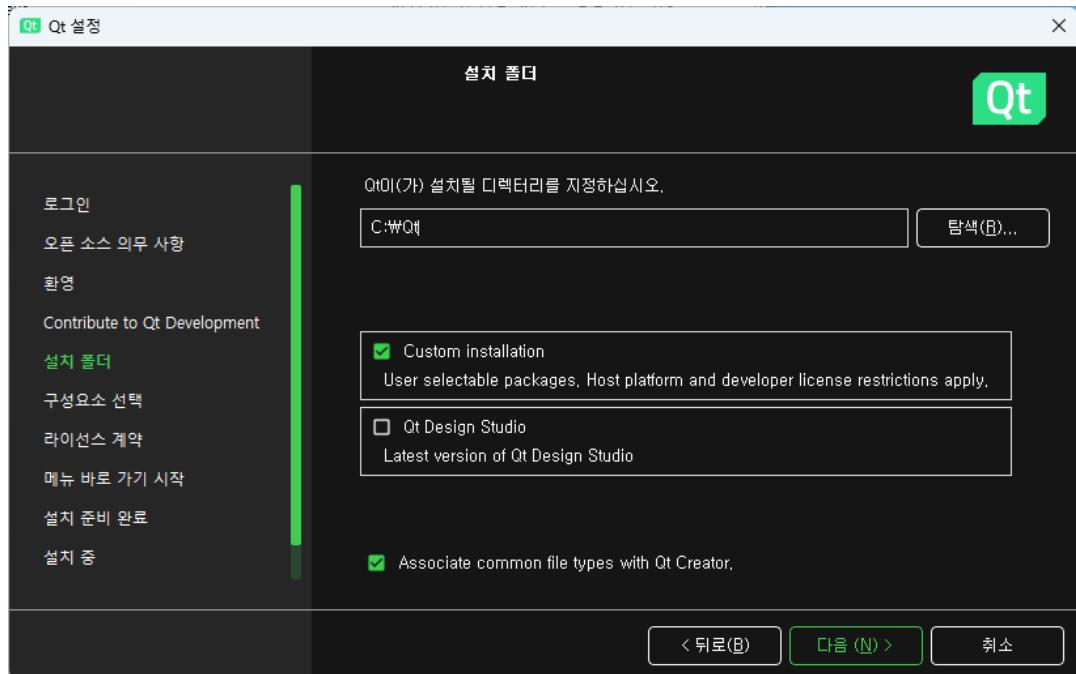


Jesus loves you.



The above figure shows a dialogue that asks you to agree or disagree to improve Qt by sending real-time information about errors that may occur when using the Qt Creator IDE. If you agree, select the first option as shown above, otherwise select the second option.

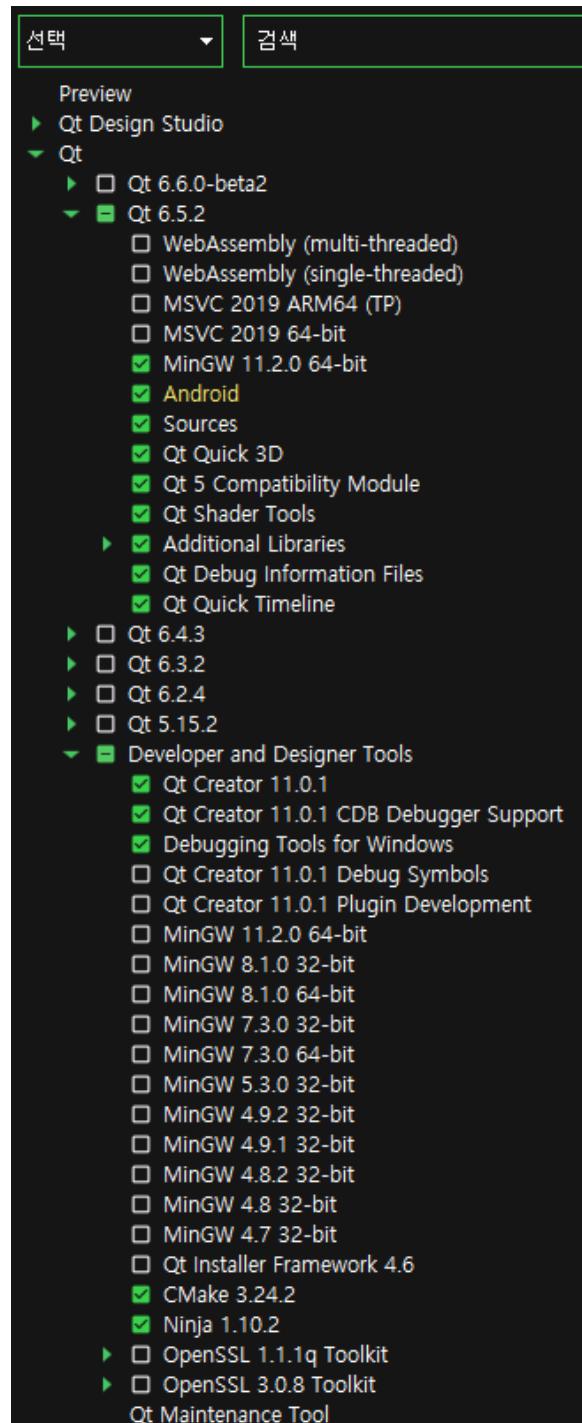
Jesus loves you.



If you select Custom Installation, you can choose your own installation components.



The selections you need to make from the above components are as shown in the figure below.



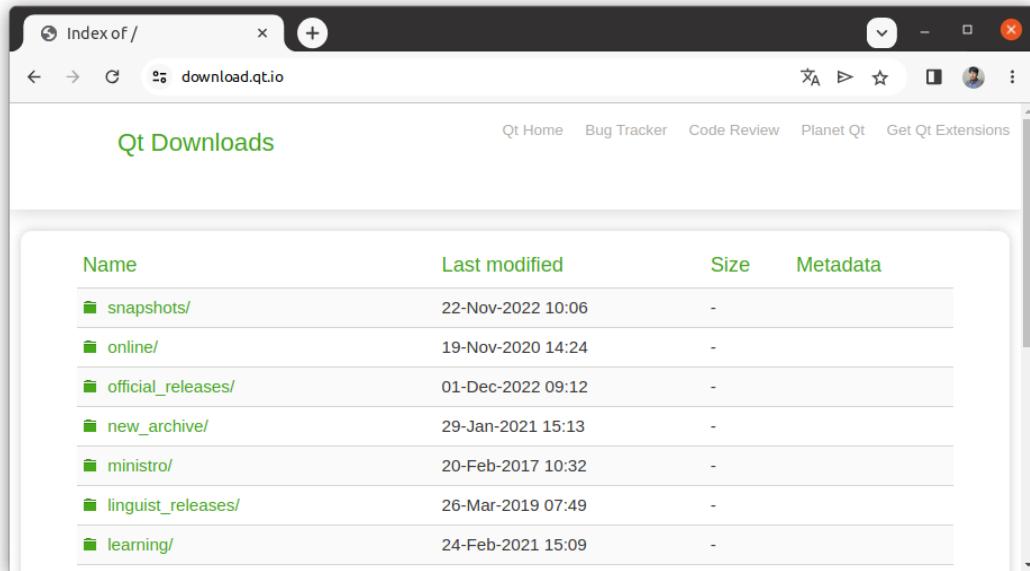
Install Qt version 6.5.2 from the versions. Qt is divided into LTS and non-LTS versions. LTS stands for Long Term Service and provides more continuous updates than non-LTS versions. However, if another higher version specifies LTS in the future, the lowest LTS version will be discontinued. Currently, Qt offers LTS versions of 6.5 and 5.15.

For this example, we will select 6.5.2. You can install other versions, but it is recommended that you install version 6.5.2 whenever possible. The table below describes the main items in each of the components.

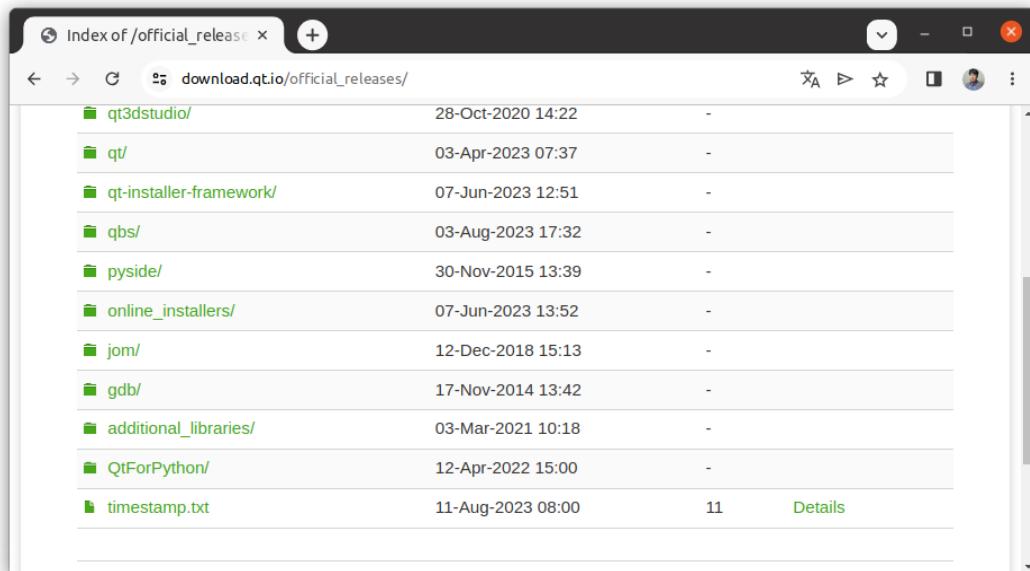
<Table> Major Component descriptions

카테고리	Component	Description
Qt 6.X.X	WebAssembly (multi-threaded)	Required to develop WebAssembly applications using Qt.
	WebAssembly (single-threaded)	Required to develop WebAssembly applications using Qt.
	MSVC 2019 ARM64(TP)	Microsoft Visual C++ 2019 is a 64-bit compiler for ARM. You need to install the MSVC 2019 ARM64 compiler separately to compile the source code you have written with this item.
	MSVC 2019 64-bit	Microsoft Visual C++ 2019 64Bit compiler. You need to install the MSVC 2019 64bit compiler separately when compiling the source code you have written.
	MinGW 11.2.0 64-bit	Open Source GCC/G++ 64 Bit Compiler
	Android	Compiler for use on the mobile Android platform.
	Source	Required for debugging the Qt API, for example to debug the Qt API source code using breakpoints at specific source code points.
	Qt Quick 3D	A module for working with 3D in QML using the Qt Quick 3D module.
	Qt 5 Compatibility Module	Provided to maintain Qt 5 compatibility with Qt 6.

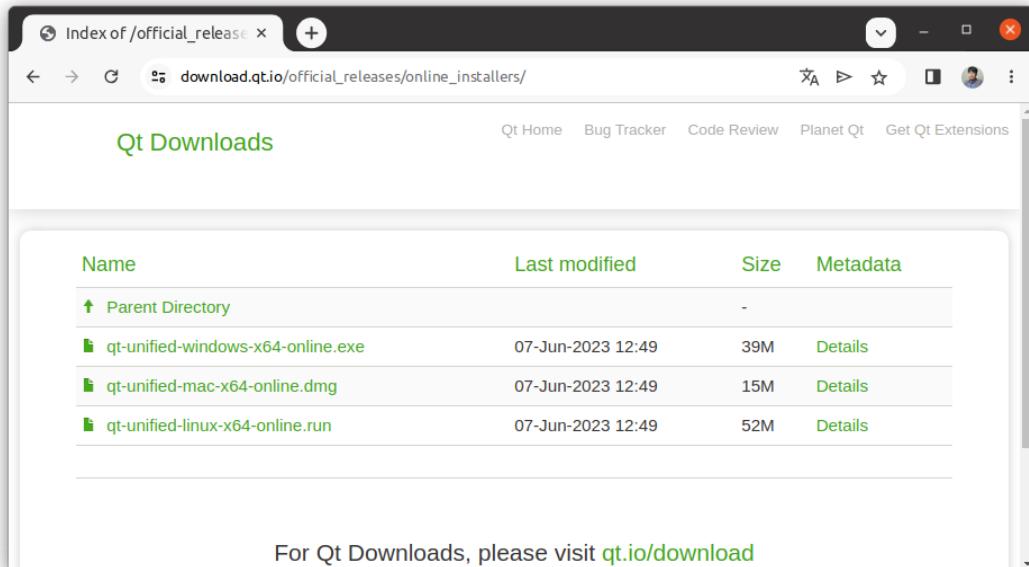
The Linux platform (operating system) has several distributions. For example, Ubuntu, Mint, and others, but we will use Ubuntu here. You can use any other distribution.



To download the Qt online installation, visit [download.qt.io](https://download.qt.io). When you see a page like the one in the image above, click [official\_release].



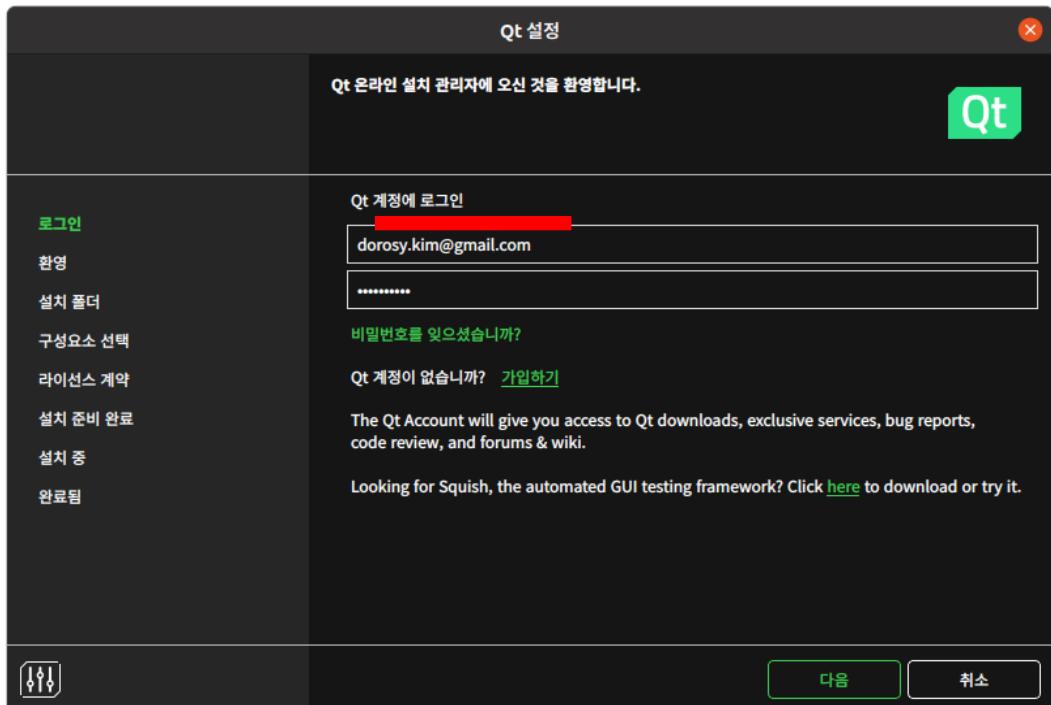
Click [online\_installers].



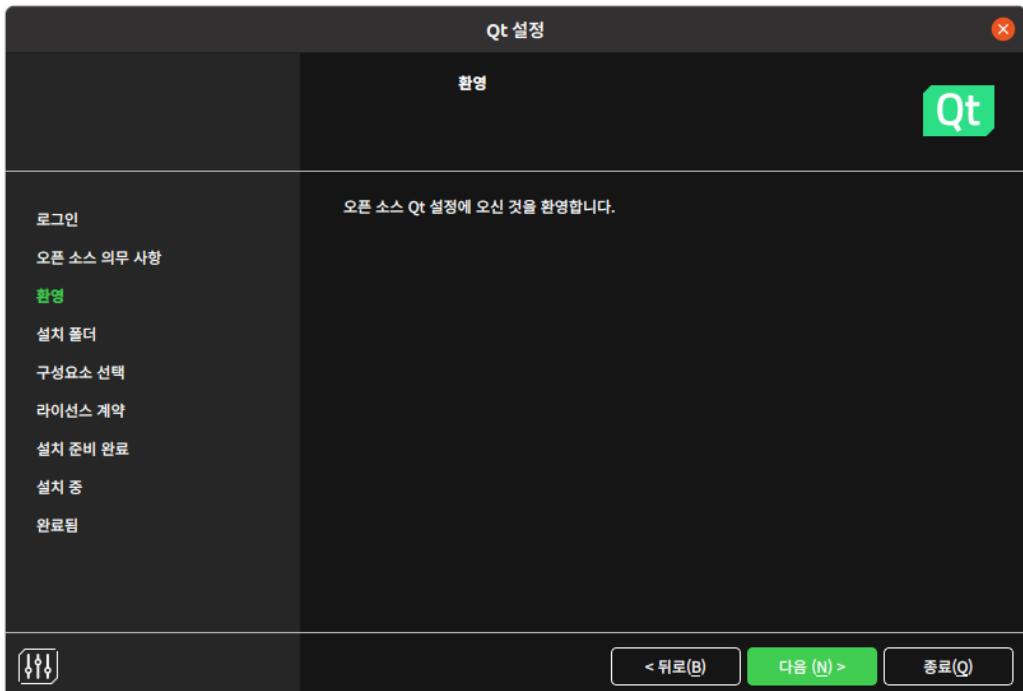
Download the file with the "run" extension.

```
dev@ubuntu:~/Downloads$ ls -tlr
합계 53744
-rw-rw-r-- 1 dev dev 55032849 8월 11 15:43 qt-unified-linux-x64-4.6.0-online.run
dev@ubuntu:~/Downloads$ chmod 755 qt-unified-linux-x64-4.6.0-online.run
dev@ubuntu:~/Downloads$ ./qt-unified-linux-x64-4.6.0-online.run
```

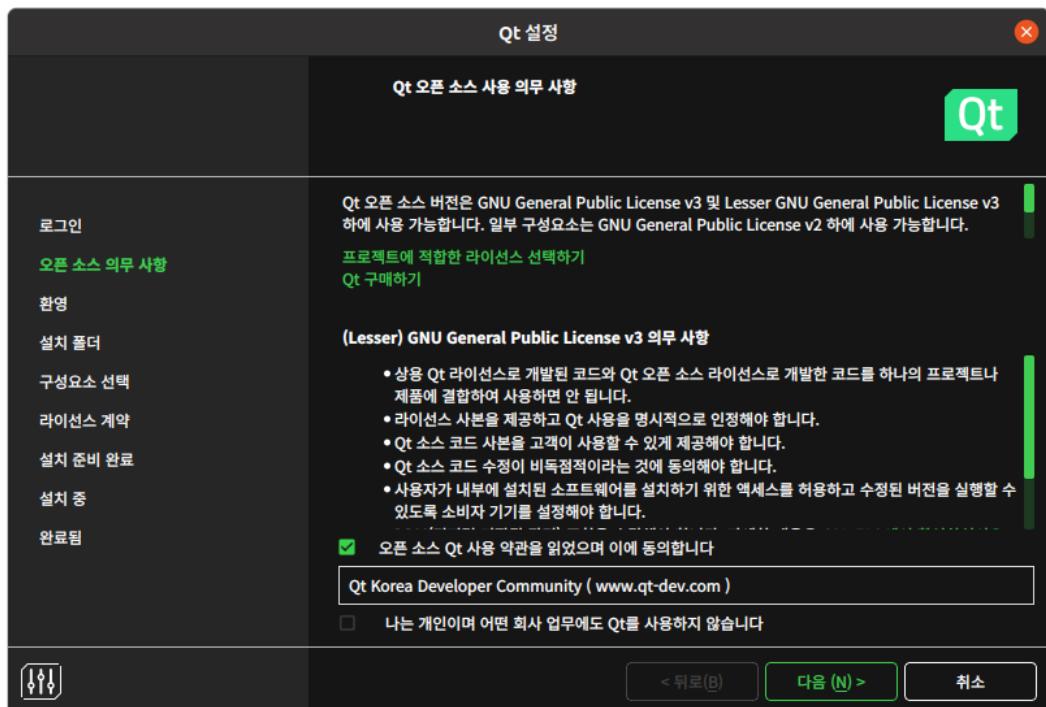
When the download is complete, navigate to the download directory. The download file does not have execute permissions, so give it execute permissions with chmod. Then run the Qt online installation file.



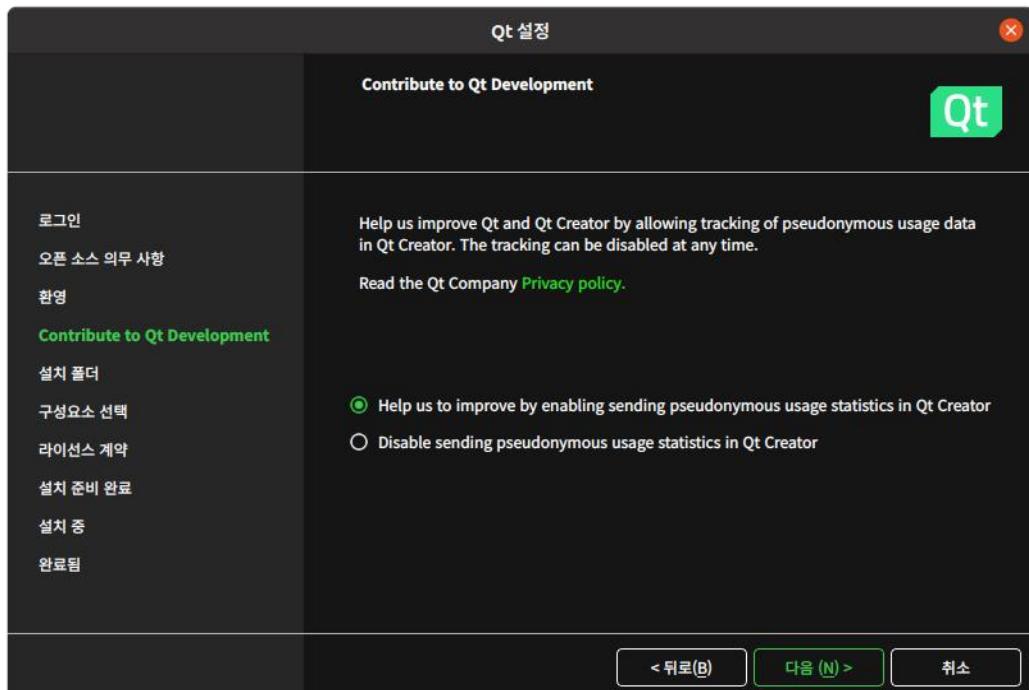
If you don't have a Qt account, click the Sign-Up button as shown in the screen above, and you will be taken to a web page where you can sign up. After signing up, return to the dialogue, enter your username and password, and click the [Next] button.



Jesus loves you.

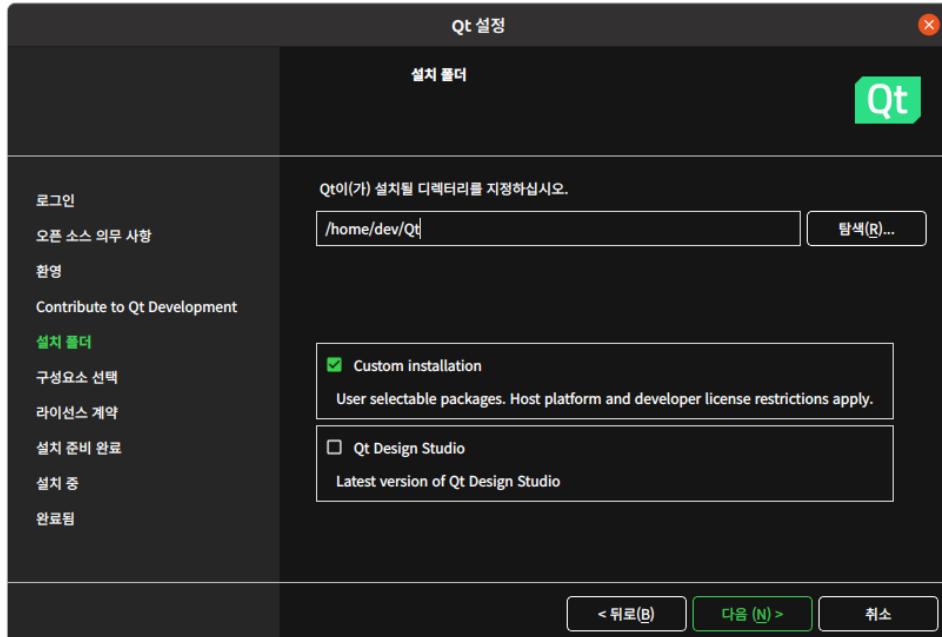


You can read about the open-source Qt Terms of Use. Enable the checkboxes as shown in the picture above and click the Next button.



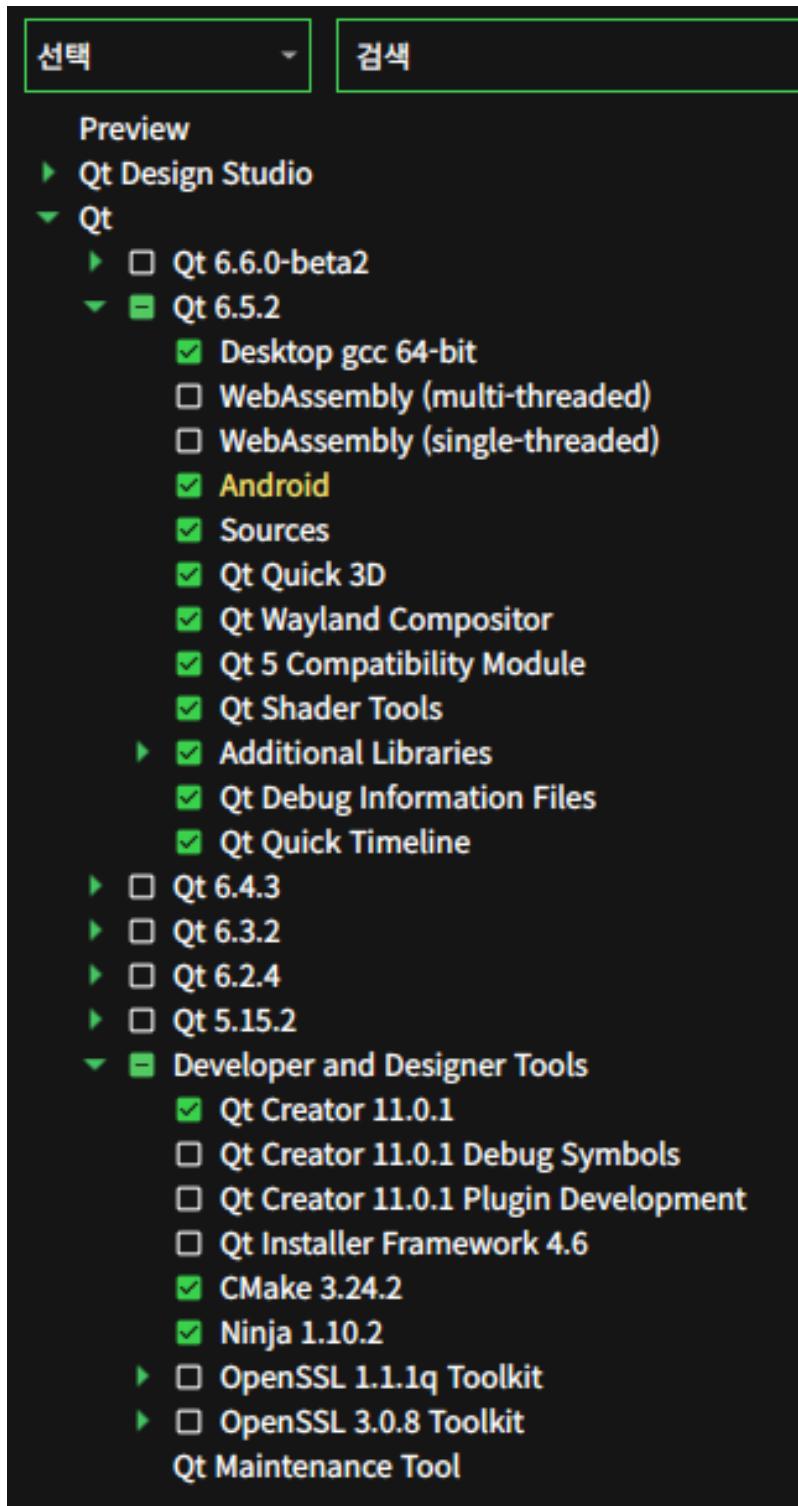
This dialogue asks you if you agree to allow information such as errors to be sent to the

Qt developers when using tools such as Qt Creator. Select the first option if you do, or the second if you do not.

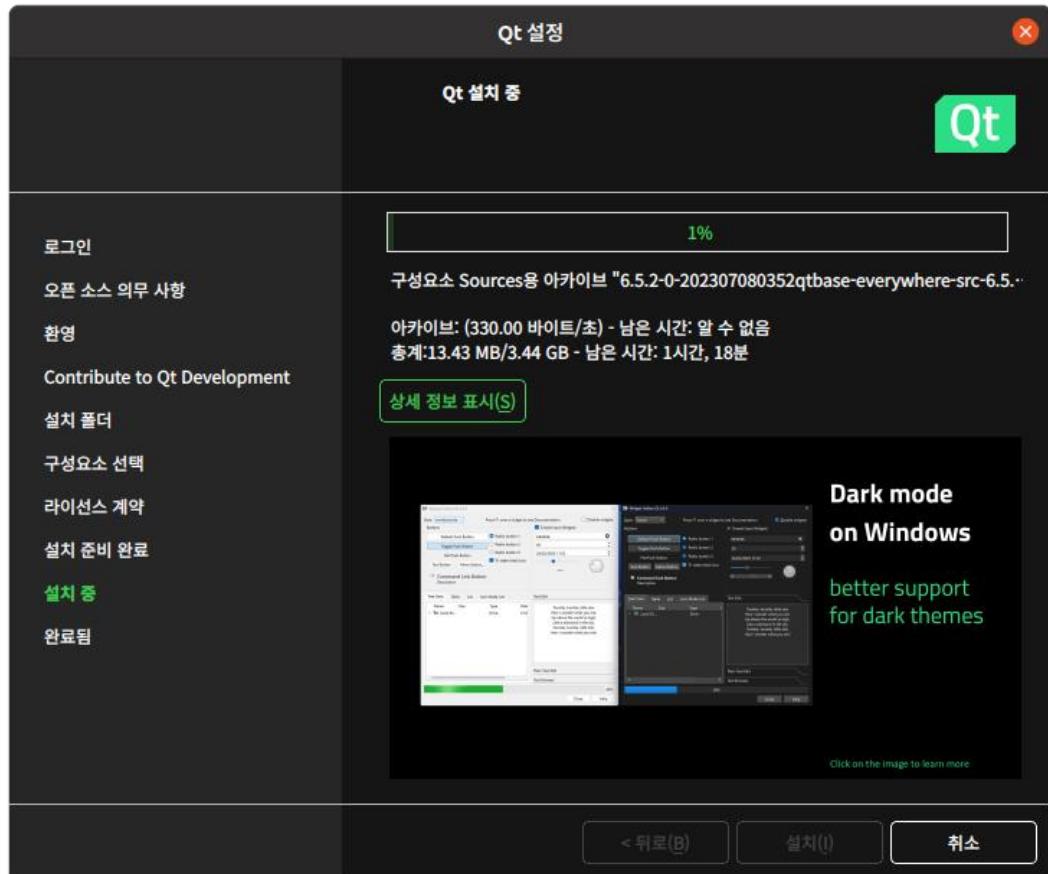


Select the [Custom Install] item. This item allows you to select the Component to install. In this case, select the [Custom Install] item as shown in the image above and click [Next].





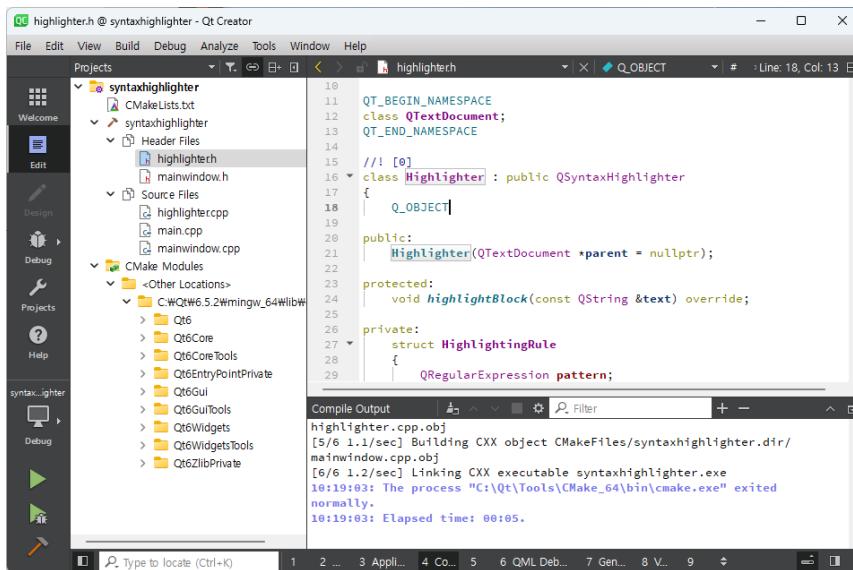
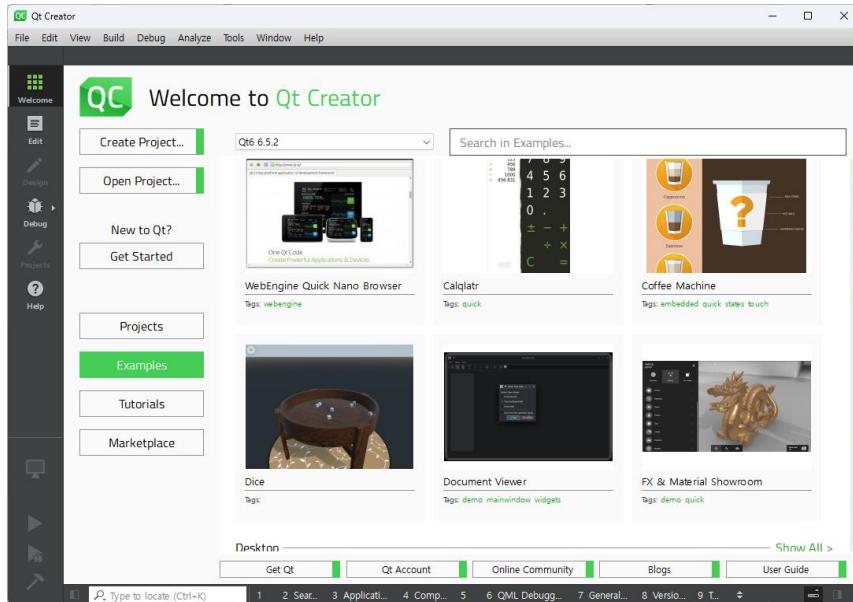
Select as shown in the figure above and click [Next]. For the description of the installation items, refer to the items previously described in MS Windows.



## 2. Useful development tools provided by Qt

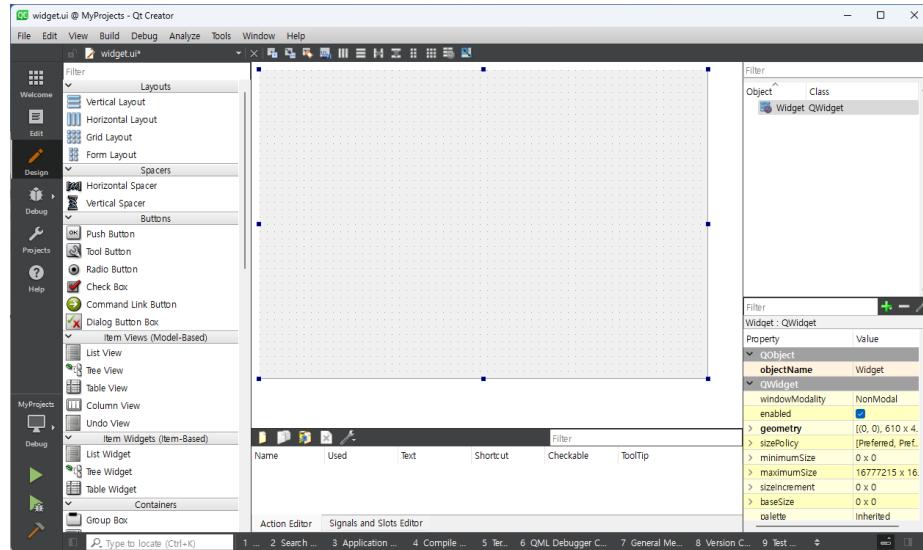
- Qt Creator IDE Tool

Qt provides Qt Creator IDE Tool as a tool for writing source code.



The Qt Creator IDE tool also provides source code authoring and debugging capabilities.

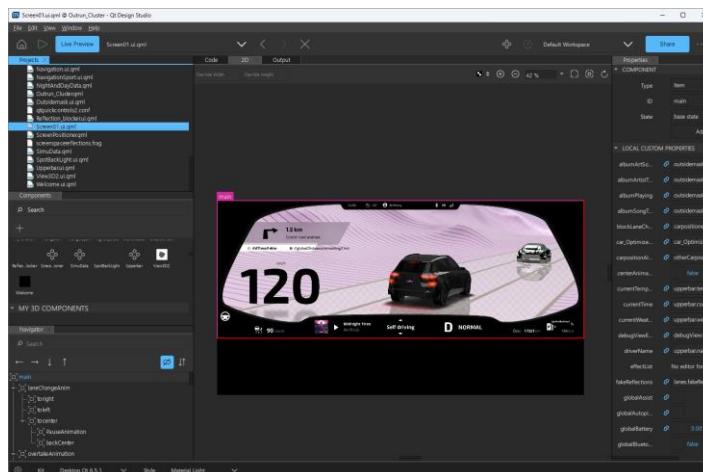
## ● Qt Designer Tool



The Qt Designer tool is provided to facilitate the implementation of GUIs. It allows the user to easily place the widgets of the desired GUI. This tool is integrated into Qt Creator. However, you can use Visual Studio instead of Qt Creator. Therefore, it can be run independently for Visual Studio users.

## ● Qt Design Studio

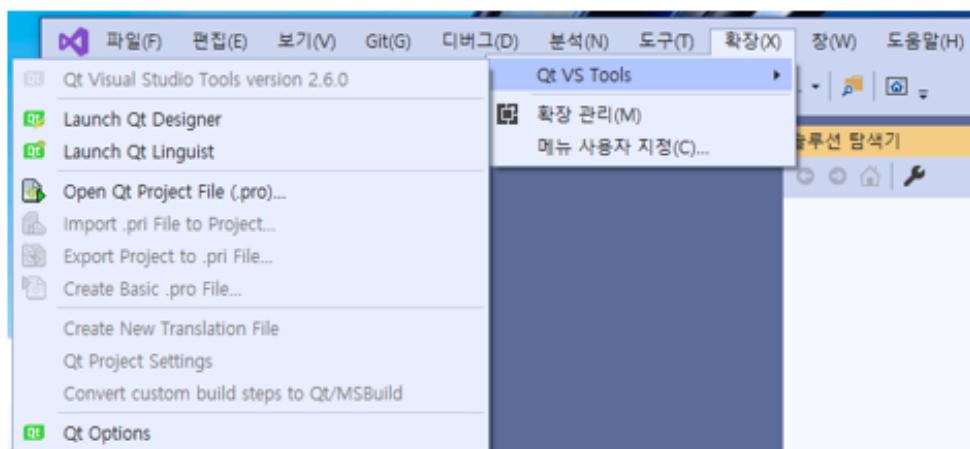
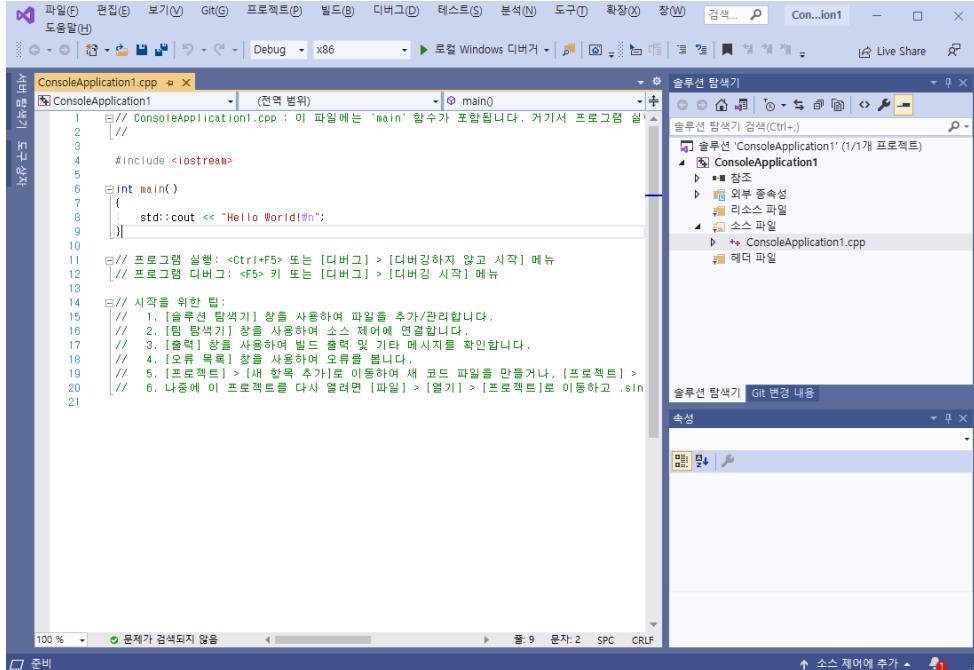
Qt Design Studio is a tool that makes it easy to write QML. It provides the same functionality as Qt Designer, but the difference is that Qt Designer is a tool for creating C++-based GUIs, while Qt Design Studio is a tool for creating QML.



## ● Developing Qt in MS Visual Studio

In addition to Qt Creator, you can use MS Visual Studio to develop applications. In Qt, you can install the add-in. You can download the add-in from the URL below.

- ✓ Download URL: [https://download.qt.io/official\\_releases/vsaddin/](https://download.qt.io/official_releases/vsaddin/)

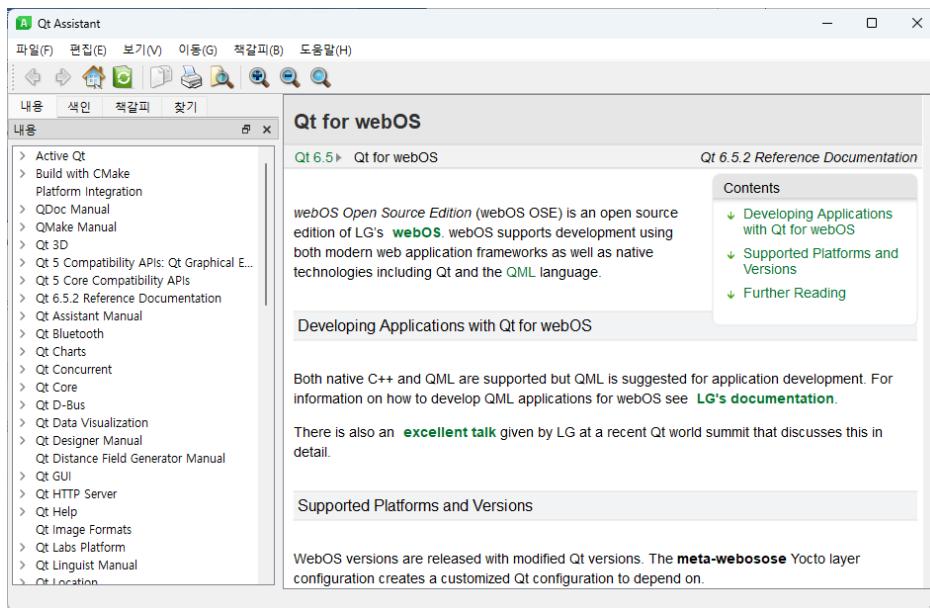


## ● Qt Assistant Tool

This is a tool to provide help. As you develop applications with Qt, you may find yourself

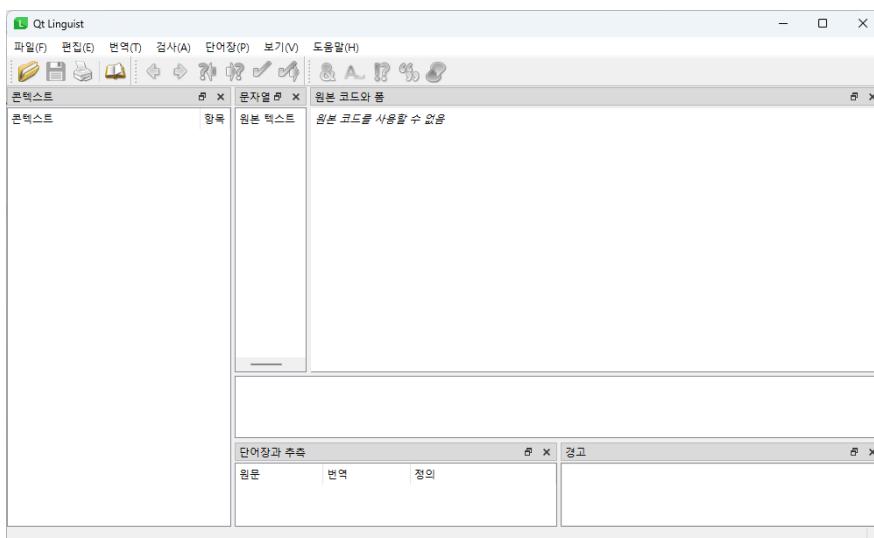
Jesus loves you.

searching for classes or QML that you don't recognize.



## ● Qt Linguist Tool

This tool is intended for multilingual support and can display different GUI languages depending on the language used in each country in your application.

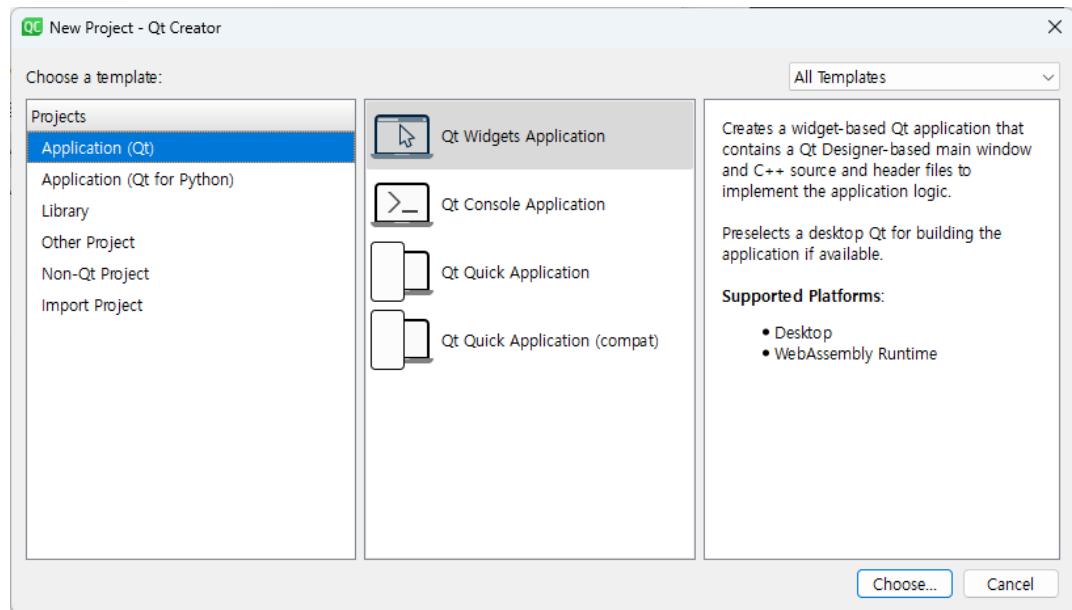


### 3. Building a project with CMake

CMake is a tool provided to make it easier to build projects. It provides the ability to easily build the implemented source code and create an executable. CMake is a build tool, and many development frameworks, including Visual Studio as well as Qt, use CMake as a build tool.

Qt offers a choice between CMake and qmake. Qt still uses qmake a lot, but it is increasingly using CMake.

Qt Creator includes a feature that automatically creates project files when you create a project. You can choose between CMake, qmake, and qbs when creating a project.



As shown in the figure above, you can choose between CMake, qmake and qbs when creating an application-based project for Qt Widget Application and Qt Console Application.

However, since version 10.x.x of the Qt Creator IDE tool, only CMake is supported to automatically create project files when you select Qt Quick Application as a project.

This does not mean that you can only use CMake. When you create a project, you can still create a project file manually, just not automatically. So, starting with Qt Creator IDE

Jesus loves you.

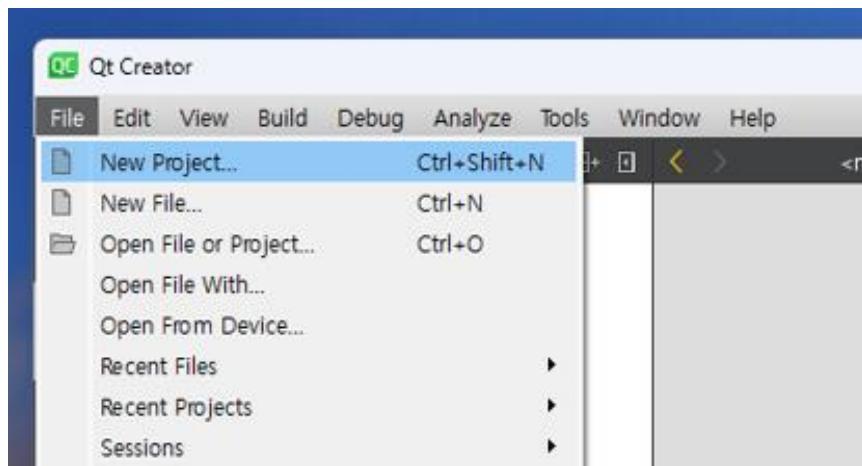
10.x.x, you can still use qmake, just not automatically.

In this chapter, you will learn how to use CMake to build a console application and how to build a GUI application.

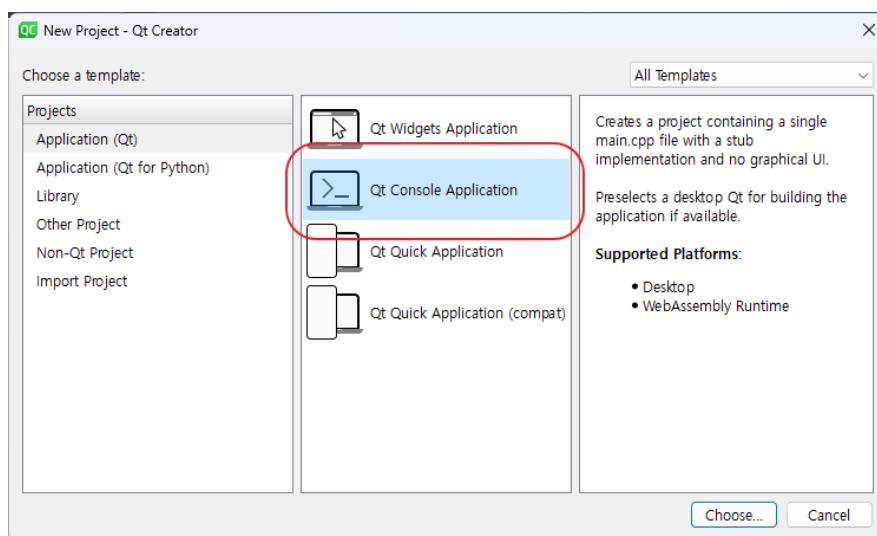
### 3.1. Implementing Console Applications with CMake

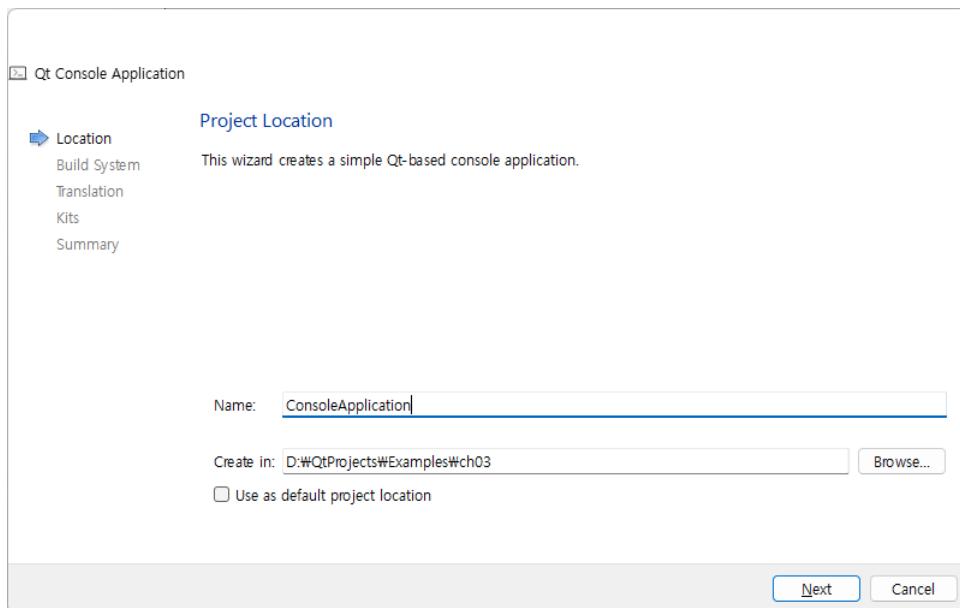
In this chapter, we will implement a console application using CMake. A Console application is an application that does not have a GUI. For example, similar to Server Side or Backend.

Create a project as shown in the figure below. To create a project, select [File] -> [New Project] from the Qt Creator menu. The shortcut is [Ctrl + Shift + N].



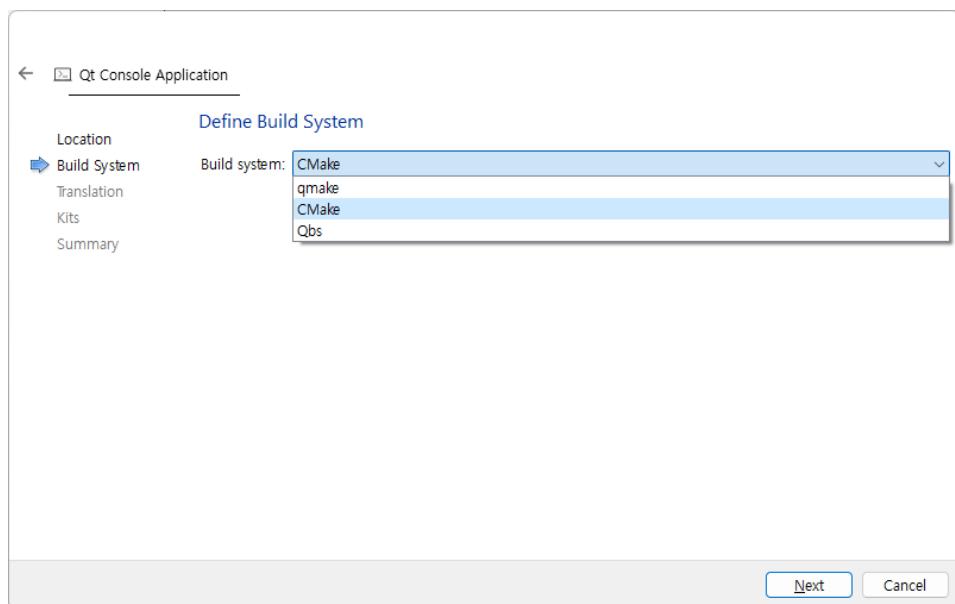
Then select [Qt Console Application] from the dialog window, as shown in the image below.



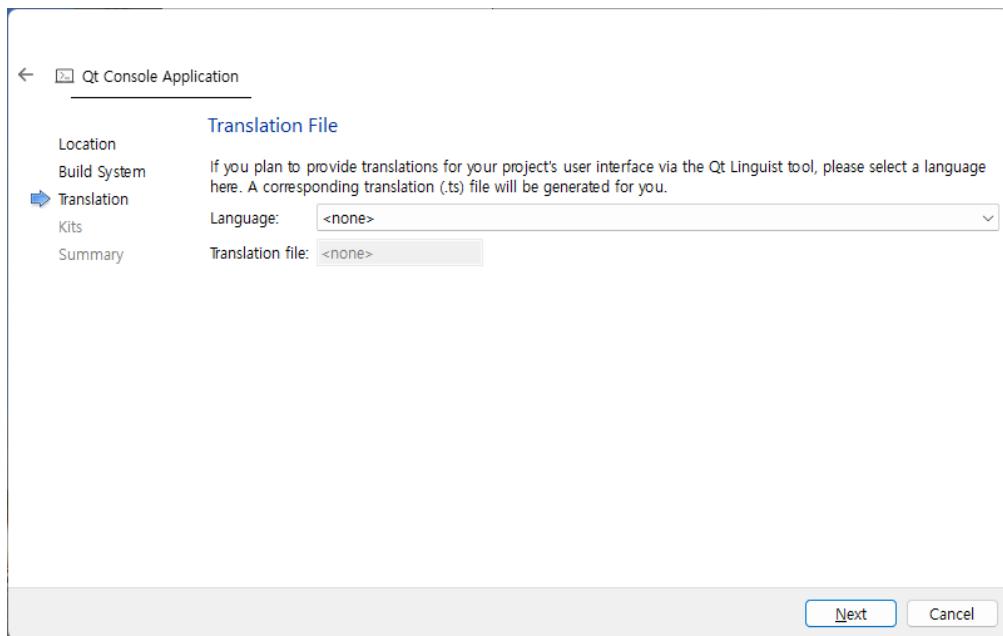


In the Name field, enter a name for the project. The Create In field is where the project will be created.

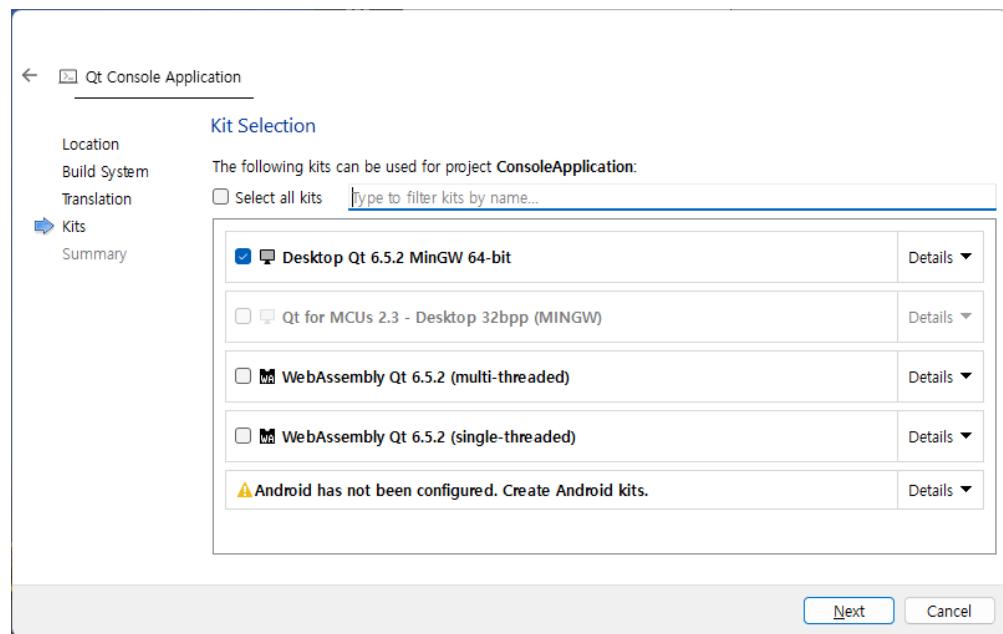
As shown in the image above, specify the project name and the directory where the project will be located, then click the [Next] button.



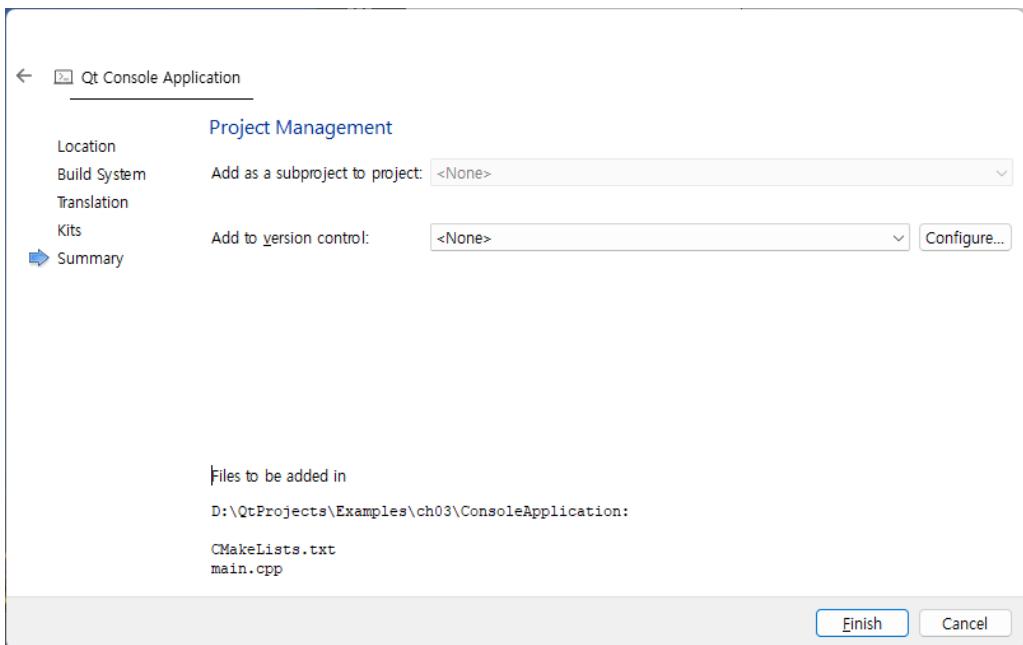
The following is a dialog to select the project build tool. As shown in the image above, select CMake and click the [Next] button.



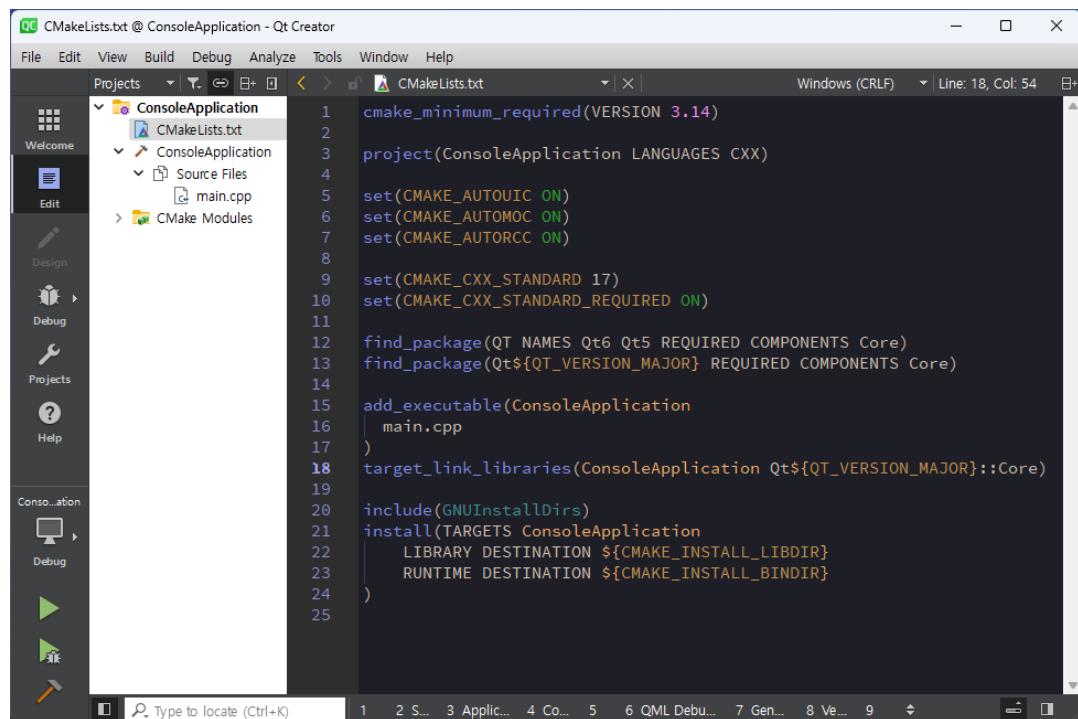
This is the dialog to select the language for multilingual language support. In this case, click the [Next] button without making any changes, as shown in the figure above.



The next dialog is for selecting a compiler. As shown in the figure above, select the MinGW compiler and then click the [Next] button.



The following is a dialog to select a source code version control system. Leave it as default and click the [Finish] button.



When the project creation is complete, as shown in the image above, double-click the CMakeList.txt file in the project navigation pane on the left.

CMakeList.txt is the project file for building this project. Let's take a closer look at CMakeList.txt.

```
cmake_minimum_required(VERSION 3.14)

project(ConsoleApplication LANGUAGES CXX)

set(CMAKE_AUIC ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Qt NAMES Qt6 Qt5 REQUIRED COMPONENTS Core)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Core)

add_executable(ConsoleApplication
    main.cpp
)
target_link_libraries(ConsoleApplication Qt${QT_VERSION_MAJOR}::Core)

include(GNUInstallDirs)
install(TARGETS ConsoleApplication
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

The above is the contents of the CMakeList.txt file. The first line is the version of CMake. Here we specify CMake. Here we require at least version 3.14.

```
project(ConsoleApplication LANGUAGES CXX)
```

Specify the name of the project and that the language used is C++.

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

The first line tells it to use C++ version 17 or later. The second line tells it to print an error if the compilation is too old for the C++ version.

```
find_package(Qt NAMES Qt6 Qt5 REQUIRED COMPONENTS Core)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Core)
```

```
target_link_libraries(ConsoleApplication Qt${QT_VERSION_MAJOR}::Core)
```

The source code above specifies which modules Qt will use. Qt is organized into many different modules. For example, to use the Network module (or API) in your project, you would use the following.

```
find_package(Qt6 REQUIRED COMPONENTS Network)
target_link_libraries(mytarget PRIVATE Qt6::Network)
```

And the add\_executable( ) function can be used to specify the source code files to add to the project, as shown below.

```
add_executable(ConsoleApplication
    main.cpp
)
```

- main.cpp source code and build and run it

Next, select the main.cpp source code in the left project pane of the Qt Creator and add it as shown below.

```
#include <QCoreApplication>
#include <QDebug>

int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    qDebug("Hello world.");

    return a.exec();
}
```

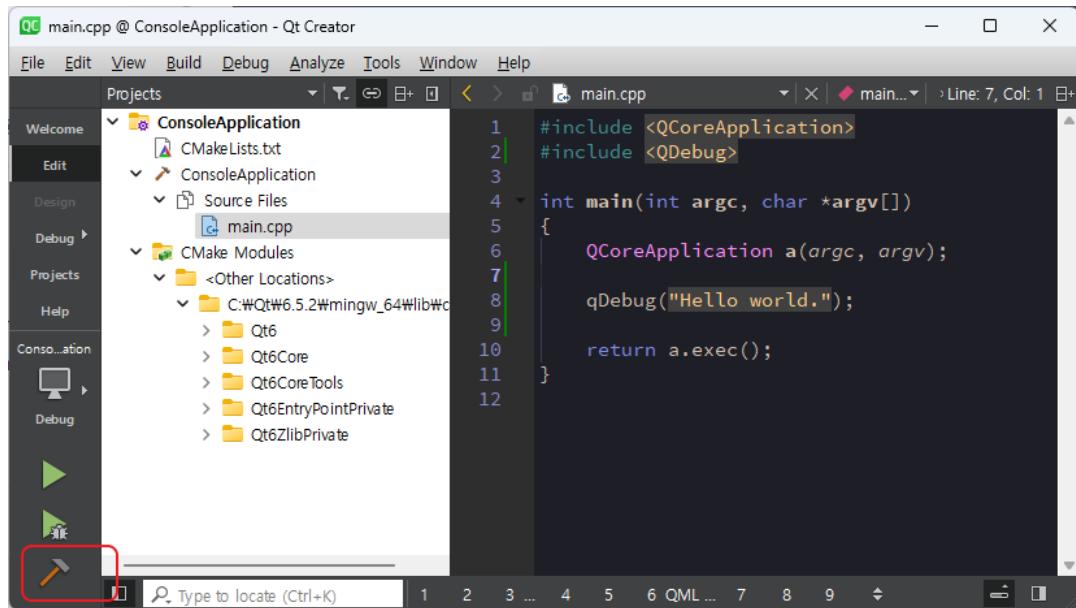
First, the QCoreApplication a(argc, argv) must pass argc, argv, which are the arguments to the main( ) function. Since the application we are running is a Console, it is a QCoreApplication, but if you have a GUI-based application, you should use a QGuiApplication.

qDebug( ) prints a message. So it prints Hello world. in the Console window. The last line serves to prevent the program from terminating.

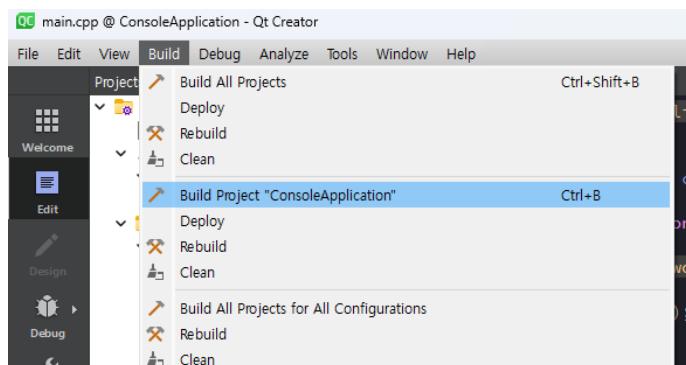
Once you've finished writing the source code as shown above, let's build and run the project. To build it, you can press the shortcut Ctrl + B. Alternatively, you can click the

Jesus loves you.

hammer icon in the Qt Creator.



Alternatively, you can click [Build] -> [Build Project "Project Name"] in the menu.



Once the build is complete, you can run it using the shortcut Ctrl + R or the arrow icon above the build icon in Qt Creator. Alternatively, you can select [Build] -> [Run] from the menu. Once you run it, you should see it run as shown in the image below.

To see the message in the Console window, click [3 Application] at the bottom of the Qt Creator to see the message printed in the Console window, as shown in the image below.

The screenshot shows the Qt Creator interface with the following details:

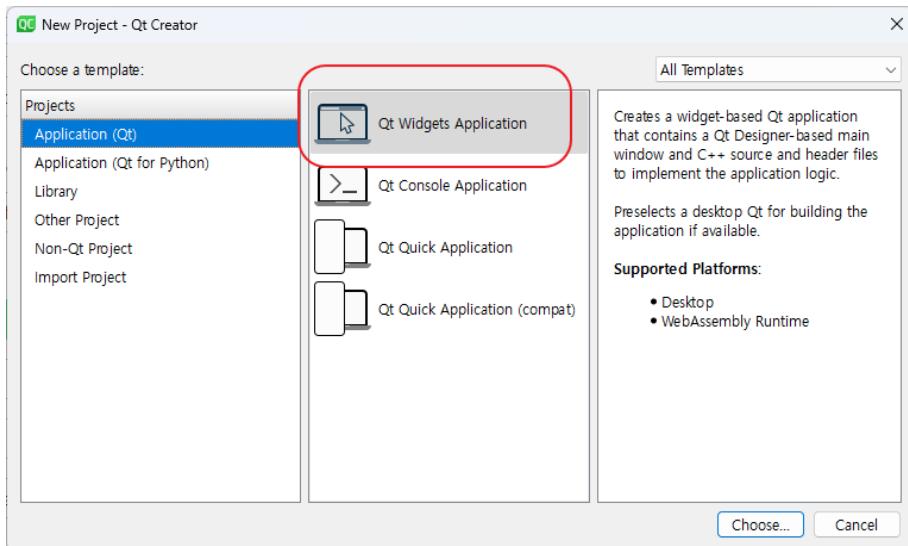
- Project Tree:** Shows a project named "Console Application". It contains "Source Files" with "main.cpp" selected. Other files listed include "MakeLists.txt", "onsoleApplication", and "Qt6Core".
- Code Editor:** Displays the content of main.cpp:

```
#include <QCoreApplication>
#include <QDebug>

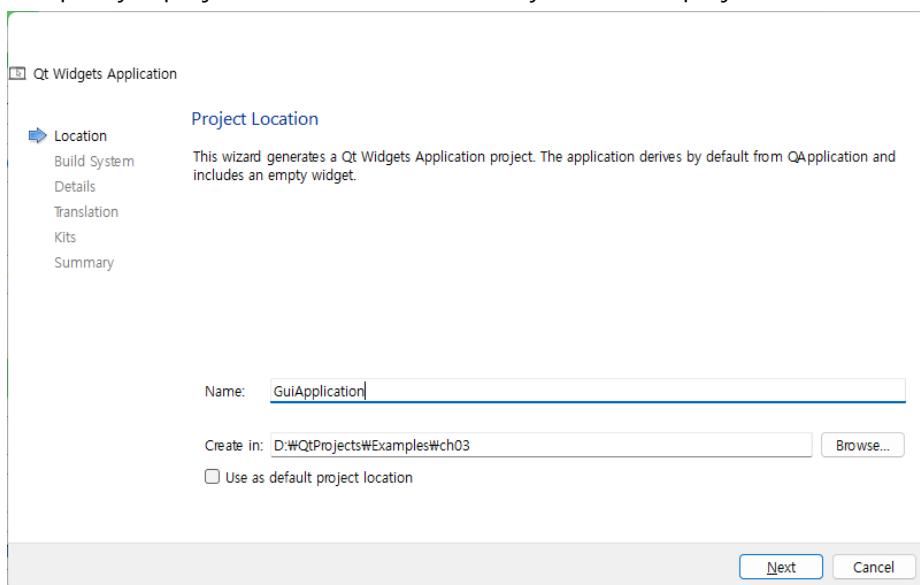
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);
    qDebug("Hello world.");
    return a.exec();
}
```
- Output Panel:** Shows the output of the application run. The text "Hello world." is highlighted with a red box and circled with a red arrow pointing from the bottom of the panel up towards the output text.
- Bottom Bar:** Contains tabs for "Search..." (circled in red), "3 Application...", "4 Compilation...", "5 ...", and "6 QML Debugger...".

## 3.2. Implementing GUI Application with CMake

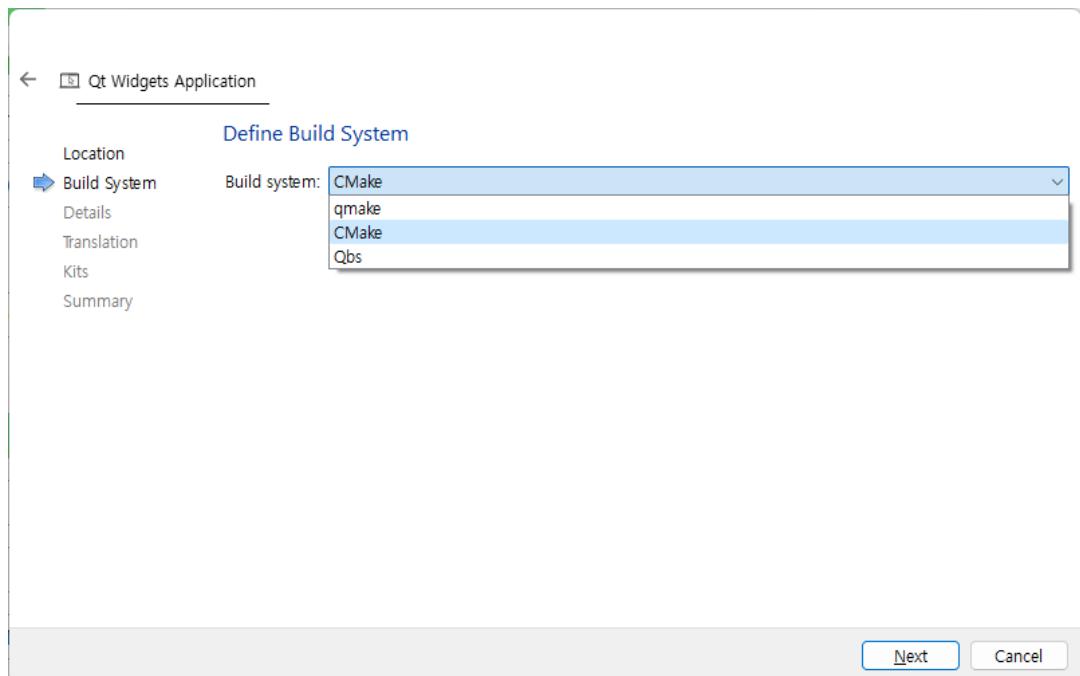
This time, let's create a GUI-based application when creating a project. In the project creation dialog, select the Qt Widget Application item as shown in the image below and click the [Choose] button below it.



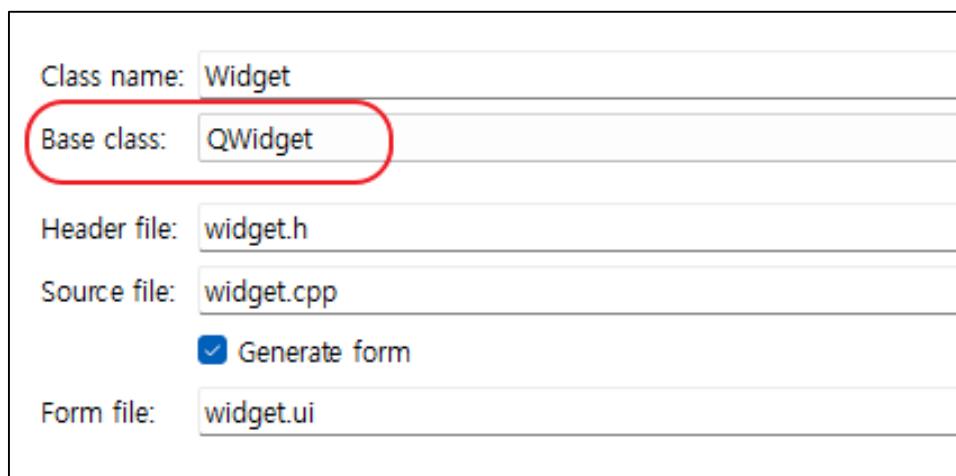
Next, specify a project name and the directory where the project will be located.



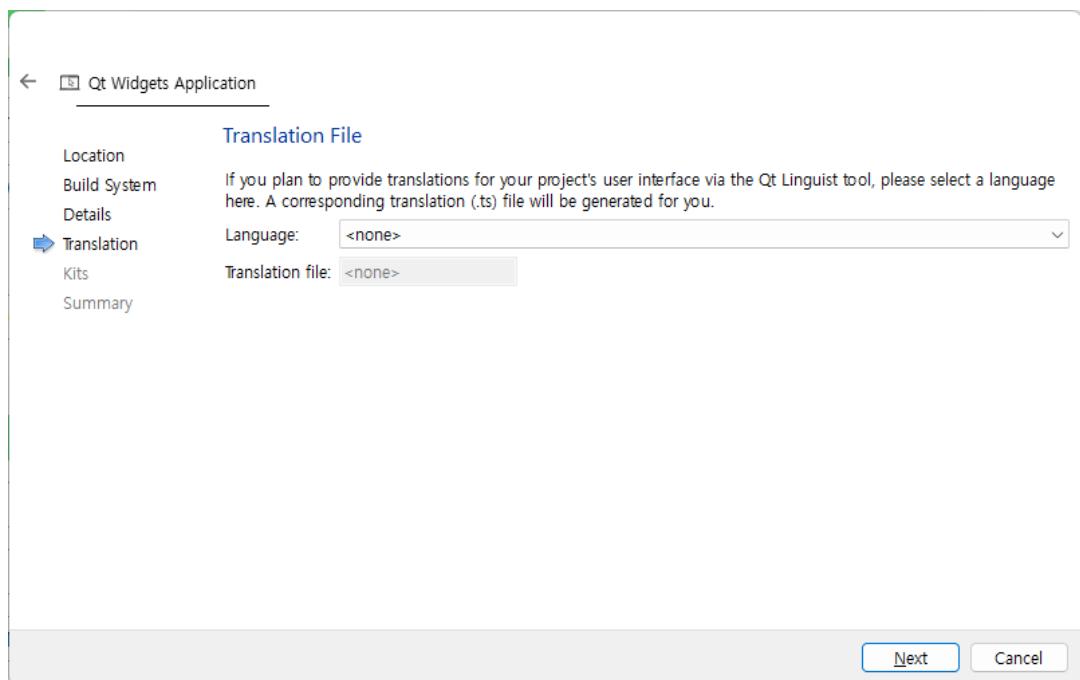
Next, select CMake as your project build tool.



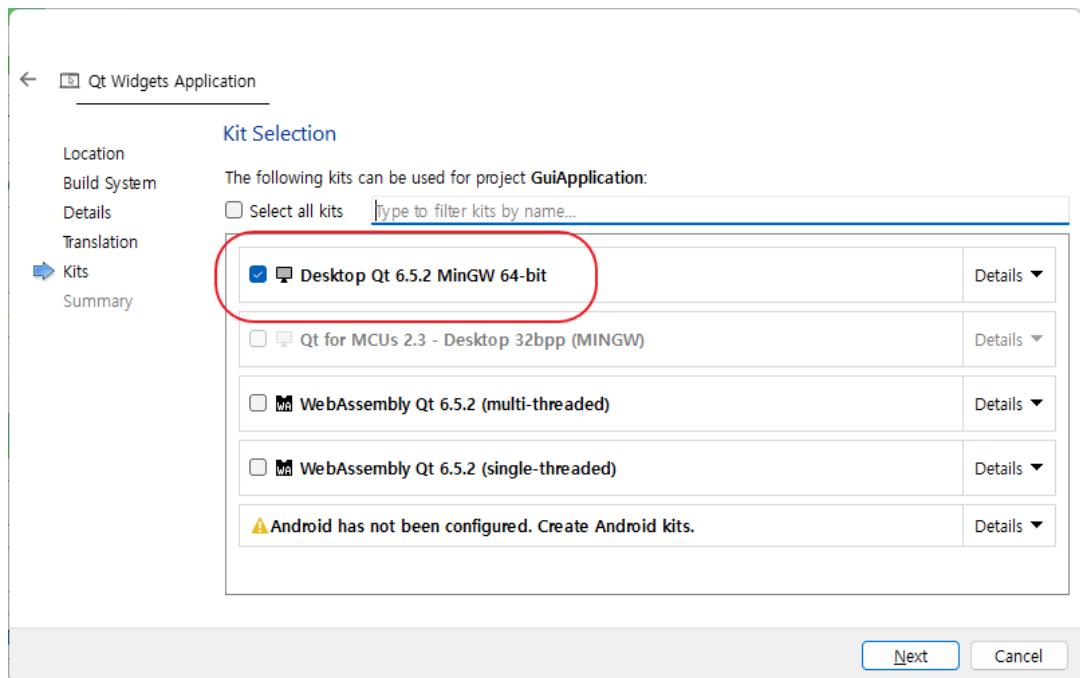
Next, select QWidget as the Base Class, as shown in the image below, and click the [Next] button.



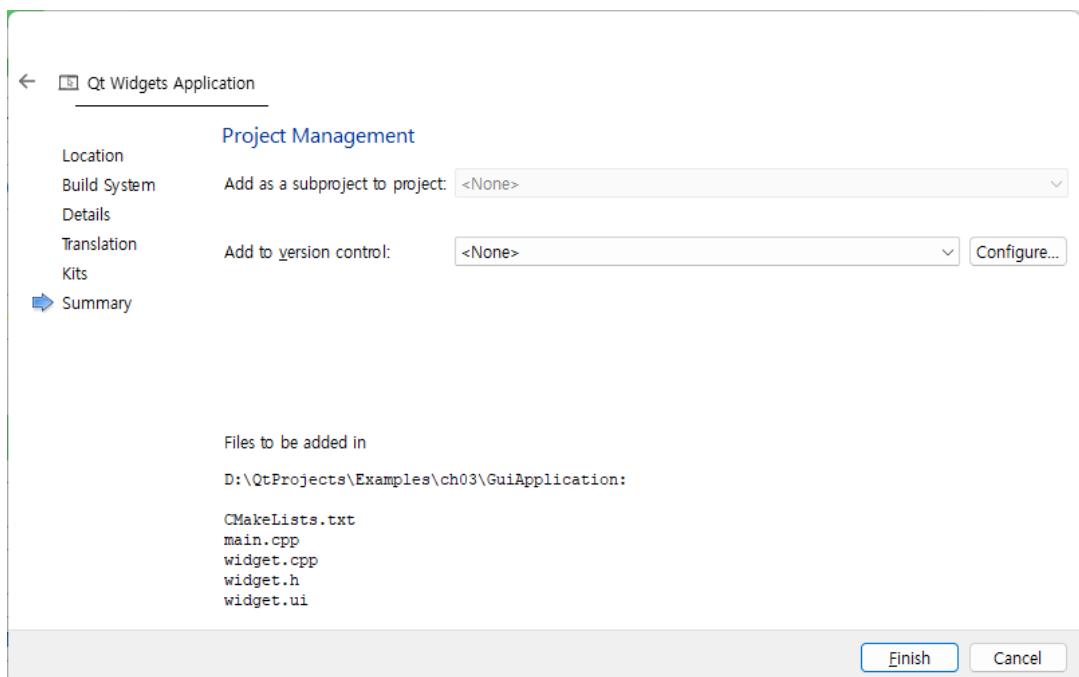
Next, there is a dialog to select the language that supports multilingualism. Leave the default here and click the [Next] button.



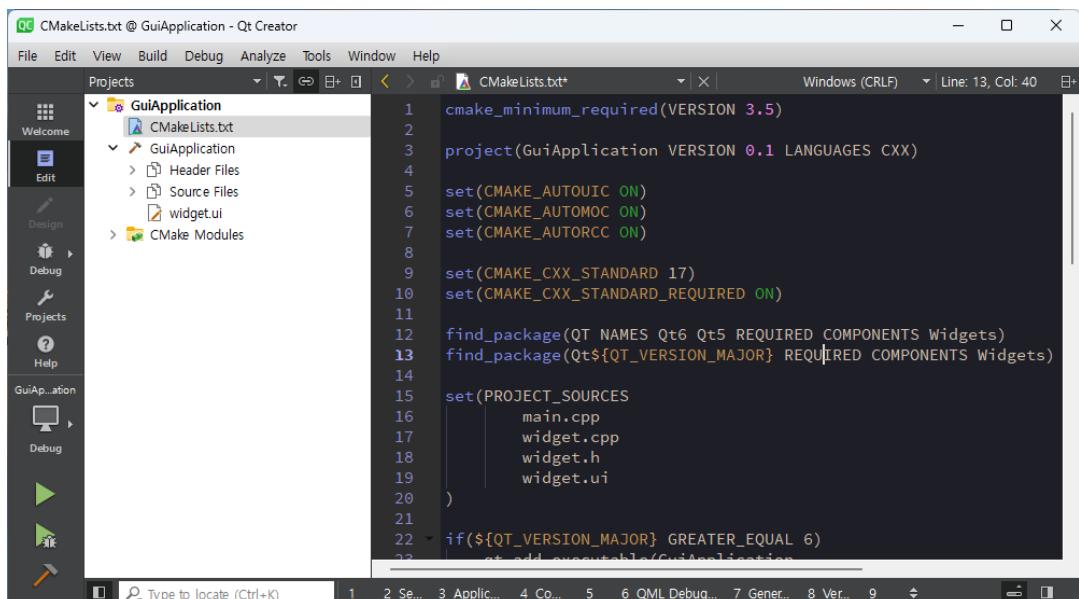
The next dialog is to select a compiler. In the compiler selection screen, select the MinGW compiler and click the [Next] button.



Jesus loves you.



Click the [Finish] button below with <None> set as shown in the image above. After clicking the [Finish] button, the project is created and you can click the CmakeList.txt file in the left navigation pane of Qt Creator.



The CMakeList.txt file looks a bit more complicated than the previous Console. Let's take a look at the commands line by line.

```
project(GuiApplication VERSION 0.1 LANGUAGES CXX)
```

There is no version information in the Console, but you can specify it yourself, as shown above.

```
set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)
```

CMAKE\_AUTOUIC is a command to automatically generate Qt's uic (Qt User Interface Compiler) files.

For example, without it, Qt will automatically generate files for you, but it will not automatically generate the source code for UI files in XML format with a .ui extension.

In this example, we used a uic called widget.ui, so when we build it, the ui\_widget.h file is automatically generated, but without CMAKE\_AUTOUIC, the ui\_widget.h file is not automatically generated.

CMAKE\_AUTOMOC automatically generates the moc (Meta Object Compiler) file. And CMAKE\_AUTORCC automatically generates the resource file.

```
if(${QT_VERSION_MAJOR} GREATER_EQUAL 6)
    qt_add_executable(GuiApplication
        MANUAL_FINALIZATION
        ${PROJECT_SOURCES}
    )
else()
    if(ANDROID)
        add_library(GuiApplication SHARED
            ${PROJECT_SOURCES}
    )
    else()
        add_executable(GuiApplication
            ${PROJECT_SOURCES}
    )
endif()
endif()
```

In CMake, you can use if statements. The above source code tells us to execute the content in parentheses if Qt version is 6 or higher, otherwise, if the platform is Android, execute the content in parentheses. Otherwise, it means to execute the content in the parentheses of else( ).

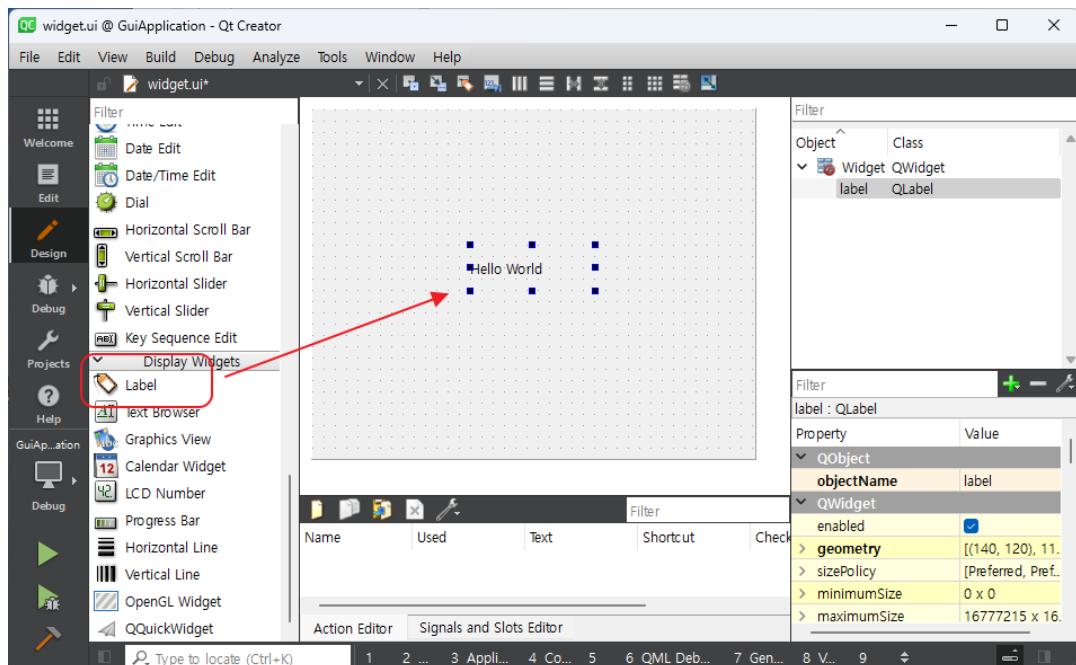
add\_library( ) is used to create a library.

For example, when creating a project, you can define whether it should be an executable application or a library. Here, `add_library( )` is used to create a library project. And the `add_executable( )` function is used to create an executable application.

- Placing GUI Widgets with UI Files

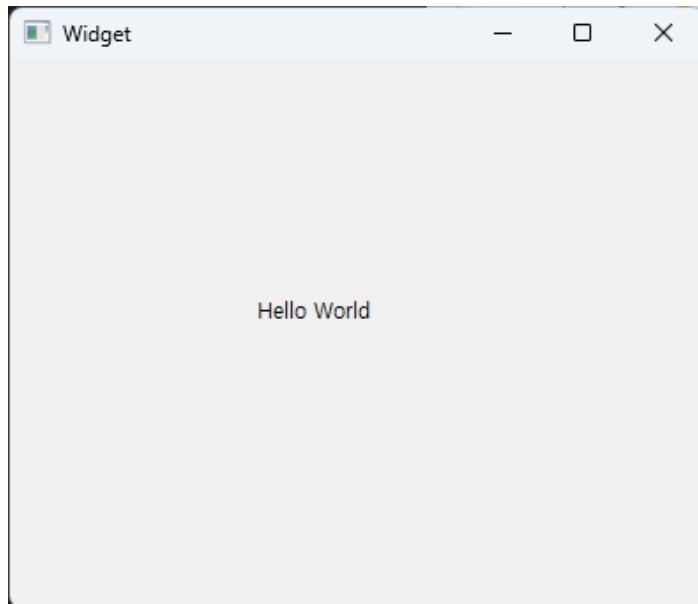
In the project navigation pane, you should see that a file with the `.ui` extension has been created. This file is where you can place the GUI Widgets feature that allows you to easily place the widgets you want with the mouse. As an example, let's place the `QLabel` widget and run it after building.

Double-click the `widget.ui` file and you should see something like the screen shown in the image below. In the screen below, let's place a `QLabel`, change the content of the `widget` to `Hello World`, and run it.



To return to the source code, press `<Esc>`, which will close the Designer where you can place GUI widgets. Alternatively, you can return to the source code window by pressing the [Edit] icon on the left side of the Qt Creator.

Once you've configured the widget as shown above, let's run the build. When you run it, you can see it running as shown in the image below.



So far, we've used UI Designer to create our GUI. You can also use files with the ui extension, but you can also write GUI code by writing your own source code without using UI files.

- Widget Class

Create and run an instance of the Widget class in main.cpp.

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

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    Widget w;
    w.show();
    return a.exec();
}
```

If you look at the header file for the Widget class, you'll see that the private keyword has an instance of ui.

```
#ifndef WIDGET_H
#define WIDGET_H
#include <QWidget>
```

```
QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
};

#endif // WIDGET_H
```

ui\_widget.h is a file that uic automatically converts to source code based on files with a .ui extension, creating the ui\_widget.h file.

We've used a simple QLabel here, but you can create a more complex GUI and handle its events later.

For example, you could place a button on the GUI and create a function in the Widget class that handles the button.

## 4. Building a project with qmake

Qt provides qmake, a tool for building projects. In addition to source code, a project contains image files, configuration files, uic, moc, and other files. It is a tool that makes it easy to build with these files.

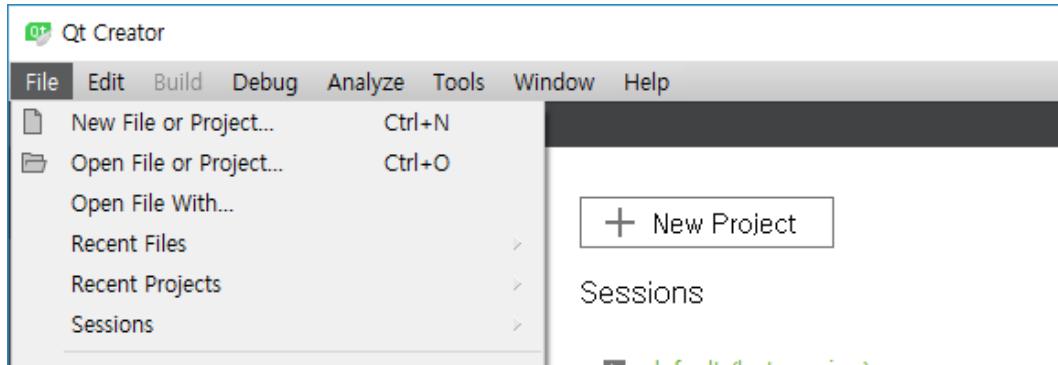
You can also condition certain features when building in Debug mode or Release mode. qmake uses the .pro extension and the filename is the project name. CMake's project file name is CMakeList.txt, but qmake's project file name is [project name.pro]. For example, if your project is named MyApp, the project name would be MyApp.pro.

qmake is by far the most popular build tool. Although there has been a recent trend toward using CMake, many projects still use qmake, and it is still the most popular build tool.

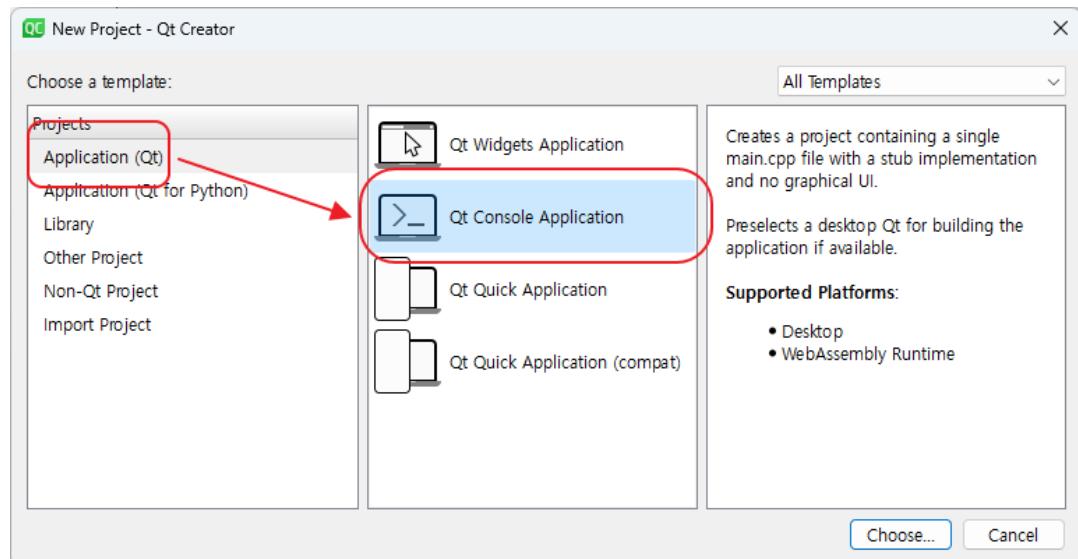
In this chapter, you will learn how to use qmake to develop both console and GUI applications.

## 4.1. Implementing Console applications with qmake

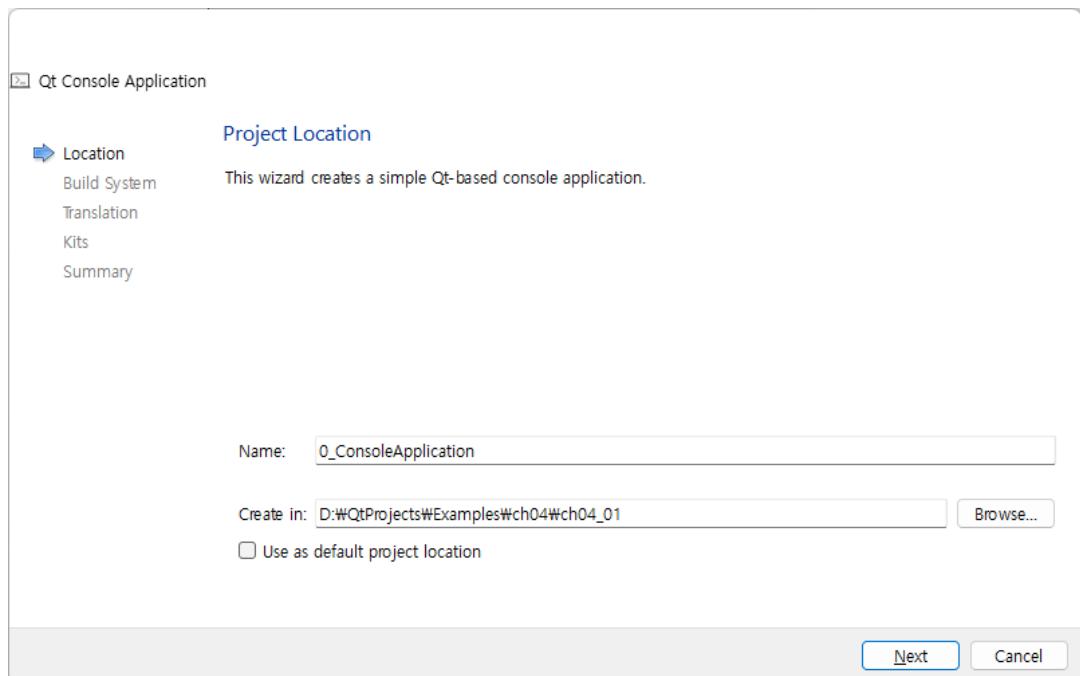
In this chapter, we will use qmake to create an application that prints "Hello World" to the Console window or Application Output window without a GUI. From the Qt Creator menu, click [File] -> [New File or Project].



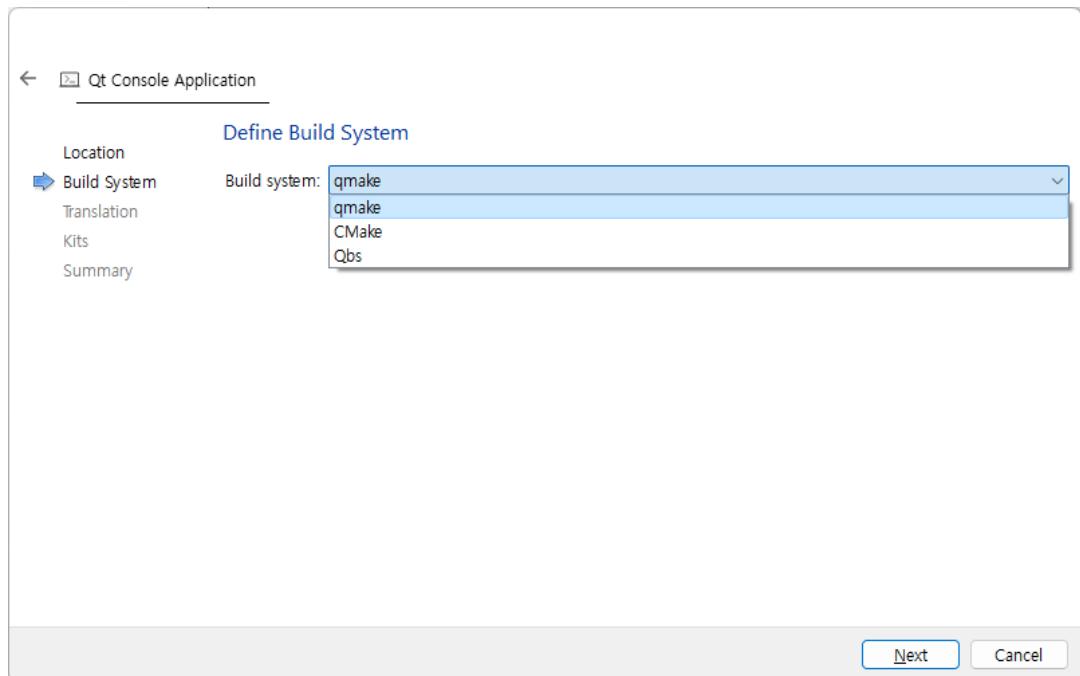
Clicking the [New File or Project] menu will load a dialog window as shown in the image below.



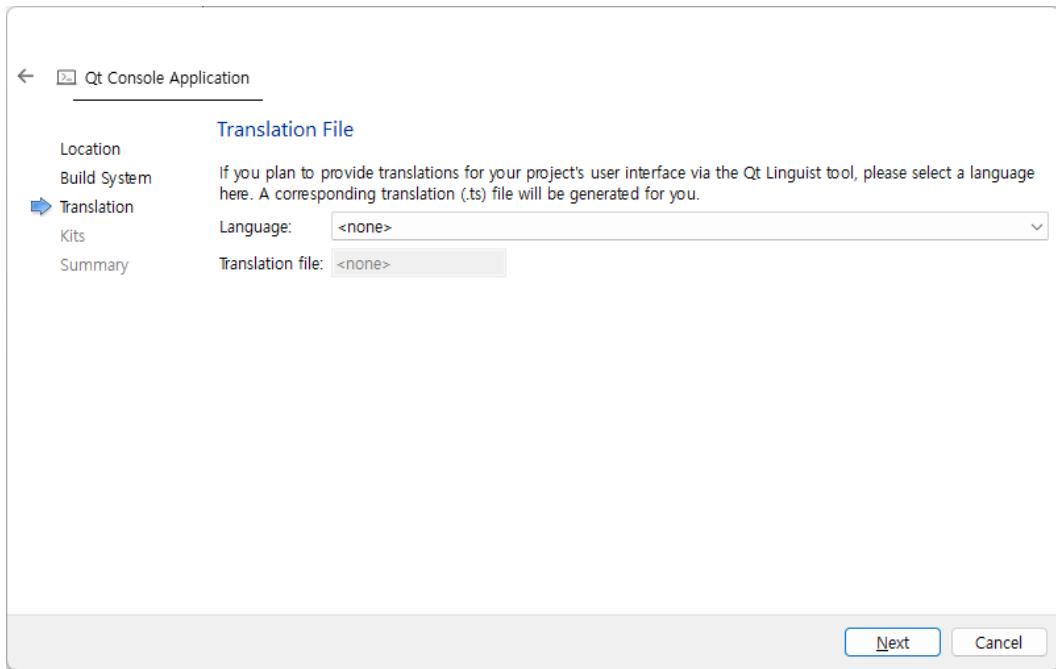
Select the [Qt Console Application] item as shown in the image above and click the [Choose] button at the bottom.



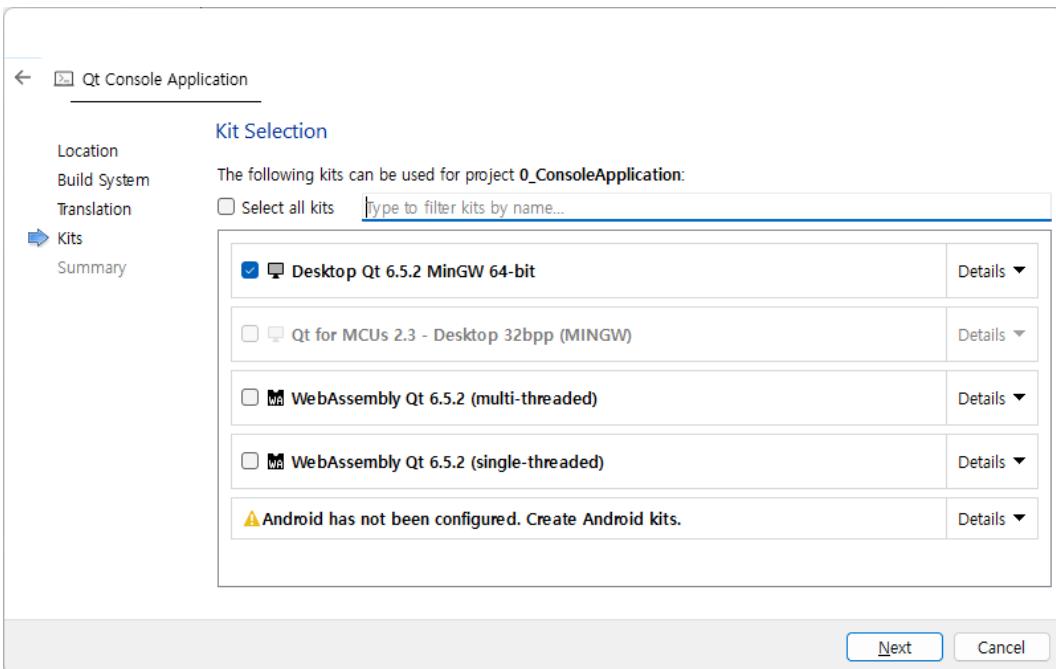
In the [Name] field, specify the name of the project, and in the [Create In] field, specify the name of the directory where the project will be located.



Select qmake as your build tool.



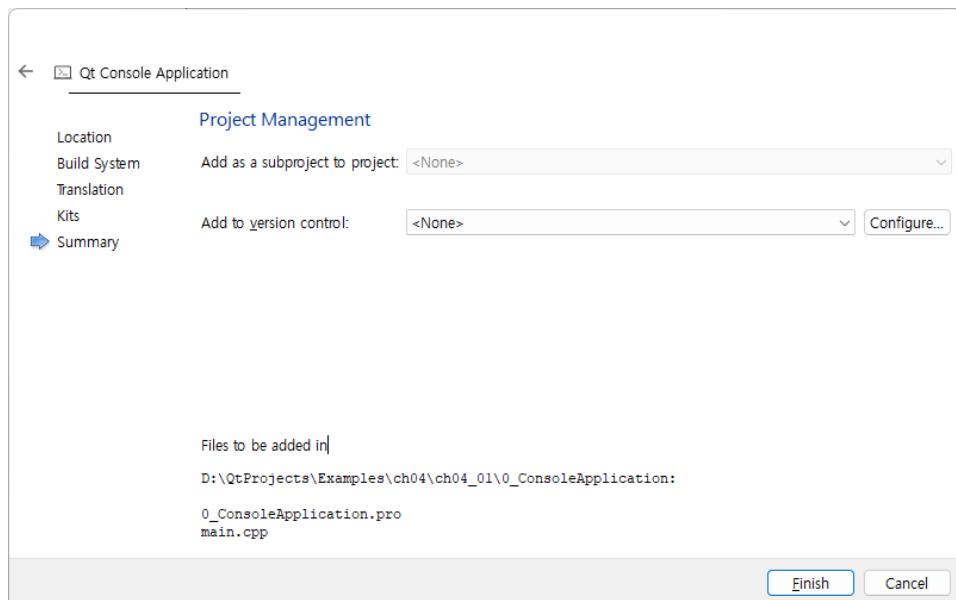
The dialog screen above is the dialog for multilingual support. In this dialog window, do not change anything and click the [Next] button at the bottom.



The above dialog is the window to select the dialog to compile. Select the MinGW compiler, as shown in the image above. The reason you don't see the MSVC compiler here is because it was not selected when Qt was installed. If you want to select MSVC as

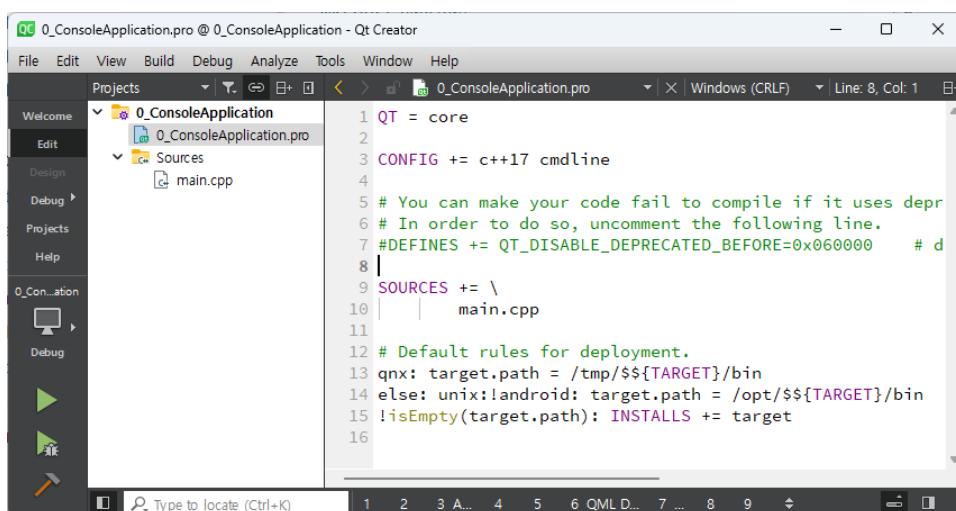
Jesus loves you.

the compiler, select and install the MSVC Component when installing Qt. Then you need to install the MSVC compiler. It is okay to install MSVC first.



This is the dialog for selecting a version control system (such as Git). Do not change anything in this dialog and click the Finish button at the bottom to complete the project creation dialog.

When the project creation is complete, the Qt Creator window will load the created project, as shown in the image below. Once it has finished loading, double-click the project file (with the .pro extension).



As you can see in the image above, the project file is fairly simple. Let's take a closer

look at the complete code. The source code below is the source code for the project file.

```
QT = core

CONFIG += c++17 cmdline

# You can make your code fail to compile if it uses deprecated APIs.
# In order to do so, uncomment the following line.
#DEFINES += QT_DISABLE_DEPRECATED_BEFORE=0x060000
#           disables all the APIs deprecated before Qt 6.0.0

SOURCES += \
    main.cpp

# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target
```

The QT keyword means that only the core module should be included in the project if you are only using this project's Console-based application. Qt organizes the libraries it provides into modules.

For example, to use the network APIs or libraries, you need to add the network module to the Qt keyword as shown below.

```
QT = core
QT += network
```

In the CONFIG keyword, "C++17" means that the version of C++ to use is version 17. "cmdLine" means for Console applications. This option is available on all platforms.

In addition to this method, on MS Windows you can use CONFIG += console, and on macOS you can use CONFIG -= app\_bundle.

In addition to the predefined variables, you can use CONFIG as a custom variable. For example, if you declare a variable called opengl as shown below, you can use it as follows.

```
CONFIG += opengl
```

You can register a variable called opengl in CONFIG as shown above and use it under certain conditions as shown below.

```
opengl {
```

```
TARGET = application-gl  
} else {  
    TARGET = application  
}
```

The SOURCE keyword specifies the source code. In this case, the SOURCE keyword contains only main.cpp because only the source code main.cpp exists. For example, if you have MyCode.h and MyCode.cpp source code, the MyCode.h and MyCode.cpp source code filenames are added to the SOURCE and HEADER code, as shown below.

```
SOURCES += \  
    MyCode.cpp \  
    main.cpp  
  
HEADERS += \  
    MyCode.h
```

In the source above, the Back Slash (" \ ") character means to wrap a new line.

In the source code below, the "qnx" keyword means the platform is QNX. target.path is the directory where the built executable will be located.

And in the project file, you can use branching statements such as if statements. else means if the platform is not QNX, and unix:!android: means the location of the built executable if the platform is Linux and not the Android platform. The TARGET variable is the executable name of the project.

```
# Default rules for deployment.  
qnx: target.path = /tmp/$${TARGET}/bin  
else: unix:!android: target.path = /opt/$${TARGET}/bin  
!isEmpty(target.path): INSTALLS += target
```

Up to this point, we've covered the project files. The contents of main.cpp were explained in the previous chapter, so we won't explain them in this chapter.

Let's write an example in main.cpp to print "Hello world" as shown below.

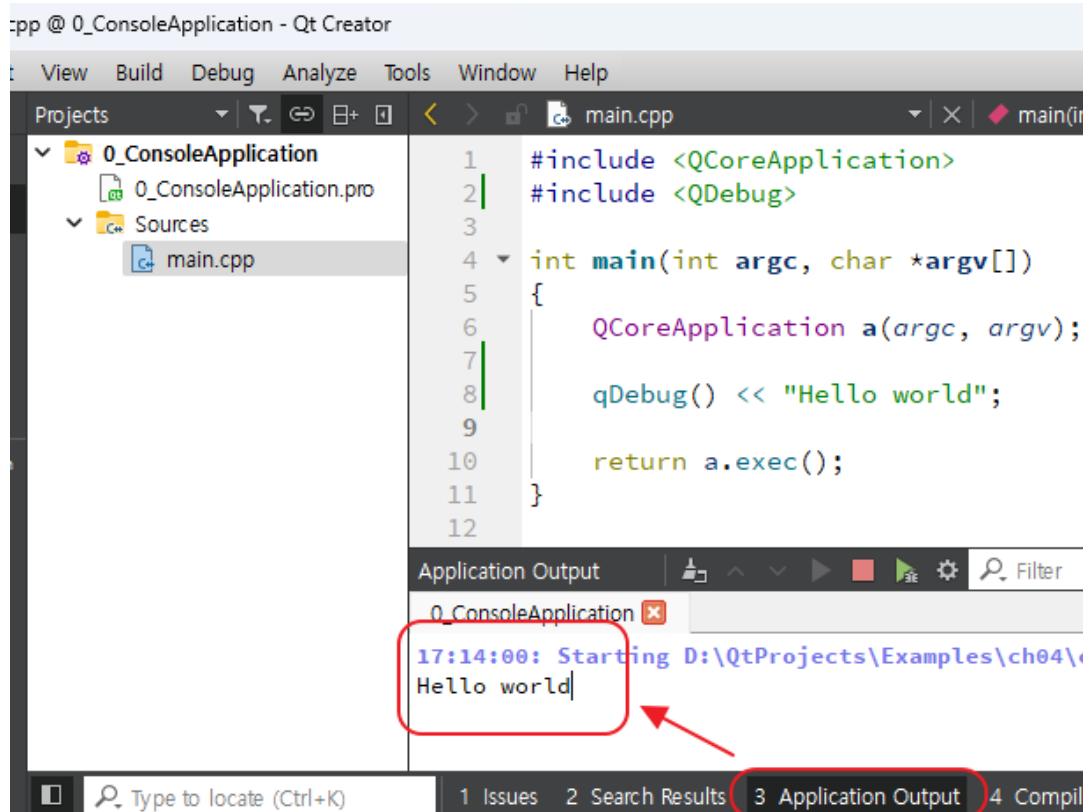
```
#include <QCoreApplication>  
#include <QDebug>  
  
int main(int argc, char *argv[])  
{
```

```
QCoreApplication a(argc, argv);

qDebug() << "Hello world";

return a.exec();
}
```

Let's write the source code as above, build it, and run it.

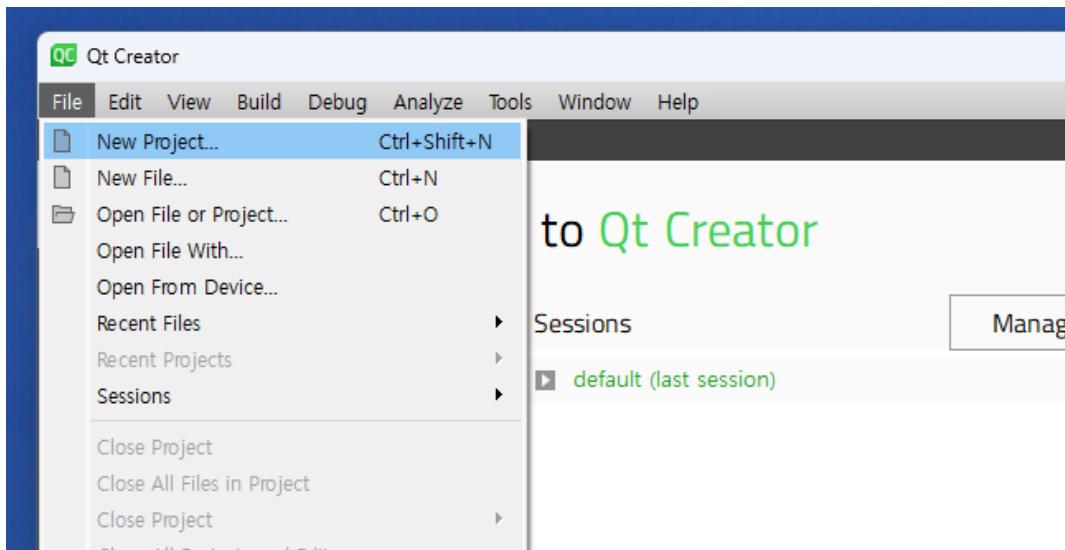


If you click on the Application Output window at the bottom of the Qt Creator window, as shown in the image above, you will see that it prints "Hello world".

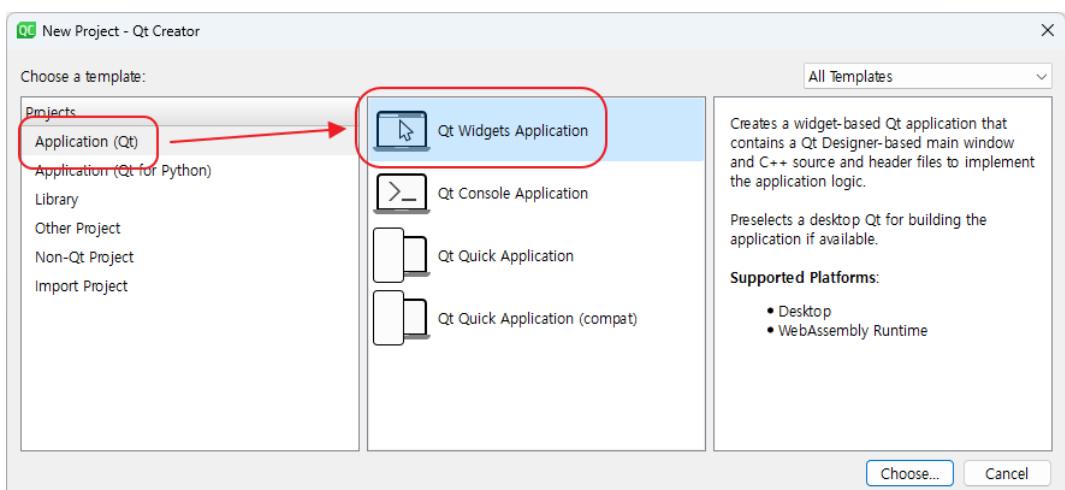
## 4.2. Implementing GUI application with qmake

In this chapter, we'll implement an example that places a button on the GUI and prints "Hello world" when clicked.

We'll cover events in more detail later, but for now, let's just implement the button event. Run the Qt Creator tool as shown below. And create a new project.



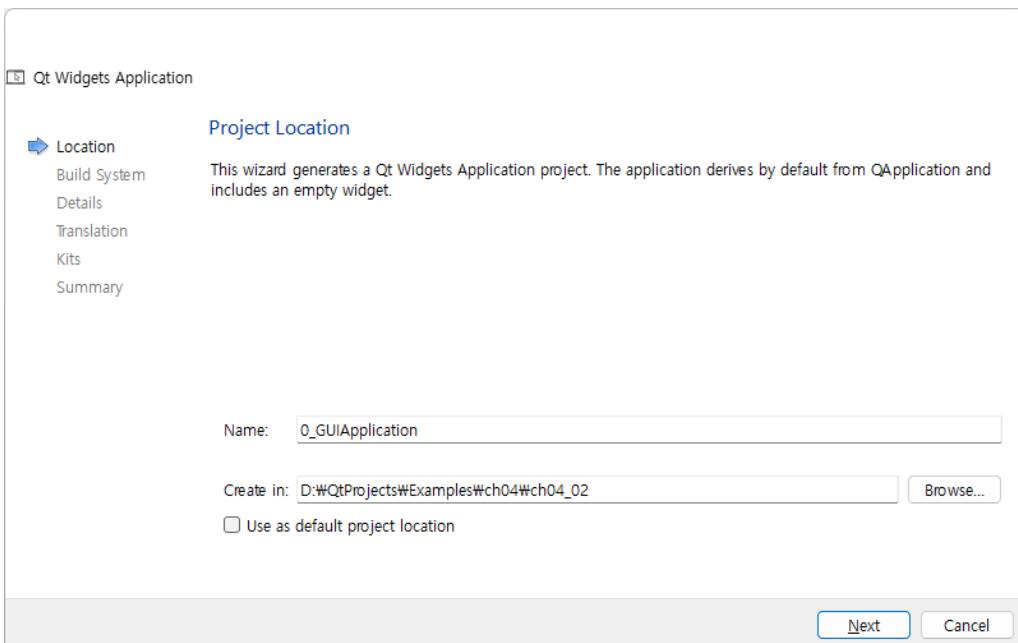
The shortcut is to click **Ctrl + Shift + N**.



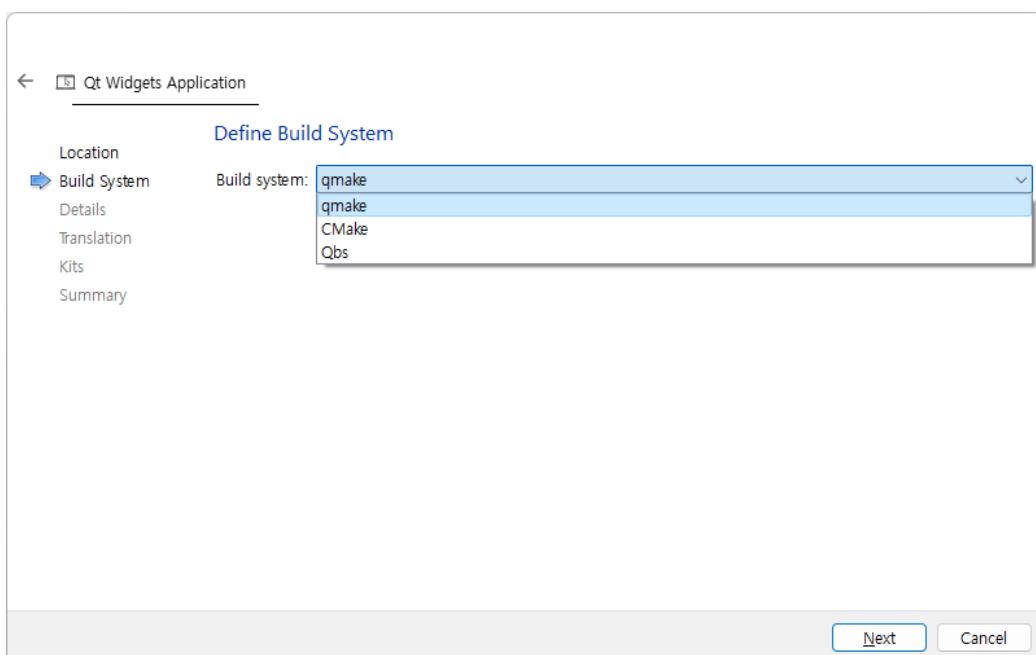
In the dialog, select the [Application Qt] item from the list on the left, select the [Qt Widget Application] item in the middle, and click the [choose] button at the bottom.

Jesus loves you.

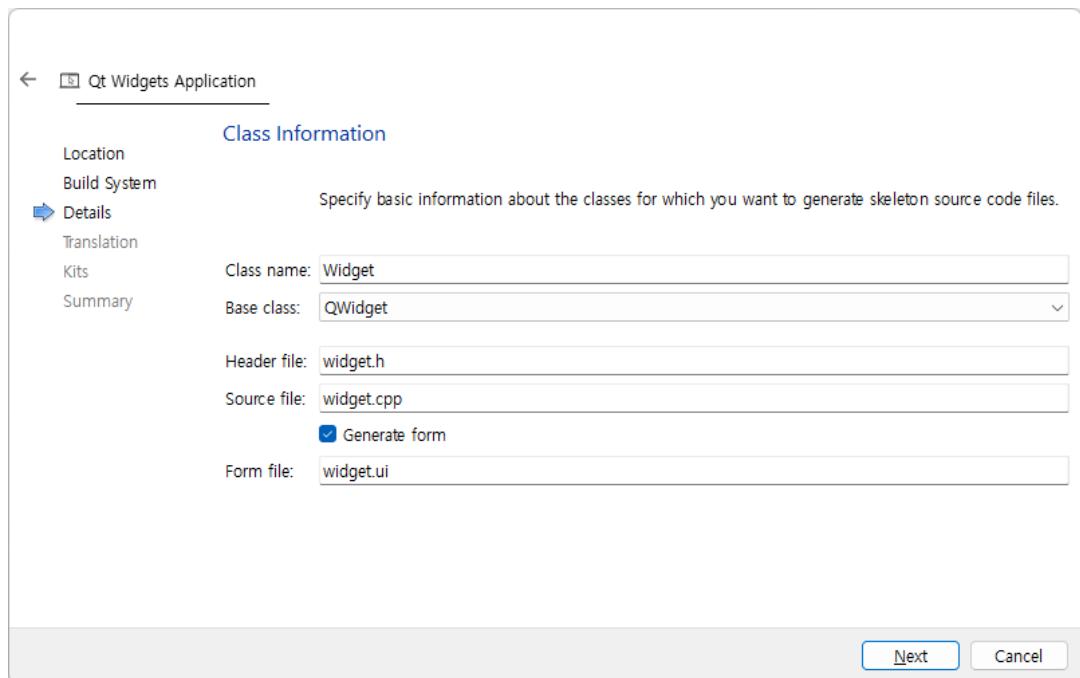
Next, enter a name for your project in the Name field. In the Create in section, select the directory (folder) where the project will be located and click the [Next] button.



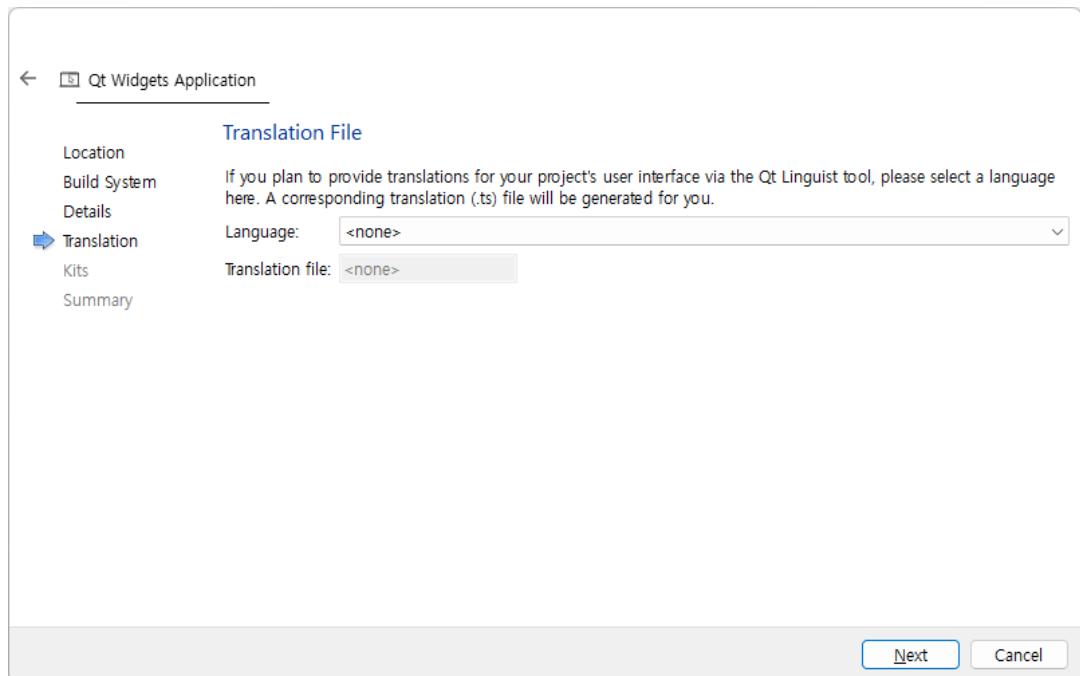
The following is a dialog to select the build tool. Select qmake and click the [Next] button.



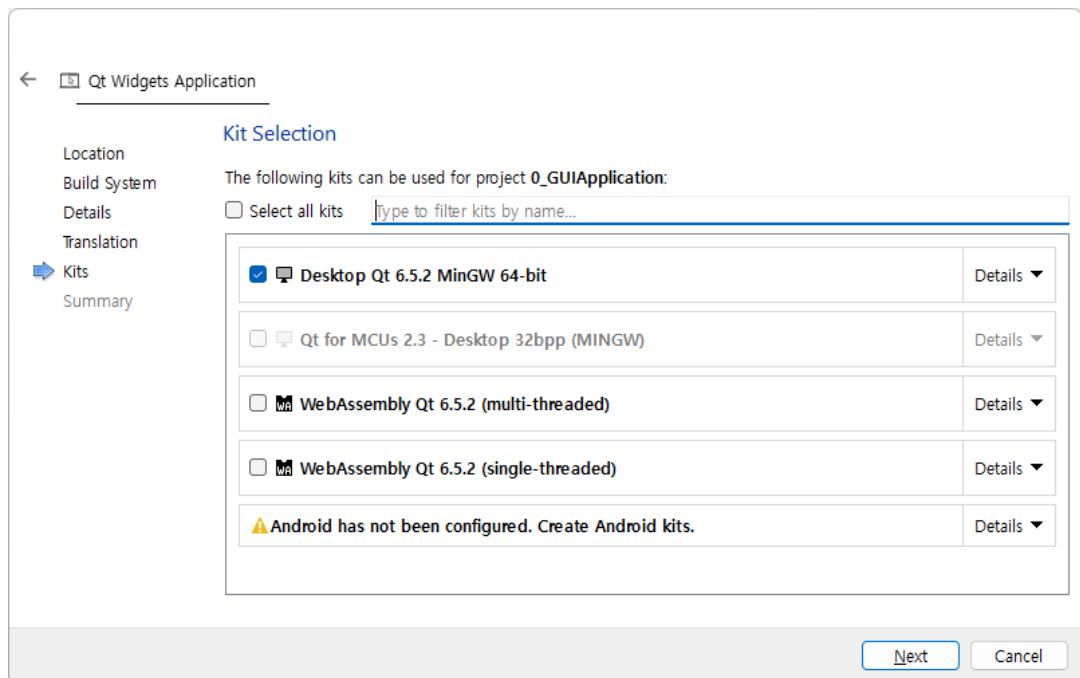
Next is a window to enter the Class information to be created. This window specifies the GUI class to be automatically created. Select [QWidget] from the combo box of the Base class item and click the [Next] button.



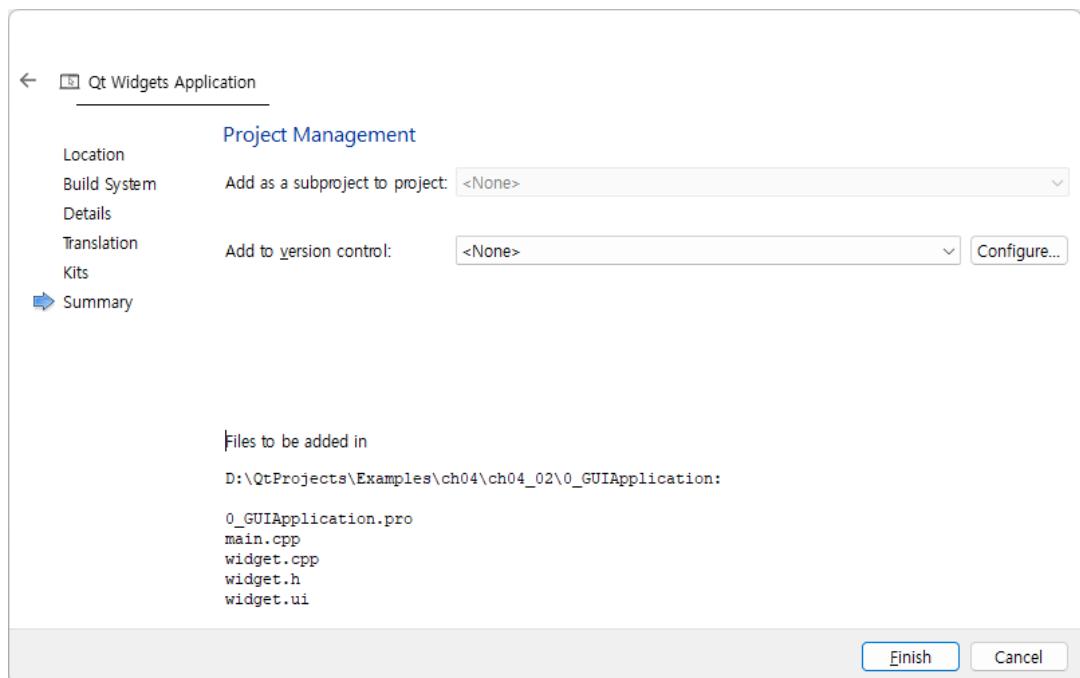
The following is the dialog window for multilingual support. In this window, do not change anything and click the [Next] button.

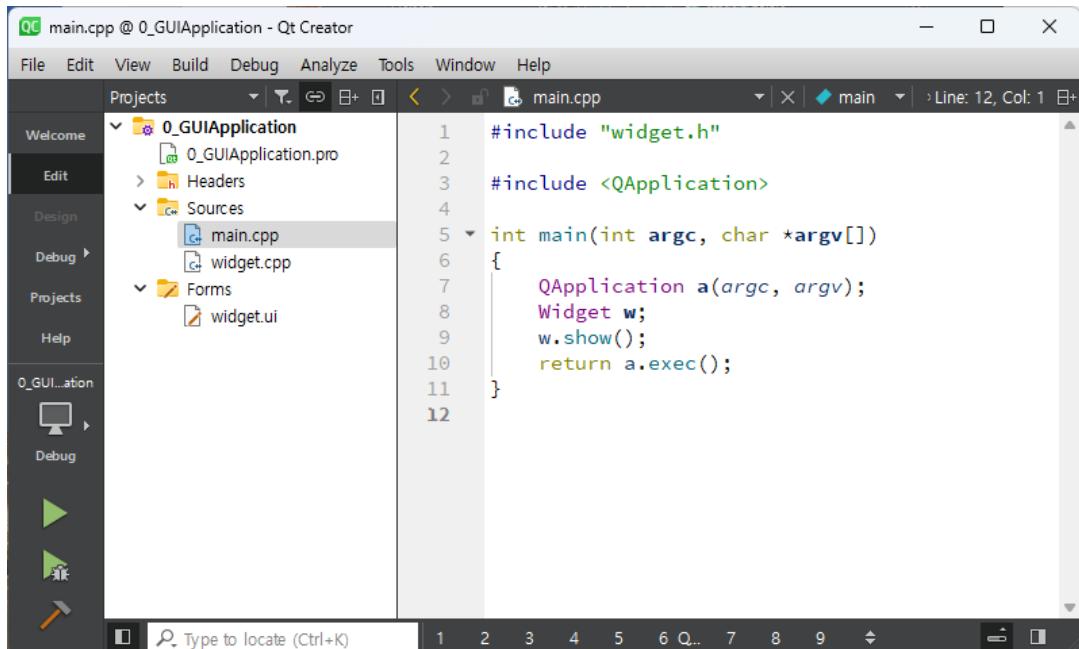


This is the screen to select the compiler to build the application with. In this case, we will select the MinGW compiler.



Next, you'll be asked to select a version control system. Don't select anything here and click the Finish button at the bottom to complete the project creation.





The screenshot shows the Qt Creator interface. The left sidebar has 'Edit' selected. The 'Projects' tab is active, displaying a project named '0\_GUIApplication'. Inside the project, there are 'Headers' and 'Sources' folders. The 'Sources' folder contains 'main.cpp' (selected), 'widget.cpp', and a 'Forms' folder with 'widget.ui'. The main editor window shows the content of 'main.cpp':

```
#include "widget.h"  
  
#include <QApplication>  
  
int main(int argc, char *argv[]){  
    QApplication a(argc, argv);  
    Widget w;  
    w.show();  
    return a.exec();  
}
```

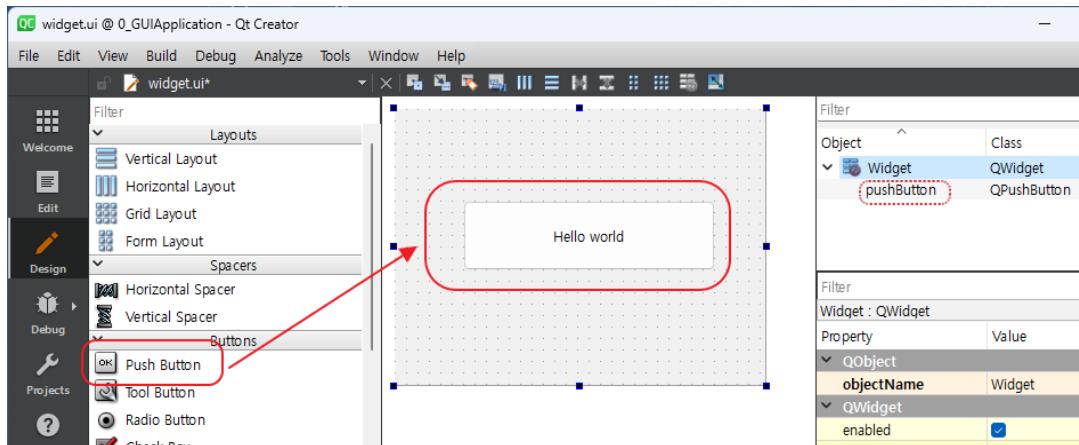
When you finish creating the project, the source code will be automatically generated as shown above. `main.cpp` is the source code that starts the program. In the `main.cpp` source code, create an instance of a class called `Widget`. Then we call the `show()` function to display the GUI of this instance. The `Widget` class is the class that Qt Creator automatically generated for us to implement the GUI.

You will also notice that a `widget.ui` file has been created. This file is used by the Qt Designer tool provided by Qt to easily drag and drop GUI widgets onto the GUI with the mouse. The `widget.ui` file stores the location and attribute information of the widgets placed by the Qt Designer tool. The file is formatted as XML.

Qt automatically converts this file to C++ source code at compile time. `widget.ui` is mapped to the `Widget` class, where the name of the class and the name of the ui file are stored identically, but they do not have to be the same.

Also, although only one ui file exists here, it is possible to have more than one GUI. For example, one class can be mapped to multiple UI files.

Here, double-clicking on the `widget.ui` file will automatically launch Qt Designer in the Qt Creator window. Double-click the `widget.ui` file to launch Qt Designer, as shown in the figure below.



From the Button tab on the left, drag the Push Button to the screen as shown in the image above. Then rename the button to "Hello world" as shown in the image above.

You can also change the unique name of the button above. The unique name for this instance is shown in the top right corner. In this case, we'll name it "pushButton" and save the ui file (Ctrl + S). Then click the [Edit] icon on the left side of the Qt Creator window to switch to the source code window.

The source code below is widget.h. Write the Slot function (event) that is called when you click this source code button like below.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
```

```
private slots:  
    void slot_clicked();  
  
};  
#endif // WIDGET_H
```

Create a slot\_clicked( ) function as shown in the source code above, and write the implementation of this Slot function in widget.cpp as follows.

In this function, we will use the function provided by Qt to print "Hello world". Include the QDebug header at the top of the widget.cpp file. Then, write the source code to print "Hello world" in the slot\_clicked( ) function.

```
#include "widget.h"  
#include "ui_widget.h"  
#include <QDebug>  
  
...  
  
void Widget::slot_clicked()  
{  
    qDebug() << "Hello world";  
}
```

Finally, to call the slot\_clicked( ) function when the button is clicked, we use the connect( ) function to connect the signal (event) and the Slot function, slot\_clicked( ).

```
...  
Widget::Widget(QWidget *parent)  
    : QWidget(parent)  
    , ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
    connect(ui->pushButton, SIGNAL(clicked()), this, SLOT(slot_clicked()));  
}  
...
```

The first argument to the connect( ) function above is the object that generates the signal. The unique name of this object is defined in the Qt Designer.

The second argument is the type of signal to be emitted. There are several signals: click,

Jesus loves you.

double-click, click and release, etc.

Here we specify the signal to be fired when this button is clicked. The third argument is the name of the instance with the Slot function to associate with this signal. In this case, we use this because we are specifying ourselves. The final argument specifies the Slot function to call. In this case, we specify the slot\_clicked( ) function.

At this point, you're done writing all the source code. Below is the complete widget.cpp source code.

```
#include "widget.h"
#include "ui_widget.h"

#include <QDebug>

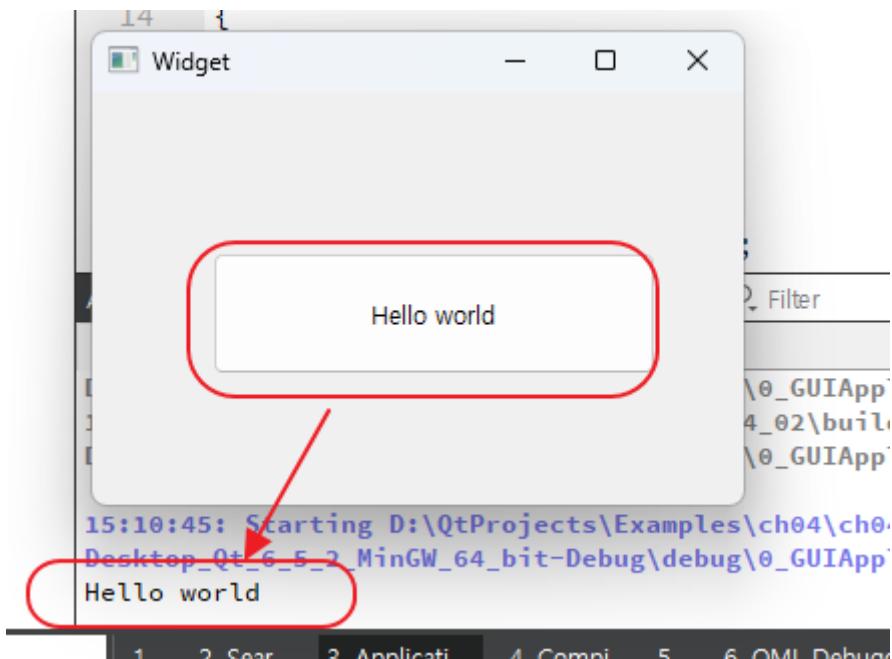
Widget::Widget(QWidget *parent)
    : QWidget(parent)
    , ui(new Ui::Widget)
{
    ui->setupUi(this);

    connect(ui->pushButton, SIGNAL(clicked()), this, SLOT(slot_clicked()));
}

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

void Widget::slot_clicked()
{
    qDebug() << "Hello world";
}
```

Next, let's build the application and then run it to see if it runs as shown below.

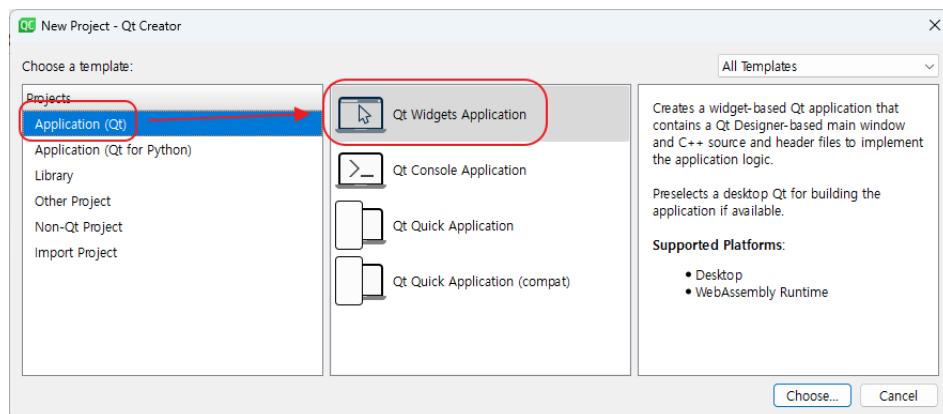


## 5. Qt GUI Widgets

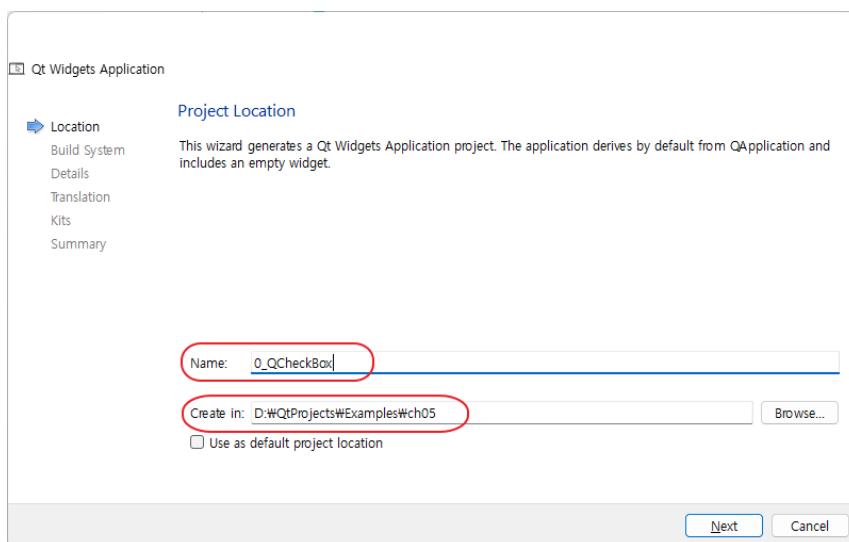
In Qt, buttons, checkboxes, radio buttons, etc. are called Widgets. Therefore, in this chapter, we will discuss the widgets provided by Qt.

- ✓ QCheckBox and QButtonGroup

Let's create an example using a QCheckBox and a QButtonGroup. In this example, we will also see how to use the image resource in the QCheckBox. When creating the project, select [QWidget Application] as shown below.

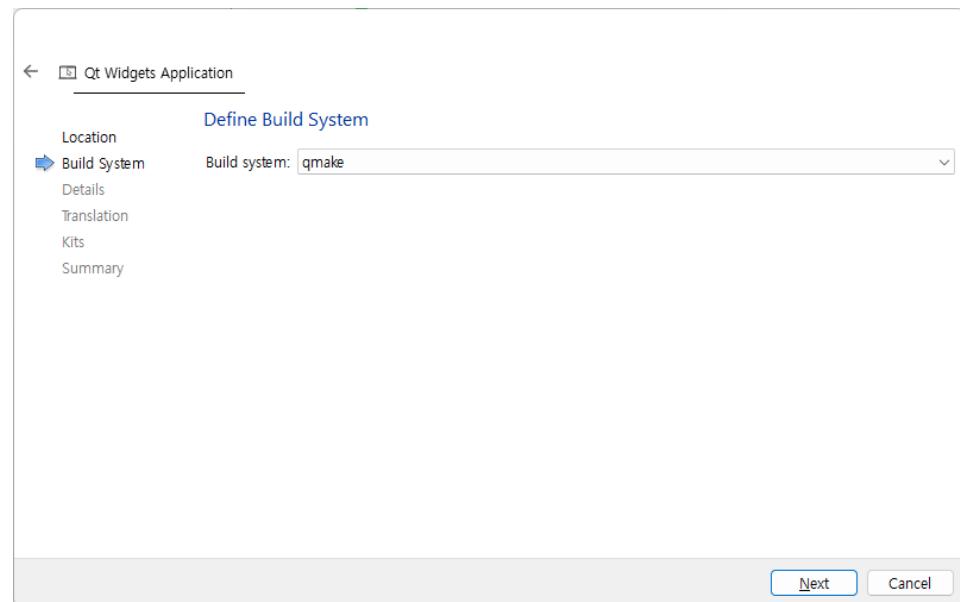


Next, select a project name and the directory where the project will be located and click the Next button.



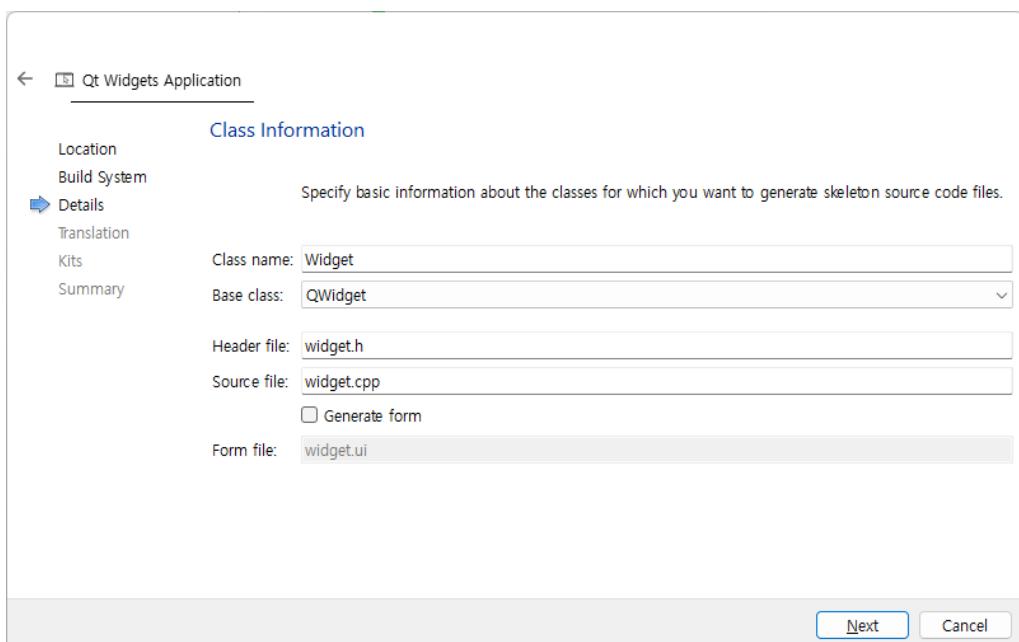
Jesus loves you.

In the window for choosing a project build tool, let's select qmake.



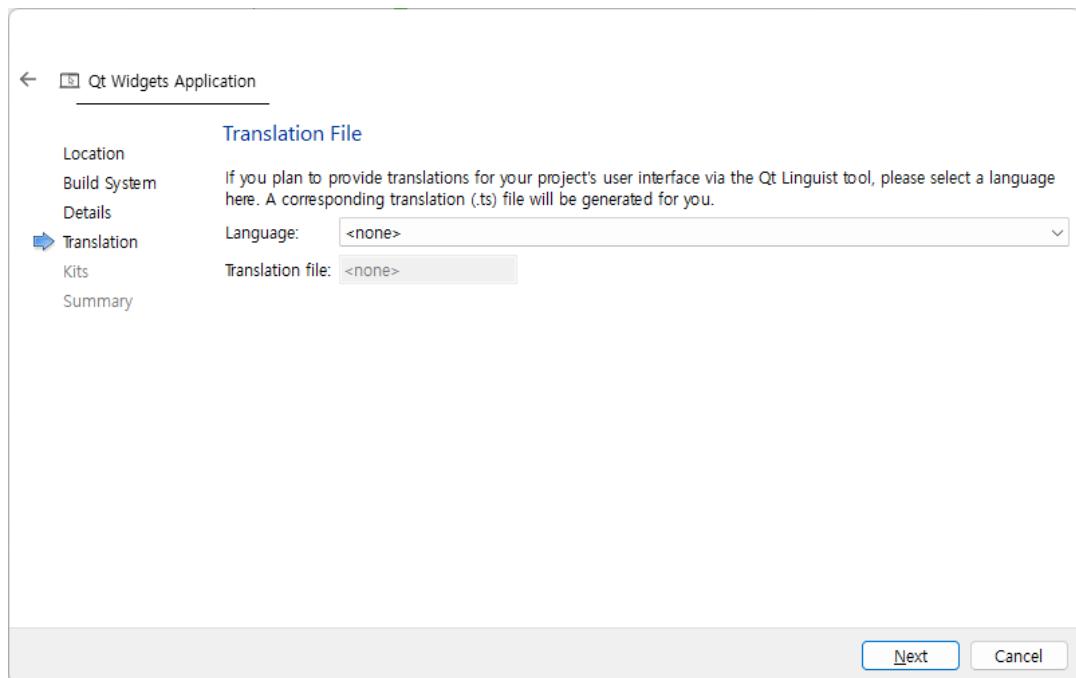
The next window is where you enter the information for the Class that will be created when the project is created. Class name is the name of the class. For Base class, select QWidget. If you select QWidget, the class will be created so that the class you create inherits from QWidget.

In this example, we will not use the Qt Designer, but rather write the GUI Widget directly in source code. Uncheck the Generate form item as shown in the dialogue window below.

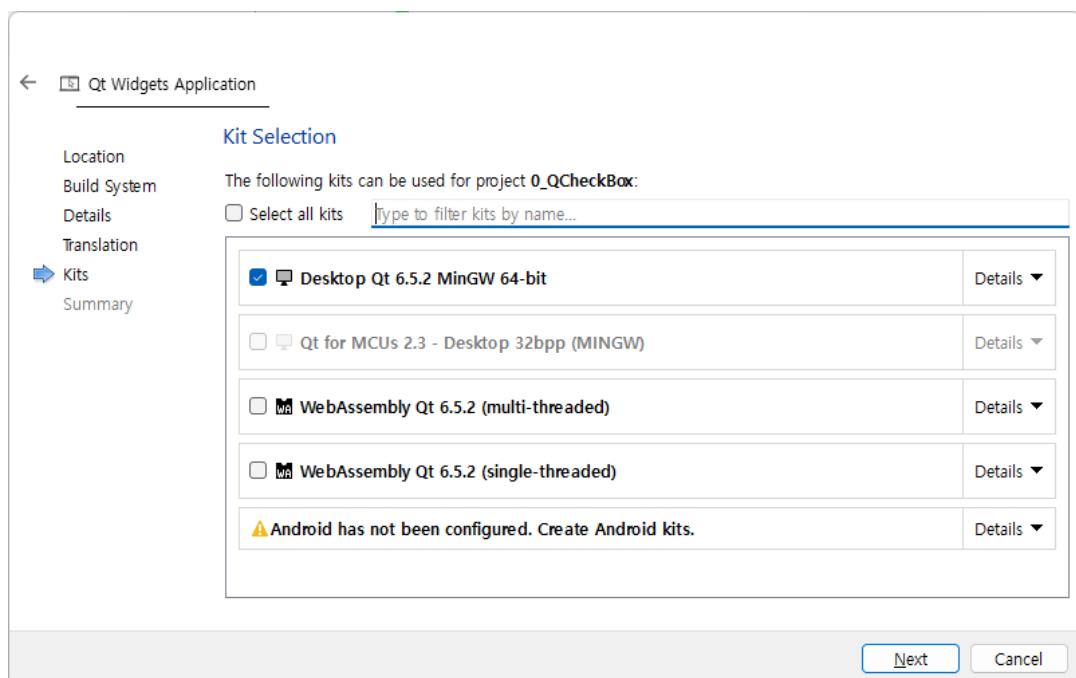


Jesus loves you.

Next, click the [Next] button without changing anything.



The compiler chooses MinGW.



The next window will ask you to select a version control system. Click the Finish button without changing anything.

Once you're done creating your project, let's write the following code in the widget.h file.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QCheckBox>
#include <QButtonGroup>

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    QButtonGroup      *m_chk_group[2];
    QCheckBox        *m_exclusive[3];
    QCheckBox        *m_non_exclusive[3];

private slots:
    void slot_chkChanged();

};

#endif // WIDGET_H
```

As shown above, we declare a QButtonGroup and a QCheckBox object, and we declare a slot\_chkChanged( ) function to handle signals (events).

This Slot function will handle signals from the QCheckBox. To declare a Slot function, you need to declare an access restrictor (private or public) and the keyword "slots".

Next, create widget.cpp as shown below.

```
#include "widget.h"

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    QStringList str1[3] = {"Game", "Office", "Develop"};
    QStringList str2[3] = {"P&rogramming", "Q&t", "O&S"};
```

```
int xpos = 30;
int ypos = 30;

m_chk_group[0] = new QButtonGroup(this);
m_chk_group[1] = new QButtonGroup(this);

for(int i = 0 ; i < 3 ; i++)
{
    m_exclusive[i] = new QCheckBox(str1[i], this);
    m_exclusive[i]->setGeometry(xpos, ypos, 120, 30);
    m_chk_group[0]-> addButton(m_exclusive[i]);

    m_non_exclusive[i] = new QCheckBox(str2[i], this);
    m_non_exclusive[i]->setGeometry(xpos + 120, ypos, 120, 30);
    m_chk_group[1]-> addButton(m_non_exclusive[i]);

    connect(m_exclusive[i], SIGNAL(clicked()),
            this,           SLOT(slot_chkChanged ()));
}

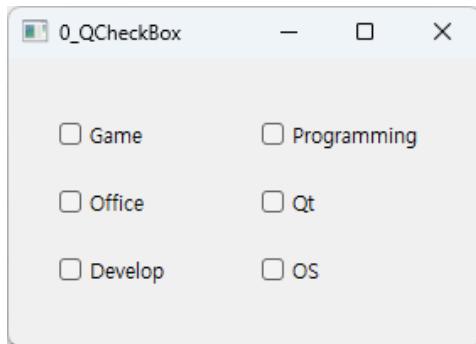
ypos += 40;
}

m_chk_group[0]->setExclusive(false);
m_chk_group[1]->setExclusive(true);
}

void Widget::slot_chkChanged()
{
    for(int i = 0 ; i < 3 ; i++) {
        if(m_exclusive[i]->checkState())
        {
            qDebug("checkbox %d selected ", i+1);
        }
    }
}

Widget::~Widget()
{
```

Let's write the above source code, build it, and run it.

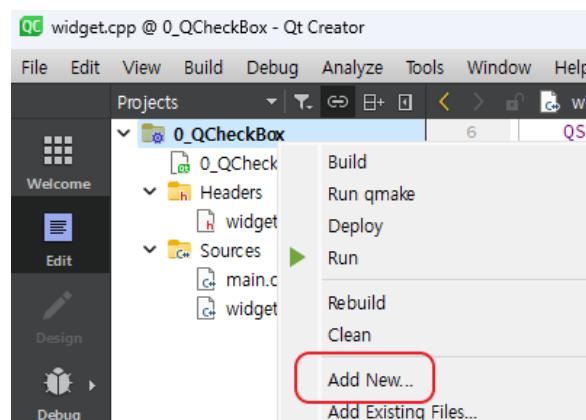


The Game, Office, and Develop items on the left and Programming, Qt, and OS on the right are separated into QButtonGroups. The QCheckBox items on the left allow multiple selections, but the items on the right are implemented so that multiple selections are not possible. To disable multiple selection, you can use the `setExclusive( )` member function provided by the QButtonGroup class.

```
...
m_chk_group[0]->setExclusive(false);
m_chk_group[1]->setExclusive(true);
...
```

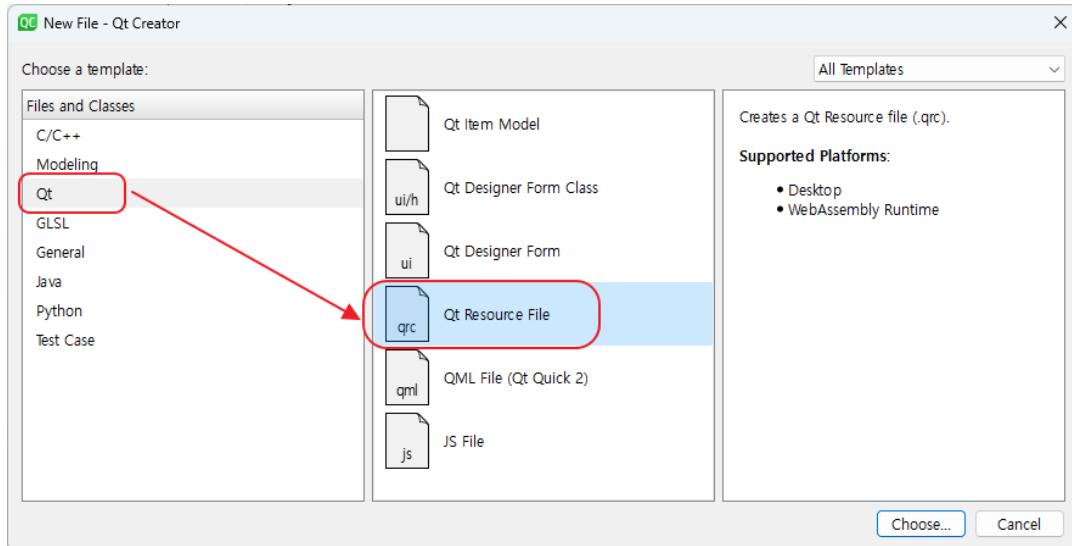
As shown above, the first argument passed to the `setExclusive( )` function can be used to enable or disable multiple selection. Using false will allow multiple selection and using true will prevent multiple selection despite being a checkbox.

In this example, we'll use the image resource in Qt to create a QCheckBox. Create a resource directory in your project directory and copy the image you want to use. In the Qt Creator window, position the mouse over the project's name and right-click. This will load a menu.

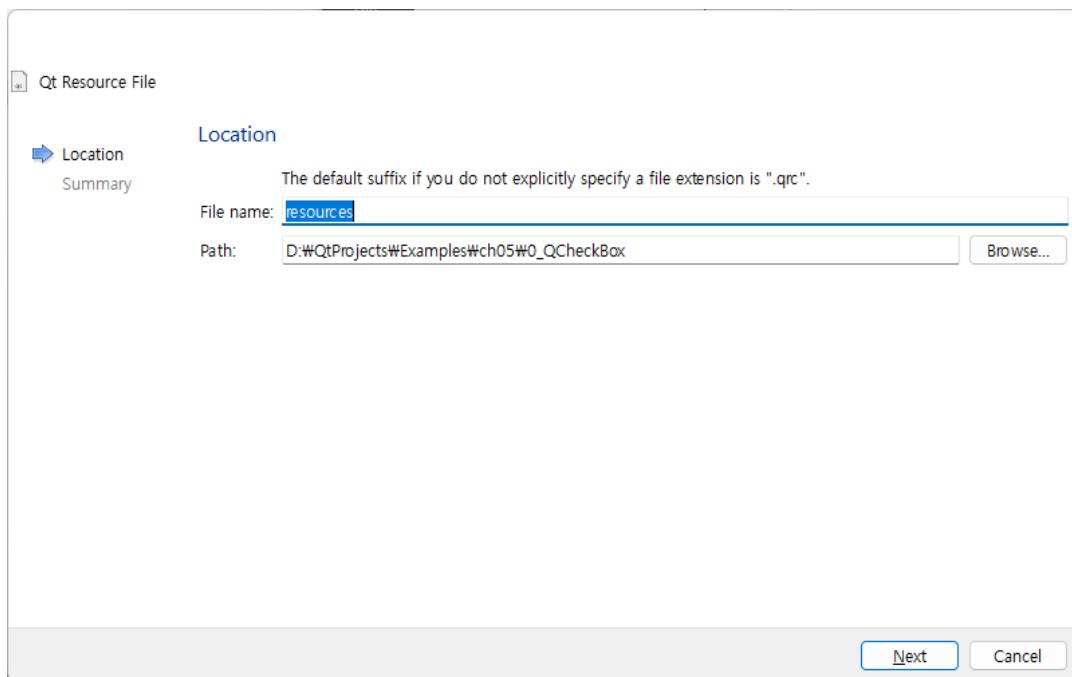


Jesus loves you.

Click on the [Add New] menu. Then, in the dialogue window, select the Qt item from the left-hand side, and then click the Qt Resource File item on the middle tab.



In the next dialogue window, enter the name of the resource file. The resource name can be anything you want. In this case, we'll use "resources".



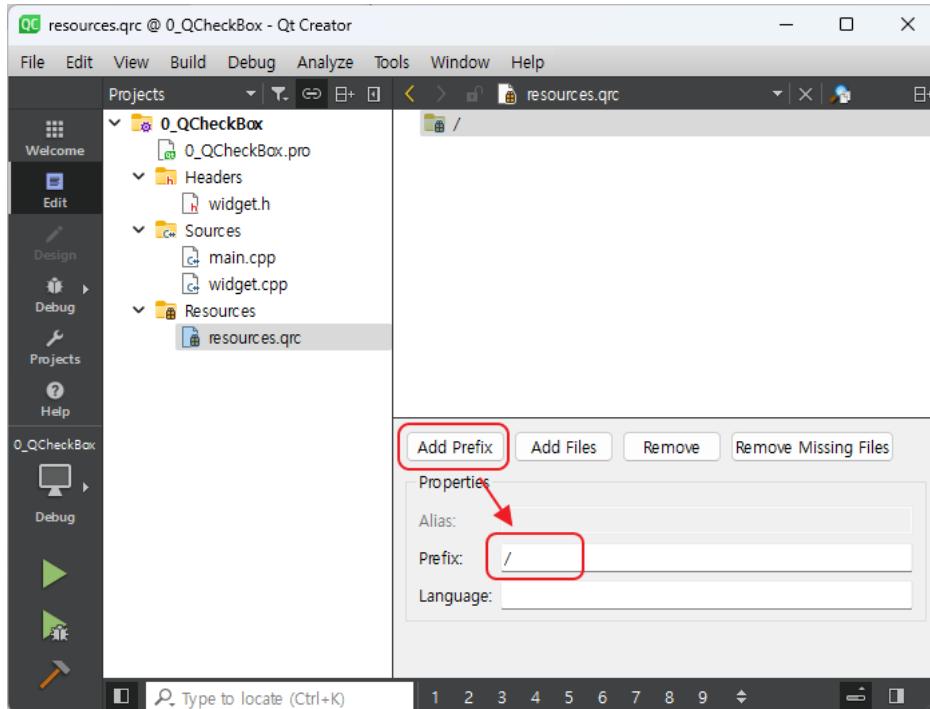
In the next dialogue window, do not change anything and click the Finish button at the bottom.

You should then see that the resources.qrc file has been created, as shown in the image

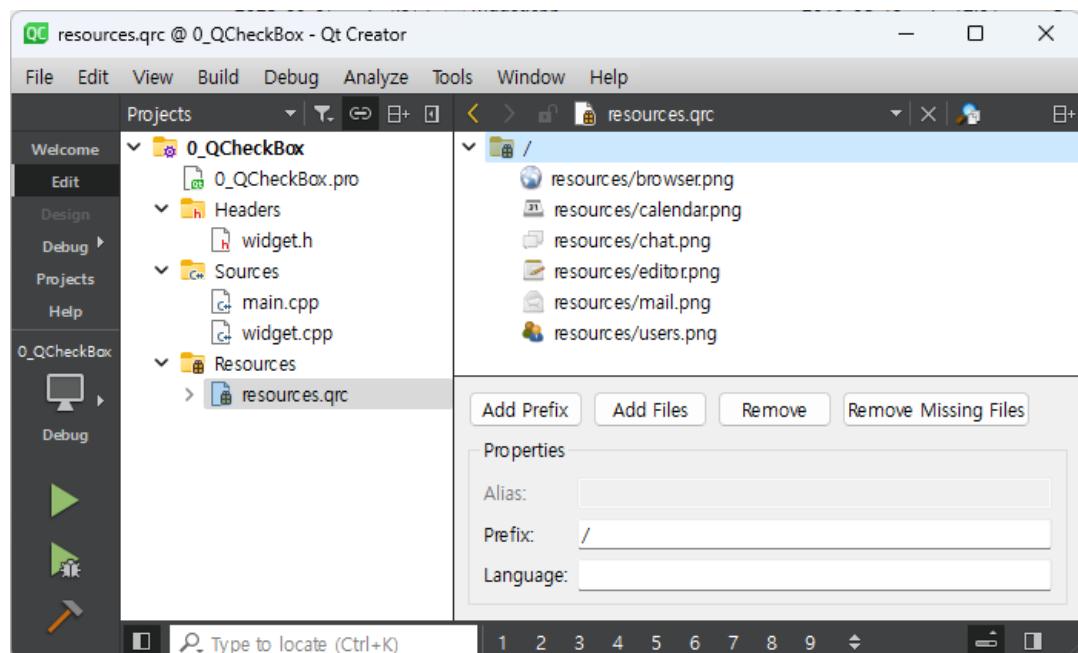
Jesus loves you.

below.

Next, press [Add Prefix]. Then enter "/" in the Prefix field at the bottom.



Next, click the Add Files button and add your image files.



Register the image files as shown in the image above. After you have finished registering

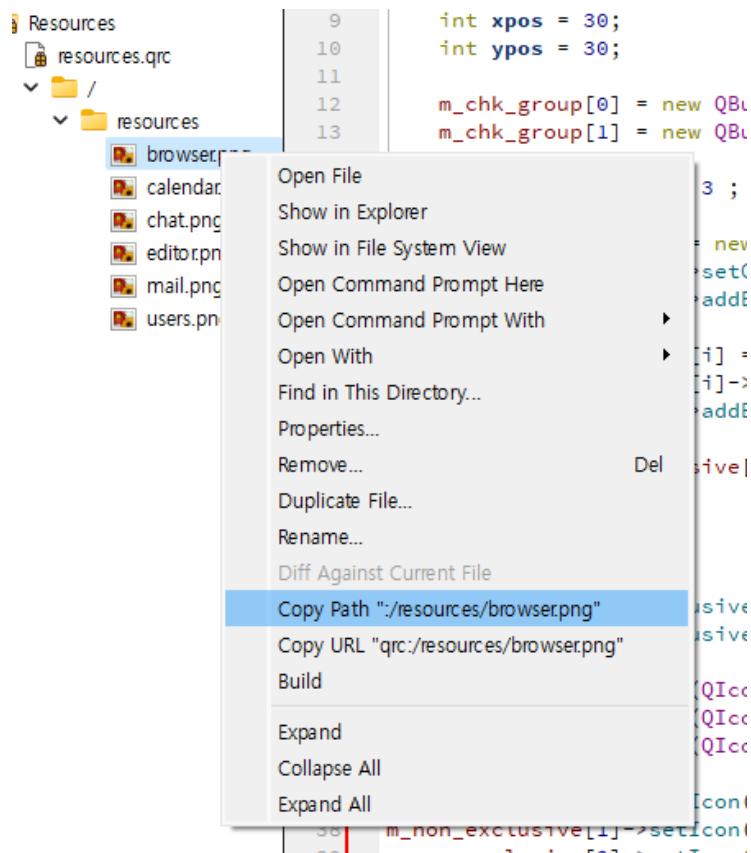
the images, add the following to the constructor function in the widget.cpp source code

```
...
m_chk_group[0]->setExclusive(false);
m_chk_group[1]->setExclusive(true);

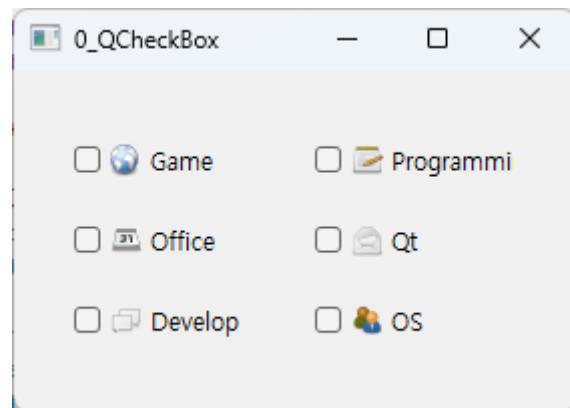
m_exclusive[0]->setIcon(QIcon(":resources/browser.png"));
m_exclusive[1]->setIcon(QIcon(":resources/calendar.png"));
m_exclusive[2]->setIcon(QIcon(":resources/chat.png"));

m_non_exclusive[0]->setIcon(QIcon(":resources/editor.png"));
m_non_exclusive[1]->setIcon(QIcon(":resources/mail.png"));
m_non_exclusive[2]->setIcon(QIcon(":resources/users.png"));
...
```

To find out where each image's resource file is located, hover over the image in the left project pane of Qt Creator and right-click to bring up a menu. From the menu, select Copy Path or Copy URL to copy the image's resource name into memory. You can then paste the source code.



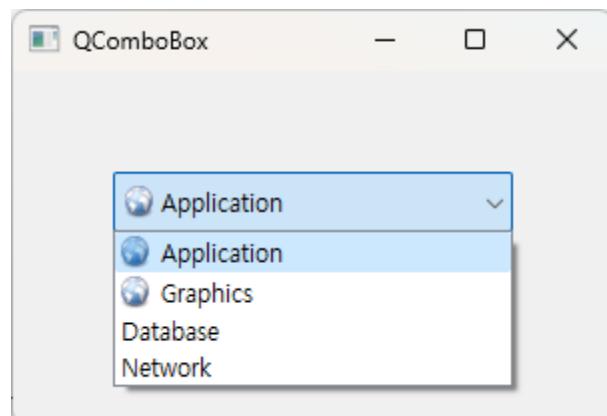
If you're up to this point, let's build and run it.



✓ QComboBox

When the user clicks on the widget, a pop-up menu appears and provides a GUI to select one of the registered items. To register an item on the QComboBox widget, you can use text or an image with the item.

```
combo = new QComboBox(this);
combo->setGeometry(50, 50, 200, 30);
...
combo->addItem(QIcon(":resources/browser.png"), "Application");
combo->addItem(QIcon(":resources/mail.png"), "Graphics");
combo->addItem("Database");
combo->addItem("Network");
...
```

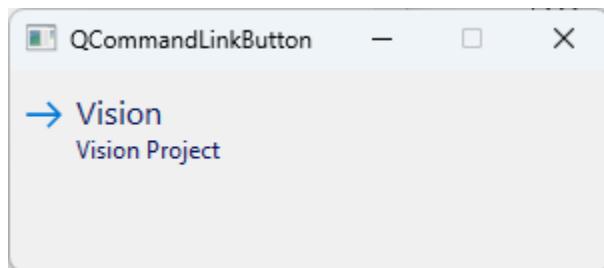


The full source of the example is in Ch05 > 01\_QComboBox.

✓ QCommandLinkButton

This widget is a widget that provides the same functionality as the QPushButton widget. As a feature, this widget provides the same style as the Link Button provided by MS Windows.

```
...
cmmBtn = new QCommandLinkButton ("Vision", "Vision Project", this);
cmmBtn->setFlat(true);
...
```



✓ QDate and QDateEdit

QDateEdit provides a GUI for displaying or changing dates. QDateEdit can display the date set in the QDate class. The QDate class can specify the year, month, and day, or it can get the current date from the system and connect it to the QDateEdit widget for display.

```
QDate dt1 = QDate(2020, 1, 1);
QDate dt2 = QDate::currentDate();

dateEdit[0] = new QDateEdit(dt1.addYears(2), this);
dateEdit[0]->setGeometry(10, 10, 140, 40);

dateEdit[1] = new QDateEdit(dt1.addMonths(3), this);
dateEdit[1]->setGeometry(160, 10, 140, 40);
```

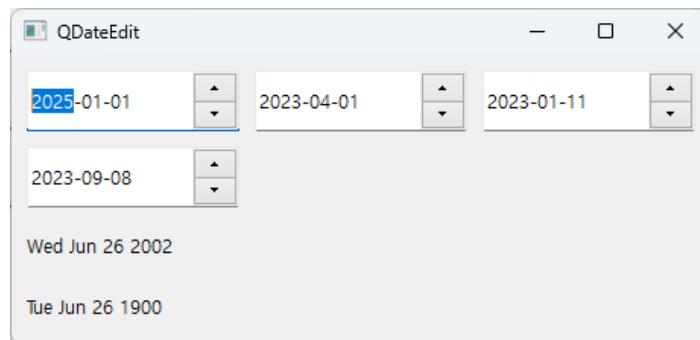
```

dateEdit[2] = new QDateEdit(dt1.addDays(10), this);
dateEdit[2]->setGeometry(310, 10, 140, 40);

dateEdit[3] = new QDateEdit(dt2, this);
dateEdit[3]->setGeometry(10, 60, 140, 40);

...

```



The QDate class provides a way to display a date in the desired format, with member functions that return a QString data type.

```

QDate dt = QDate::currentDate();
QString str = dt.toString("yyyy.MM.dd");

```

Expression characters	Visibility
d	1~31 (1~31)
dd	Display 1-31 with 2 digits (01-31)
ddd	Display the day of the week with 3-digit characters (Mon to Sun)
dddd	Display days of the week with complete letters (Monday to Sunday)
M	Display as a number (1-12)
MM	Display 01~12 with 2 digits (01~12)
MMM	Display the month as a 3-digit letter (Jan to Dec)
MMMM	Display months as complete letters (January to December)

yy	Display the year as 2 digits (00-99)
yyyy	Display the year as 4 digits (2002)

The complete source for the example is in the 03\_QDateEdit directory.

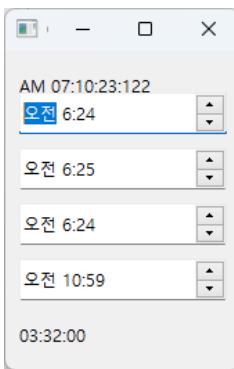
- ✓ QTime class and QTimeEdit widget class

The QTime class makes it easy to implement time-related functions for application development, such as displaying time and comparing it to certain conditions. QTimeEdit provides the ability to display the time obtained from a QTime class in a GUI interface.

```
QTime ti = QTime(6, 24, 55, 432); // Hour, Minute, Second, Milli second  
  
QTimeEdit *qte;  
qte = new QTimeEdit(ti, this);  
qte->setGeometry(10, 30, 150, 30);  
...
```

Class QTime provides a number of member functions to facilitate time-related manipulations. For example, to add a specified number of seconds and milliseconds to the current time, the addSecs( ) and addMSecs( ) member functions can be used to add seconds and milliseconds to the time to get the desired result.

```
qte[1] = new QTimeEdit(ti1.addMSecs(200), this);  
qte[1]->setGeometry(10, 30, 150, 30);  
  
qte[2] = new QTimeEdit(ti1.addSecs(2), this);  
qte[2]->setGeometry(10, 30, 150, 30);  
...
```



Class QTime can display the time in various formats using the `toString()` member function.

```
QTime ti = QTime(7, 10, 23, 122);
QLabel *lb_str = new QLabel(ti.toString("AP hh:mm:ss:zzz"), this);

lb_str->setGeometry(10, 10, 150, 30);
...
```

<Table> Time display formats

표현 문자	표시 형태
h	Display Hour as a number from 0 to 23
hh	Display Hour as a number from 00 ~ 23
m	Display Minute as a number from 0~59
mm	Display Minute as a number from 00~59
s	Display Second as a number from 0~59
ss	Display Second as a number from 00~59
z	Display Milli second as a number from 0~999
zzz	Display Milli second as a number from 000~999
AP	Display AM/PM (Uppercase)
ap	Display am/pm (Lowercase)

See the 04\_QTimeEdit directory for the complete source of the examples.

✓ QDateTime and QDateTimeEdit

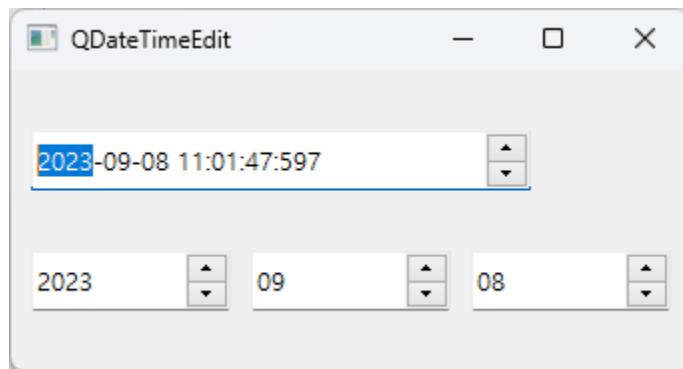
Class QDateTime is a class that can handle dates and times together, and class QDateTimeEdit can display dates and times. The setDisplayFormat( ) member function of class QDateTimeEdit can display the date and time according to a format.

```
QDateTimeEdit *qde1;  
qde1 = new QDateTimeEdit(QDateTime::currentDateTime(), this);  
qde1->setDisplayFormat( "yyyy-MM-dd hh:mm:ss:zzz" );  
qde1->setGeometry(10, 30, 250, 50); // x, w, width, height
```

QDateTimeEdit can change the date and time displayed in the widget with spin box buttons and allows you to specify a range when changing the date and time.

```
QDateTimeEdit *qde[3];  
  
qde[0] = new QDateTimeEdit(QDate::currentDate(), this);  
qde[0]->setMinimumDate(QDate::currentDate().addYears(-3));  
qde[0]->setMaximumDate(QDate::currentDate().addYears(3));  
qde[0]->setDisplayFormat("yyyy");  
qde[0]->setGeometry(10, 90, 100, 30);  
  
qde[1] = new QDateTimeEdit(QDate::currentDate(), this);  
qde[1]->setMinimumDate(QDate::currentDate().addMonths(-2));  
qde[1]->setMaximumDate(QDate::currentDate().addMonths(2));  
qde[1]->setDisplayFormat("MM");  
qde[1]->setGeometry(120, 90, 100, 30);  
  
qde[2] = new QDateTimeEdit(QDate::currentDate(), this);  
qde[2]->setMinimumDate(QDate::currentDate().addDays(-20));  
qde[2]->setMaximumDate(QDate::currentDate().addDays(20));  
qde[2]->setDisplayFormat("dd");  
qde[2]->setGeometry(230, 90, 100, 30);  
...
```

The setMinimumDate( ) and setMaximumDate( ) member functions can change the date only within the range set by specifying a range.

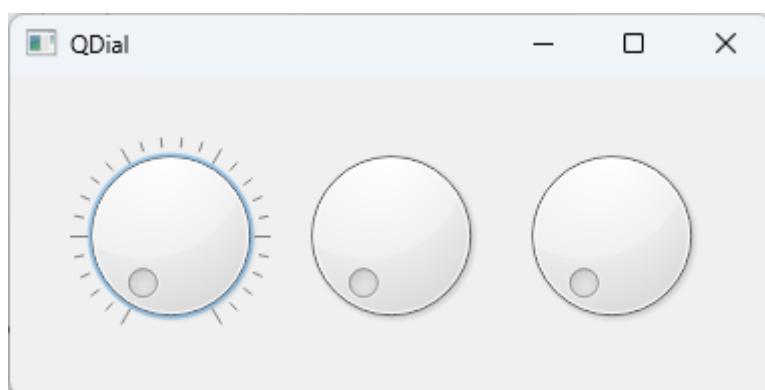


In addition to dates, the QDateTimeEdit class can specify minimum and maximum values of time using the `setMinimumTime()` and `setMaximumTime()` member functions. See the `05_QDateTimeEdit` directory for the complete source of examples.

✓ QDial

The QDial widget class provides a dial-like GUI interface. For example, it provides a GUI for turning a dial to adjust the volume.

```
for(int i = 0 ; i < 3 ; i++, xpos += 110) {  
    dial[i] = new QDial(this);  
    dial[i]->setRange(0, 100);  
    dial[i]->setGeometry(xpos, 30, 100, 100);  
}  
dial[0]->setNotchesVisible(true);  
connect(dial[0], &QDial::valueChanged, this, &Widget::changedData);  
...
```



The `setRange( )` member allows you to specify the range of the `QDial` widget. The `setNotchesVisible( )` member provides the ability to display graduations on the `QDial` widget. When you drag the `QDial` widget with the mouse, you can register the `valueChanged( )` signal to get the current value of the changed `QDial`.

```
Widget::Widget(QWidget *parent) : QWidget(parent)
{
    int xpos = 30;
    for(int i = 0 ; i < 3 ; i++, xpos += 110) {
        dial[i] = new QDial(this);
        dial[i]->setRange(0, 100);
        dial[i]->setGeometry(xpos, 30, 100, 100);
    }
    dial[0]->setNotchesVisible(true);
    connect(dial[0], &QDial::valueChanged, this, &Widget::changedData);
}

void Widget::changedData()
{
    qDebug("QDial 1 value : %d", dial[0]->value());
}
...
```

When the value of the first dialogue of `QDial` is changed, the `changeData( )` Slot function is called. We have not yet learned how to handle events using Signal / Slot, so let's just understand that the Slot function is the function that handles events generated by the `connect( )` function. More details will be covered in the Signal and Slot sections. See the `06_QDail` directory for example sources.

- ✓ QSpinBox and QDoubleSpinBox

The `QSpinBox` class provides a GUI for changing integer values of the `int` data type using the up and down buttons. To use the double data type, you can use the `QDoubleSpinBox` widget.

The `QSpinBox` and `QDoubleSpinBox` widget classes allow you to limit the range of values that the user can change, and you can use characters in the Prefix and Suffix parts of the number to indicate specific characters or units. For example, a currency symbol could be

used inside the widget.

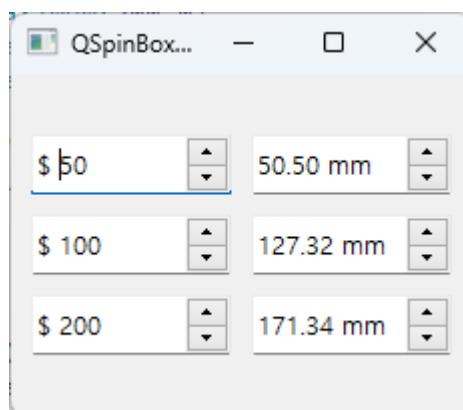
```
...
int ypos = 30;
int val[3] = {50, 100, 200};
double double_val[3] = {50.5, 127.32, 171.342};

for(int i = 0 ; i < 3 ; i++)
{
    spin[i] = new QSpinBox(this);
    spin[i]->setMinimum(10);
    spin[i]->setMaximum(300);
    spin[i]->setValue(val[i]);
    spin[i]->setGeometry(10, ypos, 100, 30);

    doublespin[i] = new QDoubleSpinBox(this);
    doublespin[i]->setMinimum(10.0);
    doublespin[i]->setMaximum(300.0);
    doublespin[i]->setValue(double_val[i]);
    doublespin[i]->setGeometry(120, ypos, 100, 30);

    spin[i]->setPrefix("$ ");
    doublespin[i]->setSuffix(" mm");

    ypos += 40;
}
...
...
```



See the 7\_QSpinBox\_QDoubleSpinBox directory for the full source of the examples.

✓ QPushButton and QFocusFrame

The QPushButton widget provides button functionality. QFocusFrame is useful if you need to use an Outer Line for the outside. In addition to this, QFocusFrame can use QStyle (a Style Sheet used in HTML). Here's how to use QFocusFrame to draw an Outer Line around the outside of a QPushButton widget.

```
QPushButton *btn[3];

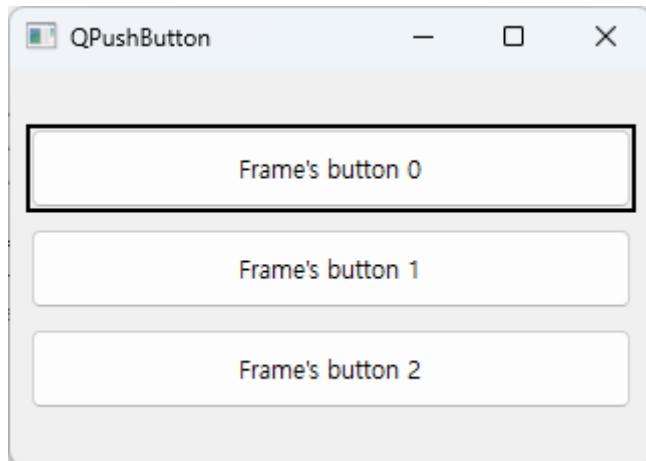
int ypos = 30;
for(int i = 0 ; i < 3 ; i++) {
    QString str = QString("Frame's button %1").arg(i);
    btn[i] = new QPushButton(str, this);
    btn[i]->setGeometry(10, ypos, 300, 40);
    ypos += 50;
}

connect(btn[0], &QPushButton::clicked, this, &Widget::btn_click);
connect(btn[0], &QPushButton::pressed, this, &Widget::btn_pressed);
connect(btn[0], &QPushButton::released, this, &Widget::btn_released);

QFocusFrame *btn_frame = new QFocusFrame(this);

btn_frame->setWidget(btn[0]);
btn_frame->setAutoFillBackground(true);
...
```

The first argument to the QPushButton class constructor is the text to be displayed on the button. The second argument specifies the parent class of the QPushButton class.



See the 08\_QPushButton\_QFocusFrame directory for the full source of the examples.

#### ✓ QFontComboBox

The QFontComboBox widget provides a GUI for selecting fonts on the GUI. The widget lists the fonts in alphabetical order and also shows the appearance of the fonts.

```
QFontComboBox *fontcb[5];
for(int i = 0 ; i < 5 ; i++)
    fontcb[i] = new QFontComboBox(this);

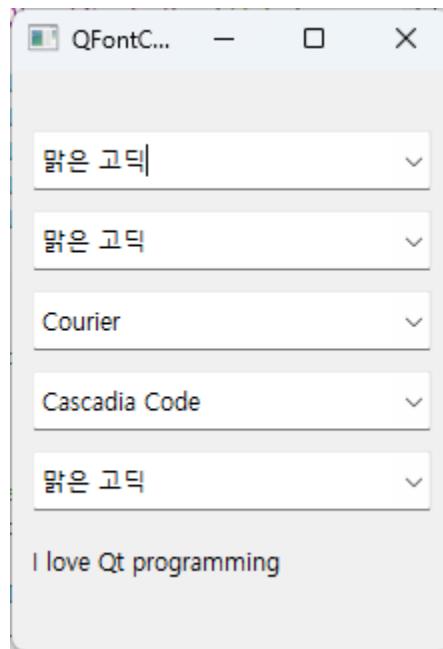
fontcb[0]->setFontFilters(QFontComboBox::AllFonts);
fontcb[1]->setFontFilters(QFontComboBox::ScalableFonts);
fontcb[2]->setFontFilters(QFontComboBox::NonScalableFonts);
fontcb[3]->setFontFilters(QFontComboBox::MonospacedFonts);
fontcb[4]->setFontFilters(QFontComboBox::ProportionalFonts);

int ypos = 30;
for(int i = 0 ; i < 5 ; i++) {
    fontcombo[i]->setGeometry(10, ypos, 200, 30);
    ypos += 40;
}
...
```

setFontFilters( ) provides the ability to filter the list of fonts to be listed on the QFontComboBox widget by using constants to display specific fonts.

See the 09\_QFontComboBox directory for example sources.

Jesus loves you.



The following table lists the constants available in the `setFontFilters( )` member function.

Constant	Value	Description
<code>QFontComboBox::AllFonts</code>	0	All fonts
<code>QFontComboBox::ScalableFonts</code>	0x1	Dynamic auto-convertible fonts on zoom
<code>QFontComboBox::NonScalableFonts</code>	0x2	Fonts that don't offer dynamic automatic conversion
<code>QFontComboBox::MonospacedFonts</code>	0x3	Fonts that provide a consistent character width shape
<code>QFontComboBox::ProportionalFonts</code>	0x4	Fonts with balanced width and height

#### ✓ QLabel and QLCDNumber

The `QLabel` widget provides the ability to display text or images in your application. The `QLCDNumber` widget can only display numbers and can display numbers in the form of a digital clock. The `QLCDNumber` widget can be used with the ":" character, which is

used to display the time.

```
...
QLabel *lbl[3];
lbl[0] = new QLabel("I love Qt programming", this);
lbl[0]->setGeometry(10, 30, 130, 40);

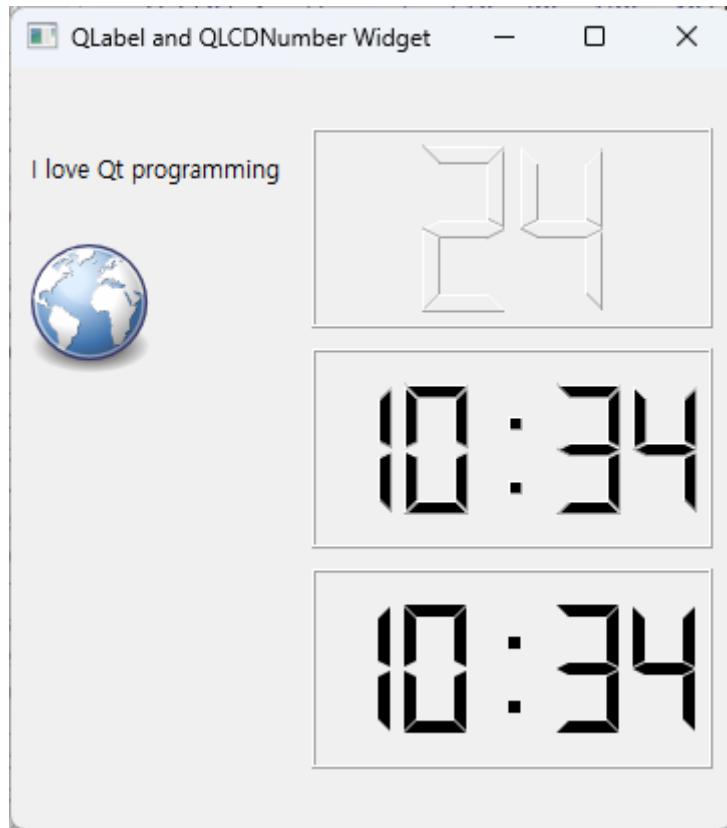
QPixmap pix = QPixmap(":resources/browser.png");
lbl[1] = new QLabel(this);
lbl[1]->setPixmap(pix);
lbl[1]->setGeometry(10, 70, 100, 100);

QLCDNumber *lcd[3];
lcd[0] = new QLCDNumber(2, this);
lcd[0]->display(24);
lcd[0]->setGeometry(150, 30, 200, 100);
lcd[0]->setSegmentStyle(QLCDNumber::Outline);

lcd[1] = new QLCDNumber(5, this);
lcd[1]->display("10:34");
lcd[1]->setGeometry(150, 140, 200, 100);
lcd[1]->setSegmentStyle(QLCDNumber::Filled);

lcd[2] = new QLCDNumber(5, this);
lcd[2]->display("10:34");
lcd[2]->setGeometry(150, 250, 200, 100);
lcd[2]->setSegmentStyle(QLCDNumber::Flat);
...
```

The QPixmap class provides an API for rendering images on the GUI. Using the QPixmap class, images can be displayed on the GUI using the setPixmap( ) member function of the QLabel widget.



See the 10\_QLabel\_QLCDNumber directory for the full source of the examples.

#### ✓ QLineEdit

The QLineEdit widget provides a GUI for entering and modifying text. The QLineEdit class provides functions such as copy, paste, and cut in the widget.

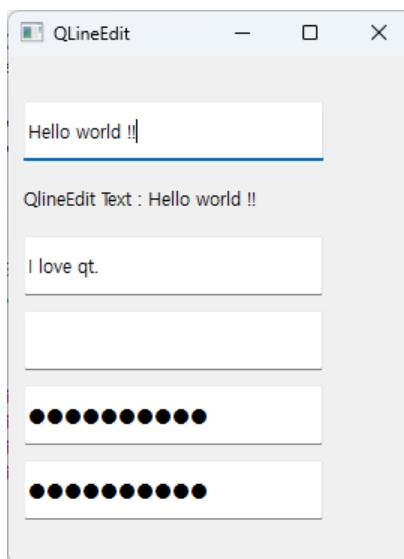
```
Widget::Widget(QWidget *parent) : QWidget(parent)
{
    edit[0] = new QLineEdit("", this);
    lbl = new QLabel("QlineEdit Text : ", this);

    connect(edit[0], SIGNAL(textChanged(QString)),
            this,      SLOT(textChanged(QString)));

    edit[0]->setGeometry(10, 30, 200, 40);
    lbl->setGeometry(10, 80, 250, 30);
}
```

```
int ypos = 120;
for(int i = 1 ; i < 5 ; i++) {
    edit[i] = new QLineEdit("I love qt.", this);
    edit[i]->setGeometry(10, ypos, 200, 40);
    ypos += 50;
}
edit[1]->setEchoMode(QLineEdit::Normal);
edit[2]->setEchoMode(QLineEdit::NoEcho);
edit[3]->setEchoMode(QLineEdit::Password);
edit[4]->setEchoMode(QLineEdit::PasswordEchoOnEdit);
}

void Widget::textChanged(QString str)
{
    lbl->setText(QString("QlineEdit Text : %1").arg(str));
}
...
```



QLineEdit can make text invisible or process passwords. To do this, the `setEchoMode()` member function must take the following constants as arguments

Constant	Value	Description
QLineEdit::Normal	0	Same style as default
QLineEdit::NoEcho	1	Styles where text is invisible and the cursor position does not change

QLineEdit::Password	2	Text appears as "*" character
QLineEdit::PasswordEchoOnEdit	3	Same as the default style, but shows "*" when the focus is moved when the text changes

See the 11\_QLineEdit directory for example sources.

✓ QMenu and QMenuBar

The QMenu and QMenuBar class widgets provide menu functionality. The QMenu widget provides the addAction( ) and addMenu( ) member functions to create menus. Below is the source code for these functions.

```
Widget::Widget(QWidget *parent) : QWidget(parent)
{
    menuBar = new QMenuBar(this);

    menu[0] = new QMenu("File");
    menu[0]->addAction("Edit");
    menu[0]->addAction("View");
    menu[0]->addAction("Tools");

    act[0] = new QAction("New", this);
    act[0]->setShortcut(Qt::CTRL | Qt::Key_A);
    act[0]->setStatusTip("This is a New menu.");

    act[1] = new QAction("Open", this);
    act[1]->setCheckable(true);

    menu[1] = new QMenu("Save");
    menu[1]->addAction(act[0]);
    menu[1]->addAction(act[1]);

    menu[2] = new QMenu("Print");
    menu[2]->addAction("Page Setup");
    menu[2]->addMenu(menu[1]);

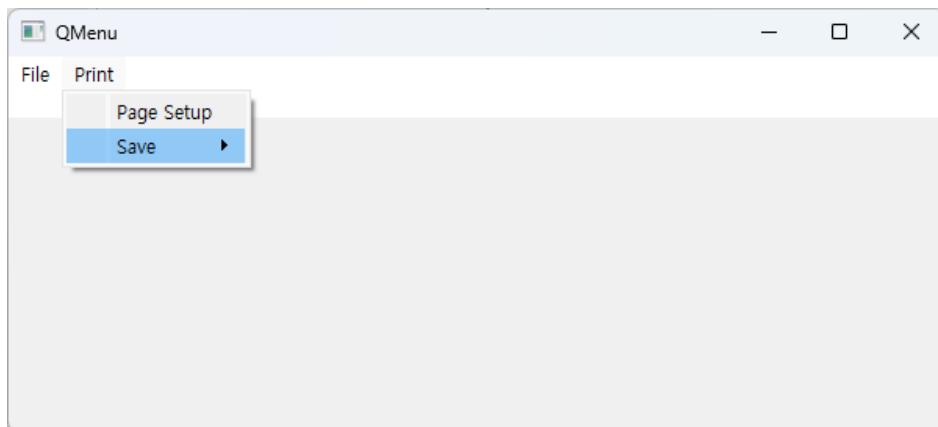
    menuBar->addMenu(menu[0]);
    menuBar->addMenu(menu[2]);
```

```
lbl = new QLabel("",this);
connect(menuBar, SIGNAL(triggered(QAction*)),
        this,      SLOT(trigerMenu(QAction*)));

menuBar->setGeometry(0, 0, 600, 40);
lbl->setGeometry(10, 200, 200, 40);
}

void Widget::trigerMenu(QAction *act)
{
    QString str = QString("Selected Menu : %1").arg(act->text());
    lbl->setText(str);
}
...
```

The QMenu class widgets are the "File", "Save", and "Print" parent menus we created. And the menus created with the QAction class are class widgets for adding submenus.



You can refer to the 12\_QMenu\_QMenuBar directory for example sources.

#### ✓ QProgressBar

You can use the QProgressBar widget as a widget for displaying progress, and the widget can be oriented horizontally or vertically. When placed horizontally, the QProgressBar widget can change the progress direction from left to right and right to left. Conversely, when placed vertically, it can display progress from bottom to top or top to bottom.

```
progress[0] = new QProgressBar(this);
progress[0]->setMinimum(0);
progress[0]->setMaximum(100);
progress[0]->setValue(50);
progress[0]->setOrientation(Qt::Horizontal);

progress[1] = new QProgressBar(this);
progress[1]->setMinimum(0);
progress[1]->setMaximum(100);
progress[1]->setValue(70);
progress[1]->setOrientation(Qt::Horizontal);
progress[1]->setInvertedAppearance(true);

progress[2] = new QProgressBar(this);
progress[2]->setMinimum(0);
progress[2]->setMaximum(100);
progress[2]->setValue(50);
progress[2]->setOrientation(Qt::Vertical);

progress[3] = new QProgressBar(this);
progress[3]->setMinimum(0);
progress[3]->setMaximum(100);
progress[3]->setValue(70);
progress[3]->setOrientation(Qt::Vertical);
progress[3]->setInvertedAppearance(true);

progress[0]->setGeometry(10,30, 300, 30);
progress[1]->setGeometry(10,70, 300, 30);

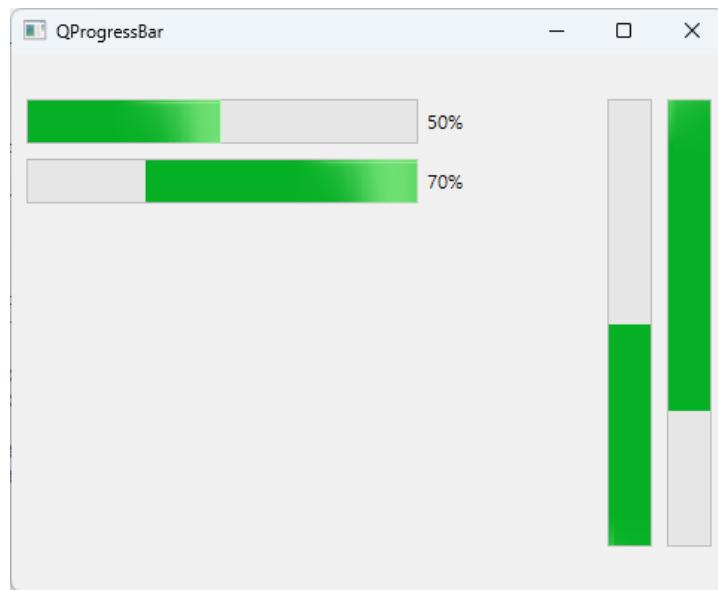
progress[2]->setGeometry(400,30, 30, 300);
progress[3]->setGeometry(440,30, 30, 300);
...
```

The `setMinimum()` and `setMaximum()` functions can be used to set the minimum and maximum values of the `QProgressBar`, and the `setRange()` member function provides the same functionality.

The `setValue()` member function is used to set the current progress value between the minimum and maximum values of the `QProgressBar`, and the `setOrientation()` function can display the `QProgressBar` in either a horizontal or vertical orientation.

The `setInvertedAppearance()` member function is a member function for setting the

direction of advance. If you set true as an argument, you can reverse the default setting.



You can refer to the 13\_QProgressBar directory for example sources.

#### ✓ QRadioButton

The QRadioButton widget provides a GUI for the user to select one of several items. For example, you can select either On (checked) or Off (unchecked).

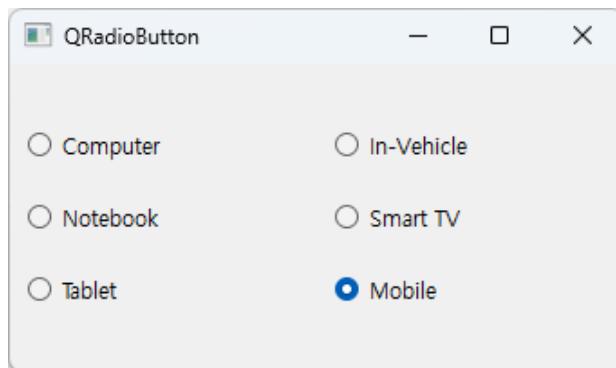
```
QRadioButton *radio1[3];
QRadioButton *radio2[3];

QString str1[3] = {"Computer", "Notebook", "Tablet"};
int ypos = 30;
for(int i = 0 ; i < 3 ; i++)
{
    radio1[i] = new QRadioButton(str1[i], this);
    radio1[i]->setGeometry(10, ypos, 150, 30);
    ypos += 40;
}

QString str2[3] = {"In-Vehicle", "Smart TV", "Mobile"};
ypos = 30;

for(int i = 0 ; i < 3 ; i++)
```

```
{  
    radio2[i] = new QRadioButton(str2[i], this);  
  
    if(i == 2)  
        radio2[i]->setChecked(true);  
  
    radio2[i]->setGeometry(180, ypos, 150, 30);  
    ypos += 40;  
}  
  
QButtonGroup *group1 = new QButtonGroup(this);  
QButtonGroup *group2 = new QButtonGroup(this);  
  
group1-> addButton(radio1[0]);  
group1-> addButton(radio1[1]);  
group1-> addButton(radio1[2]);  
  
group2-> addButton(radio2[0]);  
group2-> addButton(radio2[1]);  
group2-> addButton(radio2[2]);  
...
```



As shown in the example run screen, the left three items in the six items can be separated from the right QRadioButtons by grouping them together. The source of this example can be found in the 14\_QRadioButton directory.

✓ QScrollArea

The QScrollArea widget provides the ability to display all of the GUI by using a scroll bar to navigate to the occluded portion of the GUI if all of the widgets cannot be displayed

Jesus loves you.

on the window in which the GUI is displayed.

For example, if you want to display an image on the screen without shrinking it, and the GUI widget runs out of size, you can create a scroll on the left or bottom that allows you to view the image as you scroll with the mouse.

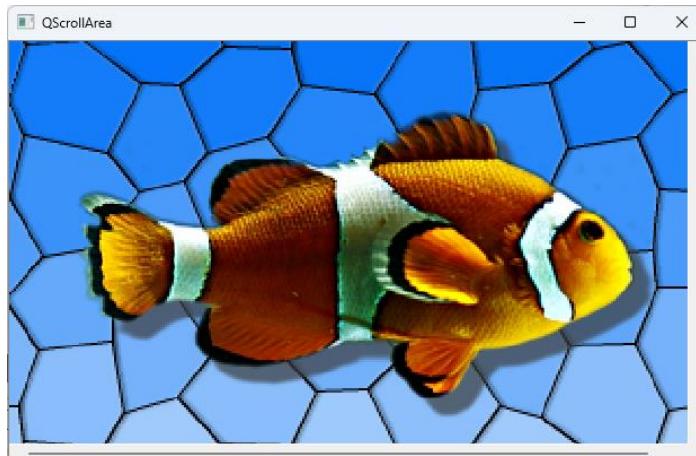
```
QImage image;
QScrollArea *area;

QLabel *lbl = new QLabel(this);
image = QImage(":resources/fish.png");
lbl->setPixmap(QPixmap::fromImage(image));

area = new QScrollArea(this);
area->setWidget(lbl);
area->setBackgroundRole(QPalette::Dark);
area->setGeometry(0, 0, image.width(), image.height());
...
```

The image object of class QImage renders (displays) an image. The rendered image is displayed in an lbl object of class QLabel.

To display the image in the lbl object inside the QScrollArea, we use the setWidget( ) member function provided by QScrollArea. If the image is larger than the QScrollArea widget, the scrollbar is automatically activated.



You can refer to the 15\_QScrollArea directory for example sources.

✓ QScrollBar

The QScrollBar widget class is similar to the appearance of a slider widget. The QScrollBar widget provides the ability to be positioned directly in either vertical or horizontal orientation when scrolling left or right or up or down.

```
...
class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = 0);
    ~Widget();
private:
    QScrollBar *vscrollbar[3];
    QScrollBar *hscrollbar[3];
    QLabel     *lbl[3];
private slots:
    void valueChanged1(int value);
    void valueChanged2(int value);
    void valueChanged3(int value);
};
...
```

As shown in the previous source code, we declare objects of classes QScrollBar and QLabel, and when we drag the scales of the vertical QScrollBar objects, vscrollbar[0] through vscrollbar[2], we display the changed values in the QLabel widget. Also, change the values of vscrollbar[0] through vscrollbar[2] to be the same as the values of their respective hscrollbars.

```
...
Widget::Widget(QWidget *parent) : QWidget(parent)
{
    int xpos = 10;
    int ypos = 50;
    for(int i = 0 ; i < 3 ; i++)
    {
        vscrollbar[i] = new QScrollBar(Qt::Vertical, this);
        vscrollbar[i]->setRange(0, 100);
        vscrollbar[i]->setGeometry(xpos, 30, 20, 200);

        lbl[i] = new QLabel(QString("%1").arg(vscrollbar[i]->value()), this);
    }
}
```

```
lbl[i]->setGeometry(xpos + 2, 220, 30, 30);
xpos += 50;

hscrollbar[i] = new QScrollBar(Qt::Horizontal, this);
hscrollbar[i]->setRange(0, 100);
hscrollbar[i]->setGeometry(150, ypos, 200, 20);
ypos += 30;
}

connect(vscrollbar[0], SIGNAL(valueChanged(int)),
         this,           SLOT(valueChanged1(int)));
connect(vscrollbar[1], SIGNAL(valueChanged(int)),
         this,           SLOT(valueChanged2(int)));
connect(vscrollbar[2], SIGNAL(valueChanged(int)),
         this,           SLOT(valueChanged3(int)));
}

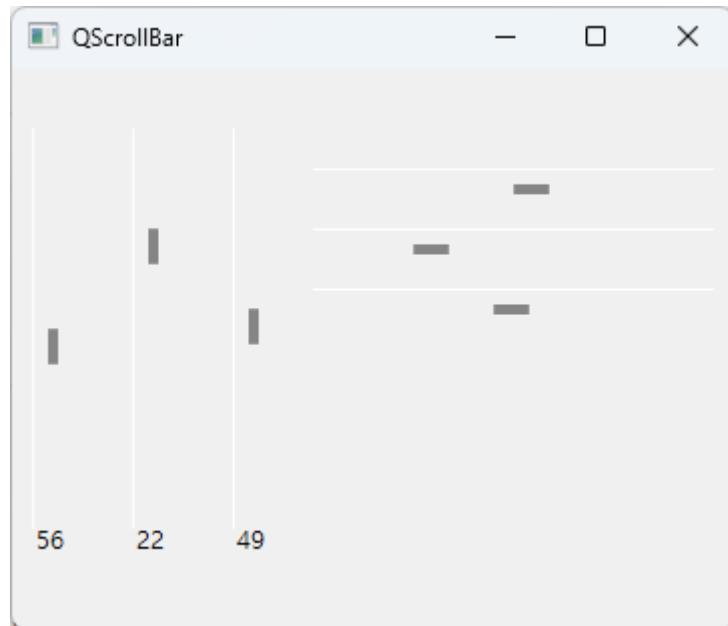
void Widget::valueChanged1(int value)
{
    lbl[0]->setText(QString("%1").arg(value));
    hscrollbar[0]->setValue(vscrollbar[0]->value());
}

void Widget::valueChanged2(int value)
{
    lbl[1]->setText(QString("%1").arg(value));
    hscrollbar[1]->setValue(vscrollbar[1]->value());
}

void Widget::valueChanged3(int value)
{
    lbl[2]->setText(QString("%1").arg(value));
    hscrollbar[2]->setValue(vscrollbar[2]->value());
}
...
```

The connect( ) function calls each Slot function when the value of vscrollbar[0] through vscrollbar[2] changes.

For example, if the value of the vscrollbar[1] first scale is changed by a mouse drag, the valueChanged2( ) Slot function is called. This Slot function changes the value of the QLabel to the value received as the argument of the Slot function. It then changes the value of hscrollbar[1].



You can refer to the 16\_QScrollBar directory for the complete source of the examples.

✓ QSizeGrip

Widgets of class QSizeGrip can resize the widget to fit within a finite-sized window area. For example, in a GUI like MS Windows' Explorer, which shows a tree area on the left and file and directory properties on the right, you can drag the borders of the tree area with the mouse to shrink or expand it. The following is the source code for the `widget.h` file.

```
...
class SubWindow : public QWidget
{
    Q_OBJECT

public:
    SubWindow(QWidget *parent = nullptr) : QWidget(parent, Qt::SubWindow)
    {
        QSizeGrip *sizegrip = new QSizeGrip(this);
        sizegrip->setFixedSize(sizegrip->sizeHint());

        this->setLayout(new QVBoxLayout);
```

```

        layout()->addWidget(new QTextEdit);

        sizegrip->setWindowFlags(Qt::WindowStaysOnTopHint);
        sizegrip->raise();
    }

    QSize sizeHint() const
    {
        return QSize(200, 100);
    }
};

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

};

...

```

The Widget class is declared as the parent class (window widget) and the SubWindow class is the child window widget of the Widget window. In the source code, QVBoxLayout uses a vertically oriented layout to arrange the widgets.

Layout is covered in the next chapter. For now, it is enough to know that it is declared as the layout for the Sub Window.

QTextEdit widgets are widgets provided for editing text, such as Notepad. The QTextEdit widget can be set to the full area size of the SubWindow area so that it can dynamically resize when the SubWindow size changes. The following source code is from main.cpp.

```

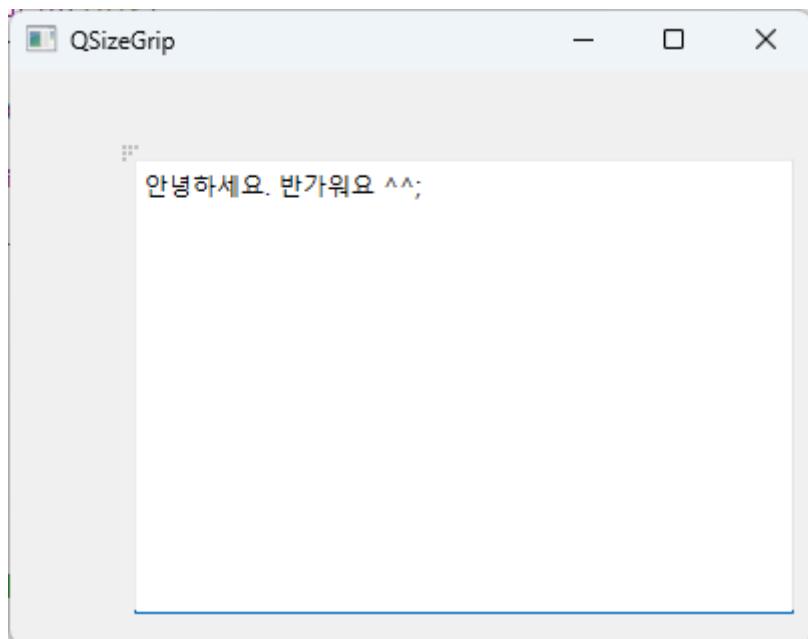
#include <QApplication>
#include "widget.h"

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

```

```
Widget w;  
w.resize(400, 300);  
  
SubWindow subWindow(&w);  
subWindow.move(200, 180);  
  
w.show();  
  
return a.exec();  
}
```

As you can see in the example above, this QTextEdit widget is set to the full area size of the SubWindow area. When you change the size of the SubWindow, the QTextEdit dynamically changes its size.



You can refer to the 17\_QSizeGrip directory for example sources.

#### ✓ QSlider

The QSlider widget provides a GUI to change the value of a setting within a specified range of minimum and maximum values. This widget is similar to the QScrollBar. You can set the minimum and maximum values using the setMinimum( ) and setMaximum( )

member functions. To set the minimum and maximum values, you can use the `setRange()` member function.

```
...
Widget::Widget(QWidget *parent) : QWidget(parent)
{
    int xpos = 20, ypos = 20;
    for(int i = 0 ; i < 6 ; i++) {
        if(i <= 2) {
            slider[i] = new QSlider(Qt::Vertical, this);
            slider[i]->setGeometry(xpos, 20, 30, 80);
            xpos += 30;
        }
        else if(i >= 3) {
            slider[i] = new QSlider(Qt::Horizontal, this);
            slider[i]->setGeometry(130, ypos, 80, 30);
            ypos += 30;
        }
        slider[i]->setRange(0, 100);
        slider[i]->setValue(50);
    }

    xpos = 20;
    for(int i = 0 ; i < 3 ; i++) {
        QString str = QString("%1").arg(slider[i]->value());
        lbl[i] = new QLabel(str, this);
        lbl[i]->setGeometry(xpos+10, 100, 30, 40);
        xpos += 30;
    }
    connect(slider[0], SIGNAL(valueChanged(int)),
            this,      SLOT(valueChanged1(int)));
    connect(slider[1], SIGNAL(valueChanged(int)),
            this,      SLOT(valueChanged2(int)));
    connect(slider[2], SIGNAL(valueChanged(int)),
            this,      SLOT(valueChanged3(int)));
}

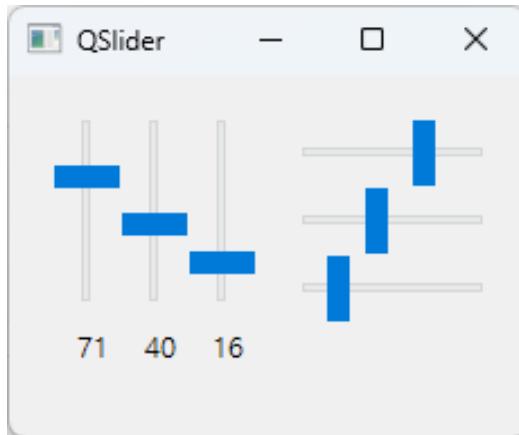
void Widget::valueChanged1(int value)
{
    lbl[0]->setText(QString("%1").arg(value));
    slider[3]->setValue(slider[0]->value());
}
```

```
void Widget::valueChanged2(int value)
{
    lbl[1]->setText(QString("%1").arg(value));
    slider[4]->setValue(slider[1]->value());
}

void Widget::valueChanged3(int value)
{
    lbl[2]->setText(QString("%1").arg(value));
    slider[5]->setValue(slider[2]->value());
}
...
```

The QSlider widget provides a horizontal and a vertical GUI, and the first argument of the QSlider's constructor function is the Qt::Vertical or Qt::Horizontal constant, which can be used to change the orientation of the widget.

The QLabel widget displays the value of the QSlider's scale in a vertical orientation. It fires a signal event when the position of the QSlider's scale changes, changing the text in the QLabel widget to the current value of the QSlider.



예제 소스는 18\_QSlider 디렉토리를 참조하면 된다.

✓ QTabWidget

This widget is useful when you want to place many widgets or when the window is limited in size. This widget provides the ability to dynamically move the page if all tabs

cannot be displayed in the limited size.

To place a widget declared as a QWidget on a tab instance widget, you can use the addTab( ) member function. To place a widget on each tab, you can specify the parent class as a tab widget, just as you would specify the parent class as an argument to place a widget, and the widget will place the widget within that tab.

```
...
Widget::Widget(QWidget *parent) : QWidget(parent)
{
    QTabWidget *tab = new QTabWidget(this);
    QWidget *browser_tab = new QWidget;
    QWidget *users_tab = new QWidget;

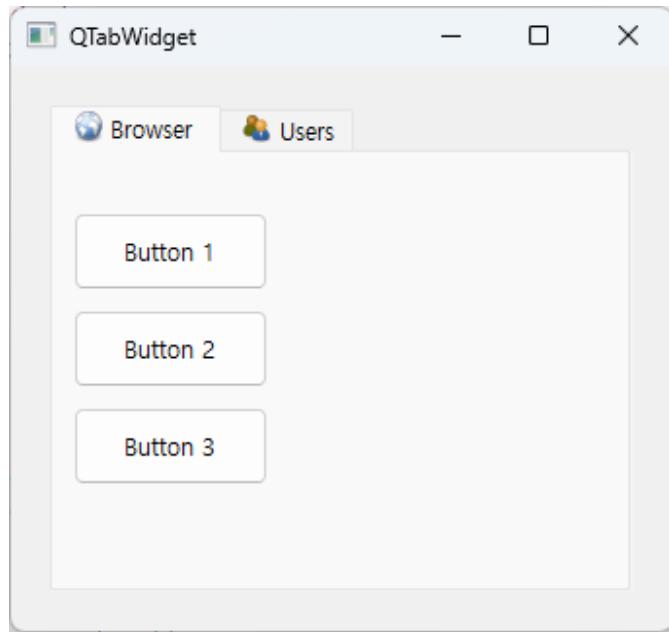
    tab->addTab(browser_tab, QIcon(":resources/browser.png"), "Browser");
    tab->addTab(users_tab, QIcon(":resources/users.png"), "Users");
    tab->setGeometry(20, 20, 300, 250);

    QStringList btn_str[3] = {"Button 1", "Button 2", "Button 3"};
    QPushButton *btn[3];

    int ypos = 30;
    for(int i = 0 ; i < 3 ; i++)
    {
        btn[i] = new QPushButton(btn_str[i], browser_tab);
        btn[i]->setGeometry(10, ypos, 100, 40);
        ypos += 50;
    }

    connect(tab, SIGNAL(currentChanged(int)), this, SLOT(currentTab(int)));
}

void Widget::currentTab(int index)
{
    qDebug("Current Tab : %d", index);
}
...
```



You can refer to the 19\_QTabWidget directory for the full source of the examples.

✓ QToolBar and QAction

The QToolBar widget provides a GUI like a tool menu bar on the window. The QToolBar widget can add menus to the Toolbar using the addAction( ) member function.

```
QToolBar *toolbar = new QToolBar(this);

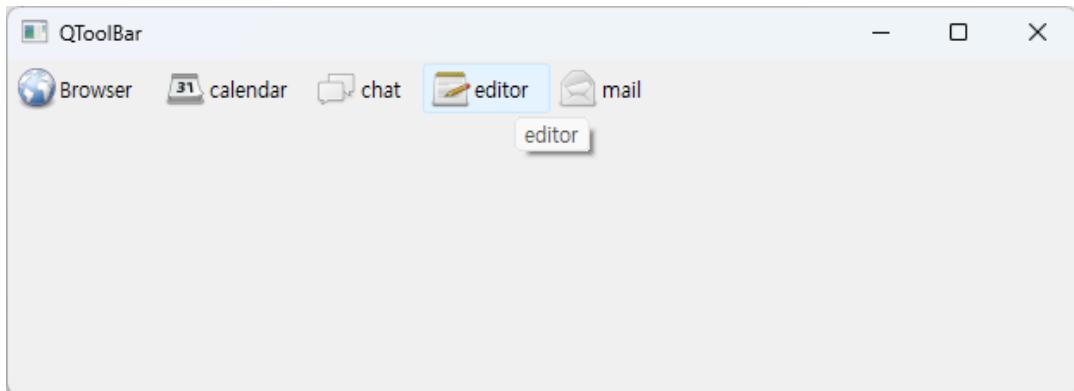
QAction *act[5];
act[0] = new QAction(QIcon(":/resources/browser.png"), "Browser", this);
act[1] = new QAction(QIcon(":/resources/calendar.png"), "calendar", this);
act[2] = new QAction(QIcon(":/resources/chat.png"), "chat", this);
act[3] = new QAction(QIcon(":/resources/editor.png"), "editor", this);
act[4] = new QAction(QIcon(":/resources/mail.png"), "mail", this);

act[0]->setShortcut(Qt::Key_Control | Qt::Key_E);
act[0]->setToolTip("This is a ToolTip.");

toolbar->setToolButtonStyle(Qt::ToolButtonTextBesideIcon);

for(int i = 0 ; i < 5 ; i++)
{
```

```
toolbar->addAction(act[i]);  
}  
...
```



As shown in the previous figure, the following constant values are provided to allow you to display only the icon of the QAction menu button, or to display both the icon and the button name. To set the following constant values, you can use the `setToolButtonStyle( )` member function.

Constant	Value	Description
Qt::ToolButtonIconOnly	0	Icons only
Qt::ToolButtonTextOnly	1	Button names only
Qt::ToolButtonTextBesideIcon	2	Icons inside text
Qt::ToolButtonTextUnderIcon	3	Icons below text

You can refer to the 20\_QToolBar directory for example sources.

#### ✓ QWidget

The widgets we have seen so far have been implemented by inheriting from `QWidget`s. For example, events from the mouse, keyboard, or window can be used by widgets like `QPushButton` because they inherit from `QWidget`.

The `QWidget` class provides the ability to render text, shapes (lines, circles, rectangles, etc.), and images in the widget area using the `paintEvent( )` virtual function. In addition,

the paintEvent( ) function can be called by calling the update( ) member function of the QWidget class. For example, if you use the update() member function inside the Slot function, which is called when a button is clicked, the paintEvent( ) virtual function is called. This redraws the Paint area of the QWidget.

Class QWidget provides a resizeEvent( ) virtual function. This virtual function is called when the size of the QWidget changes. Within the QWidget's area, it provides various virtual functions to handle mouse events, keyboard events, and whether the widget's area is actively focused.

```
#include <QPainter>
#include <QtEvents>
#include <QLineEdit>

class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    QLineEdit *edit;

protected:
    virtual void paintEvent(QPaintEvent *event);
    virtual void resizeEvent(QResizeEvent *event);

    virtual void mousePressEvent(QMouseEvent *event);
    virtual void mouseReleaseEvent(QMouseEvent *event);
    virtual void mouseDoubleClickEvent(QMouseEvent *event);
    virtual void mouseMoveEvent(QMouseEvent *event);

    virtual void keyPressEvent(QKeyEvent *event);
    virtual void keyReleaseEvent(QKeyEvent *event);
    virtual void focusInEvent(QFocusEvent *event);
    virtual void focusOutEvent(QFocusEvent *event);
};
```

The following source code is the widget.cpp source code, which implements each of the virtual functions.

```
#include "widget.h"

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    edit = new QLineEdit("", this);
    edit->setGeometry(120, 20, 100, 30);
}

void Widget::paintEvent(QPaintEvent *event)
{
    Q_UNUSED(event);
    QString img_full_name;

    QPainter painter(this);

    img_full_name = QString(":resources/browser.png");

    QImage image(img_full_name);
    painter.drawPixmap(0, 0,
                       QPixmap::fromImage(image.scaled(100, 100,
                                                       Qt::IgnoreAspectRatio,
                                                       Qt::SmoothTransformation)));

    painter.end();
}

void Widget::resizeEvent(QResizeEvent *event)
{
    Q_UNUSED(event);
    qDebug("[Resize Event call]");
    qDebug("width : %d, height : %d", this->width(), this->height());
}

void Widget::mousePressEvent(QMouseEvent *event)
{
    qDebug() << "[Mouse Press] x, y : " << event->pos();
}

void Widget::mouseReleaseEvent(QMouseEvent *event)
{
    qDebug() << "[Mouse Release] x, y : " << event->pos();
```

```
}

void Widget::mouseDoubleClickEvent(QMouseEvent *event)
{
    qDebug() << "[Mouse Double clicked] x, y : " << event->pos();
}

void Widget::mouseMoveEvent(QMouseEvent *event)
{
    qDebug() << "[Mouse Move] x, y : " << event->pos();
}

void Widget::keyPressEvent(QKeyEvent *event)
{
    qDebug("Key Press Event.");

    switch(event->key())
    {
        case Qt::Key_A :

            if(event->modifiers())
                qDebug("A");
            else
                qDebug("a");

            qDebug("%x", event->key());

            break;

        default:
            break;
    }
}

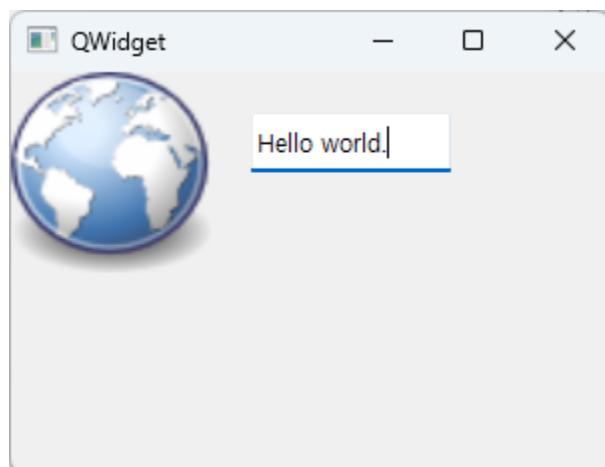
void Widget::keyReleaseEvent(QKeyEvent *event)
{
    Q_UNUSED(event);
    qDebug("Key Release Event.");
}

void Widget::focusInEvent(QFocusEvent *event)
{
```

```
Q_UNUSED(event);
qDebug("focusInEvent Event.");
}

void Widget::focusOutEvent(QFocusEvent *event)
{
    Q_UNUSED(event);
    qDebug("focusOutEvent Event.");
}

Widget::~Widget()
{
}
```



See the 21\_QWidget directory for example sources.

#### ✓ QTabBar

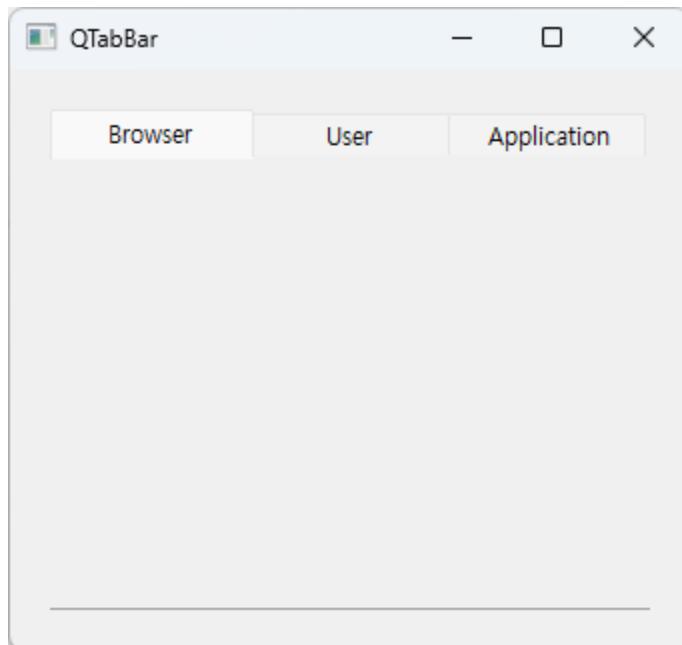
The QTabBar class widget provides a tabbed GUI. It provides similar functionality to the QTabWidget. You can use the setTabIcon( ) function to use icons for the tabs. The text displayed on each tab can be colored to distinguish it. To specify the text for a tab, use the setTabTextColor( ) function.

```
...
Widget::Widget(QWidget *parent) : QWidget(parent)
{
```

```
QTabBar *tab = new QTabBar(this);
tab->addTab("Browser");
tab->addTab("User");
tab->addTab("Application");
tab->setShape(QTabBar::RoundedNorth);
tab->setGeometry(20, 20, 300, 250);

connect(tab, SIGNAL(currentChanged(int)),
        this, SLOT(currentTab(int)));
}

void Widget::currentTab(int index)
{
    qDebug("Current Tab : %d", index);
}
...
```



QTabBar provides the ability to use tabs with rounded corners or a shape with only the left corner of the tab rounded, which can be specified using the setShape( ) function. The following table lists the tab shape values provided as ENUM types.

Constant	Value	Description
QTabBar::RoundedNorth	0	Default appearance
QTabBar::RoundedSouth	1	The toolbar is rounded at the bottom

QTabBar::RoundedWest	2	The left part of the toolbar is rounded
QTabBar::RoundedEast	3	The right part of the toolbar is rounded
QTabBar::TriangularNorth	4	Triangle shape
QTabBar::TriangularSouth	5	Similar to the format used in spreadsheets
QTabBar::TriangularWest	6	The left part of the toolbar is shaped like a triangle
QTabBar::TriangularEast	7	The right part of the toolbar is shaped like a triangle

You can refer to the 22\_QTabBar directory for example sources.

#### ✓ QToolBox

The QToolbox class widget provides a GUI for widget items in the form of a new oriented tabbed column. Text and icons can be used as names for each tab.

```

...
Widget::Widget(QWidget *parent) : QWidget(parent)
{
    box = new QToolBox(this);
    lay = new QVBoxLayout(this);

    but1 = new QPushButton("DataBase - 1",this);
    but2 = new QPushButton("Network - 2",this);
    but3 = new QPushButton("Graphics - 3",this);

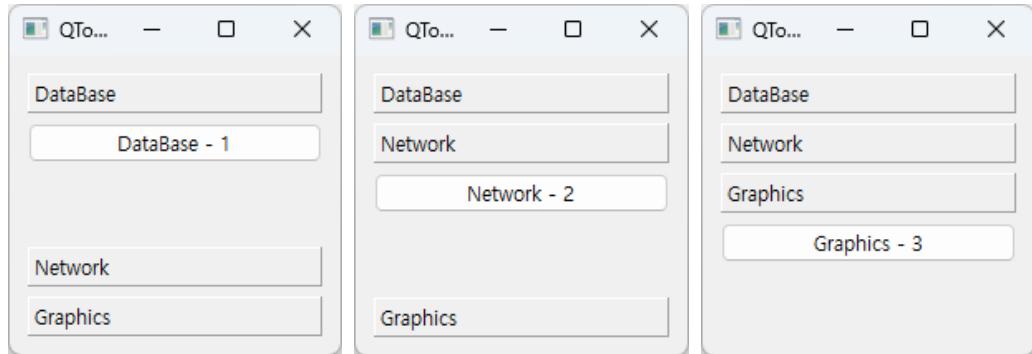
    box->addItem(but1,"DataBase");
    box->addItem(but2,"Network");
    box->addItem(but3,"Graphics");
    lay->addWidget(box);
    setLayout(lay);

    connect(box, SIGNAL(currentChanged(int)), this, SLOT(changedTab(int)));
}

void Widget::changedTab(int index)
{

```

```
    qDebug("current index : %d", index);
}
...
}
```



You can refer to the 23\_QToolBox directory for example sources.

✓ **QToolButton**

The QToolButton widget provides button-like functionality using text or an icon. The icon for a QToolButton can be specified using the QIcon class. The icon can be shown as enabled or disabled depending on its state, and the button is not available in the disabled state.

The setToolButtonStyle( ) member function can be used to change the style, and the setIconSize( ) function can be used to specify the size of the icon.

```
...
Widget::Widget(QWidget *parent) : QWidget(parent)
{
    QToolBar *tool = new QToolBar(this);
    QVBoxLayout *layout = new QVBoxLayout;

    QToolButton *button = new QToolButton;
    button->setIcon(QIcon(":resources/browser.png"));

    QToolButton *button1 = new QToolButton;
    button1->setIcon(QIcon(":resources/calendar.png"));

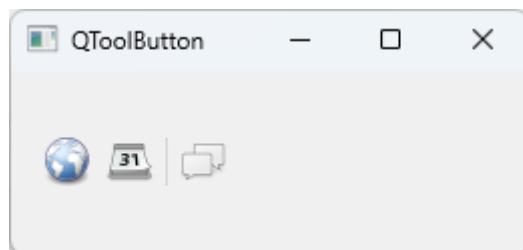
    QToolButton *button2 = new QToolButton;
```

```
button2->setIcon(QIcon(":resources/chat.png"));

tool->addWidget(button);
tool->addWidget(button1);
tool->addSeparator();
tool->addWidget(button2);
layout->addWidget(tool);

this->setLayout(layout);
}

...
```

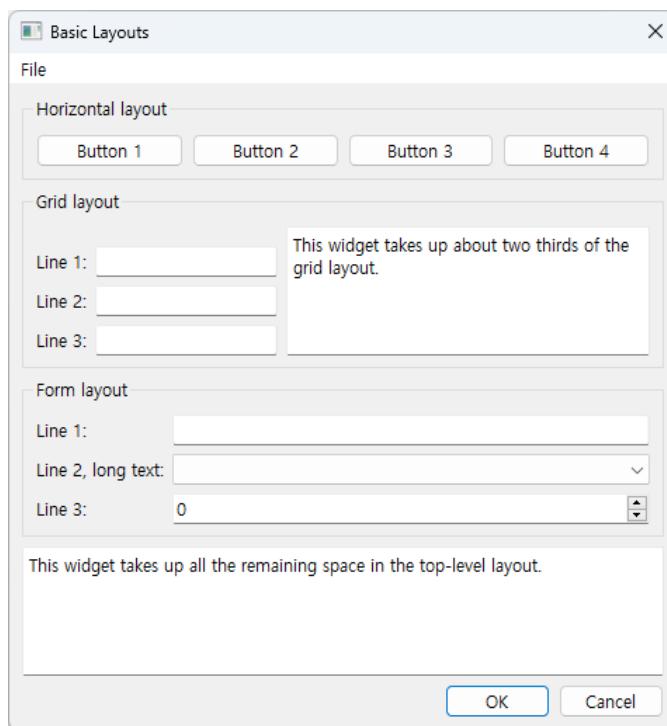


You can refer to the 24\_QToolBox directory for example sources.

## 6. Layout

If you use the `setGeometry()` member function of the `QWidget` class to position a widget on the GUI at specific X and Y coordinates, the widget's position will not change when the window changes size.

However, if you use a layout, the widgets will dynamically change their size on the GUI whenever the window is resized.



As the size of the window changes, the layout ensures that the widgets are aligned in the optimal position to maintain a consistently sized appearance. The following table lists the most commonly used layout classes in Qt.

Class	Description
QHBoxLayout	Arrange widgets in landscape orientation
QVBoxLayout	Positioning Widgets in Vertical Orientation
QGridLayout	Arrange widgets in a grid or checkerboard style

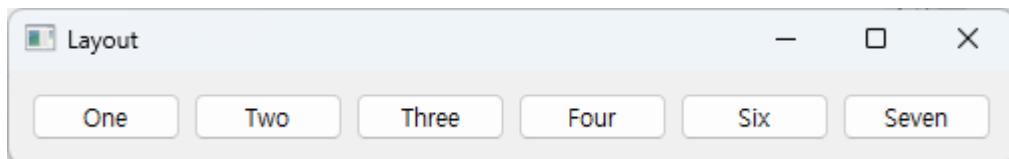
QFormLayout	A format that places widgets in two columns.
-------------	--

✓ QHBoxLayout

QHBoxLayout allows widgets to be placed in a horizontal orientation. A QPushButton widget can be added using the addWidget( ) member function, as shown in the following example.

```
QHBoxLayout *hboxLayout = new QHBoxLayout();
QPushButton *btn[6];

QString btnStr[6] = {"One", "Two", "Three", "Four", "Six", "Seven"};
for(int i = 0 ; i < 6 ; i++)
{
    btn[i] = new QPushButton(btnStr[i]);
    hboxLayout->addWidget(btn[i]);
}
```

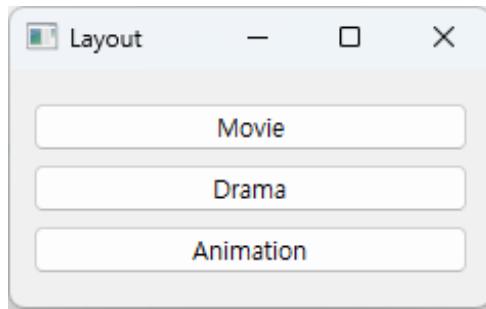


✓ QVBoxLayout

QVBoxLayout can position widgets in a vertical orientation.

```
QVBoxLayout *vboxLayout = new QVBoxLayout();
QPushButton *vbtn[6];

QString vbtnStr[3] = {"Movie", "Drama", "Animation"};
for(int i = 0 ; i < 3 ; i++) {
    vbtn[i] = new QPushButton(vbtnStr[i]);
    vboxLayout->addWidget(vbtn[i]);
}
```



✓ QGridLayout

QGridLayout can arrange widgets in a checkerboard-like style. QGridLayout can merge certain rows or split them into multiple cells so that only one widget can be placed.

```
QGridLayout *gridLayout = new QGridLayout();
QPushButton *gbtn[5];

for(int i = 0 ; i < 5 ; i++)
    gbtn[i] = new QPushButton(btnStr[i]);

gridLayout->addWidget(gbtn[0], 0, 0);
gridLayout->addWidget(gbtn[1], 0, 1);
gridLayout->addWidget(gbtn[2], 1, 0, 1, 2);
gridLayout->addWidget(gbtn[3], 2, 0);
gridLayout->addWidget(gbtn[4], 2, 1);
```



✓ Using nested layouts

You can place layouts within layouts. The following example shows a nested structure.

```
...
```

```
Widget::Widget(QWidget *parent) : QWidget(parent)
{
    QHBoxLayout *hboxLayout = new QHBoxLayout();
    QPushButton *btn[6];

    QStringList btnStr[6] = { "One", "Two", "Three", "Four", "Six", "Seven" };
    for(int i = 0 ; i < 6 ; i++) {
        btn[i] = new QPushButton(btnStr[i]);
        hboxLayout->addWidget(btn[i]);
    }

    QVBoxLayout *vboxLayout = new QVBoxLayout();
    QPushButton *vbtn[6];
    QStringList vbtnStr[3] = {"Movie", "Drama", "Animation"};

    for(int i = 0 ; i < 3 ; i++) {
        vbtn[i] = new QPushButton(vbtnStr[i]);
        vboxLayout->addWidget(vbtn[i]);
    }

    QGridLayout *gridLayout = new QGridLayout();
    QPushButton *gbtn[5];

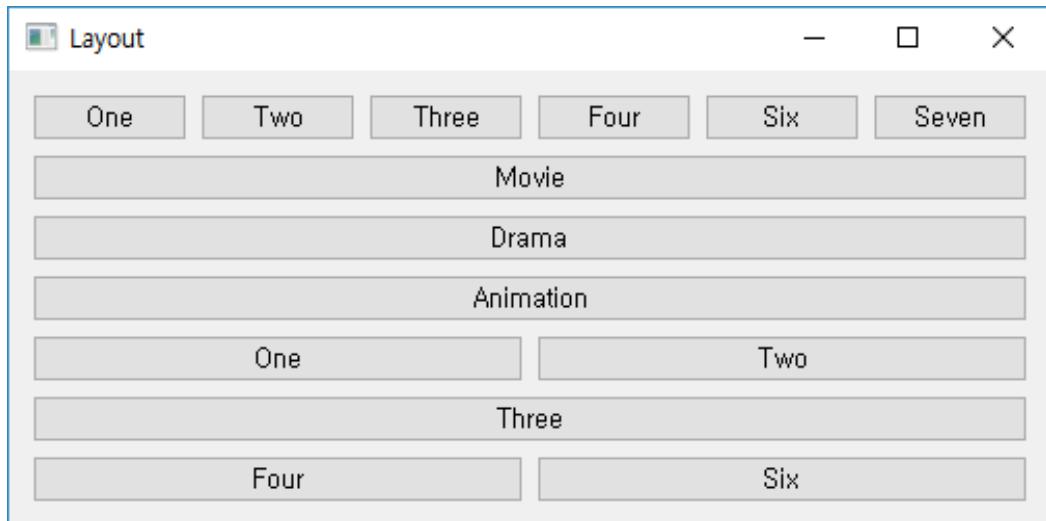
    for(int i = 0 ; i < 5 ; i++)
        gbtn[i] = new QPushButton(btnStr[i]);

    gridLayout->addWidget(gbtn[0], 0, 0);
    gridLayout->addWidget(gbtn[1], 0, 1);
    gridLayout->addWidget(gbtn[2], 1, 0, 1, 2);
    gridLayout->addWidget(gbtn[3], 2, 0);
    gridLayout->addWidget(gbtn[4], 2, 1);

    QVBoxLayout *defaultLayout = new QVBoxLayout();
    defaultLayout->addLayout(hboxLayout);
    defaultLayout->addLayout(vboxLayout);
    defaultLayout->addLayout(gridLayout);

    setLayout(defaultLayout);
}
```

Jesus loves you.



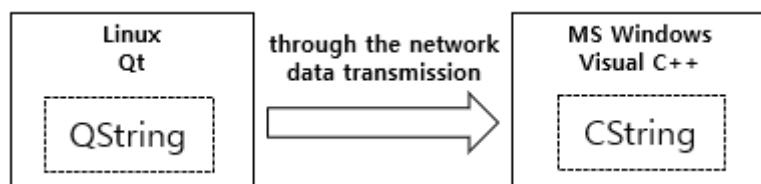
You can refer to the 00\_Layout directory for example sources.

## 7. Data types and classes provided by Qt

Qt provides a variety of datatypes for developer convenience. For example, it supports regular expressions to find specific patterns in a string, such as `QString`, or to add or delete specific characters to a string.

Qt also supports data types to solve problems caused by changes in data types when exchanging data between different systems. For example, in an application developed with Qt on the Ubuntu Linux operating system, you can send the string "Hello World" as a `QString`, which is provided by Qt.

If the recipient of this string is an application developed with Visual C++ on the MS Windows operating system and stores the string "Hello World" as a `CString` string, two different data types are stored.



As shown in the figure above, to solve the problem of cross-platform data, you can convert a `QString` to a `QByteArray` and send it as a `Char`, which can be used on the receiving end to solve the cross-platform data problem. Qt also supports void-type data types, such as the `QVariant` data type. The following table lists the most commonly used data types in Qt.

Type	Size	Description
<code>bool</code>	8 bit	true / false
<code>qint8</code>	8 bit	<code>signed char</code>
<code>qint16</code>	16 bit	<code>signed short</code>
<code>qint32</code>	32 bit	<code>signed int</code>
<code>qint64</code>	64 bit	<code>long long int</code>
<code>quint8</code>	8 bit	<code>unsigned char</code>

quint16	16 bit	unsigned short
quint32	32 bit	unsigned int
quint64	64 bit	unsigned long long int
float	32 bit	Floating point number using IEEE 754 format
double	64 bit	Floating point number using IEEE 754 format
const char*	32 bit	Pointer to a string constant, excluding zeros at the end

Qt provides generic and template functions for determining/comparing values of data types. For example, it provides qAbs( ) to find absolute values and qBound( ) to find values between a maximum and minimum. It provides qMin( ) to find the minimum value and qMax( ) to find the maximum value.

✓ Using the absolute value

```
int absoluteValue;
int myValue = -4;

absoluteValue = qAbs(myValue); // absoluteValue == 4
```

✓ Find a value between the minimum and maximum

```
int myValue = 10;
int minValue = 2;
int maxValue = 6;

int boundedValue = qBound(minValue, myValue, maxValue);
// boundedValue == 6
```

✓ Functions for comparing decimal points

```
// 0.0과 비교
qFuzzyCompare(0.0, 1.0e-200); // return false
qFuzzyCompare(1 + 0.0, 1 + 1.0e-200); // return true
```

✓ Find the maximum value

```
int myValue = 6;
int yourValue = 4;

int maxValue = qMax(myValue, yourValue);
int minValue = qMin(myValue, yourValue);
```

✓ Rounding int types

```
qreal valueA = 2.3;
qreal valueB = 2.7;

int roundedValueA = qRound(valueA); // roundedValueA = 2
int roundedValueB = qRound(valueB); // roundedValueB = 3
```

✓ Rounding 64-bit int types

```
qreal valueA = 42949672960.3;
qreal valueB = 42949672960.7;

int roundedA = qRound(valueA); // roundedA = 42949672960
int roundedB = qRound(valueB); // roundedB = 42949672961
```

✓ String Data Type Class

In addition to classes that deal with simple strings, Qt provides a number of classes that support Unicode Standard Version (Unicode 4.0) characters in the form of data streams and multi-byte characters.

클래스	설명
QByteArray	바이트 단위의 배열을 지원
QByteArrayMatcher	Using the index of a byte-by-byte array implemented as a QByteArray to find if a matching string exists
QChar	Support for 16-bit Unicode characters
QLatin1Char QLatin1String	Provided to support US-ASCII/Latin-1 encoded strings
QLocale	Classes that convert number or character representations to

	match the representation of different languages.
QString	Classes that support Unicode string characters
QStringList	Set class for string lists
QStringMatcher	Classes provided for finding matching strings on strings
QStringRef	Substring wrapping classes like size( ), position( ), and toString( )
QTextStream	Provides STREAM function for text-based WRITE/READ
QDataStream	Provides STREAM functionality for binary-based WRITE/READ

#### ✓ QByteArray

Class QByteArray provides an array of bytes (8-bit). Class QByteArray provides the append( ), prepend( ), insert( ), replace( ), and remove( ) member functions to handle arrays.

```

QByteArray x("Q");

x.prepend("I love");      // x == I love Q
x.append("t -^^*");      // x == I love Qt -^^*
x.replace(13, 1, "*");   // x == I love Qt *^^*

QByteArray x("I love Qt -^^*");
x.remove(13, 4); // x == I love Qt

```

#### ✓ QByteArrayMatcher

This class is provided for finding matching byte array patterns in a byte array.

```

// 전체 QByteArray
QByteArray x(" hello Qt, nice to meet you.");
QByteArray y("Qt"); // The string you want to find in X

QByteArrayMatcher matcher(y);

// Store where the string starts in the index variable
int index = matcher.indexIn(x, 0);

qDebug( "index : %d", index);

```

```
qDebug( "QByte : %c%c", x.at(index), x.at(index+1));
```

✓ QChar

Character class to support 16-bit Unicode.

```
QLabel *lbl = new QLabel("", this);
QString str = "Hello Qt";
QChar *data = str.data();
QString str_display;

while(!data->isNull())
{
    str_display.append(data->unicode());
    ++data;
}

lbl->setText(str_display); // Hello Qt
```

✓ QLatin1String

This class is provided to support US-ASCII/Latin-1 encoded strings.

```
QLatin1String latin("Qt");
QString str = QString("Qt");

if(str == latin)
    qDebug("Equal.");
else
    qDebug("Not equal.");

bool is_equal = latin.operator==(str);

if(is_equal)
    qDebug("Equal.");
else
    qDebug("Not equal.");
```

✓ QLocale

This class is used to convert to different language character sets.

```
QLocale egyptian(QLocale::Arabic, QLocale::Egypt);
QString s1 = egyptian.toString(1.571429E+07, 'e');
QString s2 = egyptian.toInt(10);

double d = egyptian.toDouble(s1);
int i = egyptian.toInt(s2);
```

✓ **QString**

Class QString supports Unicode strings and provides the ability to store 16-bit QChars.

```
QString str = "Hello";
```

QString provides the ability to replace string constants, such as const char \*, using the fromUtf8() function.

```
static const QChar data[4] = {0x0055, 0x006e, 0x10e3, 0x03a3 };
QString str(data, 4);
```

Provides the ability to store a QChar at a specific location in a stored string.

```
QString str;
str.resize(4);
str[0] = QChar('U');
str[1] = QChar('n');
str[2] = QChar(0x10e3);
str[3] = QChar(0x03a3);
```

To compare strings, QString can use the if statement to make the following comparisons

```
QString str;

if ( str == "auto" || str == "extern" || str == "static" || str == "register" )
{
    // ...
}
```

If you want to know the position of a specific string you are looking for, you can use the indexOf() function to find the position of the string you are looking for.

```
QString str = "We must be <b>bold</b>, very <b>bold</b>";
int j = 0;

while ((j = str.indexOf("<b>", j)) != -1) {
```

```
qDebug() << "Found <b> tag at index position" << j;
++j;
}
```

✓ **QStringList**

QString provides the ability to manage strings stored in QString as an array like QList, and provides member functions such as append(), prepend(), and insert().

```
QStringList strList;
strList << "Monday" << "Tuesday" << "Wednesday";

QString str;
str = strList.join(",");
// str == " Monday, Tuesday, Wednesday"
```

✓ **QTextStream**

The QTextStream class provides STREAM processing for handling large amounts of text data. The STREAM method allows you to access large amounts of data efficiently and quickly to read or write data. The following example uses QTextStream to read data from a file using the QFile class.

```
QFile file("in.txt");
if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
    return;

QTextStream in(&file);
while (!in.atEnd()) {
    QString line = in.readLine();
    ...
}
```

✓ **Data Classes**

In addition to the basic data types provided by Qt, Qt provides a number of classes for flexible data manipulation. For example, QBitArray for handling data in bits, QByteArray for handling data in bytes in an array, and so on provide convenient classes for data manipulation. In this section, we'll look at some of the classes that are useful for

manipulating data.

클래스	설명
QBitArray	Bit Array to provide bitwise operations (AND, OR, XOR, NOT)
QMargins	A class to handle each Margin in the left, top, right, and bottom rectangles
QPoint	A class you provide to handle X, Y, and Z values
QQuaternion	A class for handling Quaternions composed of vectors and scalars
QRect	Left (qint32), top (qint32), right (qint32), bottom (qint32) of rectangle
QRegExp	Classes for handling regular expressions
QRegion	Used to define clipboard regions in Painter
QSize	A class to store width and height
QVariant	A class of union type that can be stored as void types
QVector2D	A class for representing Vectors or vertices in two-dimensional space
QVector3D	A class that can take three coordinates: x, y, z
QVector4D	Used to represent a vector or vertex in four-dimensional space.

#### ✓ QBitArray

QBitArray 클래스는 Bit 단위를 데이터로 관리할 수 있는 기능을 제공한다. AND, OR, XOR 그리고 NOT 연산을 통해 Bit 연산 기능을 제공한다.

```
QBitArray ba(200);

QBitArray ba;
ba.resize(3);
ba[0] = true;
ba[1] = false;
ba[2] = true;

QBitArray x(5);
x.setBit(3, true); // x: [ 0, 0, 0, 1, 0 ]
QBitArray y(5);
y.setBit(4, true); // y: [ 0, 0, 0, 0, 1 ]
```

```
x |= y; // x: [ 0, 0, 0, 1, 1 ]
```

✓ QMargins

The QMargins class can store Left, Top, Right, and Bottom, and the values can be saved or modified using the setLeft( ), setRight( ), setTop( ), and setBottom( ) member functions.

```
QMargins margin;  
  
margin.setLeft(10);  
margin.setTop(12);  
margin.setRight(14);  
margin.setBottom(16);  
  
qDebug() << "QMargins Value : " << margin;  
// QMargins Value : QMargins(10, 12, 14, 16)
```

✓ QPoint

The QPoint class provides functionality that is used to display coordinate points.

```
QPoint p;  
p.setX(p.x() + 1);  
p += QPoint(1, 0);  
p.rx()++;
```

QPoint는 operator를 통해 아래와 같이 사용할 수 있다.

```
QPoint p( 3, 7);  
QPoint q(-1, 4);  
  
p += q; // p becomes (2, 11)
```

✓ QQuaternion

QQuaternion can use quaternions consisting of vectors and scalars. Quaternions provide the ability to manage data used in 3D, including scalars, X, Y, and Z coordinates in 3D space, and angles such as rotations.

```
QQuaternion yRot;
```

```
yRot = QQuaternion::fromAxisAndAngle(0.0f, 1.0f, 0.0f, horiAng * radiDeg);
```

✓ QRect 와 QRectF

Class QRect can be used to store the coordinate values of a rectangle using an integer (Integer) and a real number (Float) for QRectF. You can store the X, Y, Width, and Height values of a rectangle.

```
QRect r1(100, 200, 11, 16);
QRect r2(QPoint(100, 200), QSize(11, 16));

QRectF rf1(100.0, 200.1, 11.2, 16.3);
QRectF rf2(QPointF(100.0, 200.1), QSizeF(11.2, 16.3));
```

✓ QRegExp

The QRegExp class provides regular expression functionality.

```
QRegExp rx("^\\d\\d?"); // For integer values from 0 to 99

rx.indexIn("123");      // return -1
rx.indexIn("-6");       // return -1
rx.indexIn("6");        // return 0
```

✓ QRegion

QRegion is used in conjunction with the QPainter::setClipRegion( ) function. In Painter, you can use it to clip a specific region inside a rectangle to use as the clipboard as a rectangle, as follows

```
void MyWidget::paintEvent(QPaintEvent *)
{
    QRegion r1(QRect(100, 100, 200, 80), QRegion::Ellipse);
    QRegion r2(QRect(100, 120, 90, 30));
    QRegion r3 = r1.intersected(r2);

    QPainter painter(this);
    painter.setClipRegion(r3);
    ...
}
```

✓ QSize

The QSize class is primarily used to store width and height using int values.

```
QSize size(100, 10);
size.setHeight() += 5;
// size (100,15)
```

✓ QVariant

Class QVariant provides the ability to specify the data type of an undefined type. For example, you can specify something like void \*.

```
QDataStream out(...);
QVariant v(123);           // Storing an int type
int x = v.toInt();         // x = 123

out << v;
v = QVariant("hello");    // Storing a string of type QByteArray
v = QVariant(tr("hello")); // Storing strings of type QString

int y = v.toInt();
QString s = v.toString();
out << v;
...

QDataStream in(...);
in >> v;
int z = v.toInt();
qDebug("Type is %s", v.typeName());

v = v.toInt() + 100;
v = QVariant(QStringList());
```

✓ QVector2D, QVector3D and QVector4D

QVector2D is used to deal with Vectors and Vertices in 2D coordinates. QVector3D is a class with Z appended to X and Y. And the QVector4D class provides functionality with W added.

```
...
void GeometryEngine::initCubeGeometry()
{
    VertexData vertices[] = {
        // Vertex data for face 0
        {{QVector3D(-1.0f, -1.0f, 1.0f), QVector2D(0.0f, 0.0f)},
         {QVector3D( 1.0f, -1.0f, 1.0f), QVector2D(0.33f, 0.0f)},
         {QVector3D(-1.0f, 1.0f, 1.0f), QVector2D(0.0f, 0.5f)},
         {QVector3D( 1.0f, 1.0f, 1.0f), QVector2D(0.33f, 0.5f)},
        // Vertex data for face 1
        {QVector3D( 1.0f, -1.0f, 1.0f), QVector2D( 0.0f, 0.5f)},
        {QVector3D( 1.0f, -1.0f, -1.0f), QVector2D(0.33f, 0.5f)},
        {QVector3D( 1.0f, 1.0f, 1.0f), QVector2D(0.0f, 1.0f)},
        {QVector3D( 1.0f, 1.0f, -1.0f), QVector2D(0.33f, 1.0f)},
    ...
}
```

## 8. Container Classes

The Container class is used to store certain types of data in the form of sets or arrays. For example, if you need to store several items as `QString`, you can use a Container class like `QList<QString>`.

Container classes are easier to use and safer than the containers provided by STL. They are also lightweight. Therefore, the Containers provided by Qt can be used as a replacement for the Containers provided by the STL in C++.

<표> Qt에서 제공하는 Container 클래스

클래스	설명
<code>QHash&lt;Key, T&gt;</code>	Template class that provides a hash table-based Dictionary
<code>QMap&lt;Key, T&gt;</code>	Binary Search Tree-based template classes
<code>QPair&lt;T1, T2&gt;</code>	Classes that handle item data that exists as pairs.
<code>QList&lt;T&gt;</code>	Template classes provided for handling values in the form of a List
<code>QLinkedList&lt;T&gt;</code>	Template Classes for Providing Linked Lists
<code> QVector&lt;T&gt;</code>	Classes provided to handle dynamic <code>QVector</code> arrays In Qt 6, it has been merged into <code>QList</code> .
<code>QStack&lt;T&gt;</code>	Provided for using Stack-based <code>push()</code> , <code>pop()</code> , etc.
<code>QQueue&lt;T&gt;</code>	Provided for manipulating data in FIFO structures
<code>QSet&lt;T&gt;</code>	Classes provided for quick hash-based searches
<code> QMap&lt;Key, T&gt;</code>	Provides a Dictionary by connecting data mapped by key values into a single combination.
<code> QMultiMap&lt;Key, T&gt;</code>	This class inherits from <code>QMap</code> and can have multiple mapping values.
<code> QMultiHash&lt;Key, T&gt;</code>	This class inherits from <code>QHash</code> and allows you to use Hash with multiple mapping values.

✓ **QHash<Key, T>**

Class QHash provides a hash table-based Dictionary. The data is stored as a pair of key and value. It provides the ability to quickly search for the data you want to find by key and value. QHash provides very similar functionality to QMap, but its internal algorithm is faster than QMap.

```
QHash<QString, int> hash;  
  
hash["one"] = 1;  
hash["three"] = 3;  
hash["seven"] = 7;
```

You can use the insert( ) function as a way to store a Key, Value pair in a QHash. And to get the value of Value, you can use the value( ) member function.

```
hash.insert("twelve", 12);  
int num1 = hash["thirteen"];  
int num2 = hash.value("thirteen");
```

✓ **QMap<Key, T>**

The usage of QMap is similar to QHash. The following is an example source code that uses a QString as Key and an int as Value.

```
QMap<QString, int> map;  
  
map["one"] = 1;  
map["three"] = 3;  
map["seven"] = 7;  
  
map.insert("twelve", 12);  
  
int num1 = map["thirteen"];  
int num2 = map.value("thirteen");
```

The following example shows how you can use the contains( ) function as a way to get a Value value from a Key value.

```
int timeout = 30;
```

```
if (map.contains("TIMEOUT"))
    timeout = map.value("TIMEOUT");
```

✓ `QPair<T1, T2>`

The `QPair` class can store two items as a pair that are organized into a single item. As shown in the example below, the `QPair` class can be used by declaring it as follows.

```
QPair<QString, double> pair;

pair.first = "pi";
pair.second = 3.14159265358979323846;
```

✓ `QList< T>`

`QList<T>` provides fast index-based access and is also very fast to delete stored data. `QList` is an index-based class, which is more convenient to use than the Iterator-based basis of `QLinkedList` and faster than  `QVector` in terms of memory allocation when storing data.

```
QList<int> integerList;
QList<QDate> dateList;
QList<QString> list = { "one", "two", "three" };
```

`QList` can use the return value via the comparison operator as shown in the example below.

```
if (list[0] == "Bob")
    list[0] = "Robert";
```

`QList` makes it easy to retrieve a stored location in the list using the `at( )` function.

```
for (int i = 0 ; i < list.size() ; ++i)
{
    if (list.at(i) == "Jane")
        cout << "Found Jane at position " << i << endl;
}
```

✓ `QLinkedList<T>`

The QLinkedList class is an iterator-based class that provides the ability to store and delete items in a list. It can be declared as shown in the following example.

```
QLinkedList<int> integerList;  
QLinkedList<QTime> timeList;
```

To add an item to the list, you can use the operator as shown in the following example.

```
QLinkedList<QString> list;  
  
list << one << two << three ;  
// list: [ one , two , three ]
```

✓ **QStack<T>**

QStack is a class for providing Stack algorithms. It is a structure in which data inserted later comes first (LIFO).

```
QStack<int> stack;  
stack.push(1);  
stack.push(2);  
stack.push(3);  
  
while (!stack.isEmpty())  
    cout << stack.pop() << endl;
```

✓ **QQueue<T>**

QQueue is a class for providing a Queue algorithm, which is a structure (FIFO) where the inserted data comes out first.

```
queue.enqueue(1);  
queue.enqueue(2);  
queue.enqueue(3);  
  
while (!queue.isEmpty())  
    cout << queue.dequeue() << endl;
```

✓ **QSet<T>**

QSet is very fast when searching. The internal structure of a QSet is implemented as a

QHash. The following example shows how to declare and use it

```
QSet<QString> set;  
  
set.insert("one");  
set.insert("three");  
set.insert("seven");  
  
set << "twelve" << "fifteen" << "nineteen";
```

✓ **QMultiMap<Key, T>**

This class inherits from the QMap class and provides the ability to use multiple mapping values and is an extension to QMap.

```
QMultiMap<QString, int> map1, map2, map3;  
map1.insert("plenty", 2000); // map1.size() == 2  
map2.insert("plenty", 5000); // map2.size() == 1  
map3 = map1 + map2; // map3.size() == 3
```

✓ **QMultiHash<Key, T>**

The QMultiHash class is suitable for storing multiple hash values. QHash does not allow multiple identical values, but QMultiHash is characterized by allowing multiple identical values.

```
QMultiHash<QString, int> hash1, hash2, hash3;  
hash1.insert("plenty", 100);  
hash1.insert("plenty", 2000); // hash1.size() == 2  
hash2.insert("plenty", 5000); // hash2.size() == 1  
  
hash3 = hash1 + hash2; // hash3.size() == 3
```

## 9. Signal and Slot

Qt uses signals and slots as mechanisms for handling events. For example, the act of clicking a button is called a Signal in Qt. The function that is called when a signal is raised is called a Slot function. When an event called a signal occurs, the slot function associated with the signal is called.

A signal is an event that occurs when something happens. All event handling in Qt uses the mechanisms of signals and slots.

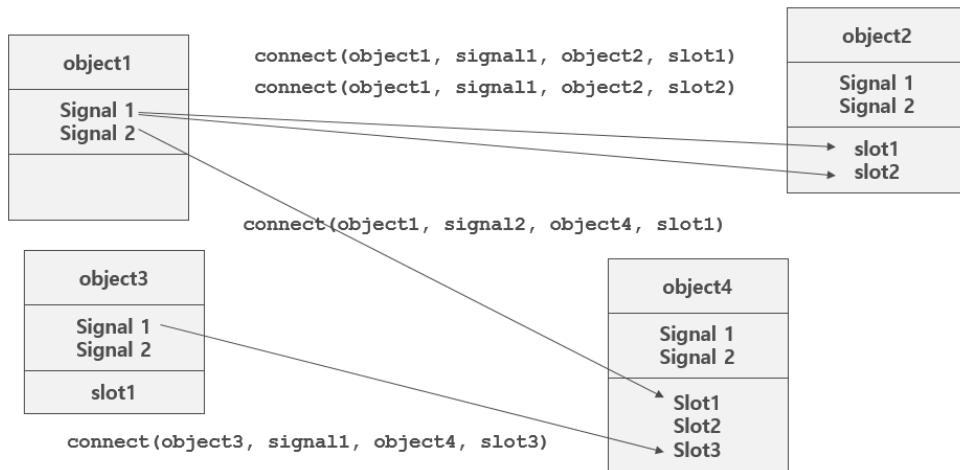
For example, suppose a user named A sends a message to a user named B in a chat program in Qt. From B's perspective, the act of receiving a message from A is defined as a Signal.

B would then call the Slot function associated with the signal that received the message. Qt uses signals and slots for all event handling. In addition to the GUI and network modules just described as examples, all APIs provided by Qt use the event mechanisms of signals and slots. Signals and slots simplify the program source code, which can shorten development time and simplify complex program structures.

Suppose you want to develop a network chat program without using Qt's signal slots. You need to develop a program with multiple threads. This can be complicated because you need to use multiple threads, for example, one to handle messages from a particular user, one to handle whispers, and one to notify you when a new user is registered. However, Qt's use of signals and slots makes it simple to implement a chat program without using threads.

Every GUI widget that Qt provides has a number of predefined signals. For example, the QPushButton has various signals defined, such as click, double click, mouse over, etc.

You can think of signals and slots as a pipeline. A single signal can call multiple slot functions. And multiple signals can call a single slot.



<그림> Signal 과 Slot 의 Architecture

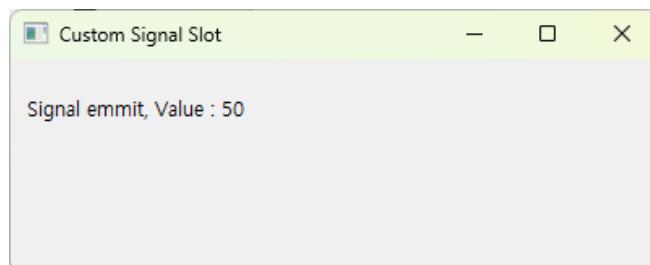
The function for connecting Signal and Slot functions is the `connect()` function of class `QObject`, which allows you to connect Signal and Slot. The first argument to the `connect()` member function is the object (instance of the class) on which the event occurred, and the second argument is the signal (event) of the object.

For example, if you have a button named A, the first argument is the object name of the button named A, and the second argument can be the signal of a click or double-click on the button. So you would specify the click event as the second argument. The third argument specifies the name of the object that contains the signal and the slot function to call. The fourth argument specifies the slot function to call when the signal is fired.

So far, we've covered the concepts of Signal and Slot. Now let's write an example program using them.

#### ✓ Signal and Slot example

This example source code outputs the text of a label placed on a window when a signal is fired.



In this example, there are two classes, a class called SignalSlot and a class called Widget. The following example source code is for a class called SignalSlot.

```
...
class SignalSlot : public QObject
{
    Q_OBJECT

public:
    void setValue(int val) {
        emit valueChanged(val);
    }

signals:
    void valueChanged(int newValue);

private:
    int m_value;
};
```

As you can see in the example above, there is a function called valueChanged( ) at the bottom of the signals: keyword. This is the signal function.

Signal functions do not have an implementation part and only need to be defined in the header, as shown in the source code. The valueChanged( ) signal function specifies an int argument. This argument can be used to pass a value to the Slot function as an argument when the signal is raised. We've used a single value here. You can use no argument or multiple arguments as needed.

In the example source code above, the public keyword of the class SignalSlot is the setValue( ) member function. When this member function is called, you can see that the source code uses the keyword emit within the function. The emit keyword fires a signal event. The following example source code is the header part of the Widget class.

```
...
class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = nullptr);
    ~Widget();
```

```
private:  
    QLabel *lbl;  
  
public slots:  
    void setValue(int val);  
};
```

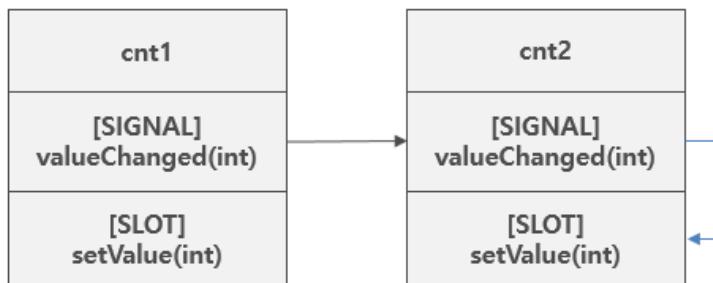
At the bottom of the Widget class, you'll notice that we've used the slots keyword with the public keyword. The slots keyword can be used with either the private or public keyword. The member function specified using the slots keyword defines the Slot function. The following example source code is from widget.cpp.

```
#include "widget.h"  
  
Widget::Widget(QWidget *parent) : QWidget(parent)  
{  
    lbl = new QLabel("", this);  
    lbl->setGeometry(10, 10, 250, 40);  
  
    SignalSlot myObject;  
  
    // New Style  
    connect(&myObject, &SignalSlot::valueChanged, this, &Widget::setValue);  
  
    /* Old Style  
    connect(&myObject, SIGNAL(valueChanged(int)), this, SLOT(setValue(int)));  
    */  
  
    myObject.setValue(50);  
}  
  
void Widget::setValue(int val)  
{  
    QString text = QString("Signal emmit, Value:%1").arg(val);  
    lbl->setText(labelText);  
}
```

You can connect a Signal and a Slot using the connect( ) member function. There are two ways to connect a Signal and a Slot. The method annotated as New Style is the method for connecting Signals and Slots that was added in Qt 5.5 and later versions. Both the New Syntax and the Old Style can be used. In Qt 5.5 and earlier, only the Old

Style method is available. The New Style style does not support multiple arguments. In addition, the New Style allows regular member functions, not just Slot functions, to be used as the fourth argument in the connect( ) function, just like Slot functions.

As you can see, the Signal and Slot functions don't have to be implemented in different classes. A Signal that exists in the same class can call a Slot function. And a Signal can call a Slot function in addition to a Signal, as shown in the following figure.



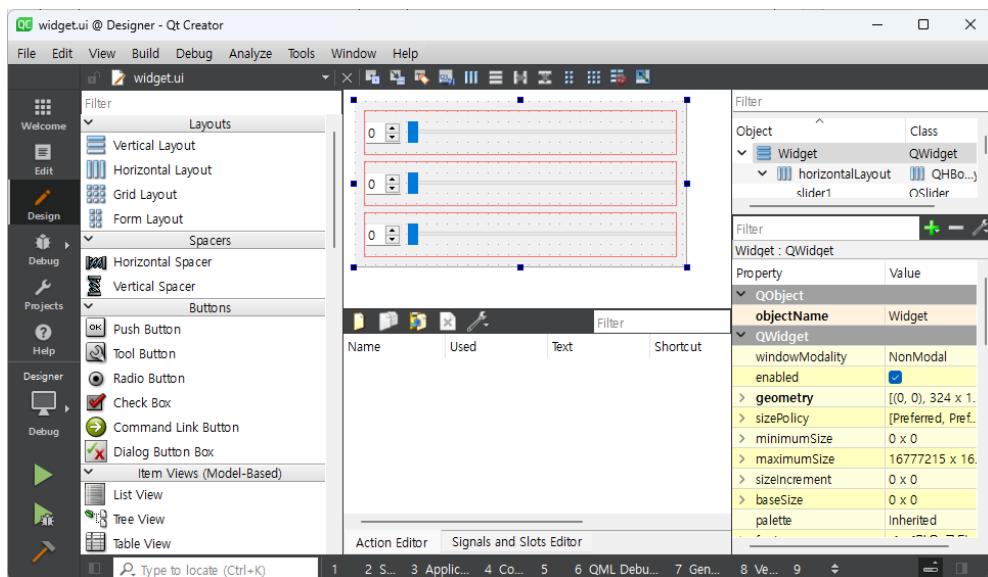
```
cnt1 = new Counter("counter 1");
cnt2 = new Counter("counter 2");

connect(cnt1, SIGNAL(valueChaged(int)), cnt2, SIGNAL(valueChaged(int)));
connect(cnt2, SIGNAL(valueChaged(int)), cnt2, SLOT(setValue(int)));
```

If you look at the top of the classes that use Signal and Slot, you can see that they use the keyword Q\_OBJECT. This keyword is mandatory when using Signal and Slot in Qt and must be specified as Q\_OBJECT in the class header. It is sometimes possible to use Signal and Slot without declaring Q\_OBJECT, which will result in an error, so be careful. The full source for this example can be found in the 00\_CustomSignalSlot directory.

## 10. Designing a GUI with Qt Designer

Qt provides the Qt Designer to help you quickly and easily implement the GUI you want. Qt Designer allows the user to place widgets on the GUI by dragging them with the mouse. When the Qt Creator IDE tool was not available, Qt Designer was provided as a standalone tool. However, it is now integrated into the Qt Creator IDE tool. The Qt Designer tool is still available as a standalone executable program. The reason for this is for developers who still use IDE tools like Visual Studio.



GUI files with the ui file extension are in XML format, as shown in the source code below. This file is automatically created in XML by the Designer tool in Qt Creator when the user places a widget using the mouse. Then, when you build, the GUI file in XML is converted to C++ source code and built.

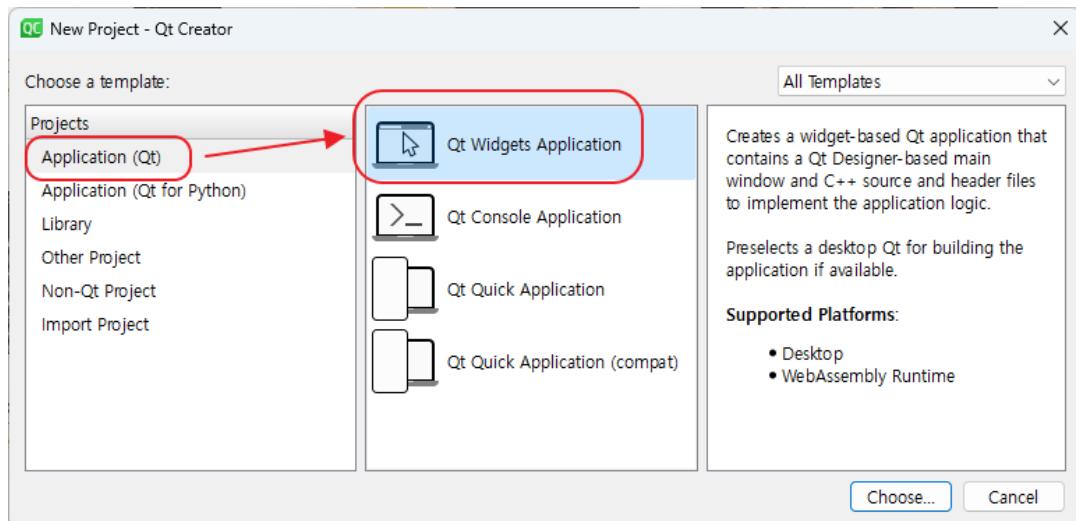
```
<?xml version="1.0" encoding="UTF-8"?>
<ui version="4.0">
<class>Widget</class>
<widget class="QWidget" name="Widget">
<property name="geometry">
<rect>
<x>0</x>
<y>0</y>
```

```
<width>338</width>
<height>178</height>
</rect>
</property>
<property name="windowTitle">
<string>Widget</string>
</property>
<widget class="QWidget" name="horizontalLayoutWidget">
...

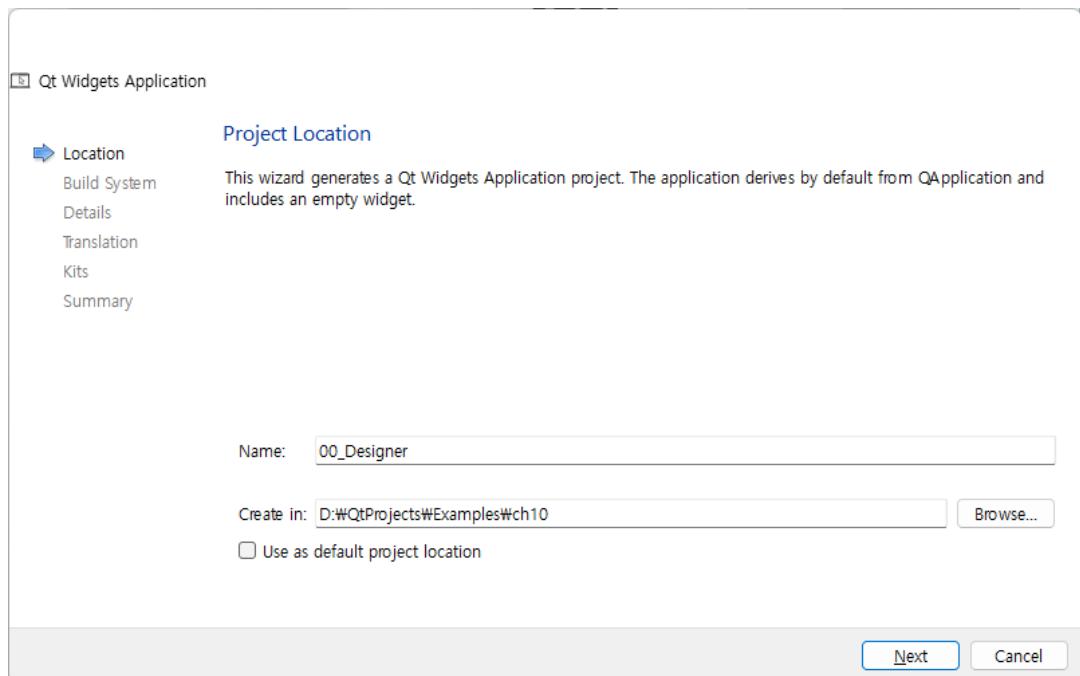
```

✓ Example using Qt Designer

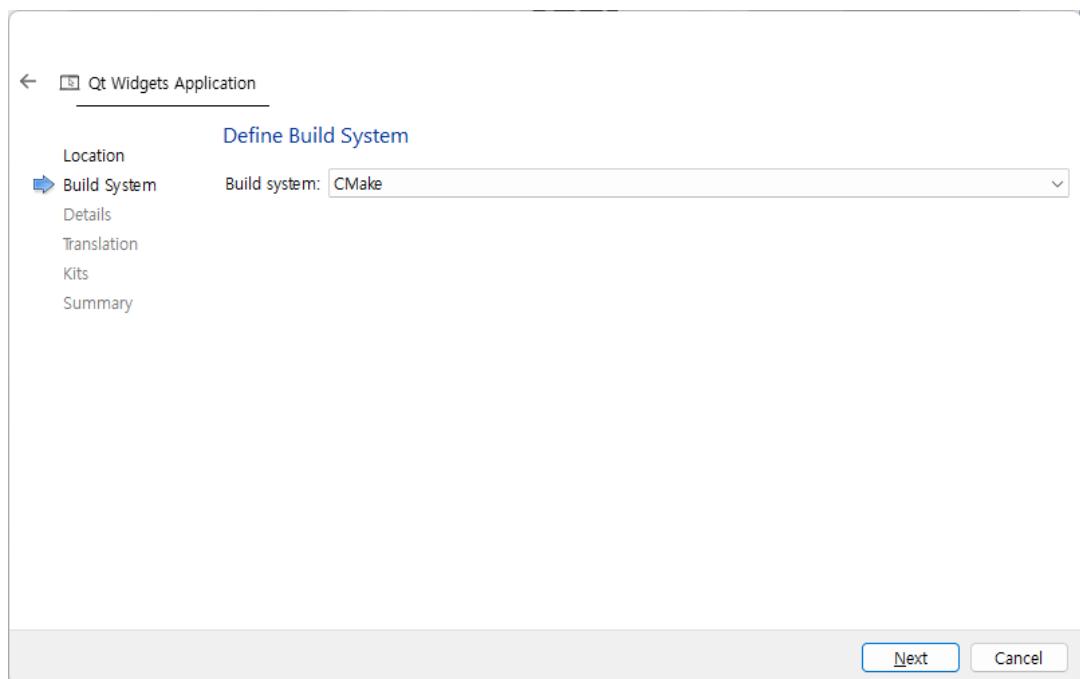
Run Qt Creator as shown in the following figure, and then click [File] > [New file or Project...] from the menu. When the dialog loads, select the [Application] item from the [Projects] item on the left side of the dialog.



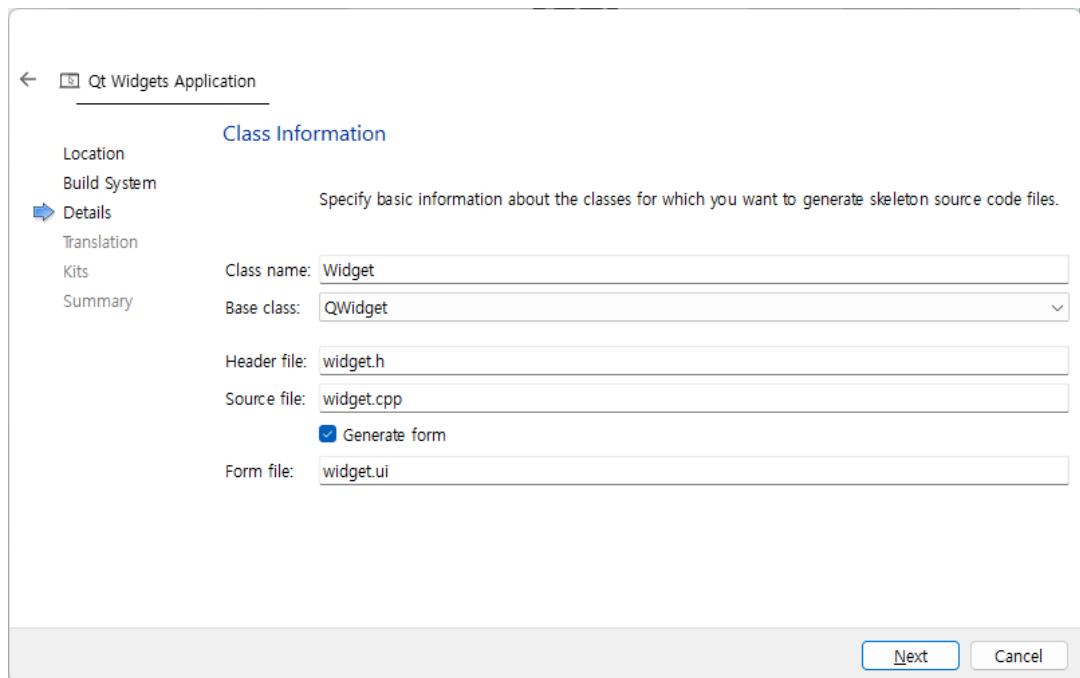
Then select Qt Widgets Application from the items in the center and click the [Choose] button at the bottom. On the next screen, enter a name for the project and the location where it will be created and click the [Next] button.



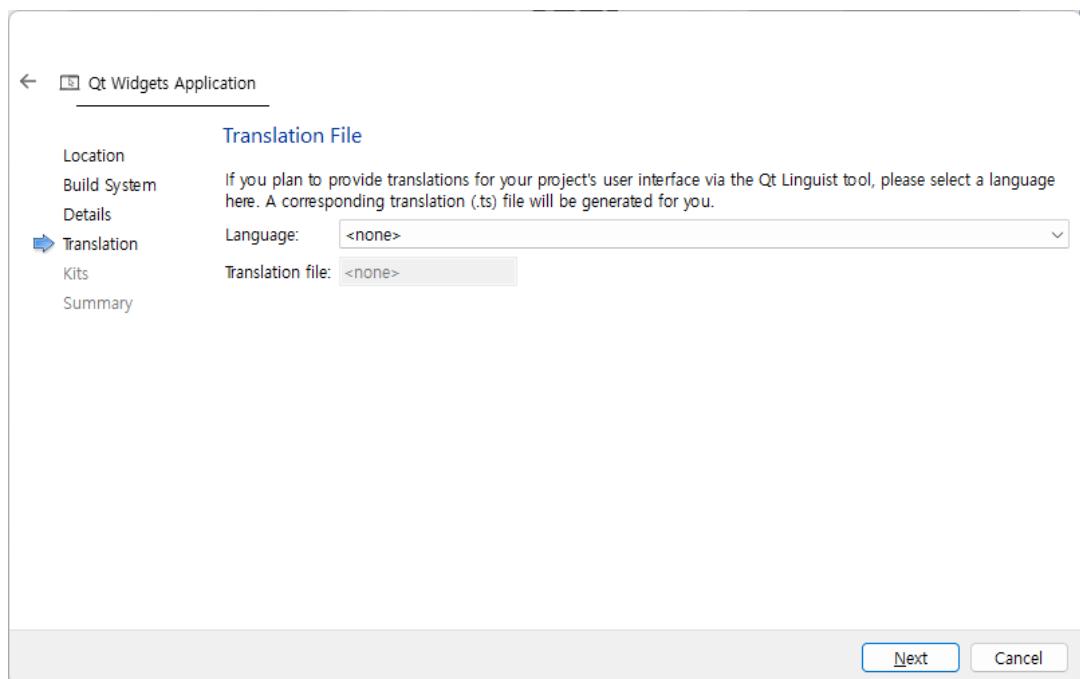
You can choose qmake as your build tool, but we'll use CMake.



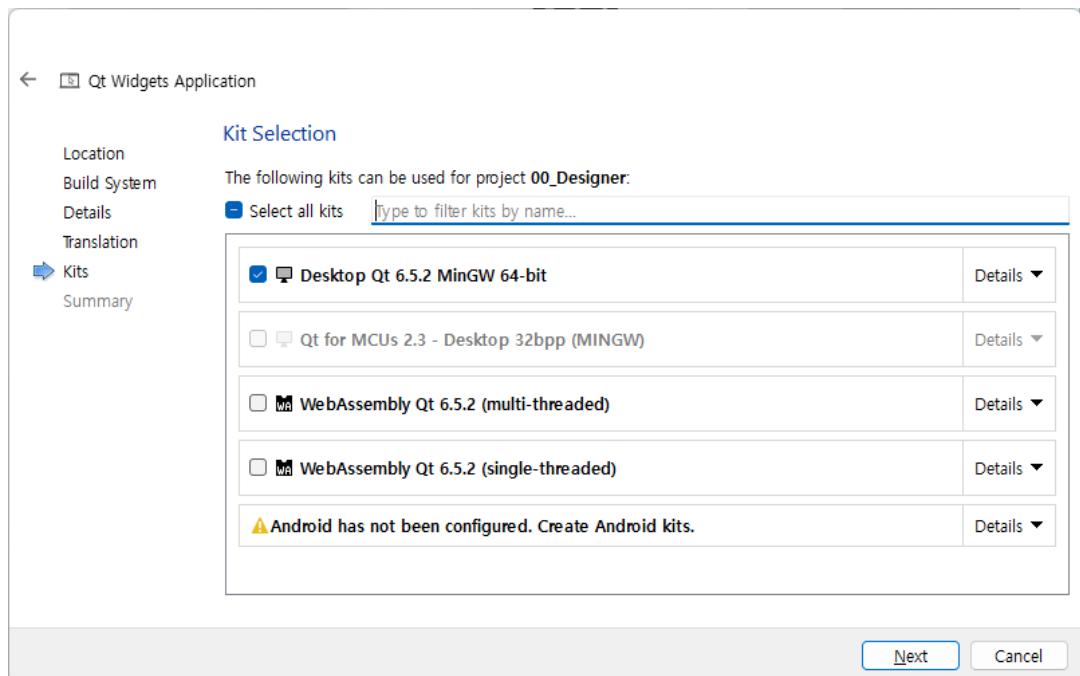
Next, enter the Class Information as shown below and click the [Next] button at the bottom.



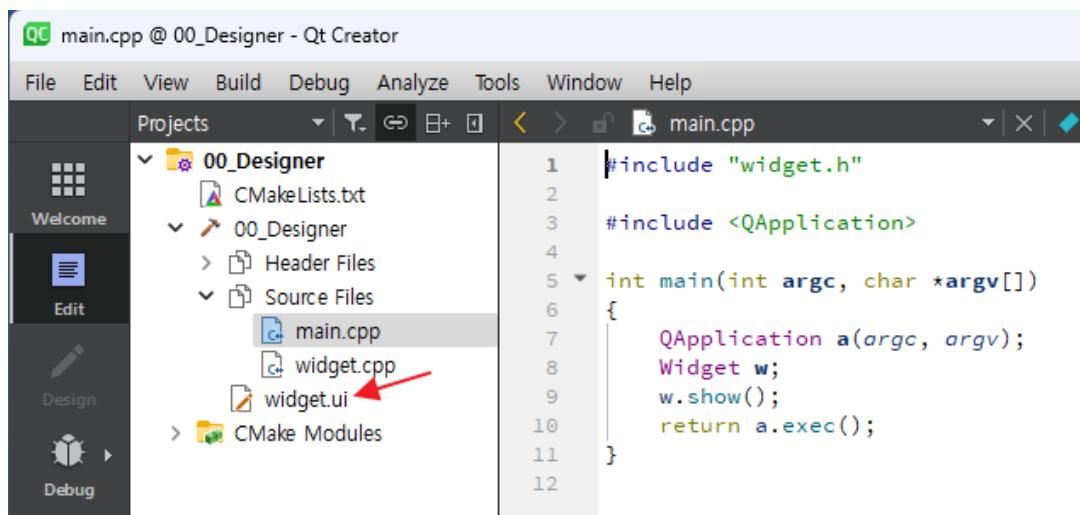
On the next screen, click the [Next] button without selecting anything.

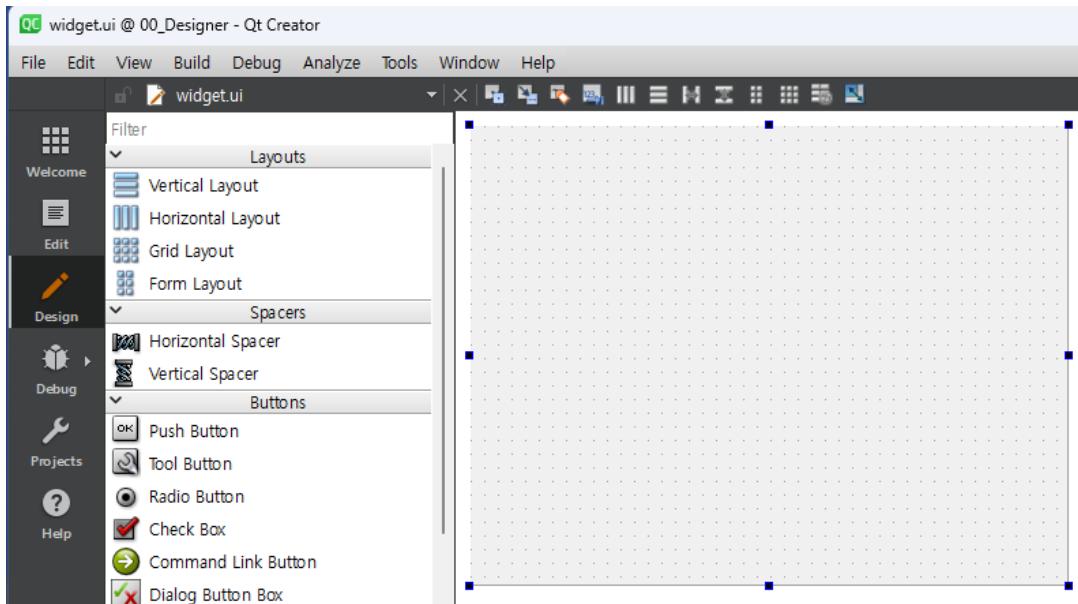


Next, select the compiler you want to build your application with. Select a MinGW compiler. Select one. After completing the selection, click the [Next] button at the bottom.

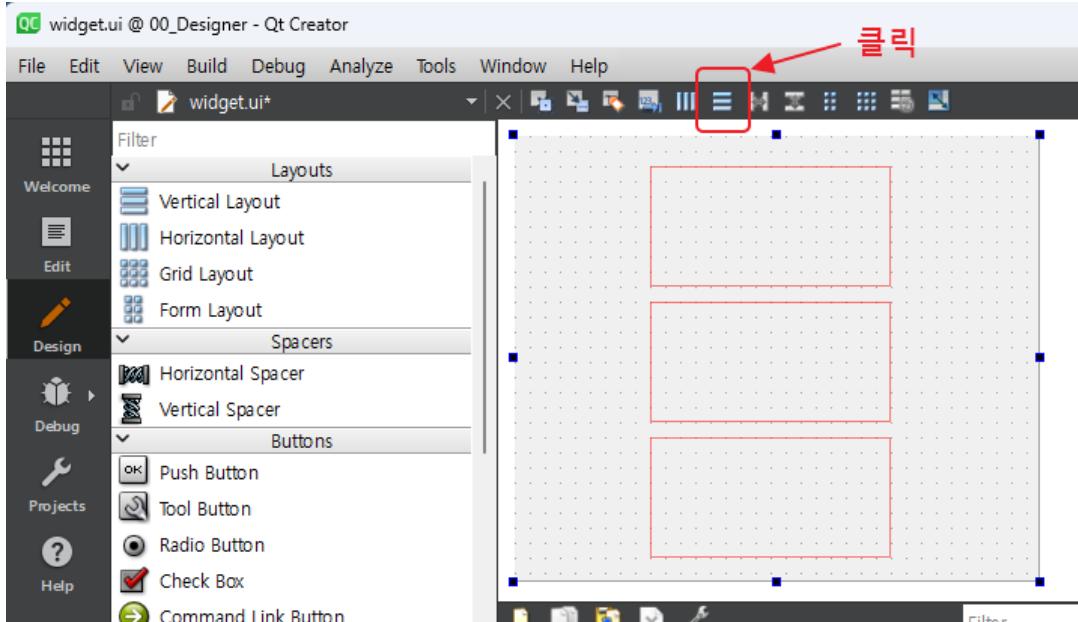


In the next dialog window, click the [Finish] button at the bottom to complete the project creation. When the project is created, double-click the `widget.ui` source code in the project window on the left side of Qt Creator to switch to the Designer screen, as shown in the following figure.

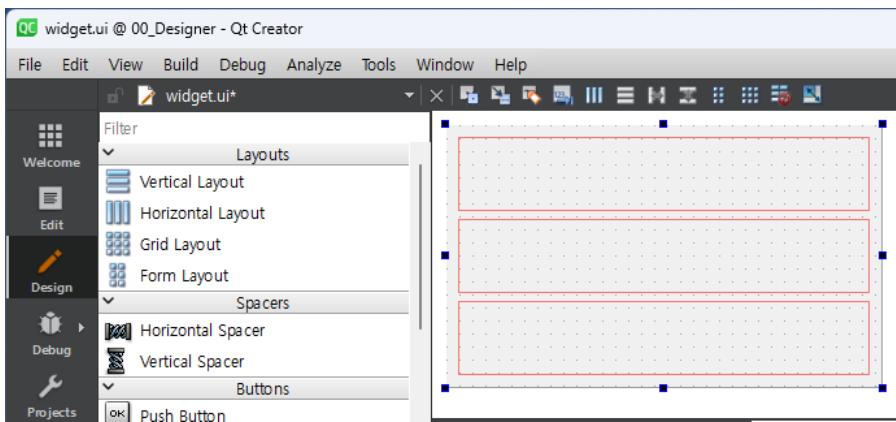




Let's place the widgets on the screen where we switched to the Designer tool, as shown above. First, drag the three Horizontal Layouts from the left [Layout] tab to the GUI Widgets. Then, click the icon button with three rectangles stacked vertically on top of each other in the top toolbar, as shown in the following image.



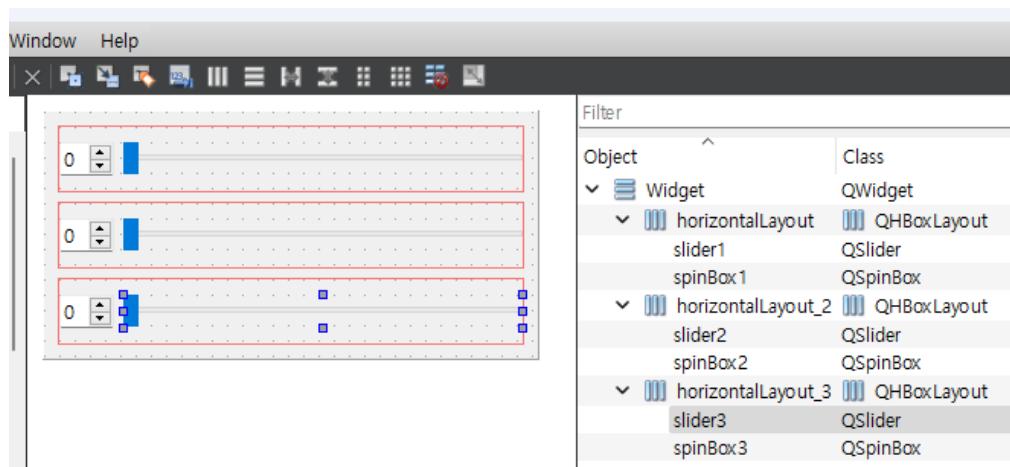
As shown in the image above, when you click the icon, the Horizontal Layout on the window will automatically resize as the window size changes.



Once you have the screens laid out as shown in the previous image, place a QSpinBox and a QSlider in each Horizontal Layout.

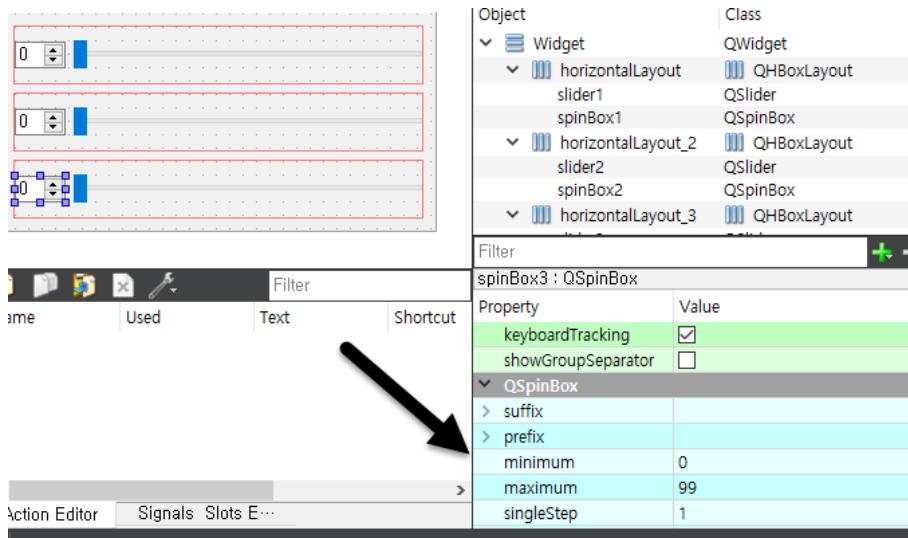
Layout	Class	Object Name
First Horizontal Layout	QSpinBox	spinBox1
	QSlider	slider1
Second Horizontal Layout	QSpinBox	spinBox2
	QSlider	slider2
Third Horizontal Layout	QSpinBox	spinBox3
	QSlider	slider3

Drag each widget to change the Object Name of the widgets as shown in the table below. You can change the Object Name by double-clicking the Object Name item in the bottom right corner.



Jesus loves you.

After renaming the Object Name, change the minimum value of each widget to 0 and the maximum value to 99, as shown in the image below.



Once you have all of your widgets placed on the GUI, build and run the application. Once the example application is running, try resizing the widgets with the mouse and see if they resize automatically.



When you reposition the scale of each QSlider widget you place in the Designer tool with the mouse, the value of the QSlider changes. This is an example of setting the changed value to the value of the QSpinBox when this value changes. The source code below is the `widget.h` header file.

```
#ifndef WIDGET_H
#define WIDGET_H
#include <QWidget>

namespace Ui { class Widget; }
```

```

class Widget : public QWidget
{
    Q_OBJECT
public:
    explicit Widget(QWidget *parent = nullptr);
    ~Widget();
private:
    Ui::Widget *ui;
private slots:
    void slider1_valueChanged(int value);
    void slider2_valueChanged(int value);
    void slider3_valueChanged(int value);
};

#endif // WIDGET_H

```

In the widget.h header file, there is an Object name that declares the ui. This object name is the accessor to the widget you placed in the Designer tool. For example, to access an object named slider2 placed by the Designer tool, you can use the ui object in the source code.

The Slot function at the bottom of the example source code below is the Slot function that is called when the value of the QSlider you placed in Designer changes. The following is the widget.cpp source code.

```

#include "widget.h"
#include "ui_widget.h"

Widget::Widget(QWidget *parent) : QWidget(parent), ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->slider1, SIGNAL(valueChanged(int)),
            this,           SLOT(slider1_valueChanged(int)));
    connect(ui->slider2, SIGNAL(valueChanged(int)),
            this,           SLOT(slider2_valueChanged(int)));
    connect(ui->slider3, SIGNAL(valueChanged(int)),
            this,           SLOT(slider3_valueChanged(int)));
}

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

```

```
void Widget::slider1_valueChanged(int value)
{
    ui->spinBox1->setValue(value);
}

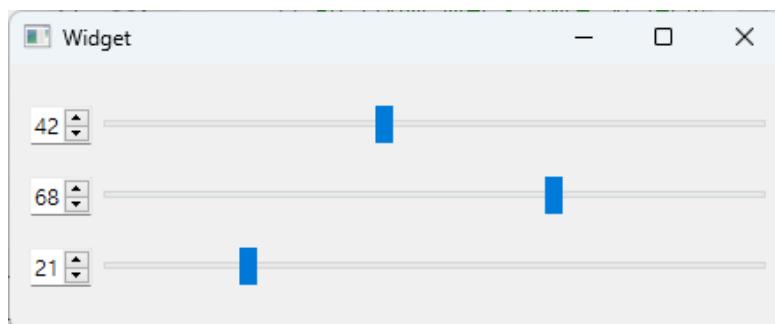
void Widget::slider2_valueChanged(int value)
{
    ui->spinBox2->setValue(value);
}

void Widget::slider3_valueChanged(int value)
{
    ui->spinBox3->setValue(value);
}
```

One thing to note in the above example is that the arguments to the connect( ) function are used in Old Style. If you use the New Style, you will get the error "no matching member function for call to 'connect'". This error occurs because the valueChanged( ) member function provided by QSpinBox is an overloaded member function that provides two member functions: an int type and a QString type.

```
void valueChanged(int i)
void valueChanged(const QString &text)
```

If you have an overloaded signal, as shown in the source code above, you should use the Old Style. Here is a screen shot of the example we wrote so far.



The full source code for the example can be found in the 00\_Designer directory.

## 11. Dialog

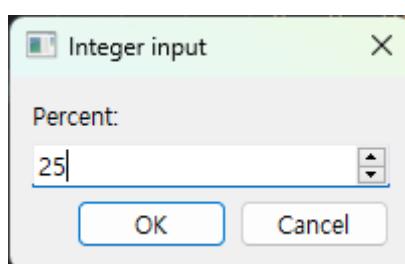
Dialogs are used by applications to communicate with the user when events occur during their behavior. They also provide a GUI to accept input from the user or to select one of several options. The following table lists the most commonly used dialogs provided by Qt.

종류	설명
QInputDialog	Dialog to accept value input from the user
QColorDialog	Dialog to select a specific color
QFileDialog	Provides a GUI interface to select a file or directory.
QFontDialog	Dialog for selecting a font
QProgressDialog	Dialogs to show progress such as percentages
QMessageBox	Modalized dialogs

- ✓ [QInputDialog](#)

The QInputDialog class can accept value input from the user.

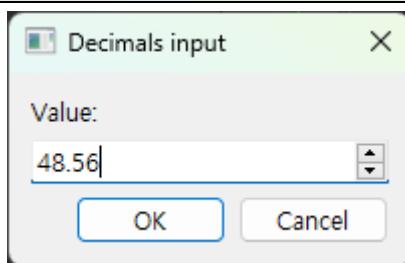
```
bool retVal;
int i = QInputDialog::getInt(this, "Integer input", "Percent:",
                             25, 0, 100, 1, &retVal);
if (retVal)
    qDebug("true %d", i);
```



The getInt( ) member function of class QInputDialog provides a dialog that accepts integer values from the user. The getInt( ) member function of class QInputDialog provides a dialog that accepts an integer value from the user.

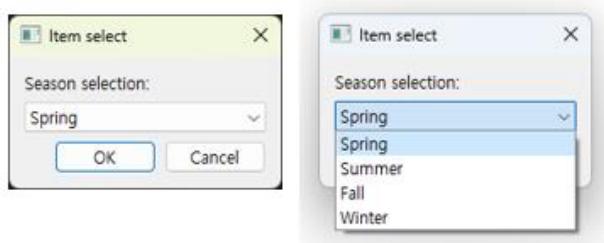
The first argument is the parent class, the second argument is the title to display in the window's title bar, and the third argument is the item name to display to the left of the input value widget item. The fourth argument is the default setting value, the fifth and sixth are the range of values the user can enter, and the next argument is the increment value for the dialog's spin box. The last argument stores a value to check if the OK or Cancel button was clicked in the dialog.

```
bool retValue;
double dVal = QInputDialog::getDouble(this, " Decimals input", "Value:",
                                         48.56, -100, 100, 2, &retValue);
```



The getDouble( ) member function of class QInputDialog can accept a real number as input. The getItem( ) member function provides the ability to select one of the strings (or words).

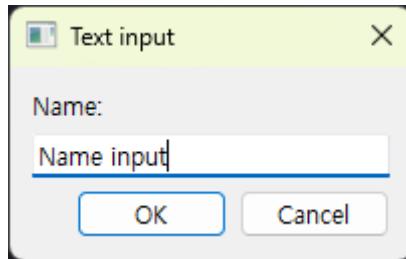
```
QStringList items;
items << "Spring" << "Summer" << "Fall" << "Winter";
bool ok;
QString item = QInputDialog::getItem(this, "Item select", "Season selection: ",
                                       items, 0, false, &ok);
```



Jesus loves you.

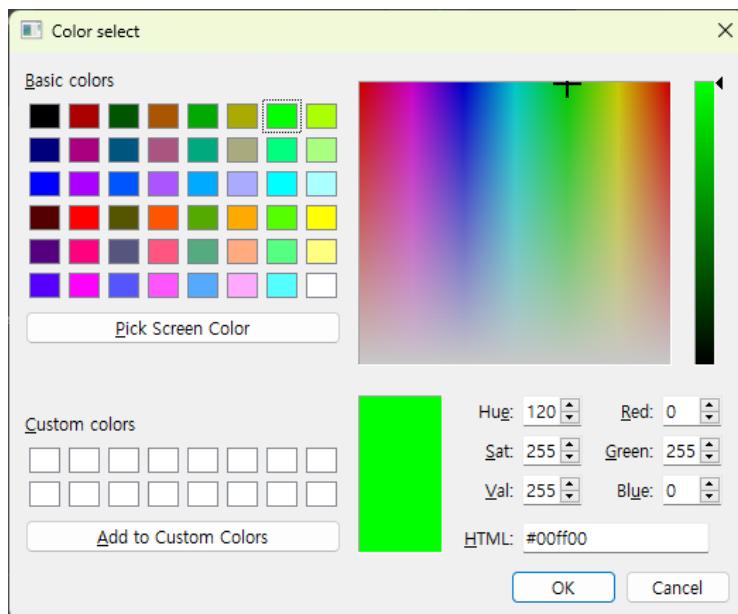
QInputDialog 클래스의 getText() 멤버 함수는 사용자로부터 텍스트를 입력 받을 수 있는 다이얼로그를 제공한다.

```
bool ok;  
QString text = QInputDialog::getText(this, "Text input", "Name: ",  
                                     QLineEdit::Normal, "Name input", &ok);
```



✓ QColorDialog

The QColorDialog class provides functionality for the user to view a color palette and select the desired color.



```
QColor color;  
color = QColorDialog::getColor(Qt::green, this, "Color select",
```

```
QColorDialog::DontUseNativeDialog);  
  
if (color.isValid())  
    qDebug() << Q_FUNC_INFO << "Valid color.";
```

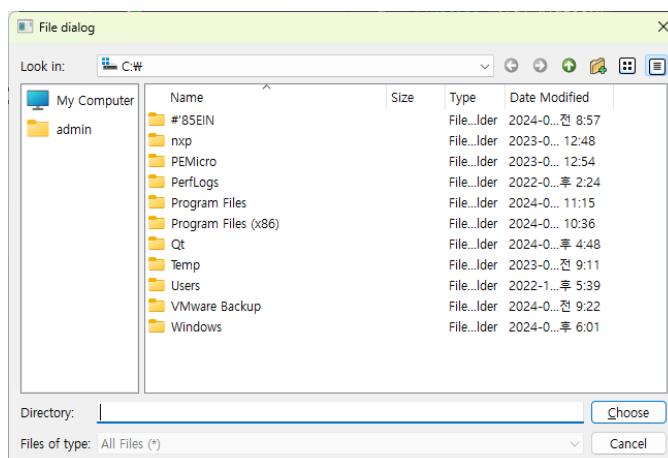
✓ **QFileDialog**

The QFileDialog class provides a dialog that allows the user to select a file. You can filter for specific extensions or specific files and display them to the user.

The getOpenFileNames( ) member function of the QFileDialog class provides the ability to multi-select files that exist within a directory. The getExistingDirectory( ) member function allows the user to select a directory. And the getSaveFileName( ) member function allows the user to specify a file to save.

```
QFileDialog::Options options;  
options = QFileDialog::DontResolveSymlinks | QFileDialog::ShowDirsOnly;  
options |= QFileDialog::DontUseNativeDialog;  
  
QString directory = QFileDialog::getExistingDirectory(this,  
                                         "File dialog", "C:", options);
```

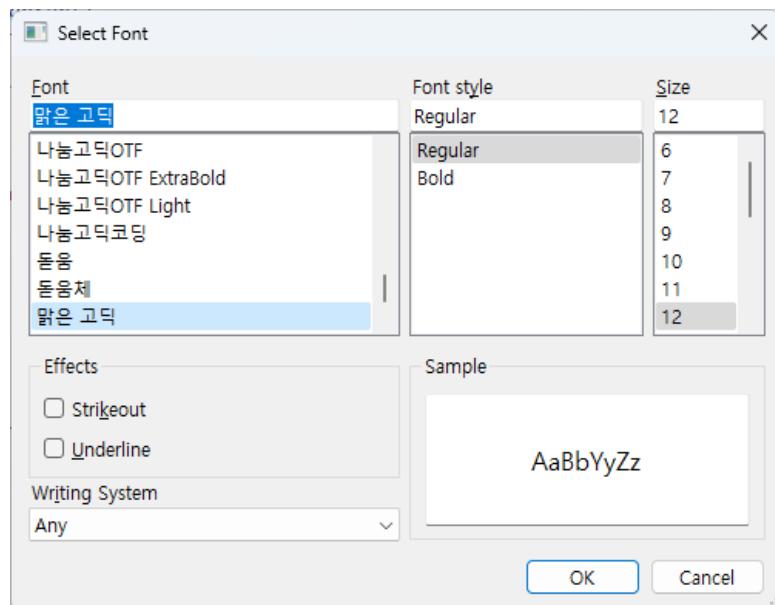
getExistingDirectory( ) 멤버 함수는 사용자로부터 디렉토리를 선택할 수 있는 기능을 제공한다. 첫 번째 인자는 부모 클래스, 두 번째 인자는ダイ얼로그의 타이틀 바 제목, 세 번째 인자는 지정한 디렉토리가 디폴트로 지정하기 위해 사용하는 인자이다. 마지막 인자는 파일ダイ얼로그의 상수 값을 이용해 필터링하기 위한 옵션이다.



상수	설명
QFileDialog::ShowDirsOnly	Show only directories
QFileDialog::DontResolveSymlinks	Omit symbolic links
QFileDialog::DontConfirmOverwrite	Don't show warning messages when overwriting
QFileDialog::DontUseNativeDialog	Use to disable the system default file dialog
QFileDialog::ReadOnly	Using the file dialog in read mode
QFileDialog::HideNameFilterDetails	Using filters to hide files

✓ QFontDialog

QFontDialog provides a dialog that allows the user to select a font. The first argument specifies a variable to determine whether the user clicked the [OK] button or the [CANCEL] button located at the bottom of the dialog.



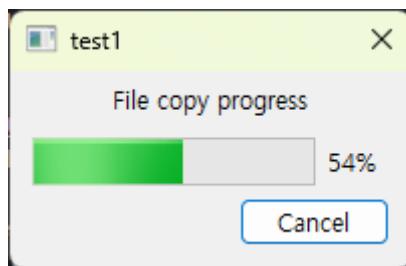
QFontDialog provides a dialog that allows the user to select a font. The first argument specifies a variable to determine whether the user clicked the [OK] button or the [CANCEL] button located at the bottom of the dialog.

The second argument specifies a font that can be specified as the default selection on the font dialog.

```
bool ok;
QFont font;
font = QFontDialog::getFont(&ok, QFont( "Courier 10 Pitch" ), this);
```

✓ QProgressDialog

The QProgressDialog class is used to show the user the current progress. For example, you might want to show progress in a dialog for something that takes some time, such as copying a large file.



The following is the source code for an example dialog that uses the QProgressDialog class to show the user progress. Create a project with the QWidget class as the base class, and then create the widget.h header file as follows

```
#ifndef WIDGET_H
#define WIDGET_H
#include <QWidget>
#include <QProgressDialog>
#include <QTimer>

namespace Ui { class Widget; }

class Widget : public QWidget
{
    Q_OBJECT
public:
    explicit Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
    QProgressDialog *pd;
```

```

QTimer *timer;
int steps;

public slots:
    void perform();
    void cancel();
};

#endif // WIDGET_H

```

In the example above, QTimer is used to call a timer event once every second to increment the progress value of the QProgressDialog by 1%. The following source code is the widget.cpp source code.

```

#include "widget.h"
#include "ui_widget.h"
#include <QDebug>

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

    steps = 0;
    pd = new QProgressDialog("File copy progress", "Cancel", 0, 100);
    connect(pd, SIGNAL(canceled()), this, SLOT(cancel()));

    timer = new QTimer(this);
    connect(timer, SIGNAL(timeout()), this, SLOT(perform()));
    timer->start(1000);
}

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

void Widget::perform()
{
    pd->setValue(steps);
    steps++;

    if (steps > pd->maximum())

```

```

    timer->stop();
}

void Widget::cancel()
{
    timer->stop();
}

```

✓ QMessageBox

The QMessageBox class provides a modal dialog. The buttons that can convey information to the user and that the QMessageBox class makes available in the dialog window provide a variety of buttons for use, as shown in the following table.

Constant	Value	Description
QMessageBox::Ok	0x00000400	OK Button
QMessageBox::Open	0x00002000	Open File Button
QMessageBox::Save	0x00000800	Save Button
QMessageBox::Cancel	0x00400000	Cancel Button
QMessageBox::Close	0x00200000	Close Button
QMessageBox::Discard	0x00800000	Discard without saving button
QMessageBox::Apply	0x02000000	Apply button
QMessageBox::Reset	0x04000000	Reset button
QMessageBox::RestoreDefaults	0x08000000	Resave button
QMessageBox::Help	0x01000000	Help button
QMessageBox::SaveAll	0x00001000	Save All button
QMessageBox::Yes	0x00004000	YES 버튼
QMessageBox::YesToAll	0x00008000	Apply Batch YES button
QMessageBox::No	0x00010000	NO button
QMessageBox::NoToAll	0x00020000	Apply Bulk NO button
QMessageBox::Abort	0x00040000	Stop button

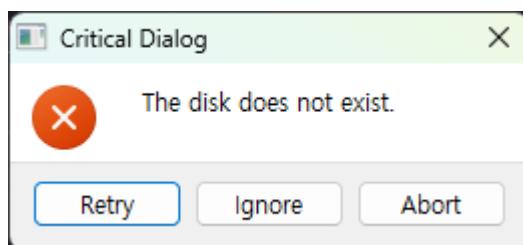
QMessageBox::Retry	0x00080000	Retry button
QMessageBox::Ignore	0x00100000	Ignore button
QMessageBox::NoButton	0x00000000	An invalid button

The following example uses the QMessageBox class to enable the [Abort], [Retry], and [Ignore] buttons.

```
QMessageBox::StandardButton reply;

reply = QMessageBox::critical(this,
                            "Critical Dialog",
                            "The disk does not exist.",
                            QMessageBox::Abort |
                            QMessageBox::Retry |
                            QMessageBox::Ignore);

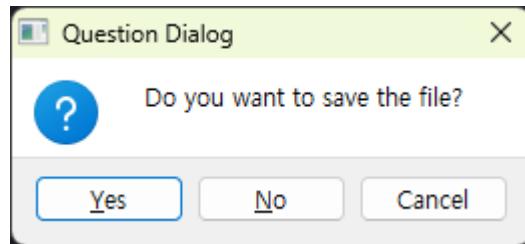
if (reply == QMessageBox::Abort)
    ...
else if (reply == QMessageBox::Retry)
    ...
```



The QMessageBox class provides the information( ), question( ), and warning( ) member functions. The following is an example of using the question( ) member function.

```
QMessageBox::StandardButton reply;
reply = QMessageBox::question(this, "Question Dialog",
                            "Do you want to save the file?",
                            QMessageBox::Yes | QMessageBox::No | QMessageBox::Cancel);

if (reply == QMessageBox::Abort)
    ...
else if (reply == QMessageBox::Retry)
    ...
```



- ✓ Example of implementing a custom dialog

In this example, we'll inherit from the QDialog class and implement a dialog of the user's choice. The following example source code is the dialog.h header file.

```
#ifndef DIALOG_H
#define DIALOG_H
#include <QDialog>
#include <QLabel>
#include <QLineEdit>
#include <QPushButton>
#include <QVBoxLayout>
#include <QHBoxLayout>

class Dialog : public QDialog
{
    Q_OBJECT
public:
    Dialog(QWidget *parent = 0);
    ~Dialog();

private:
    QLabel      *lbl;
    QLineEdit   *edit;
    QPushButton *okbtn;
    QPushButton *cancelbtn;

private slots:
    void slot_okbtn();
    void slot_cancelbtn();
};

#endif // DIALOG_H
```

The dialog.h header file is a header file for implementing a class that inherits from the

QDialog class. This dialog accepts a name and has two buttons. The following example source code is the complete source code for dialog.cpp.

```
#include "dialog.h"

Dialog::Dialog(QWidget *parent) : QDialog(parent)
{
    setWindowTitle("Custom Dialog");

    lbl = new QLabel("Name");
    edit = new QLineEdit("");
    okbtn = new QPushButton("Confirm");
    cancelbtn = new QPushButton("Cancel");

    QHBoxLayout *hlay1 = new QHBoxLayout();
    hlay1->addWidget(lbl);
    hlay1->addWidget(edit);
    QHBoxLayout *hlay2 = new QHBoxLayout();
    hlay2->addWidget(okbtn);
    hlay2->addWidget(cancelbtn);

    QVBoxLayout *vlay = new QVBoxLayout();
    vlay->addLayout(hlay1);
    vlay->addLayout(hlay2);
    setLayout(vlay);
}

void Dialog::slot_okbtn()
{
    emit accepted();
}

void Dialog::slot_cancelbtn()
{
    emit rejected();
}

Dialog::~Dialog()
{
```

The slot\_okbtn( ) Slot function is called when the [OK] button of the dialog is clicked. The slot\_cancelbtn( ) Slot function is called when the [Cancel] button is clicked. The

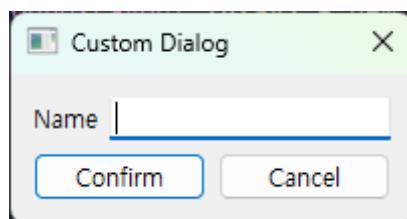
following example source code is from the main.cpp source code.

```
#include <QApplication>
#include <QDebug>
#include "dialog.h"

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

    Dialog dlg;
    int retVal = dlg.exec();
    if(retVal == QDialog::Accepted)
    {
        qDebug() << Q_FUNC_INFO << "QDialog::Accepted";
    }
    else if(retVal == QDialog::Rejected)
    {
        qDebug() << Q_FUNC_INFO << "QDialog::Rejected";
    }

    return a.exec();
}
```



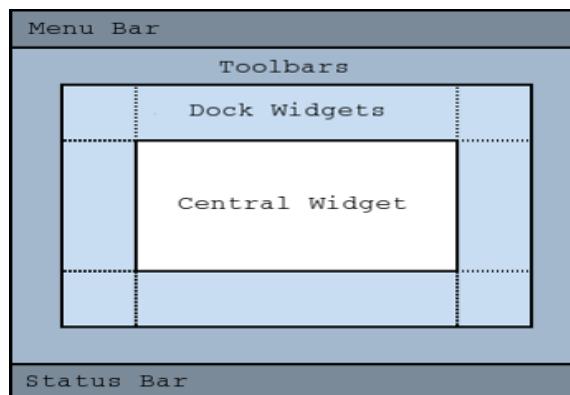
As shown in the example above, a `dlg` object of class `Dialog` returns an `int` value of 1 when the user clicks the [OK] button. If the user clicks the [Cancel] button, it returns 0. For the complete source of the example, refer to the `00_CustomDialog` directory.

## 12. Implementing a GUI with QMainWindow

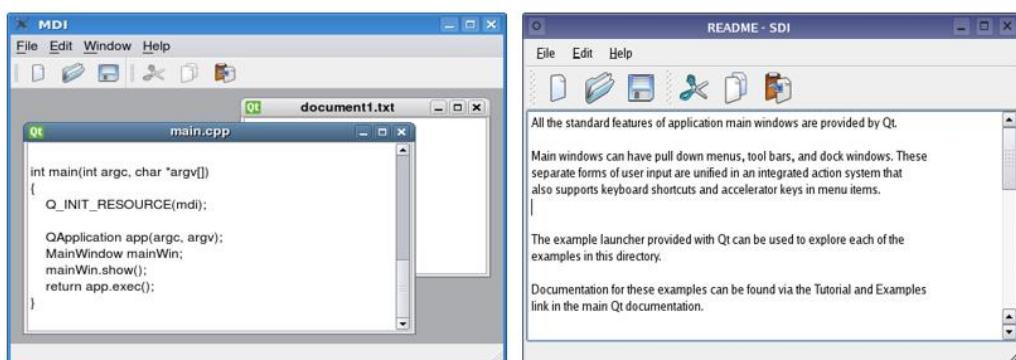
So far, we've covered an example of using Qt to place a GUI-based widget on a window. We have used QWidget to organize the GUI in such a way that there is only one window screen.

However, if your GUI is complex and needs to provide many functions to the user, you may want to use QMainWindow rather than QWidget to implement the GUI to provide the user with an intuitive GUI.

For example, you can place widgets in specific areas such as Menu Bar, Toolbars, Status Bar, Dock Widget, Central Widget, etc.



Qt can also implement the Multi Document Interface (MDI) method. The MDI method can be implemented using the QMdiArea class. If you need to implement a complex windowed GUI, it is recommended to use the QMdiArea class in conjunction with QMainWindow.



- ✓ Example MDI-based GUI using QMdiArea class

In this example, we'll use the QMdiArea class to implement an MDI-based GUI. This example is provided when you install Qt. You can find the complete source code in the MDI Example in Examples. This example source code is implemented in two classes.

The MainWindow class inherits from the QMainWindow class and implements the main window GUI. This class implements the Menu Bar, Tool Bar, and the Central Widget area.

Class MDIMainWindow is a widget class that inherits from class QTextEdit widget class. This class is used as a multi-edit widget to be centered in the MainWindow, i.e., it is implemented to provide the ability to edit multiple text files within a single window area, such as Ultra Editor, Notepad, etc. The following example source code is the header source code for the MainWindow class.

```
#include <QMainWindow>
#include "MDIMainwindow.h"

class MainWindow : public QMainWindow
{
    Q_OBJECT
public:
    MainWindow(QWidget *parent = nullptr);
    ~MainWindow();
private slots:
    void newFile();
    void open();
};

...
```

The newFile( ) Slot function is called when the [New] menu is clicked from the menu bar. The open( ) function is called when the [Open] menu is clicked. The following example is the mainwindow.cpp source code.

```
#include "mainwindow.h"
#include <QMenu>
#include <QAction>
#include <QMenuBar>
#include <QToolBar>
#include <QDockWidget>
#include <QListWidget>
#include <QStatusBar>
```

```
#include <QDebug>

MainWindow::MainWindow(QWidget *parent) : QMainWindow(parent)
{
    QMenu *fileMenu;
    QAction *newAct;
    QAction *openAct;

    newAct = new QAction(QIcon(":/images/new.png"), tr("&New"), this);
    newAct->setShortcuts(QKeySequence::New);
    newAct->setStatusTip(tr("Create a new file"));
    connect(newAct, SIGNAL(triggered()), this, SLOT(newFile()));

    openAct = new QAction(QIcon(":/images/open.png"), tr("&Open"), this);
    openAct->setShortcuts(QKeySequence::Open);
    openAct->setStatusTip(tr("Open an existing file"));
    connect(openAct, SIGNAL(triggered()), this, SLOT(open()));

    fileMenu = menuBar()->addMenu(tr("&File"));
    fileMenu->addAction(newAct);
    fileMenu->addAction(openAct);

    QToolBar *fileToolBar;
    fileToolBar = addToolBar(tr("File"));
    fileToolBar->addAction(newAct);
    fileToolBar->addAction(openAct);

    QDockWidget *dock = new QDockWidget(tr("Target"), this);
    dock->setAllowedAreas(Qt::LeftDockWidgetArea | Qt::RightDockWidgetArea);

    QListWidget *customerList = new QListWidget(dock);
    customerList->addItems(QStringList()
        << "One" << "Two" << "Three" << "Four" << "Five");

    dock->setWidget(customerList);
    addDockWidget(Qt::RightDockWidgetArea, dock);
    setCentralWidget(new MDIMainWindow());
    statusBar()->showMessage(tr("Ready"));
}

MainWindow::~MainWindow()
{
```

```
}

void MainWindow::newFile()
{
    qDebug() << Q_FUNC_INFO;
}

void MainWindow::open()
{
    qDebug() << Q_FUNC_INFO;
}
```

The constructor of the MainWindow class defines the menu items to be placed in the menu and toolbar on the MDI window GUI. It associates each menu item with a Slot function that fires a Signal event when clicked.

The QDockWidget is located on the left side of the MDI window and provides a GUI that allows the user to detach it into a new window on the GUI. The MDIMainWindow class is used by the MDI window as a child window, which provides functionality such as providing multiple child windows within the GUI. The following is the header source code for the MDIMainWindow class.

```
#include <QMainWindow>
#include <QObject>

class MDIMainWindow : public QMainWindow
{
    Q_OBJECT
public:
    explicit MDIMainWindow(QWidget *parent = nullptr);
};
```

This class constructor uses the QMdiArea class and QMdiSubWindow class to create a window that will be registered as a sub-window under MDIMainWindow. The following example source code is the MDIMainwindow.cpp source code.

```
#include "MDIMainwindow.h"
#include <QMdiArea>
#include <QMdiSubWindow>
#include <QPushButton>
```

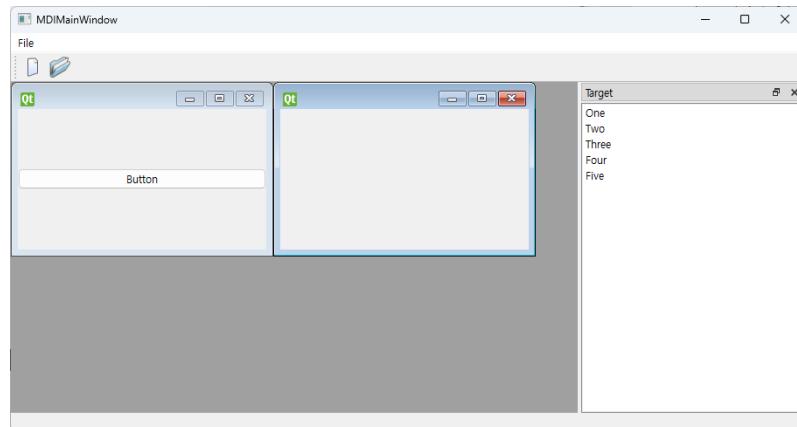
Jesus loves you.

```
MDIMainWindow::MDIMainWindow(QWidget *parent) : QMainWindow(parent)
{
    setWindowTitle(QString::fromUtf8("My MDI"));

    QMdiArea* area = new QMdiArea();
    area->setSizePolicy(QSizePolicy::Expanding, QSizePolicy::Expanding);

    QMdiSubWindow* subWindow1 = new QMdiSubWindow();
    subWindow1->resize(300, 200);
    QPushButton *btn = new QPushButton(QString("Button"));
    subWindow1->setWidget(btn);

    QMdiSubWindow* subWindow2 = new QMdiSubWindow();
    subWindow2->resize(300, 200);
    area->addSubWindow(subWindow1);
    area->addSubWindow(subWindow2);
    setCentralWidget(area);
}
```



You can refer to the 00\_MDIMainWindow directory for example sources.

## 13. Stream

In this context, a Stream is an easy way to write/read data to/from a specific variable. For example, if you need to write/read data from a variable of type quint32 (4 Bytes) into a QByteArray, you can easily handle it by using QDataStream or QTextStream provided by Qt.

QDataStream is used to write/read binary data, and QTextStream is used to write/read text-based data.

In this chapter, let's see how to use QDataStream and QTextStream using an example.

- ✓ Example using QDataStream

In this example, we will write and read data to and from a QDataStream. In the encoding( ) function, we will store data of type quint32, quint8, and quint32 in a QByteArray using a QDataStream. We'll store 123, 124, and 125 in each variable.

And in the decoding( ) function, we will implement an example of reading the stored data from the QByteArray one after the other. Below is the source code for the example

```
#include <QCoreApplication>
#include <QIODevice>
#include <QDataStream>
#include <QDebug>

QByteArray encoding()
{
    quint32 value1 = 123;
    quint8  value2 = 124;
    quint32 value3 = 125;

    QByteArray outData;
    QDataStream outStream(&outData, QIODevice::WriteOnly);

    outStream << value1;
    outStream << value2;
```

```
outStream << value3;

qDebug() << "outData size : " << outData.size() << " Bytes";

return outData;
}

void decoding(QByteArray _data)
{
    QByteArray inData = _data;

    quint32 inValue1 = 0;
    quint8 inValue2 = 0;
    quint32 inValue3 = 0;

    QDataStream inStream(&inData, QIODevice::ReadOnly);

    inStream >> inValue1; // 123
    inStream >> inValue2; // 124
    inStream >> inValue3; // 125

    qDebug("[First : %d] [Second : %d] [Third : %d]"
           , inValue1, inValue2, inValue3 );
}

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

    QByteArray encData = encoding();
    decoding(encData);

    return a.exec();
}
```

In the encoding( ) function, the data 123, 124, 125 stored in the quint32, quint8, quint32 variable types are stored in a QByteArray using a QDataStream.

Then, store the values stored in the QByteArray in encData. Then, pass encData as the first argument to the decoding( ) function. The decoding( ) function reads the QByteArray value passed in as the first argument using a QDataStream. Therefore, the value is output using qDebug( ) on the last line of the decoding( ) function. You can find the source

code for this example in the 00\_DataStream directory.

✓ Example with QTextStream

This example is written in a similar way to the previous example. In the writeData( ) function, we will write the value stored in the QByteArray using a QTextStream.

And the readData( ) function will do the opposite, reading the value stored in the QByteArray. The following example shows the complete source code for this example.

```
#include <QCoreApplication>

#include <QIODevice>
#include <QTextStream>
#include <QDebug>

QByteArray writeData(QByteArray _data)
{
    QByteArray temp = _data;

    QByteArray outData;
    QTextStream outStream(&outData, QIODevice::WriteOnly);

    for(qsizetype i = 0 ; i < temp.size() ; i++) {
        outStream << temp.at(i);
    }
    outStream.flush();

    return outData;
}

void readData(QByteArray _data)
{
    QTextStream outStream(&_data, QIODevice::ReadOnly);

    QByteArray inData;
    for(qsizetype i = 0 ; i < _data.size() ; i++)
    {
        char data;
        outStream >> data;
        inData.append(data);
    }
}
```

```
}

qDebug("READ DATA : [%s]", inData.data());
}

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

    QByteArray retData = writeData("Hello world."); // 12 Bytes
    qDebug() << "WRITE DATA size : " << retData.size() << " Bytes";

    readData(retData);

    return a.exec();
}
```

In the writeData function, we will pass "Hello world" as the first argument. This string is stored in a QByteArray. We store this stored data in a QByteArray using a QTextStream.

And in the readData( ) function, we'll pass in the QByteArray containing "Hello world." The readData( ) function will read the data from the QByteArray containing "Hello world." using a QTextStream. Therefore, the last line of the readData( ) function will output "Hello world."

For the source code of this example, see [01\\_QTextStream](#).

## 14. File input and output

Qt provides the QFile class for handling file input and output. In addition, QTextStream and QDataStream classes can be used together to efficiently read/write large amounts of data. You can use only the member functions provided by QFile to READ/WRITE files with small file sizes, but QTextStream and QDataStream classes can be used to handle files with large file sizes, and they can be used to handle large files efficiently.

The following example shows how to read data from a file using only the QFile class without using a data stream.

- ✓ Reading data from a file using the QFile class

```
QFile file("in.txt");
if (!file.open(QIODevice::ReadOnly | QIODevice::Text))
    return;

while (!file.atEnd()) {
    QByteArray line = file.readLine();
    ...
}
```

The open( ) member function of class QFile provides the ability to open a file. When opening a file, you can specify in the open( ) function argument whether the file should be opened in READ mode, WRITE mode, or a combination of READ/WRITE.

The QIODevice::ReadOnly option allows the file to be read only. The QIODevice::Text option allows the file to be used in TEXT mode.

Constant	Value	Description
QIODevice::NotOpen	0x0000	Use when not opening a file
QIODevice::ReadOnly	0x0001	Use in read-only mode
QIODevice::WriteOnly	0x0002	Use in write-only mode
QIODevice::ReadWrite	Read   Write	Use read/write mode together
QIODevice::Append	0x0004	Use to append to the end of a file

QIODevice::Truncate	0x0008	If the file was previously open, delete the previously used file to make way for the new connection.
QIODevice::Text	0x0010	Use 'Wn' at the end when reading in TEXT mode. In MS Windows, use 'WrWn' at the end.
QIODevice::Unbuffered	0x0020	Use devices directly without using buffers

In the example source code above, the `atEnd()` member function used as a condition in the while statement iterates until it reaches the end of the file. The `readLine()` member function provides the ability to read until the '`\n`' character is encountered.

Class `QFile` can read/write files using classes `QTextStream` and `QDataStream`, which handle data STREAMS. The following example shows how to use the `QFile` and `QTextStream` classes to read data from a file.

```
#include <QCoreApplication>
#include <QFile>
#include <QString>
#include <QDebug>
#include <QTextStream>

void write(QString filename)
{
    QFile file(filename);
    if(!file.open(QFile::WriteOnly | QFile::Text)) {
        qDebug() << "Open fail.";
        return;
    }

    QTextStream out(&file);
    out << "Write Test";

    file.flush();
    file.close();
}

void read(QString filename)
{
    QFile file(filename);
    if(!file.open(QFile::ReadOnly | QFile::Text)) {
```

```
qDebug() << " Open fail.";
return;
}

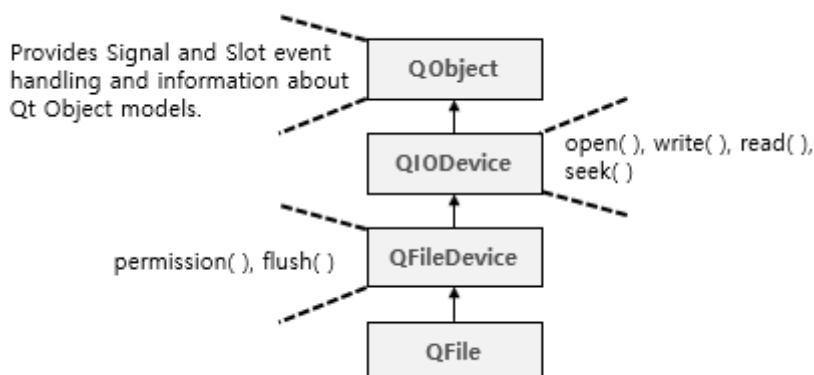
QTextStream in(&file);
qDebug() << in.readAll();

file.close();
}

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    QString filename = "C:/Qt/MyFile.txt";
    write(filename);
    read(filename);
    return a.exec();
}
```

Class QFile provides functions such as Open, Exists, Link, Remove, Rename, etc. for handling files. QFile class is implemented by inheriting from QFileDevice class. The QFileDevice class implements exception handling functions such as Permission, Flush, and Error handling for files.

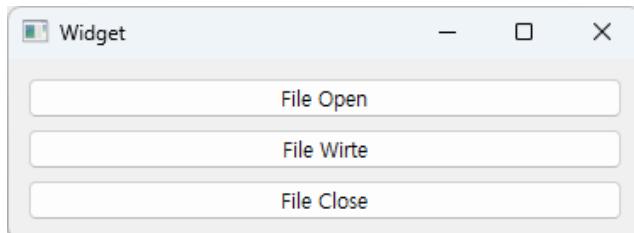
Class QFileDevice inherits from class QIODevice. QIODevice class implements functions to open, write, read, and seek files. QIODevice is inherited from and used by many other classes besides QFile. For example, in multimedia, it is used to read media such as sound data from MP3 files.



The reason for the inheritance of the QFile class is that if you need to implement an API

to read/write data from a new device, you can use the same inheritance as QFile and use the already implemented functions like Open, Write, Read, and Seek.

- ✓ Example using the QFile class



This example is an example of implementing the Open, Write and Close file functions. When you click the [File Open] button, you can select one of the files in the file dialog. This opens the file you selected and reads data from the file using the QTextStream class. The read data is read line by line and output to the Console using qDebug().

Click the [File Write] button to move to the last position of the file using the seek( ) member function of the QFile class and write the string "Hello World" to the file. Click the [File Close] button to close the file. The following example shows the widget.h source code.

```
#include <QWidget>
#include <QFile>

namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT
public:
    explicit Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
    QFile *mFile;

private slots:
```

```
void slotPbtOpenPress();
void slotPbtWritePress();
void slotPbtClosePress();
void slotAboutToClose();
};
```

In the constructor of this class, we initialize an mFile object. Then, when the file gets a Close signal, it calls the slotAboutToClose( ) Slot function. The following example source code is from widget.cpp.

```
#include "widget.h"
#include "ui_widget.h"
#include <QDebug>
#include <QTextStream>
#include <QFileDialog>

Widget::Widget(QWidget *parent) : QWidget(parent), ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->pbtOpen, SIGNAL(pressed()), this, SLOT(slotPbtOpenPress()));
    connect(ui->pbtWrite, SIGNAL(pressed()), this, SLOT(slotPbtWritePress()));
    connect(ui->pbtClose, SIGNAL(pressed()), this, SLOT(slotPbtClosePress()));

    mFile = new QFile();
    connect(mFile, SIGNAL(aboutToClose()), this, SLOT(slotAboutToClose()));
}

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

void Widget::slotPbtOpenPress()
{
    QString fileName;
    fileName = QFileDialog::getOpenFileName(this, "Open File",
                                           QDir::currentPath(), "Files (*.txt)");

    mFile->setFileName(fileName);
    if(!mFile->open(QIODevice::ReadWrite | QIODevice::Text)) {
        qDebug() << "File open fail.";
        return;
    }
}
```

```
}

QTextStream in(mFile);
while(in.atEnd())
    qDebug() << in.readLine();
}

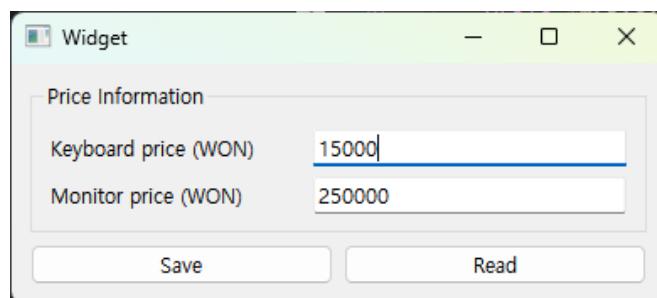
void Widget::slotPbtWritePress()
{
    QTextStream in(mFile);
    mFile->seek(mFile->size());
    in << "End.\n";
}

void Widget::slotPbtClosePress()
{
    if(mFile->isOpen())
        mFile->close();
}

void Widget::slotAboutToClose()
{
    qDebug() << Q_FUNC_INFO;
}
```

- ✓ Example using the QFile class and QDataStream

This example uses the QDataStream class. As shown in the example run screen, the keyboard price and monitor price are saved to a file by clicking the [Save] button.

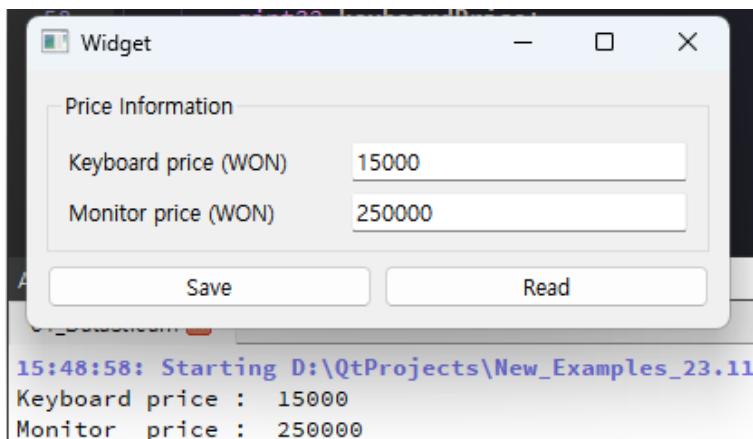


If we convert the strings of keyboard price and monitor price to numbers when saving to a file, we only need a total of 8 bytes to store each value. This is because it can be

stored using two 32-bit (4Byte) variables of type int.

In this way, when communicating data between heterogeneous systems, it is often necessary to send and receive data in binary data format according to a defined protocol. The advantage in this case is that only as much storage space is used as needed, minimizing unnecessary data waste.

For example, UART, I2S, etc. are often used in embedded environments for interprocess communication or data communication between remote devices of different types. Then, when you click the [Read Price Information] button on the GUI, the data is read from the file using QDataStream. Then, the read data is output to the Console using the qDebug() function.



See the 01\_DataStream directory for the complete source code of the example. The following example source code is for the widget.h header file.

```
#include <QWidget>
#include <QFile>

class Widget : public QWidget
{
    Q_OBJECT
public:
    explicit Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
    QFile *mFile;
```

```
private slots:  
    void slotPbtFileSave();  
    void slotPbtFileRead();  
};
```

Clicking the [Save] button calls the slotPbtFileSave( ) Slot function. Clicking the [Read] button calls the slotPbtFileRead( ) Slot function. The following is the widget.cpp source code.

```
#include "widget.h"  
#include "ui_widget.h"  
#include <QDebug>  
#include <QDataStream>  
#include <QMMessageBox>  
  
Widget::Widget(QWidget *parent) :  
    QWidget(parent), ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
  
    connect(ui->pbtSave, SIGNAL(pressed()),  
            this, SLOT(slotPbtFileSave()));  
    connect(ui->pbtFileRead, SIGNAL(pressed()),  
            this, SLOT(slotPbtFileRead()));  
  
    mFile = new QFile();  
}  
  
Widget::~Widget()  
{  
    delete ui;  
}  
  
void Widget::slotPbtFileSave()  
{  
    QString fileName;  
    fileName = QString("d:/price.data");  
  
    mFile->setFileName(fileName);  
    if(!mFile->open(QIODevice::WriteOnly | QIODevice::Truncate))  
    {  
        qDebug() << "File open fail.";
```

```
        return;
    }
else
{
    qint32 keyboardPrice = ui->leKeyboard->text().toInt();
    qint32 monitorPrice = ui->leMonitor->text().toInt();

    QDataStream out(mFile);
    out << keyboardPrice;
    out << monitorPrice;

    mFile->close();
}
}

void Widget::slotPbtFileRead()
{
    if(!mFile->open(QIODevice::ReadOnly))
    {
        qDebug() << "File open fail.";
        return;
    }
else
{
    qint32 keyboardPrice;
    qint32 monitorPrice;

    QDataStream in(mFile);
    in >> keyboardPrice;
    in >> monitorPrice;

    mFile->close();

    qDebug() << "Keyboard price : " << keyboardPrice;
    qDebug() << "Monitor price : " << monitorPrice;
}
}
```

## 15. Qt Property

The property system provided by Qt is similar to the property system provided by C++. Properties are used to set and get values from objects. For example, let's consider setting or getting a value in a particular class, as shown in the following example source code.

```
class Person : public QObject
{
    Q_OBJECT
public:
    explicit Person(QObject *parent = nullptr);

    QString getName() const {
        return m_name;
    }

    void setName(const QString &n) {
        m_name = n;
    }
private:
    QString m_name;
};
```

As shown in the example source code above, a class named Person provides the getName( ) and setName( ) member functions, which are declared as Public accessors.

The getName( ) member function returns the value of the m\_name variable. The setName( ) member function sets the value of m\_name. Declare an object of class Person and use it as follows.

```
Person goodman;

goodman.setName("Kim Dae Jin");
qDebug() << "Goodman name : " << goodman.getName();
```

We have declared a Person class object and set the value of a variable using the setName( ) member function. Then, the following code shows how to get the value declared by the setName( ) member function and output the value with the getName( ) member function.

Let's approach the method we saw in the example source code with the Property System provided by the Qt example.

```
...
class Person : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString name READ getName WRITE setName)

public:
    explicit Person(QObject *parent = nullptr);
    QString getName() const {
        return m_name;
    }

    void setName(const QString &n) {
        m_name = n;
    }

private:
    QString m_name;
};

...
```

As shown in the example source code above, we have registered the `getName()` and `setName()` member functions in the `Q_PROPERTY` macro. And you can use the member functions registered in the `Q_PROPERTY` macro as follows.

```
Person goodman;

goodman.setProperty("name", "Kim Dae Jin");
qDebug() << "Goodman name : " << goodman.getName();

QVariant myName = goodman.property("name");
qDebug() << "My name is " << myName.toString();

// [ Result ]
// Goodman name : "Kim Dae Jin"
// My name is "Kim Dae Jin"
```

You can access the `getName()` and `setName()` member functions to get or set the value of the `name` variable, but the `Q_PROPERTY` macro allows you to use `setProperty()` and

property( ) to get or set the value.

In a class, the Q\_PROPERTY macro doesn't seem very useful. However, if you are using QML, it can be very useful.

When you develop a UI in QML and develop functionality in C++, you often need to pass data back and forth between QML and C++. In this case, you can use the Q\_PROPERTY macro to get or change the value of a variable in C++ from QML.

The Q\_PROPERTY macro has a number of options available, including the following

```
Q_PROPERTY(type name  
          (READ getFunction [WRITE setFunction] |  
           MEMBER memberName [(READ getFunction |  
                               WRITE setFunction)] )  
          [RESET resetFunction]  
          [NOTIFY notifySignal]  
          [REVISION int]  
          [DESIGNABLE bool]  
          [SCRIPTABLE bool]  
          [STORED bool]  
          [USER bool]  
          [CONSTANT]  
          [FINAL])
```

The MEMBER keyword can specify a value to be read from the READ keyword. For example, in the example source code above, the m\_name variable declared in private could be specified as follows

```
class Person : public QObject  
{  
    Q_OBJECT  
    Q_PROPERTY(QString name MEMBER m_name READ getName WRITE setName)  
    ...
```

Among the keywords provided by the Q\_PROPERTY macro, the NOTIFY keyword can be specified as a signal. You can use signals as shown in the following examples.

```
class Person : public QObject  
{  
    Q_OBJECT  
    Q_PROPERTY(QString name MEMBER m_name READ getName WRITE setName  
               NOTIFY nameChanged)
```

```
public:  
    explicit Person(QObject *parent = nullptr);  
    QString getName() const {  
        return m_name;  
    }  
    void setName(const QString &n) {  
        m_name = n;  
        emit nameChanged(n);  
    }  
private:  
    QString m_name;  
  
signals:  
    void nameChanged(const QString &n);  
...
```

✓ Example using Q\_PROPERTY

In this example, we will change the value of the setName( ) member function of the Person class by using the setProperty( ) member function provided by the QObject class when the [Change] button is clicked, as shown in the figure below.



And when the setName( ) member function is called, it will fire the signal event specified by the NOTIFY keyword in the Q\_PROPERTY macro. The following example is the header source code for widget.h.

```
...  
class Widget : public QWidget  
{  
    Q_OBJECT  
public:  
    explicit Widget(QWidget *parent = nullptr);  
    ~Widget();
```

```
public slots:  
    void buttonPressed();  
    void nameChanged(const QString &n);  
  
private:  
    Ui::Widget *ui;  
    Person *goodman;  
};  
...
```

The buttonPressed( ) Slot function is called when the [Change] button is clicked.

And the nameChanged( ) Slot function is called when the nameChanged( ) signal of the Person class occurs. The following example source code is from the widget.cpp example.

```
...  
Widget::Widget(QWidget *parent) : QWidget(parent), ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
    connect(ui->pushButton, &QPushButton::pressed, this, &Widget::buttonPressed);  
  
    goodman = new Person();  
    connect(goodman, &Person::nameChanged, this, &Widget::nameChanged);  
}  
  
void Widget::buttonPressed()  
{  
    QString name = ui->leName->text();  
    goodman->setProperty("name", name);  
}  
  
void Widget::nameChanged(const QString &n)  
{  
    qDebug() << Q_FUNC_INFO << "Name Changed : " << n;  
    QVariant myName = goodman->property("name");  
    qDebug() << "My name is " << myName.toString();  
}  
  
Widget::~Widget()  
{  
    delete ui;  
}
```

...

When the nameChanged( ) Slot function is called, it uses the getName( ) member function provided by the Person class to get a value using the property( ) function provided by the QObject class. The following example source code is the header file for the Person class.

```
...
class Person : public QObject
{
    Q_OBJECT
    Q_PROPERTY(QString name MEMBER m_name READ getName
               WRITE setName NOTIFY nameChanged)
public:
    explicit Person(QObject *parent = nullptr);
    QString getName() const {
        return m_name;
    }
    void setName(const QString &n) {
        m_name = n;
        emit nameChanged(n);
    }
private:
    QString m_name;
signals:
    void nameChanged(const QString &n);
};

...
```

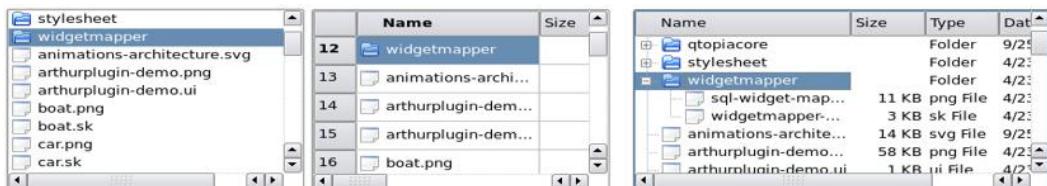
In the Person class, we can use the Q\_PROPERTY macro to access the getName( ) and setName( ) member functions. We assign the nameChanged( ) signal to the NOTIFY keyword of the Q\_PROPERTY macro.

So when the signal event is fired using emit in the setName( ) member function, the associated Slot function of the Widget class is called.

The complete source code for the example can be found in the 00\_Simple\_Property directory.

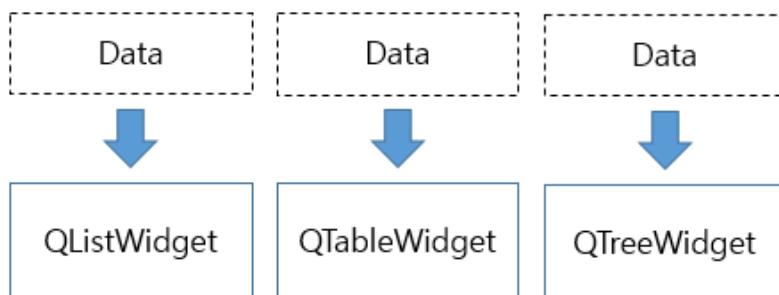
## 16. Model and View

Qt provides a number of classes for displaying data in table-like widgets, including QListWidget, QTableWidget, QTreeWidget, QListView, QTableView, QTreeView, and QColumnView, as shown in the following figure.



QListWidget has the same UI as QListView. However, QListWidget and QListView differ in the way they insert/modify/delete data.

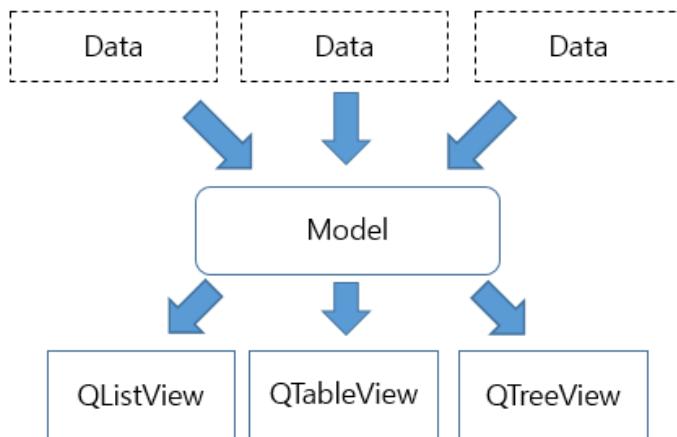
Classes that use the word Widget instead of View at the end of the class name provide member functions that allow you to insert, modify, and delete data directly, as shown in the figure below.



All widget classes with the word Widget at the end of the class name provide member functions that allow you to insert, modify, and delete data directly. For example, the QListWidget widget class allows you to insert data using the `insertItem( )` function.

However, each widget class has a slightly different usage, so you should familiarize yourself with the usage of each widget class.

Also, widget classes that use the word View at the end, such as the QListView, QTableView, and QTreeView classes, do not use their member functions to insert, modify, or delete data, but instead use the Model class to insert, modify, or delete data.

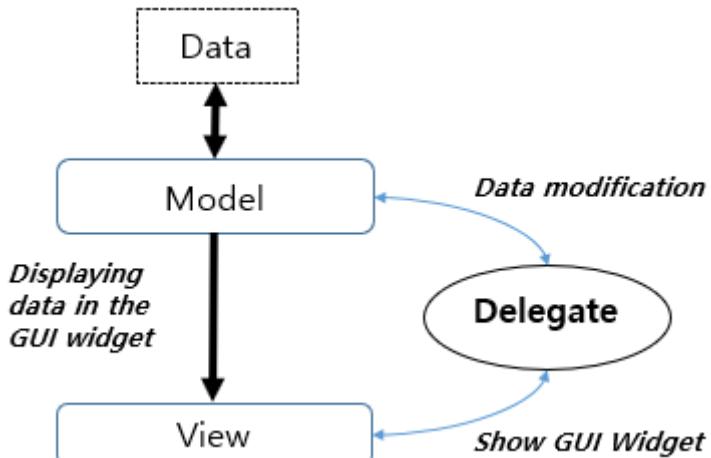


As shown in the figure above, the QListView, QTableView, and QTreeView classes use the Model class instead of using the member functions provided by each class to insert, modify, and delete data in the widget.

The advantage of using this approach is that you can use the same Model whether you are using a QListView or a QTableView.

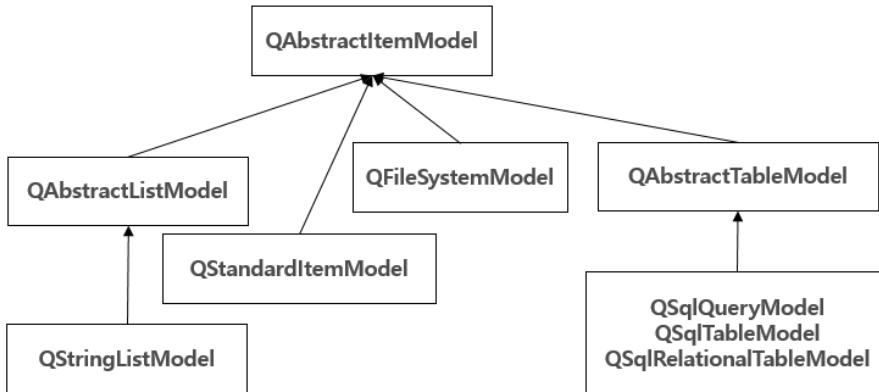
The Model/View provided by Qt can handle data using Delegates, as shown in the following figure.

For example, if you double-click a specific item of data displayed on the View widget to modify the data, an event should be fired, and the Model/View can use a Delegate to handle this event.



The Model class provides functionality for managing (inserting/modifying/deleting) data. For example, the QSqlQueryModel class provides member functions that allow you to

directly query SQL statements. Therefore, you can use the Model after editing the data imported by a separate database query, but you can insert data directly using the QSqlQueryModel class. In addition, it provides various Model classes as shown in the following figure.



The QStringListModel class provides the ability to manage simple lists of data of the QString data type.

```

QStringListModel *model = new QStringListModel();

QStringList list;
list << "Hello World" << "Qt Programming" << "Model is Good";

model->setStringList(list);
...
  
```

The QStandardItemModel class provides the ability to organize data in a tabular or tree-like fashion.

```

QStandardItemModel model(4, 4);

for (int row = 0; row < 4; ++row)
{
    for (int column = 0; column < 4; ++column) {
        QString data = QString("row %0, column %1").arg(row).arg(column);
        QStandardItem *item = new QStandardItem(data);
        model.setItem(row, column, item);
    }
}
...
  
```

The QFileSystemModel class provides the ability to manage files and directories that

originate from a file system.

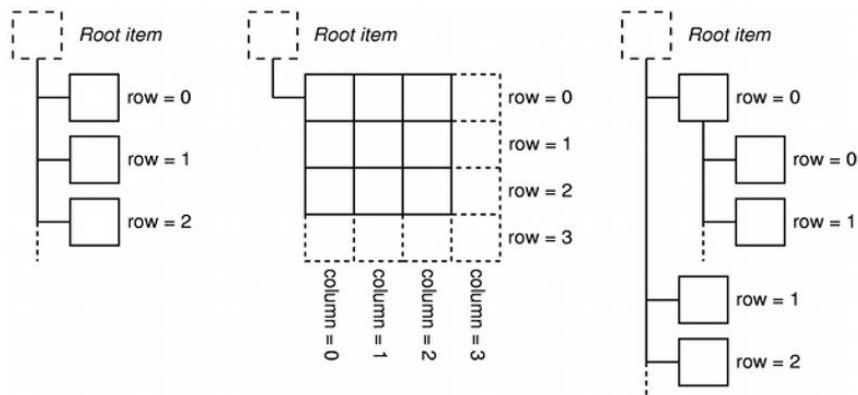
```
QFileSystemModel *model = new QFileSystemModel;  
model->setRootPath(QDir::currentPath());  
  
QTreeView *tree = new QTreeView(this);  
tree->setModel(model);  
...
```

The QSqlQueryModel class provides the ability to access data from database tables using SQL statements, and the QSqlTableModel class provides the ability to import data into the Model by passing a specific table name as an argument when importing data from the database.

For example, you can get data from a database table by entering the table name as the first argument to the setTable( ) member function.

```
QSqlQueryModel *model = new QSqlQueryModel;  
  
model->setQuery("SELECT name, salary FROM employee");  
  
model->setHeaderData(0, Qt::Horizontal, tr("Name"));  
model->setHeaderData(1, Qt::Horizontal, tr("Salary"));  
  
QTableView *view = new QTableView;  
view->setModel(model);  
view->show();  
...
```

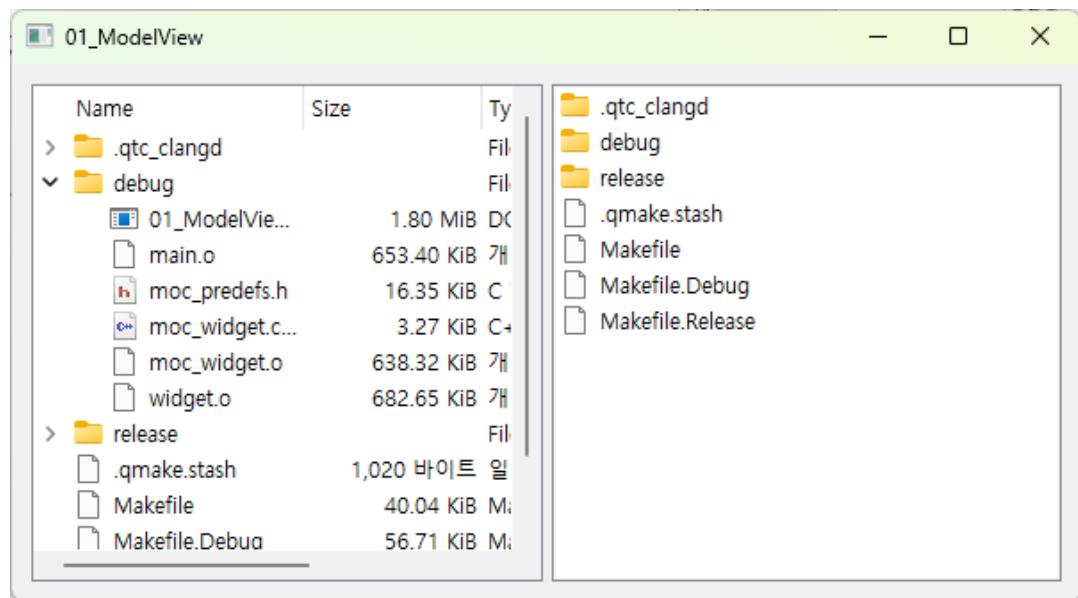
When using a Model/View to display large amounts of data, there are three types of models, as shown in the following figure.



For the leftmost data type, the `QListView` class is a good choice. If you need to display it as a table, `QTableView` is a good choice. And if you need to display it in the form of a tree, `QTreeView` is a good choice.

- ✓ Examples using the `QTreeView` and `QListView` classes

This example reads data from a file, stores it in a Model, and then connects the Model with a View to display the file system.



On the left, the `QTreeView` class was used to display data retrieved from the filesystem. The `QListView` widget on the right displays the files and directories that exist in the current directory.

```
...
Widget::Widget(QWidget *parent) : QWidget(parent)
{
    resize(600, 300);
    QSplitter *splitter = new QSplitter(this);

    QFileSystemModel *model = new QFileSystemModel;
    model->setRootPath(QDir::currentPath());

    QTreeView *tree = new QTreeView(splitter);
    tree->setModel(model);
    tree->setRootIndex(model->index(QDir::currentPath()));
```

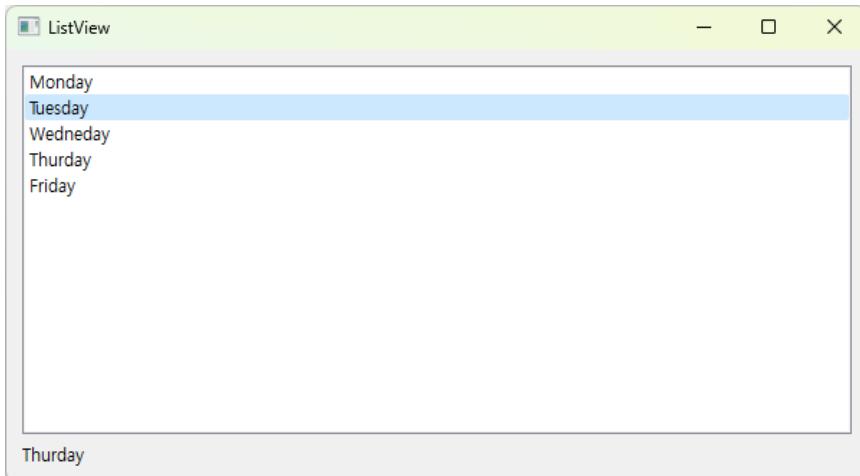
```
QListView *list = new QListView(splitter);
list->setModel(model);
list->setRootIndex(model->index(QDir::currentPath()));

QVBoxLayout *layout = new QVBoxLayout();
layout->addWidget(splitter);
setLayout(layout);
}

...
```

You can find the source code for this example in the 00\_ModelView directory.

✓ Example using the QListView class



The figure above shows an example run screen using the QListView class and the QAbstractItemModel class. The QListView is a widget that is suitable for displaying a single column and multiple lines.

```
...
resize(600, 300);
QStringList strList;

strList << "Monday" << "Tuesday" << "Wednesday" << "Thurday" << "Friday";
QAbstractItemModel *model = new QStringListModel(strList);

QListView *view = new QListView();
view->setModel(model);
```

```
QModelIndex index = model->index(3, 0);
QString text = model->data(index, Qt::DisplayRole).toString();

QLabel *lbl = new QLabel("");
lbl->setText(text);

QVBoxLayout *lay = new QVBoxLayout();
lay->addWidget(view);
lay->addWidget(lbl);

setLayout(lay);
...
```

You can refer to the source in 01\_ModelView directory.

✓ Example using the QTableView class

The QTableView class is a suitable widget for displaying tabular data, as shown in the figure below.

	Subject	Description	Date
Col 1	Monitor	LCD	2030-10-04
Col 2	CPU	Samsung	2030-10-04

```
...
QStandardItemModel *model = new QStandardItemModel(0, 3);

model->setHeaderData(0, Qt::Horizontal, QObject::tr("Subject"));
model->setHeaderData(1, Qt::Horizontal, QObject::tr("Description"));
model->setHeaderData(2, Qt::Horizontal, QObject::tr("Date"));

model->setVerticalHeaderItem(0, new QStandardItem("Col 1"));
model->setVerticalHeaderItem(1, new QStandardItem("Col 2"));
```

```
model->setData(model->index(0, 0), "Monitor");
model->setData(model->index(0, 1), "LCD");
model->setData(model->index(0, 2), QDate(2030, 10, 4));

model->setData(model->index(1, 0), "CPU");
model->setData(model->index(1, 1), "Samsung");
model->setData(model->index(1, 2), QDate(2030, 12, 5));

QTableView *table = new QTableView();
table->setModel(model);

QVBoxLayout *lay = new QVBoxLayout();
lay->addWidget(table);

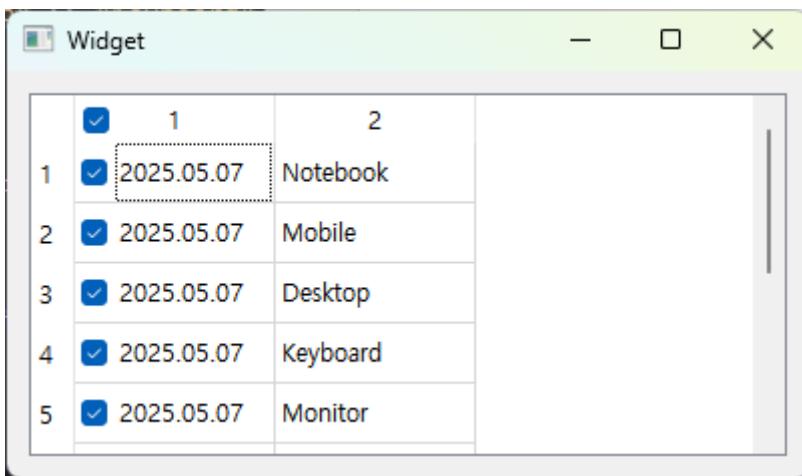
setLayout(lay);
...
```

For the full source code, see the 02\_TableModel directory.

#### ✓ QTableWidget Example

In this example, we're going to use a QTableWidget to insert data without using a Model.

Let's see how to insert a QCheckBox widget into the first Column.



As you can see in the image above, there is a checkbox in the header of the QTableWidget. By changing this checkbox, the values in the columns of all lines can be changed to be the same as the value of the checkbox in the header. And the value of each check box can be changed by the user.

Qt provides a way to customize the header of a QTableWidget to give it a desired appearance or to add specific widgets to it.

You can add a customized header by inheriting from the QHeaderView class and using the setHorizontalHeader( ) member function of the QTableWidget class to add a customized header. The following example source code is from widget.cpp.

```
...
#include "checkboxheader.h"

Widget::Widget(QWidget *parent) : QWidget(parent), ui(new Ui::Widget)
{
    ui->setupUi(this);
    ui->tableWidget->setRowCount(5);
    ui->tableWidget->setColumnCount(2);

    CheckBoxHeader* header;
    header = new CheckBoxHeader(Qt::Horizontal, ui->tableWidget);

    ui->tableWidget->setHorizontalHeader(header);
    connect(header, &CheckBoxHeader::checkBoxClicked,
            this, &Widget::checkBoxClicked);

    QStringList nameList;
    nameList << "Notebook" << "Mobile" << "Desktop" << "Keyboard" << "Monitor";

    for(int i = 0; i < 5 ; i++)
    {
        ui->tableWidget->insertRow(i);
        QTableWidgetItem *dateItem = new QTableWidgetItem("2021.05.07");
        dateItem->setCheckState(Qt::Checked);

        ui->tableWidget->setItem(i,0, dateItem);
        ui->tableWidget->setItem(i,1, new QTableWidgetItem(nameList.at(i)));
    }
}

void Widget::checkBoxClicked(bool state)
{
    for(int i = 0 ; i < 5 ; i++) {
        QTableWidgetItem *item = ui->tableWidget->item(i, 0);
        if(state)
```

```
    item->setCheckState(Qt::Checked);
else
    item->setCheckState(Qt::Unchecked);
}
...
...
```

In the example above, the CheckBoxHeader class is a class that inherits from and implements the QHeaderView class. To use this class as a header, you can use the setHorizontalHeader( ) member function.

And checkBoxClicked( ) is a Slot function associated with the event that occurs when the header's checkbox is changed. This Slot function changes the value of the checkbox in the first column to the same value based on the value of the checkbox in the header. The following example source code is the header file for the CheckBoxHeader class.

```
...
class CheckBoxHeader : public QHeaderView
{
    Q_OBJECT
public:
    CheckBoxHeader(Qt::Orientation orientation, QWidget* parent = nullptr);
    bool isChecked() const { return isChecked_; }
    void setIsChecked(bool val);

signals:
    void checkBoxClicked(bool state);

protected:
    void paintSection(QPainter* painter, const QRect& rect,
                      int logicalIndex) const;

    void mousePressEvent(QMouseEvent* event);

private:
    bool isChecked_;
    void redrawCheckBox();
};

...
```

paintSection( ) is a virtual function. When the mouse is clicked in the header area, this function is called to change the checkbox state based on the value of the checkbox. The

following example is the source code for checkboxheader.cpp.

```
#include "checkboxheader.h"

CheckBoxHeader::CheckBoxHeader(Qt::Orientation orientation, QWidget* parent)
    : QHeaderView(orientation, parent)
{
    isChecked_ = true;
}

void CheckBoxHeader::paintSection(QPainter* painter, const QRect& rect,
                                  int logicalIndex) const
{
    painter->save();
    QHeaderView::paintSection(painter, rect, logicalIndex);
    painter->restore();

    if (logicalIndex == 0) {
        QStyleOptionButton option;
        option.rect = QRect(1,3,20,20);
        option.state = QStyle::State_Enabled | QStyle::State_Active;

        if (isChecked_)
            option.state |= QStyle::State_On;
        else
            option.state |= QStyle::State_Off;

        option.state |= QStyle::State_Off;
        style()->drawPrimitive(QStyle::PE_IndicatorCheckBox, &option, painter);
    }
}

void CheckBoxHeader::mousePressEvent(QMouseEvent* event)
{
    Q_UNUSED(event)
    setIsChecked(!isChecked());

    emit checkBoxClicked(isChecked());
}

void CheckBoxHeader::redrawCheckBox()
{
    viewport()->update();
```

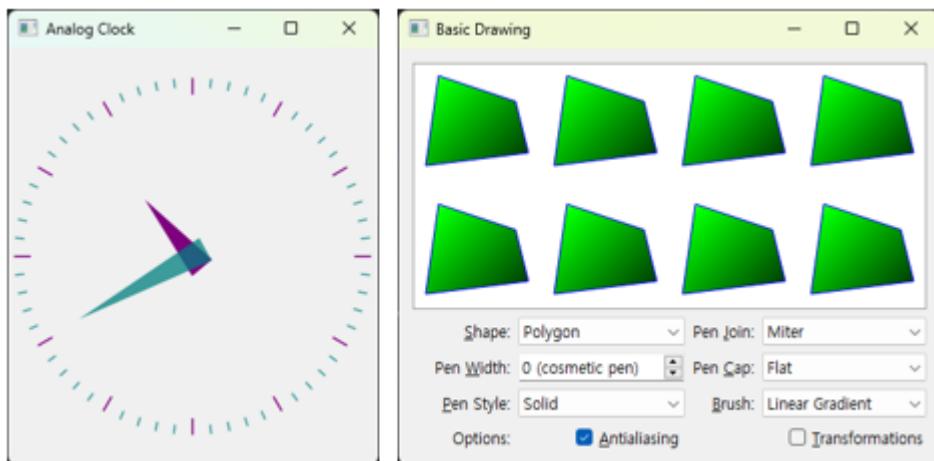
```
}
```

```
void CheckBoxHeader::setIsChecked(bool val)
{
    if (isChecked_ != val) {
        isChecked_ = val;
        redrawCheckBox();
    }
}
```

mousePressEvent( ) is a virtual function that is called when a mouse click event occurs in the header widget area. See the 03\_TableWidget directory for the complete source of examples.

## 17. 2D Graphics with QPainter Class

Qt can display text, lines, and shapes in the GUI widget area using the QPainter class. In addition to basic drawing capabilities, image files can be displayed in the widget area using the QImage, QPixmap, and QPicture classes. Effects such as Gradients, Transformation, and Composition can be applied to the QPainter area.



To use the QPainter class within the widget area, you can use the paintEvent( ) virtual function of the QWidget class, as shown in the following example.

```
...
#include <QPainter>

class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = 0);
    ~Widget();

    void paintEvent(QPaintEvent *event) override;
};

...
```

As shown in the example source code above, you can inherit from the QWidget class and use the paintEvent( ) function in your implementation. The paintEvent( ) function is

automatically called when the widget becomes visible on the monitor from being hidden, when the widget is moved, when the widget is zoomed, etc.

And if you want to fire an event that requires the widget area to be redrawn, you can use the update( ) function to automatically call the paintEvent( ) function. Note that you should not call the paintEvent( ) function directly; if you need to call the paintEvent( ) function, you should call the update( ) function.

The following example shows the source code for an example of drawing a shape in the paintEvent( ) function.

```
...
void Widget::paintEvent(QPaintEvent *event)
{
    QPainter painter;
    painter.begin(this);

    painter.setPen(Qt::blue);
    painter.drawLine(10, 10, 100, 40); // Line
    painter.drawRect(120, 10, 80, 80); // Rectangle

    QRectF rect(230.0, 10.0, 80.0, 80.0);
    painter.drawRoundedRect(rect, 20, 20); // Rounded Rectangle

    QPointF p1[3]={ QPointF(10.0, 110.0),
                    QPointF(110.0, 110.0),
                    QPointF(110.0, 190.0)};

    painter.drawPolyline(p1, 3); // Drawing line with points

    QPointF p2[3]={ QPointF(120.0, 110.0),
                    QPointF(220.0, 110.0),
                    QPointF(220.0, 190.0)};
    painter.drawPolygon(p2, 3); // Drawing shape with point

    painter.setFont(QFont("Arial", 20)); // Specifying fonts
    painter.setPen(Qt::black);
    QRect fontRect(10, 150, 220, 180); // Area to display text
    painter.drawText(fontRect, Qt::AlignCenter,
                     "Qt Korea Community(https://www.qt-dev.com)");
    painter.end();
}
```

...

In the example source code above, the begin( ) and end( ) member functions of the QPainter class do not draw the actual result of the drawing immediately. The begin( ) member function starts drawing in a virtual area, and when the end( ) member function is called, it applies the results of the drawings made between the begin( ) and end( ) functions to the actual drawing area.

For example, if the begin( ) and end( ) functions were not used, each call to the function that draws each drawing element would be applied to the actual drawing area. If this is the result of a simple drawing, the drawing is invisible to the eye.

However, if the computer system is performing complex operations, the sequential drawing may be visible. To avoid this problem, begin( ) and end( ) are used to draw the drawing into a virtual memory area, and then when the end( ) function is called, the entire drawing is copied into the actual memory area being drawn, making the drawing process invisible.



QPainter provides the ability to vary the color, thickness, and style of lines and shape outlines when drawing within an area.

```
...
QPainter painter;
painter.begin(this);

QPen pen(Qt::blue, 3, Qt::SolidLine, Qt::RoundCap, Qt::RoundJoin);

painter.setPen(pen);
```

```
QRect rect1(10.0, 20.0, 80.0, 50);
painter.drawEllipse(rect1);

pen.setStyle(Qt::DashLine);
painter.setPen(pen);
QRect rect2(110.0, 20.0, 80.0, 50.0);
painter.drawEllipse(rect2);

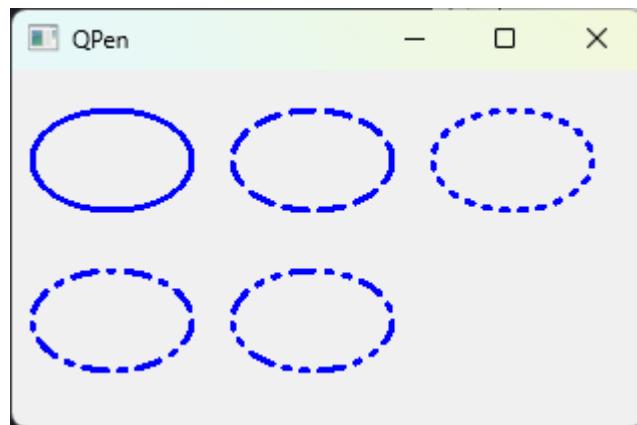
pen.setStyle(Qt::DotLine);
painter.setPen(pen);
QRect rect3(210.0, 20.0, 80.0, 50.0);
painter.drawEllipse(rect3);

pen.setStyle(Qt::DashDotLine);
painter.setPen(pen);
QRect rect4(10.0, 100.0, 80.0, 50.0);
painter.drawEllipse(rect4);

pen.setStyle(Qt::DashDotDotLine);
painter.setPen(pen);
QRect rect5(110.0, 100.0, 80.0, 50.0);
painter.drawEllipse(rect5);

pen.setStyle(Qt::CustomDashLine);
painter.setPen(pen);
QRect rect6(210.0, 100.0, 80.0, 50.0);
painter.drawEllipse(rect6);

...
```



When drawing a line, the QPainter class can specify the style of the line's corners using the setJoinStyle( ) member function of the QPen class.

```
...
QPen pen(Qt::black);
pen.setWidth(20);

QPointF p1[3] = {QPointF(30.0, 80.0),
QPointF(20.0, 40.0),
QPointF(80.0, 60.0) };

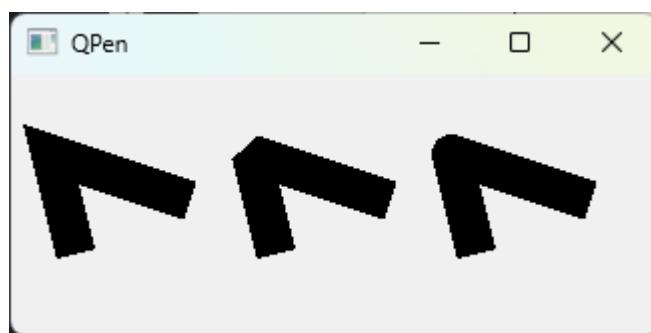
pen.setJoinStyle(Qt::BevelJoin);
painter.setPen(pen);
painter.drawPolyline(p1, 3);

QPointF p2[3] = {QPointF(130.0, 80.0),
QPointF(120.0, 40.0),
QPointF(180.0, 60.0) };

pen.setJoinStyle(Qt::MiterJoin);
painter.setPen(pen);
painter.drawPolyline(p2, 3);

QPointF p3[3] = {QPointF(230.0, 80.0),
QPointF(220.0, 40.0),
QPointF(280.0, 60.0) };

pen.setJoinStyle(Qt::RoundJoin);
painter.setPen(pen);
painter.drawPolyline(p3, 3);
...
```



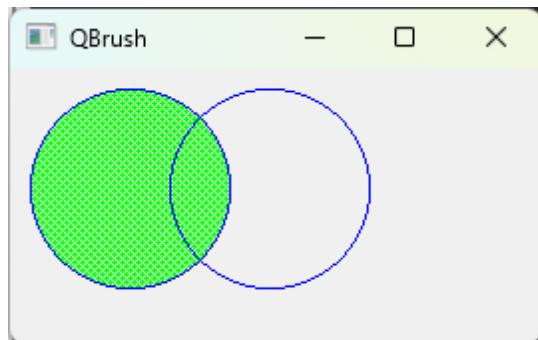
Jesus loves you.

The `setBrush( )` member function provided by the `QPainter` class allows you to fill the interior of a shape with a specific color.

```
QPen pen(Qt::blue);

painter.setBrush(QBrush(Qt::green, Qt::Dense3Pattern));
painter.setPen(Qt::blue);
painter.drawEllipse(10, 10, 100,100);

painter.setBrush(Qt::NoBrush);
painter.setPen(Qt::blue);
painter.drawEllipse(80, 10, 100, 100);
...
```



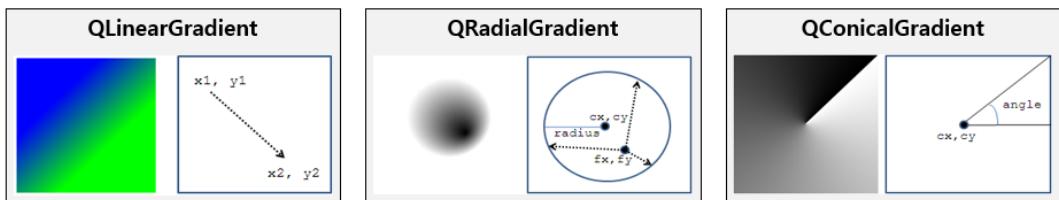
`QBrush` can fill the interior with a specific color. You can also call an image to fill the interior.

```
...
QPixmap pixmap(":resources/qtblog.png");
int w = pixmap.width();
int h = pixmap.height();
pixmap.scaled(w, h, Qt::IgnoreAspectRatio, Qt::SmoothTransformation);

QBrush brush(pixmap);
painter.setBrush(brush);
painter.setPen(Qt::blue);
painter.drawRect(0, 0, w, h);
...
```



To use the Gradients effect, you can use the Gradients effect as shown in the following image.

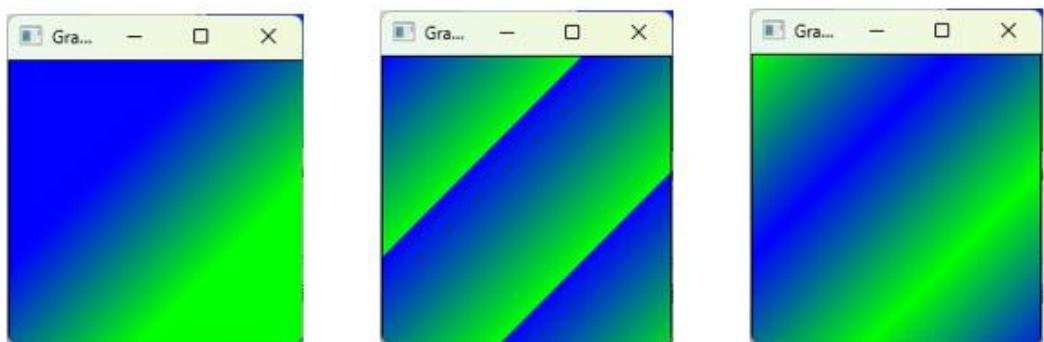


The following example source code is an example of using the Gradients effect using the `QLinearGradient` class.

```
QLinearGradient ling(QPointF(70, 70), QPoint( 140, 140 ) );
ling.setColorAt(0, Qt::blue);
ling.setColorAt(1, Qt::green);

ling.setSpread( QGradient::PadSpread );
// ling.setSpread( QGradient::RepeatSpread );
// ling.setSpread( QGradient::ReflectSpread );

QBrush brush(ling);
painter.setBrush(brush);
painter.drawRect(0, 0, 200, 200);
...
```



QGradient::PadSpread

QGradient::RepeatSpread

QGradient::ReflectSpread

Jesus loves you.

Qt can use scaling, rotation, and perspective techniques using the `QTransform` class.

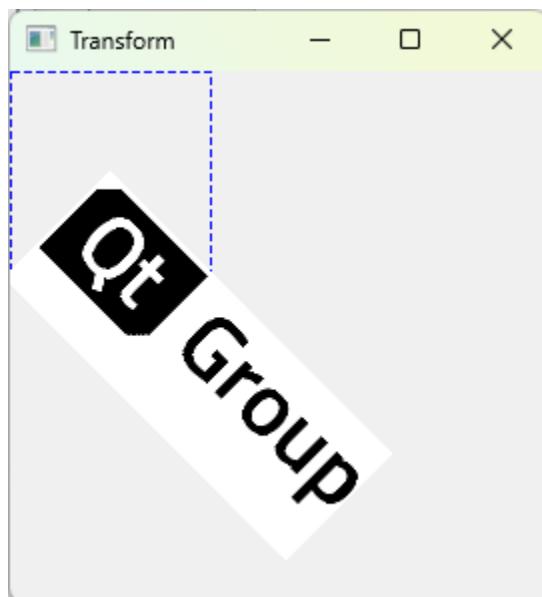


```
...
QImage image(":/resources/qtblog.png");

 QPainter painter(this);
 painter.setPen(QPen(Qt::blue, 1, Qt::DashLine));
 painter.drawRect(0, 0, 100, 100);

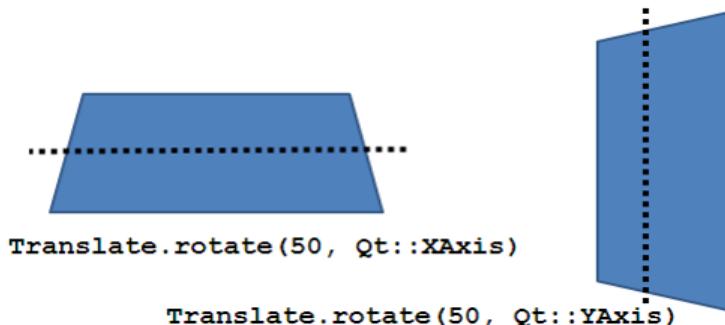
 QTransform transform;
 transform.translate(50, 50);
 transform.rotate(45);
 transform.scale(0.5, 0.5);

 painter.setTransform(transform);
 painter.drawImage(0, 0, image);
...
```



The following example is an application of the Perspective technique. As a way to apply  
Pages 195      **17. 2D Graphics with QPainter Class**

the Perspective technique, we provide the ability to pan along the X-, Y-, or Z-axis.

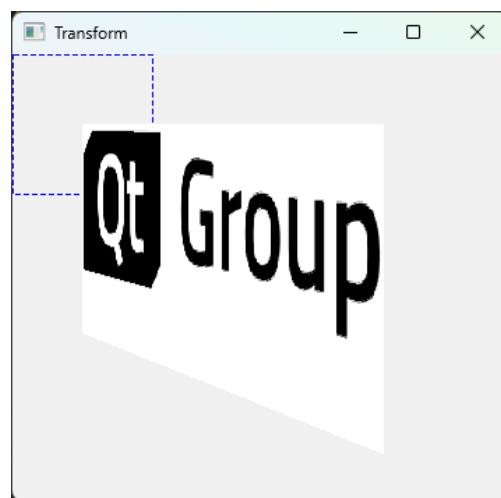


<그림> Perspective 기법

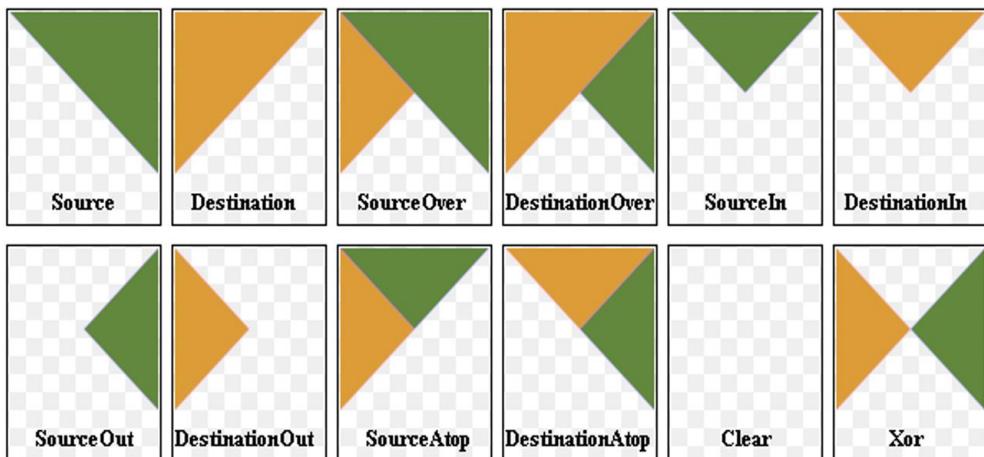
```
QPainter painter(this);
painter.setPen(QPen(Qt::blue, 1, Qt::DashLine));
painter.drawRect(0, 0, 100, 100);

QTransform transform;
transform.translate(50, 50);
transform.rotate(70, Qt::YAxis);

painter.setTransform(transform);
painter.drawImage(0, 0, image);
...
```



QImage allows you to apply patterns to nested regions using the Composition technique, as shown in the following illustration.



<그림> Composition 종류

```
...
painter.drawImage(0, 0, destinationImage);
painter.setCompositionMode(QPainter::CompositionMode_DestinationOver);
painter.drawImage(0, 0, sourceImage);
...
```

✓ Implementing the customization button

In this example, we'll use the QPainter and QWidget classes to implement a button, such as the QPushButton widget. You can implement custom widgets primarily using the QPainter class and classes for handling user input events.

So in this example, we will implement a button using the QPainter class and mouse events to handle user input. To implement the button, we need the following image. The image is attached to the example source code for reference.



Create a header file with the ImageButton class name to implement your custom button, as shown in the following example.

```
#ifndef IMAGEBUTTON_H
#define IMAGEBUTTON_H
```

```
#include <QWidget>
#include <QPainter>

class ImageButton : public QWidget
{
    Q_OBJECT
public:
    explicit ImageButton(QWidget *parent = 0);
    void setDisabled(bool val);
    void paintEvent(QPaintEvent *event) override;

private:
    QString imgFileName;
    qint32 behaviour;
    bool disabled;

signals:
    void clicked();

protected:
    virtual void enterEvent(QEnterEvent* event);
    virtual void leaveEvent(QEvent* event);
    virtual void mousePressEvent(QMouseEvent* event);
    virtual void mouseReleaseEvent(QMouseEvent* event);
    virtual void mouseDoubleClickEvent(QMouseEvent *event);
};

#endif // IMAGEBUTTON_H
```

In the example above, the `setDisabled()` member function provides the functionality for disabling the button. The `disabled` variable stores the value set in `setDisabled()`. The `paintEvent()` function changes the image of the mouse based on the mouse's events.

For example, when the mouse is positioned in the button widget area, it changes the button image to the second image as shown in the image above. And when the mouse is clicked within the widget area, it fires the `clicked()` signal. The following example is the source code for `ImageButton.cpp`.

```
#include "imagebutton.h"

#define BEHAVIOUR_NOMAL      0
#define BEHAVIOUR_ENTER      1
```

```
#define BEHAVIOUR_LEAVE      2
#define BEHAVIOUR_PRESS       3
#define BEHAVIOUR_RELEASE     4
#define BEHAVIOUR_DISABLE     5

ImageButton::ImageButton(QWidget *parent) :
    QWidget(parent),
    disabled(false)
{
    behaviour = BEHAVIOUR_NOMAL;

    QImage image(":/resources/normal.png");
    this->setFixedWidth(image.width());
    this->setFixedHeight(image.height());
}

void ImageButton::setDisabled(bool val)
{
    disabled = val;
    update();
}

void ImageButton::paintEvent(QPaintEvent *event)
{
    Q_UNUSED(event)

    QPainter painter;
    painter.begin(this);

    if(disabled == true) {
        imgFileName = QString(":/resources/disable.png");
    }
else
{
    if(this->behaviour == BEHAVIOUR_NOMAL)
        imgFileName = QString(":/resources/normal.png");
    else if(this->behaviour == BEHAVIOUR_ENTER)
        imgFileName = QString(":/resources/enter.png");
    else if(this->behaviour == BEHAVIOUR_LEAVE)
        imgFileName = QString(":/resources/normal.png");
    else if(this->behaviour == BEHAVIOUR_PRESS)
        imgFileName = QString(":/resources/press.png");
}
```

```
}

QImage image(imgFileName);
painter.drawImage(0, 0, image);
painter.end();
}

void ImageButton::enterEvent(QEnterEvent *event)
{
    Q_UNUSED(event);
    this->behaviour = BEHAVIOUR_ENTER;
    update();
}

void ImageButton::leaveEvent(QEvent *event)
{
    Q_UNUSED(event);
    this->behaviour = BEHAVIOUR_NOMAL;
    update();
}

void ImageButton::mousePressEvent(QMouseEvent *event)
{
    Q_UNUSED(event);
    this->behaviour = BEHAVIOUR_PRESS;
    update();

    emit clicked();
}

void ImageButton::mouseReleaseEvent(QMouseEvent *event)
{
    Q_UNUSED(event);

    this->behaviour = BEHAVIOUR_ENTER;
    update();
}

void ImageButton::mouseDoubleClickEvent(QMouseEvent *event)
{
    Q_UNUSED(event)
}
```

enterEvent( ) is called when the mouse is positioned in the widget area. The leaveEvent( ) function is fired when the mouse moves out of the widget area; mousePressEvent( ) is fired when the mouse is clicked inside the widget area; mouseReleaseEvent( ) is fired when the mouse button is clicked and released; and mouseDoubleClickEvent( ) is fired when the mouse button is double-clicked.

The mouse event virtual function provided by Qt uses the update( ) function. This function calls the paintEvent( ) function. Once you have implemented the ImageButton class with the above header file and source file, you can declare and use an object of the implemented ImageButton class. Let's create a header file for the Widget class, as shown in the following example source code.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>

namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

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

private:
    Ui::Widget *ui;

public slots:
    void clicked();

};

#endif // WIDGET_H
```

As shown in the example above, the clicked( ) Slot function is the function that is called when the signal event of the ImageButton class is fired. Let's write the source code file

for the Widget class like this

```
#include "widget.h"
#include "ui_widget.h"
#include <QHBoxLayout>
#include <QDebug>
#include "imagebutton.h"

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

    ImageButton *imgBtn1 = new ImageButton(this);
    ImageButton *imgBtn2 = new ImageButton(this);

    QHBoxLayout *hLay = new QHBoxLayout(this);
    hLay->addWidget(imgBtn1);
    hLay->addWidget(imgBtn2);

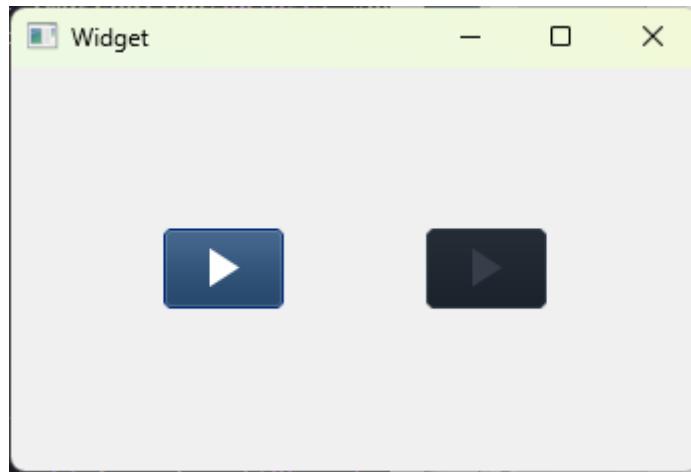
    setLayout(hLay);
    connect(imgBtn1, &ImageButton::clicked, this, &Widget::clicked);
    imgBtn2->setDisabled(true);
}

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

void Widget::clicked()
{
    qDebug() << Q_FUNC_INFO;
}
```

We use the `connect( )` function to connect the `imgBtn1` object to the Slot function when an event occurs. And to make the second `imgBtn2` disabled, we use the `setDisabled( )` function implemented in the `ImageButton` class. Let's build and run the example as shown in the figure below. The complete example source code can be found in the Ch04 > 06\_CustomButton directory.

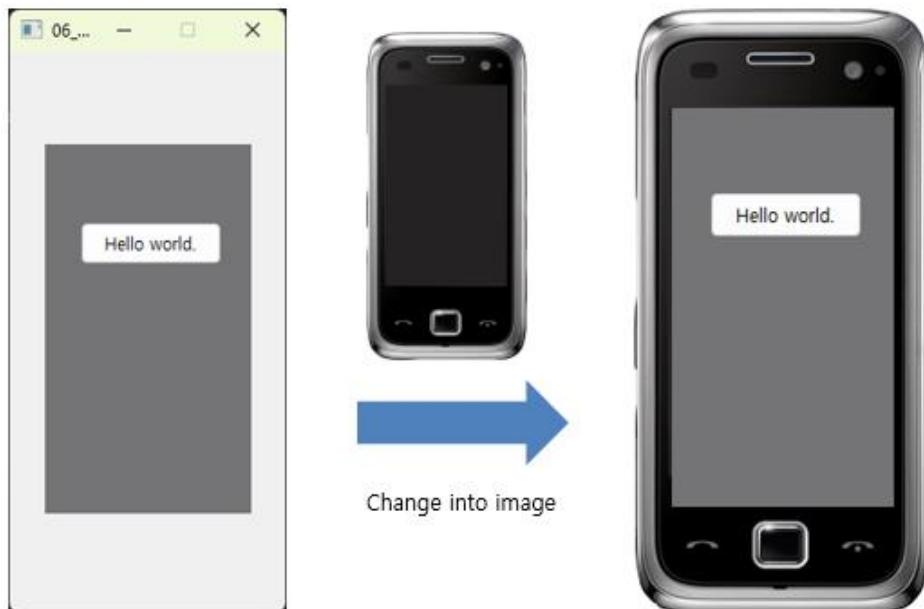
Jesus loves you.



- ✓ Implement an example to change the window to the desired appearance

In this example, we will see how to reshape a window and change the window background to a specific image.

For example, we'll see how to change the shape of the window, such as making it an elliptical window instead of a rectangular one, or rounding the corners.

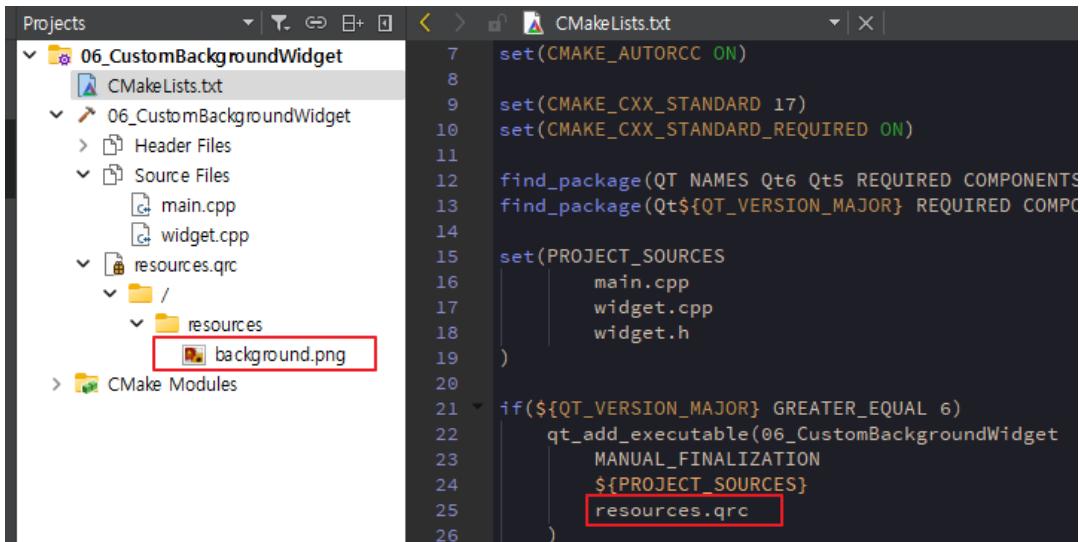


As you can see in the image above, the cell phone image is used as the background image for the window. And the corners of the smartphone image used as the background

are rounded. Therefore, the above image is used as the shape of the QWidget.

Create a QWidget-based application when you create a project. After creating the project, add the RESOURCE file to the project like below so that you can use the smartphone image.

The image to be used as the background is located in the resources directory under the 06\_CustomBackgroundWidget directory, which is the example directory, and there is a file named background.png. You can use this file.



The screenshot shows the Qt Creator interface. On the left, the Projects panel displays a project named "06\_CustomBackgroundWidget" containing a CMakeLists.txt file, Header Files, Source Files (main.cpp, widget.cpp), and a resources.qrc file which contains a resources folder with a background.png file. On the right, the code editor shows the CMakeLists.txt file with the following content:

```
7 set(CMAKE_AUTORCC ON)
8
9 set(CMAKE_CXX_STANDARD 17)
10 set(CMAKE_CXX_STANDARD_REQUIRED ON)
11
12 find_package(Qt NAMES Qt6 Qt5 REQUIRED COMPONENTS
13 find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPON
14
15 set(PROJECT_SOURCES
16   main.cpp
17   widget.cpp
18   widget.h
19 )
20
21 if(${QT_VERSION_MAJOR} GREATER_EQUAL 6)
22   qt_add_executable(@06_CustomBackgroundWidget
23     MANUAL_FINALIZATION
24     ${PROJECT_SOURCES}
25   )
26 )
```

The "background.png" file in the resources.qrc and the "resources.qrc" line in the CMakeLists.txt file are highlighted with red boxes.

Once the resource registration is complete as shown in the image above, open the widget.h header file and write it like below.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QPushButton>

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
```

```
QPixmap m_bgImage;
QPushButton *pbtHello;

private slots:
    void slot_pbtHello();

protected:
    virtual void paintEvent(QPaintEvent* event);
};

#endif // WIDGET_H
```

m\_bgImage stores an image to be used as the background. pbtHello will be used to place the QPushButton on the window.

And the paintEvent( ) function will render the Background image on the window and change the background to gray in the middle. Next, create the widget.cpp source code file as shown below.

```
#include "widget.h"
#include <QPainter>
#include <QPaintEvent>

Widget::Widget(QWidget *parent)
    : QWidget(parent, Qt::FramelessWindowHint)
{
    m_bgImage = QPixmap(":/resources/background.png", "png");

    QBitmap bitmap = m_bgImage.createHeuristicMask();
    setFixedSize(m_bgImage.size());
    setMask(bitmap);

    pbtHello = new QPushButton("Hello world.", this);
    connect(pbtHello, SIGNAL(pressed()), this, SLOT(slot_pbtHello()));

    pbtHello->setGeometry(50, 120, 100, 30);
}

void Widget::slot_pbtHello()
{
    qDebug() << Q_FUNC_INFO;
```

Jesus loves you.

```
}

void Widget::paintEvent(QPaintEvent *event)
{
    QPainter painter(this);

    painter.drawPixmap(event->rect(), m_bgImage);

    painter.setPen(QColor(116, 116, 118));
    painter.setBrush(QColor(116, 116, 118));
    painter.drawRect(25, 65, 145, 260);
}

Widget::~Widget()
{
```

The Qt::FramelessWindowHint at the top was added to remove the window title bar. We load the background image from the constructor and save it. Then we mask it to fit the size of the window. Next, we place a QPushButton on the Widget.

In the paintEvent( ) function, we render the Background image onto the Widget. We set gray as the background color of the window. Next, let's build and run it.



For the example above, you can refer to the 06\_CustomBackgroundWidget directory.

- ✓ Implementing an example of the Maintain Image Scale Ratio feature

This is an example of rendering (displaying) an image file in the widget area. In this example, the image is rendered and the size of the image is scaled up or down as the size of the widget changes.

For example, the image is scaled up or down as a percentage of the image's size relative to the widget's area size. And depending on the ratio of the image size, the area where the image is not displayed is painted black and the image is centered, as shown in the figure below.



This is an example of calculating the proportion of the image when the widget changes size and rendering it proportional to the size of the widget, as shown in the image above.

The following example source code is the `paintEvent( )` function. As we discussed earlier, the `paintEvent( )` function is called when the window size changes, so you don't need to call the `paintEvent( )` function, it will be called automatically.

```
...
void Widget::paintEvent(QPaintEvent *event)
{
    Q_UNUSED(event)

    QPainter painter;
    painter.begin(this);

    int w = this->window()->width();
    int h = this->window()->height();

    painter.setPen(QColor(0, 0, 0));
```

```

painter.fillRect(0, 0, w, h, Qt::black);

QPixmap imgPixmap = QPixmap(":/images/picture.png")
    .scaled(w, h, Qt::KeepAspectRatio);

int imgWidth = imgPixmap.width();
int imgHeight = imgPixmap.height();

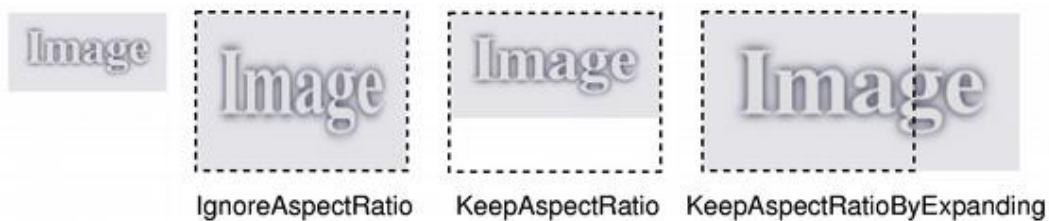
int xPos = 0;
int yPos = 0;

if(w > imgPixmap.width())
    xPos = (w - imgWidth) / 2;
else if( h > imgPixmap.height())
    yPos = (h - imgHeight) / 2;

painter.drawPixmap(xPos, yPos, imgPixmap);
painter.end();
}
...

```

QPixmap provides the ability to decode image files. The scaled( ) function of the QPixmap class takes two arguments: the first for the size (horizontal), the second for the size (vertical). The third argument is a method that determines how the image should be rendered. The third argument of the scaled( ) function can take any of the following formats, as shown in the following table.



Constant	Value	Description
Qt::IgnoreAspectRatio	0	Display full in horizontal and vertical sizes regardless of image size ratio
Qt::KeepAspectRatio	1	Maintain image proportions regardless of horizontal and vertical dimensions
Qt::KeepAspectRatioByExpanding	2	Display images at their original size regardless of aspect ratio and image size

Jesus loves you.

		ratio
--	--	-------

The drawPixmap( ) member function of the QPainter class provides the ability to display an image rendered by the QPixmap class in the QPainter area. The first and second arguments are the X and Y starting coordinates.

The third and final argument specifies an object of class QPixmap. The complete source for this example can be found in the 07\_ScaledImageRender directory.

## 18. Implementing Chromakey image processing with QPainter

In this chapter, we'll implement a simple Chromakey application using QPainter, which we covered in the previous chapter.

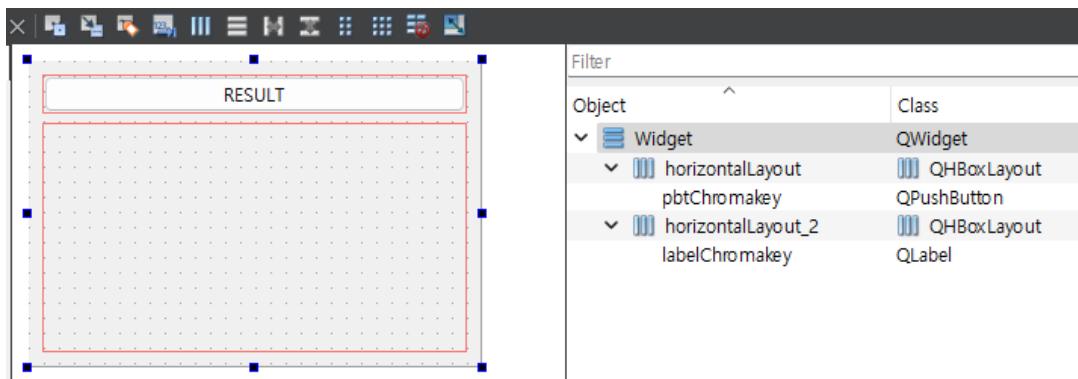
Chromakey refers to a method of filtering out certain colors and replacing them with pixels from another image. For example, you've probably seen many videos where the background of a weather newscast is replaced with another image or video. In this example, we're going to change the background of a photo image format to another image.



In the image above, the background color in the SOURCE image is green. We want to replace the background color in the SOURCE image with the TARGET image. To do this, we need a way to separate the background from the people in the SOURCE image. The background of the SOURCE image is green.

Therefore, to separate the people from the background, we need to find the threshold value of green, the background color of the SOURCE image, to separate the people from the green background. Then, change the separated background to the TARGET image to get the RESULT image.

Create a new project based on QWidget and place the widget on the GUI as follows.



Place the QPushButton and QLabel as shown in the image above. Then, add the ImageProcessing class to your project, as shown in the following example source code, and then write the header file like this

```
#ifndef IMAGEPROCESSING_H
#define IMAGEPROCESSING_H

#include <QImage>
#include <QColor>

class ImageProcessing
{
public:
    ImageProcessing(int width, int height, int dataSize);
    void chromakeyProcess(QImage& sourceImage,
                          QImage& targetImage,
                          QImage& resultImage);
private:
    int imageWidth;
    int imageHeight;
    int imageDataSize;
};

#endif // IMAGEPROCESSING_H
```

ImageProcessing 클래스의 생성자 첫 번째 와 두 번째 인자는 크로마키 처리를 할 크기를 인자로 전달한다. 그리고 세 번째 인자는 이미지 사이즈를 넘겨준다. 이미지 사이즈 크기를 세 번째 인자에 넘겨줄 때 가로 크기와 세로 크기를 곱한 값에 4를 곱해야 한다.

Because we're using RGB32, which means that RGBA uses 4 values each, and each value is 1 byte in size, we can calculate the total size as follows

가로 크기 x 세로 크기 x RGBA(4) = 이미지 크기

The unit of image size is pixels. One of the things to keep in mind when using the chromakey technique is that the size of the SOURCE image, the size of the TARGET image, and the size of the RESULT image must all be the same.

The chromakeyProcess() member function that we will implement in the ImageProcessing class will chromakey process the SOURCE and TARGET images and pass the result to the third argument. The following example source code is the implementation source code for the ImageProcessing class header.

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

ImageProcessing::ImageProcessing(int width, int height, int dataSize)
{
    this->imageWidth = width;
    this->imageHeight = height;
    this->imageDataSize = dataSize;
}

void ImageProcessing::chromakeyProcess(QImage& sourceImage,
                                         QImage& targetImage,
                                         QImage& resultImage)
{
    uchar *pSourceData = sourceImage.bits();
    uchar *pTargetData = targetImage.bits();
    uchar *pResultData = resultImage.bits();

    QColor maskColor = QColor::fromRgb(sourceImage.pixel(1,1));

    int kred    = maskColor.red();
    int kgreen  = maskColor.green();
    int kblue   = maskColor.blue();

    int sPixRed, sPixGreen, sPixBlue;

    for (int inc = 0; inc < this->imageDataSize ; inc += 4)
    {
```

```

sPixRed    = pSourceData[inc+2];
sPixGreen = pSourceData[inc+1];
sPixBlue   = pSourceData[inc];

if((abs(kred - sPixRed) + abs(kgreen - sPixGreen) +
   abs(kblue - sPixBlue)) / 5 < 22 )
{
    pResultData[inc+2] = pTargetData[inc+2];
    pResultData[inc+1] = pTargetData[inc+1];
    pResultData[inc]   = pTargetData[inc+0];
}
else
{
    pResultData[inc+2] = pSourceData[inc+2];
    pResultData[inc+1] = pSourceData[inc+1];
    pResultData[inc]   = pSourceData[inc+0];
}
}
}
}

```

We used QImage as an argument to the chromakeyProcess( ) member function. We used the QImage class because it gives us direct access to the RGBA value of each pixel in the image, even though there are many other classes such as QPixmap.

In RGBA for each pixel, R stands for Red, G for Green, B for Blue, and A for Alpha, which is the transparency value.

The QImage class provides many other formats besides RGBA. Also, depending on the RGBA format used by the QImage class, the order of the RGBAs can change from BGRA, RGBA, ABGR, etc. so it is important to check the correct order of the RGBAs according to the order of the RGBAs for each format.

In addition, QImage also provides a format that does not use the Alpha value, and it also provides a format that uses only 1 byte to display RGB. Here is the source of the header file for the Widget class.

```

#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include "imageprocessing.h"

```

```
namespace Ui {
class Widget;
}

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

private:
    ImageProcessing *imgProcess;

    QImage sourceQImage;
    QImage targetQImage;
    QImage resultQImage;

    int sourceQImageWidth;
    int sourceQImageHeight;
    int sourceQImageDataSize;

private slots:
    void slotChromakey();

private:
    Ui::Widget *ui;
};

#endif // WIDGET_H
```

The sourceQImage object of class QImage contains the SOURCE image, targetQImage contains the TARGET image, and resultQImage is the image object to store the results from the ImageProcessing class. The following example source code is an implementation of the Widget class.

```
...
#define IMGSOURCE(":/images/jana_480p.png"
#define IMGTARGET(":/images/target_480p.png"

Widget::Widget(QWidget *parent) : QWidget(parent), ui(new Ui::Widget)
{
```

Jesus loves you.

```
ui->setupUi(this);
connect(ui->pbtChromakey, SIGNAL(clicked()), this, SLOT(slotChromakey()));

sourceQImage = QImage(IMGSOURCE);
targetQImage = QImage(IMGTARGET);
resultQImage = QImage(targetQImage.width(),
                      targetQImage.height(),
                      QImage::Format_RGB32);

sourceQImageWidth = targetQImage.width();
sourceQImageHeight = targetQImage.height();
sourceQImageContentSize = targetQImage.width() * targetQImage.height() * 4;

imgProcess = new ImageProcessing(sourceQImageWidth,
                                 sourceQImageHeight,
                                 sourceQImageContentSize);
}

void Widget::slotChromakey()
{
    imgProcess->chromakeyProcess(sourceQImage, targetQImage);

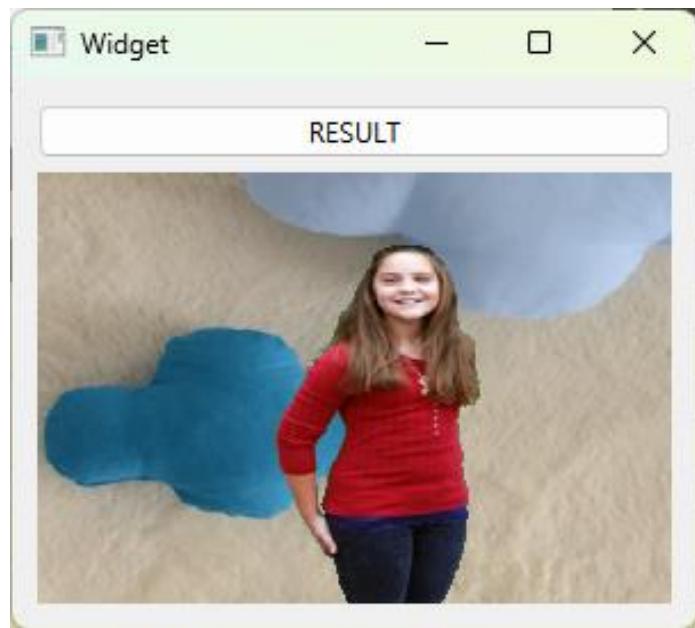
    int width = ui->labelChromakey->width();
    int height = ui->labelChromakey->height();
    QPixmap drawPixmap = QPixmap::fromImage(resultQImage).scaled(width, height);

    ui->labelChromakey->setPixmap(drawPixmap);
}

Widget::~Widget()
{
    delete ui;
}
...
```

The third argument to the QImage class specifies what kind of RGB format the QImage should use. Here we use QImage::Format\_RGB32. This format means that it uses RGBA values for each pixel. To see what kind of formats are provided by the QImage class, see the Assistant help to see the different formats available for QImage. Let's run the example we created and click the RESULT button to see the results.

Jesus loves you.



As shown in the image above, we have displayed the Result image in the QLabel widget area. For the complete source code and resources for the example, see the 00\_Chromakey directory.

## 19. Timer

The QTimer class can be called repeatedly based on a specified time. The following example source code is an example of using QTimer.

```
QTimer *timer = new QTimer(this);
connect(timer, SIGNAL(timeout()), this, SLOT(update()));

timer->start(1000);
```

As shown in the example source code above, we declare an object of class QTimer, and we need to connect the Signal and Slot functions using the connect( ) function in order to call them repeatedly at a specified time.

In the connect( ) function, the first argument specifies a QTimer object. The second specifies a Signal. The timeout( ) Signal will call the update( ) Slot function specified in the fourth argument after the specified time has elapsed.

The last line, the start( ) member function of class QTimer, repeatedly calls the Slot function specified by the connect( ) function after the time specified by the first argument has elapsed. The first argument has a unit of milliseconds.

Class QTimer can stop the timer with the stop( ) member function. If you want the timer to be called only once, rather than repeatedly, you can use the singleShot( ) member function. The singleShot( ) member function can be used as shown in the following example source code.

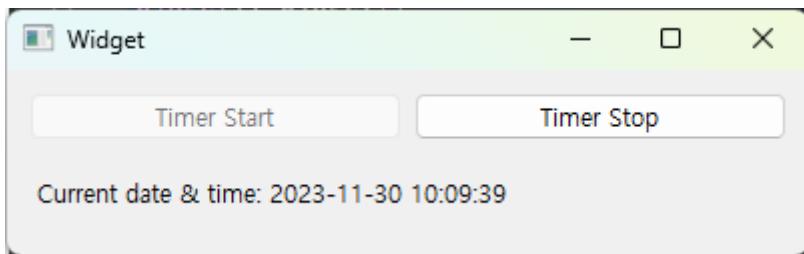
```
QTimer::singleShot(200, this, SLOT(updateCaption()));
```

The first argument to the singleShot( ) member function is the elapsed time, in milliseconds. The third argument specifies the Slot function to be called after the time in the first argument has elapsed.

- ✓ Example using the QTimer class

In the following example, the Slot function is called at 1 second intervals. Clicking the [Start Timer] button starts the timer, as shown in the following figure. Clicking the [Stop

Timer] button stops the timer.



The current time at the bottom of the QTimer example run screen is an example of how to get the current time from the system and output it to the QLabel widget every 1000 milliseconds (1 second) as specified by the QTimer. The following example source code is the widget.h source code.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QTimer>

namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

public:
    explicit Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
    QTimer *m_timer;

public slots:
    void startPressed();
    void stopPressed();
    void elapsedTime();
};


```

```
#endif // WIDGET_H
```

startPressed( ) is a Slot function that is called when the [Start Timer] button is clicked. stopPressed( ) is a Slot function that is called when the [Stop Timer] button is clicked. elapsedTime( ) is called when the time specified by the QTimer has elapsed. The following example is the widget.cpp source code.

```
#include "widget.h"
#include "ui_widget.h"
#include <QDateTime>

Widget::Widget(QWidget *parent) :
    QWidget(parent),
    ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->pbtStart, &QPushButton::pressed,
            this,           &Widget::startPressed);
    connect(ui->pbtStop,  &QPushButton::pressed,
            this,           &Widget::stopPressed);

    ui->pbtStart->setEnabled(true);
    ui->pbtStop->setEnabled(false);

    m_timer = new QTimer();
    connect(m_timer, &QTimer::timeout, this, &Widget::elapsedTime);
}

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

void Widget::startPressed()
{
    ui->pbtStart->setEnabled(false);
    ui->pbtStop->setEnabled(true);

    m_timer->start(1000);
}

void Widget::stopPressed()
```

```
{  
    ui->pbtStart->setEnabled(true);  
    ui->pbtStop->setEnabled(false);  
  
    m_timer->stop();  
}  
  
void Widget::elapsedTime()  
{  
    QDateTime curr;  
    curr = QDateTime::currentDateTime();  
  
    QString timeStr = QString(" Current date & time: %1 ")  
                    .arg(curr.toString("yyyy-MM-dd hh:mm:ss"));  
  
    ui->leCurrentTime->setText(timeStr);  
}
```

You can find the source code for this example in the 00\_Timer directory.

## 20. Thread programming

Qt provides the QThread class to support Threads. When using Threads with the QThread class, synchronization is sometimes required in certain situations. Synchronization is necessary because variables referenced by multiple users in a Thread environment may refer to incorrect values to other users when one user changes the value of the variable. Therefore, it provides a way to prevent other users from referencing or changing a variable from the start to the end of the source code of a particular function that runs as a Thread, thus preventing them from referencing the wrong value before the variable was changed.

Qt provides the QMutex class to support synchronization. The following example source code is part of a class that inherits from and implements the QThread class.

```
#include <QThread>
#include <QMutex>
#include <QDateTime>

class MyThread : public QThread
{
    Q_OBJECT
public:
    void run() override {
        while(!m_threadStop)
        {
            m_mutex.lock();
            ...
            m_mutex.unlock();
            ...
            sleep(1);
        }
    }
public:
    MyThread(int n);

private:
    bool m_threadStop;
    QMutex m_mutex;
```

```
};  
...
```

As shown in the example source code above, this is an example of implementing the MyThread class using QThread. The run( ) function is a virtual member function inherited from QThread, and you can implement the functions you want to run in the thread in this function.

Because the run( ) function is internally implemented, calling the start( ) function provided by QThread from outside the class will automatically call the run( ) function.

```
MyThread *thread = new MyThread(this);  
thread->start();
```

Sometimes you may want to declare the run( ) function as Public and call the run( ) function externally. In this case, the run( ) function does not behave like a Thread, i.e., it behaves like a regular function, not a Thread, so you should be careful not to call the run( ) function directly. The QMutex class used in the run( ) function is for synchronization.

As shown in the example source code above, the lock( ) member function provided by the QMutex class declares the synchronization to begin, and the unlock( ) member function declares that this is the last leg of the synchronization, and variables declared within this leg are inaccessible until an external class calls the unlock( ) member function.

For example, suppose your network application currently has 10 connections, and you have stored the value of the current 10 connections in an int public variable named users. If the number of current connections increases to 11 due to a new connection, incrementing the value of the users variable by 1 at the same time that you have a process that checks the current connections at the same time could give the wrong current connection information to other connections.

Therefore, if you synchronize the section of source code that increments the value of the users variable between lock( ) and unlock( ) to increment the value of the users variable by 1, the users variable will be queued to wait to be changed or referenced.

The QMutex class is useful if you need to synchronize, but it is recommended that you use it only when necessary because it can block and cause your program to freeze if you use it too often.

- ✓ Examples of Threads that satisfy Reentrancy and Thread-Safety

Reentrancy means that when two or more threads are running, regardless of the order in which the threads are executed, after one thread is executed, the next thread is executed as if it had been executed.

Thread-Safety refers to the use of mechanisms such as mutexes to ensure safety when accessing shared memory regions, such as Static or Heap memory regions, in the presence of two or more threads. The following source code is an example of creating two Threads and implementing a Thread without considering Reentrancy and Thread-Safety.

```
static QMutex mutex;
static QWaitCondition incNumber;
static int numUsed;

class Producer : public QThread
{
    Q_OBJECT
public:
    void run() override {
        for(int i = 0 ; i < 10 ; i++) {
            sleep(1);
            ++numUsed;
        }
    }
};

public:
    Producer() {}
};

class Consumer : public QThread
{
    Q_OBJECT
public:
    void run() override {
        for(int i = 0 ; i < 10 ; i++) {
            qDebug("[Consumer] numUsed : %d", numUsed);
        }
    }
};

public:
    Consumer() { }
};
```

Implement the Producer and Consumer classes as shown in the example source code above, and write the following in main.cpp

```
#include <QCoreApplication>
#include "mythread.h"

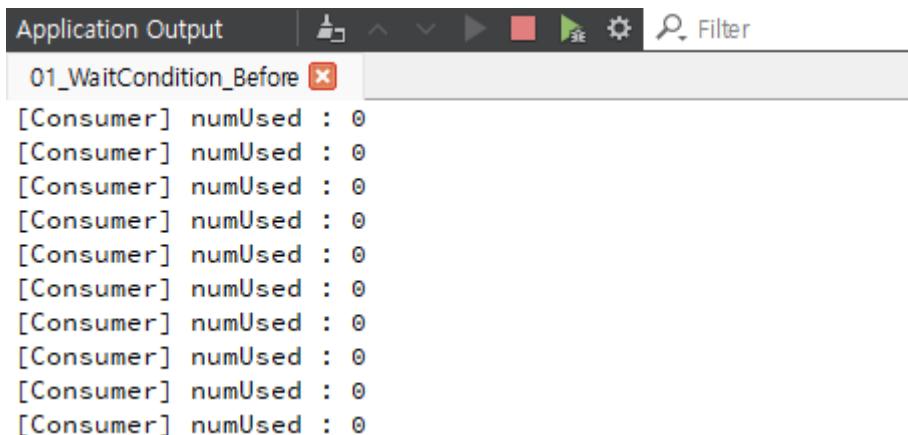
int main(int argc, char *argv[])
{
    QCoreApplication a(argc, argv);

    Producer producer;
    Consumer consumer;

    producer.start();
    consumer.start();

    return a.exec();
}
```

In the example source code above, when we declare two implemented Thread class objects and call the start( ) member function provided by QThread, it starts the Thread as shown in the following figure.



The screenshot shows the 'Application Output' window from a debugger. The title bar says 'Application Output'. Below it is a toolbar with icons for file operations, search, and filter. A tab labeled '01\_WaitCondition\_Before' is selected. The main area displays the output of the Consumer thread, which consists of ten identical lines: '[Consumer] numUsed : 0'.

```
[Consumer] numUsed : 0
```

As shown in the example run screen, the run( ) function of the Consumer class will be executed first, followed by the run( ) function of the Producer class.

The following example is an example of implementing a Thread that satisfies Reentrancy and Thread-Safety. Let's modify the Consumer and Producer classes implemented in the previous example as follows.

```
#ifndef MYTHREAD_H
#define MYTHREAD_H
#include <QWaitCondition>
#include <QMutex>
#include <QThread>

static QMutex mutex;
static QWaitCondition incNumber;
static int numUsed;

class Producer : public QThread
{
    Q_OBJECT
public:
    Producer() {}
};

class Consumer : public QThread
{
    Q_OBJECT
public:
    Consumer() {}
};

static QMutex mutex;
static QWaitCondition incNumber;
static int numUsed;

class Producer : public QThread
{
    Q_OBJECT
public:
    Producer() {}
};

class Consumer : public QThread
{
    Q_OBJECT
public:
    Consumer() {}
};

static QMutex mutex;
static QWaitCondition incNumber;
static int numUsed;

class Producer : public QThread
{
    Q_OBJECT
public:
    Producer() {}
};

class Consumer : public QThread
{
    Q_OBJECT
public:
    Consumer() {}
};

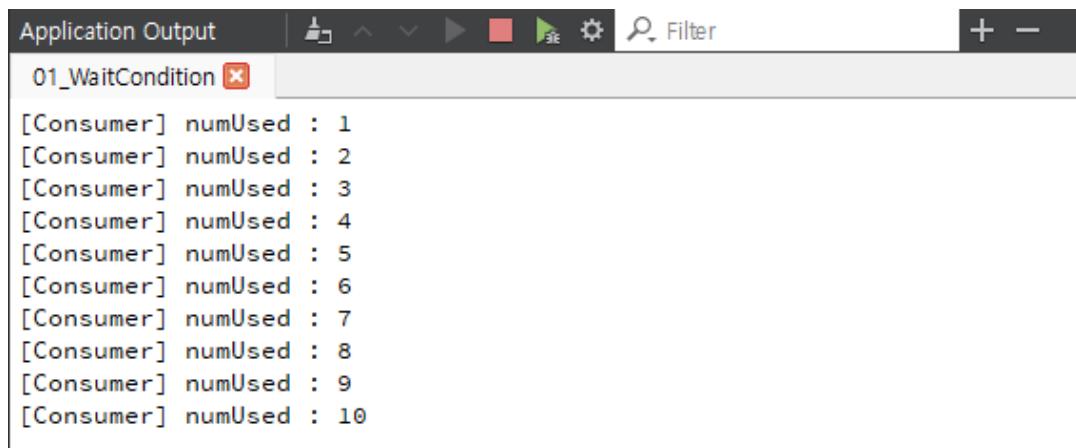
static QMutex mutex;
static QWaitCondition incNumber;
static int numUsed;
```

```
public:  
    Consumer() { }  
};  
#endif // MYTHREAD_H
```

In the Producer and Consumer classes that we wrote in the previous example source code, the threads behave regardless of the execution between the two threads. However, the modified Producer and Consumer classes are an example of implementing a Thread class that satisfies Reentrancy and Thread-Safety.

The Producer and Consumer Threads start when the value of numUsed changes, i.e., the Consumer is in the state of being waited for by the wait( ) member function of class QWaitCondition. The Producer class increments the value of numUsed by 1 and wakes up the Consumer class using the wakeAll( ) member function of the QWaitCondition class.

In this way, it is easy to implement a Thread that satisfies Reentrancy and Thread-Safety in two or more Threads.



```
Application Output | Filter  
01_WaitCondition  
[Consumer] numUsed : 1  
[Consumer] numUsed : 2  
[Consumer] numUsed : 3  
[Consumer] numUsed : 4  
[Consumer] numUsed : 5  
[Consumer] numUsed : 6  
[Consumer] numUsed : 7  
[Consumer] numUsed : 8  
[Consumer] numUsed : 9  
[Consumer] numUsed : 10
```

You can find the source code for this example in the 01\_WaitCondition directory.

✓ Implementing Threads with the QtConcurrent class

Qt에서는 QThread 를 사용해 Thread 를 구현하는 방법보다 간단하게 Multi Thread를 It provides a QtConcurrent class that can be implemented. The QtConcurrent class can make certain functions behave like Threads without writing a Thread class. The following example source code is an example source code using QtConcurrent.

```
#include <QThread>
```

```
#include <QtConcurrent/QtConcurrent>
#include <QtConcurrent/QtConcurrentRun>

void hello(QString name)
{
    qDebug() << "Hello" << name << "from" << QThread::currentThread();
    for(int i = 0 ; i < 10 ; i++)
    {
        QThread::sleep(1);
        qDebug("[%s] i = %d", name.toLocal8Bit().data(), i);
    }
}

void world(QString name)
{
    qDebug() << "World" << name << "from" << QThread::currentThread();

    for(int i = 0 ; i < 3 ; i++)
    {
        QThread::sleep(1);
        qDebug("[%s] i = %d", name.toLocal8Bit().data(), i);
    }
}

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

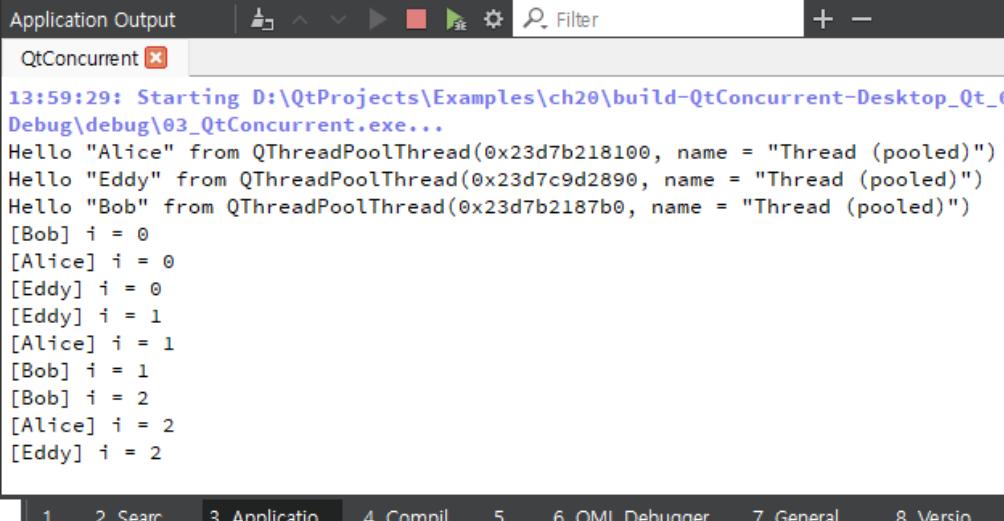
    QFuture<void> f1 = QtConcurrent::run(hello, QString("Alice"));
    QFuture<void> f2 = QtConcurrent::run(hello, QString("Bob"));
    QFuture<void> f3 = QtConcurrent::run(hello, QString("Eddy"));

    f1.waitForFinished();
    f2.waitForFinished();
    f3.waitForFinished();

    return a.exec();
}
```

As shown in the example source code above, specify the function name as the first argument to the run( ) member function of the QtConcurrent class. This allows you to run the function like a Thread.

Jesus loves you.



The screenshot shows the 'Application Output' tab in the Qt Creator interface. The title bar says 'Application Output'. Below it is a toolbar with icons for file operations, search, and settings, followed by a 'Filter' button. The main area displays the output of a program named 'QtConcurrent'. The log shows the application starting and then printing messages from three threads named Alice, Eddy, and Bob, each showing the value of variable 'i' at different points in the loop. The output is as follows:

```
13:59:29: Starting D:\QtProjects\Examples\ch20\build-QtConcurrent-Desktop_Qt_6
Debug\debug\03_QtConcurrent.exe...
Hello "Alice" from QThreadPoolThread(0x23d7b218100, name = "Thread (pooled)")
Hello "Eddy" from QThreadPoolThread(0x23d7c9d2890, name = "Thread (pooled)")
Hello "Bob" from QThreadPoolThread(0x23d7b2187b0, name = "Thread (pooled)")
[Bob] i = 0
[Alice] i = 0
[Eddy] i = 0
[Eddy] i = 1
[Alice] i = 1
[Bob] i = 1
[Bob] i = 2
[Alice] i = 2
[Eddy] i = 2
```

At the bottom of the window, there is a navigation bar with tabs labeled 1, 2, Search..., 3 Application..., 4 Compilation..., 5 ..., 6 QML Debugger..., 7 General..., and 8 Version...

The source code for this example can be found in the 02\_QtConcurrent directory.

## 21. XML

Qt provides an API to work with extensible markup language (XML). To use the XML module provided by Qt, add the following to your project file. If you are using qmake as your build tool, add the following to your project file

```
QT += xml
```

If you're using CMake, you'll need to add the following

```
find_package(Qt6 REQUIRED COMPONENTS Xml)
target_link_libraries(mytarget PRIVATE Qt6::Xml)
```

Qt provides two ways to work with XML: the Document object model (DOM) and the Simple API for XML (SAX).

The DOM stores all of the XML content in memory as a tree and then reads it in. Because it's in memory, it's easy to make edits, such as modifications and deletions.

The SAX method is easier to implement than the DOM method, and it is characterized by the fact that it does not store read data or data after analysis in memory, so it is difficult to modify or delete. Both the DOM and SAX methods are available in Qt and provide various APIs.

Qt provides the `QXmlStreamReader` class to read data in XML format and the `QXmlStreamWriter` class to write data in XML format.

The `QXmlStreamReader` and `QXmlStreamWriter` classes can be used by connecting to a `QFile` object provided by Qt.

As shown in the following example, a `QFile` object can be specified as the first argument to the `setDevice( )` member function provided by the `QXmlStreamReader` class. The following is an example of an XML-formatted data file

```
<?xml version="1.0" encoding="utf-8"?>
<students>
    <student>
        <firstName>Kim</firstName>
        <lastName>Jin Wa</lastName>
        <grade>3</grade>
```

```

</student>
<student>
    <firstName>Choi</firstName>
    <lastName>In Su</lastName>
    <grade>4</grade>
</student>
...
</students>
```

You can use the `QXmlStreamReader` class to read files stored in XML format, as shown in the example above. The following is some example source code that uses the `QXmlStreamReader` class to read XML from a file stored in the above XML format.

```

...
QFile file(QFileDialog::getOpenFileName(this,"Open"));

QXmlStreamReader xmlReader;
xmlReader.setDevice(&file);

QList<Student> students;
xmlReader.readNext();

// XML Read to the end of the file
while (!xmlReader.isEndDocument())
{
    if (xmlReader.isStartElement())
    {
        QString name = xmlReader.name().toString();
        if (name == "firstName" || name == "lastName" || name == "grade")
        {
            QMessageBox::information(this, name, xmlReader.readElementText());
        }
    }
    else if (xmlReader.isEndElement())
    {
        xmlReader.readNext();
    }
}
...
```

The first argument to the `setDevice( )` member function of the `QXmlStreamReader` class reads the file selected by the file dialog and stores the read data in a `QList` named

students.

As shown in the example above, Qt can read XML-formatted data stored in a file. You can also write XML-formatted data to a file using the `QXmlStreamWriter` class.

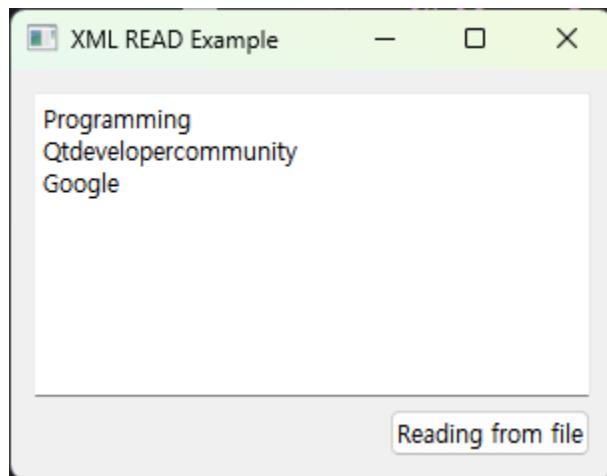
```
QXmlStreamWriter stream(&output);

stream.setAutoFormatting(true);
stream.writeStartDocument();
...
stream.writeStartElement("bookmark");
stream.writeAttribute("href", "https://www.qt-dev.com");
stream.writeTextElement("title", "Qt developer community");
stream.writeEndElement(); // bookmark
...
```

You can easily use the `QXmlStreamWriter` class as shown in the example above. Next, let's take a look at how to work with XML with a real-world example.

- ✓ Example of READING an XML file with `QXmlStreamReader`

This example shows how to read XML-formatted data stored in a file and display it on `QTextEdit`, and the example execution screen is as follows.



Click the [Read from XML file] button to select a file from the file dialog, as shown in the figure above. The selected file is an XML file, the `sample.xml` file is provided in this example directory as `sample.xml`, and the contents of the XML file are as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
```

```
<xbel version="1.0">
    <folder folded="yes">
        <title>Programming</title>
        <bookmark href="https://qt-dev.com">
            <title> Qt developer community </title>
            </bookmark>

            <bookmark href="https://www.google.com">
                <title>Google</title>
                </bookmark>
    </folder>
</xbel>
```

As shown above, this is an example of reading XML-formatted data stored in the sample.xml file and outputting it to the QTextEdit widget, and the example header file looks like this

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QFile>

namespace Ui {
class Widget;
}

class Widget : public QWidget
{
    Q_OBJECT

public:
    explicit Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
    QFile      *mReadFile;

public slots:
    void readButtonClicked();
};


```

```
#endif // WIDGET_H
```

The mReadFile object is an object of class QFile that will read the XML file, and the readButtonClicked( ) function is a Slot function that is called when the [Read from XML File] button is clicked, as shown in the example run screen. The following example source code is the widget.cpp source code.

```
#include "widget.h"
#include "ui_widget.h"
#include <QFileDialog>
#include <QXmlStreamReader>
#include <QDebug>

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

    connect(ui->pushButton, &QPushButton::pressed,
            this,           &Widget::readButtonClicked);

    mReadFile = new QFile();
}

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

void Widget::readButtonClicked()
{
    QString fName = QFileDialog::getOpenFileName(this,
                                                "Open XML File",
                                                QDir::currentPath(),
                                                "XML Files (*.xml)");

    mReadFile->setFileName(fName);

    if(!QFile::exists(fName)) {
        return;
    }
}
```

```

if(!mReadFile->open(QIODevice::ReadOnly)) {
    ui->textEdit->setText("File open failed.");
    return;
}

QXmlStreamReader reader(mReadFile);

QString inputData;
while(!reader.atEnd())
{
    reader.readNext();
    if(!reader.text().isEmpty()) {
        QString data = reader.text().toString();
        data.replace(" ", "");
        data.replace('\n', "");
        data.replace('\t', "");

        if(data.length() > 0) {
            inputData.append(data).append("<br>");
        }
    }
}

ui->textEdit->setText(inputData);
}

```

As shown in the example source code above, when the `readButtonClicked()` Slot function is called, the file dialog is loaded.

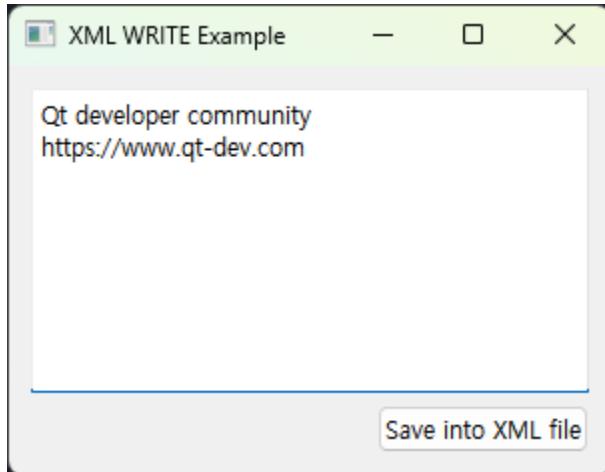
If an XML file is selected in the file dialog, the XML data is read using the `QXmlStreamReader` class. The `readNext()` member function of the `QXmlStreamReader` class reads the XML tags one after the other, and the `text()` member function reads the text of the actual tag.

The read text is then stored in a `QString` for output to `QTextEdit`. After saving, the text stored in the saved `QString` is output to `QTextEdit` after the XML file has been read. See the `00_ReaderExample` directory for the complete source of the example.

- ✓ Implementing an example of saving to an XML file using `QXmlStreamWriter`

This example shows how to save XML-formatted data to a file. As shown in the figure

below, this example reads the data output on QTextEdit, changes it to XML format data, and saves it to a file.



Clicking the [Save as XML file] button will save the XML to the "C:/output.xml" file, as shown in the example run screen above. Here is the widget.h source code file.

```
#include <QWidget>
#include <QFile>
namespace Ui { class Widget; }

class Widget : public QWidget
{
    Q_OBJECT
public:
    explicit Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget      *ui;
    QFile           *mWriteFile;
    QList<QString> mOldData;

public slots:
    void writeButtonClicked();
};
```

The mWriteFile object is the object of the QFile. mOldData is the variable that holds the data output to QTextEdit. The following example source code is from the widget.cpp source code.

```
...
#include <QFileDialog>
#include <QXmlStreamReader>
#include <QDebug>

Widget::Widget(QWidget *parent) : QWidget(parent), ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->pushButton, &QPushButton::pressed,
            this,           &Widget::writeButtonClicked);

    mWriteFile = new QFile();
    mOriData.append("Qt developer community");
    mOriData.append("https://www.qt-dev.com");

    for(int i = 0 ; i < mOriData.count() ; i++)
        ui->textEdit->append(mOriData.at(i));
}

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

void Widget::writeButtonClicked()
{
    QFile file("db:/output.xml");
    file.open(QIODevice::WriteOnly);

    QXmlStreamWriter xmlWriter(&file);
    xmlWriter.setAutoFormatting(true);
    xmlWriter.writeStartDocument();
    xmlWriter.writeStartElement("Qt");
    xmlWriter.writeStartElement("Info");
    xmlWriter.writeTextElement("Name", mOriData.at(0));
    xmlWriter.writeTextElement("URL", mOriData.at(1));
    xmlWriter.writeEndElement(); // Info End tag
    xmlWriter.writeEndElement(); // Qt End tag
    file.close();
}
...
```

In the writeButtonClicked( ) function, the writeStartElement( ) function of the

QXmlStreamWriter class is the XML start tag. The writeTextElement( ) function is where we enter the text to be stored in the tag. Finally, the writeEndElement( ) function saves the last tag to the actual file. When you close the file after the tag is created as shown above, the XML data is saved in the file "D:/output.xml" and the result is as follows.

```
<?xml version="1.0" encoding="UTF-8"?>
<Qt>
  <Info>
    <Name>Qt developer community</Name>
    <URL>https://www.qt-dev.com</URL>
  </Info>
</Qt>
```

You can find the source code for this example in the 01\_WriteExample directory.

- ✓ Implement an example DOM method

This example uses the Document object model (DOM) method to read XML-formatted data from a file. Save the file as dom.xml, as shown in the image below.

```
<?xml version="1.0" encoding="UTF-8"?>
<Building>
  <KR Name="Trade Center"></KR>
  <KR Name="Lotte Tower"></KR>
  <KR Name="Namsan Tower"></KR>
</Building>
```

To read the above XML data from a file, we can use the QDomDocument class, and the source code would look like this

```
#include <QApplication>
#include <QtXml>
#include <QtDebug>

void retrievElements(QDomElement root, QString tag, QString att)
{
    QDomNodeList nodes = root.elementsByTagName(tag);

    qDebug() << "Node counts = " << nodes.count();
    for(int i = 0; i < nodes.count(); i++)
    {
        QDomNode elm = nodes.at(i);
```

```

        if(elm.isElement())
        {
            QDomElement e = elm.toElement();
            qDebug() << "Attribute : " << e.attribute(att);
        }
    }

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

    QDomDocument document;
    QFile file(":/dom.xml");
    if(!file.open(QIODevice::ReadOnly | QIODevice::Text)) {
        qDebug() << "Failed to open file.";
        return -1;
    } else {
        if(!document.setContent(&file)) {
            qDebug() << "Failed to reading.";
            return -1;
        }
        file.close();
    }

    QDomElement root = document.firstChildElement();

    retrievElements(root, "KR", "Name");
    qDebug() << "Reading finished";

    return a.exec();
}

```

The first argument to the retrievElements( ) function is an object of class QDomElement. The second argument specifies the "KR" tag from the XML data, and the third argument specifies the "Name" attribute. The specified tags and attributes are printed to the Console using qDebug( ).

```

Node counts = 3
Attribute : "Trade Center"
Attribute : "Lotte Tower"
Attribute : "Namsan Tower"

```

Jesus loves you.

Reading finished

You can find the source code for this example in the 02\_DomExample directory.

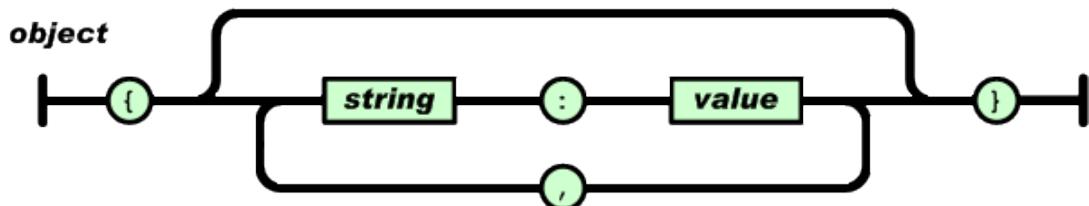
## 22. JSON

The purpose of JSON (JavaScript Object Notation) is the same as XML. The difference is that it is a lightweight way of exchanging data, which has the advantage of using less data than XML.

For example, XML wastes a lot of data to store the words needed for tags. However, JSON uses " and [ ] symbols instead of tags, so it saves space to store data compared to XML. The following is a comparison of XML and JSON data formats.

XML	JSON
<pre>&lt;?xml version="1.0" encoding="UTF-8"?&gt; &lt;note&gt;   &lt;to&gt;Tove&lt;/to&gt;   &lt;from&gt;Jani&lt;/from&gt;   &lt;heading&gt;Reminder&lt;/heading&gt; &lt;/note&gt; &lt;note&gt;   &lt;to&gt;Tove&lt;/to&gt;   &lt;from&gt;Jani&lt;/from&gt;   &lt;heading&gt;Reminder&lt;/heading&gt; &lt;/note&gt;</pre>	<pre>{   "FirstName": "John",   "Age": 43,   "Address": {     "Street": "Downing Street 10",     "Country": "Great Britain"   },   "Phone numbers": [     "+44 1234567",     "+44 2345678"   ] }</pre>

As you can see from the example above, JSON has a tentative advantage over XML in that JSON can use less data than XML when communicating data between these devices. JSON is stored as string/value pairs.



To separate string and value, we use ":" to separate string and value, with each pair separated by ",". The following example source code is JSON-formatted data.

```
{
  "FirstName": "John", "LastName": "Doe", "Age": 43,
  "Address": {
    "Street": "Downing Street 10", "City": "Seoul", "Country": "Korea"
  },
}
```

```
"Phone numbers": [ "+44 1234567", "+44 2345678" ]  
}
```

As shown in the example above, the JSON module provided by Qt accepts six data types: Bool, Double, String, Array, Object, and Null. They can be represented by starting with { (left brace) and ending with } (right brace).

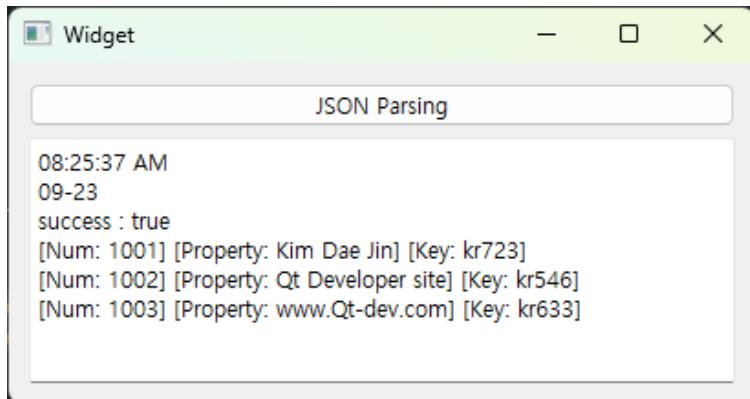
And [ (square brackets) and ] (square brackets) can be used as start and end. Additionally, braces and square brackets can be used in a nested structure. Let's see how JSON is used in a real-world example. As shown in the example above, the JSON module provided by Qt accepts six data types: Bool, Double, String, Array, Object, and Null. They can be represented by starting with { (left brace) and ending with } (right brace).

And [ (square brackets) and ] (square brackets) can be used as start and end. Additionally, braces and square brackets can be used in a nested structure. Let's see how JSON is used in a real-world example.

#### ✓ JSON Parsing Example

This example shows how to get JSON data from a Sample.json file and output it to the QPlainTextEdit widget in the GUI. The example Sample.json file contains the following contents

```
{  
    "time": "08:25:37 AM", "date": "09-23", "success": true,  
  
    "properties": [  
        { "ID": 1001, "PropertyName": "Kim Dae Jin", "key": "kr723" },  
        { "ID": 1002, "PropertyName": "Qt Developer site", "key": "kr546" },  
        { "ID": 1003, "PropertyName": "www.Qt-dev.com", "key": "kr633" }  
    ]  
}
```



Clicking the [JSON PARSING] button will output the result to the QPlainTextEdit widget in the GUI, as shown in the image above. The following is the `widget.h` source code for this example.

```
#ifndef WIDGET_H
#define WIDGET_H
#include <QWidget>

namespace Ui { class Widget; }

class Widget : public QWidget
{
    Q_OBJECT

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

private:
    Ui::Widget *ui;
    void parseJSON(const QString &data);
    void addText(const QString &addLine);

private slots:
    void slotPbtJSONParser();
};

#endif // WIDGET_H
```

In the example above, the `slotPbtJSONParser()` Slot function is the Slot function that is called when the [JSON PARSING] button is clicked. The `parseJSON()` function parses the

data read from the JSON file. The addText( ) function is for outputting the data extracted by the parseJSON( ) function to the widget. The following is the widget.cpp source code.

```
...
Widget::Widget(QWidget *parent) : QWidget(parent), ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->pbtJSONParser, SIGNAL(pressed()),
            this,                      SLOT(slotPbtJSONParser()));
}

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

void Widget::slotPbtJSONParser()
{
    QString inputFilePath;
    InputFilePath = QFileDialog::getOpenFileName(this,
                                                tr("Open File"),
                                                QDir::currentPath(),
                                                tr("JSON Files (*.json)"));

    QFile file(inputFilePath);
    if(!file.open(QIODevice::ReadOnly | QIODevice::Text)) {
        qDebug() << "Failed to open file.";
        return;
    }

    QString data = file.readAll();
    file.close();

    ui->textEdit->clear();
    parseJSON(data);
}

void Widget::parseJSON(const QString &data)
{
    QJsonDocument jsonResponse = QJsonDocument::fromJson(data.toLocal8Bit());
    QJsonObject jsonObj = jsonResponse.object();
```

```

addText(jsonObj["time"].toString().append("\n"));
addText(jsonObj["date"].toString().append("\n"));

if(jsonObj["success"].toBool() == true)
    addText(QString("success : true \n"));
else
    addText(QString("success : false \n"));

QJsonArray jsonArray = jsonObj["properties"].toArray();
foreach (const QJsonValue & value, jsonArray)
{
    QJsonObject obj = value.toObject();
    QString property = obj["PropertyName"].toString();
    QString key      = obj["key"].toString();

    QString arrayData; = QString("property : %1 , key : %2 \n")
                        .arg(property).arg(key);
    addText(arrayData);
}
}

void Widget::addText(const QString &addLine)
{
    ui->textEdit->insertPlainText(addLine);
}

```

The first argument to the parseJSON( ) function is the original JSON data read from the JSON file. The original data read is used by the jsonResopnse object of the QJsonDocument.

In the QJsonDocument::fromJson( ) function, the first argument is a QByteArray. Therefore, to replace the QString source data with a QByteArray, we use the toLocal8Bit( ) member function of class QByteArray.

The jsonResopnse object of class QJsonDocument stores information about the JSON source data processed by the fromJson( ) function. To extract the stored data using the QJsonObject class, we pass the QJsonObject type to the object( ) member function of the QJsonDocument class so that we can extract the data.

Then, to access the actual data, we pass the QJsonObject class type back to QJsonArray and use the QJsonValue class as used in the foreach statement to access the individual

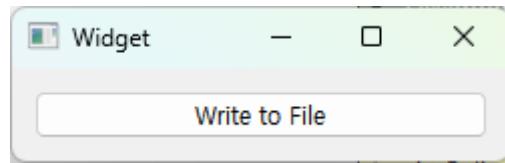
data. For example source code and JSON sample files, see the 00\_JSON\_Parser directory.

✓ Example of writing a JSON-formatted file

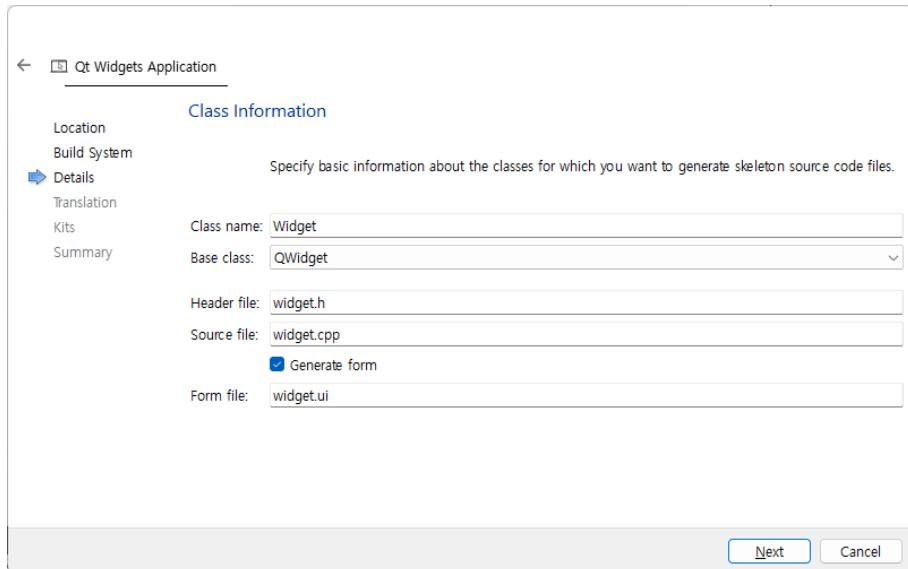
In this example, we'll write an example that saves the JSON-formatted content to a file, as shown below.

```
[  
  {  
    "Address": {  
      "City": "Yongin",  
      "Nation": "South Korea"  
    },  
    "FirstName": "Dae Jin",  
    "LastName": "Kim",  
    "School": {  
      "Grade": 3,  
      "Major": "Computer science"  
    }  
  },  
  {  
    "Address": {  
      "City": "Suwon",  
      "Nation": "South Korea"  
    },  
    "FirstName": "Gil Dong",  
    "LastName": "Hong",  
    "School": {  
      "Grade": 2,  
      "Major": "Math"  
    }  
  }  
]
```

It will save the contents of the JSON format to a file as shown above. The application you will create will save the contents of the JSON format above to a file by clicking the [Write to File] button on the run screen below.



When creating a project, a form file is created in the dialog where you enter Class Information. You can do this by checking the [Generate form] checkbox.



After creating the project, write the following in the widget.h file

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();
}
```

```
private:  
    Ui::Widget *ui;  
  
    typedef struct _tMember {  
        QString firstName;  
        QString lastName;  
  
        QString addr_nation;  
        QString addr_city;  
  
        int school_grade;  
        QString school_major;  
    } tMember;  
  
    QList<tMember> m_memList;  
  
    void writeJSON();  
  
private slots:  
    void slot_pbtWriteFile();  
};  
#endif // WIDGET_H
```

The tMember structure is a temporary storage structure for data to be stored in JSON. The writeJSON( ) member function is a function that implements the function to save the contents in JSON format. The Slot\_pbtWriteFile( ) function is a Slot function that is called when the [Write to File] button is clicked. The source code below is widget.cpp.

```
#include "widget.h"  
#include "ui_widget.h"  
  
#include <QJsonDocument>  
#include <QJsonArray>  
#include <QJsonObject>  
#include <QFile>  
#include <QDebug>  
  
Widget::Widget(QWidget *parent)  
    : QWidget(parent)  
    , ui(new Ui::Widget)
```

```
{  
    ui->setupUi(this);  
    connect(ui->pbtWirteFile, &QPushButton::clicked,  
            this, &Widget::slot_pbtWriteFile);  
}  
  
Widget::~Widget()  
{  
    delete ui;  
}  
  
void Widget::writeJSON()  
{  
    qDebug() << Q_FUNC_INFO;  
  
    QFile file("d:/write_sample.json");  
  
    if(!file.open(QIODevice::ReadWrite)) {  
        qDebug() << "File open error";  
        return;  
    }  
  
    file.resize(0);  
  
    tMember member1;  
    member1.firstName = QString("Dae Jin");  
    member1.lastName = QString("Kim");  
    member1.addr_nation = QString("South Korea");  
    member1.addr_city = QString("Yongin");  
    member1.school_grade = 3;  
    member1.school_major = QString("Computer science");  
  
    tMember member2;  
    member2.firstName = QString("Gil Dong");  
    member2.lastName = QString("Hong");  
    member2.addr_nation = QString("South Korea");  
    member2.addr_city = QString("Suwon");  
    member2.school_grade = 2;  
    member2.school_major = QString("Math");  
  
    m_memList.append(member1);  
    m_memList.append(member2);
```

```
QJsonArray jsonArray;

for(qsizetype i = 0 ; i < m_memList.size() ; i++)
{
    QJsonObject jsonObject;
    jsonObject.insert("FirstName", m_memList.at(i).firstName);
    jsonObject.insert("LastName", m_memList.at(i).lastName);

    QJsonObject jsonAddrObject;
    jsonAddrObject.insert("Nation", m_memList.at(i).addr_nation);
    jsonAddrObject.insert("City", m_memList.at(i).addr_city);

    QJsonObject jsonSchoolObject;
    jsonSchoolObject.insert("Grade", m_memList.at(i).school_grade);
    jsonSchoolObject.insert("Major", m_memList.at(i).school_major);

    jsonObject.insert("School", jsonSchoolObject);
    jsonObject.insert("Address", jsonAddrObject);

    jsonArray.append(jsonObject);
}

QJsonDocument jsonDocment;

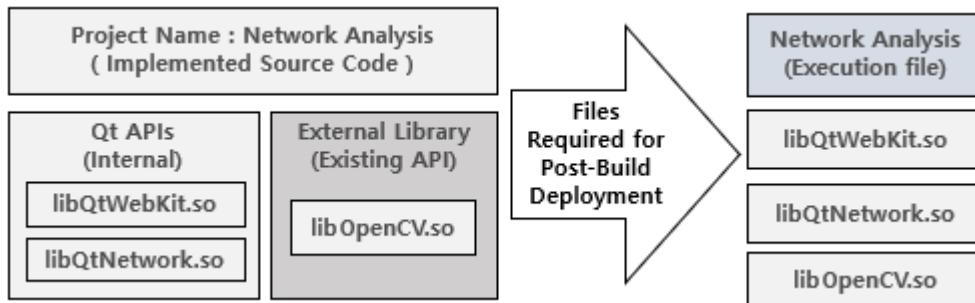
jsonDocment.setArray(jsonArray);
file.write(jsonDocment.toJson());
file.close();

}

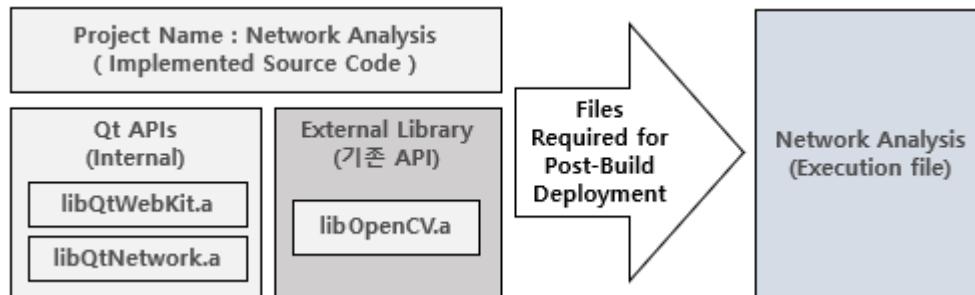
void Widget::slot_pbtWriteFile()
{
    writeJSON();
}
```

## 23. Creating library (CMake)

Qt can use shared libraries and static libraries. Shared libraries are executable and library files separated, as shown in the figure below.



With the Static library method, the libraries are combined into a single file with the executable at build time. So if you build with the Static method, you only need to deploy the executable when you deploy.



One thing to note when using external libraries in Qt is that they must be built using the same compiler that compiled them.

For example, if your application is built with a MinGW-based compiler, the external library must also use a library built with a MinGW-based compiler.

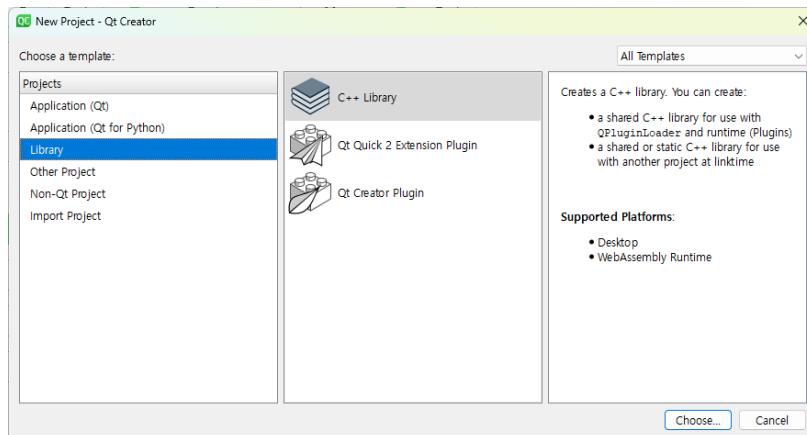
In this chapter, you will create your own shared libraries and then work through examples of using the libraries you have implemented.

## 23.1. Creating and Using Shared Libraries

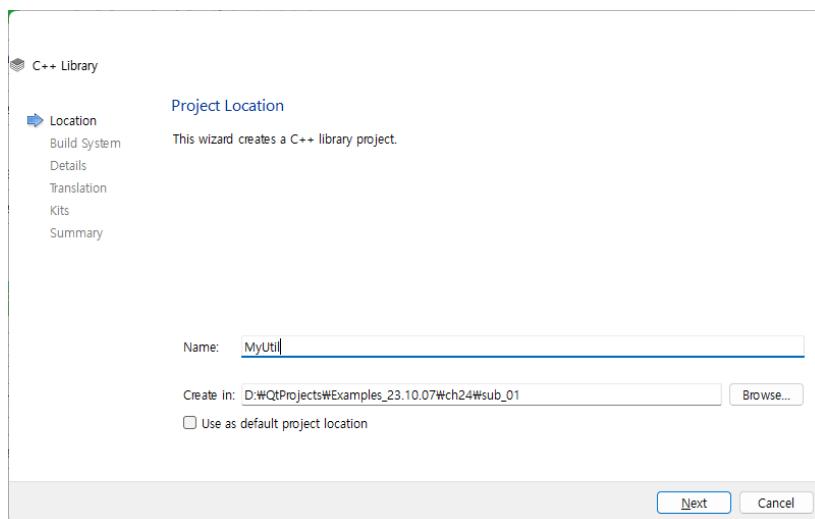
In this chapter, you will create two example projects. The first is a library project. In the library project, you will implement a library, and the second project will be a project that uses the library you implemented.

- ✓ Library implementation example

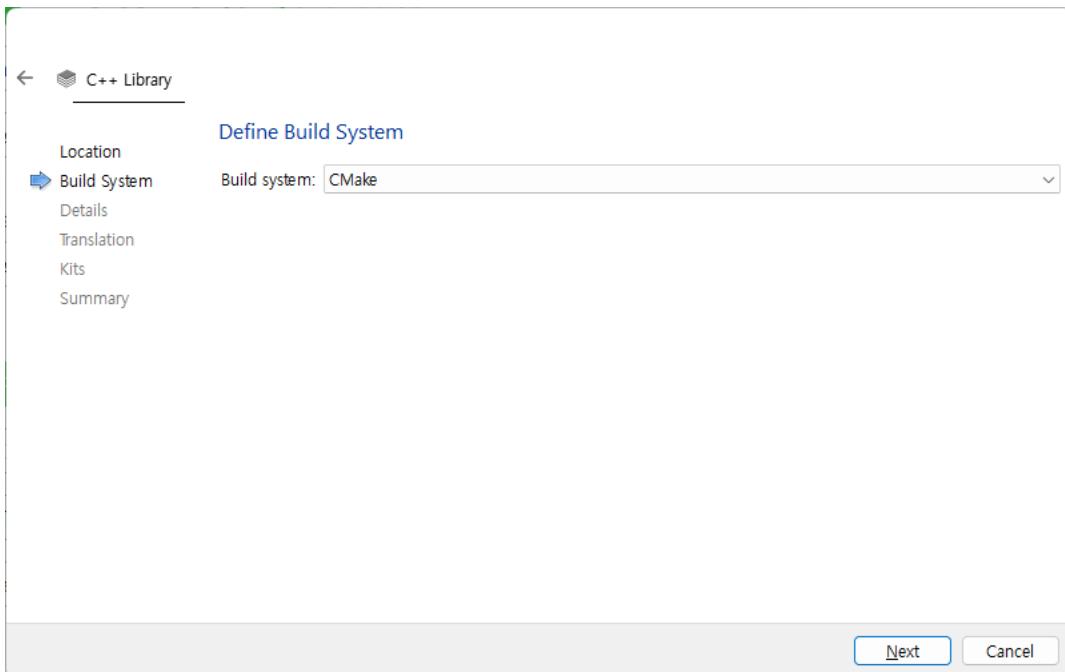
Create a project in Qt Creator. When the project creation dialog is generated, select [Library] from the left tab of the dialog window and [C++ Library] from the middle tab.



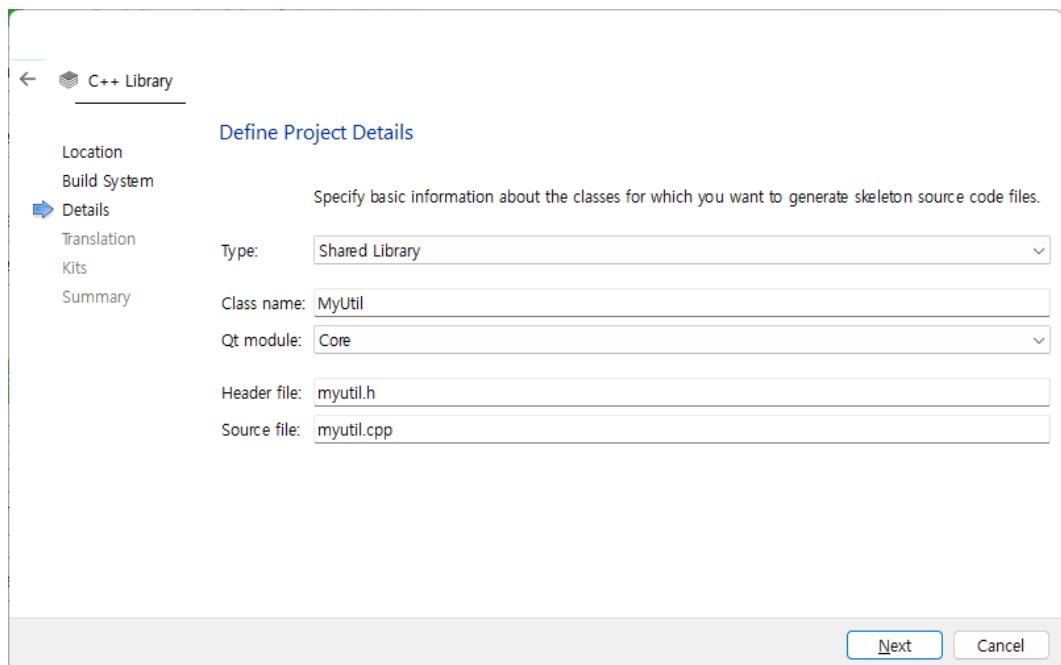
In the next dialog, enter "MyUtil" as the project name and click the [Next] button at the bottom.



Select CMake as the build system.

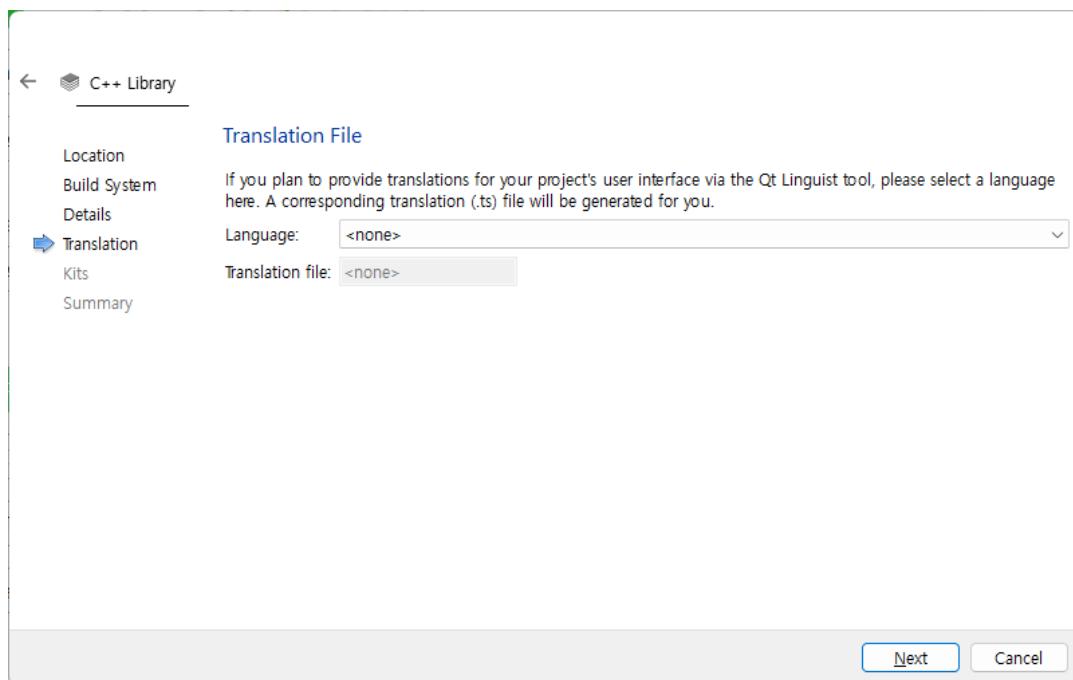


Select Shared Library as the Type and enter "MyUtil" as the Class name. For Qt Module, select "Core". Enter "myutil.h" as the Header file name and "myutil.cpp" as the Source file name.

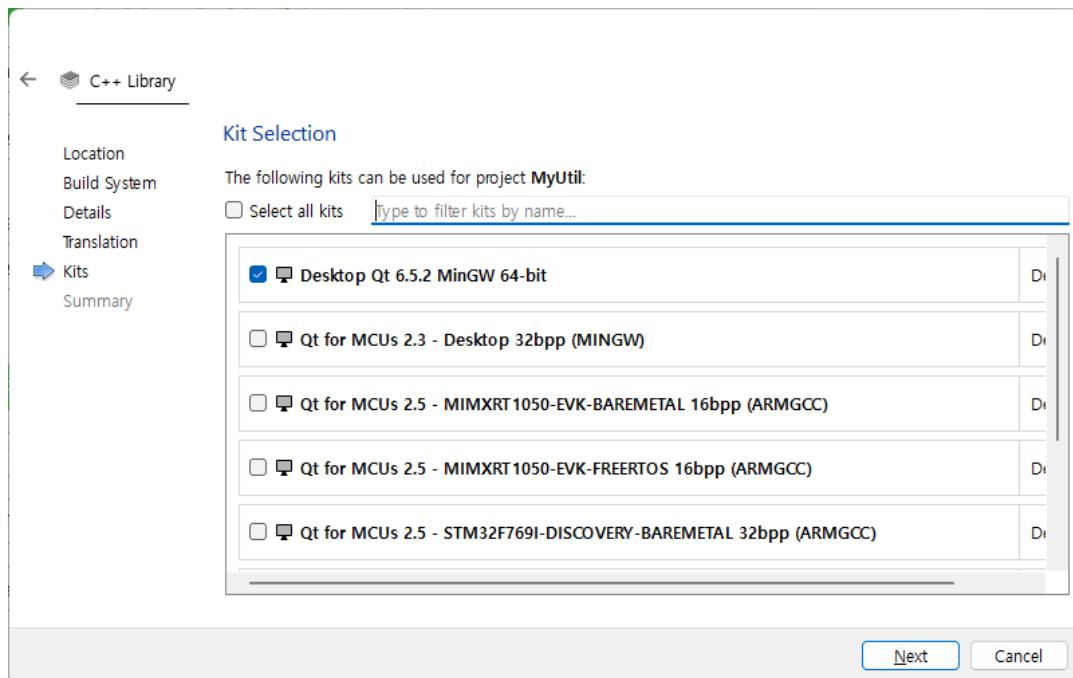


In the next dialog, leave the defaults and click the [Next] button.

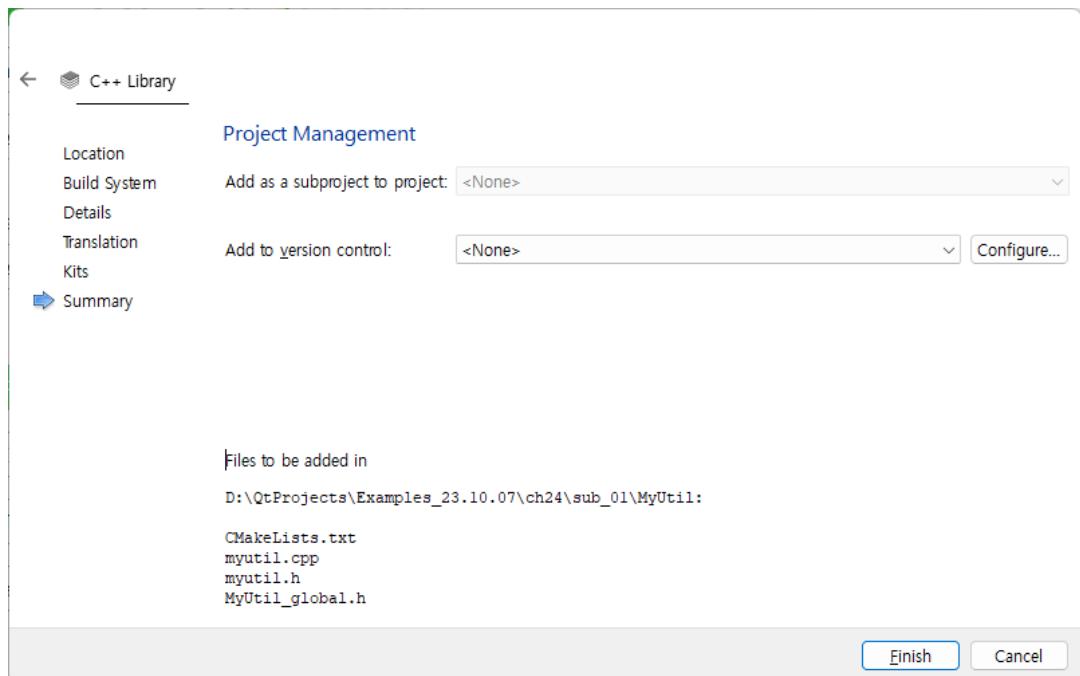
Jesus loves you.



In the Kit Selection dialog, select MinGW and click the [Next] button at the bottom.



The next dialog is a dialog to select the source code version control system. Leave it as default as shown in the figure below and click the [Finish] button.



Next, in the myutil.h header file, write the source code as shown below.

```
#ifndef MYUTIL_H
#define MYUTIL_H

#include <QObject>
#include "MyUtil_global.h"

class MYUTIL_EXPORT MyUtil : public QObject
{
public:
    explicit MyUtil(QObject *parent = nullptr);
    int getSumValue(int a, int b);

};

#endif // MYUTIL_H
```

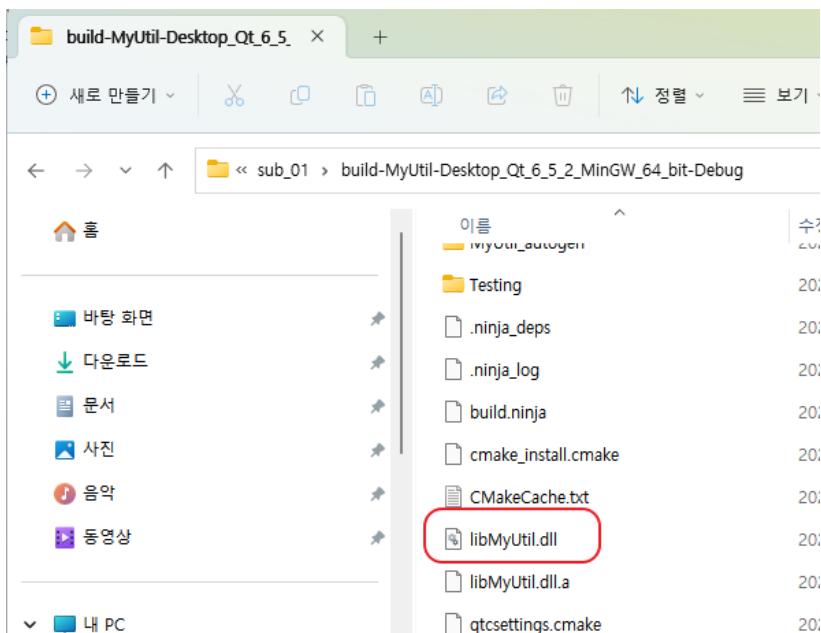
Then open myutil.cpp and write the source code like below.

```
#include "myutil.h"
```

```
MyUtil::MyUtil(QObject *parent) : QObject(parent)
{
}

int MyUtil::getSumValue(int a, int b)
{
    return a + b;
}
```

The MyUtil class simply passes two values as function arguments and returns the sum of the two values as the result. Once you've gotten this far, you can build it and then go to the directory where it was built to see that the library files have been created.

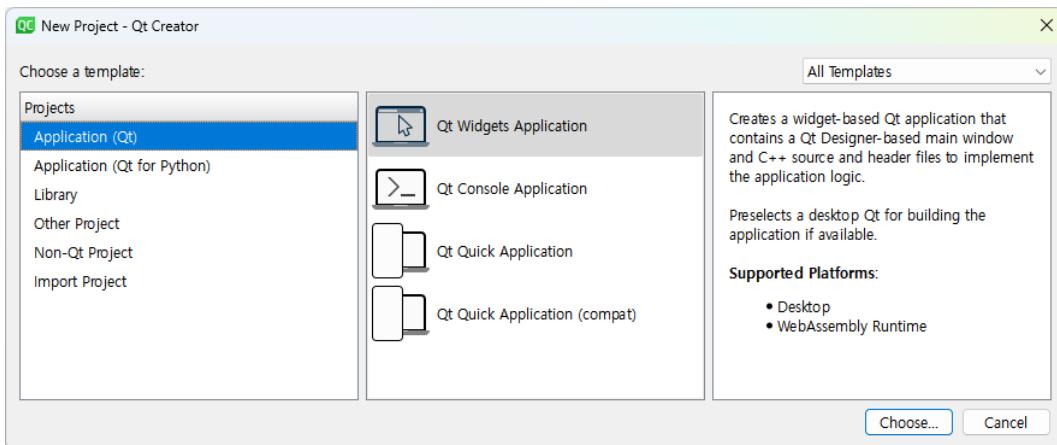


Once the library has been successfully created as shown in the image above, let's write an example that uses it.

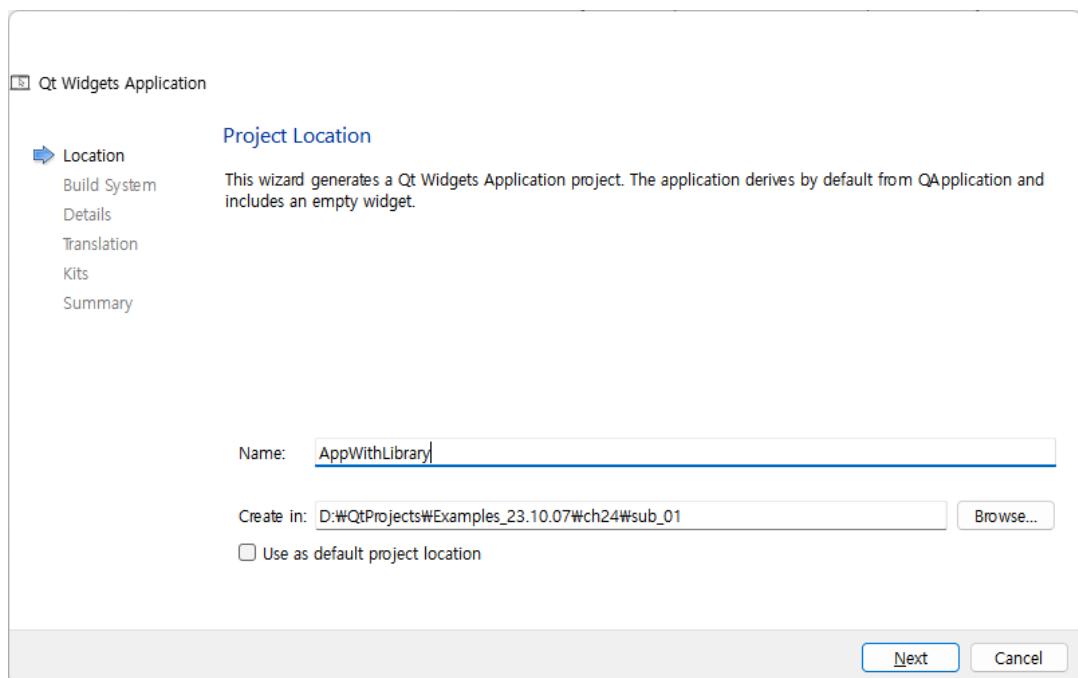
- ✓ Example of using the library created by

Let's write an example that uses the `getSumValue()` function of the MyUtil class we created in the library to return the result value when we enter two values and press the [Result] button, as shown in the image below. When creating the project, select [Qt

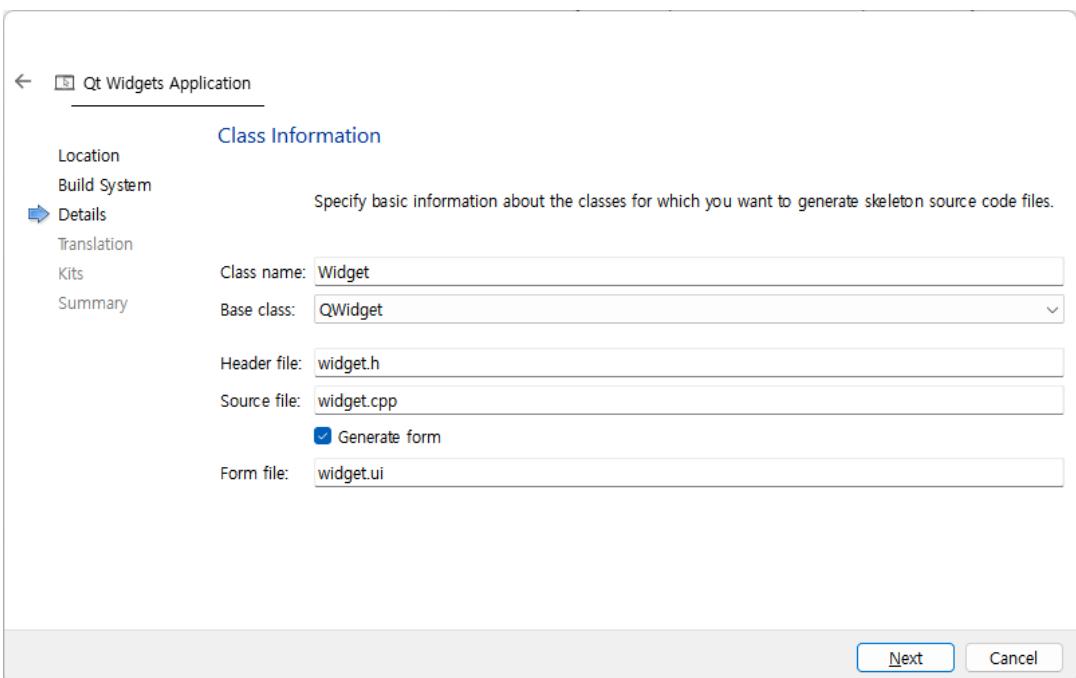
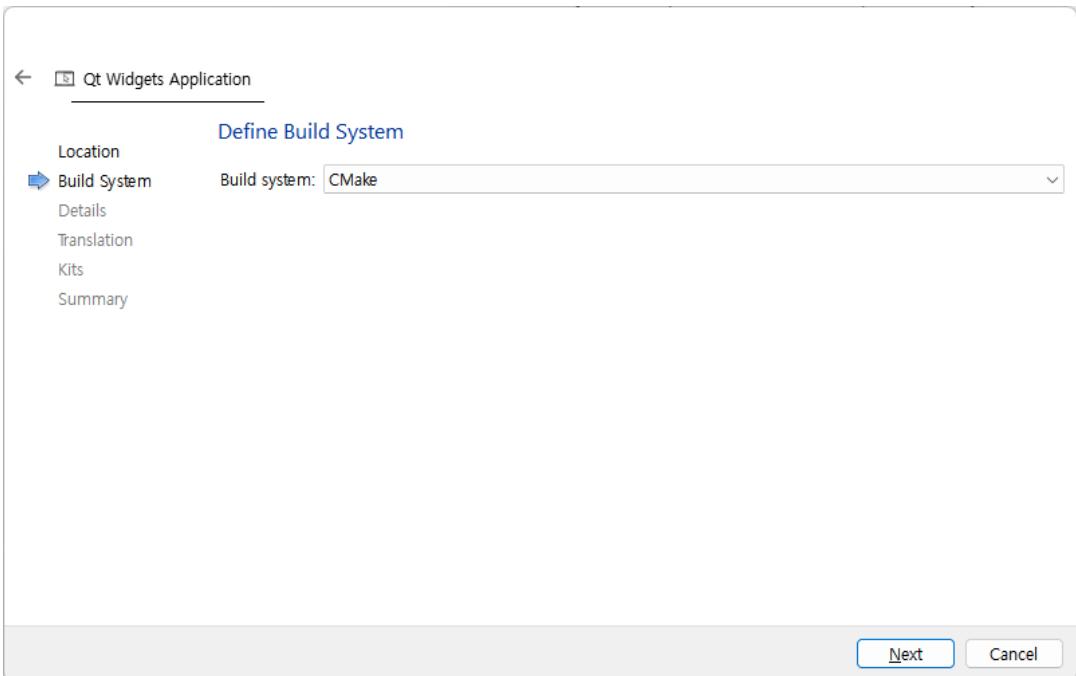
Widget Application] as shown in the image below.

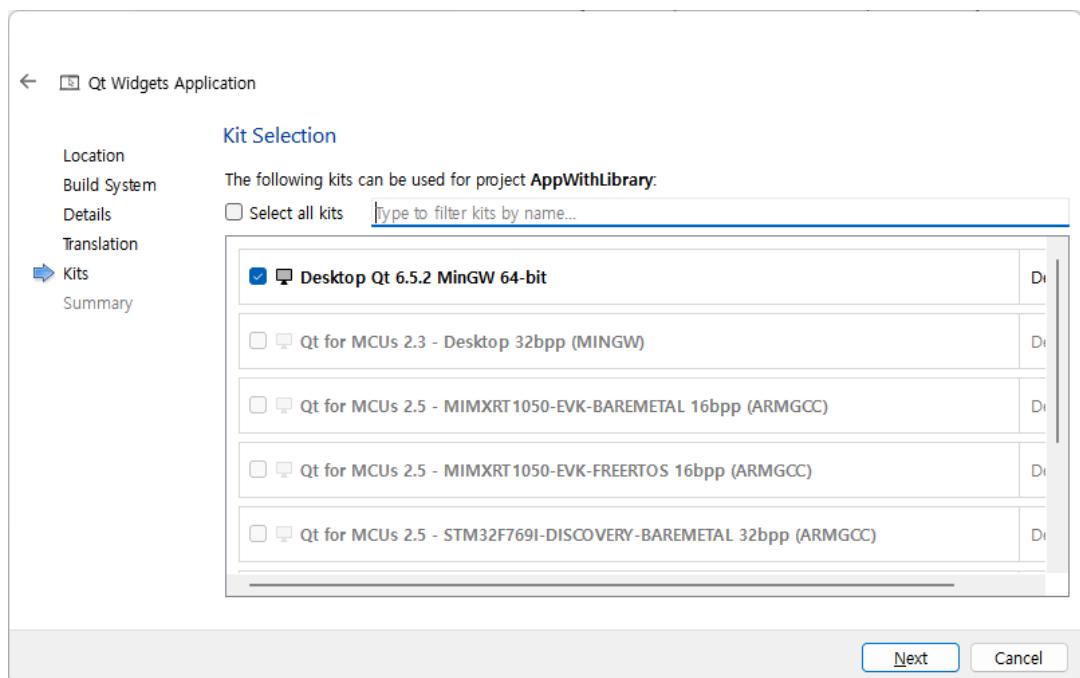
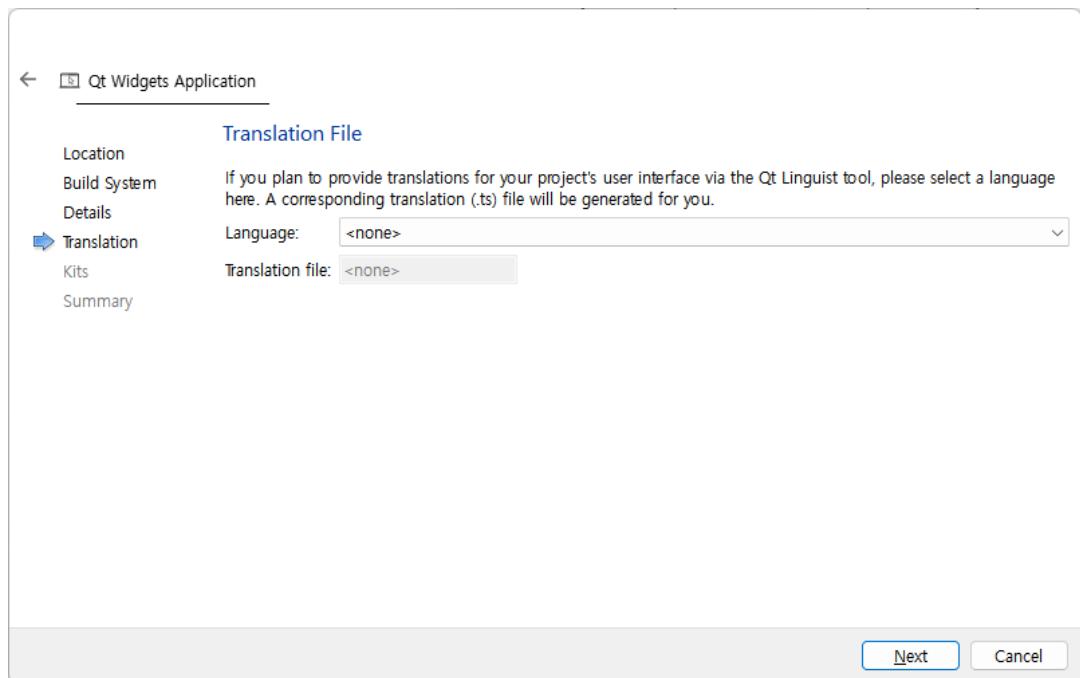


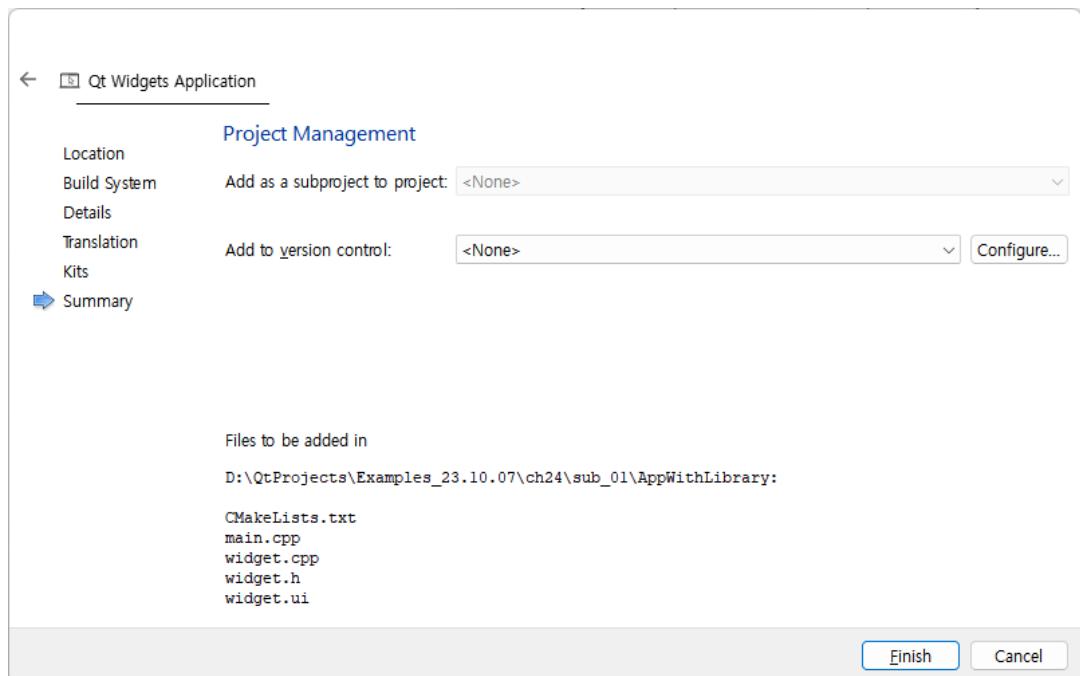
When creating a project, you need to specify the location of the library files and header files, so we will create the project in the same location where the MyUtil project was created.



Jesus loves you.







Once the project has been created, open the CMakeList.txt file and add the following to it.

```
cmake_minimum_required(VERSION 3.5)

project(AppWithLibrary VERSION 0.1 LANGUAGES CXX)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Qt NAMES Qt6 Qt5 REQUIRED COMPONENTS Widgets)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets)

set(PROJECT_SOURCES
    main.cpp
    widget.cpp
    widget.h)
```

```
    widget.ui
)

include_directories(../MyUtil)
link_directories(../build-MyUtil-Desktop_Qt_6_5_2_MinGW_64_bit-Debug)

if(${QT_VERSION_MAJOR} GREATER_EQUAL 6)
    qt_add_executable(AppWithLibrary
        MANUAL_FINALIZATION
        ${PROJECT_SOURCES}
    )
else()
    if(ANDROID)
        add_library(AppWithLibrary SHARED
            ${PROJECT_SOURCES}
        )
    else()
        add_executable(AppWithLibrary
            ${PROJECT_SOURCES}
        )
    endif()
endif()

target_link_libraries(AppWithLibrary MyUtil)

target_link_libraries(AppWithLibrary Qt${QT_VERSION_MAJOR}::Widgets)

if(${QT_VERSION} VERSION_LESS 6.1.0)
    set(BUNDLE_ID_OPTION MACOSX_BUNDLE_GUI_IDENTIFIER com.example.AppWithLibrary)
endif()
set_target_properties(AppWithLibrary PROPERTIES
    ${BUNDLE_ID_OPTION}
    MACOSX_BUNDLE_VERSION ${PROJECT_VERSION}
    MACOSX_BUNDLE_SHORT_VERSION_STRING
    ${PROJECT_VERSION_MAJOR}.${PROJECT_VERSION_MINOR}
    MACOSX_BUNDLE TRUE
```

```
WIN32_EXECUTABLE TRUE
)

include(GNUInstallDirs)
install(TARGETS AppWithLibrary
    BUNDLE DESTINATION .
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)

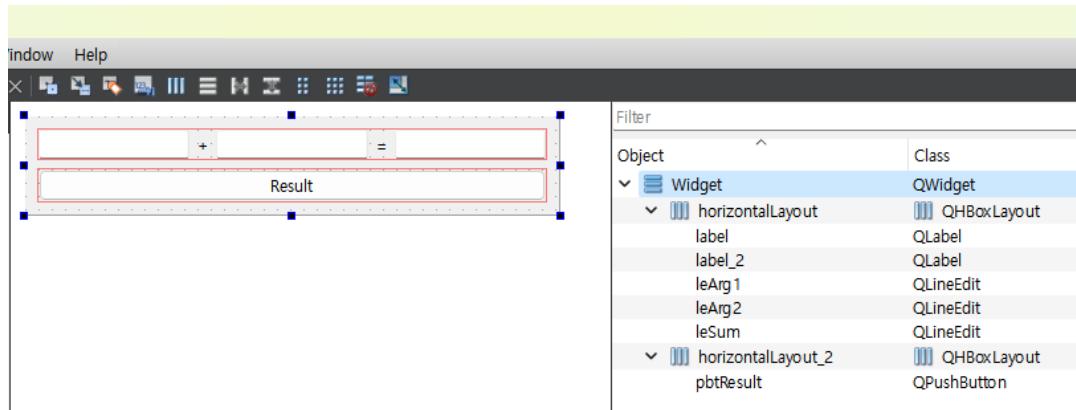
if(QT_VERSION_MAJOR EQUAL 6)
    qt_finalize_executable(AppWithLibrary)
endif()
```

Add include\_directories( ) after set( ). This function points to the location of the headers of the library functions.

Next, add a link\_directories( ) function. This function points to the location of the library files.

Finally, add the target\_link\_libraries( ) function. This function specifies the name of the library.

Once you've added the libraries as shown above, open the widget.ui file to create the UI, as shown in the image below.



Create the UI as shown above. Then, when you click the [Result] button, it will call the getSumValue( ) function written in the MyUtil class of the library to get the result and then display the result in the leSum object.

First, open the widget.h source code file and write the code like below.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include "myutil.h"

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
    MyUtil *m_myUtil;

private slots:
    void slot_pbtResult();

};

#endif // WIDGET_H
```

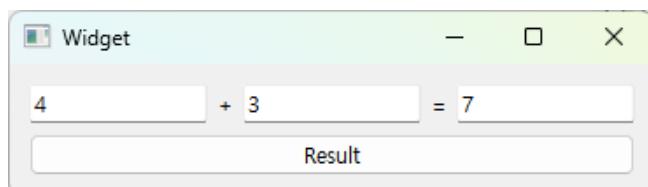
Next, let's open the `widget.cpp` source code file and write something like this

```
#include "widget.h"
#include "./ui_widget.h"
#include <QDebug>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
    , ui(new Ui::Widget)
```

```
{  
    ui->setupUi(this);  
  
    m_myUtil = new MyUtil();  
  
    connect(ui->pbtResult, &QPushButton::clicked,  
            this,           &Widget::slot_pbtResult);  
}  
  
Widget::~Widget()  
{  
    delete ui;  
}  
  
void Widget::slot_pbtResult()  
{  
    qDebug() << Q_FUNC_INFO;  
  
    qint32 arg1 = ui->leArg1->text().toInt();  
    qint32 arg2 = ui->leArg2->text().toInt();  
  
    //qint32 sumValue = 0;  
    qint32 sumValue = m_myUtil->getSumValue(arg1, arg2);  
    ui->leSum->setText(QString("%1").arg(sumValue));  
}
```

Once you've done that, let's check the results.



The source code for this example can be found in the sub\_01 directory.

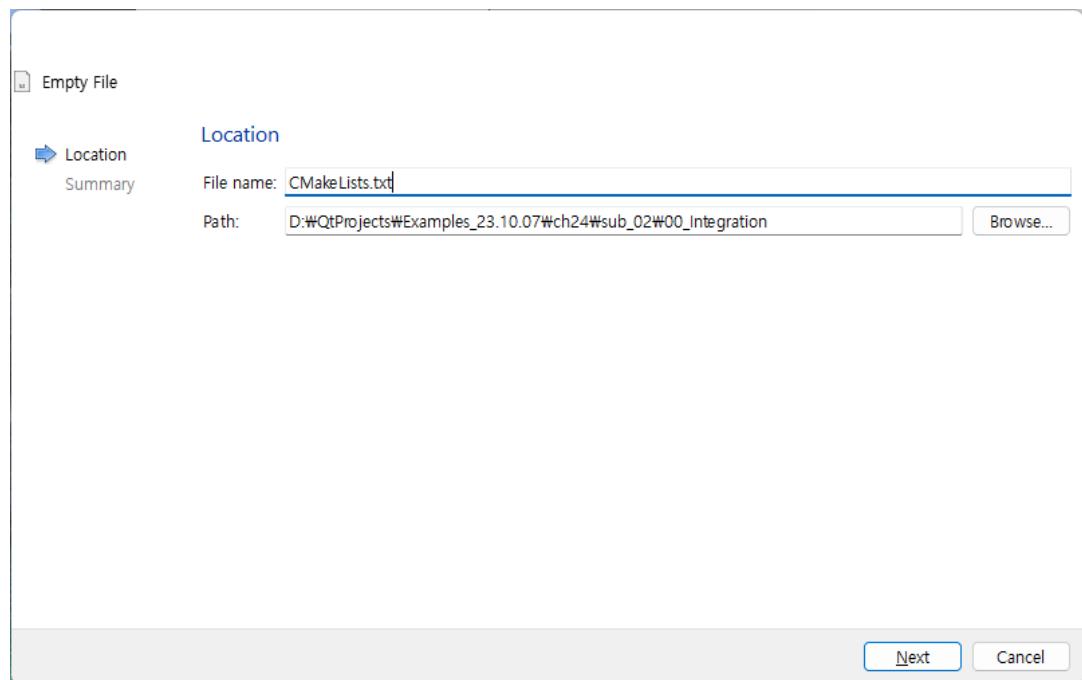
## 23.2. Building with Library

Qt provides a way to batch build two projects at once. In this chapter, we will learn how to build two projects together into a single project.

First, create a CMakeList.txt file using the Qt Creator tool. You can use a tool like Notepad to create the CMakeLists.txt file. In this example, we will use the Qt Creator tool to create the CMakeLists.txt file.

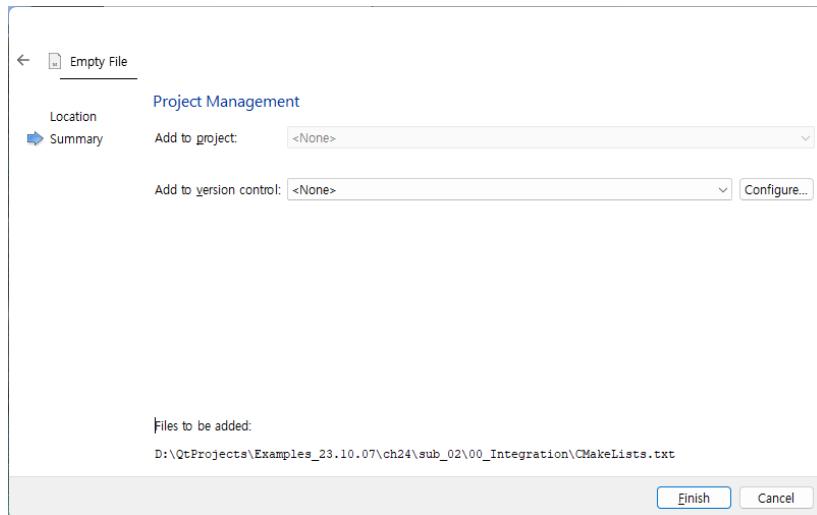
In Qt Creator, click New File from the File menu. Then select [General] from the left tab and [Empty File] from the middle tab, as shown in the image below.

Enter "CMakeLists.txt" in the File name field and click the [Next] button at the bottom, as shown in the image below.

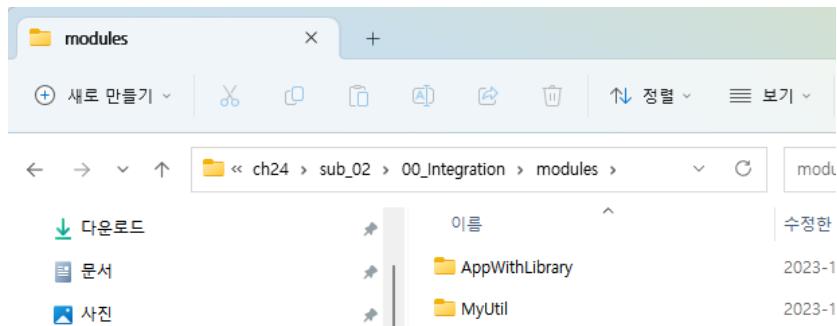


In the next dialog window, leave the defaults and click the Finish button.

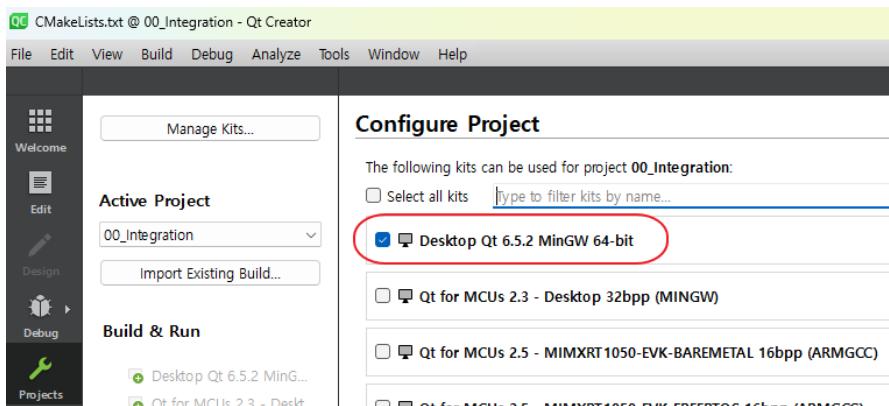
Jesus loves you.



Next, create a directory called modules in the same location where the CMakeLists.txt file was created and copy the two examples from this previous chapter into it.



Then open the CMakeLists.txt file in Qt Creator. Once this file is open, you will need to select the compiler you want to build your project with. Select MinGW, as shown in the figure below.



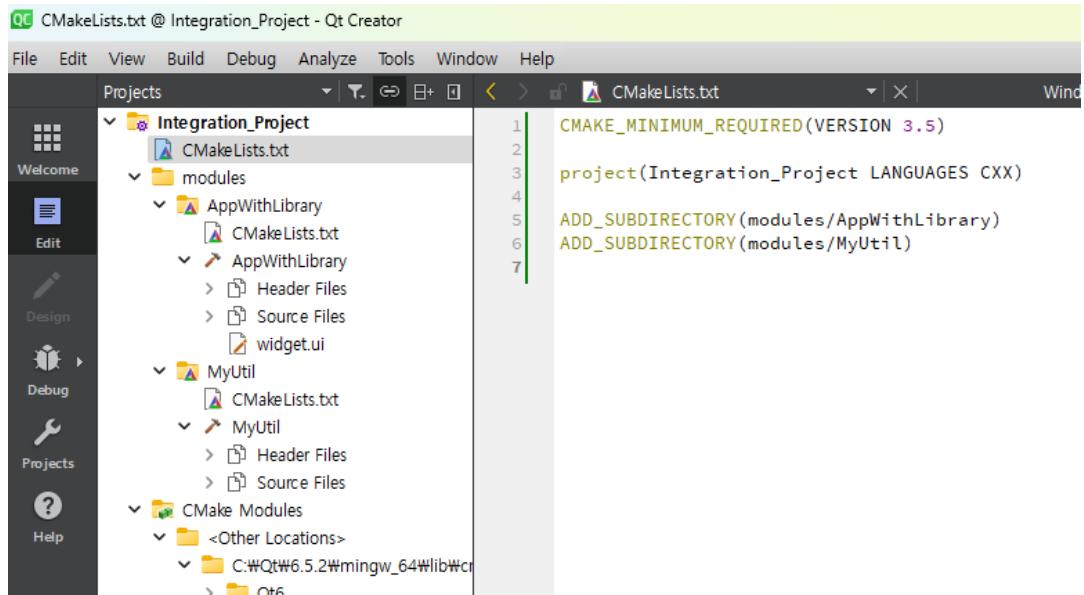
Once the empty project is created, open the CMakeLists.txt file to add the two projects, and write the following code

```
CMAKE_MINIMUM_REQUIRED(VERSION 3.5)

project(Integration_Project LANGUAGES CXX)

ADD_SUBDIRECTORY(modules/AppWithLibrary)
ADD_SUBDIRECTORY(modules/MyUtil)
```

If you write and build as above, the two projects will be built in batch. The advantage of this is that if you make any modifications, you can modify a specific project and then build it, and not have to build each project separately in Qt Creator because it will be built in batch.

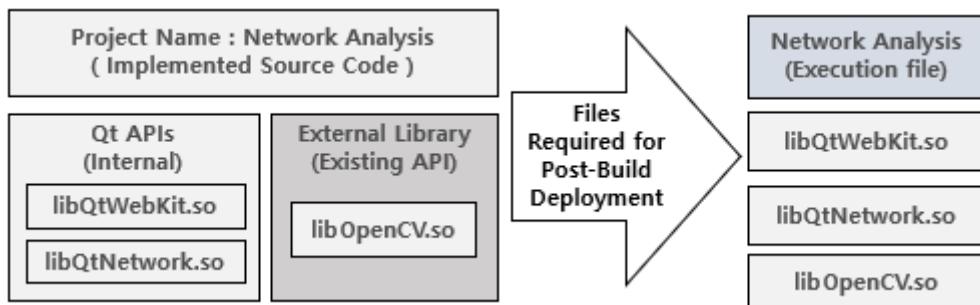


For an example of this project, see the sub\_02 > 00\_Integration directory.

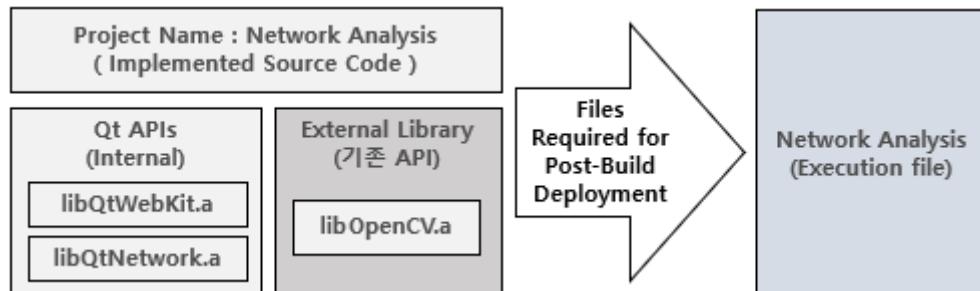
## 24. Creating library (qmake)

In this chapter, we will cover the same material as in the previous chapter. However, whereas in the previous chapter we used CMake, in this chapter we will look at how to implement libraries using qmake and how to use them.

Qt can use shared libraries and static libraries. Shared libraries are executable and library files separated, as shown in the figure below.



With the Static library method, the libraries are combined into a single file with the executable at build time. So if you build with the Static method, you only need to deploy the executable when you deploy.



One thing to note when using external libraries in Qt is that they must be built using the same compiler that compiled them.

For example, if your application is built with a MinGW-based compiler, the external library must also use a library built with a MinGW-based compiler.

To use external libraries in Qt, you can specify them in your project file as follows

```
INCLUDEPATH += "C:/Intel/OpenCL/sdk/include"
```

```
LIBS += -L"C:/Intel/OpenCL/sdk/lib/x64"  
LIBS += -lOpenCL
```

As shown in the example above, the INCLUDEPATH keyword specifies the location directory where the external library's header files are located.

"-L" is the directory where the library is located. And -lOpenCL on the third line specifies the actual shared library file to use. The "-l" option before the library name means that it is a library file. If you look in the directory where the library file is located, the extension will be so for Linux and dll or lib for MS Windows.

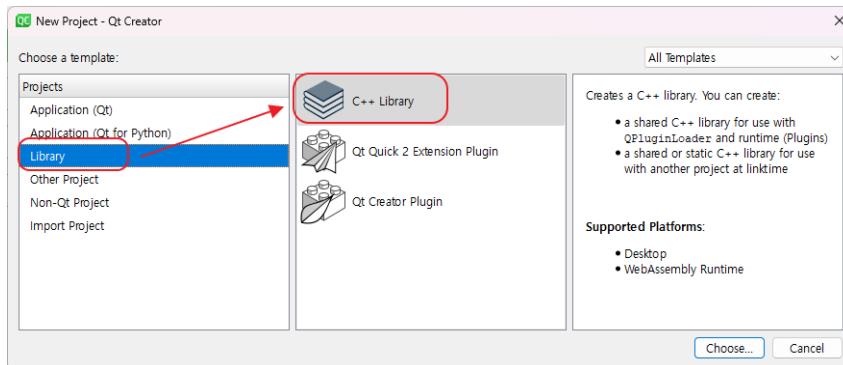
In this chapter, we'll break down how to implement libraries and how to build integrations into smaller sections.

## 24.1. Shared and Using Shared Library

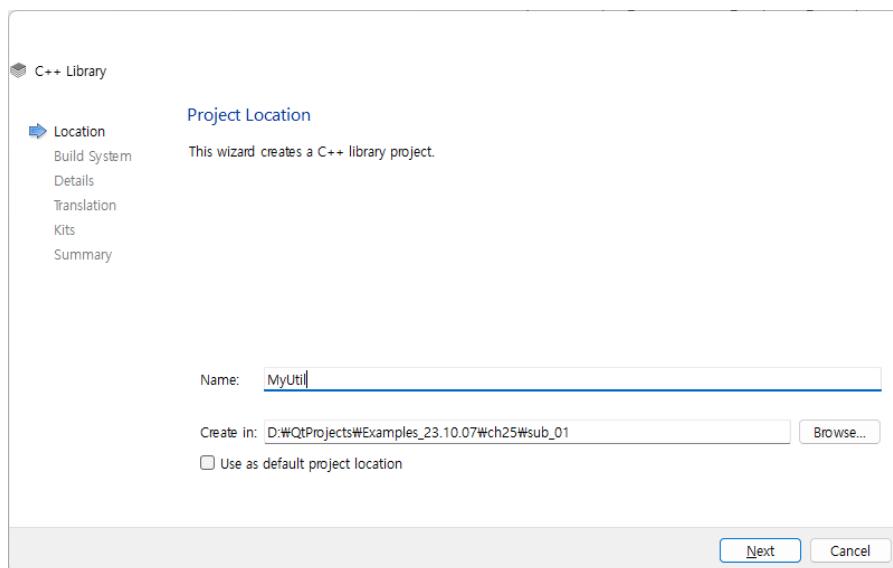
In this example, we will create two Qt projects. The first project is for implementing the library. The second project will be an example that uses the library implemented in the first project.

- ✓ library implementation example

Create a new project in Qt Creator. In the project creation dialog, select Library from the top left [Projects] tab and select the C++ Library item from the center list, as shown in the following figure.

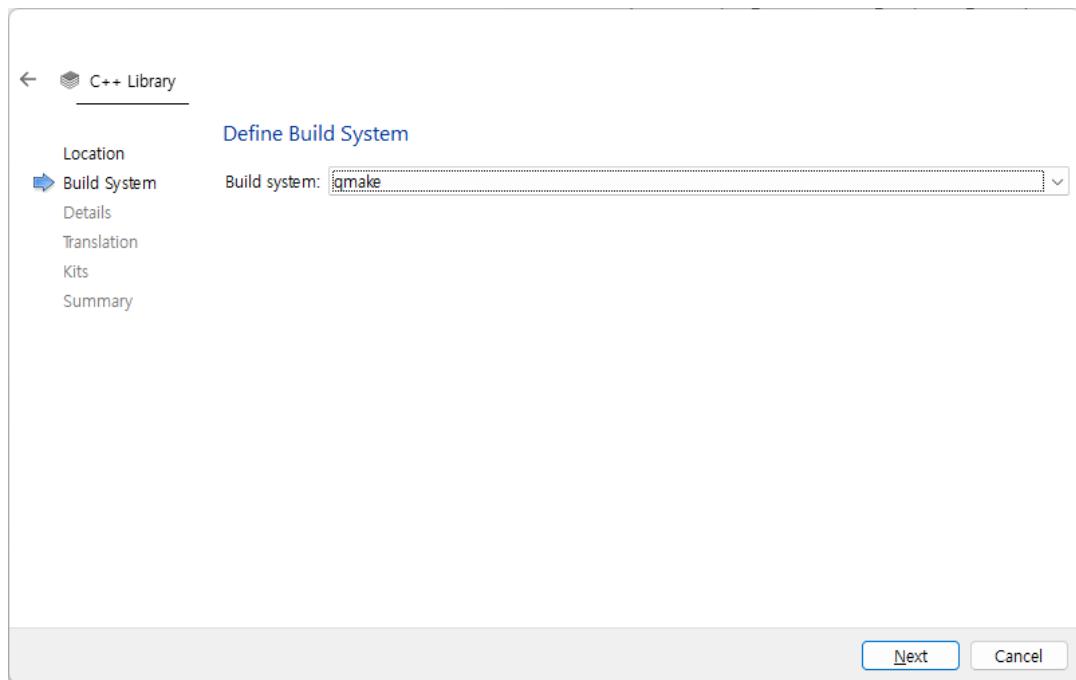


In the next dialog screen, enter "MyUtil" in the Name field.

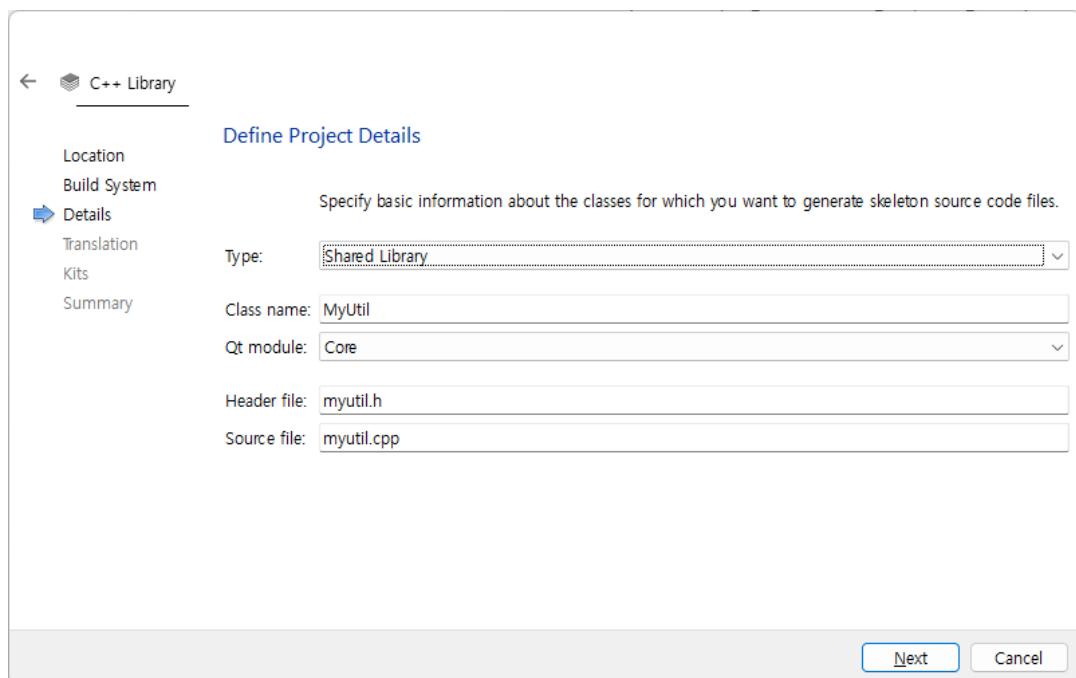


Jesus loves you.

Select qmake as your build tool.

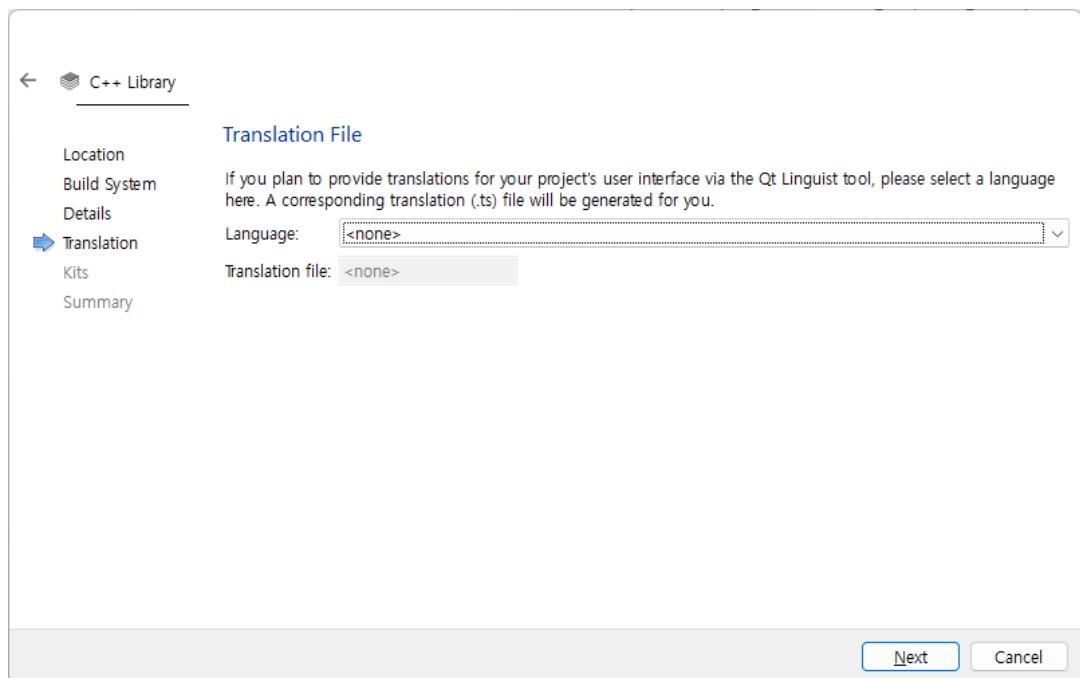


For Type, select Shared Library, as shown below. For Class name, enter MyUtil. For Qt module, enter Core. For Header file, enter myutil.h, and for Source file, enter myutil.cpp.

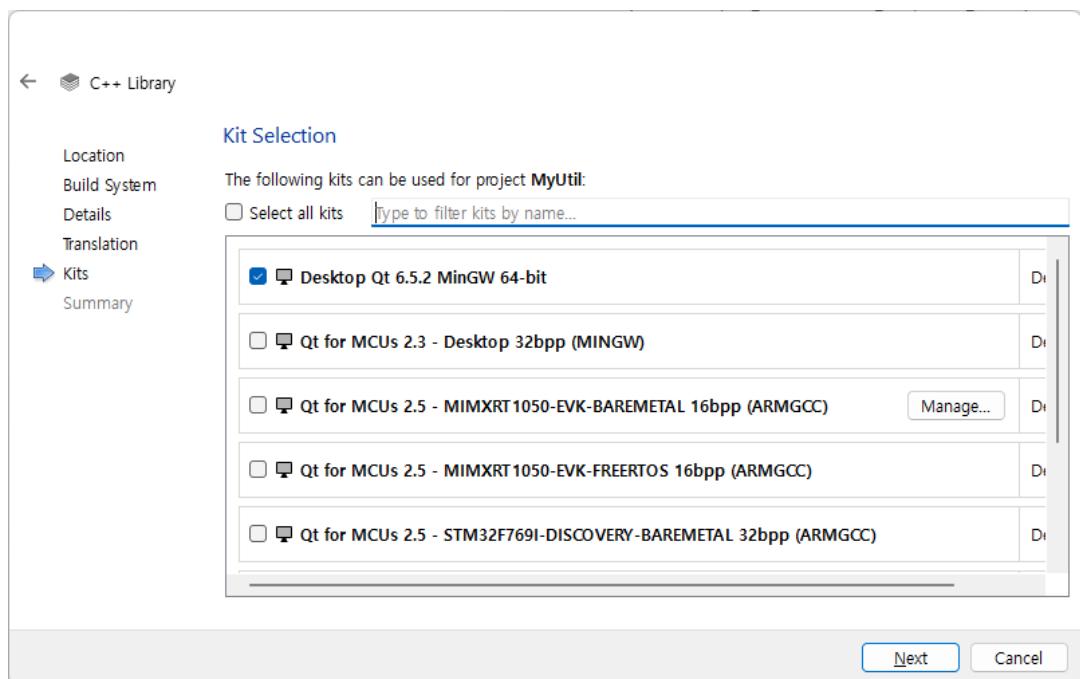


This is the setting for multilingualism. We'll leave it as default and click the [Next] button.

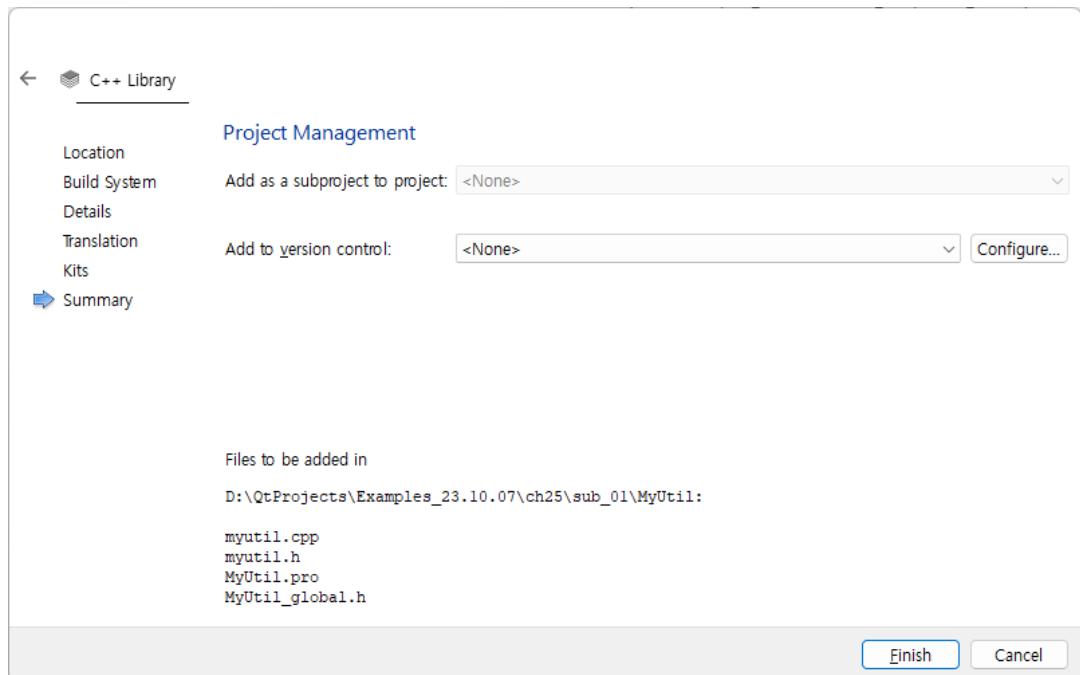
Jesus loves you.



In the Select Compiler dialog, select the MinGW compiler.



Click the Finish button at the bottom to complete the project creation, as shown in the dialog below.



Next, let's take a look at the project file after the project creation is complete.

```
QT -= gui

TEMPLATE = lib
DEFINES += MYUTIL_LIBRARY

CONFIG += c++17

SOURCES += \
    myutil.cpp

HEADERS += \
    MyUtil_global.h \
    myutil.h

# Default rules for deployment.
unix {
    target.path = /usr/lib
}
!isEmpty(target.path): INSTALLS += target
```

Unlike regular application projects, library projects have a different TEMPLATE keyword. A regular application specifies app in the TEMPLATE keyword, but a library specifies lib.

Jesus loves you.

Next, write the source code for myutil.h as shown below.

```
#ifndef MYUTIL_H
#define MYUTIL_H

#include <QObject>
#include "MyUtil_global.h"

class MYUTIL_EXPORT MyUtil : public QObject
{
    Q_OBJECT
public:
    explicit MyUtil(QObject *parent = nullptr);
    int getSumValue(int a, int b);

};

#endif // MYUTIL_H
```

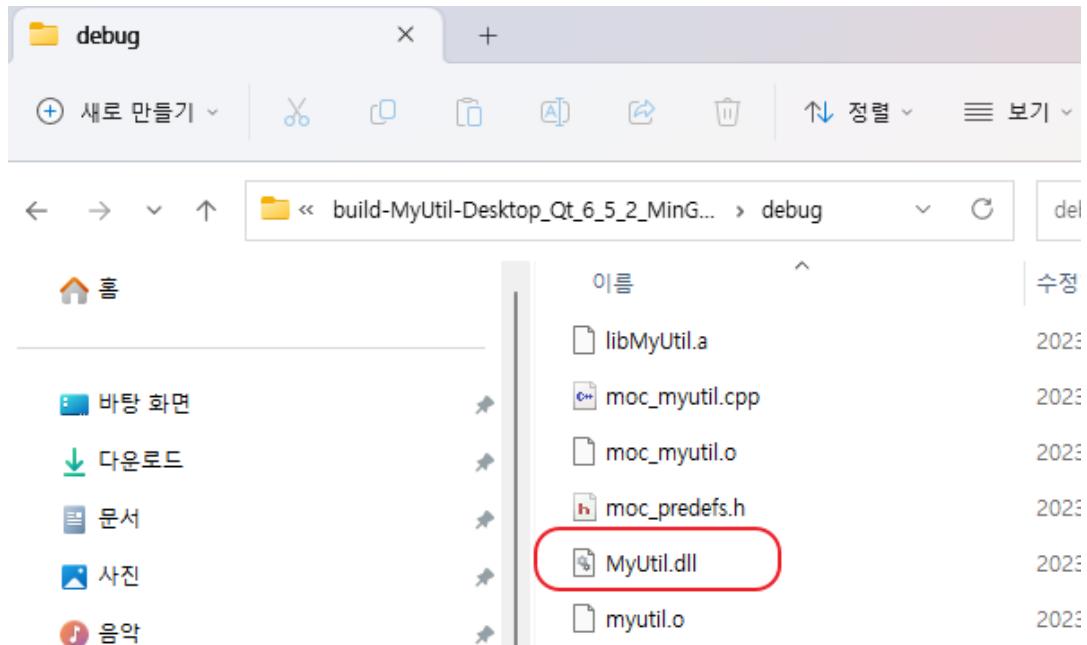
Then open myutil.cpp and write the source code like below.

```
#include "myutil.h"

MyUtil::MyUtil(QObject *parent) : QObject(parent)
{
}

int MyUtil::getSumValue(int a, int b)
{
    return a + b;
}
```

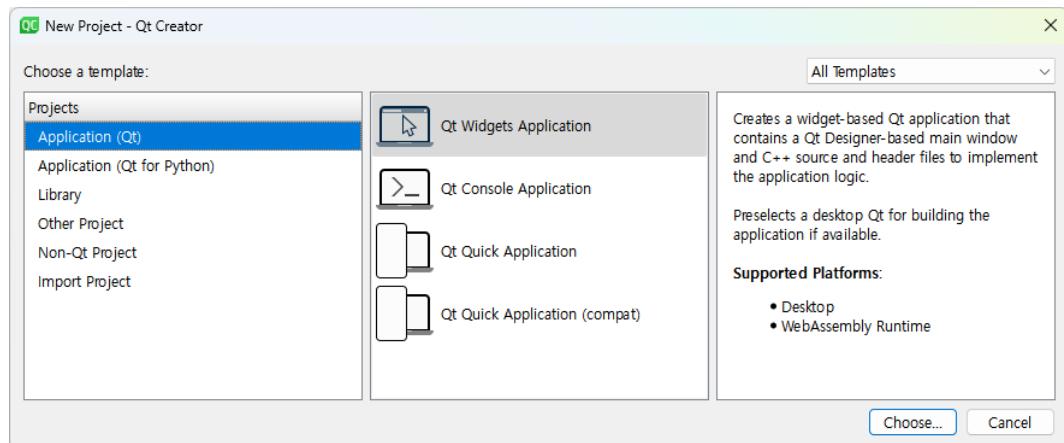
The MyUtil class simply passes two values as function arguments and returns the sum of the two values as the result. Once you've gotten this far, you can build it and then go to the directory where it was built to see that the library files have been created.



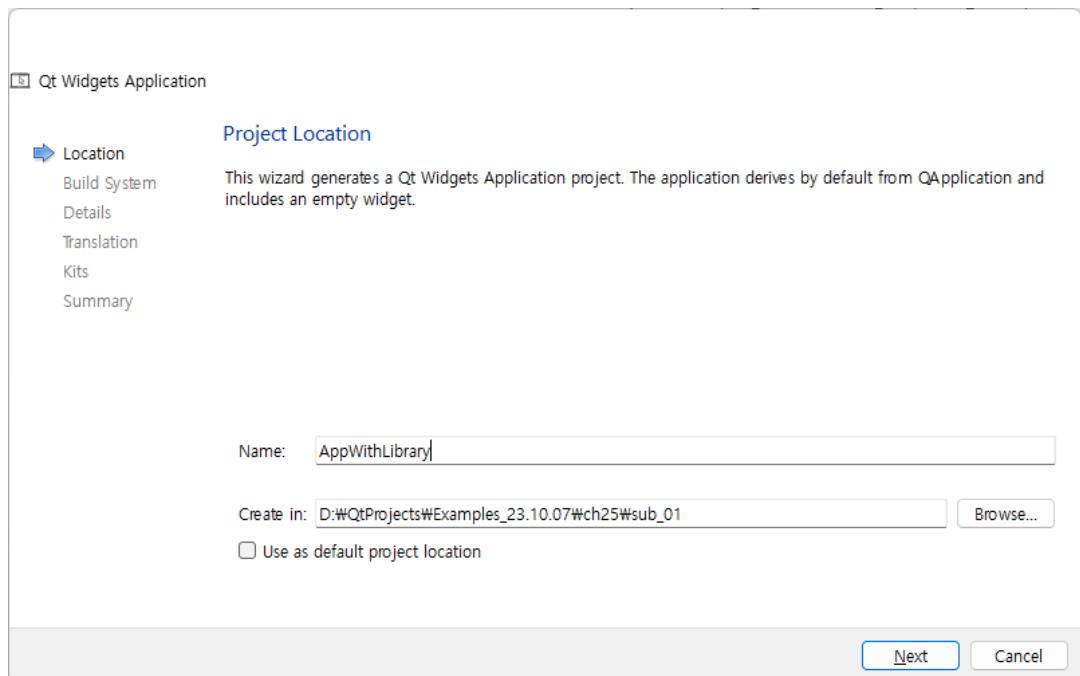
Once you've successfully created the library file as shown in the image above, let's write an example that uses this library.

- ✓ Example using the library implemented at

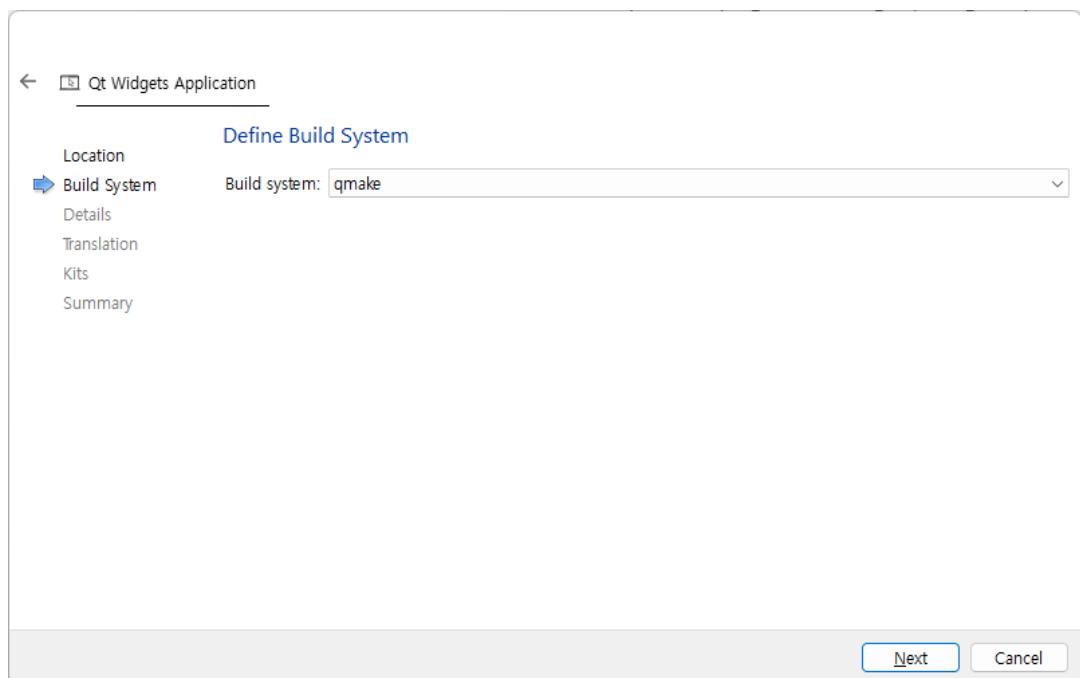
When creating the project, select [Qt Widget Application] as shown in the image below.

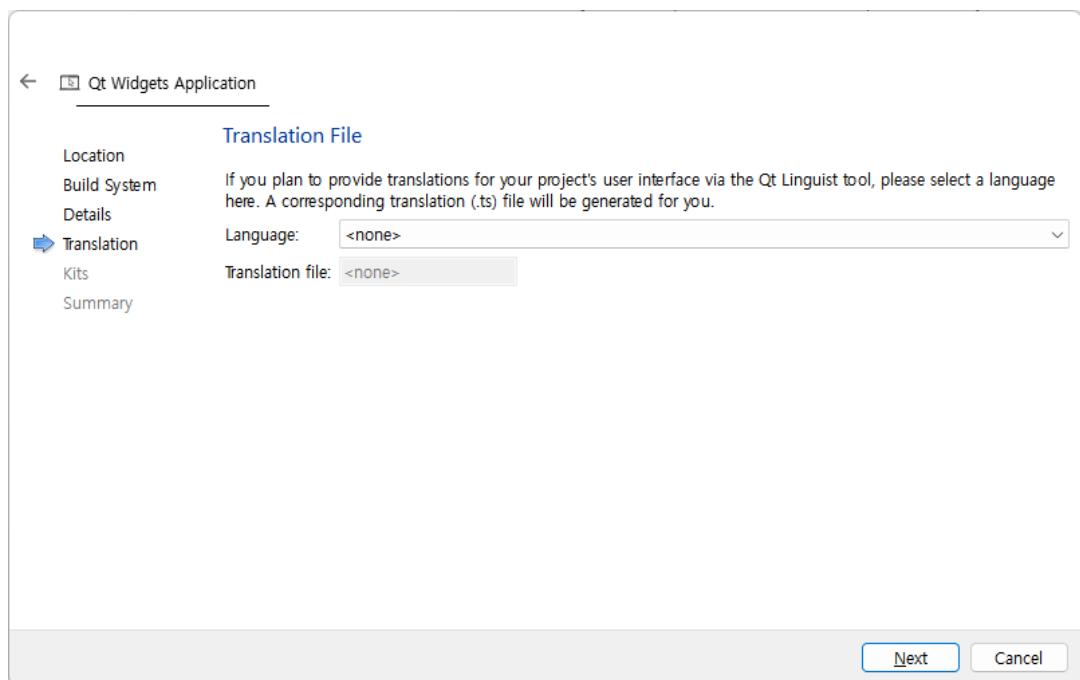
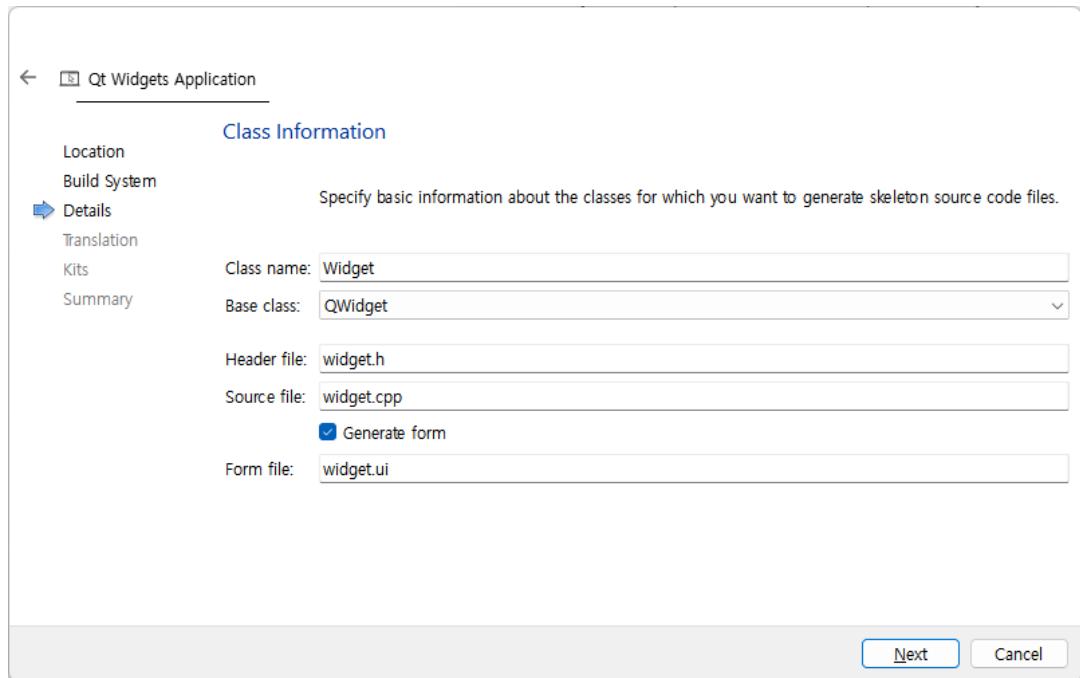


When creating a project, you need to specify the location of the library files and header files, so we will create the project in the same location where the MyUtil project was created.

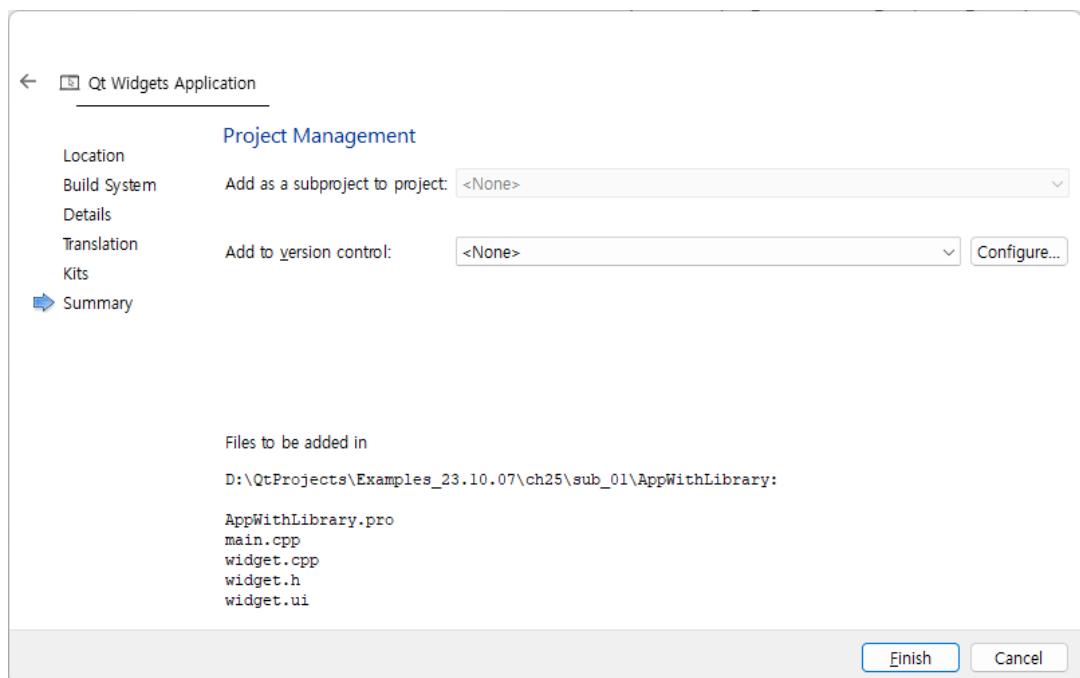
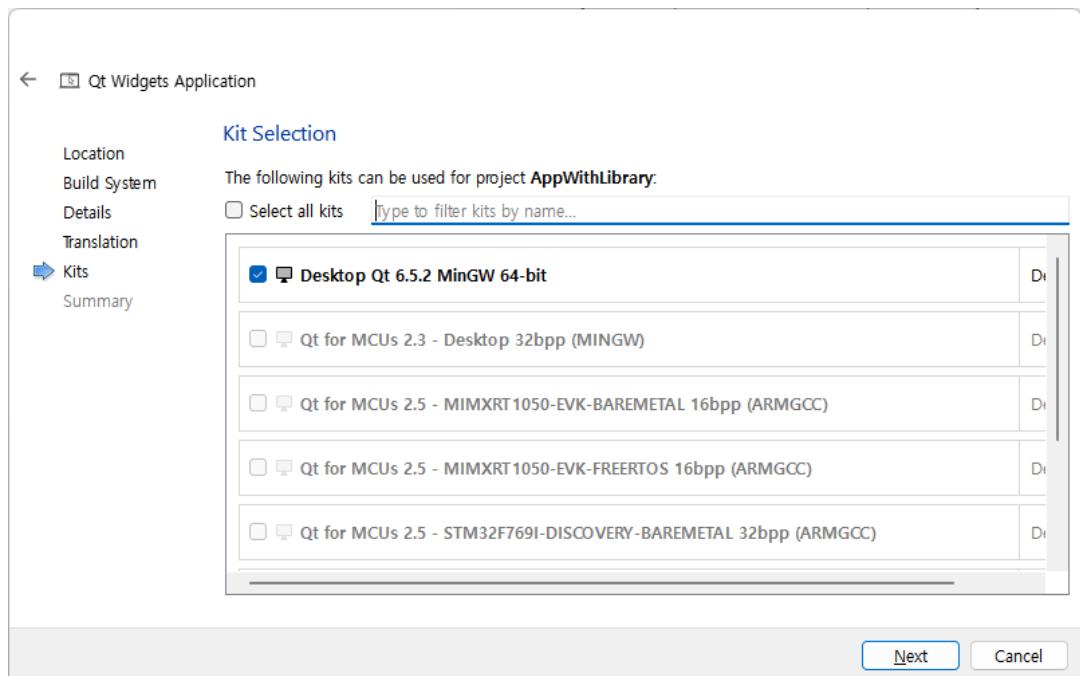


Select qmake as your build system.





Jesus loves you.



Once the project is created, open the project file and modify it like below.

```
QT      += core gui
```

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

CONFIG += c++17

SOURCES += \
    main.cpp \
    widget.cpp

HEADERS += \
    widget.h

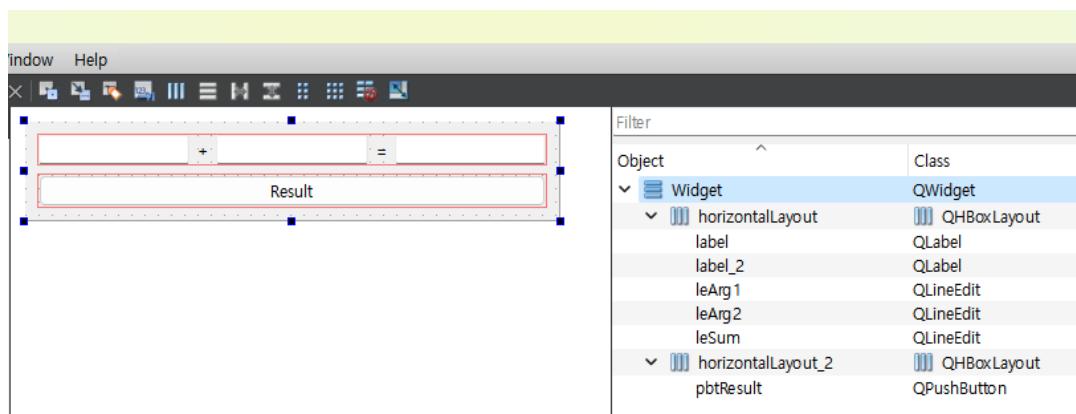
FORMS += \
    widget.ui

# Default rules for deployment.
qnx: target.path = /tmp/$${TARGET}/bin
else: unix:!android: target.path = /opt/$${TARGET}/bin
!isEmpty(target.path): INSTALLS += target

LIBS += -L$$PWD/../../[빌드된 디렉토리 명]/debug/ -lMyUtil

INCLUDEPATH += $$PWD/../../MyUtil
DEPENDPATH += $$PWD/../../MyUtil
```

Once you've added the library as shown above, open the `widget.ui` file to build the UI, as shown in the image below.



Create the UI as shown above. Then, when you click the [Result] button, it will call the

Jesus loves you.

getSumValue( ) function written in the MyUtil class of the library to get the result and then display the result in the leSum object.

First, open the widget.h source code file and write the code like below.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include "myutil.h"

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
    MyUtil *m_myUtil;

private slots:
    void slot_pbtResult();

};

#endif // WIDGET_H
```

Next, let's open the widget.cpp source code file and write something like this

```
#include "widget.h"
#include "./ui_widget.h"
#include <QDebug>
```

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

    m_myUtil = new MyUtil();

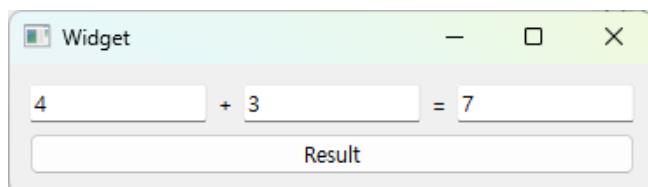
    connect(ui->pbtResult, &QPushButton::clicked,
            this,           &Widget::slot_pbtResult);
}

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

void Widget::slot_pbtResult()
{
    qint32 arg1 = ui->leArg1->text().toInt();
    qint32 arg2 = ui->leArg2->text().toInt();

    qint32 sumValue = m_myUtil->getSumValue(arg1, arg2);
    ui->leSum->setText(QString("%1").arg(sumValue));
}
```

Once you've done that, let's check the results.

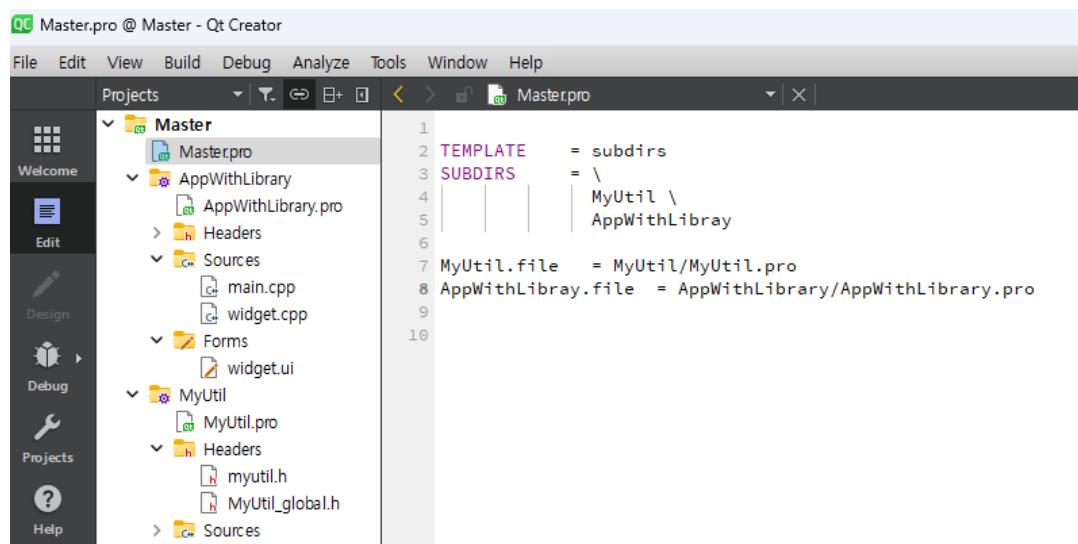


For this example, you can refer to the sub\_01 directory.

## 24.2. Building with Library

The library project and the application project are built using separate Qt Creator tools. Suppose you need to modify a library, it would be inconvenient to modify the library in the Qt Creator tool with the library project open, build it, and then return to the Qt Creator window with the application project open to apply the modified library.

However, Qt allows you to modify multiple projects in a single Qt Creator window. The advantage of this is that you can modify and apply libraries and application source code simultaneously in the same Qt Creator tool, as long as the location of the library directories in the project is correct. The following figure shows an example of two projects open in one Qt Creator tool.



To open multiple projects in Qt Creator, create a project file as shown in the image above.

```

TEMPLATE      = subdirs

SUBDIRS       = MyUtil \
                AppWithLibrary

MyUtil.file   = MyUtil/MyUtil.pro
AppWithLibrary.file = AppWithLibrary/AppWithLibrary.pro

```

Create a project file (.pro) file as shown in the example above. This project file makes

Jesus loves you.

building very convenient because by specifying two separate project files, you can modify and build your project in one Qt Creator tool.

For example, if you change the source code of a library, Qt Creator automatically builds the changed project, so you don't have to build the library separately.

See the sub\_02 directory for the source code for this example.

## 25. D-Pointer

C/C++ uses opaque pointers (or opaque types) for the purpose of version compatibility and source code secrecy. Qt also uses Opaque Pointer, but D-Pointer is an improvement of Opaque Pointer to make it more suitable for Qt.

And if you look at the header code of the Internal module (library) provided by Qt, you can see that it is implemented in the form of D-Pointer as shown below.

```
#ifndef QCOREAPPLICATION_H
#define QCOREAPPLICATION_H

#include <QtCore/qglobal.h>
...
class QCoreApplicationPrivate;
...
class Q_CORE_EXPORT QCoreApplication
{
}
```

The above example source code is the header file for the QCoreApplication class provided by Qt. If you look at this class, you can see that it declares an additional Private class called QCoreApplicationPrivate.

In other words, you can see that the word Private is used after the class name. This form is known as a D-Pointer.

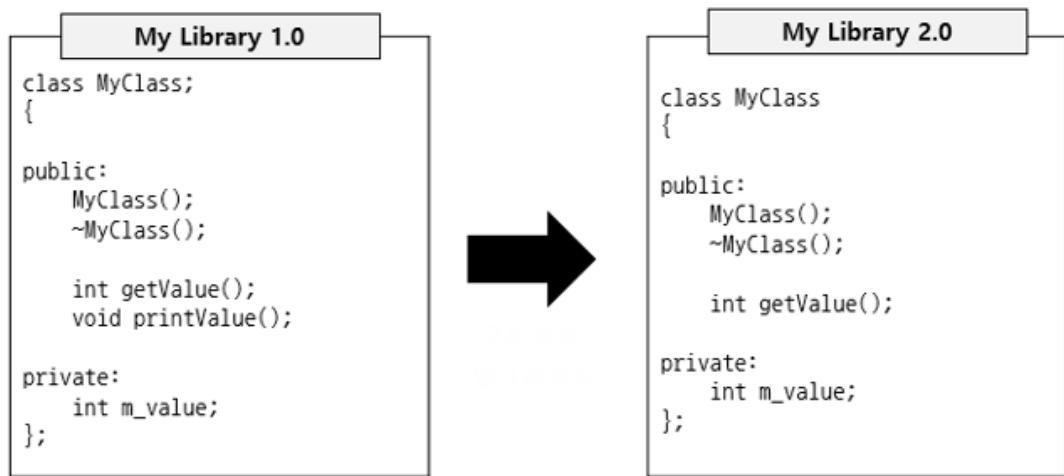
In this chapter, let's see the advantages of D-Pointer and how to use it through example code.

- ✓ Benefits and Features of D-Pointer

There are two main reasons for using D-Pointers. The first is to maintain compatibility between versions of a library.

Library developers need to keep each library version compatible. As the library version increases, the developer may be concerned about how to maintain compatibility between versions of the library due to changes in the library.

For example, member functions of a particular class may be added or removed from the library. Additions are not a big deal, but deletions are a big deal for version compatibility.



As you can see in the example above, the `printValue()` function was deleted when My Library was upgraded from version 1.0 to version 2.0.

What if an end-user developer using this library upgrades to version 2.0 and uses the `printValue()` function, they will get an error saying that the `printValue()` function is missing.

Therefore, we use Opaque Pointers as a way to help solve this problem.

Using D-Pointers doesn't completely solve the problem of version-specific compatibility, but it does allow us to maintain version compatibility.

Let's come back to D-Pointers.

If we look at the source code of the classes provided by Qt, we can see that they utilize D-Pointers. If you look at the `QCoreApplication` class, you'll see that it is a D-Pointer class that provides an additional class called `QCoreApplicationPrivate` with the word `Private` after the class name.

When implementing a library, you can maintain compatibility across versions by implementing things that are unlikely to change even if the version is updated mainly in `QCoreApplication`, and things that are likely to change a lot mainly in `Private` classes.

For example, if you delete `printValue()` in the above example source code and create a member function with the same name in `MyClass` and return information that this function has been removed when using it, developers using the library will be able to

solve problems caused by using the deleted member function more easily.

The second reason to use D-Pointers is to maintain the secrecy of your source code.

For example, when you distribute a library, you need to distribute the header files with it. However, you don't need to publish the header file of a private class.

For example, a developer using the QCoreApplication class does not need to distribute QCoreApplicationPrivate.h along with the QCoreApplication.h header file. (In D-Pointer, private class names are appended with a \_p after the class name.)

So if you want to keep your source code private, you can implement it in the QCoreApplicationPrivate class.

This way, developers who use the final library can use the library without distributing the QCoreApplicationPrivate.h header file.

✓ Rules for D-Pointer

There are rules for using D-Pointers. For example, let's use the example of using D-Pointers in a class named MyClass when implementing a library.

**Class name: MyClass  
(myclass.h)**

**Class name: MyClassPrivate  
(myclass\_p.h)**

As shown in the example above, you should create a MyClass class and implement the Private class by appending the word Private to the MyClass name. And the class name must be prefixed with "\_p" after the class name.

So the Private class of MyClass should be named MyClassPrivate and a header file should be created with the name MyClassPrivate.

And if myclass.h is the name of the class header file, then the header file for the Private class should be named myclass\_p.h.

✓ Calling Cross-Class Member Functions from a D-Pointer

There are times when you need to call member functions between MyClass and MyClassPrivate class.

For example, when calling a function of class MyClassPrivate from MyClass, use Q\_D( ). Conversely, if you need to use a function of class MyClass from MyClassPrivate, you should use Q\_Q( ).

The following example source code is an example of using the Q\_D( ) function to call the printValue( ) member function of the MyClassPrivate class from MyClass.

```
#include "myclass.h"
#include "myclass_p.h"

MyClass::MyClass()
{
    d_ptr = new MyClassPrivate(this);
    Q_D(MyClass);

    m_value = 100;
    d->setValue(200);
}

MyClass::~MyClass()
{
}

int MyClass::getValue()
{
    return m_value;
}

void MyClass::printValue()
{
    Q_D(MyClass);
    d->printValue();
}
```

The following example shows the use of the Q\_Q( ) function in the MyClassPrivate class to call the getValue( ) member function of the MyClass class.

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

MyClassPrivate::MyClassPrivate(MyClass *q)
{
```

```
    q_ptr = q;
}

MyClassPrivate::~MyClassPrivate()
{
}

void MyClassPrivate::setValue(int val)
{
    m_value = val;
}

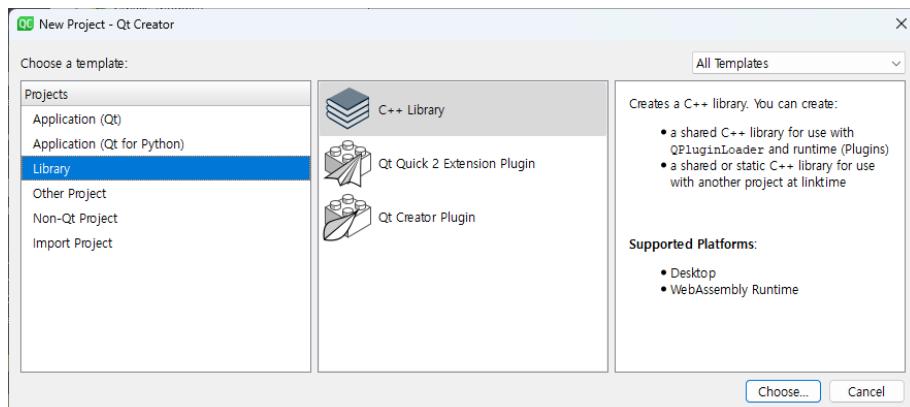
void MyClassPrivate::printValue()
{
    Q_Q(MyClass);

    qDebug() << " MyClass 의 m_value : " << q->getValue();
    qDebug() << " MyClassPrivate 의 m_value : " << m_value;
}
```

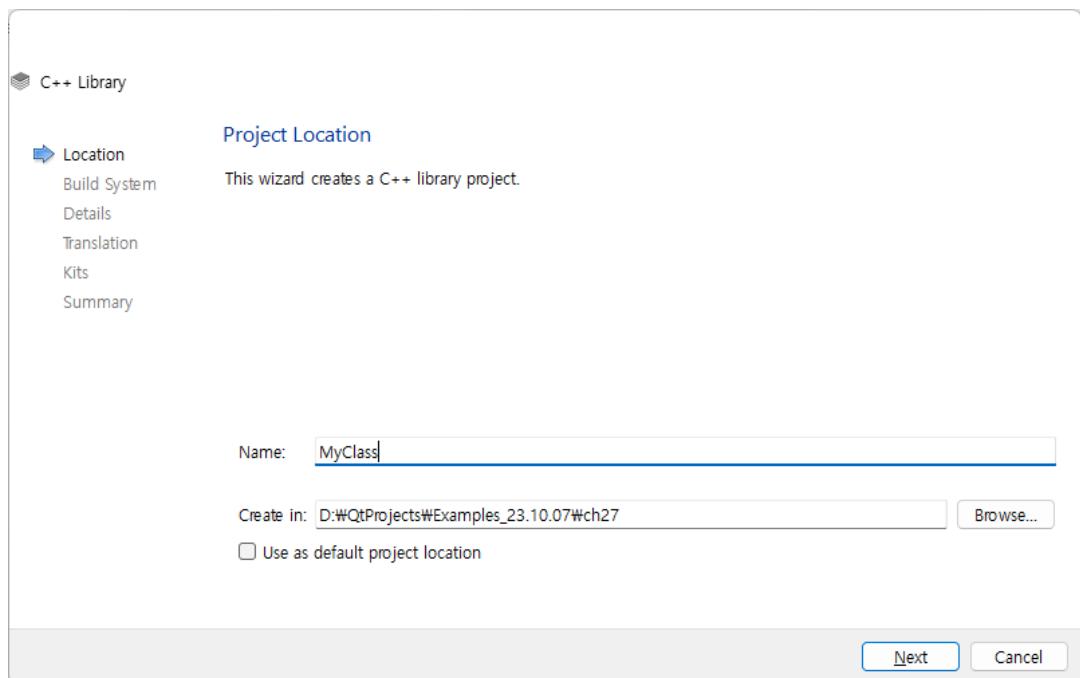
As shown in the example above, we need to use the Q\_D( ) and Q\_Q( ) functions to call member functions between MyClass and MyClassPrivate.

✓ Implementing the Library with D-Pointer

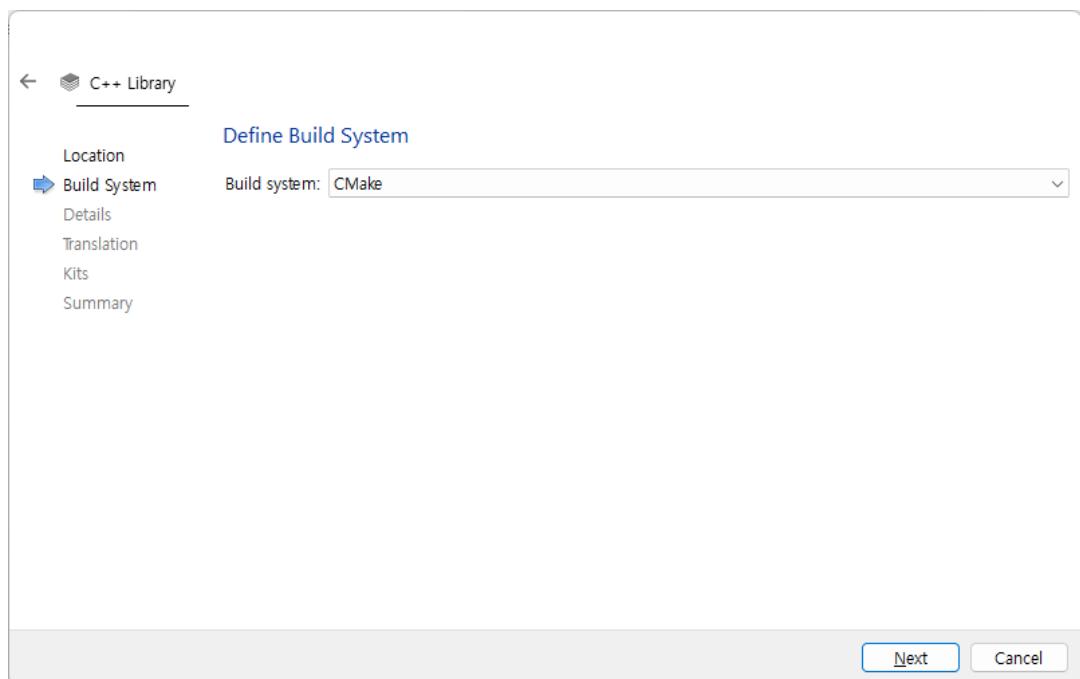
This time, we'll implement a library that uses D-Pointer. When creating a project, create a project as shown in the image below.



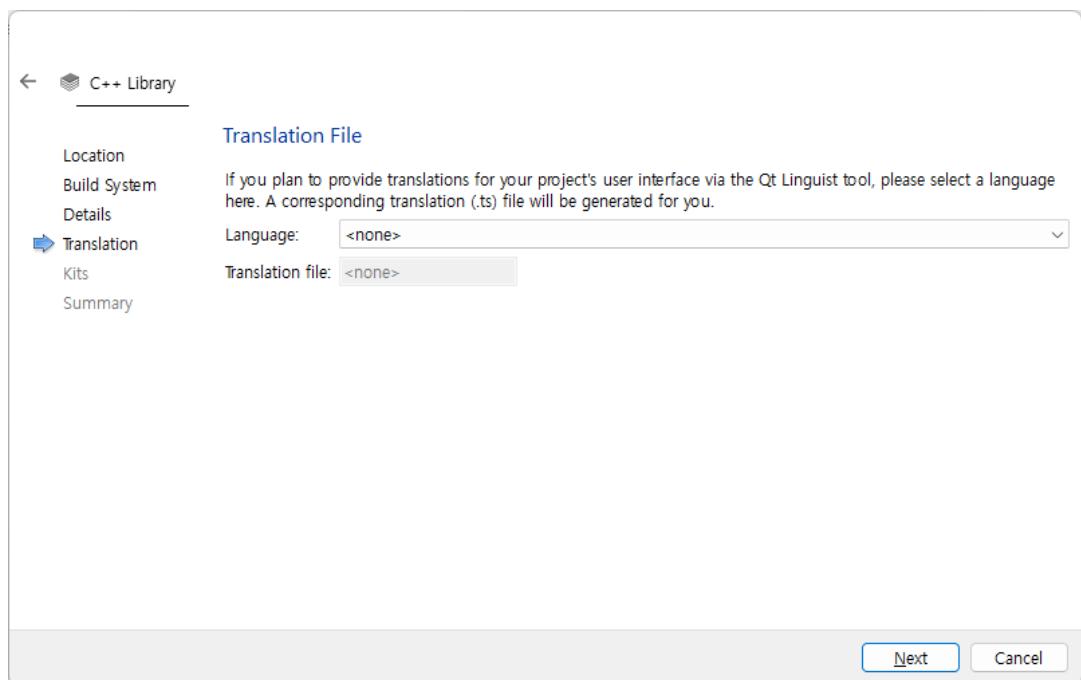
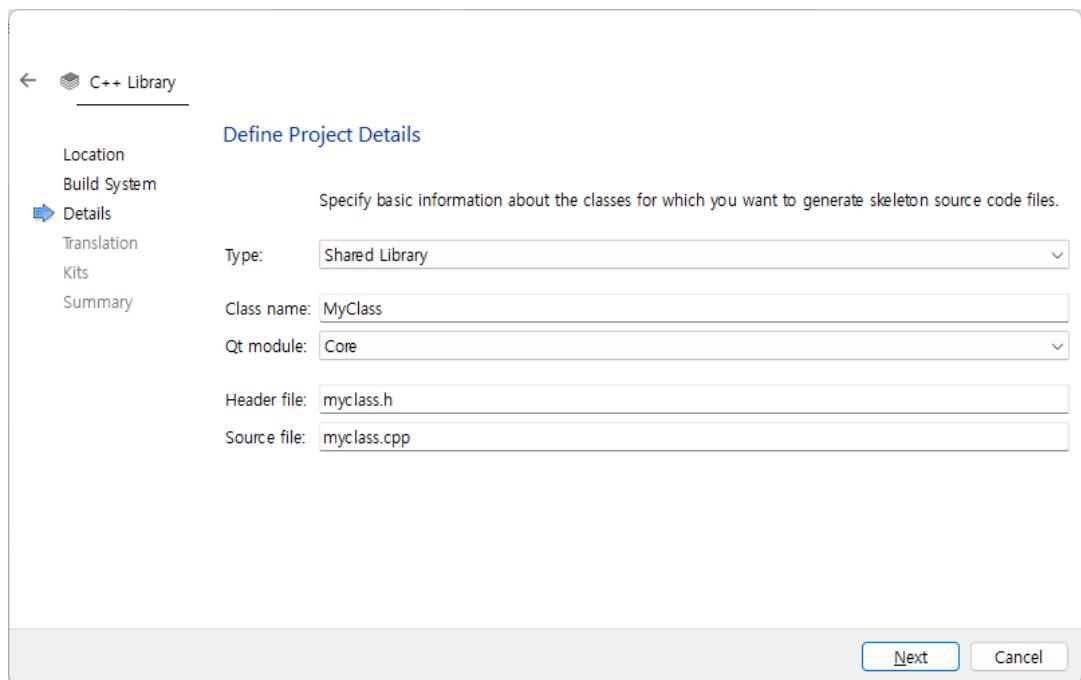
Create a project by entering "MyClass" as the name of the project.



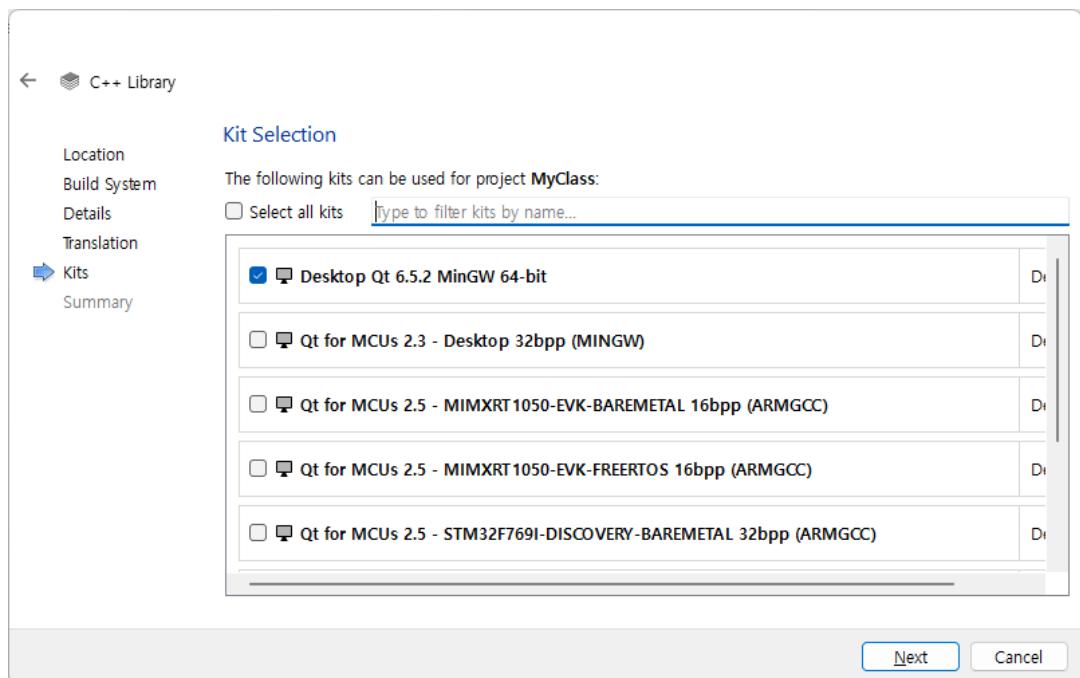
When creating a project, select CMake as the build tool.



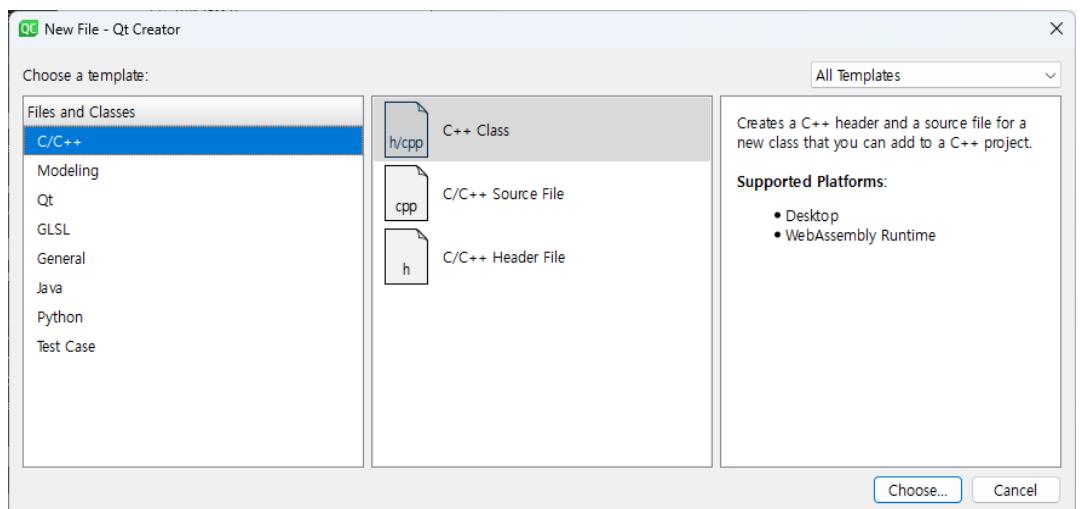
Select Shared Library as the type of project and enter the following.



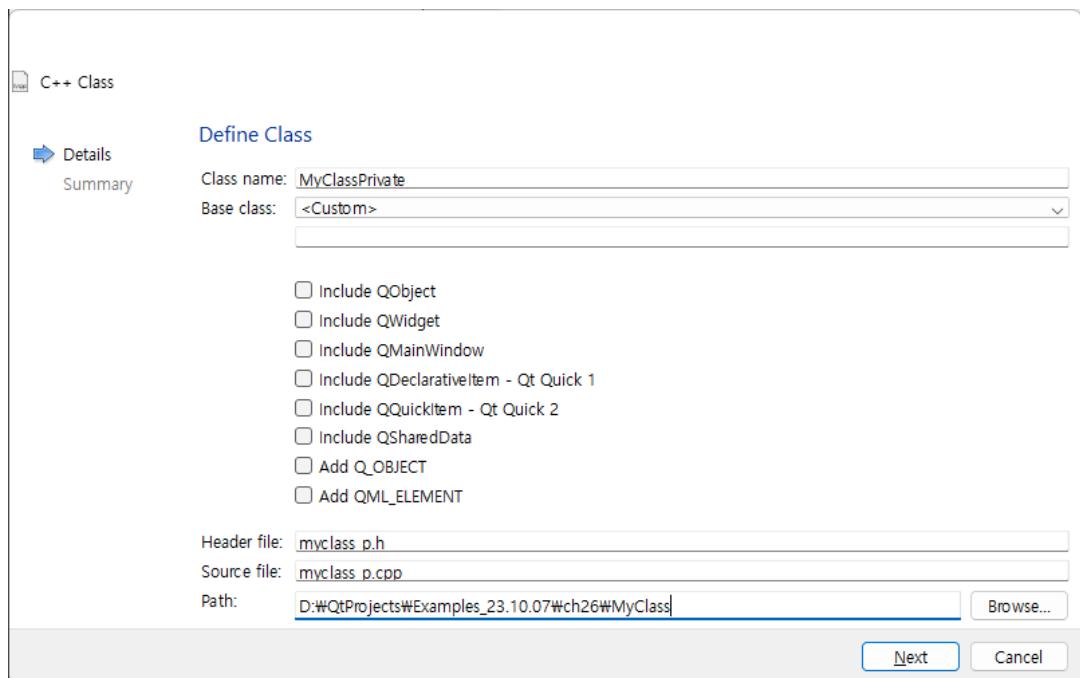
Jesus loves you.



Create a project as shown above and add the MyClassPrivate class.



Enter MyClassPrivate for the class name of the class you want to add. Enter "myclass\_p.h" for the header file name and "myclass\_p.cpp" for the source code file name.



Once you've entered the above, let's create the MyClass and MyClassPrivate classes. Let's create MyClassPrivate first. Create the Myclass\_p.h header file like below.

```
#ifndef MYCLASSPRIVATE_H
#define MYCLASSPRIVATE_H

#include "myclass.h"
#include <QDebug>

class MyClassPrivate
{
    Q_DECLARE_PUBLIC(MyClass)

public:
    MyClassPrivate(MyClass *q);
    ~MyClassPrivate();

    void setValue(int);
    void printValue();

private:
    int      m_value;
    MyClass *q_ptr;
};
```

```
#endif // MYCLASSPRIVATE_H
```

As shown above, we will call the setValue(int) function and the printValue( ) function on MyClass. The printValue( ) function will in turn call the getValue( ) function of MyClass. The myfile\_p.cpp source code file should look like this

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

MyClassPrivate::MyClassPrivate(MyClass *q)
{
    q_ptr = q;
}

MyClassPrivate::~MyClassPrivate()
{
}

void MyClassPrivate::setValue(int val)
{
    m_value = val;
}

void MyClassPrivate::printValue()
{
    Q_Q(MyClass);

    qDebug() << " MyClass 의 m_value : " << q->getValue();
    qDebug() << " MyClassPrivate 의 m_value : " << m_value;
}
```

In this case, we're going to create a MyClass. Create the myfile.h header file as shown below.

```
#ifndef MYCLASS_H
#define MYCLASS_H

#include <QtGlobal>

#define MYCLASS_EXPORT Q_DECL_EXPORT

class MyClassPrivate;
```

```
class MYCLASS_EXPORT MyClass
{
    Q_DECLARE_PRIVATE(MyClass)

public:
    MyClass();
    ~MyClass();

    int getValue();
    void printValue();

private:
    int m_value;
    MyClassPrivate *d_ptr;
};

#endif // MYCLASS_H
```

Next, create the `myclass.cpp` source code file.

```
#include "myclass.h"
#include "myclass_p.h"

MyClass::MyClass()
{
    d_ptr = new MyClassPrivate(this);
    Q_D(MyClass);

    m_value = 100;
    d->setValue(200);
}

MyClass::~MyClass()
{
}

int MyClass::getValue()
{
    return m_value;
}

void MyClass::printValue()
{
```

```
Q_D(MyClass);
d->printValue();
}
```

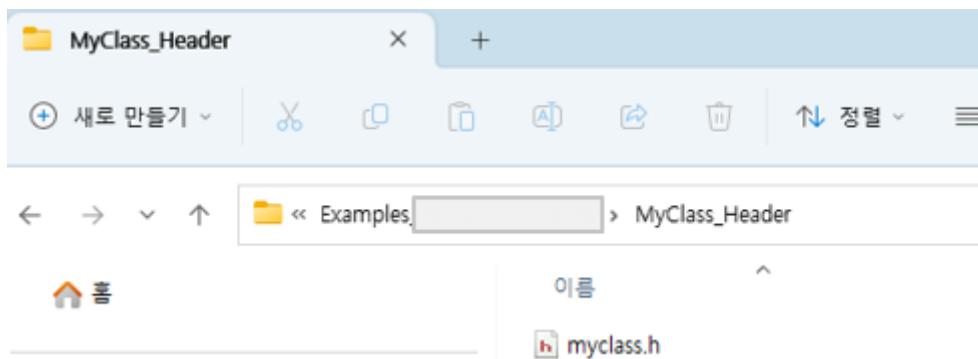
In the constructor function, call the `setValue( )` member function of `MyClassPrivate`. In order to call this function, the `Q_D( )` function must be used first; that is, any function that calls a member function of class `MyClassPrivate( )` must call the `Q_D( )` function.

And in order for `MyClassPrivate` to call a member function of `MyClass`, it must first call `Q_Q( )`.

Now that we've gotten this far, let's build the `MyClass` project. If it builds successfully without errors, `libMyClass.dll` (or `libMyClass.so` on Linux) should have been created.

Now let's write an example that uses `MyClass`. Before we start writing the example, create a `MyClass_Header` directory in the same directory location where your project is located and copy the `myclass.h` header file into it.

The reason for this is to make sure that we don't actually need to deploy `myclass_p.h`.

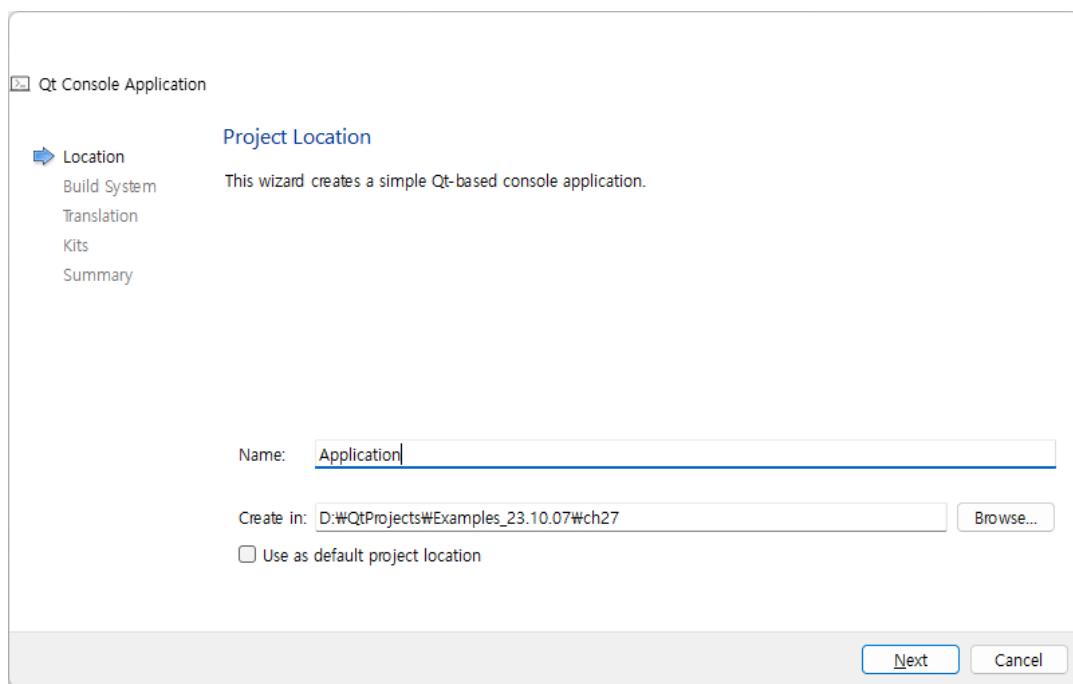
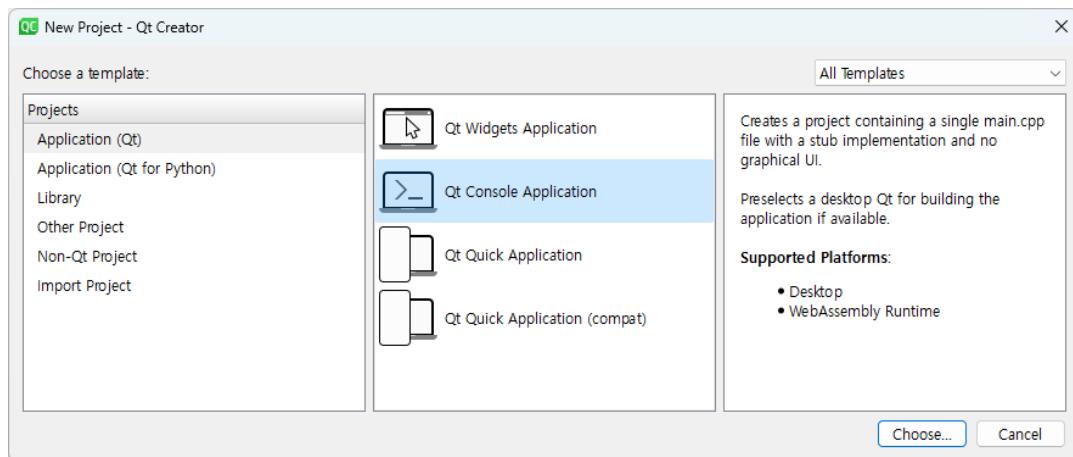


Copy the `myclass.h` header file to the `MyClass_Header` directory, as shown in the image above.

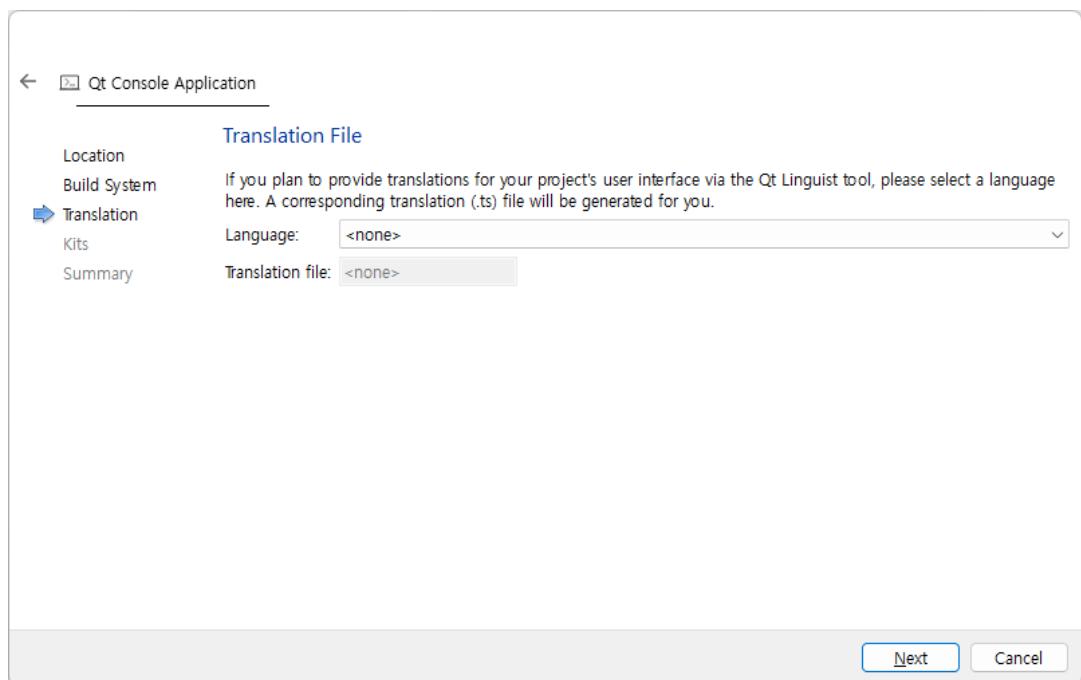
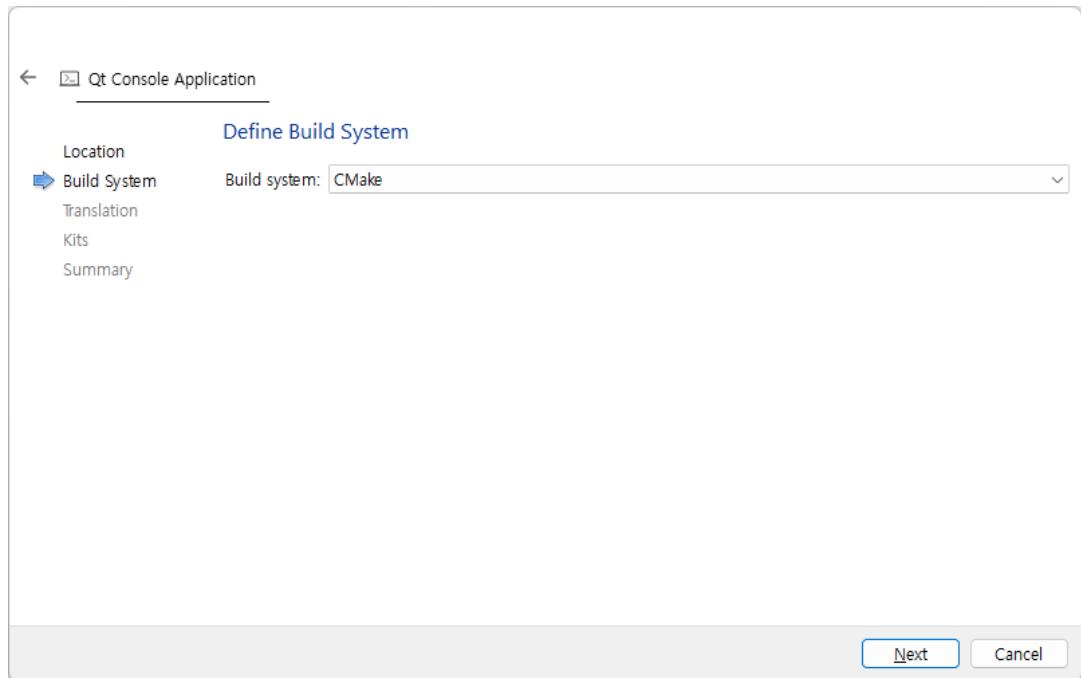
Next, let's create a project, this time with the name Application.

When creating the project, select Qt Console Application, as shown in the image below.

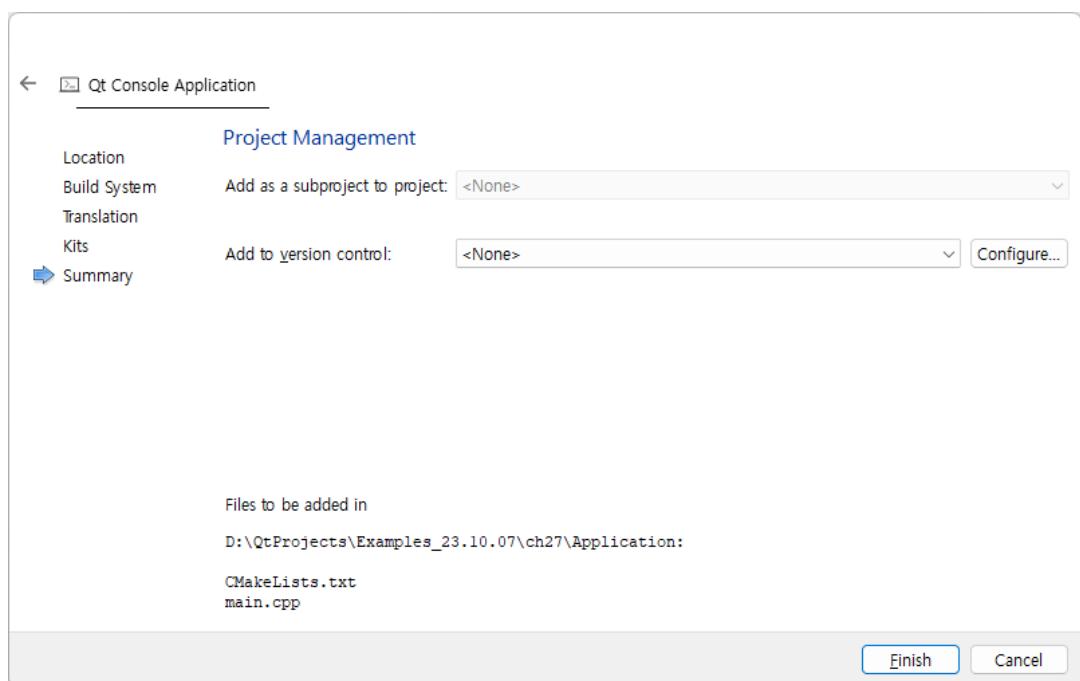
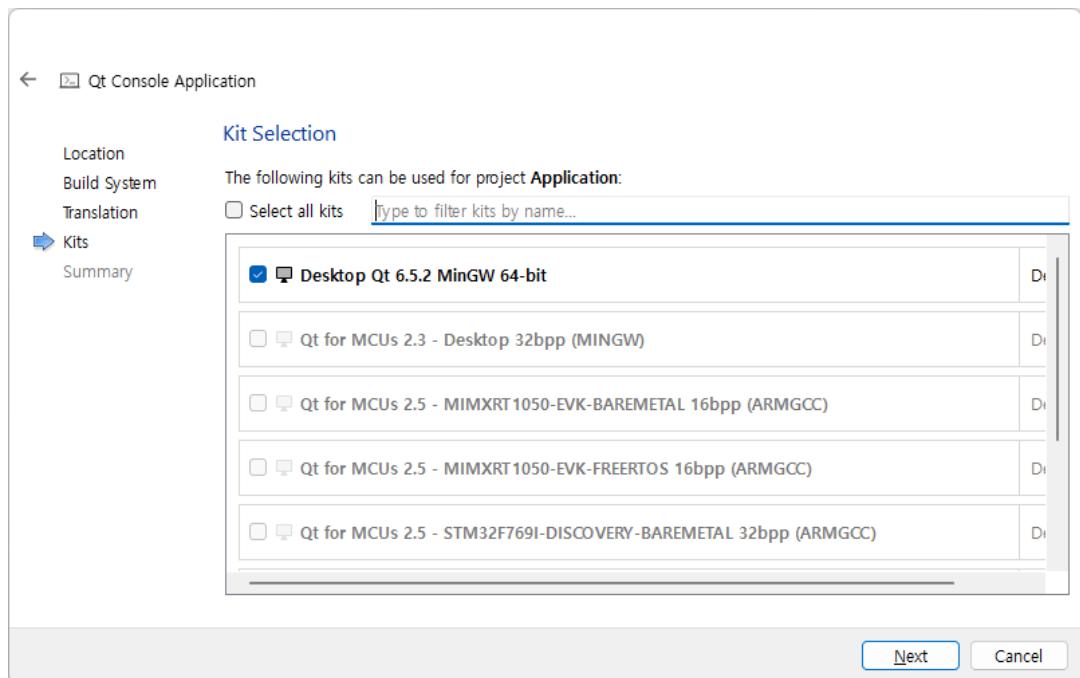
Jesus loves you.



When creating a project, select CMake as the build tool.



Jesus loves you.



Once you have created the project as shown above, open the CmakeList.txt file as shown in the image below and specify the location of the library file and header file to use the MyClass library as shown below.

And add the library name to be used in target\_link\_libraries( ).

```
cmake_minimum_required(VERSION 3.14)

project(Application LANGUAGES CXX)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

include_directories(..../MyClass_Header)
link_directories(..../build-MyClass-Desktop_Qt_6_5_2_MinGW_64_bit-Debug)

add_executable(Application
    main.cpp
)

find_package(Qt NAMES Qt6 Qt5 REQUIRED COMPONENTS Core)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Core)

target_link_libraries(Application MyClass)
target_link_libraries(Application Qt${QT_VERSION_MAJOR}::Core)

include(GNUInstallDirs)
install(TARGETS Application
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

Next, let's write the following code to use the MyClass library in main.cpp.

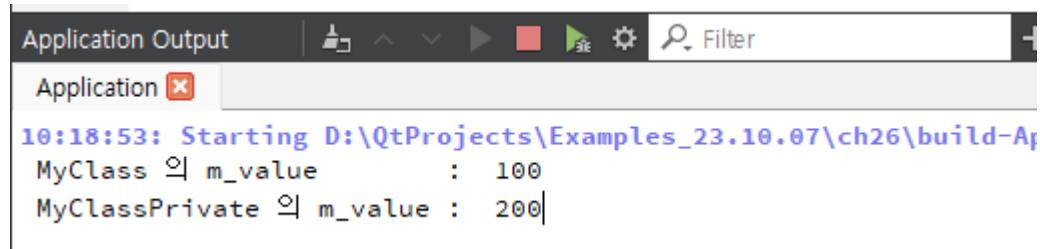
```
#include <QCoreApplication>
#include "myclass.h"

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

    MyClass myclass;
```

```
myclass.printValue();  
  
    return a.exec();  
}
```

The above source code calls the printValue( ) function of MyClass, which in turn calls the printValue( ) function of MyClassPrivate. This function in turn calls the printValue( ) function of MyClassPrivate. This results in the output shown in the figure below.



A screenshot of the Qt Creator application window showing the 'Application Output' tab. The log window displays the following text:

```
10:18:53: Starting D:\QtProjects\Examples_23.10.07\ch26\build-App\Debug  
MyClass 의 m_value : 100  
MyClassPrivate 의 m_value : 200
```

The library projects covered in this chapter can be found in the MyClass and MyClass\_Header directories. And for examples of using the libraries, see the projects in the Application directory.

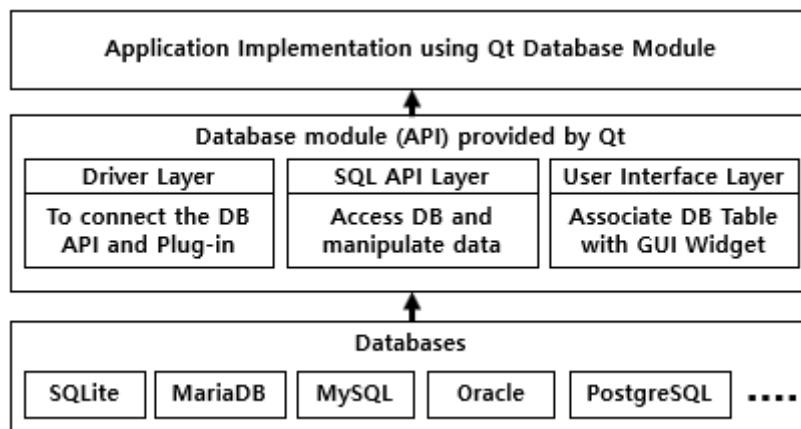
## 26. Database Programming

The database module (API) provided by Qt is easy to use compared to other development frameworks. The database modules provided by Qt allow you to access a variety of databases using a unified API.

For example, to query data stored in a MariaDB database using SQL statements, you can use the QSqlQuery class provided by Qt.

And to query data stored in a SQLite database using SQL statements, you can use the same QSqlQuery class that accessed the MariaDB database.

In other words, you can develop Qt applications using the common database module provided by Qt without using the APIs provided by each database to access and use the data.



As shown in the figure above, whether you use SQLite or MariaDB, you can use the Qt SQL module provided by Qt to handle each database.

This means that Qt can use a common database module (API) to access different databases.

The database modules provided by Qt can be categorized into three main categories.

### ① Driver Layer

This layer is responsible for acting as a low-level bridge to connect to a specific database,

i.e., it acts as a driver for connecting to the database.

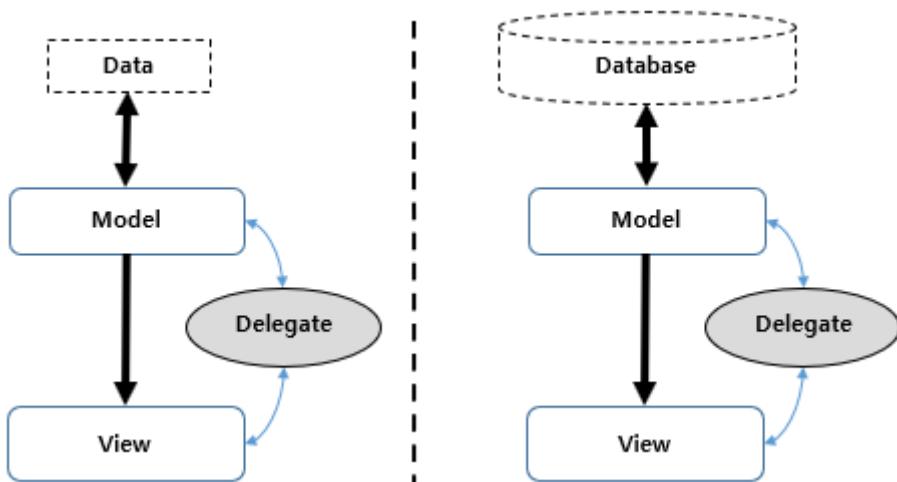
## ② SQL API Layer

It provides classes for connecting to a database and provides a database to utilize data in database tables. For example, you can use the QSqlDatabase class to connect to a database and the QSqlQuery class to query it.

## ③ User Interface Layer

This layer was covered earlier when we dealt with Model/View when dealing with data. A Model acts like a Container where data is stored. A View is a GUI widget, such as QTableView, called a View or View widget. Just as we associated a Model with a View widget, Qt provides a database Model.

For example, a QSqlQueryModel queries data from a table stored in a database and stores the data in a Model. Therefore, you can associate a QSqlQueryModel with a View widget.



As shown in the figure above, on the left is the general Model/View concept and on the right is the concept of a database Model class and a View. In this way, you can use the database Model classes provided by Qt to connect with the View widget. Qt provides QSqlQueryModel, QSqlTableModel, and QSqlRelationalTableModel classes as the most commonly used Model classes.

To use the database modules provided by Qt, you must use the "sql" keyword in your

project file, as follows

```
Qt += sql
```

If you're using CMake, you can specify it like this

```
find_package(Qt6 REQUIRED COMPONENTS Sql)
target_link_libraries(mytarget PRIVATE Qt6::Sql)
```

- ✓ Connect to the database

In order to send/receive data to/from the other party in TCP, you must first establish a connection, just as you must establish a connection before sending/receiving packets.

```
QSqlDatabase db1 = QSqlDatabase::addDatabase("QMYSQL");
QSqlDatabase db2 = QSqlDatabase::addDatabase("QSQLITE");
QSqlDatabase db3 = QSqlDatabase::addDatabase("QPSQL");
```

The QSqlDatabase class provides functionality for connecting to a database. It also supports connecting using a unique user account. The following is example source code for accessing a database.

```
QSqlDatabase db = QSqlDatabase::addDatabase("QMYSQL");

db.setHostName("bigblue");           // IP or DNS Host name
db.setDatabaseName("flightdb");     // DB name
db.setUserName("acarlson");        // Account
db.setPassword("1uTbSbAs");        // Password

bool ok = db.open();
```

As shown in the example source code above, you must specify the name of the database driver you want to use as the first argument to the addDatabase( ) function. The following table lists the database driver names provided by Qt.

Driver name	Description
QDB2	IBM DB2 7.1 and later versions
QIBASE	Borland InterBase
QMYSQ	MariaDB or MySQL

QOCI	Oracle Call Interface Driver
QODBC	Open Database Connectivity
QPSQL	PostgreSQL 7.3 later
QSQLITE2	SQLite Version 2
QSQLITE	SQLite Version 3
QTDS	Sybase Adaptive Server

- ✓ QUERY the database using SQL statements

To query data in a database table, you can use the QSqlQuery class, as shown in the following example.

```
QSqlQuery query;  
  
QString qry;  
qry = "SELECT name, salary FROM employee WHERE salary > 500";  
query.exec(qry);
```

Then, to retrieve the next data in the table in turn and get the data for the desired field, you can use

```
while (query.next())  
{  
    QString name = query.value(0).toString();  
    int salary = query.value(1).toInt();  
}  
...
```

To insert data into a table, you can use the above as follows.

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

When using SQL statements to handle large amounts of data, you can use the Placeholder method or the Positioning method. For performance reasons, the Placeholder method is recommended. The following is an example source code of the Placeholder method.

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

    query.exec();
}
```

Here's an example of the Positioning method

```
QSqlQuery query;
query.prepare( "INSERT INTO employee(id, name, salary) VALUES (?, ?, ?)");

for(...) {
    query.addBindValue(1001);
    query.addBindValue("Thad Beaumont");
    query.addBindValue(65000);

    query.exec();
}
```

#### ✓ Transaction

Qt can handle transactions using the member functions provided by the QSqlDatabase class. The following example shows an example using the transaction( ) and commit( ) member functions.

```
QSqlDatabase::database().transaction();
QSqlQuery query;
query.exec( "SELECT id FROM employee WHERE name = 'Torild Halvorsen'");

if (query.next()) {
    int employeeId = query.value(0).toInt();
    query.exec("INSERT INTO project (id, name, ownerid) "
              "VALUES (201, 'Manhattan Project', "
              + QString::number(employeeId) + ')');
}
```

```
QSqlDatabase::database().commit();
```

✓ Database Model Class

Among the database model classes provided by Qt, the most commonly used classes are the following

모델 명	설명
QSqlQueryModel	How to use SQL to READ data
QSqlTableModel	Suitable for reading/writing data from a table.
QSqlRelationalTableModel	How to use foreign keys

The following example source code shows how to use the QSqlQueryModel class to store queried data in a Model.

```
QSqlQueryModel model;
model.setQuery("SELECT * FROM employee");

for (int i = 0; i < model.rowCount(); ++i)
{
    int id = model.record(i).value("id").toInt();
    QString name = model.record(i).value("name").toString();
    qDebug() << id << name;
}
```

You can set an SQL query using the `setQuery()` member function of the `QSqlQueryModel` class and call the `record(int)` member function to read the records of a table stored in the database. The following example shows the source code for an example using the `QSqlTableModel` class.

```
QSqlTableModel model;
model.setTable("employee");
...
for (int i = 0; i < model.rowCount(); ++i) {
    QSqlRecord record = model.record(i);
    double salary = record.value("salary").toInt();
    salary *= 1.1;
    record.setValue("salary", salary);
    model.setRecord(i, record);
}
```

```
model.submitAll();
```

The records stored in the table can be read with the record( ) member function, and the records in each row can be modified with the setRecord( ) member function.

Using the setData( ) member function, you can modify a specific row and column as follows

```
QSqlTableModel model;
model.setTable("employee");

model.setData(model.index(row, column), 75000);
model.submitAll();
```

And you can delete data in bulk using the removeRows( ) member function of the QSqlTableModel class.

```
model.removeRows(row, 5);
model.submitAll();
```

The first argument to the removeRows( ) member function is the number of the starting row to delete. The second argument is the number of records to delete. Next, let's look at how to use the QSqlRelationalTableModel class.

This class provides a Model that allows you to search a table using a foreign key. For example, consider two tables as shown below.

MainTable	
Field name	Type
id	INTERGER
Date	DATE
Time	TIME
Hostname	INTERGER
IP	INTERGER

DeviceTable	
Field name	Type
id	INTERGER
Hostname	VARCHAR(255)
IP	VARCHAR(16)

As shown in the table above, we have a MainTable and a DeviceTable. If you want to insert a record in the DeviceTable first, and then insert the Hostname and IP from the DeviceTable when inserting data from the MainTable, you can use the following code.

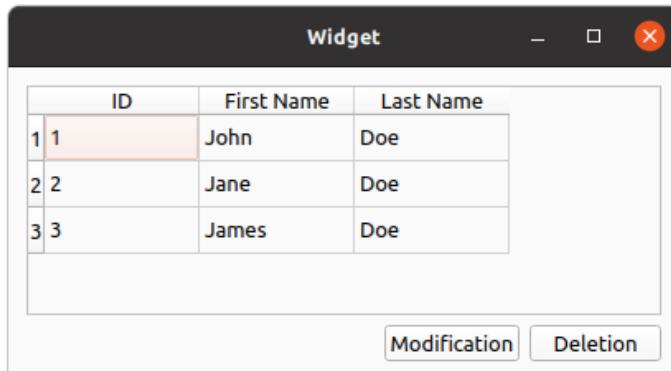
```
QSqlRelationalTableModel *modelMain;  
QSqlRelationalTableModel *modelDevice;  
  
modelMain->setRelation(3, QSqlRelation("DeviceTable", "id", "Hostname"));  
modelMain->setRelation(4, QSqlRelation("DeviceTable", "id", "IP"));  
...
```

As shown in the example above, the Hostname and IP record field data from the "DeviceTable" table are inserted into the modelMain object identically using the setRelation( ) function provided by the QSqlRelationalTableModel class.

So far we have seen the database modules provided by Qt. Next, let's try writing our own examples.

- ✓ Example using SQLite database

For this example, we're going to write an example using SQLite database. Here is a screen shot of the example.



As shown in the image above, when you build and run the project, it will create a SQLite database. Once the database is created, a table named "names" is created. This is an example of inserting data into this table and outputting the data to the QTableView widget.

Click the [Edit] button to change the First Name of the record with ID 1 to 'Eddy'. Then, change the Last Name to 'Kim'. Clicking the [Delete] button deletes the data with ID 1.

Create a project based on QWidget and write the widget.h header source code as follows. You can refer to the 00\_SQLite\_Example directory for the complete source code.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QSqlDatabase>
#include <QSqlTableModel>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;

    QSqlDatabase m_db;
    QSqlTableModel *m_model;

    bool initializeDataBase();
    void creationTable();
    void insertDataToTable();
    void initializeModel();

private slots:
```

```
void slot_pbtUpdate();
void slot_pbtDelete();
};

#endif // WIDGET_H
```

The initializeDataBase( ) function creates a SQLite database file. The creationTable( ) function creates a table in the created database. insertDataToTable( ) inserts data into the created table.

The initializeModel( ) function creates a QSqlTableModel class object, sets the names table to the model data, and sets the headers. The following example shows the widget.cpp source code.

```
#include "widget.h"
#include "./ui_widget.h"

#include <QSqlQuery>
#include <QSqlError>
#include <QFile>
#include <QDebug>

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

    if( initializeDataBase() )
    {
        creationTable();
        insertDataToTable();
        initializeModel();

        ui->tableView->setModel(m_model);

        connect(ui->pbtUpdate, SIGNAL(pressed()),
                this,           SLOT(slot_pbtUpdate()));
        connect(ui->pbtDelete, SIGNAL(pressed()),
                this,           SLOT(slot_pbtDelete()));

    }
}
```

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

bool Widget::initializeDataBase()
{
    QFile::remove("./my.db");

    m_db = QSqlDatabase::addDatabase("QSQLITE");
    m_db.setDatabaseName("./my.db");

    if( !m_db.open() ) {
        qDebug() << Q_FUNC_INFO << m_db.lastError().text();
        return false;
    }

    return true;
}

void Widget::creationTable()
{
    QSqlQuery qry;
    qry.prepare( "CREATE TABLE IF NOT EXISTS names "
                "("
                "    id INTEGER UNIQUE PRIMARY KEY, "
                "    firstname VARCHAR(30), "
                "    lastname  VARCHAR(30)"
                ")" );
}

if( !qry.exec() ) {
    qDebug() << qry.lastError().text();
}
}

void Widget::insertDataToTable()
{
    QSqlQuery qry;

    qry.prepare( "INSERT INTO names "
                "(id, firstname, lastname) "

```

```
        "VALUES "
        "(1, 'John', 'Doe')" );
if( !qry.exec() )
    qDebug() << qry.lastError();

qry.prepare( "INSERT INTO names "
            "(id, firstname, lastname) "
            "VALUES "
            "(2, 'Jane', 'Doe')" );
if( !qry.exec() )
    qDebug() << qry.lastError();

qry.prepare( "INSERT INTO names "
            "(id, firstname, lastname) "
            "VALUES "
            "(3, 'James', 'Doe')" );
if( !qry.exec() )
    qDebug() << qry.lastError();

}

void Widget::initializeModel()
{
    m_model = new QSqlTableModel(this, m_db);

    m_model->setTable("names");
    m_model->setEditStrategy(QSqlTableModel::OnManualSubmit);
    m_model->select();

    m_model->setHeaderData(0, Qt::Horizontal, tr("ID"));
    m_model->setHeaderData(1, Qt::Horizontal, tr("First Name"));
    m_model->setHeaderData(2, Qt::Horizontal, tr("Last Name"));
}

void Widget::slot_pbtUpdate()
{
    QSqlQuery qry;
    qry.prepare( "UPDATE names "
                "SET firstname = 'Eddy', "
                "lastname = 'Kim' WHERE id = 1" );
    if( !qry.exec() )
        qDebug() << qry.lastError();
}
```

```

m_model->setTable("names");
m_model->select();
ui->tableView->setModel(m_model);
}

void Widget::slot_pbtDelete()
{
    QSqlQuery qry;

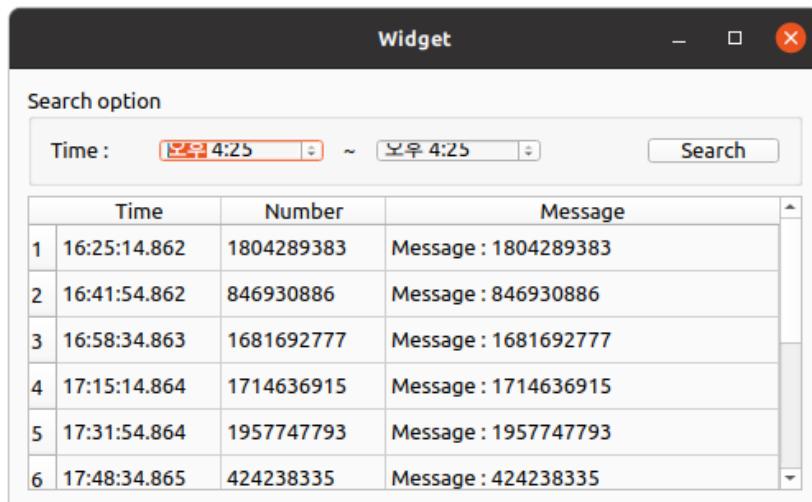
    qry.prepare( "DELETE FROM names WHERE id = 1" );
    if( !qry.exec() )
        qDebug() << qry.lastError();

    m_model->setTable("names");
    m_model->select();
    ui->tableView->setModel(m_model);
}

```

✓ Database table search example

이번 예제에서는 데이터베이스 테이블에 저장된 데이터를 QSqlTableModel로 가져와 QTableView 위젯에 출력한다. 그리고 시간을 기준으로 검색하는 기능을 구현해 보도록 하자.



As shown in the figure above, if you click the [Search] button based on the time, only

the data that matches the time condition is displayed on the screen. To search the data, you can use the `setFilter( )` member function provided by the `QSqlTableModel` class.

When you create a project, create a Widget class that inherits from the `QWidget` class, and additionally create a `DatabaseHandler` class. The following example source code is the header code for the `DatabaseHandler` class. For the complete source code, see the `01_Search_Example` directory.

```
#ifndef DATABASEHANDLER_H
#define DATABASEHANDLER_H

#include <QObject>
#include <QSql>
#include <QSqlQuery>
#include <QSqlError>
#include <QSqlDatabase>
#include <QFile>
#include <QDate>
#include <QDebug>

#define DATABASE_HOSTNAME    "QtDevDataBase"
#define DATABASE_NAME        "qtdev.db"

class DatabaseHandler : public QObject
{
    Q_OBJECT
public:
    explicit DatabaseHandler(QObject *parent = nullptr);
    ~DatabaseHandler();

    void connectToDataBase();
    bool insertIntoTable(const QVariantList &data);

private:
    QSqlDatabase db;

private:
    bool openDataBase();
    bool restoreDataBase();
    void closeDataBase();
    bool createTable();
};
```

```
#endif // DATABASEHANDLER_H
```

The openDataBase( ) function creates a database file. It declares an object of class QSqlDatabase. The restoreDataBase( ) function calls the openDataBase( ) function.

And if this function returns true, it calls the createTable( ) function. This function creates a table defined with the name RECEIVE\_MAIL. The following is the source code for databasehandler.cpp.

```
#include "databasehandler.h"
#include <QDir>
#include <QDebug>

DatabaseHandler::DatabaseHandler(QObject *parent)
    : QObject(parent)
{
}

void DatabaseHandler::connectToDataBase()
{
    QString dirPath = QDir::currentPath();
    dirPath.append("/" DATABASE_NAME);

    QFile(dirPath).remove(dirPath);
    this->restoreDataBase();
}

bool DatabaseHandler::restoreDataBase()
{
    if(this->openDataBase()){
        if(!this->createTable()){
            return false;
        } else {
            return true;
        }
    } else {
        qDebug() << "Database connection failed.";
        return false;
    }
}
```

```
bool DatabaseHandler::openDataBase()
{
    QString dirPath = QDir::currentPath();
    dirPath.append("/" DATABASE_NAME);

    db = QSqlDatabase::addDatabase("QSQLITE");
    db.setHostName(DATABASE_HOSTNAME);
    db.setDatabaseName(dirPath);
    if(db.open()){
        return true;
    } else {
        return false;
    }
}

void DatabaseHandler::closeDataBase()
{
    db.close();
}

bool DatabaseHandler::createTable()
{
    QSqlQuery query;
    if(!query.exec( "CREATE TABLE RECEIVE_MAIL ("
                    "id INTEGER PRIMARY KEY AUTOINCREMENT, "
                    "TIME      TIME          NOT NULL,"
                    "MESSAGE   INTEGER       NOT NULL,"
                    "RANDOM    VARCHAR(255) NOT NULL"
                    " )"
                    )){

        qDebug() << "Table creation failed : "
                << query.lastError().text();
        return false;
    } else {
        return true;
    }
}

bool DatabaseHandler::insertIntoTable(const QVariantList &data)
{
    QSqlQuery query;
    query.prepare("INSERT INTO "

```

```
"RECEIVE_MAIL (TIME, RANDOM, MESSAGE) "
"VALUES (:TIME, :RANDOM, :MESSAGE )";

query.bindValue(":TIME",           data[0].toTime());
query.bindValue(":MESSAGE",       data[1].toInt());
query.bindValue(":RANDOM",        data[2].toString());

if(!query.exec()){
    qDebug() << "Insert error - "
                << query.lastError().text();
    return false;
} else {
    return true;
}
}

DatabaseHandler::~DatabaseHandler()
{
}
```

The insertIntoTable( ) function inserts a record into the table you created. When inserting records into the table, it uses the placeholder method to insert the records. The following example source code is the widget.h header source code.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QSqlTableModel>
#include "databasehandler.h"

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();
```

```
private:  
    Ui::Widget *ui;  
  
    DatabaseHandler *m_dbHandler;  
    QSqlTableModel *m_model;  
  
    void setupModel(const QString &tableName, const QStringList &headers);  
    void createUserInterface();  
  
private slots:  
    void onPushButton();  
  
};  
#endif // WIDGET_H
```

Among the above functions, onPushButton( ) is a Slot function that is called when the [Search] button is clicked. The setupModel( ) function declares a QSqlTableModel class object, declares headers, and sorts them in ascending order.

The createUserInterface( ) function sets up the QTableView widget that we placed on the GUI, and sets the time in the QTimeEdit to the current time to retrieve the time as seen in the GUI. The following is the widget.cpp source code.

```
#include "widget.h"  
#include "./ui_widget.h"  
  
Widget::Widget(QWidget *parent)  
    : QWidget(parent)  
    , ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
  
    m_dbHandler = new DatabaseHandler();  
    m_dbHandler->connectToDataBase();  
  
    for(int i = 0; i < 10; i++)  
    {  
        QVariantList data;  
  
        QTime currTime = QTime::currentTime();
```

```
currTime = currTime.addSecs(i*1000);

int random = rand();

data.append(currTime);
data.append(random);
data.append("Message : " + QString::number(random));

m_dbHandler->insertIntoTable(data);
}

setupModel("RECEIVE_MAIL",
QStringList() << "ID"
<< "Time"
<< "Number"
<< "Message"
);

createUserInterface();
}

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

void Widget::setupModel(const QString &tableName,
                      const QStringList &headers)
{
    m_model = new QSqlTableModel(this);
    m_model->setTable(tableName);

    for(int i = 0, j = 0; i < m_model->columnCount(); i++, j++) {
        m_model->setHeaderData(i, Qt::Horizontal, headers[j]);
    }

    m_model->setSort(0,Qt::AscendingOrder);
}

void Widget::createUserInterface()
{
    ui->tableView->setModel(m_model);
```

```
ui->tableView->setColumnHidden(0, true);
ui->tableView->setSelectionBehavior(QAbstractItemView::SelectRows);
ui->tableView->setSelectionMode(QAbstractItemView::SingleSelection);
ui->tableView->resizeColumnsToContents();
ui->tableView->horizontalHeader()->setStretchLastSection(true);

for(int i = 0 ; i < 4 ; i++)
    ui->tableView->setColumnWidth(i, 100);

m_model->select();
ui->timeFROM-> setTime(QTime::currentTime());
ui->timeT0-> setTime(QTime::currentTime());

connect(ui->pbtSearch, SIGNAL(pressed()), this, SLOT(onPushButton()));
}

void Widget::onPushButton()
{
    QString str = QString("TIME between '%3' and '%4'")
                  .arg(ui->timeFROM->time().toString("hh:mm:ss"));

    m_model->setFilter(QString( "TIME between '%3' and '%4' ")
                        .arg(ui->timeFROM->time().toString("hh:mm:ss"),
                            ui->timeT0->time().toString("hh:mm:ss")));

    m_model->select();
}
```

## 27. Qt for Android

In this chapter, you will learn how to develop Android applications using Qt. This chapter consists of two sections.

The first lesson covers how to work with Android applications using Qt on the MS Windows operating system. In this lesson, you will learn how to build an environment on the MS Windows operating system. We will then develop an Android application using Qt and install and run the APK on an Android smartphone.

The second lesson is about working with Android applications using Qt on Linux. In this lesson, you will learn how to build an environment on the Linux operating system. We will then develop an Android application using Qt and install and run the APK on an Android smartphone.

## 27.1. Android development environment on MS Windows

To build an Android development environment on MS Windows, when installing Qt, you need to select the "Android" option in the installation dialogue, as shown in the figure below.



After selecting and installing Qt as shown above, you need to install Qt to develop Android-based apps. After installing Qt, we need to install some packages. First, let's install the JDK.

- ✓ Download and install the Open JDK

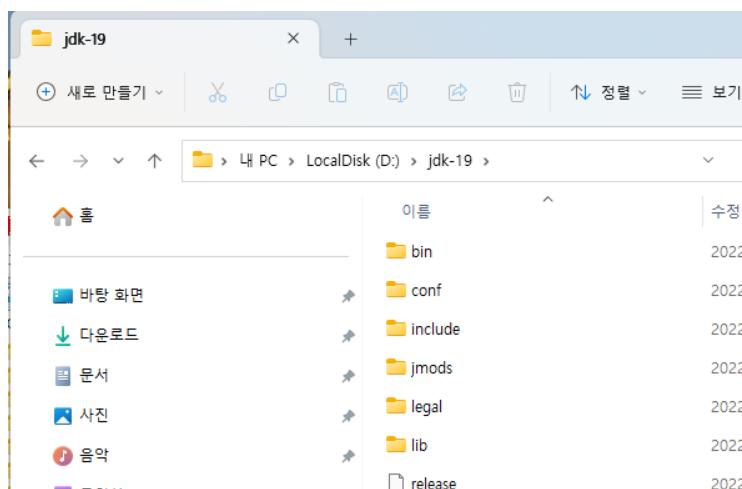
You can download the Open JDK by visiting the Open DJK site ([jdk.java.net/java-se-ri/19](http://jdk.java.net/java-se-ri/19)).

The screenshot shows a web browser window with the URL [jdk.java.net/java-se-ri/19](https://jdk.java.net/java-se-ri/19). The page title is "Java Platform, Standard Edition 19 Reference Implementations". The left sidebar lists various Java versions: GA Releases (JDK 21, JavaFX 21, JMC 8), Early-Access Releases (JDK 22, JavaFX 22, jextract, Loom, Valhalla), Reference Implementations (Java SE 21, Java SE 20, Java SE 19, Java SE 18, Java SE 17, Java SE 16, Java SE 15, Java SE 14, Java SE 13, Java SE 12, Java SE 11, Java SE 10, Java SE 9, Java SE 8, Java SE 7), Feedback Report & bug, and Archive. A red box highlights "Java SE 19". The main content area discusses the official Reference Implementation for Java SE 19 (JSR 394) based on open-source code from the JDK 19 Project in the OpenJDK Community. It mentions that binaries are available under the GNU General Public License version 2 with the Classpath Exception. A bolded note states: "These binaries are for reference use only!". Below this, it says: "These binaries are provided for use by implementers of the Java SE 19 Platform Specification and are for reference purposes only. This Reference Implementation has been approved through the Java Community Process. Production-ready binaries under the GPL are available from Oracle; and will be in most popular Linux distributions." The "RI Binary (build 19+36) under the GNU General Public License version 2" section contains two links: "Oracle Linux 8.5 x64 Java Development Kit (sha256) 187 MB" and "Windows 11 x64 Java Development Kit (sha256) 186 MB", with the Windows link highlighted by a red box. The "RI Source Code" section links to a single zip file (sha256) 170 MB. The "International use restrictions" section notes that due to limited intellectual property protection and enforcement in certain countries, the JDK source code may only be distributed to an authorized list of users.

As shown above, select the Java SE 19 version and choose the Windows 11 version. If you are using MS Windows 10 or lower, you can choose another Java SE 18 or lower version.

Unzip the downloaded file and move it to the directory shown below (you can move it to any location of your choice).

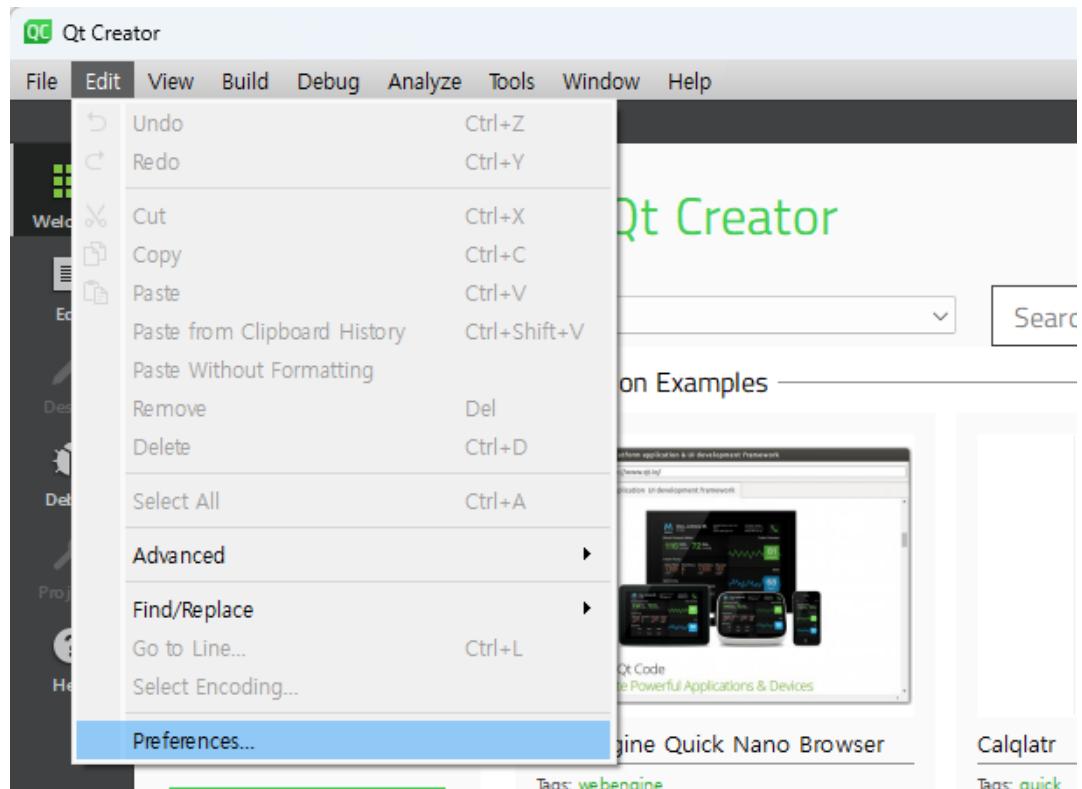
- ✓ Directory location: D:\jdk-19



Jesus loves you.

One caveat to using the Open JDK is that there are incompatible versions. If you have an incompatible version, using a lower version will solve the problem.

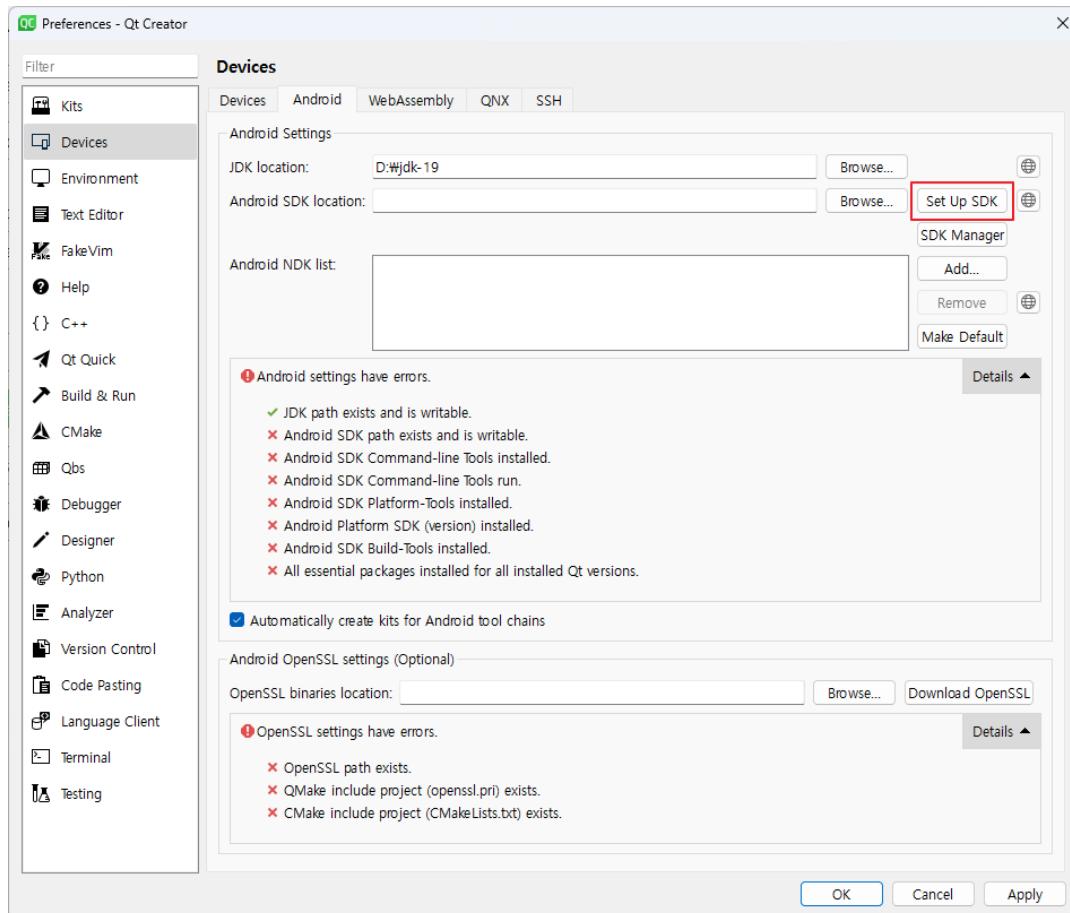
Next, run Qt Creator and select [Edit] -> [Preferences].



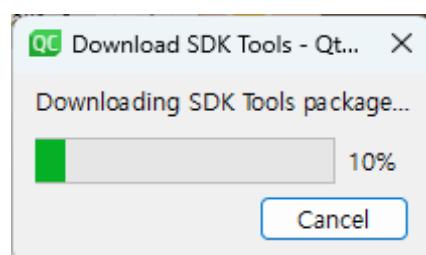
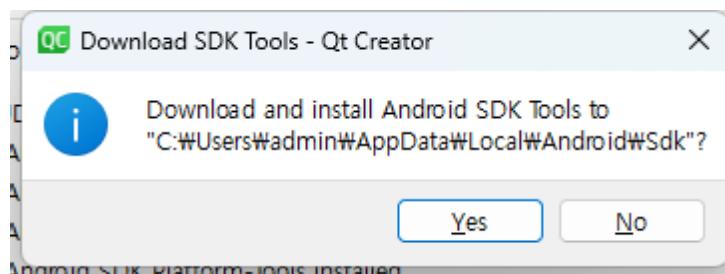
[Preferences and you should see that the JDK location is set. Next, we need to install the Android SDK.

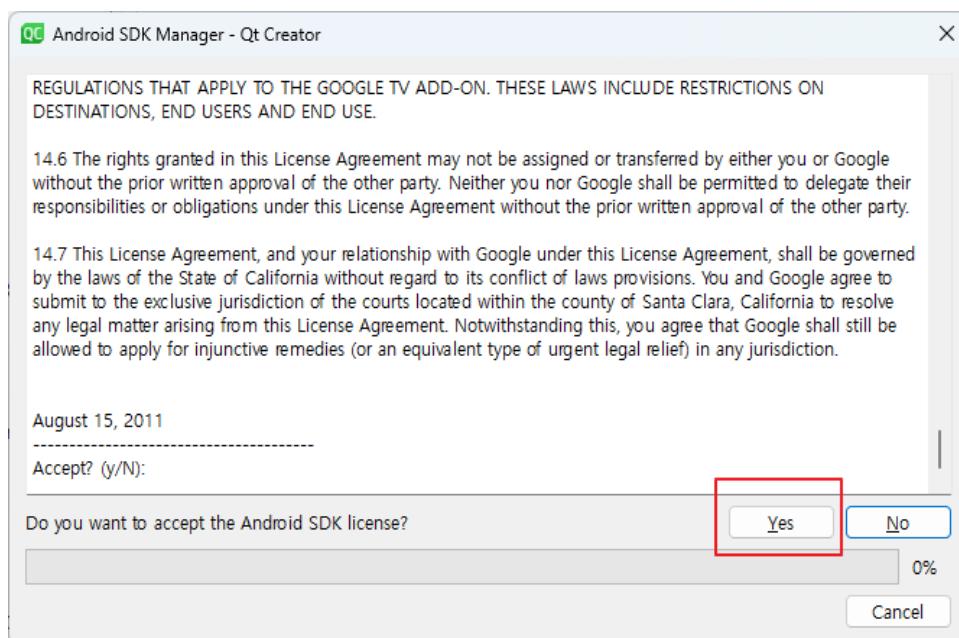
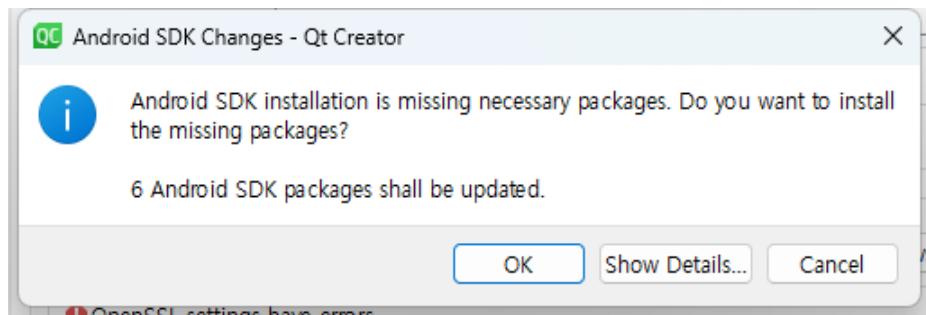
As shown in the image below, clicking the [Set Up SDK] button provides an easy way to download and install the SDK, so let's click the [Set Up SDK] button to install the SDK.

Jesus loves you.



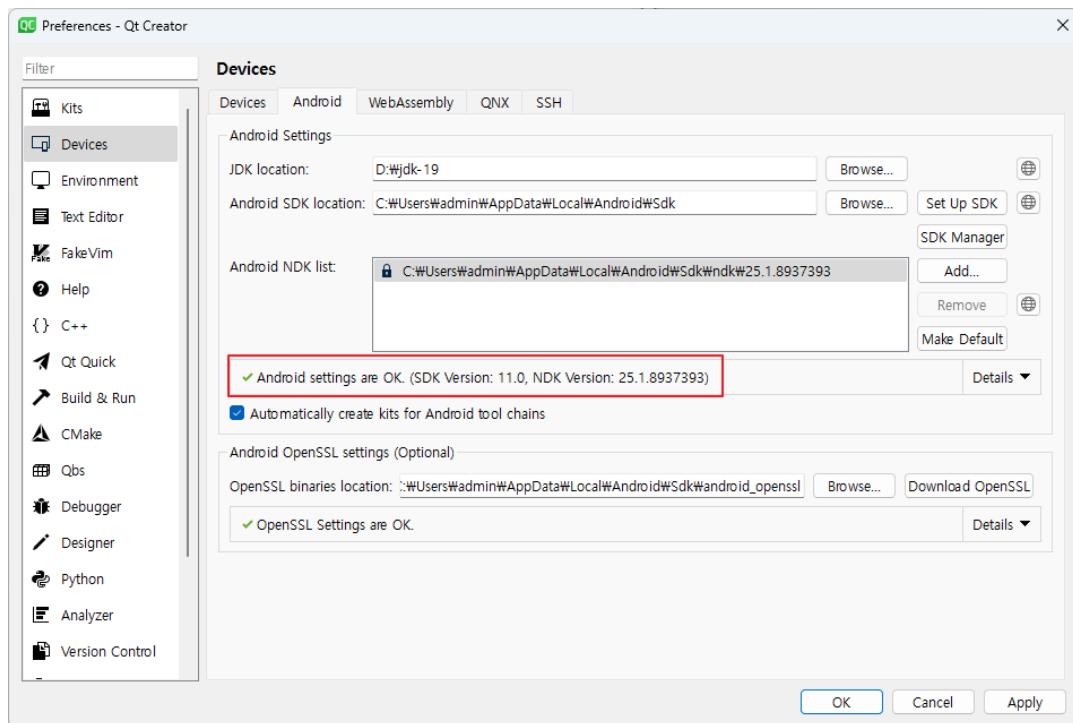
Click [Yes] in the dialogue below.





You will be asked to click the [Yes] button several times in the dialogue above.

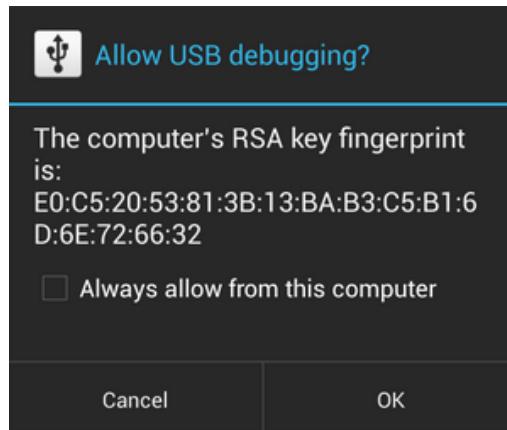
When the package installation is complete, you can see that the package download and setup is complete, as shown in the image below.



Also, sometimes the OpenSSL binaries location is not automatically installed. In this case, click the [Download OpenSSL] button to go to the site where you can download OpenSSL. After downloading from the site, you can specify the directory in the unzipped directory.

We've just finished configuring the Android app development environment using Qt. Now it's time to set up the actual Android device where you'll be debugging and deploying your app.

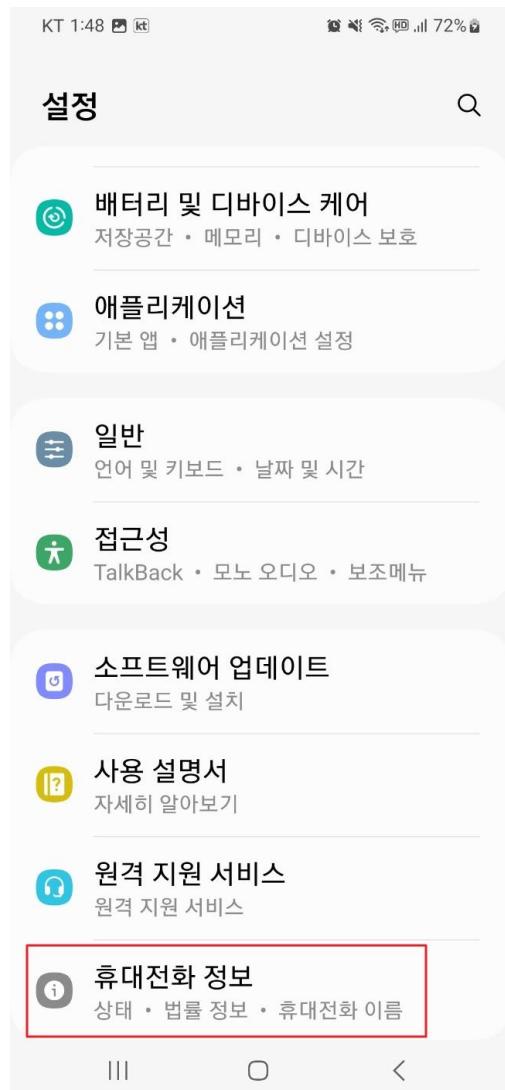
In order to debug the Qt app on the Android device, you need to enable USB debugging on Android as shown in the figure below.



Allow USB debugging mode on the actual Android device as shown in the picture above. You can find detailed instructions on how to allow USB debugging of your Android device by searching the internet.

- Enable Android Device Developer Options and Allow USB Debugging Mode

In order to debug and deploy applications written in Qt on Android devices, you need to enable developer options. Enabling developer options may also vary slightly depending on the manufacturer. Enter the settings menu as shown in the figure below.



From the Settings menu, select About phone.

< 휴대전화 정보



RF9R4038K2W

IMEI

351142481232440

상태 정보

법률 정보

규제 정보

소프트웨어 정보

배터리 정보

삼성전자 AS 문의/예약

1588-3366 / www.3366.co.kr

다른 기능을 찾고 있나요?

[소프트웨어 업데이트](#)

[초기화](#)

[문의하기](#)



## < 소프트웨어 정보

### 안드로이드 버전

13

### Google Play 시스템 업데이트

2023년 6월 1일

### 기저 대역 버전

A325NKOU4DWH3

### 커널 버전

4.14.186-27270957

#1 Fri Aug 11 16:36:23 KST 2023

### 빌드번호

TP1A.220624.014.A325NKSU4DWH3

### SE for Android 상태

Enforcing

SEPF\_SM-A325N\_12\_0001

Fri Aug 11 16:48:38 2023

### Knox 버전

Knox 3.9

Knox API level 36

HDM 2.0 - 1F



Click [Build Number] 7 times as shown in the image above to enable developer mode as

shown in the image below. Select Developer Options.

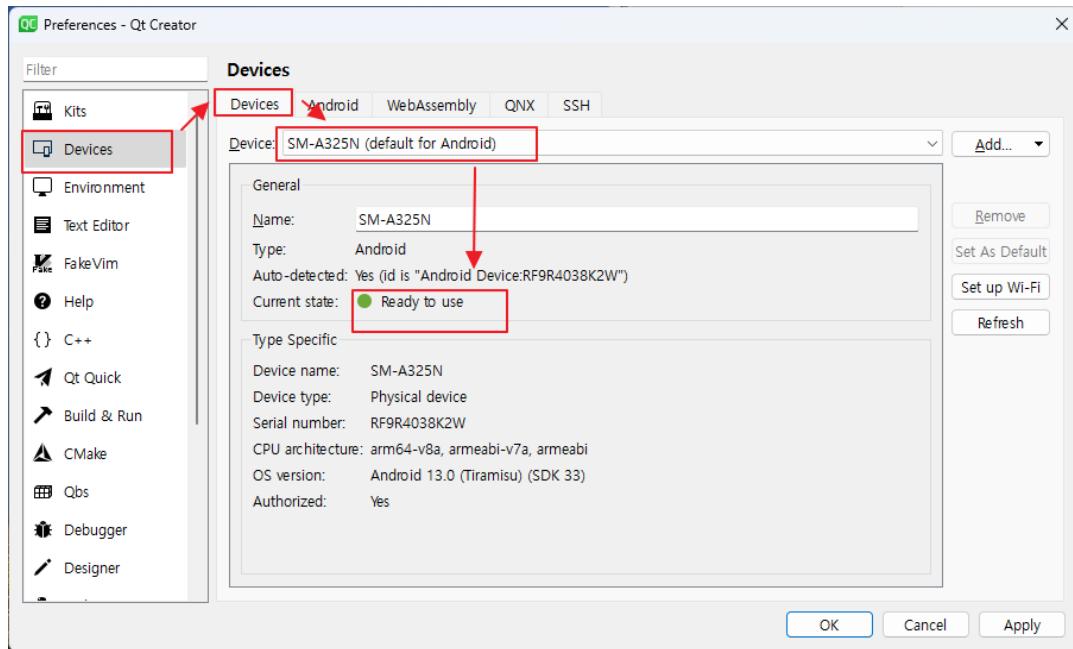


Enable USB debugging mode as shown in the figure below.



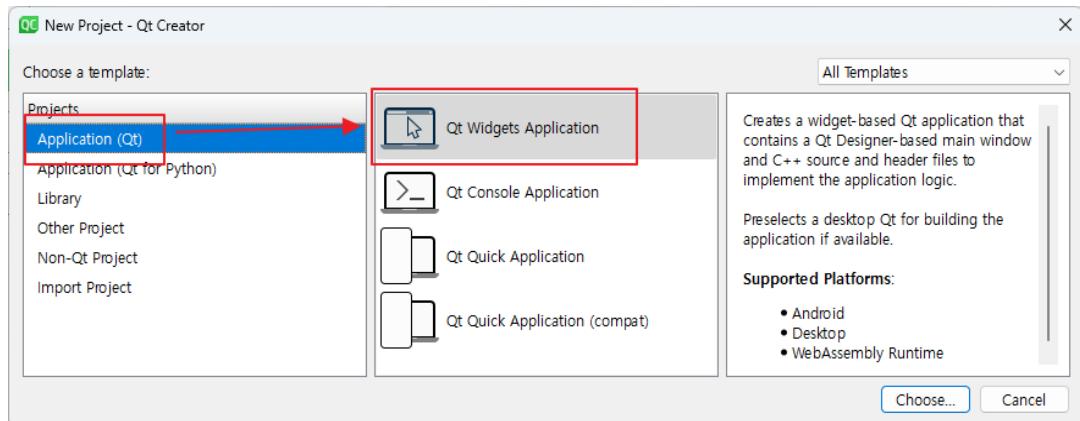
If you have completed this step, the next step is to connect the Android device in Qt Creator. Click on the [Edit] > [Preferences] menu. Then select [Device] on the right, and then select your actual Android device in the Devices section.

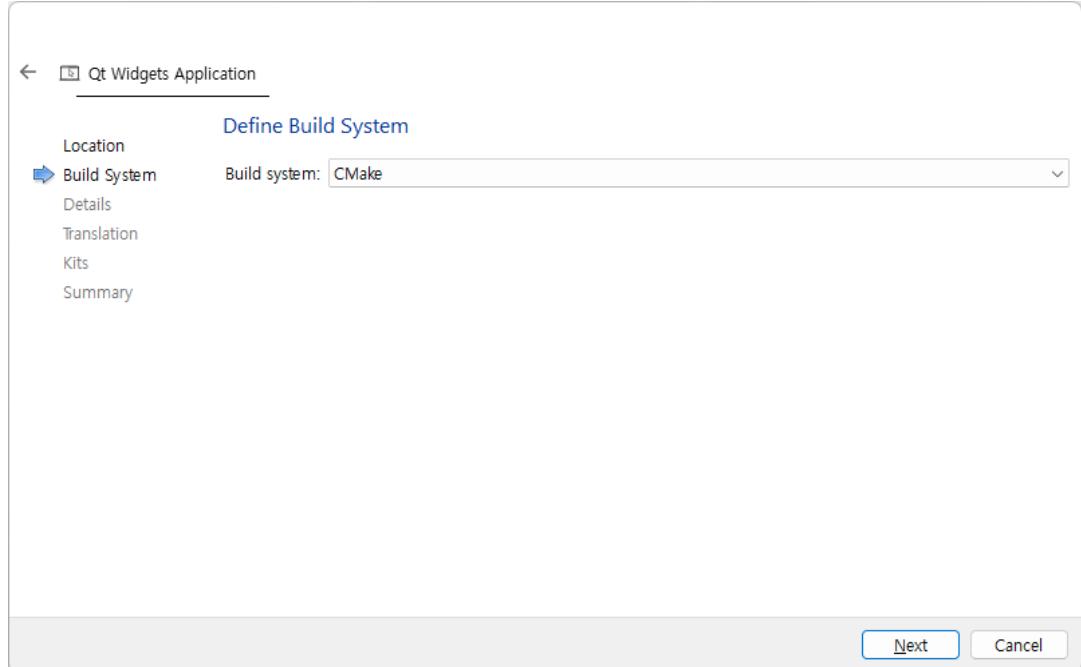
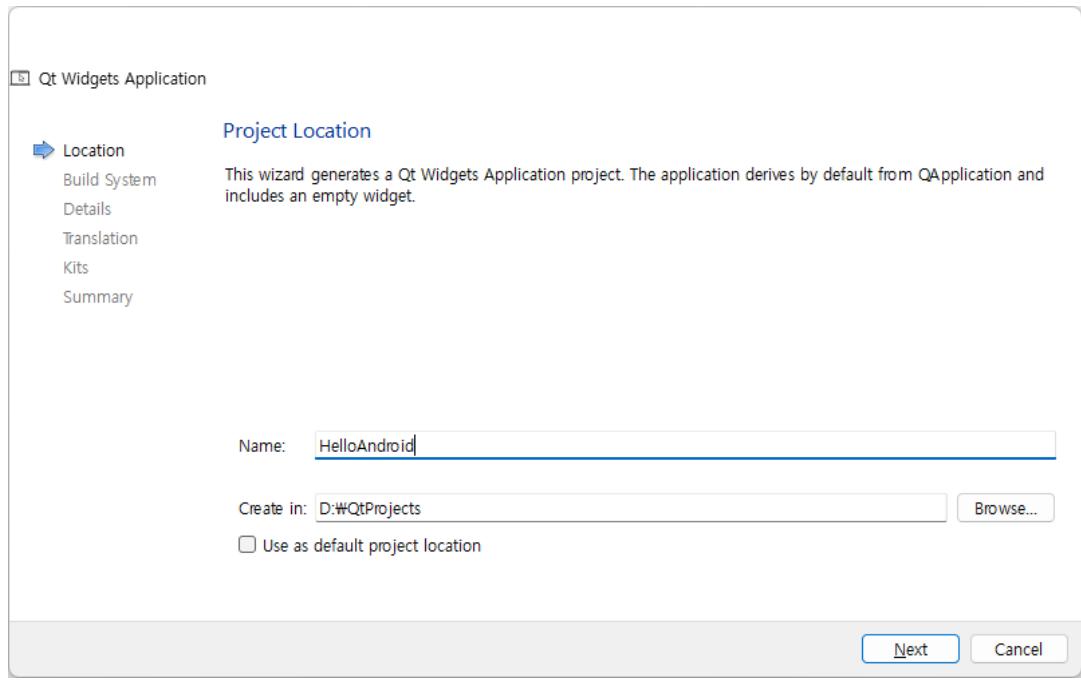
You should then see the text "Ready to use" in the Current state field, as shown in the image below.

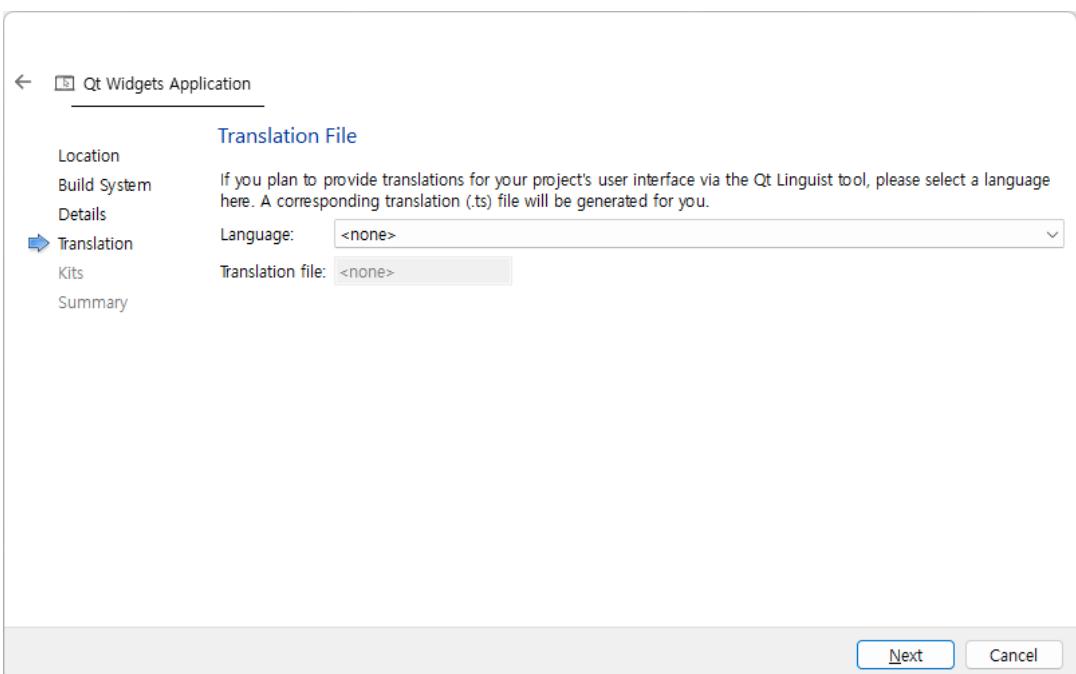
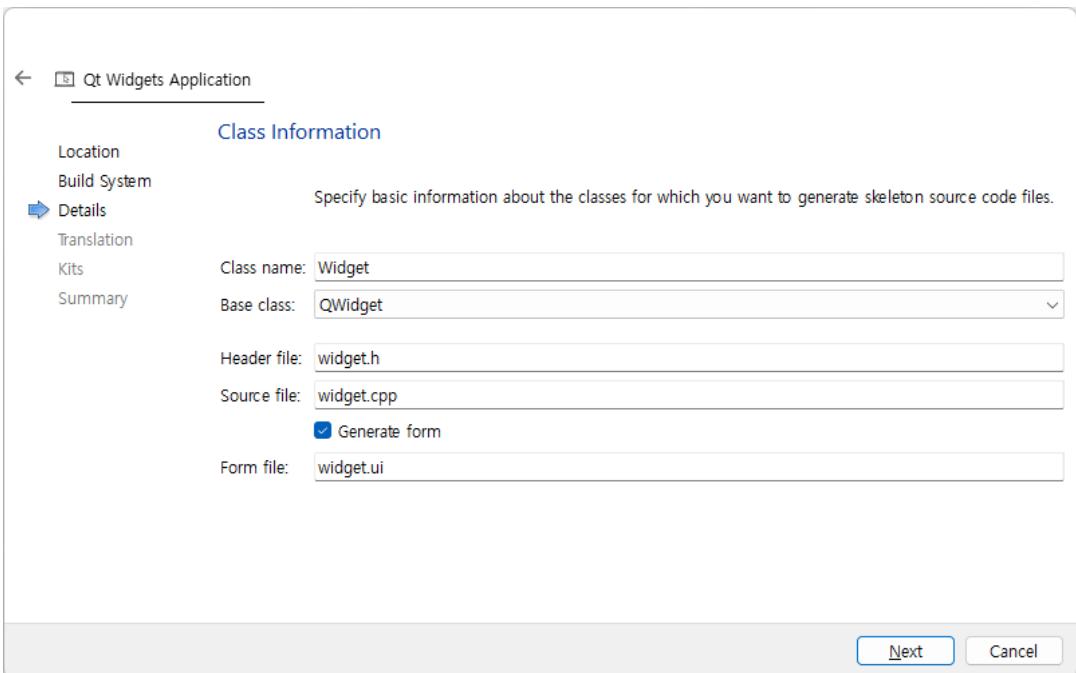


- Building and Deploying Applications Written in Qt on Android Devices

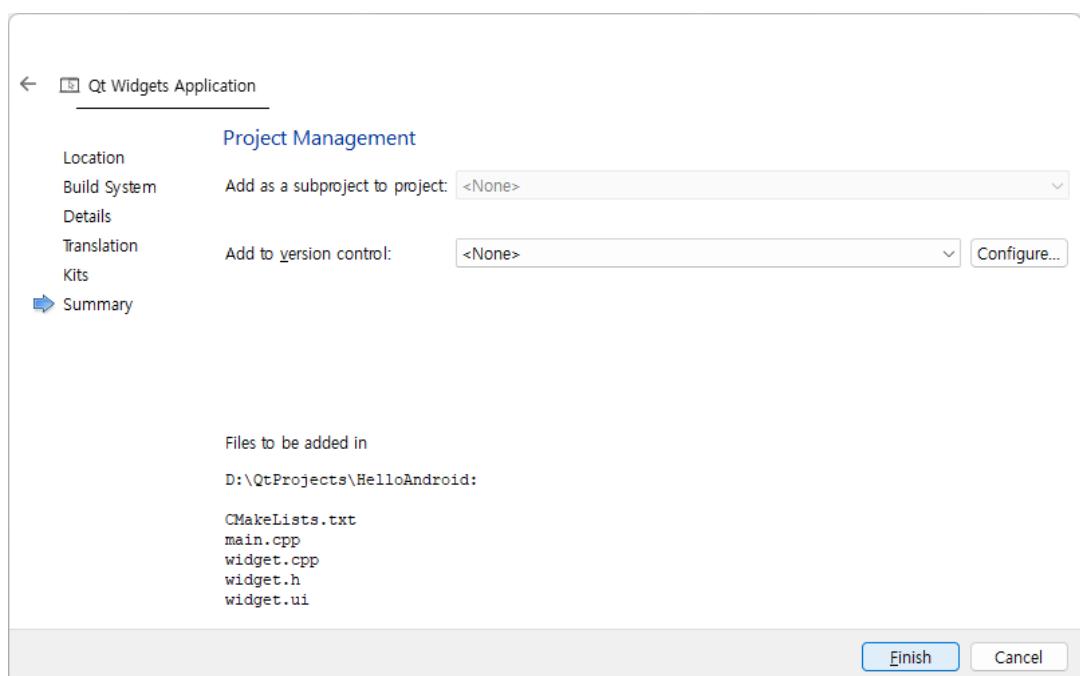
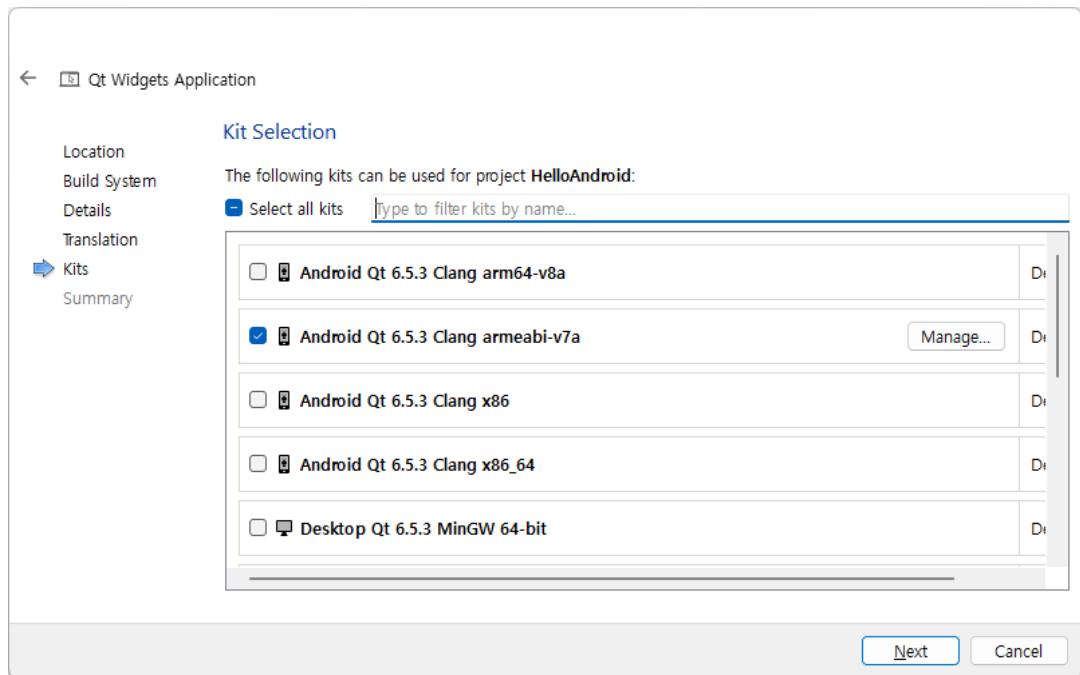
Let's build an example written in Qt, deploy it on an Android Phone, and debug it. Run Qt Creator and create a project as shown in the figure below.





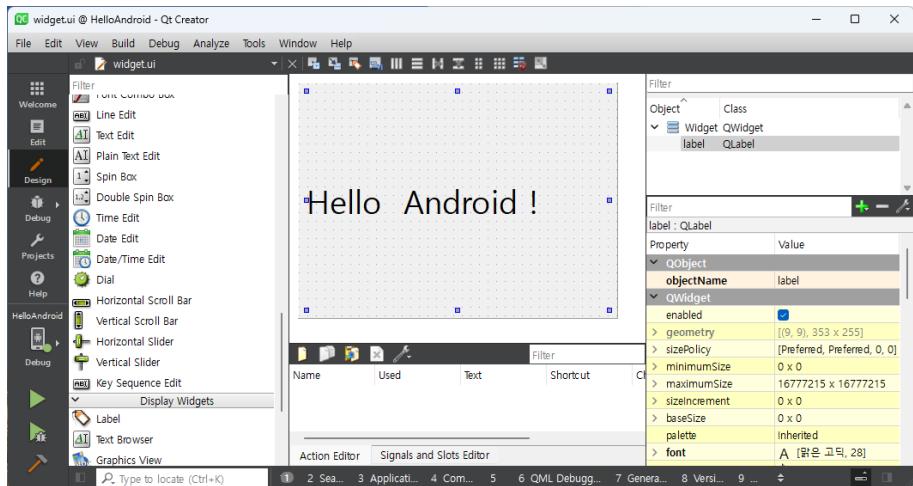


Jesus loves you.

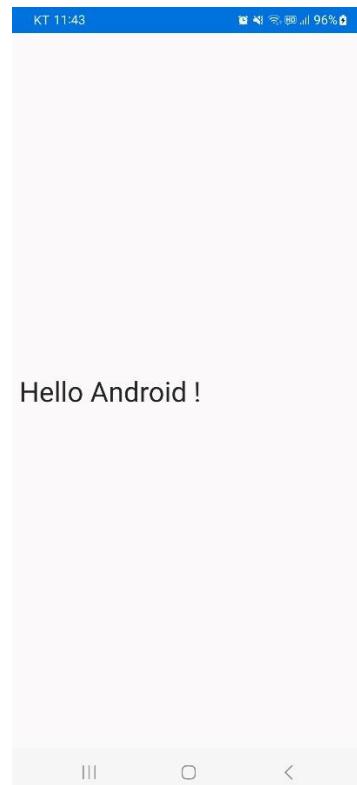


Once the project is created, double-click the `widget.ui` file. Then place a `QLabel` on the GUI. Change the text displayed on the `QLabel` to "Hello Android" and change the font size to 28.

Jesus loves you.



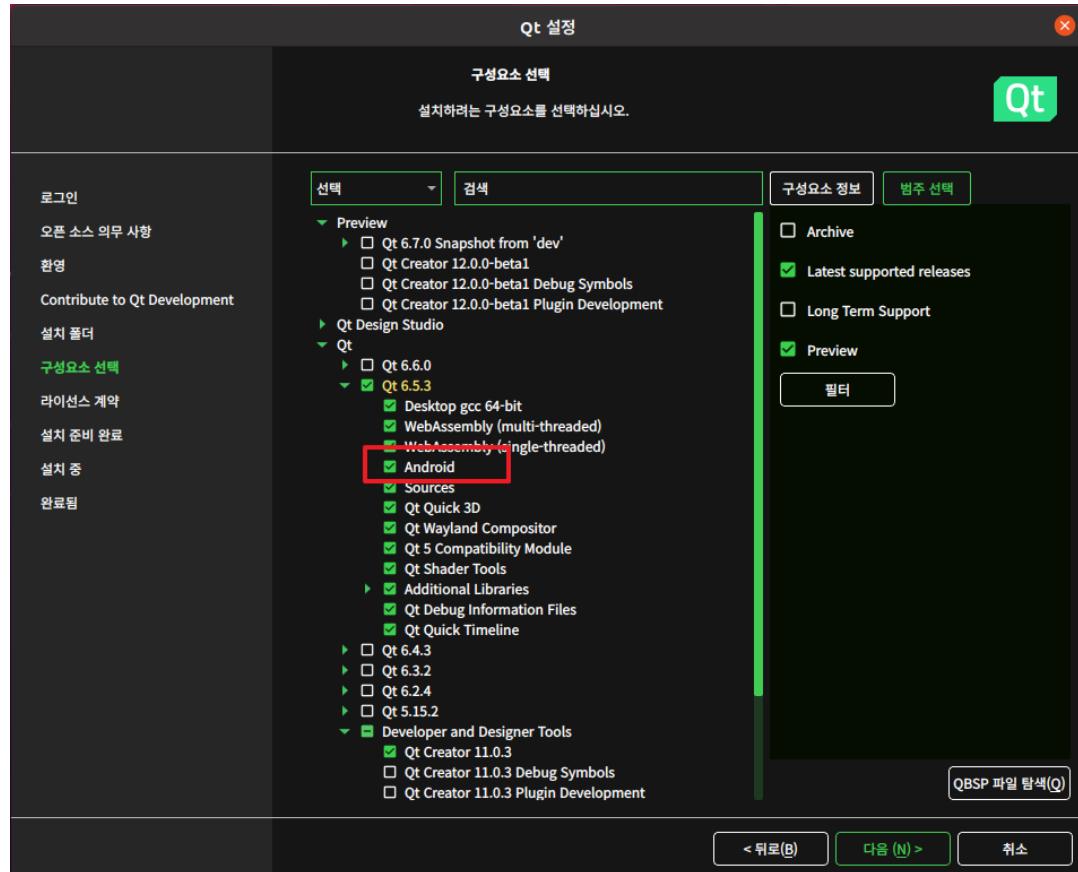
Once you have finished changing the text and font of the QLabel, save it and return to the source code to build and run it. After the build and run is complete, you can see that the app is running on the real Android device.



## 27.2. Android development environment on Linux

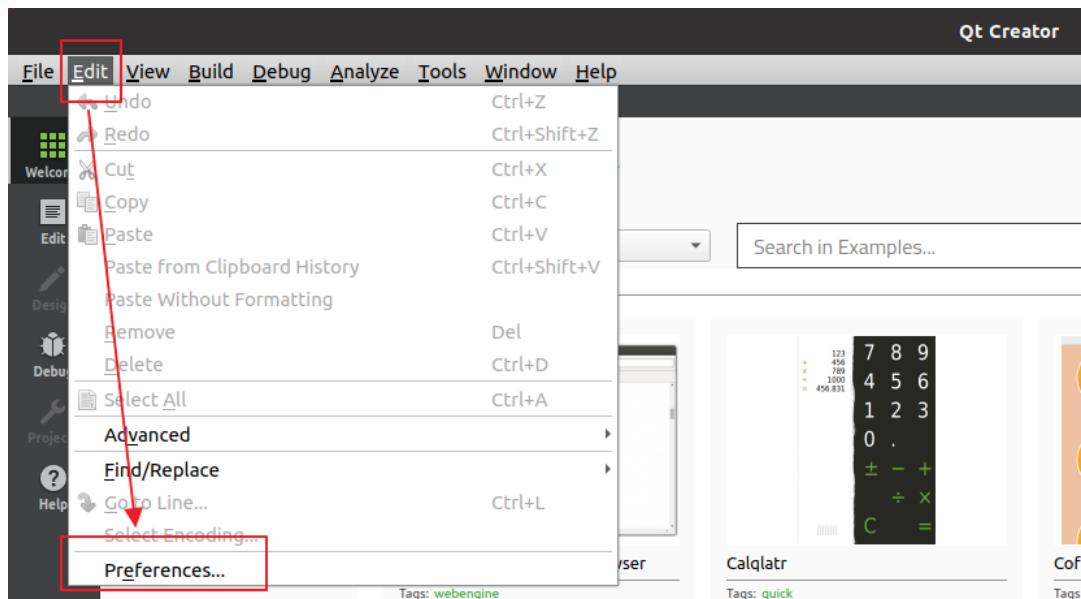
In this chapter, we will build an Android development environment on Ubuntu Linux (20.04) and create a simple app that displays "Hello Android!" in the UI.

When installing Qt on Ubuntu Linux, you will need to add an Android entry.



- Build an Android development environment

Select it and install it as shown above. Next, run Qt Creator when the installation is complete. When Qt Creator is running, select the [Edit] menu from the menu, and then select the [Preferences...] menu as shown below.



In the dialogue below, first install the Java Developer Kit (JDK) in the JDK location field, and then enter the location of the installed JDK.

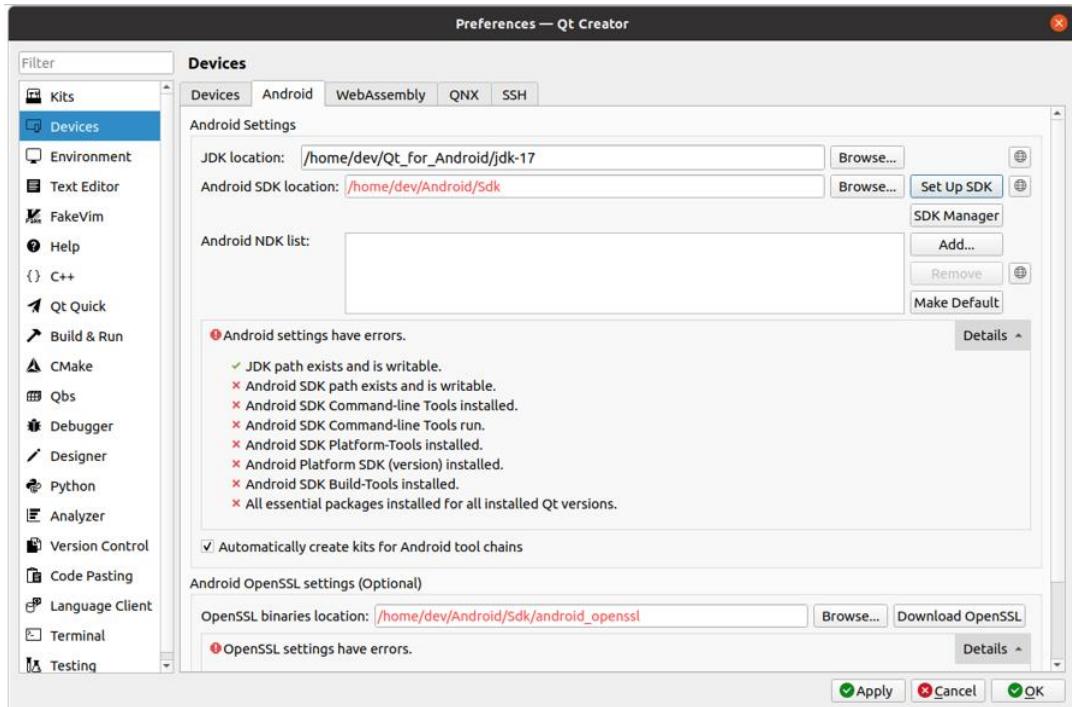
In this case, we'll use the Open JDK. And for Version, we'll use JDK version 17.

A screenshot of a web browser window. The address bar shows 'https://jdk.java.net/java-se-ri/17'. The main content is the 'Java Platform, Standard Edition 17 Reference Implementations' page. On the left, there's a sidebar with links for 'GA Releases' (JDK 21, JavaFX 21, JMC 8), 'Early-Access Releases' (JDK 22, JavaFX 22, jextract, Loom, Valhalla), 'Reference Implementations' (Java SE 21, Java SE 20, Java SE 19, Java SE 18, Java SE 17, Java SE 16), and 'Feedback' (Report a bug, Archive). The main content area has a heading 'Java Platform, Standard Edition 17 Reference Implementations'. It states that the official Reference Implementation for Java SE 17 (JSR 392) is based solely upon open-source code available from the JDK 17 Project in the OpenJDK Community. It mentions that the binaries are available under the GNU General Public License version 2, with the Classpath Exception. A section titled 'These binaries are for reference use only!' provides details about the binaries being provided for reference purposes only. It lists two binary options: 'RI Binary (build 17+35) under the GNU General Public License version 2' (Oracle Linux 8.3 x64 Java Development Kit (sha256) 179 MB and Windows 10 x64 Java Development Kit (sha256) 178 MB). Below this, there's a section for 'RI Source Code' and 'International use restrictions'.

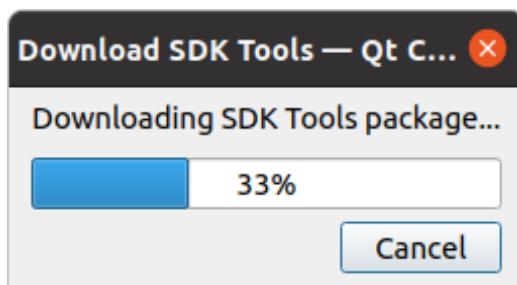
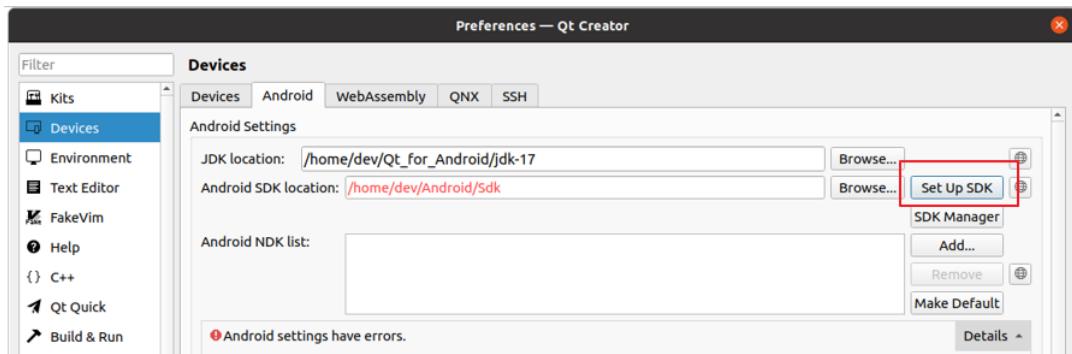
Download the Open JDK and unzip it. Then return to Qt Creator and enter the location

Jesus loves you.

of the unzipped directory of the Open JDK in the JDK Location field, as shown in the figure below.

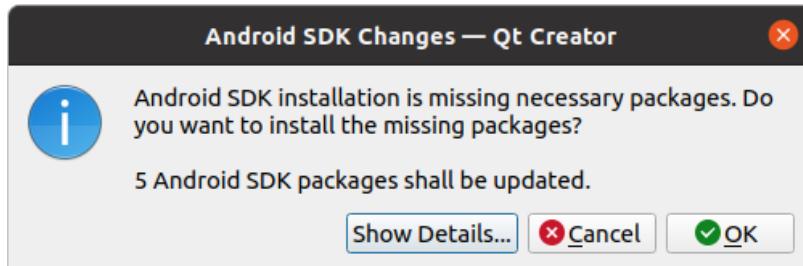


Next, click the [Set Up SDK] button.

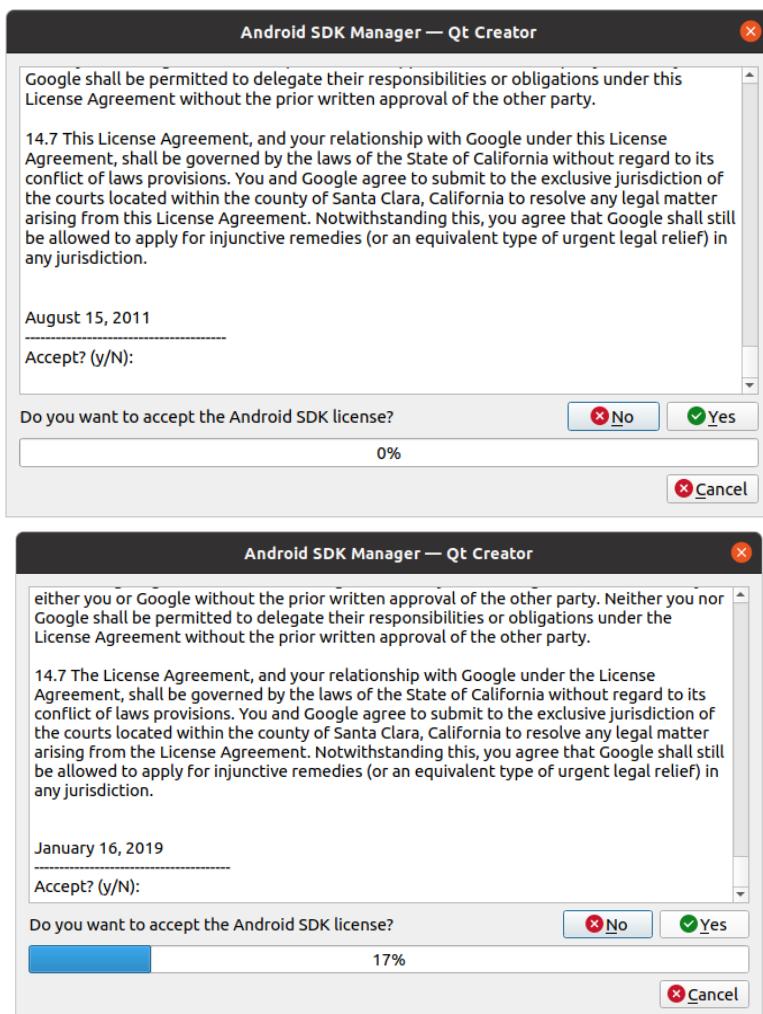


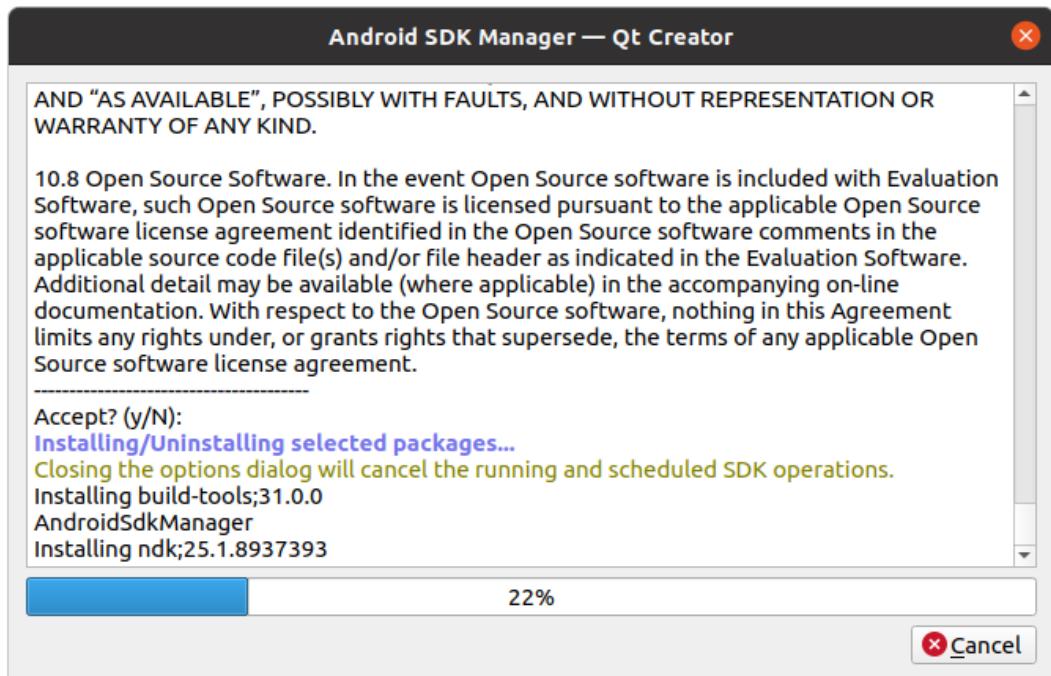
Jesus loves you.

The following is a dialogue to agree to the installation before installing the Android SDK package. Click the [OK] button.

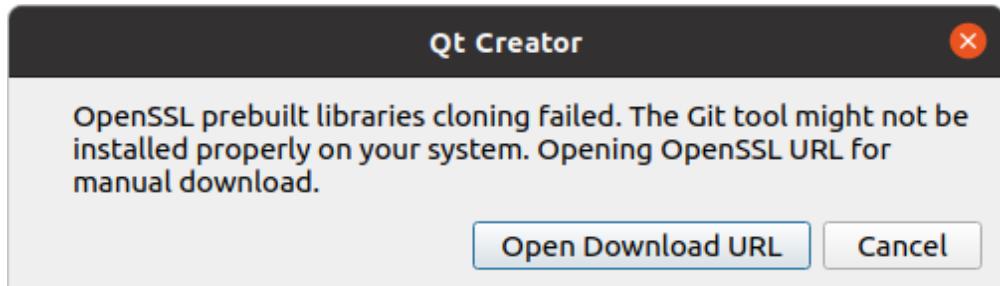
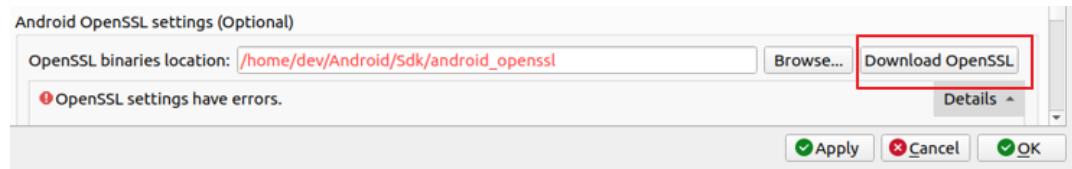


The next dialogue asks for your consent to the licence. It asks you to press the [Yes] button several times. Click [Yes] each time.





Once the installation is complete, at the bottom of the dialogue, in the OpenSSL binaries location field, download and extract OpenSSL. Then specify the location of the extracted directory as shown in the image below.

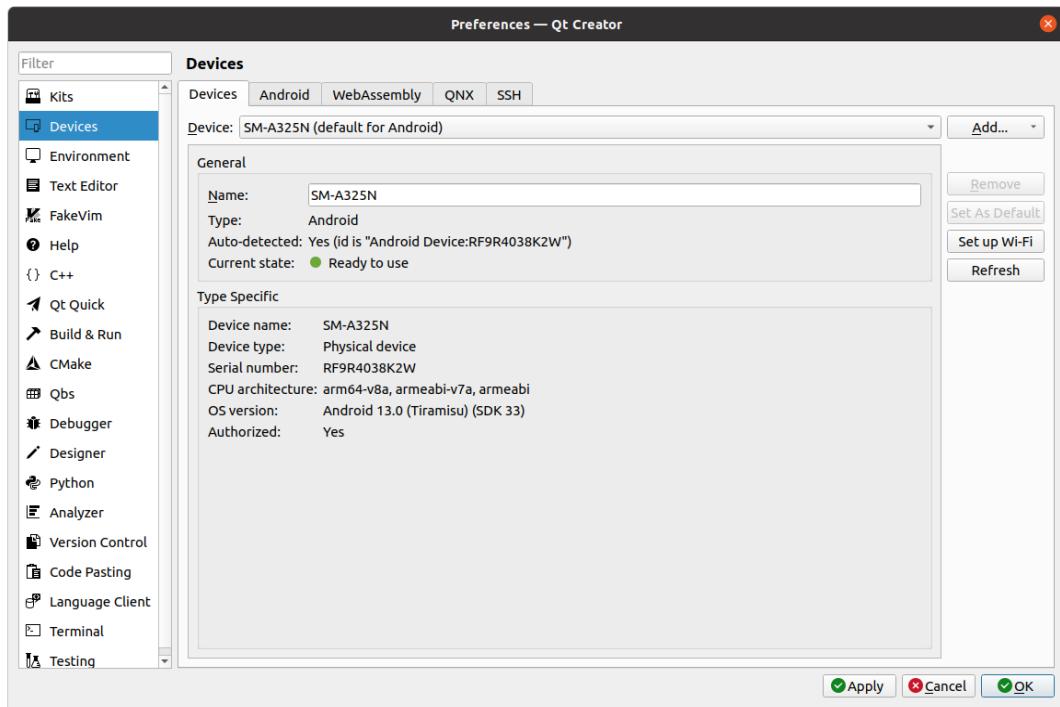


Jesus loves you.

The screenshot shows a GitHub repository page for "KDAK/android\_openssl". The repository has 121 forks and 263 stars. A modal window is open for cloning the repository via Local or Codespaces, showing options for HTTPS, GitHub CLI, and Download ZIP. The repository contains files like no-asasm, ssl\_1.1, ssl\_3, CMakeLists.txt, LICENSE, README.md, android\_openssl.cmake, build\_ssl.sh, and openssl.pri. There are also README.md and .gitignore files.

The screenshot shows the "Devices" tab in the Qt Creator Preferences dialog. The "Android" tab is selected. The "Android Settings" section includes fields for "JDK location" (set to /home/dev/Qt\_for\_Android/jdk-17) and "Android SDK location" (set to /home/dev/Android/Sdk). An "Android NDK list" field is set to /home/dev/Android/Sdk/ndk/25.1.8937393. A message box displays "Android settings are OK. (SDK Version: 11.0, NDK Version: 25.1.8937393)". Other tabs in the preferences dialog include "Devices", "Android", "WebAssembly", "QNX", and "SSH".

Next, select the [Devices] item at the top of the dialogue. In order to run the built app on the actual connected Android device, if you see the message "Ready to use" in the [Current state] item as shown in the figure below, it means that the connected Android device is connected normally.

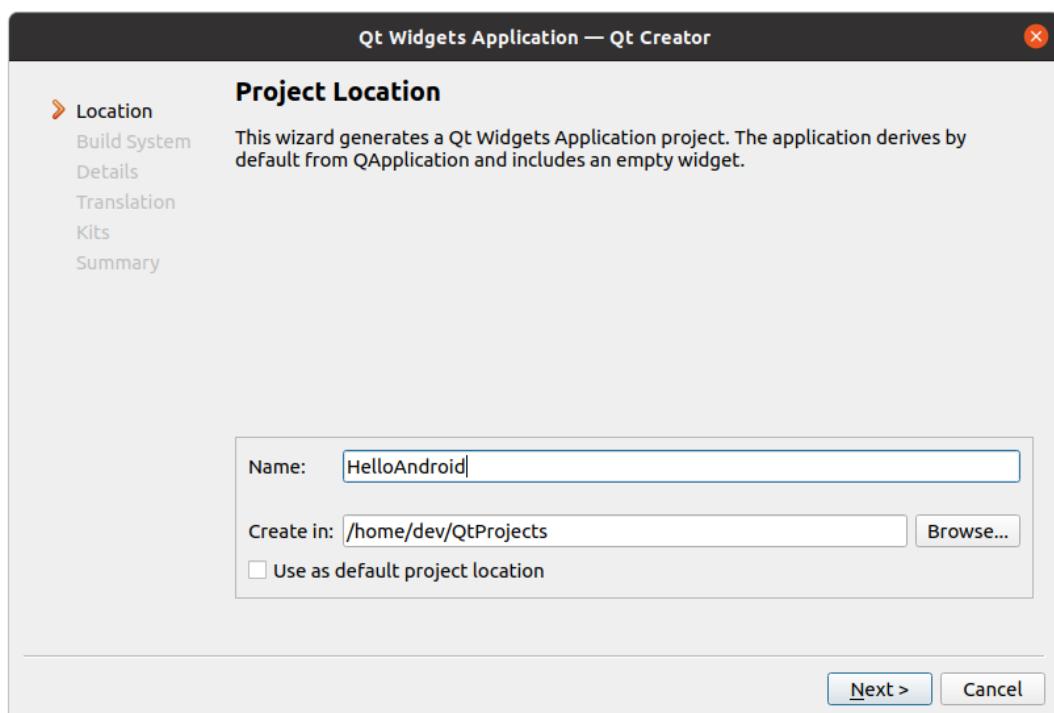
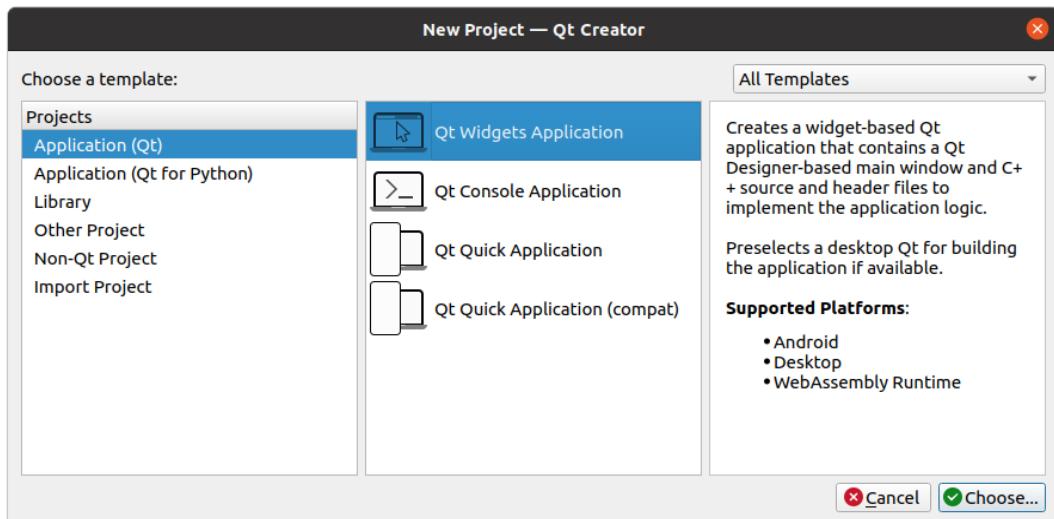


At this point, the development environment is complete. The necessary setup of the Android device was described in the previous chapter, so please refer to the Android Device Setup section in the previous chapter.

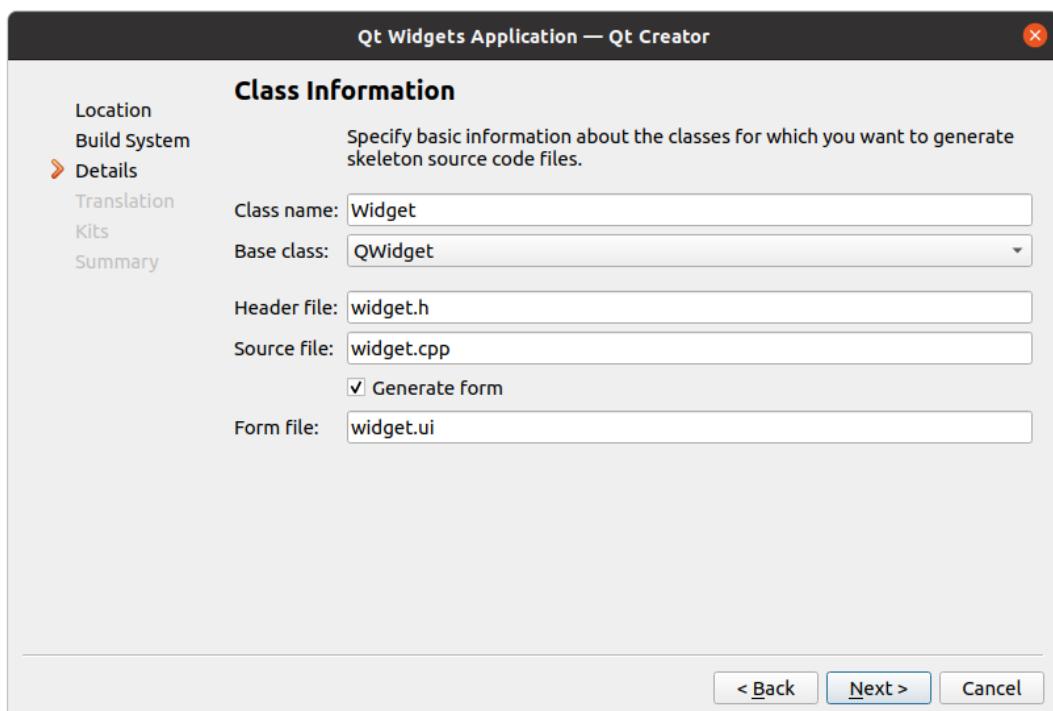
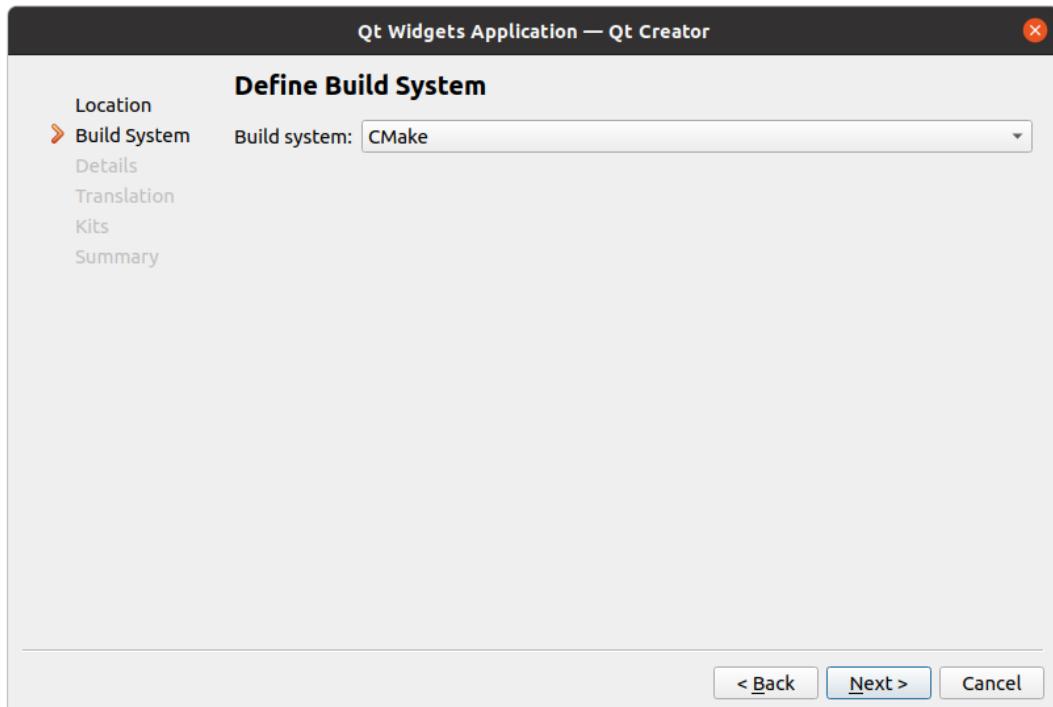
- Building and Deploying Applications Written in Qt on Android Devices

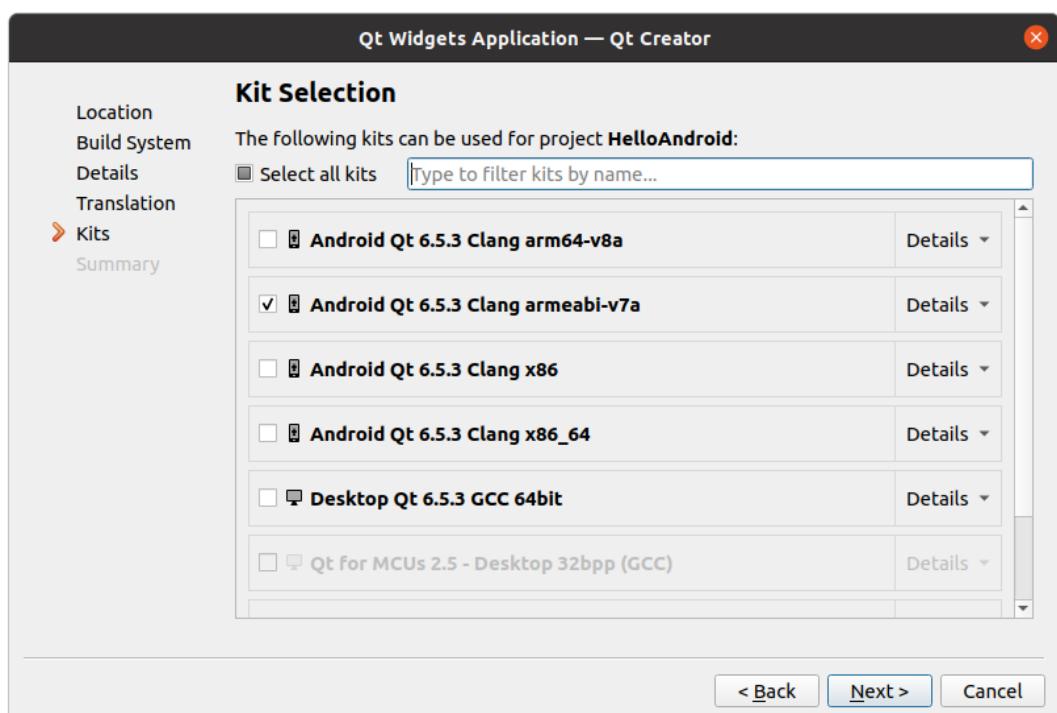
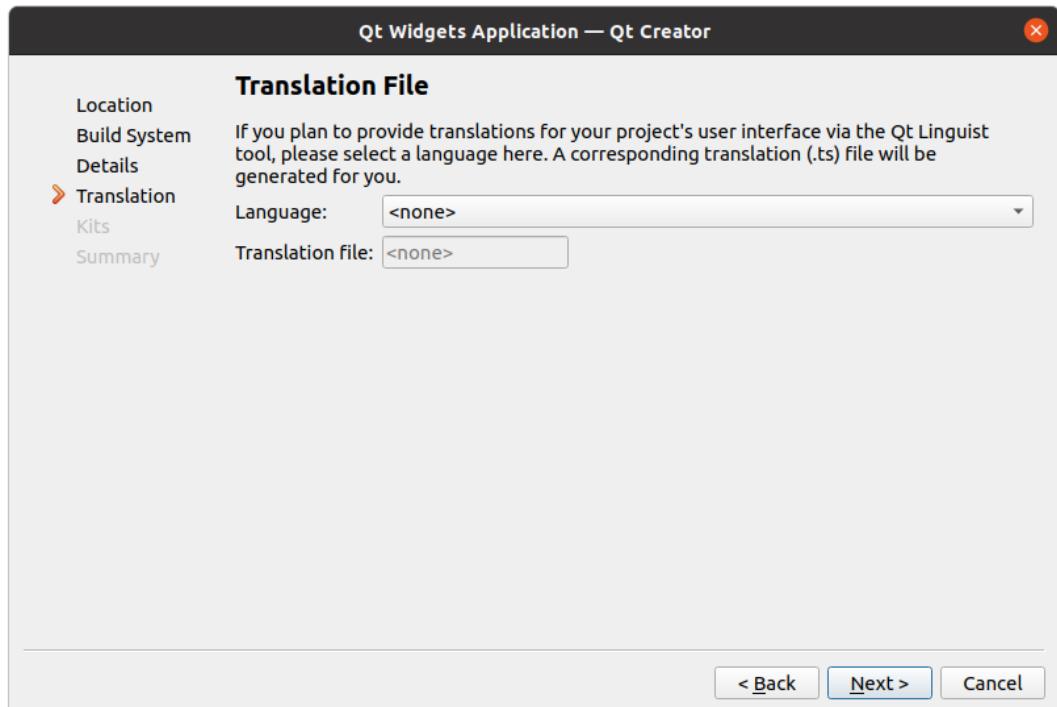
Let's create a simple app and run it on a real Android device. When creating the project, select [Qt Widget Application] and click the [Chooses] button.

Jesus loves you.

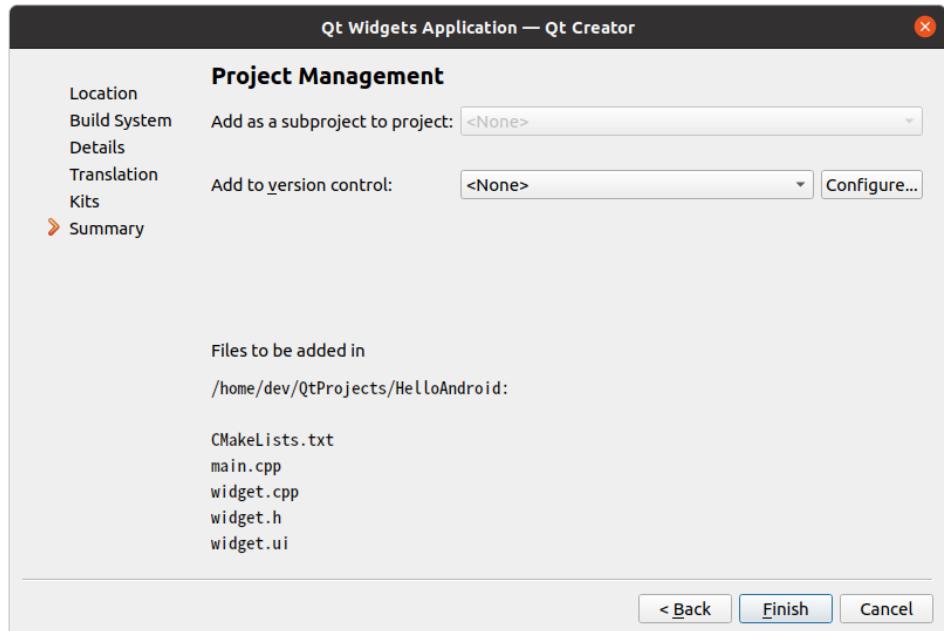


Jesus loves you.

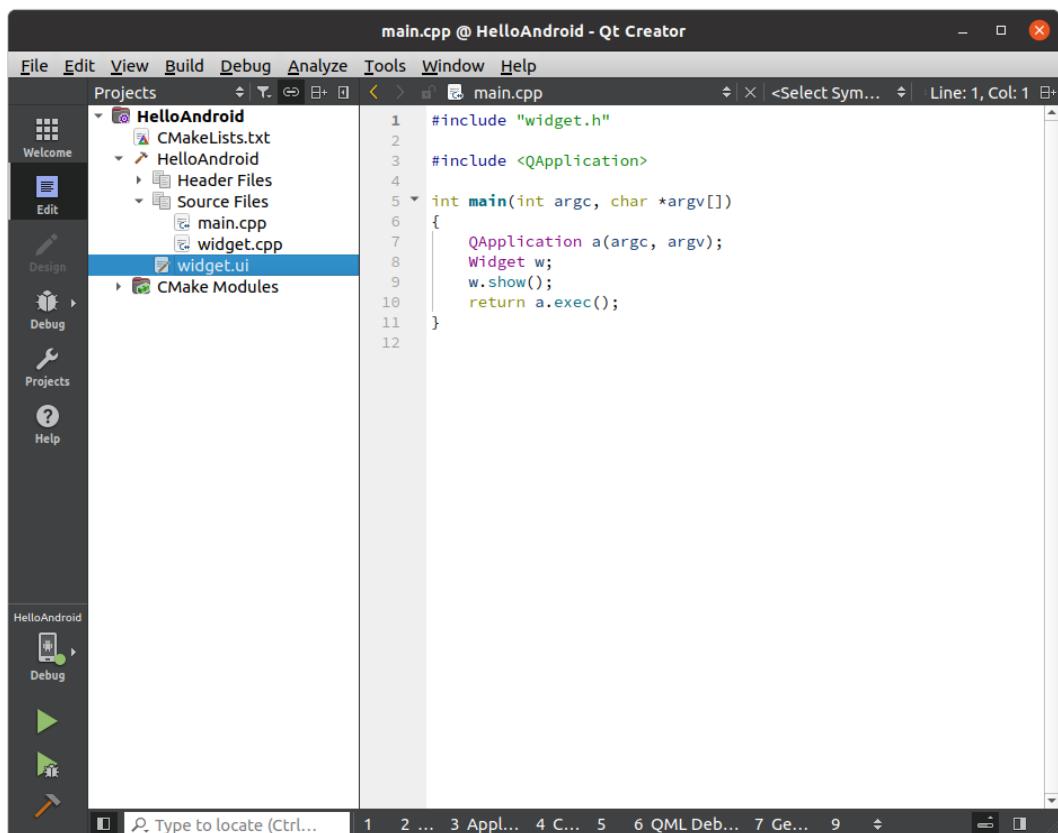




Jesus loves you.

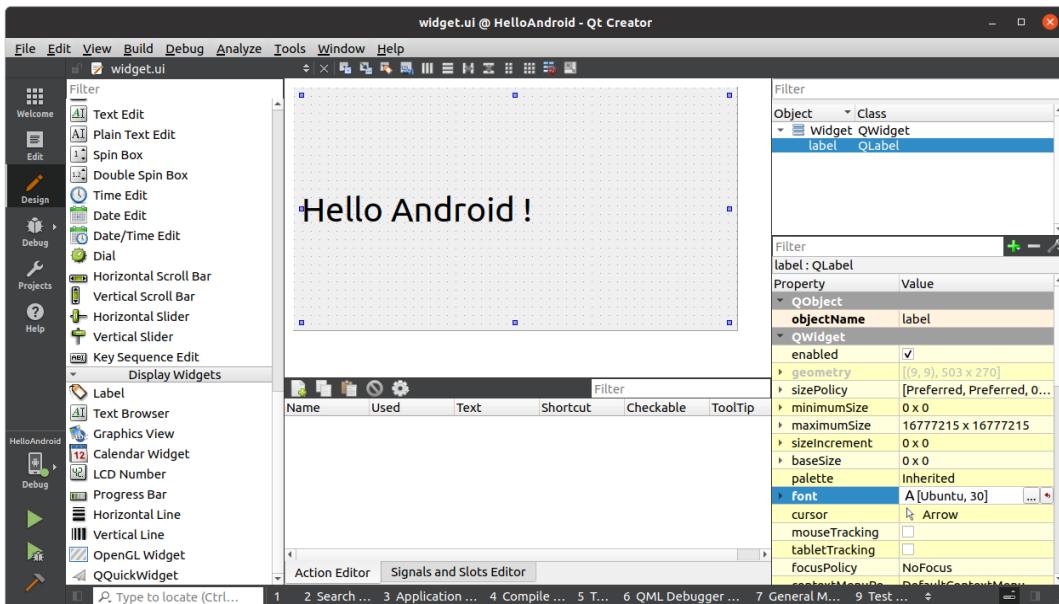


Once the project is created, double-click the widget.ui file, as shown in the image below.

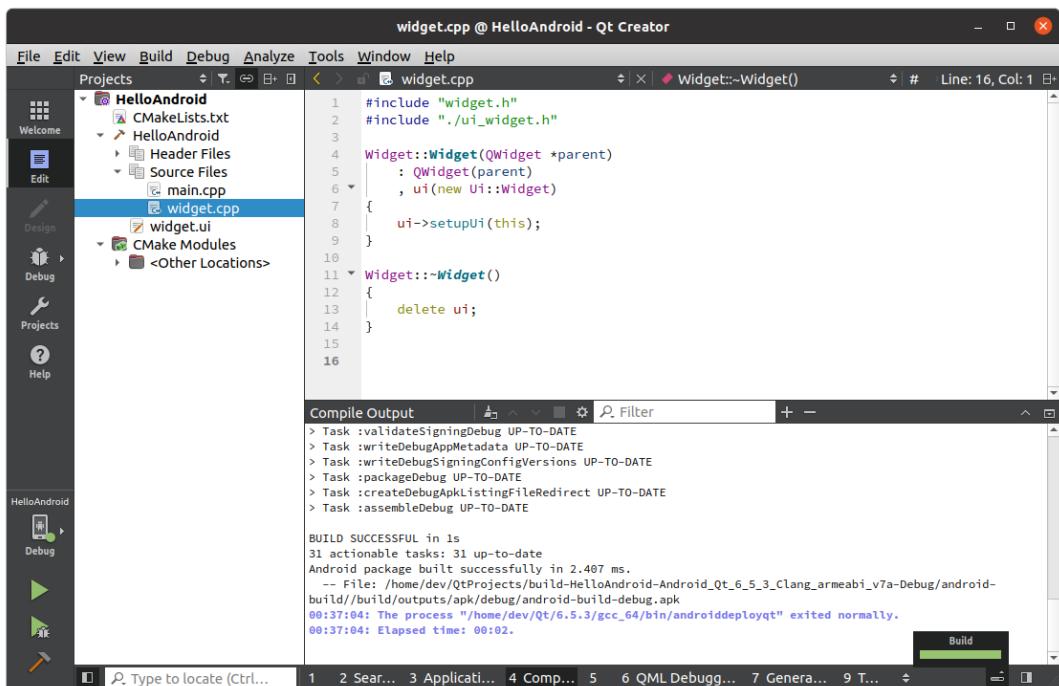


Place a QLabel on the GUI and type "Hello Android!". Then change the font size to 30.

Jesus loves you.

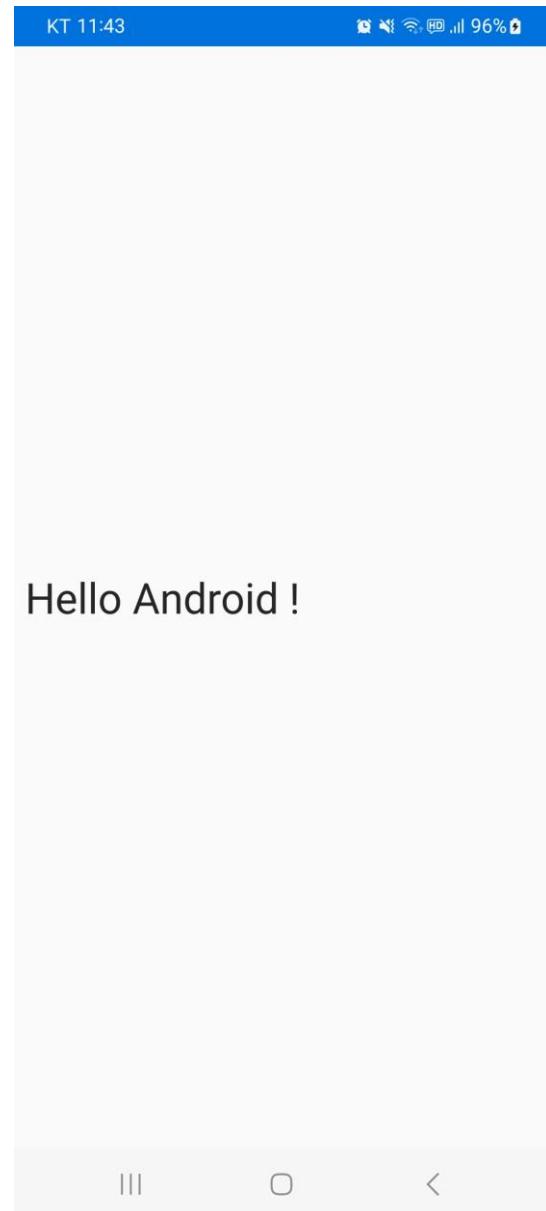


Modify and build as shown in the image above.



Once the build is finished, let's run it. When you do, you should see it running on a real Android device, as shown in the image below.

Jesus loves you.



## 28. Essential Network Programming

Developing applications using the network modules provided in Qt is easier and faster to implement than using the network libraries provided in C++. The main reason for the ease of network programming in Qt is the ability to implement network-based applications based on Signal/Slot.

For example, suppose you are implementing a traditional C++-based chat server. One of the features you need to implement is to handle messages from clients.

You have multiple clients connected to the server, and when a particular client sends a message, you need to send it to all the clients connected to the server. Let's implement this functionality using the standard network library provided by C++. First, we need to have a thread running per client. If 10 clients are connected, 10 threads should be created and running.

However, Qt makes it easy to implement this functionality. For example, if you want to send a message from a specific client, you can associate the signal it sends with a specific Slot function, i.e. instead of a Thread, you only need to associate a Signal and a Slot. The following example source code is an implementation of a Signal and a Slot using the Network module provided by Qt.

```
QHttpSocket::QHttpSocket(QObject *parent) : QObject(parent)
{
    QTcpSocket *socket = new QTcpSocket(this);
    connect(socket, SIGNAL(connected()), this, SLOT(slotConnected()));
    ...
}

void QHttpSocket::slotConnected()
{
    qDebug("[%s] CONNECTED", Q_FUNC_INFO);
    socket->write("HEAD / HTTP/1.0\r\n\r\n\r\n\r\n\r\n");
}
```

The example source code above declares an object of class QTcpSocket. When a new

client connects to the server using the `connect( )` function, it raises the `connected( )` Signal. This signal is used to call the `slotConnected( )` Slot function using the `connect( )` function.

This is easy to implement in Qt because you only need to implement the Slot function, not a Thread implementation to handle new connections.

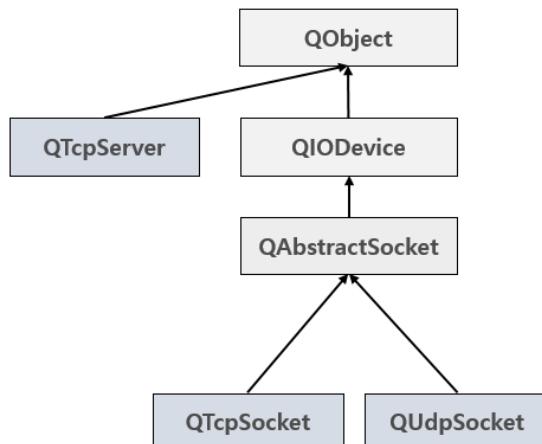
The Qt network module provides a number of APIs for implementing servers or clients based on the TCP/UDP protocol.

The main classes based on the TCP/UDP protocol are `QTcpSocket`, `QTcpServer`, and `QUdpSocket`.

✓ LOW Level Network Classes

- `QTcpSocket`: network class based on the TCP protocol
- `QTcpServer`: a class suitable for implementing servers based on the TCP protocol
- `QUdpSocket`: Network classes based on the UDP protocol

LOW level classes are suitable for developers to implement detailed network functionality.



The `QTcpSocket` and `QUdpSocket` classes are implemented by inheriting from the `QAbstractSocket` class. If you want to implement a new network protocol that is not based on the TCP/UDP protocol, you can save a lot of time by inheriting and implementing the `QAbstractSocket` class.

`QTcpServer` is a class based on the same TCP protocol as `QTcpSocket`. The difference is

Jesus loves you.

that the QTcpServer class provides the functionality needed to implement server-based applications.

In order to use the network module provided by Qt, you need to specify in your project file that you want to use the network module, as follows.

If you're using CMake, you'll need to add the following

```
find_package(Qt6 REQUIRED COMPONENTS Network)
target_link_libraries(mytarget PRIVATE Qt6::Network)
```

If you're using qmake, you'll need to add the following items

```
QT += network
```

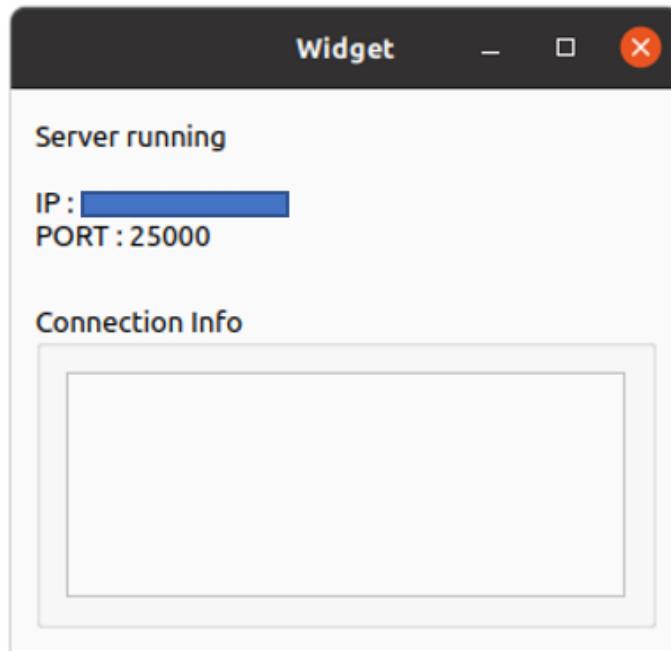
So far, we've gotten an overview of the Network provided by Qt. In the next sub-section, we'll dive into the details of what you need to implement.

## 28.1. TCP Protocol-Based Server/Client Implementation

In this chapter, we'll use the QTcpServer class and the QTcpSocket class to implement the server/client example.

- TCP Server Example Using the QTcpServer Class

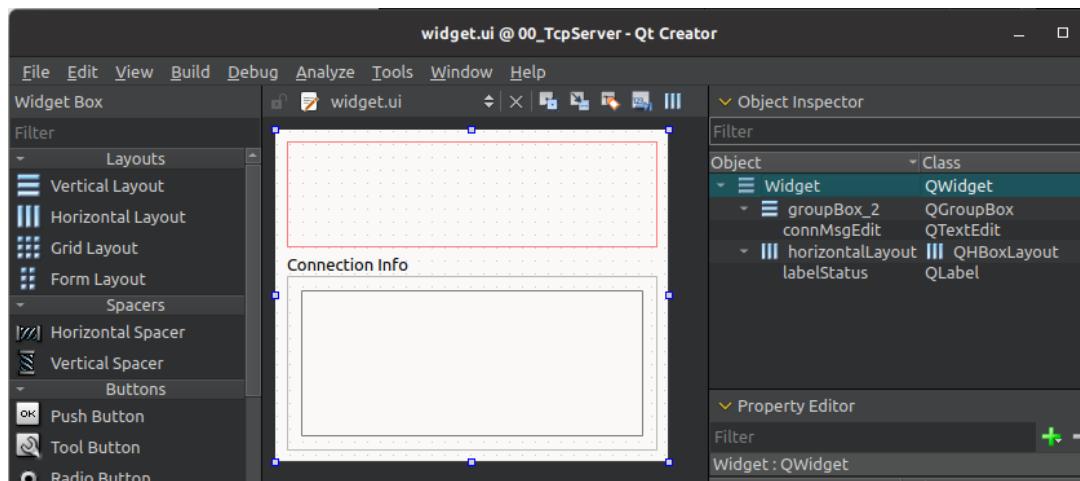
In this example, we'll use the QTcpServer class to implement a server and client application.



Let's place the widget on the GUI as shown in the image above. You can find the source code for this example in the 00\_TcpServer directory.

After creating a CMake-based project, modify and add the following entries to the CMakeLists.txt file.

```
find_package(QT NAMES Qt6 Qt5 REQUIRED COMPONENTS Widgets Network)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets Network)
...
target_link_libraries(00_TcpServer PRIVATE Qt${QT_VERSION_MAJOR}::Network)
```



We will place a QLabel widget at the top and a QTextEdit in the bottom group box that can output the time the client was connected as text.

Next, in the header file of the Widget class, write the following source code

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QObject>
#include <QWidget>

#include <QtNetwork/QTcpServer>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
    QTcpServer *tcpServer;
```

```
void initialize();

private slots:
    void newConnection();

};

#endif // WIDGET_H
```

In the header source code above, the initialize( ) function is for initializing an object of class QTcpServer. The newConnection( ) Slot function is the function that is called when a signal is raised when a client connects. The following is the widget.cpp source code.

```
#include "widget.h"
#include "./ui_widget.h"

#include <QtNetwork/QNetworkInterface>
#include <QtNetwork/QTcpSocket>
#include <QMessageBox>
#include <QTime>

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

void Widget::initialize()
{
    QHostAddress hostAddr;
    QList<QHostAddress> ipList = QNetworkInterface::allAddresses();

    // Use something other than localhost (127.0.0.1)
    for (int i = 0; i < ipList.size(); ++i) {
        if (ipList.at(i) != QHostAddress::LocalHost &&
            ipList.at(i).toIPv4Address()) {
            hostAddr = ipList.at(i);
            break;
        }
    }
}
```

```
if (hostAddr.toString().isEmpty())
    hostAddr = QHostAddress(QHostAddress::LocalHost);

tcpServer = new QTcpServer(this);
if (!tcpServer->listen(hostAddr, 25000)) {
    QMessageBox::critical(this, tr("TCP Server"),
                          tr("Cannot start the server, Error: %1.")
                          .arg(tcpServer->errorString()));
    close();
    return;
}

ui->labelStatus->setText(tr("Server running \n\n"
                             "IP : %1\n"
                             "PORT : %2\n")
                           .arg(hostAddr.toString())
                           .arg(tcpServer->serverPort()));

connect(tcpServer, SIGNAL(newConnection()), this, SLOT(newConnection()));

ui->connMsgEdit->clear();
}

void Widget::newConnection()
{
    QTcpSocket *clientConnection = tcpServer->nextPendingConnection();
    connect(clientConnection, SIGNAL(disconnected()),
            clientConnection, SLOT(deleteLater()));

    QString currTime = QTime::currentTime().toString("hh:mm:ss");
    QString text = QString("Client Connection (%1)").arg(currTime);

    ui->connMsgEdit->append(text);
    QByteArray message = QByteArray("Hello Client ~ ");
    clientConnection->write(message);
    clientConnection->disconnectFromHost();
}

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

The initialize( ) function called in the constructor function obtains the IP of the server to which the client is connecting from the system, initializes a tcpServer object of class QTcpServer, and uses the listen( ) member function to make it ready to listen for client connection requests. The first argument of the listen( ) function is the IP address and the second argument is the server port number.

The connect( ) function at the bottom of the initialize( ) function connects the Signal and Slot functions, which are called when a client connects. Therefore, when a new client connects, the newConnection( ) function is called.

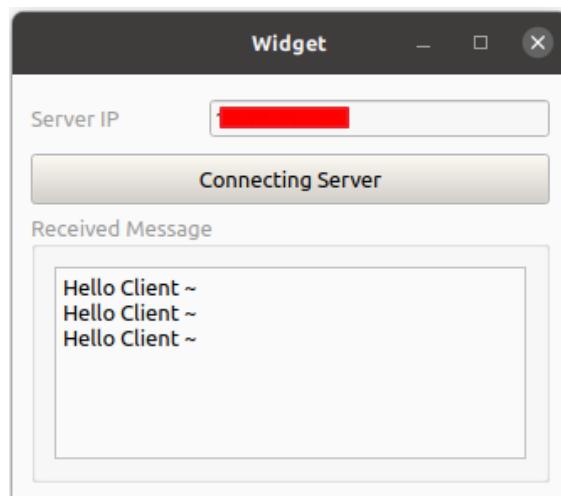
The newConnection( ) function displays the time the client connected in the QTextEdit widget on the GUI and sends the message "Hello Client ~ " to the connected client.

- TCP client example using the QTcpSocket class

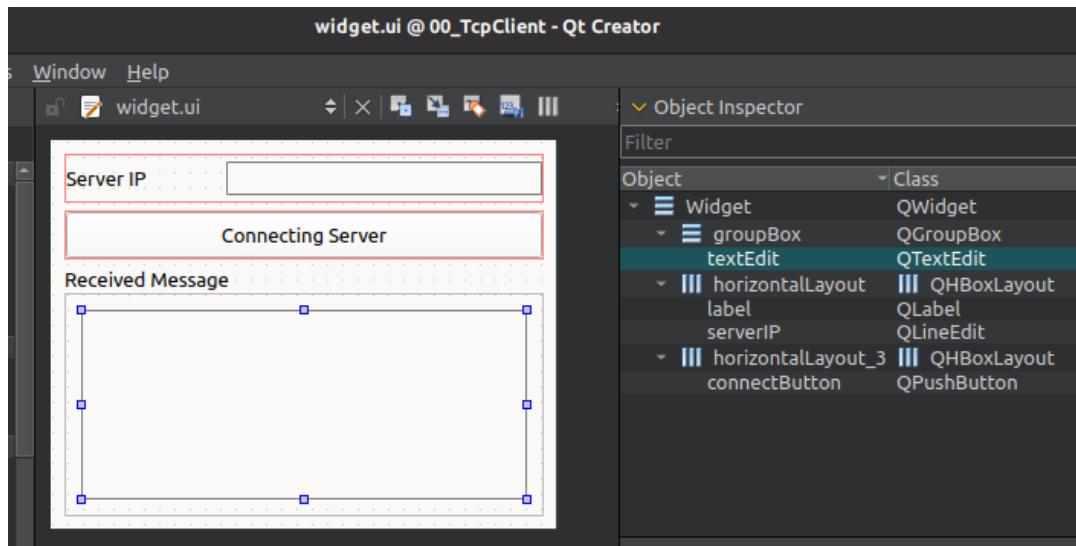
After creating a CMake-based project, modify and add entries to the CMakeLists.txt file as shown below.

```
find_package(QT NAMES Qt6 Qt5 REQUIRED COMPONENTS Widgets Network)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets Network)
...
target_link_libraries(00_TcpServer PRIVATE Qt${QT_VERSION_MAJOR}::Network)
```

Here is an example client Create a project that inherits from QWidget and place the widget on the GUI as shown in the following figure. For the complete example source code for the client, see the 00\_TcpClient directory.



Place the QLineEdit widget so that you can enter the IP and PORT of the server you want to connect to, as shown in the image above. Then place the [Connect] button and the QTextEdit at the bottom.



[Connecting Server] 버튼을 클릭하면 서버와 연결을 시도한다. 서버와 연결이 완료되면 서버로부터 받은 메시지를 QTextEdit 위젯에 출력한다. 아래 예제 소스코드와 같이 widget.h 헤더 소스코드를 작성한다.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QtNetwork/QTcpSocket>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
```

```

Ui::Widget *ui;
QTcpSocket *tcpSocket;

void initialize();

private slots:
    void connectButton();
    void readMessage();// Called when receiving a message from the server
    void disconnected();
};

#endif // WIDGET_H

```

connectButton( ) Slot 함수는 [접속] 버튼 클릭 시 호출된다. readMessage( ) Slot 함수는 서버로부터 메시지를 받는 Signal 이 발생하면 호출되는 함수이다.

그리고 disconnected( ) Slot 함수는 서버로부터 접속이 종료된 Signal 이 발생하면 호출되는 함수이다. 다음은 widget.cpp 예제소스코드이다.

```

#include "widget.h"
#include "./ui_widget.h"

Widget::Widget(QWidget *parent)
    : QWidget(parent)
    , ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->connectButton, SIGNAL(clicked()),
            this,           SLOT(connectButton()));

    initialize();
}

void Widget::initialize()
{
    tcpSocket = new QTcpSocket(this);

    connect(tcpSocket, SIGNAL(readyRead()),
            this,           SLOT(readMessage()));
    connect(tcpSocket, SIGNAL(disconnected()),
            this,           SLOT(disconnected()));
}

```

```
void Widget::connectButton()
{
    QString serverip = ui->serverIP->text().trimmed();

    QHostAddress serverAddress(serverip);
    tcpSocket->connectToHost(serverAddress, 25000);
}

void Widget::readMessage()
{
    if(tcpSocket->bytesAvailable() >= 0)
    {
        QByteArray readData = tcpSocket->readAll();
        ui->textEdit->append(readData);
    }
}

void Widget::disconnected()
{
    qDebug() << Q_FUNC_INFO << "Server Connection Close.";
}

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

The initialize( ) function declares a tcpSocket object of class QTcpSocket and associates the signal to receive a message from the server with the readMessage( ) Slot function. It also connects to the disconnected( ) Slot function when it receives a signal to end the connection with the server. Therefore, the readMessage( ) function is called when a message is received from the server, and the disconnect( ) Slot function is called when the connection with the server is terminated.

In the readMessage( ) Slot function, the tcpSocket->bytesAvailable( ) function can get the number of bytes in the message sent by the server. And the readAll( ) member function provides the ability to read the messages sent by the server.

## 28.2. Synchronous and asynchronous implementation

There are two main ways data is sent/received on a network. The first is the synchronous method and the second is the asynchronous method.

The synchronous method is when the server makes a request to the client and waits for the response without doing any other processing. This is called synchronous.

```
...
int writeBytes = socket->write("Hello server\r\n\r\n");
socket->waitForReadyRead(3000);
...
```

As shown in the source code above, you can send a message using the `write()` function and use the `waitForReadyRead()` function to wait for the message to be received from the other side.

Asynchronous is a way to handle situations where the server makes a request to the client but doesn't know when it will receive a response.

```
QHttpSocket::QHttpSocket(QObject *parent) : QObject(parent)
{
    socket = new QTcpSocket(this);
    ...
    connect(socket, SIGNAL(readyRead()), this, SLOT(slotRead()));
}

...

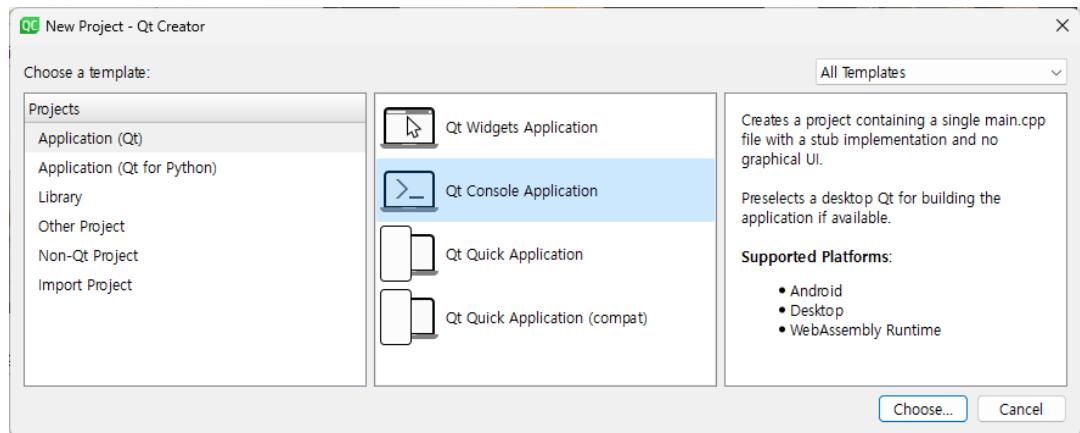
void QHttpSocket::slotRead ()
{
    qDebug() << socket->readAll();
}
```

As shown in the example above, the `slotRead()` Slot function is called when a message is received from the other party. This is called the asynchronous method.

In this chapter, we will see how to use the [qt-dev.com](#) web server to send and receive data using both synchronous and asynchronous methods.

- synchronous method example

This example creates a console-based project without using the GUI.



Once created, add the Network module to the CMakeList.txt file as shown below.

```
cmake_minimum_required(VERSION 3.14)

project(00_Sync LANGUAGES CXX)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Qt NAMES Qt6 Qt5 REQUIRED COMPONENTS Core Network)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Core Network)

add_executable(00_Sync
    main.cpp
    qhttpssocket.h qhttpssocket.cpp
)

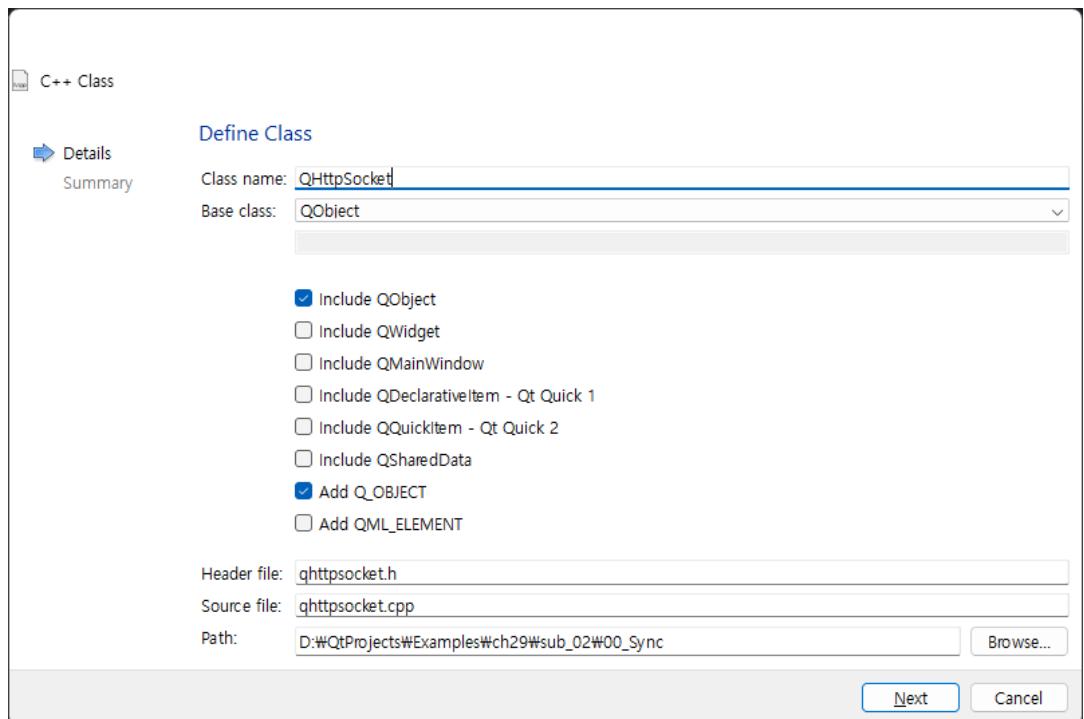
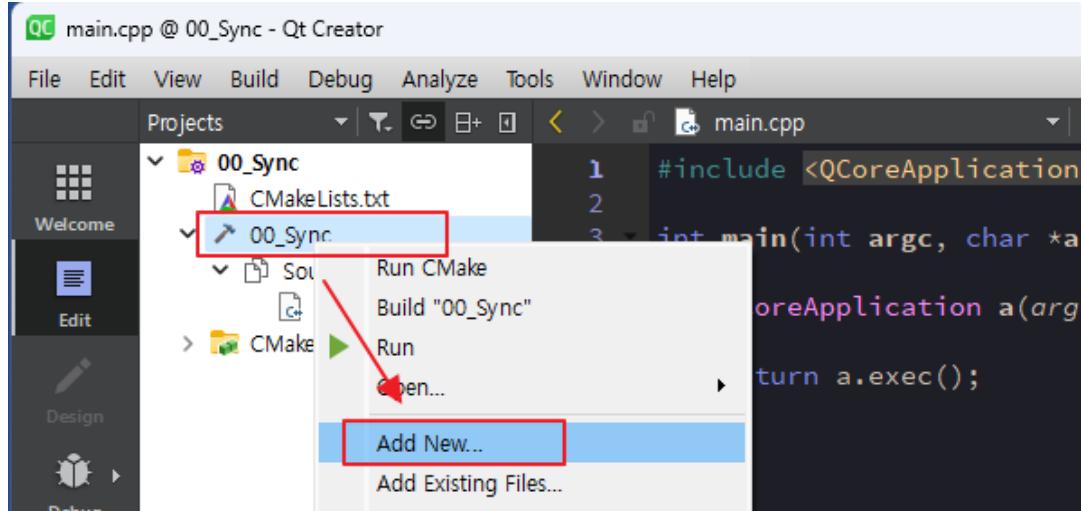
target_link_libraries(00_Sync Qt${QT_VERSION_MAJOR}::Core)
target_link_libraries(00_Sync Qt${QT_VERSION_MAJOR}::Network)

include(GNUInstallDirs)
install(TARGETS 00_Sync
```

Jesus loves you.

```
LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

Next, create a class named QHttpSocket that inherits from the QObject class.



In the header of the class, write something like this

```
#ifndef QHTTPSOCKET_H
```

```
#define QHTTPSOCKET_H

#include <QObject>
#include <QTcpSocket>

class QHttpSocket : public QObject
{
    Q_OBJECT
public:
    explicit QHttpSocket(QObject *parent = nullptr);
    ~QHttpSocket();
    void httpConnect();

signals:

public slots:
    void httpDisconnected();

private:
    QTcpSocket *socket;

};

#endif // QHTTPSOCKET_H
```

Write the qhttpsocket.cpp source code as follows

```
#include "qhttpsocket.h"

QHttpSocket::QHttpSocket(QObject *parent)
    : QObject{parent}
{
    socket = new QTcpSocket(this);
    connect(socket, SIGNAL(disconnected()),
            this, SLOT(httpDisconnected()));
}

QHttpSocket::~QHttpSocket()
{

}

void QHttpSocket::httpConnect()
{
```

```
socket->connectToHost("qt-dev.com", 80);

if(socket->waitForConnected(5000))
{
    qDebug() << "TCP Connected.";
    // send
    int writeBytes = socket->write("Hello server\r\n\r\n");
    socket->waitForBytesWritten(1000);

    qDebug() << "write bytes : " << writeBytes;

    socket->waitForReadyRead(3000);

    qDebug() << "Reading: " << socket->bytesAvailable();
    qDebug() << socket->readAll();
}
else
{
    qDebug() << "Not connected!";
}

void QHttpSocket::httpDisconnected()
{
    qDebug() << Q_FUNC_INFO;
}
```

In the httpConnect( ) function, the connectToHost( ) function provided by the QTcpSocket class connects to the URL specified in the first argument. The second argument specifies the port number to connect to.

The waitForConnected( ) function, used in the line following the connectToHost( ) function, waits for a connection to be made with the web server. The first argument is the time to wait for the server to respond.

Next, we can use the waitForReadyRead( ) function to wait until we receive data from the other side. In this way, you can implement your source code in a synchronous manner.

In main.cpp, let's create an object of class QHttpSocket and call the httpConnect( ) member function, as shown below.

```
#include <QCoreApplication>
#include "qhttpsocket.h"
```

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

    QHttpSocket *httpSocket = new QHttpSocket();
    httpSocket->httpConnect();

    return a.exec();
}
```

You can find the source code for this example in the 00\_Sync directory.

- Asynchronous method example

c This is the source code of an example asynchronous method using Signal/Slot. Create a Console-based project and add the QHttpSocket class as shown in the previous example. Then, create the qhttpsocket.h header file as shown below.

```
#ifndef QHTTPSOCKET_H
#define QHTTPSOCKET_H

#include <QObject>
#include <QTcpSocket>

class QHttpSocket : public QObject
{
    Q_OBJECT
public:
    explicit QHttpSocket(QObject *parent = nullptr);
    ~QHttpSocket();

    void httpConnect();

signals:

public slots:
    void slotConnected();
    void slotDisconnected();
    void slotBytesWritten(qint64 bytes);
    void slotReadPendingDatagram();
```

```
private:  
    QTcpSocket *socket;  
  
};  
  
#endif // QHTTPSOCKET_H
```

The following example source code is the implementation source code for the QHttpSocket class.

```
#include "qhttpsocket.h"  
  
QHttpSocket::QHttpSocket(QObject *parent)  
    : QObject{parent}  
{  
    socket = new QTcpSocket(this);  
  
    connect(socket, SIGNAL(connected()),  
            this, SLOT(slotConnected()));  
    connect(socket, SIGNAL(disconnected()),  
            this, SLOT(slotDisconnected()));  
    connect(socket, SIGNAL(bytesWritten(qint64)),  
            this, SLOT(slotBytesWritten(qint64)));  
    connect(socket, SIGNAL(readyRead()),  
            this, SLOT(slotReadPendingDatagram()));  
}  
  
QHttpSocket::~QHttpSocket()  
{  
}  
  
void QHttpSocket::httpConnect()  
{  
    socket->connectToHost("qt-dev.com", 80);  
    if(!socket->waitForConnected(3000))  
    {  
        qDebug() << "Socket Error : "  
             << socket->errorString();  
    }  
}  
  
void QHttpSocket::slotConnected()  
{
```

```
qDebug("\n [%s] CONNECTED", Q_FUNC_INFO);
socket->write("HEAD / HTTP/1.0\r\n\r\n\r\n\r\n\r\n");
}

void QHttpSocket::slotDisconnected()
{
    qDebug("\n [%s] DISCONNECTED", Q_FUNC_INFO);
}

void QHttpSocket::slotBytesWritten(qint64 bytes)
{
    qDebug("\n [%s] Bytes Written [size :%d]",
          Q_FUNC_INFO, bytes);
}

void QHttpSocket::slotReadPendingDatagram()
{
    qDebug() << socket->readAll();
}
```

In the constructor, we declare a QTcpSocket class object and associate a Signal with the Slot function when we connect to the server, exit, finish sending a message, and when we receive a message from the web server.

When we run the httpConnect( ) function, we attempt to connect to the web server. Unlike the previous synchronous method, this example uses Signal and Slot to use an asynchronous method. You can find the source code for this example in the 01\_Async directory.

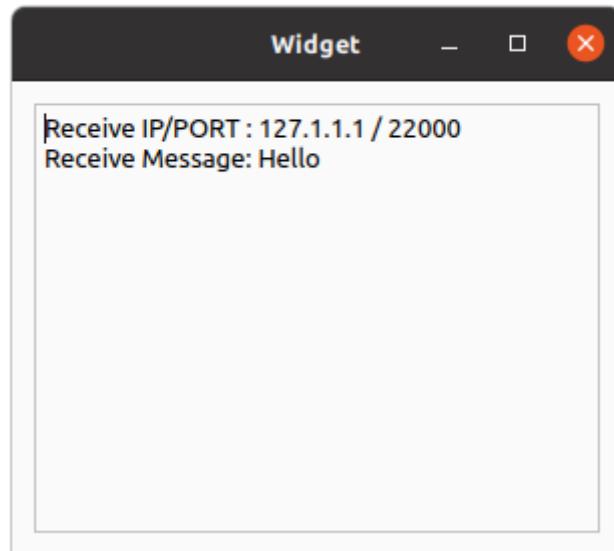
### 28.3. Implement communication based on the UDP

The UDP protocol is unreliable, meaning that the server and client must establish a connection before messages can be sent/received, like TCP, but UDP can send/receive messages bilaterally without establishing a connection.

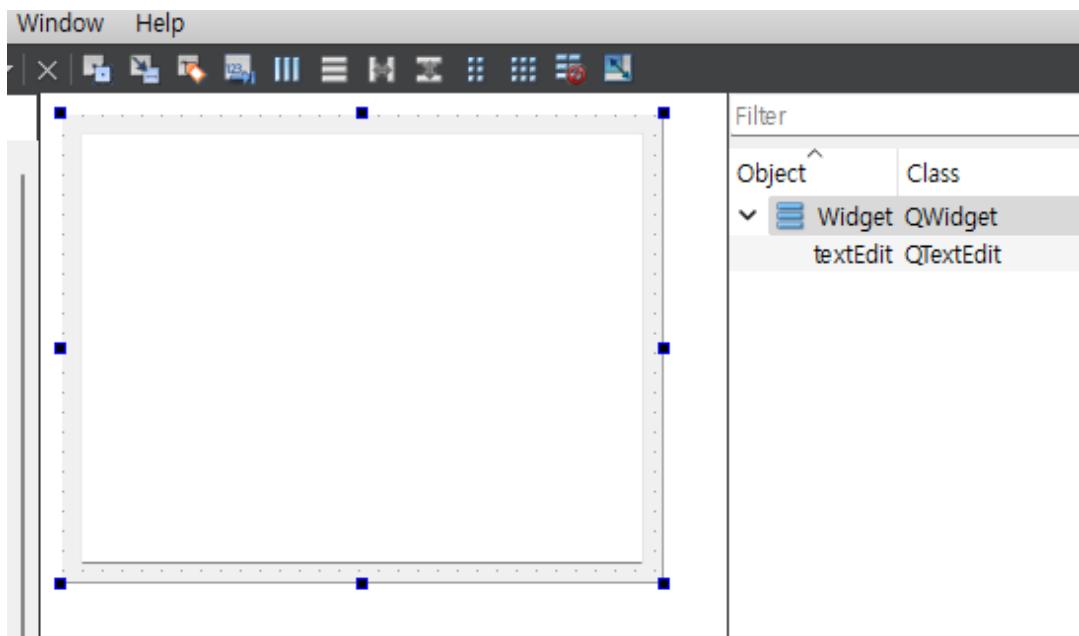
Qt provides the `QUdpSocket` class to implement applications based on the UDP protocol. Therefore, in this chapter, we will implement a server/client example using the `QUdpSocket` class.

- Implementing a server using the `QUdpSocket` class

In this example, when a message based on the UDP protocol is received, the `QTextEdit` widget displays the network information and message on the other end, as shown in the figure below. It then sends the message "Hello UDP Client~" to the other party.



Create a project based on the Qt Widget class when creating the project. In `widget.ui`, place the `QTextEdit` widget as shown below.



Then write the following code in the widget.h header file.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QUdpSocket>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
    QUdpSocket *udpSocket;

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

```
};

#endif // WIDGET_H
```

If you have done the above, write the following in the widget.cpp source file.

```
#include "widget.h"
#include "ui_widget.h"

#define SERVER_PORT 21000

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

    udpSocket = new QUdpSocket(this);
    udpSocket->bind(QHostAddress("127.1.1.1"), SERVER_PORT);

    connect(udpSocket, SIGNAL(readyRead()),
            this,      SLOT(readPendingDatagram()));
}

void Widget::readPendingDatagram()
{
    QByteArray buffer;
    buffer.resize(udpSocket->pendingDatagramSize());

    QHostAddress sender;
    quint16 senderPort;

    udpSocket->readDatagram(buffer.data(), buffer.size(),
                            &sender, &senderPort);

    QString msg = QString("Receive IP/PORT : %1 / %2 <br>"
                          "Receive Message: %3 <br>")
                .arg(sender.toString())
                .arg(senderPort)
                .arg(buffer.data());
```

```
ui->textEdit->append(msg);

QByteArray writeData;
writeData.append("SERVER : Hello UDP Client.~ ");
udpSocket->writeDatagram(writeData, sender, senderPort);

}

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

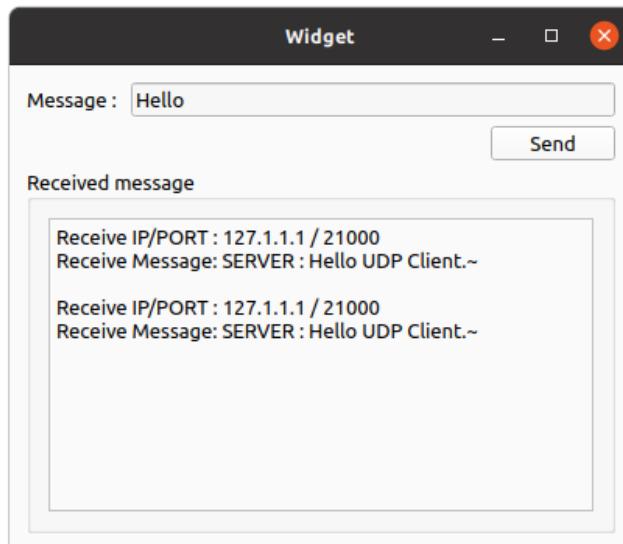
In the constructor function, the IP and PORT are set using the bind( ) function provided by the QUpdSocket class. When the message is received in the connect( ) function, the readyRead( ) signal is raised. This signal calls the readPendingDatagram( ) Slot function.

In the readPendingDatagram( ) Slot function, the pendingDatagramSize( ) member function returns the size of the received message. It then prints the received message to the QTextEdit widget and sends the message back to the client.

The source code for this example can be found in the 00\_UDP\_Server directory.

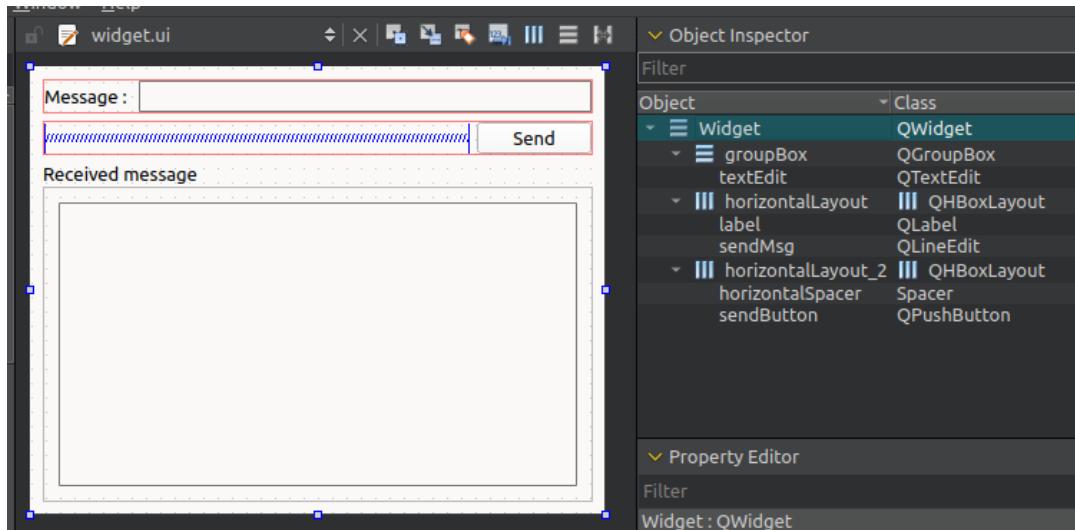
- Implementing the client using the QUpdSocket class

The following example is an example client implementation



As shown in the image above, enter a message to send to the UDP server and press the [Send] button to receive the "Hello UDP Client~" message from the server at the bottom.

When creating a project, create a project based on Qt Widget. Then, place the widget on the GUI as shown in the figure below.



Place the widget as shown above and write the source code below in the `widget.h` header file.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QUdpSocket>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
```

```
Ui::Widget *ui;
Q_udpSocket *udpSocket;

public slots:
    void sendButton();
    void readPendingDatagram();
};

#endif // WIDGET_H
```

As you can see in the example above, the sendButton( ) Slot function is called when the [Send] button is clicked, and the readPendingDatagram( ) Slot function is called when a message is received. The following example is the widget.cpp source code.

```
#include "widget.h"
#include "./ui_widget.h"

#define SERVER_PORT 21000
#define CLIENT_PORT 22000

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

    connect(ui->sendButton, &QPushButton::pressed,
            this,           &Widget::sendButton);

    udpSocket = new Q_udpSocket(this);
    udpSocket->bind(QHostAddress("127.1.1.1"), CLIENT_PORT);

    connect(udpSocket, SIGNAL(readyRead()),
            this,           SLOT(readPendingDatagram()));
}

void Widget::sendButton()
{
    QByteArray msg;
    msg = ui->sendMsg->text().toLocal8Bit();

    QHostAddress sender("192.168.117.1");
```

```
    udpSocket->writeDatagram(msg, sender, SERVER_PORT);
}

void Widget::readPendingDatagram()
{
    QByteArray buffer;
    buffer.resize(udpSocket->pendingDatagramSize());

    QHostAddress sender;
    quint16 senderPort;

    udpSocket->readDatagram(buffer.data(), buffer.size(),
                           &sender, &senderPort);

    QString msg = QString("Receive IP/PORT : %1 / %2 <br>"
                          "Receive Message: %3 <br>")
        .arg(sender.toString())
        .arg(senderPort)
        .arg(buffer.data());

    ui->textEdit->append(msg);
}

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

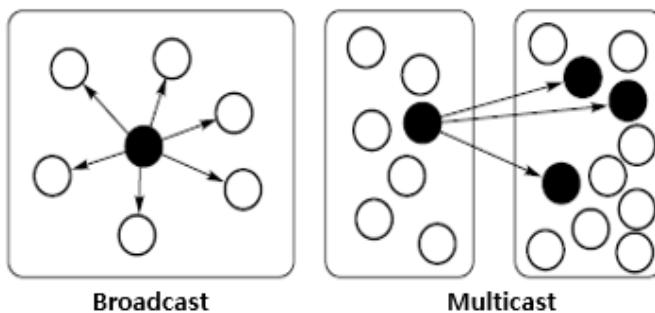
The `readPendingDatagram()` Slot function prints the received message in the `QTextEdit` widget when it receives a message from the server. The source code for this example can be found in the `00_UDP_Client` directory.

## 28.4. Broadcast

When categorized from the perspective of sender and receiver, UDP-based protocols can be used to send and receive data on a computer network in two ways: broadcast and multicast.

Broadcast is a method in which a single sender can deliver a message to all receivers. For example, if a sender is sending data to a specific PORT, all receivers listening to the same PORT of the sender can receive the message.

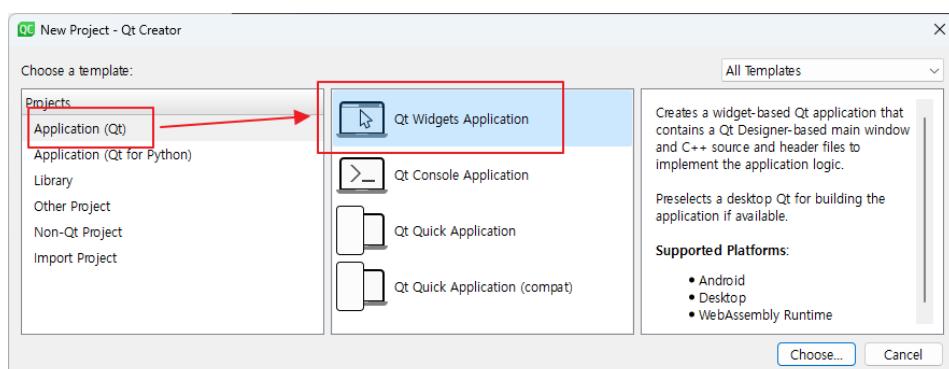
With Broadcast, when a sender sends data, all receivers in that network group can receive it.



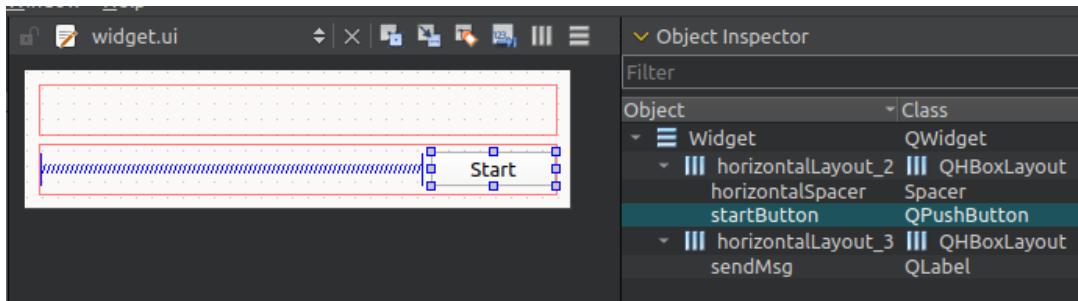
Multicast is a method that allows only specified users to send/receive messages. Therefore, in this chapter, we will cover Broadcast and Multicast in detail with examples.

- Example of sending data using the Broadcast method

Create a Qt Widget-based project.



c After creating the project, double-click the widget.ui file to place the widget on the GUI as shown below.



After clicking the Start button, all recipients on the network group will be able to receive the message. Next, write the following code in the widget.h header file.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QUdpSocket>
#include <QTimer>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;

    QUdpSocket *udpSocket;
    QTimer *timer;
    int msgNumber;

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

```
void broadcastSend();  
};  
#endif // WIDGET_H
```

A timer object of class QTimer sends a Broadcast message repeatedly at a specified time. The following is the source code for widget.cpp.

```
#include "widget.h"  
#include "./ui_widget.h"  
  
Widget::Widget(QWidget *parent)  
    : QWidget(parent)  
    , ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
  
    timer = new QTimer(this);  
    udpSocket = new QUdpSocket(this);  
    msgNumber = 1;  
  
    connect(ui->startButton, SIGNAL(clicked()),  
            this, SLOT(startButton()));  
  
    connect(timer, SIGNAL(timeout()),  
            this, SLOT(broadcastSend()));  
}  
  
void Widget::startButton()  
{  
    if(!timer->isActive())  
        timer->start(1000);  
}  
  
void Widget::broadcastSend()  
{  
    ui->sendMsg->setText(QString("Sending message %1").arg(msgNumber));  
  
    QByteArray datagram = "Broadcast number "  
                           + QByteArray::number(msgNumber);  
    udpSocket->writeDatagram(datagram.data(), datagram.size(),  
                               QHostAddress::Broadcast, 35000);  
    msgNumber++;
```

```
}

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

When the [Start] button is clicked, the timer object of the QTimer class is executed periodically once every second. The broadcastSend( ) function is executed every second. In the broadcastSend( ) function, the third argument of the writeDatagram( ) function can be used to send data in the broadcast method.

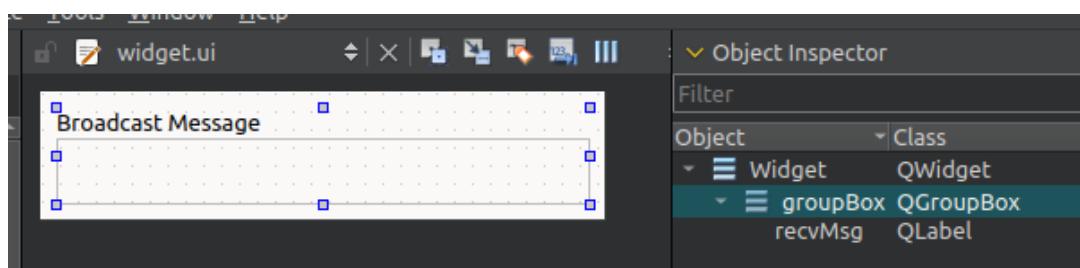
Using the QHostAddress::Broadcast value as the third argument sends the message to all computers in the network group.



The source code for this example refers to the 00\_Broadcast\_Sender directory.

- Example of receiving Broadcast

c The following example is an example of receiving a Broadcast message. After creating a Qt Widget-based project, place the widget on the GUI as shown below.



c Then, write the widget.h header file like this

```
#ifndef WIDGET_H
#define WIDGET_H
```

```
#include <QWidget>
#include <QUdpSocket>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;

private slots:
    void readDatagrams();

};

#endif // WIDGET_H
```

Next, write the widget.cpp source file as shown below.

```
#include "widget.h"
#include "./ui_widget.h"

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

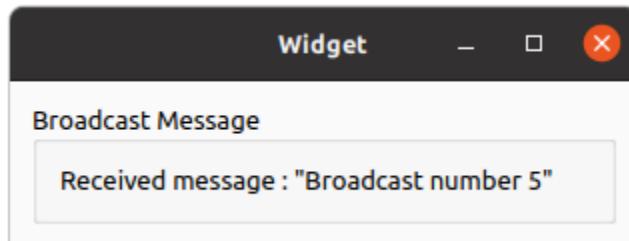
    udpSocket = new QUdpSocket(this);
    udpSocket->bind(35000, QUdpSocket::ShareAddress);

    connect(udpSocket, SIGNAL(readyRead()), this, SLOT(readDatagrams()));
}

void Widget::readDatagrams()
```

```
{  
    while (udpSocket->hasPendingDatagrams())  
    {  
        QByteArray datagram;  
        datagram.resize(udpSocket->pendingDatagramSize());  
  
        udpSocket->readDatagram(datagram.data(), datagram.size());  
  
        ui->recvMsg->setText(tr("Received Message : \"%1\"")  
                             .arg(datagram.data()));  
    }  
}  
  
Widget::~Widget()  
{  
    delete ui;  
}
```

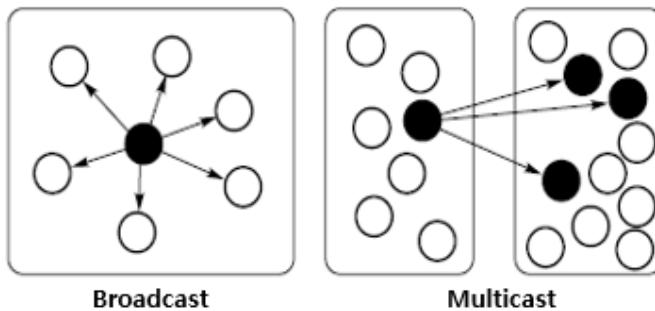
c When you use the bind( ) function in the constructor function, you can receive messages from the sending example by using the same PORT as the sending example.



You can find the source code for this example in the 00\_Broadcast\_Receiver directory.

## 28.5. Multicast

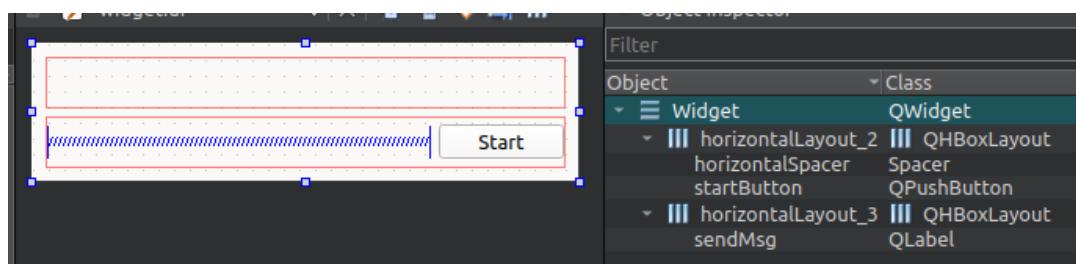
In this chapter, you'll learn how to use Multicast to send and receive messages with examples. Multicast allows you to send and receive messages to specific computers (or users).



As a first example, we'll implement an example of sending a message using the Multicast method, and secondly, we'll learn how to receive a message.

- Implementing the Sender example with Multicast

Create a project that inherits from the QWidget class and place the widget on the GUI as shown in the following figure.



As shown in the image above, click the [Start] button to send messages using the Multicast method. Next, open the `widget.h` header file and write the source code like below.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
```

```
#include <QUdpSocket>
#include <QTimer>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;

    QUdpSocket udpSocket4;
    QUdpSocket udpSocket6;
    QHostAddress groupAddress4;
    QHostAddress groupAddress6;

    QTimer *timer;
    int msgNumber;

private slots:
    void startButton();
    void multicastSend();
};

#endif // WIDGET_H
```

QTimer 클래스의 timer 오브젝트는 지정한 시간을 반복해 브로드캐스트 메시지를 전송 한다. 다음은 widget.cpp 소스코드이다.

```
#include "widget.h"
#include "ui_widget.h"

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

```
groupAddress4 = QHostAddress(QStringLiteral("239.255.43.21"));
groupAddress6 = QHostAddress(QStringLiteral("ff12::2115"));

timer = new QTimer(this);
udpSocket4.bind(QHostAddress(QHostAddress::AnyIPv4), 0);
udpSocket6.bind(QHostAddress(QHostAddress::AnyIPv6),
udpSocket4.localPort());

msgNumber = 1;

connect(ui->startButton, SIGNAL(clicked()),
        this, SLOT(startButton()));

connect(timer, SIGNAL(timeout()),
        this, SLOT(multicastSend()));
}

void Widget::startButton()
{
    if(!timer->isActive())
        timer->start(1000);
}

void Widget::multicastSend()
{
    ui->sendMsg->setText(QString("Sending message %1").arg(msgNumber));

    QByteArray datagram = "Multicast number "
                          + QByteArray::number(msgNumber);

    udpSocket4.writeDatagram(datagram, groupAddress4, 45000);
    if (udpSocket6.state() == QAbstractSocket::BoundState)
        udpSocket6.writeDatagram(datagram, groupAddress6, 45000);

    msgNumber++;
}

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

When you click the [Start] button, the timer object of the QTimer class is executed periodically, once every second. And every second, it runs the multicastSend ( ) function. In the multicastSend ( ) function, the first argument to the writeDatagram( ) function is the message to send, the second is a specific group on the network, and the third is the PORT number to send to.

Next, let's build and run it.



You can find the source code for this example in the 00\_Multicast\_Sender directory.

- Implementing the Receiver example with Multicast

In this example, we will write an example to receive the mesh sent in the previous example. When creating a project, create a project based on Qt Widgets. Then open widget.ui and place the widgets on the GUI as shown below.



Create the widget.h header file as shown above.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QUdpSocket>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE
```

```
class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;

private:
    QUdpSocket udpSocket4;
    QUdpSocket udpSocket6;
    QHostAddress groupAddress4;
    QHostAddress groupAddress6;

private slots:
    void readDatagrams();

};

#endif // WIDGET_H
```

Next, open the widget.cpp source file and write the source code as shown below.

```
#include "widget.h"
#include "ui_widget.h"
#include <QNetworkDatagram>

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

    groupAddress4 = QHostAddress(QStringLiteral("239.255.43.21"));
    groupAddress6 = QHostAddress(QStringLiteral("ff12::2115"));

    udpSocket4.bind(QHostAddress::AnyIPv4, 45000,
                    QUdpSocket::ShareAddress);
    udpSocket4.joinMulticastGroup(groupAddress4);
```

```
if (!udpSocket6.bind(QHostAddress::AnyIPv6, 45000,
                      QUdpSocket::ShareAddress) ||
    !udpSocket6.joinMulticastGroup(groupAddress6))
qDebug() << Q_FUNC_INFO << "IPv4 Multicast only.";

connect(&udpSocket6, &QUdpSocket::readyRead,
        this,           &Widget::readDatagrams);
}

void Widget::readDatagrams()
{
    QByteArray datagram;

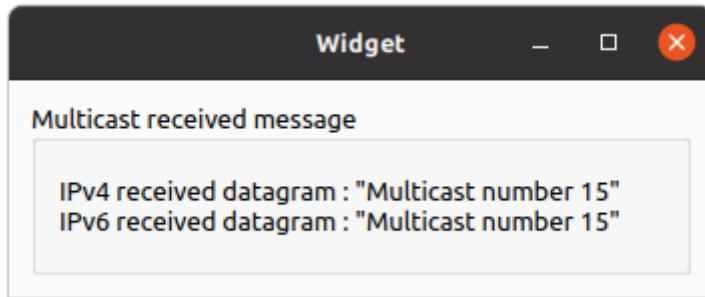
    while (udpSocket4.hasPendingDatagrams())
    {
        datagram.resize(int(udpSocket4.pendingDatagramSize()));
        udpSocket4.readDatagram(datagram.data(), datagram.size());
        ui->recvMsg->setText(tr("IPv4 received datagram : \"%1\"")
                               .arg(datagram.constData()));
    }

    while (udpSocket6.hasPendingDatograms())
    {
        QNetworkDatagram dgram = udpSocket6.receiveDatagram();
        ui->recvMsg->setText(ui->recvMsg->text() +
                               tr("\nIPv6 received datagram : \"%1\"")
                               .arg(dgram.data().constData()));
    }
}

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

You must use the same Address to receive messages from the Sender example. By using the same group Address, we can receive messages from the Sender example. After writing the source code, build and run it as shown in the image below.

Jesus loves you.



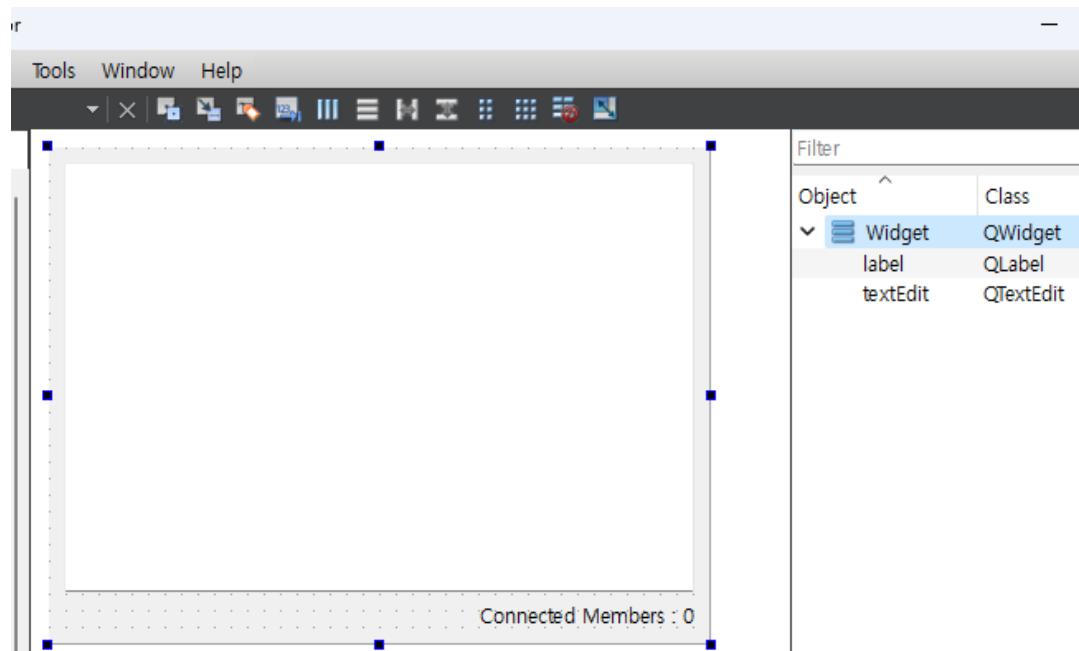
For the complete source code for this example, see the 00\_Multicast\_Receiver directory.

## 28.6. Implementing chatting server and client

In this chapter, we'll implement a simple chat server example and a client example using the QTcpServer and QTcpSocket classes.

- Implement the chat server example

Place the widget on the GUI as shown in the figure above. On the GUI, the QTextEdit widget displays information about new users joining, messages they send, and disconnected users.



Once you have it set up like above, next add the ChatServer class to your project and write the following in your chatserver.h header file

```
#ifndef CHATSERVER_H
#define CHATSERVER_H

#include <QObject>
#include <QTcpServer>
#include <QTcpSocket>
```

```

class ChatServer : public QTcpServer
{
    Q_OBJECT
public:
    ChatServer(QObject *parent = nullptr);
    ~ChatServer();

private slots:
    void readyRead();
    void disconnected();
    void sendUserList();

signals:
    void clients_signal(int users);
    void message_signal(QString msg);

protected:
    void incomingConnection(qintptr socketfd);

private:
    QSet<QTcpSocket*> clients;
    QMap<QTcpSocket*,QString> users;
};

#endif // CHATSERVER_H

```

The incomingConnection( ) function declared in the protected access restrictor in the header above is the function that is called when a new client connects. The clients\_signal( ) signal is raised in this function.

We pass the number of users currently connected to the server as an argument to this signal, and we associate this function with the signal so that the readyRead( ) slot function can be called when we receive a message for the client. So when the client sends a message, the readyRead( ) slot function is called.

The disconnected( ) signal is raised when the client disconnects, and the disconnected( ) Slot function associated with this signal is called. The following example source code is from ChatServer.cpp.

```

#include "chatserver.h"
#include <QRegularExpression>
#include <QRegularExpressionMatch>

```

```
ChatServer::ChatServer(QObject *parent)
    : QTcpServer(parent)
{
}

void ChatServer::incomingConnection(qintptr socketfd)
{
    QTcpSocket *client = new QTcpSocket(this);
    client->setSocketDescriptor(socketfd);
    clients.insert(client);

    emit clients_signal(clients.count());

    QString str;
    str = QString("New Member: %1")
        .arg(client->peerAddress().toString());

    emit message_signal(str);

    connect(client, SIGNAL(readyRead()), this,
            SLOT(readyRead()));
    connect(client, SIGNAL(disconnected()), this,
            SLOT(disconnected()));
}

void ChatServer::readyRead()
{
    QTcpSocket *client = (QTcpSocket*)sender();
    while(client->canReadLine())
    {
        QString line = QString::fromUtf8(client->readLine()).trimmed();

        QString str;
        str = QString("Read line: %1").arg(line);

        emit message_signal(str);

        QRegularExpression meRegex("^/me:(.*)$");
        QRegularExpressionMatch match = meRegex.match(line);

        if(match.hasMatch())
```

```
{  
    QString user = match.captured(1);  
    users[client] = user;  
    foreach(QTcpSocket *client, clients)  
    {  
        client->write(QString("Server: %1 connected\n")  
                      .arg(user).toUtf8());  
    }  
  
    //sendUserList();  
}  
else if(users.contains(client))  
{  
    QString message = line;  
    QString user = users[client];  
  
    QString str;  
    str = QString("User name: %1, Message: %2")  
          .arg(user, message);  
    emit message_signal(str);  
  
    foreach(QTcpSocket *otherClient, clients)  
        otherClient->write(QString(user+": "+message+"\n")  
                           .toUtf8());  
}  
}  
}  
  
void ChatServer::disconnected()  
{  
    QTcpSocket *client = (QTcpSocket*)sender();  
  
    QString str;  
    str = QString("Disconnect: %1")  
          .arg(client->peerAddress().toString());  
  
    emit message_signal(str);  
  
    clients.remove(client);  
  
    emit clients_signal(clients.count());
```

```

QString user = users[client];
users.remove(client);

sendUserList();
foreach(QTcpSocket *client, clients)
    client->write(QString("Server: %1 Disconnect").arg(user).toUtf8());
}

void ChatServer::sendUserList()
{
    QStringList userList;
    foreach(QString user, users.values())
        userList << user;

    foreach(QTcpSocket *client, clients)
        client->write(QString("User:" + userList.join(",") + "\n").toUtf8());
}

ChatServer::~ChatServer()
{
    deleteLater();
}

```

Creating a new QTcpSocket in the incomingConnection( ) function is intended to create a new client's socket object. Therefore, the object of class QTcpSocket created in this function is stored in the client QMap container named users.

The readyRead( ) function is a Slot function that is called when a client connected to the server sends a message. This function sends the message from the client to all clients connected to the server.

The disconnected( ) function is a Slot function that is called when a client disconnects. It removes the object of class QTcpSocket of the disconnected client from the QMap container named users. The following is the widget.h header file. Write the source code as shown below.

```

#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include "chatserver.h"

```

```
QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
    ChatServer *server;

private slots:
    void slot_clients(int users);
    void slot_message(QString msg);
};

#endif // WIDGET_H
```

In the Widget class, the slot\_clients( ) Slot function is a signal that is fired by the ChatServer class when a new client joins or disconnects.

And the slot\_message( ) function is a Slot function that is called when a client sends a message or disconnects from the ChatServer class. The source for the following example is the widget.cpp source code.

```
#include "widget.h"
#include "./ui_widget.h"

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

    server = new ChatServer();
    connect(server, SIGNAL(clients_signal(int)), this,
            SLOT(slot_clients(int)));
```

```
connect(server, SIGNAL(message_signal(QString)), this,
        SLOT(slot_message(QString)));

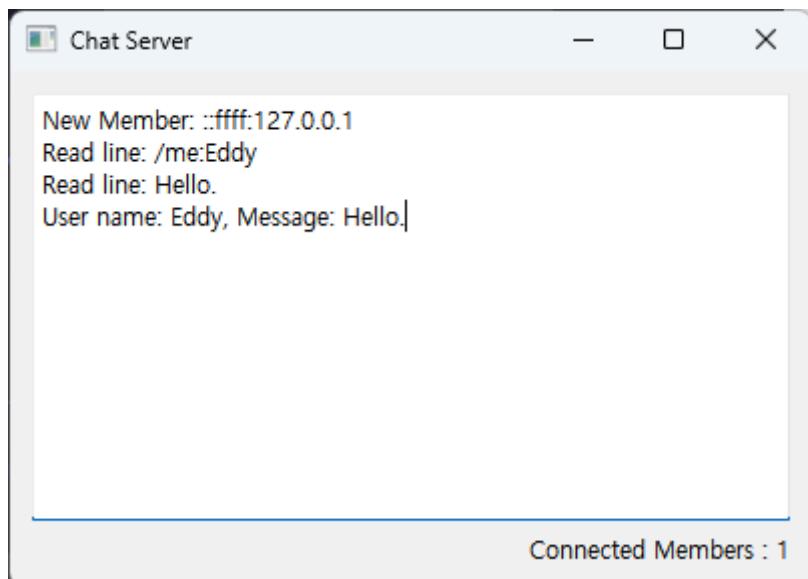
server->listen(QHostAddress::Any, 35000);
}

void Widget::slot_clients(int users)
{
    QString str = QString("Connected Members : %1").arg(users);
    ui->label->setText(str);
}

void Widget::slot_message(QString msg)
{
    ui->textEdit->append(msg);
}

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

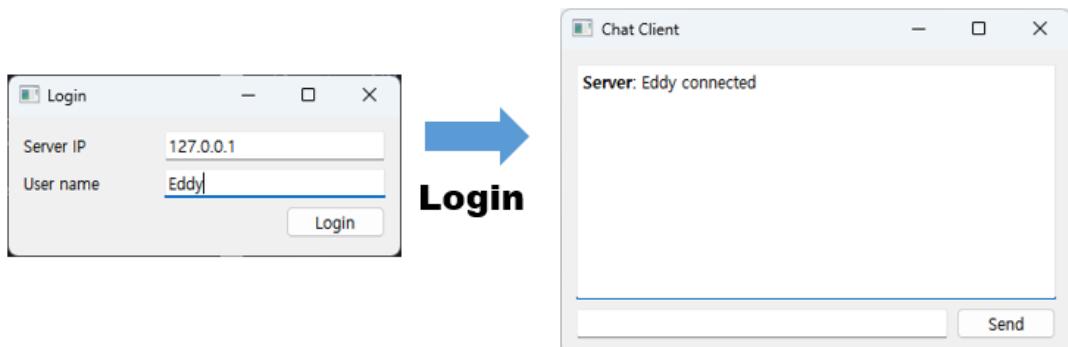
The slot\_clients( ) Slot function updates the number of clients in the GUI when called. And the slot\_message( ) Slot function outputs a message to the QTextEdit window. If you have completed the above code, build and run it.



As shown in the figure above, the user's access information and message information are displayed on the GUI. You can refer to the 00\_ChatServer directory for the source code of this example.

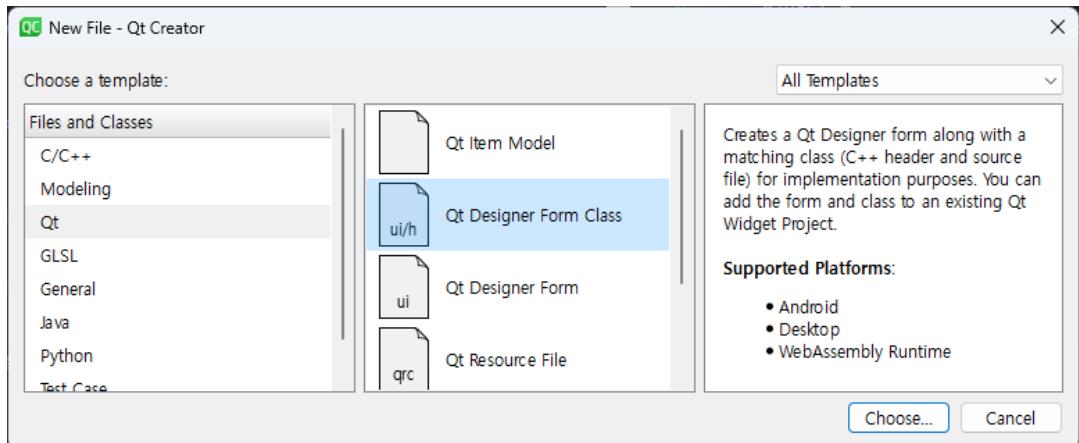
- Implement the chat client example

c In this lesson, we'll implement a chat server and a client that sends and receives messages. The chat client consists of two screens. The first is the login screen, as shown in the image below. Clicking the [login] button, as shown in the login widget, will connect to the chat server with the server IP address. The login widget is then hidden and the chat client widget is activated (shown).

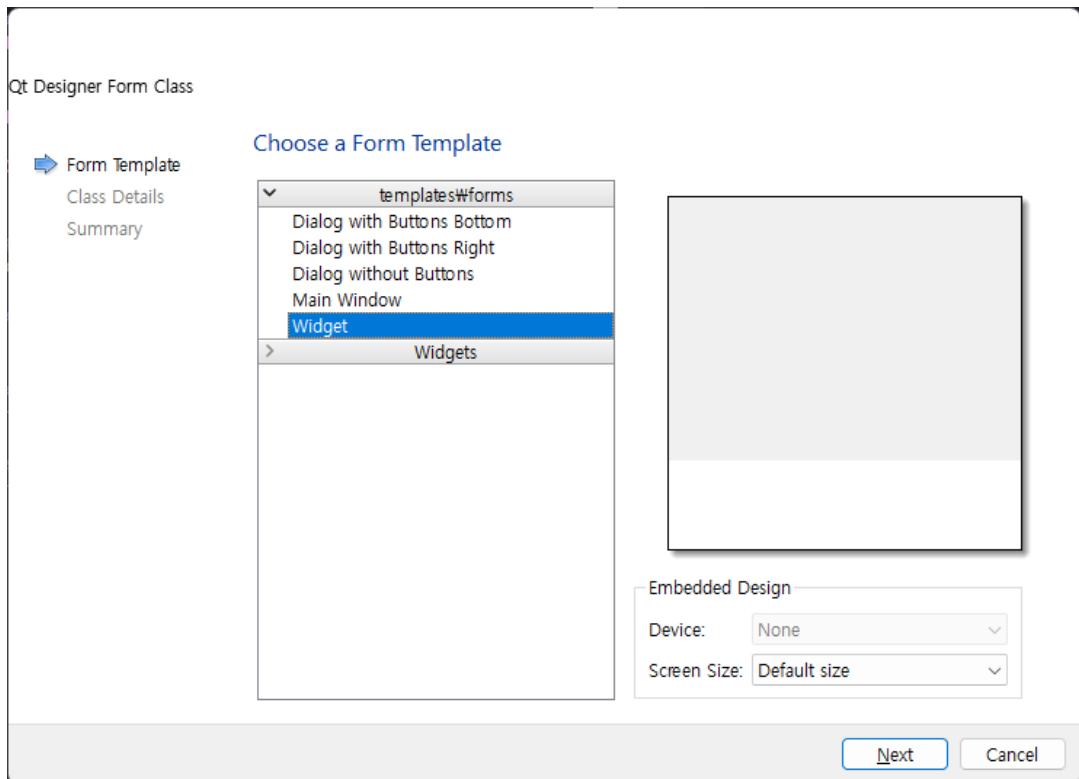


The QTextEdit widget in the middle of the chat client widget outputs the message received from the server. The QLineEdit at the bottom is where you enter the message to be sent to the server. Then click the [Send] button to send the message to the chat server.

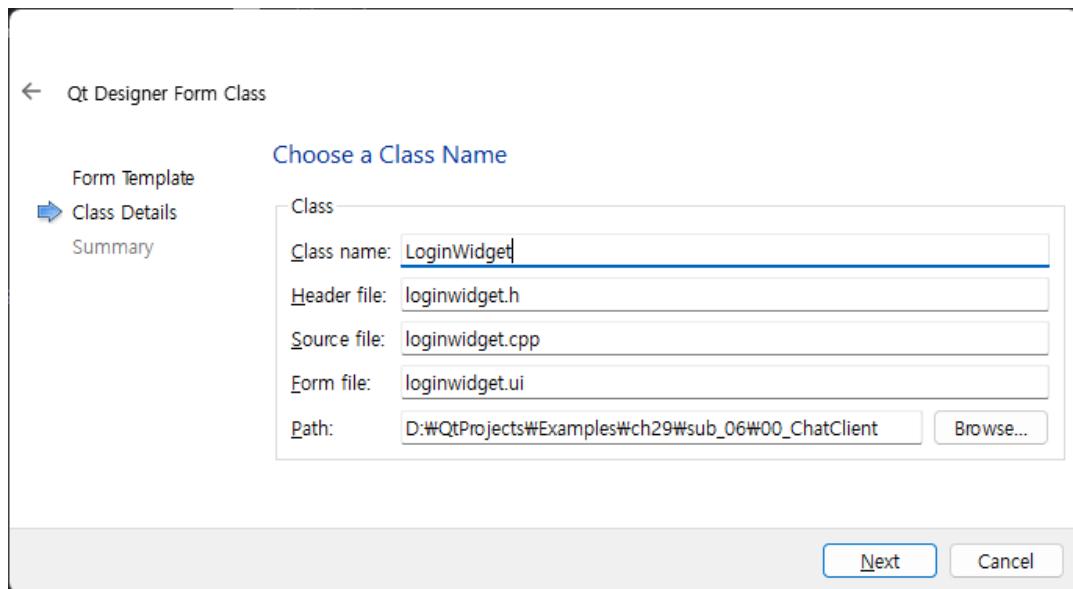
Create a Qt Widget-based project. Select [Qt Designer Form Class] as shown in the figure below and click [Choose] button at the bottom.



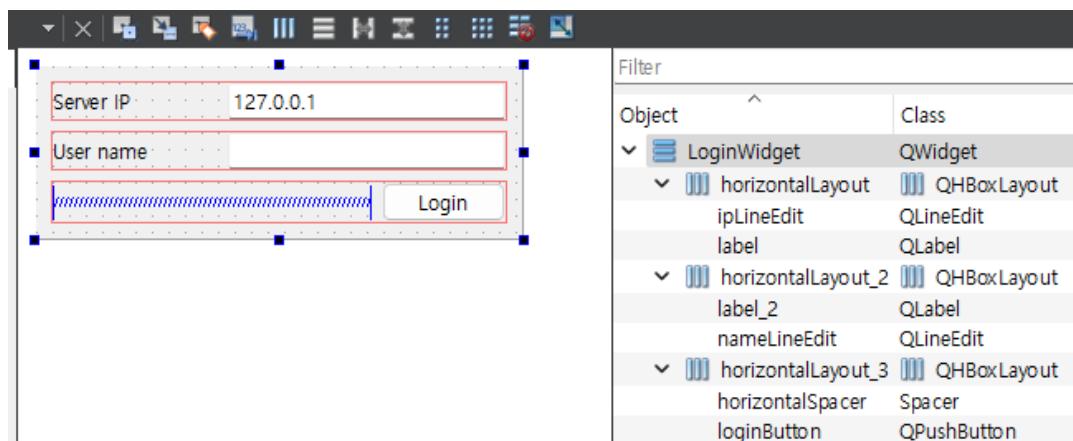
c On the next dialog screen, select [Widget] and click the [Next] button.



Enter the Class information as shown in the dialog below.



Once you have added the Form Class as shown above, open the loginwidget.ui file and place the widgets on the GUI as shown in the image below.



Next, create the loginwidget.h header file as shown below.

```
#ifndef LOGINWIDGET_H
#define LOGINWIDGET_H

#include <QWidget>

namespace Ui {
class LoginWidget;
}

class LoginWidget : public QWidget
```

```
{  
    Q_OBJECT  
  
public:  
    explicit LoginWidget(QWidget *parent = nullptr);  
    ~LoginWidget();  
  
private:  
    Ui::LoginWidget *ui;  
  
private slots:  
    void loginBtnClicked();  
  
signals:  
    void sig_loginInfo(QString addr, QString name);  
};  
  
#endif // LOGINWIDGET_H
```

Next, create the widget.cpp source file as shown below.

```
#include "loginwidget.h"  
#include "ui_loginwidget.h"  
  
LoginWidget::LoginWidget(QWidget *parent) :  
    QWidget(parent),  
    ui(new Ui::LoginWidget)  
{  
    ui->setupUi(this);  
    connect(ui->loginButton, &QPushButton::pressed,  
            this, &LoginWidget::loginBtnClicked);  
}  
  
void LoginWidget::loginBtnClicked()  
{  
    QString serverIp = ui->ipLineEdit->text().trimmed();  
    QString name = ui->nameLineEdit->text().trimmed();  
  
    emit sig_loginInfo(serverIp, name);  
}  
  
LoginWidget::~LoginWidget()  
{
```

```
    delete ui;  
}
```

When you click the [Login] button on the GUI, the Server IP and User name are saved in a QString. Then, the value stored in the QString is passed as an argument to the sig\_loginInfo( ) signal. This signal is associated with the Slot function of the Widget class. The following is the source code for the widget.h header file.

```
#ifndef WIDGET_H  
#define WIDGET_H  
  
#include <QWidget>  
#include <QTcpSocket>  
#include "loginwidget.h"  
  
QT_BEGIN_NAMESPACE  
namespace Ui { class Widget; }  
QT_END_NAMESPACE  
  
class Widget : public QWidget  
{  
    Q_OBJECT  
  
public:  
    Widget(QWidget *parent = nullptr);  
    ~Widget();  
  
private:  
    Ui::Widget *ui;  
    LoginWidget *loginWidget;  
  
    QTcpSocket *socket;  
    QString    ipAddr;  
    QString    userName;  
  
private slots:  
    void loginInfo(QString addr, QString name);  
    void sayButton_clicked();  
    void connected();  
    void readyRead();  
};
```

```
#endif // WIDGET_H
```

The loginInfo( ) Slot function is a Slot function that is called when the [Login] button is clicked in the LoginWidget class, and a signal is generated to pass the server IP address and username entered in the login window. The first argument is the server IP address and the second argument is the username. The following example shows the widget.cpp source code.

```
#include "widget.h"
#include "./ui_widget.h"

#include <QRegularExpression>
#include <QRegularExpressionMatch>

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

    loginWidget = new LoginWidget();

    connect(loginWidget, &LoginWidget::sig_loginInfo,
            this,           &Widget::loginInfo);

    connect(ui->sayButton, &QPushButton::pressed,
            this,           &Widget::sayButton_clicked);

    loginWidget->show();

    socket = new QTcpSocket(this);
    connect(socket, SIGNAL(readyRead()),
            this,   SLOT(readyRead()));
    connect(socket, SIGNAL.connected(),
            this,   SLOT(connected()));
}

void Widget::loginInfo(QString addr, QString name)
{
    ipAddr = addr;
    userName = name;
```

```
socket->connectToHost(ipAddr, 35000);
}

void Widget::sayButton_clicked()
{
    QString message = ui->sayLineEdit->text().trimmed();

    if(!message.isEmpty())
    {
        socket->write(QString(message + "\n").toUtf8());
    }

    ui->sayLineEdit->clear();
    ui->sayLineEdit->setFocus();
}

void Widget::connected()
{
    loginWidget->hide();
    this->window()->show();

    socket->write(QString("/me:" + userName + "\n").toUtf8());
}

void Widget::readyRead()
{
    while(socket->canReadLine())
    {
        QString line = QString::fromUtf8(socket->readLine()).trimmed();

        QRegularExpression re("^(?:[^\r\n]+):(.*)$");
        QRegularExpressionMatch match = re.match(line);

        if(match.hasMatch())
        {
            QString user = match.captured(1);
            QString message = match.captured(2);

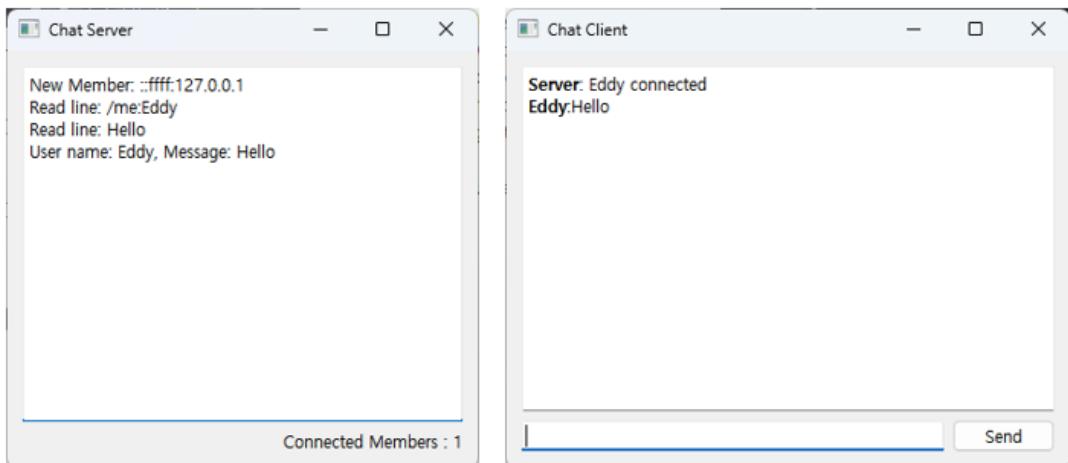
            ui->roomTextEdit->append("<b>" + user + "</b>:" + message);
        }
    }
}
```

```
}
```

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

When the [Send] button is clicked, the sayButton\_clicked( ) Slot function is called. The connected( ) Slot function is called when the connection with the chat server is complete. readyRead( ) is a Slot function that is called when the chat server sends a message. This Slot function outputs the message to the QTextEdit widget.

Let's build this example and then try it out. The server example should be running before running this example.



You can find the source code for this example in the 00\_ChatClient directory.

## 29. Qt WebSocket

WebSocket is a TCP-based protocol that was created to overcome the shortcomings of HTTP used on the web.

The HTTP protocol is the protocol used on the web. Web browsers and web servers are the most common examples of how clients and servers use the HTTP protocol to communicate data.

The Web browser requests data from the Web server, and the Web server sends the requested data to the Web browser. Then, when the transfer is complete, the connection between the Web browser and the Web server is terminated.

Because the Web server always terminates the response whenever the Web browser makes a request, this is a significant waste of time and resources.

WebSockets were created to overcome these shortcomings. WebSockets do not close the connection when the response is complete. Therefore, if a web browser is connected to a web server, it can reuse the connected session. This saves time and wasted resources on reconnecting.

WebSockets can be used by a variety of applications, not just web browsers and web servers.

In other words, you can use WebSockets in the same way that you used TCP Sockets in the server/client concept, just as you used TCP Sockets to communicate.

To provide WebSockets, Qt provides the Qt WebSockets module. The Qt WebSockets module makes it easy to implement applications that can communicate with web browsers.

In addition, the Qt WebSockets module makes it easy to implement server applications that can communicate with web browsers, in the same way that Node.js makes it easy to develop server-side applications.

To use the WebSocket module in Qt, you need to add the following to your project file. If you are using CMake, you can add it like this

```
find_package(Qt6 REQUIRED COMPONENTS WebSockets)
target_link_libraries(mytarget PRIVATE Qt6::WebSockets)
```

If you're using qmake, add it like this

```
QT += websockets
```

The Qt WebSockets module provides the `QWebSocket` class for easy application development. You can use `QWebSocket` for server programming, but it also provides the `QWebSocketServer` class to make it easier to develop server applications.

The `QWebSocket` class can be used in much the same way as `QTcpSocket`, which was covered in the Network Programming chapter, and uses Signals and Slots for easy implementation without the need for a complex Thread implementation.

For example, when a connection is completed using `QWebSocket`, the `connected()` signal provided by `QWebSocket` is emitted. When this signal is fired, the associated Slot function is called, making it easy to implement without threads. Here is some example source code that shows the connection between Signal and Slot.

```
EchoClient::EchoClient(const QUrl &url, bool debug, QObject *parent)
    : QObject(parent), m_url(url)
{
    connect(&m_webSocket, &QWebSocket::connected,
            this,           &EchoClient::onConnected);
    connect(&m_webSocket, &QWebSocket::disconnected,
            this,           &EchoClient::closed);

    m_webSocket.open(QUrl(url));
}

void EchoClient::onConnected()
{
    connect(&m_webSocket, &QWebSocket::textMessageReceived,
            this,           &EchoClient::onTextMessageReceived);

    m_webSocket.sendTextMessage(QStringLiteral("Hello, world!"));
}

void EchoClient::onTextMessageReceived(QString message)
{
    qDebug() << "Message received:" << message;
```

}

The above example is part of an example source code implemented using the QWebSocket class. When the connection to the server is established, the connected( ) signal is raised and the onConnected( ) Slot function associated with this signal is called. Then, as shown in the onConnected( ) function, when a message is received from the server, a signal is raised and the onTextMessageReceived( ) Slot function associated with this signal is executed.

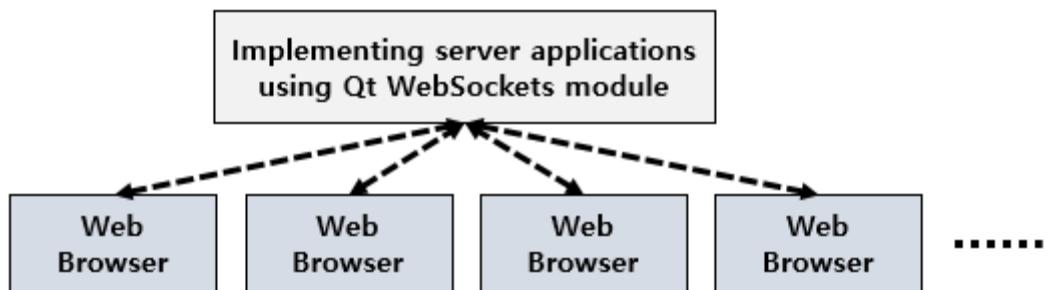
When the connection to the server is disconnected (Close), the disconnected( ) signal is raised and the Slot function called closed( ) is called.

As you can see from the example above, the Qt WebSockets module makes it easy to implement an application that can communicate with a web browser.

In the following, we will implement an example using the Web Socket protocol between a web browser and an example implemented using the Qt WebSockets module.

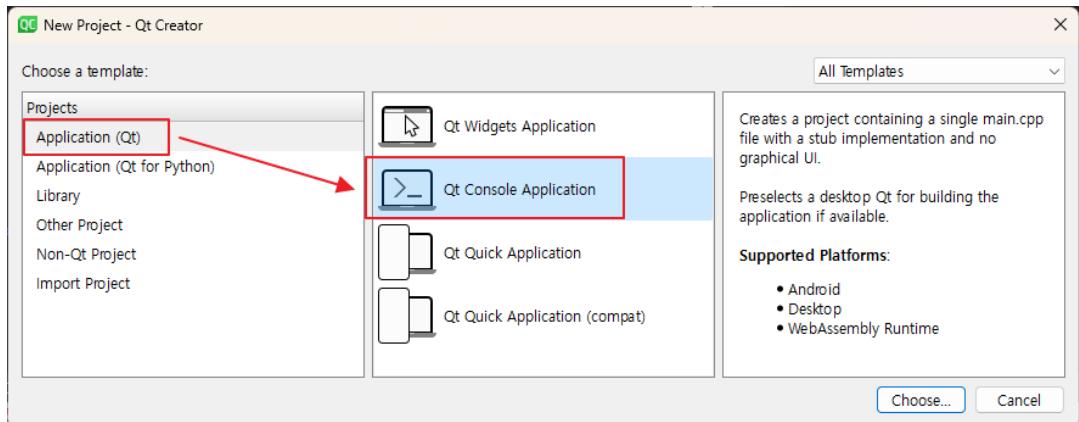
- ✓ Implement a server application to communicate with a web browser.

In this example, we will first write the source code for communicating with the server application from the Web Browser and then implement the server application using the Qt WebSockets module.

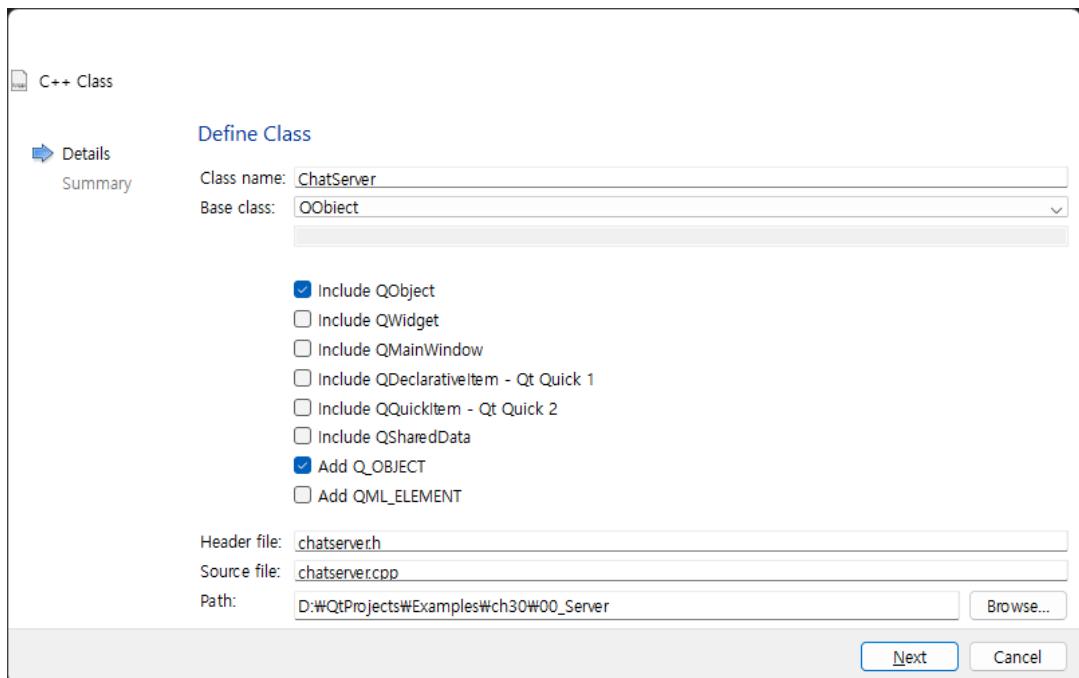


We'll implement a WebSocket that works in a web browser using HTML5, as shown in the image above. And we'll implement the server application using Qt.

First, let's implement an example server application based on Qt. Create a Console-based project as shown in the image below.



After creating the project, add the ChatServer class as shown below.



And to make the Qt WebSockets module available to your project, add the following to your CMakeList.txt file to enable the WebSocket module.

```
cmake_minimum_required(VERSION 3.14)

project(00_Server LANGUAGES CXX)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)
```

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Qt NAMES Qt6 Qt5 REQUIRED COMPONENTS Core WebSockets)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Core WebSockets)

add_executable(00_Server
    main.cpp
    chatserver.h chatserver.cpp
)
target_link_libraries(00_Server Qt${QT_VERSION_MAJOR}::Core)
target_link_libraries(00_Server Qt${QT_VERSION_MAJOR}::WebSockets)

include(GNUInstallDirs)
install(TARGETS 00_Server
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

Next, open the chatserver.h header file and write the source code like below.

```
#ifndef CHATSERVER_H
#define CHATSERVER_H

#include <QObject>
#include <QWebSocketServer>
#include <QWebSocket>

class ChatServer : public QObject
{
    Q_OBJECT
public:
    explicit ChatServer(quint16 port, QObject *parent = nullptr);
    virtual ~ChatServer();

private Q_SLOTS:
    void onNewConnection();
    void processMessage(QString message);
    void socketDisconnected();
```

```
private:  
    QWebSocketServer *m_pWebSocketServer;  
    QList<QWebSocket *> m_clients;  
};  
  
#endif // CHATSERVER_H
```

onNewConnection( ) Slot function is called when a new connection signal occurs.

The processMessage( ) Slot function forwards messages sent by the client to all clients (Web browsers) connected to the server.

The socketDisconnected( ) Slot function removes the QWebSocket class object of the connected client when a particular client connection is terminated. The following example source code is from the chatserver.cpp source file. Write the source code as shown below.

```
#include "chatserver.h"  
  
ChatServer::ChatServer(quint16 port, QObject *parent)  
    : QObject{parent}  
{  
    m_pWebSocketServer = new QWebSocketServer(  
        QStringLiteral("Chat Server"),  
        QWebSocketServer::NonSecureMode,  
        this);  
  
    if (m_pWebSocketServer->listen(QHostAddress::Any, port))  
    {  
        qDebug() << "Chat Server listening on port" << port;  
        connect(m_pWebSocketServer, &QWebSocketServer::newConnection,  
                this, &ChatServer::onNewConnection);  
    }  
}  
  
void ChatServer::onNewConnection()  
{  
    qDebug() << "New Connection ";  
  
    QWebSocket *pSocket = m_pWebSocketServer->nextPendingConnection();  
  
    connect(pSocket, &QWebSocket::textMessageReceived,
```

```
        this, &ChatServer::processMessage);
connect(pSocket, &QWebSocket::disconnected,
        this, &ChatServer::socketDisconnected);

    m_clients << pSocket;
}

void ChatServer::processMessage(QString message)
{
    QWebSocket *pSender = qobject_cast<QWebSocket *>(sender());
    Q_FOREACH (QWebSocket *pClient, m_clients)
    {
        if (pClient != pSender)
        {
            pClient->sendTextMessage(message);
        }
    }

    qDebug() << "Send Message : " << message;
}

void ChatServer::socketDisconnected()
{
    qDebug() << "Client disconnect";

    QWebSocket *pClient = qobject_cast<QWebSocket *>(sender());
    if (pClient)
    {
        m_clients.removeAll(pClient);
        pClient->deleteLater();
    }
}

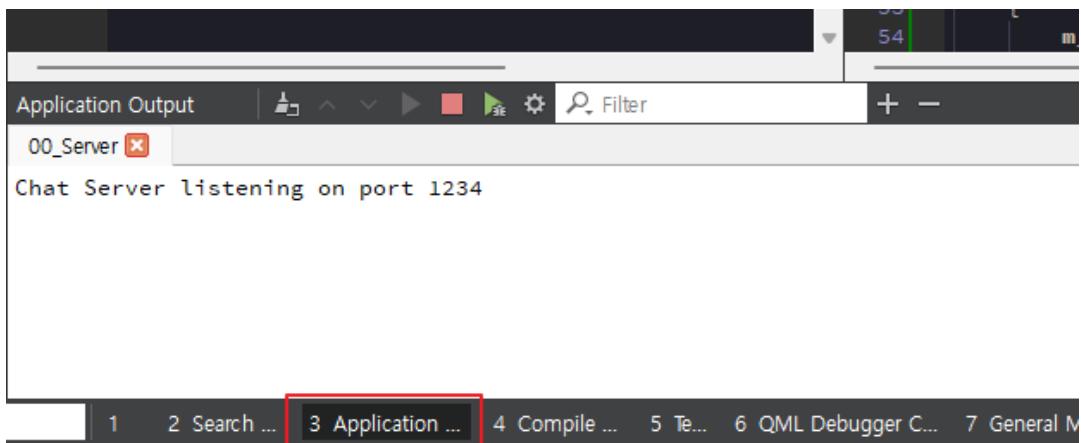
ChatServer::~ChatServer()
{
    m_pWebSocketServer->close();
    qDeleteAll(m_clients.begin(), m_clients.end());
}
```

위와 같이 작성한 ChatServer 클래스를 main.cpp에서 다음과 같이 수행한다. 다음 예제 소스코드는 main.cpp 소스코드이다.

```
#include <QtCore/QCoreApplication>
#include "chatserver.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    ChatServer server(1234);
    return a.exec();
}
```

In the example source code above, the first argument of the ChatServer object is the port number. If you run the application with the above code, you'll see it loaded in the Application Output window of Qt Creator as shown below.



The server is complete. Now let's write some HTML so that a web browser can access the server we created. You don't need to be a Qt Creator to write source code for a client that works in a web browser. You can use a simple editor tool such as Notepad to write the HTML source code.

After writing the HTML source code, you can save it with a name of your choice, as shown below. In this case, we'll save it as chatclient.html.

```
<html>
  <head>
    <title>WebSocket Chatting Client</title>
  </head>
  <body>
    <h1>WebSocket Chatting Client</h1>
    <p>
      <button onClick="initWebSocket();">Server Connection</button>
    </p>
  </body>
</html>
```

```
<button onClick="stopWebSocket();">Disconnect</button>
<button onClick="checkSocket();">Status</button>
</p>
<p>
    <textarea id="debugTextArea"
              style="width:400px;height:100px;">
    </textarea>
</p>
<p>
    <input type="text" id="inputNick" value="nickname" />
    <input type="text" id="inputText"
           onkeydown="if(event.keyCode==13)sendMessage();"/>

    <button onClick="sendMessage();">Send</button>
</p>

<script type="text/javascript">
    var debugTextArea = document.getElementById("debugTextArea");
    function debug(message) {
        debugTextArea.value += message + "\n";
        debugTextArea.scrollTop = debugTextArea.scrollHeight;
    }

    function sendMessage() {
        var nickname = document.getElementById("inputNick").value;
        var msg = document.getElementById("inputText").value;
        var strToSend = nickname + ": " + msg;
        if ( websocket != null )
        {
            document.getElementById("inputText").value = "";
            websocket.send( strToSend );
            console.log( "string sent : ", ''' +strToSend+ ''' );
            debug(strToSend);
        }
    }

    var wsUri = "ws://localhost:1234";
    var websocket = null;

    function initWebSocket() {
        try {
            if (typeof MozWebSocket == 'function')

```

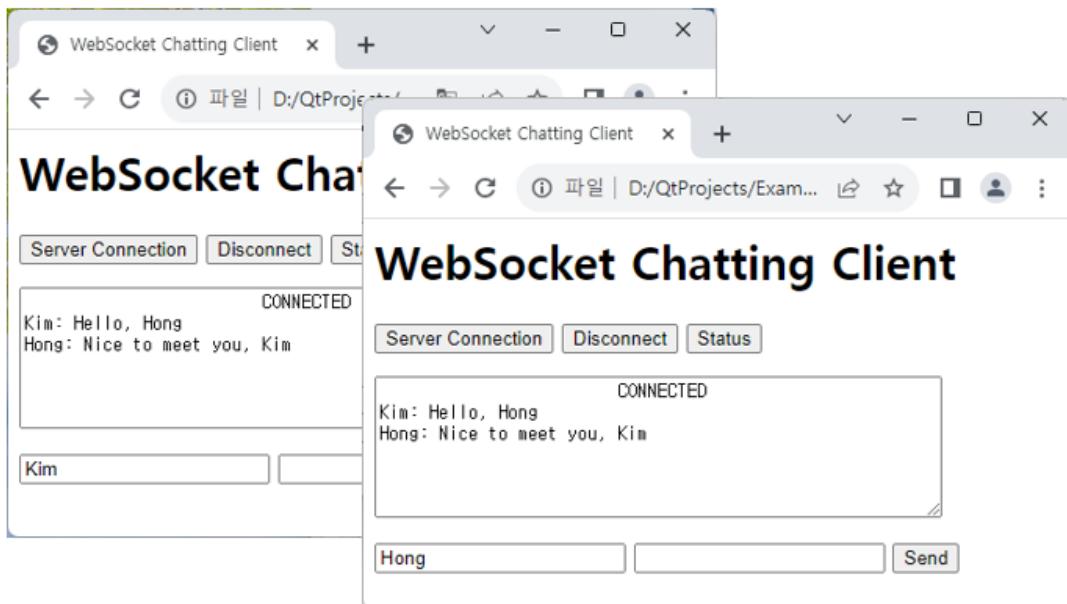
```
WebSocket = MozWebSocket;
if ( websocket && websocket.readyState == 1 )
    websocket.close();
websocket = new WebSocket( wsUri );
websocket.onopen = function (evt) {
    debug("CONNECTED");
};
websocket.onclose = function (evt) {
    debug("DISCONNECTED");
};
websocket.onmessage = function (evt) {
    console.log( "Message received :", evt.data );
    debug( evt.data );
};
websocket.onerror = function (evt) {
    debug('ERROR: ' + evt.data);
};
} catch (exception) {
    debug('ERROR: ' + exception);
}
}

function stopWebSocket() {
    if (websocket)
        websocket.close();
}

function checkSocket() {
    if (websocket != null) {
        var stateStr;
        switch (websocket.readyState) {
            case 0: {
                stateStr = "CONNECTING";
                break;
            }
            case 1: {
                stateStr = "OPEN";
                break;
            }
            case 2: {
                stateStr = "CLOSING";
                break;
            }
        }
    }
}
```

```
        }
        case 3: {
            stateStr = "CLOSED";
            break;
        }
        default: {
            stateStr = "UNKNOW";
            break;
        }
    }
    debug("WebSocket state = " +
        websocket.readyState + " (" +
        stateStr + ")");
} else {
    debug("WebSocket is null");
}
}
</script>
</body>
</html>
```

After completing the above, run the HTML code in two web browsers as shown in the figure below. Then, click the [Connect to Server] button, enter a message, and click the [Send] button.

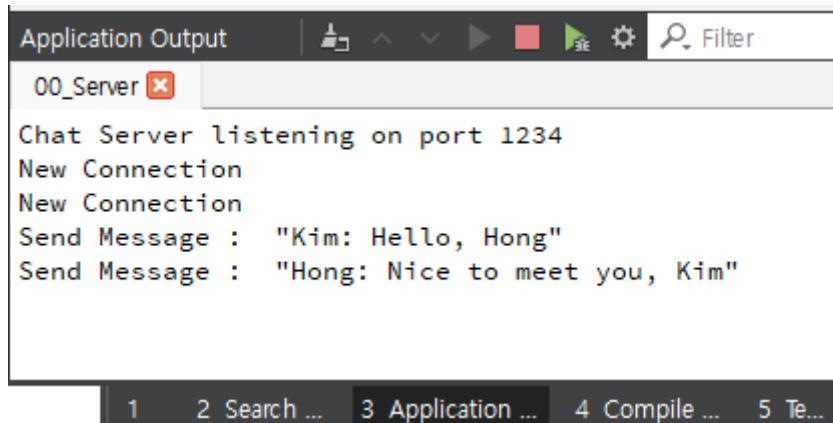


As shown in the figure above, click the [Server Connection] button to try to connect to

Jesus loves you.

the server. If the connection is successfully completed, the message CONNECTED will be displayed in the message window. Enter the message below and click the [Send] button to send the message to the server. The server then sends the received message back to the connected client.

You can check the server's log of the messages sent by the client in the [Application Output] window of Qt Creator as shown below.



The screenshot shows the 'Application Output' window in Qt Creator. The title bar says 'Application Output'. Below it, a tab labeled '00\_Server' is selected. The main area displays the following text:

```
Chat Server listening on port 1234
New Connection
New Connection
Send Message : "Kim: Hello, Hong"
Send Message : "Hong: Nice to meet you, Kim"
```

At the bottom of the window, there is a toolbar with icons for file operations and a status bar showing tabs 1 through 5.

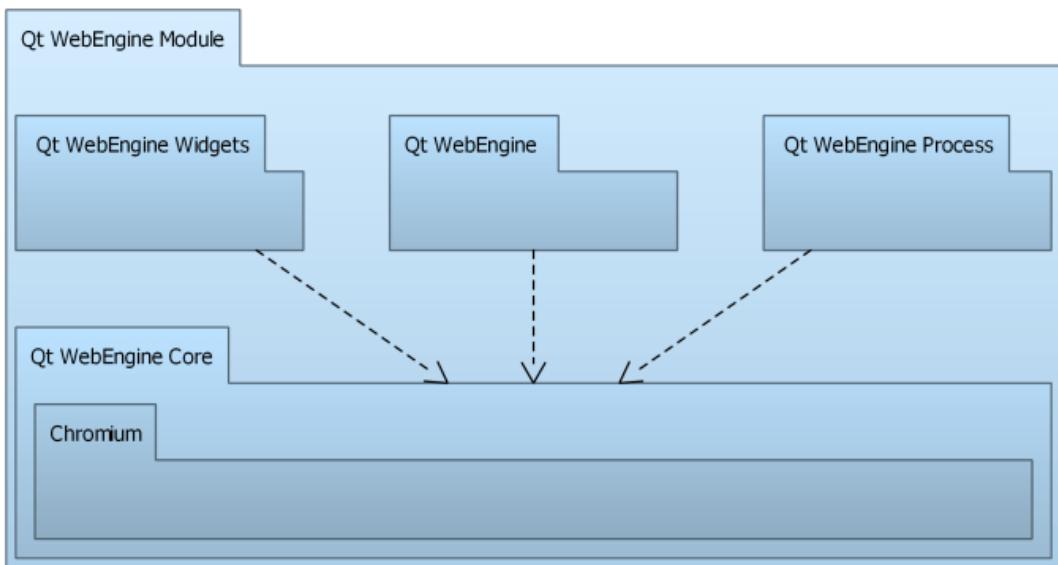
The server example source code can be found in the 00\_Server directory. And the HTML source code can be found in 00\_HTML\_Source.

## 30. Qt WebEngine

The Qt WebEngine module provides the ability to easily create web browser-based applications.

The Qt WebEngine module is available for C++ and QML types. Its main features include the ability to render HTML, XHTML, and SVG documents. It can also use Cascading Style Sheets (CSS) and control JavaScript and HTML documents if they are available.

The architecture of the Qt WebEngine module is categorized into three modules.



### ✓ Qt WebEngine Widget

The Qt WebEngine provides web application-based widgets, such as QWidget, that allow you to create widget-like areas where web content is rendered. However, this module provides widgets for use in C++.

### ✓ Qt WebEngine

This module provides the same functionality as the Qt WebEngine Widgets module, but is available in QML.

### ✓ Qt WebEngine Process

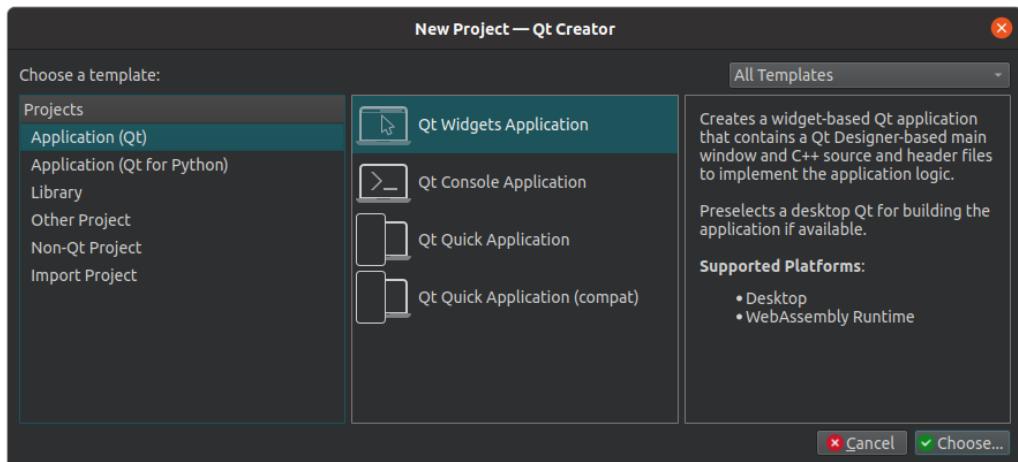
This module is implemented based on the Chromium Project. Chromium provides its own network and Painting Engine. Therefore, this module provides functionality for communicating with the Core provided by Chromium.

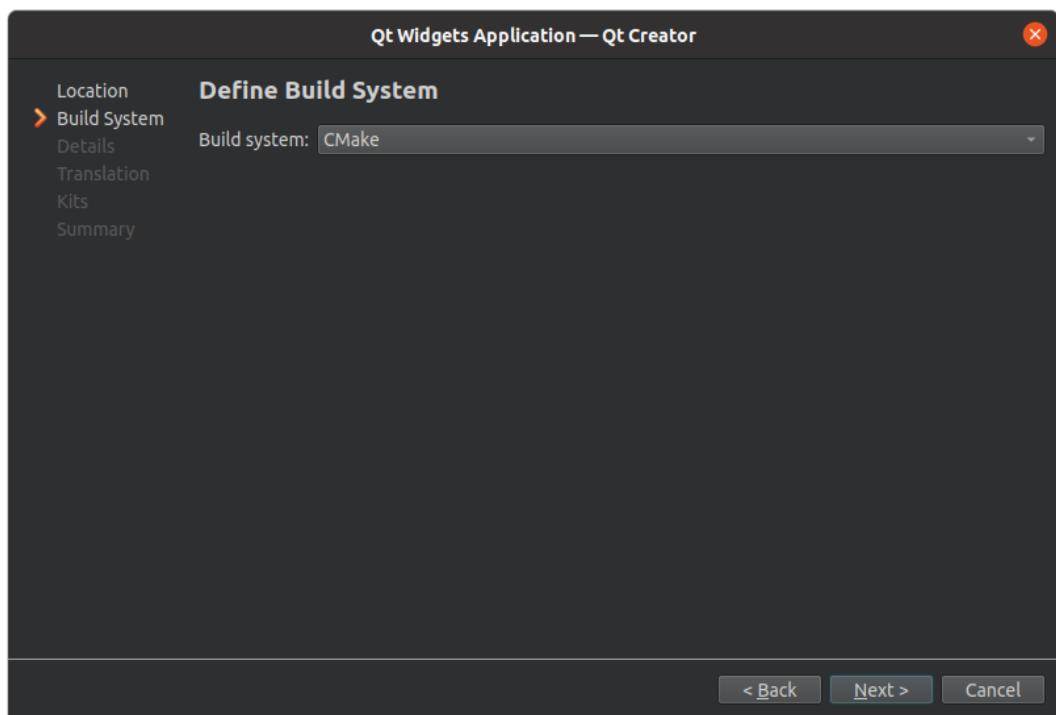
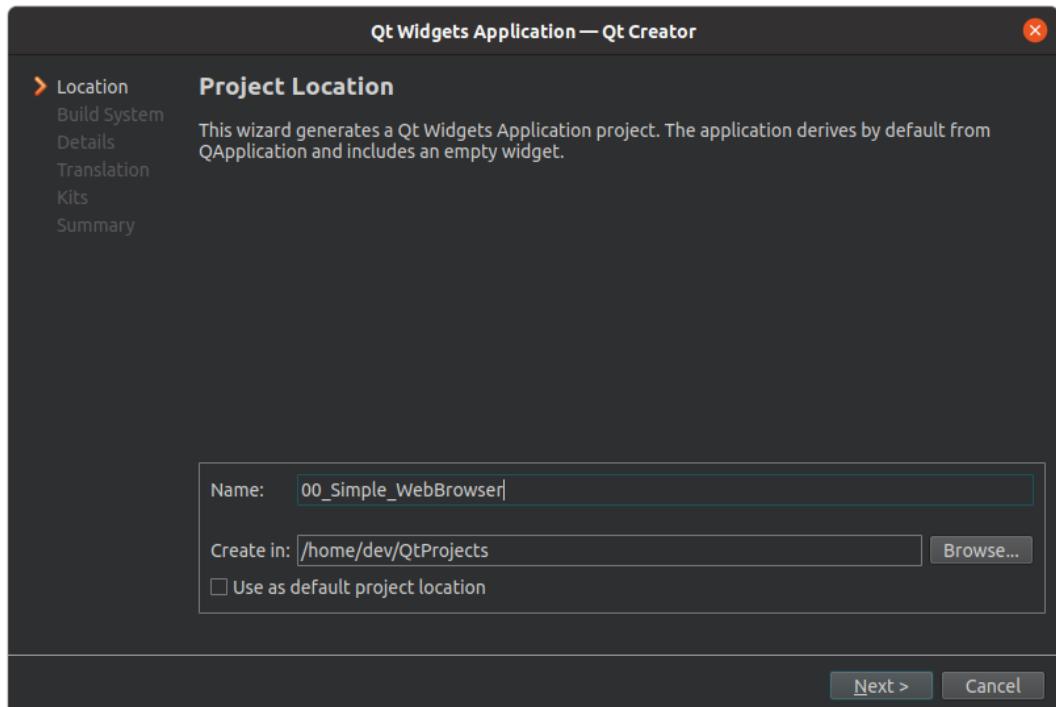
In this chapter, we will cover a simple web browser example using the Qt WebEngine module.

It is important to note that the MinGW compiler is not supported on MS Windows when using the Qt WebEngine module. When using the Qt WebEngine module on MS Windows, you must use the MSVC compiler.

### ✓ Implement a simple Web Browser example

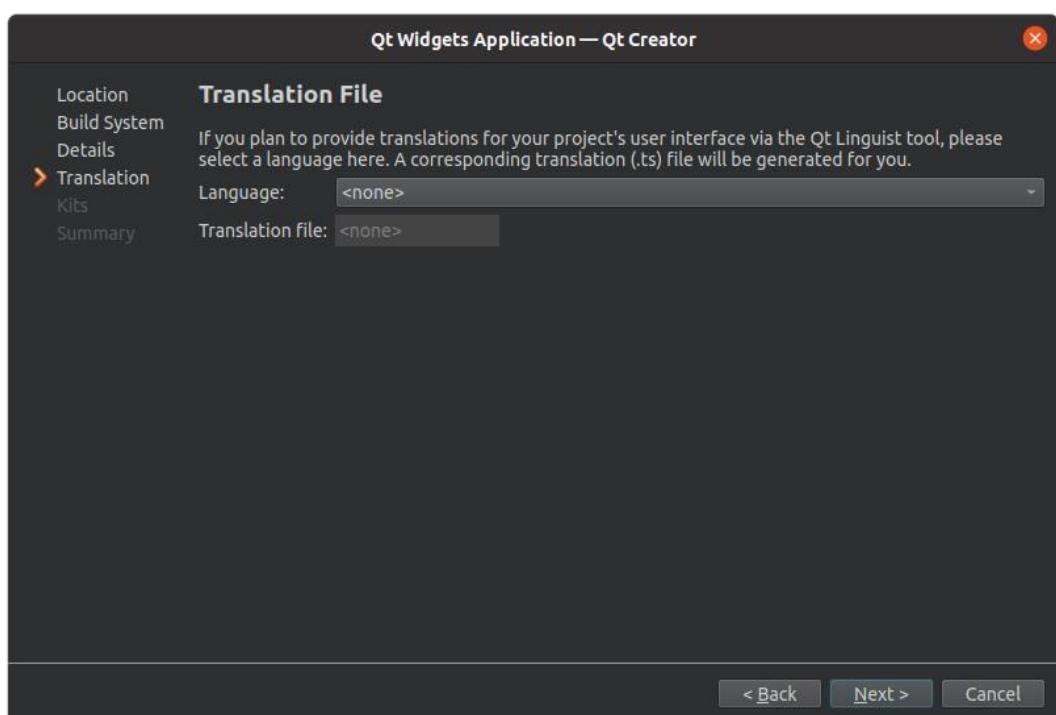
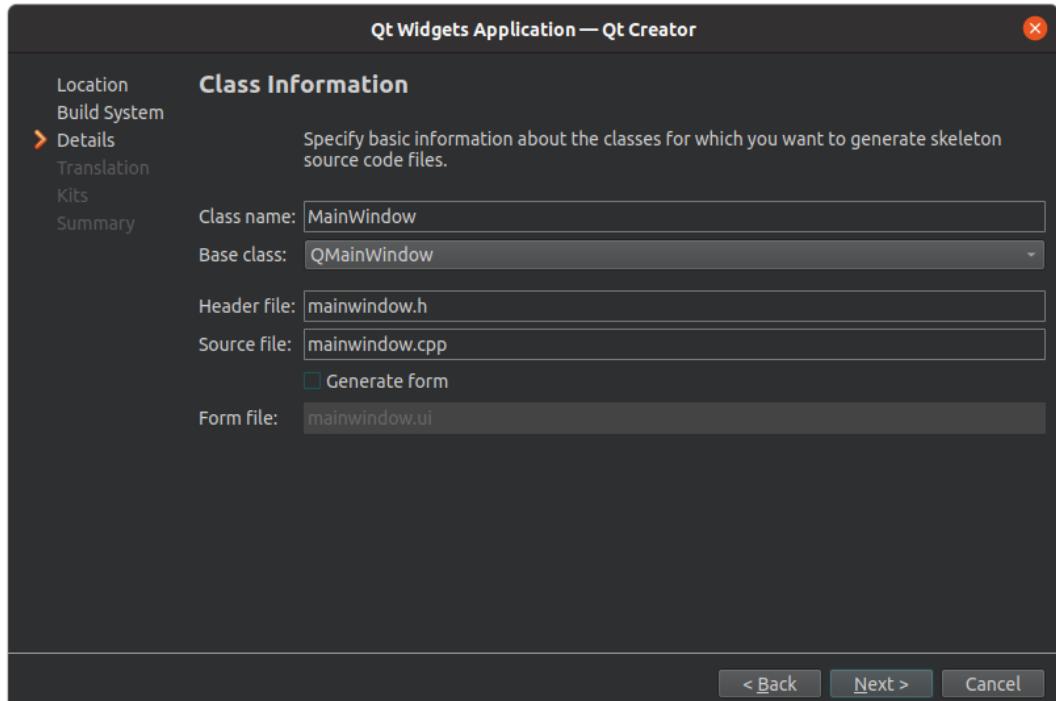
When creating a project, select and create a project based on Qt Widgets.



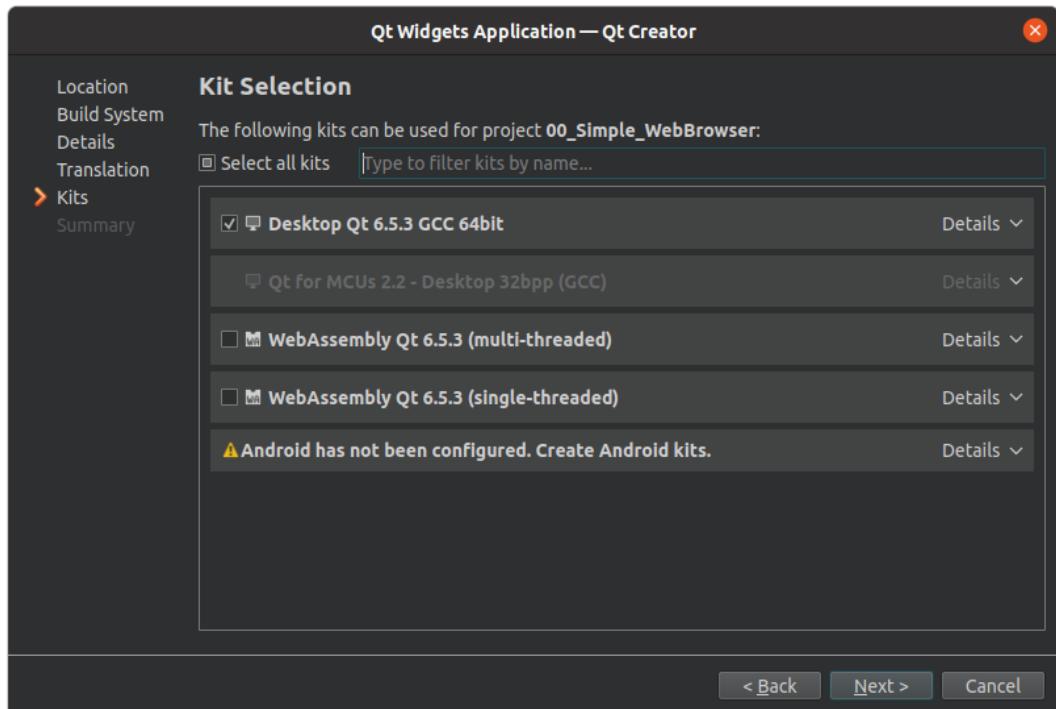


Jesus loves you.

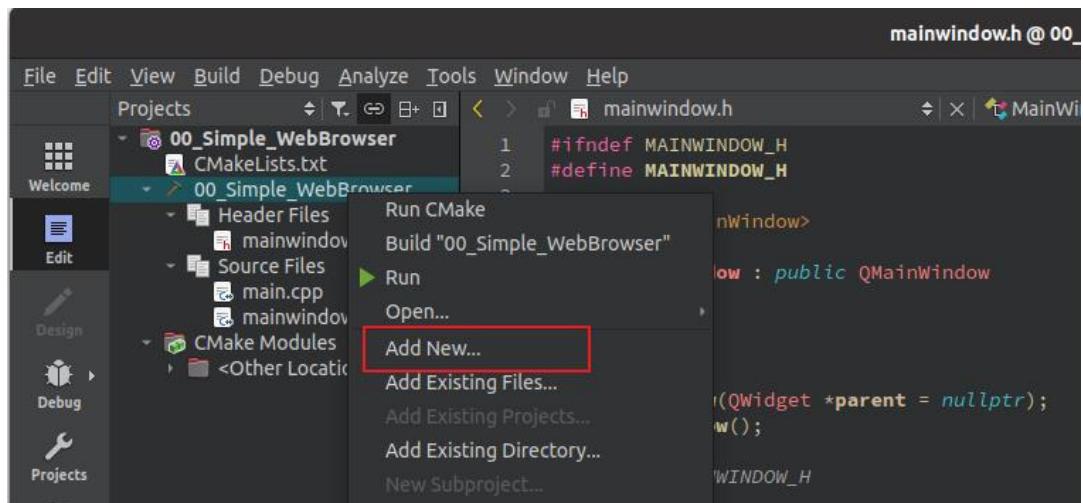
In the Class Information dialog, create the MainWindow class as shown below. And uncheck the [Generate form] item.



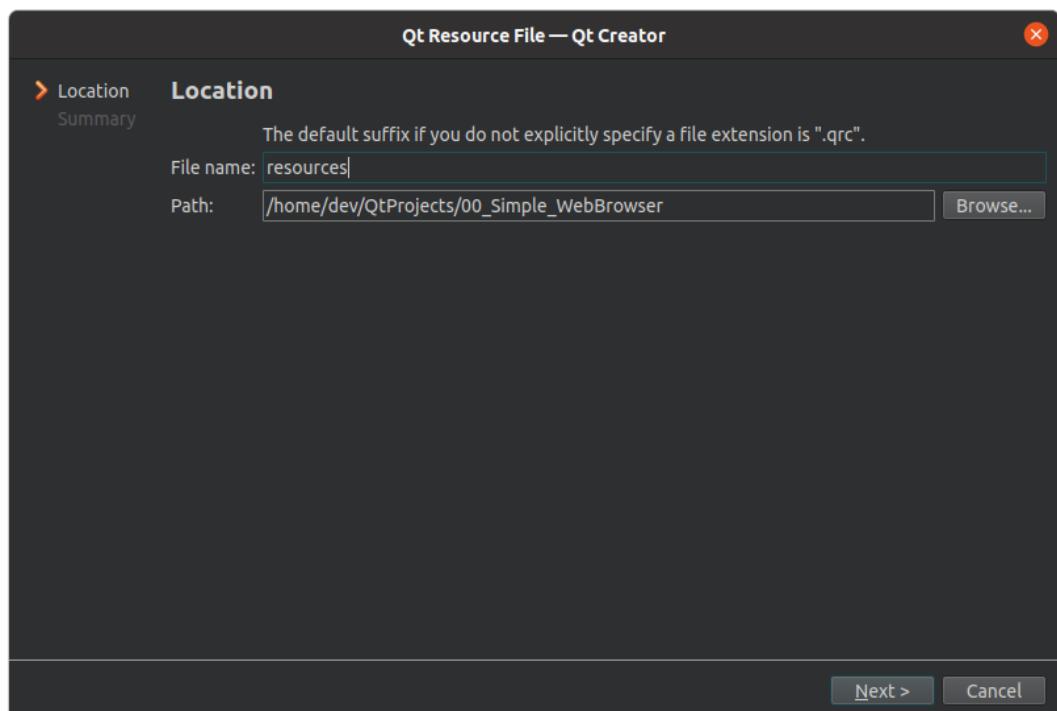
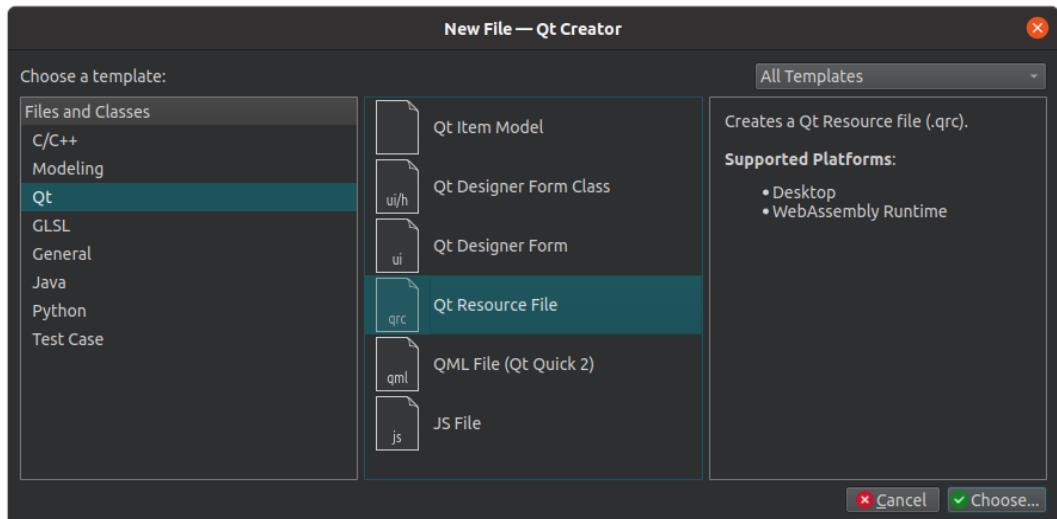
Jesus loves you.



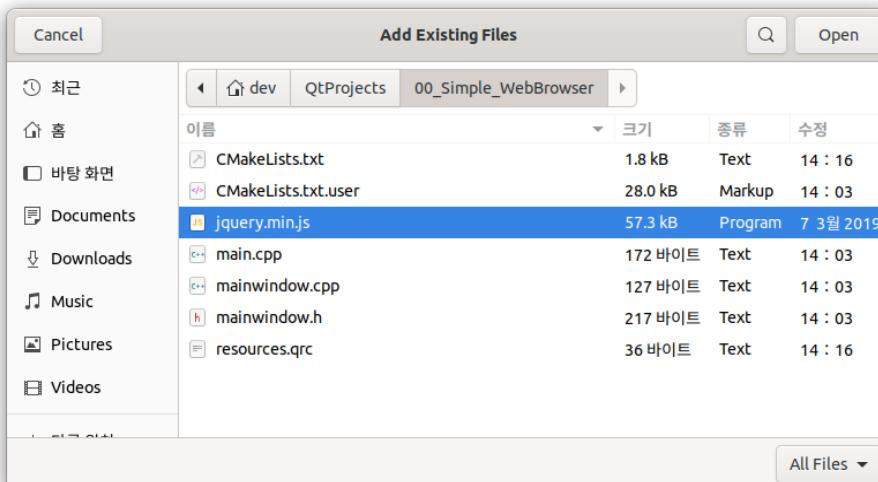
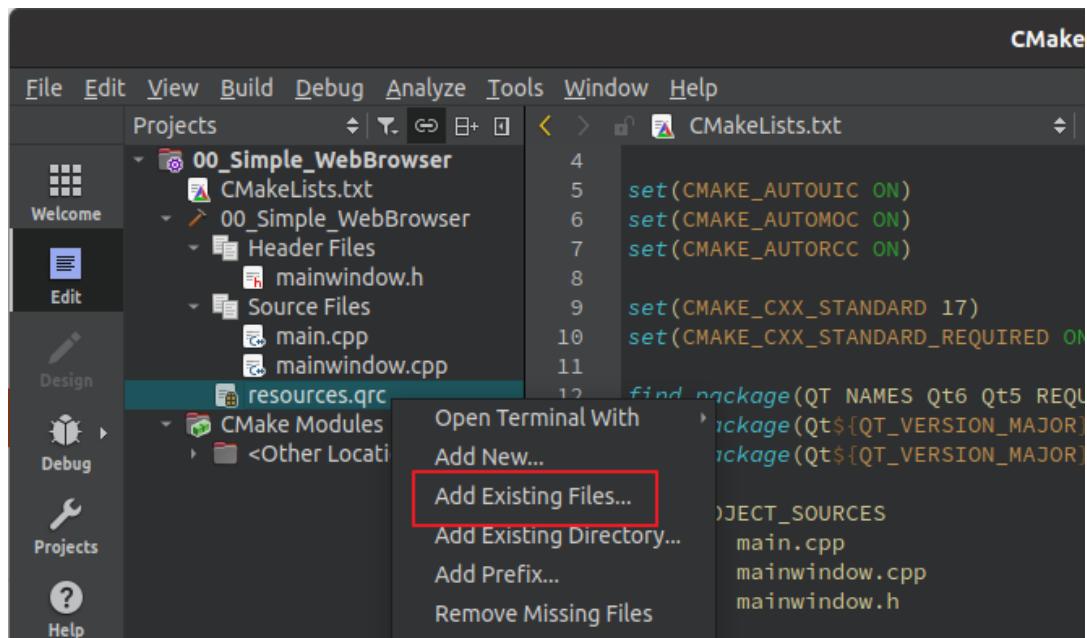
Once you have created your project as above, add Qt Resources to your project.



Jesus loves you.



Once you have added the resource file, add the jquery.min.js file to the resource. You can find this file in the example source code directory. So use the file in this directory.



Next, in order to use the Qt WebEngine module in your project, you need to add the following to your CMakeList.txt file

```
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS WebEngineWidgets)
target_link_libraries(00_Simple_WebBrowser PRIVATE Qt6::WebEngineWidgets)
```

If you're using qmake, you'll need to add the following

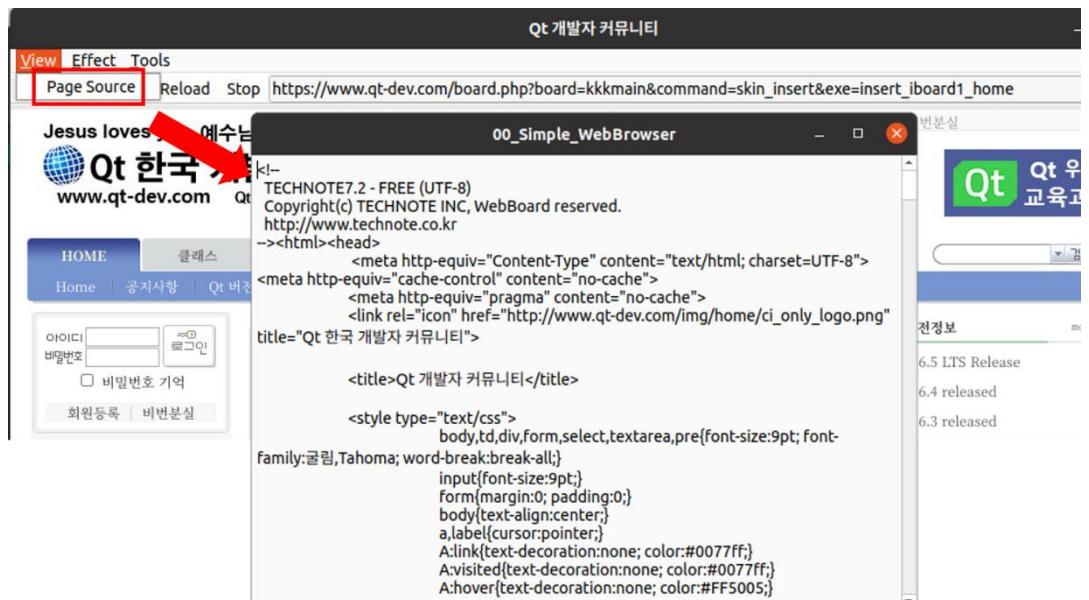
```
QT += webenginewidgets
```

We'll be using CMake, so you can write the following in your CMakeList.txt file.

Before we write any source code, let's take a look at the running screen of this example to see what it does.

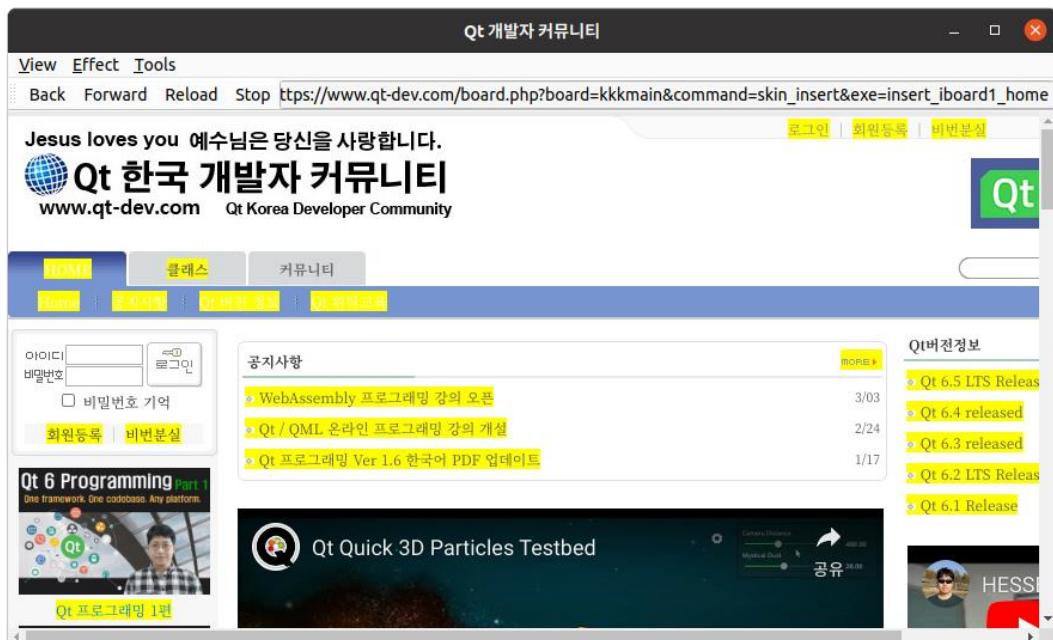


As you can see in the image above, the example loads the front page of the Qt Developer Community site. The first View menu in the menu is the [Page Source] menu. Clicking this menu will load a window where you can view the source code of the current web page.



Next, select [Effect] -> [Highlight Links] from the menu, and all the links will change to a

yellow background, as shown in the image below.



In the Tools menu, click [Remove GIF images] to remove all GIF images used on the webpage from the HTML tags.

In the toolbar menu at the bottom of the menu, the first [Back] will take you back to the previous page.

The second [Forward] icon will take you to the next page, and the third [Reload] will refresh the current page. The fourth [Stop] stops the current loading process. And the address bar on the right is the current URL address bar.

Let's implement the functions described so far. First, open the mainwindow.h header file and write the following source code.

```
#ifndef MAINWINDOW_H
#define MAINWINDOW_H

#include <QMainWindow>
#include <QWebEngineView>
#include <QLineEdit>

class MainWindow : public QMainWindow
{
    Q_OBJECT
```

```
public:  
    MainWindow(const QUrl& url,  
               QWidget *parent = nullptr);  
    ~MainWindow();  
  
protected slots:  
  
    void adjustLocation();  
    void changeLocation();  
    void adjustTitle();  
    void setProgress(int p);  
    void finishLoading(bool);  
  
    void viewSource();  
  
    void highlightAllLinks();  
    void removeGifImages();  
  
private:  
    QString jQuery;  
    QWebEngineView *view;  
    QLineEdit *locationEdit;  
    int progress;  
};  
#endif // MAINWINDOW_H
```

The `adjustLocation()` Slot function provides the ability to display the current web page address in the address bar. When the web page is finished loading, the `loadFinished()` Signal of class `QWebEngineView` is fired. When this signal is raised, the `adjustLocation()` Slot function is called.

The `changeLocation()` Slot function gets the address of the address window when the user clicks the Enter key in the address window, and requests the web page to the web server entered in the address window to render the page.

The `adjustTitle()` Slot function displays the string from the web page's `<title>` tag in the `QMainWindow` widget title bar. The `finishLoading()` function is called when the web page is finished loading.

The `viewSource()` Slot function is called when you click [Page Source] in the View menu.

The `highlightAllLinks( )` Slot function is called when you click the [Highlight all links] button in the Effect menu.

And the `removeGifImages( )` function is called when you click [Remove GIF images] from the Tools menu.

In this project, we used jQuery. jQuery is an open-source JavaScript that makes JavaScript easier to use. Next, write the `mainwindow.cpp` source code as shown below.

```
#include "mainwindow.h"
#include <QFile>
#include <QToolBar>
#include <QMenu>
#include <QMenuBar>
#include <QAction>
#include <QTextEdit>

MainWindow::MainWindow(const QUrl& url, QWidget *parent)
    : QMainWindow(parent)
{
    setAttribute(Qt::WA_DeleteOnClose, true);
    progress = 0;

    QFile file;
    file.setFileName(":/jquery.min.js");
    file.open(QIODevice::ReadOnly);
    jjQuery = file.readAll();
    jjQuery.append("\nvar qt = { 'jQuery': jQuery.noConflict(true) };");
    file.close();

    view = new QWebEngineView(this);
    view->load(url);
    connect(view, &QWebEngineView::loadFinished,
            this, &MainWindow::adjustLocation);
    connect(view, &QWebEngineView::titleChanged,
            this, &MainWindow::adjustTitle);
    connect(view, &QWebEngineView::loadProgress,
            this, &MainWindow::setProgress);
    connect(view, &QWebEngineView::loadFinished,
            this, &MainWindow::finishLoading);

    locationEdit = new QLineEdit(this);
    locationEdit->setSizePolicy(QSizePolicy::Expanding,
```

```
        locationEdit->sizePolicy().verticalPolicy());  
connect(locationEdit, &QLineEdit::returnPressed,  
       this, &MainWindow::changeLocation);  
  
QToolBar *toolBar = addToolBar(tr("Navigation"));  
toolBar->addAction(view->pageAction(QWebEnginePage::Back));  
toolBar->addAction(view->pageAction(QWebEnginePage::Forward));  
toolBar->addAction(view->pageAction(QWebEnginePage::Reload));  
toolBar->addAction(view->pageAction(QWebEnginePage::Stop));  
toolBar->addWidget(locationEdit);  
  
QMenu *viewMenu = menuBar()->addMenu(tr("&View"));  
 QAction *viewSourceAction = new QAction(tr("Page Source"), this);  
connect(viewSourceAction, &QAction::triggered,  
       this, &MainWindow::viewSource);  
viewMenu->addAction(viewSourceAction);  
  
QMenu *effectMenu = menuBar()->addMenu(tr("&Effect"));  
effectMenu->addAction(tr("Highlight all links"),  
                      this, &MainWindow::highlightAllLinks);  
  
QMenu *toolsMenu = menuBar()->addMenu(tr("&Tools"));  
toolsMenu->addAction(tr("Remove GIF images"),  
                      this, &MainWindow::removeGifImages);  
  
setCentralWidget(view);  
}  
  
void MainWindow::viewSource()  
{  
    QTextEdit *TextEdit = new QTextEdit(nullptr);  
    TextEdit->setAttribute(Qt::WA_DeleteOnClose);  
    TextEdit->adjustSize();  
    TextEdit->move(this->geometry().center() - TextEdit->rect().center());  
    TextEdit->show();  
  
    view->page()->toHtml([TextEdit](const QString &html){  
        TextEdit->setPlainText(html);  
    });  
}  
  
void MainWindow::adjustLocation()
```

```
{  
    locationEdit->setText(view->url().toString());  
}  
  
void MainWindow::changeLocation()  
{  
    QUrl url = QUrl::fromUserInput(locationEdit->text());  
    view->load(url);  
    view->setFocus();  
}  
  
void MainWindow::adjustTitle()  
{  
    if (progress <= 0 || progress >= 100)  
        setWindowTitle(view->title());  
    else  
        setWindowTitle(QStringLiteral("%1 (%2%)")  
                      .arg(view->title()).arg(progress));  
}  
  
void MainWindow::setProgress(int p)  
{  
    progress = p;  
    adjustTitle();  
}  
  
void MainWindow::finishLoading(bool)  
{  
    progress = 100;  
    adjustTitle();  
    view->page()->runJavaScript(jQuery);  
}  
  
void MainWindow::highlightAllLinks()  
{  
    QString code = QStringLiteral("qt.jQuery('a').each( function () "  
                                "{ qt.jQuery(this).css('background-color', "  
                                              'yellow') } )");  
  
    view->page()->runJavaScript(code);  
}
```

```
void MainWindow::removeGifImages()
{
    QString code = QStringLiteral("qt.jQuery('[src*=gif]').remove()");
    view->page()->runJavaScript(code);
}

MainWindow::~MainWindow()
```

The example source code we've written so far refers to the 00\_Simple\_WebBrowser directory.

## 31. Qt HTTP Server

The Qt HTTP Server module provided by Qt provides functionality that makes it easy to implement a Web server.

For example, you can provide services to clients, such as web browsers or other applications, through an HTTP-based REST API.

In my opinion, the main reason to use this module is that it allows you to easily implement a lightweight Web server. It also makes it easy to implement SSL-based services.

To use this module, you need to add the following to your project file. If you're using CMake, you'll need to add it like this

```
find_package(Qt6 REQUIRED COMPONENTS HttpServer)
target_link_libraries(mytarget PRIVATE Qt6::HttpServer)
```

If you're using qmake, you'll need to add the following

```
QT += httpserver
```

The Qt HTTP Server module provides the QHttpServer class to implement a Web server. This class makes it easy to implement a Web server.

First, we need to use the listen( ) member function to listen for requests from the web browser (client), as shown below.

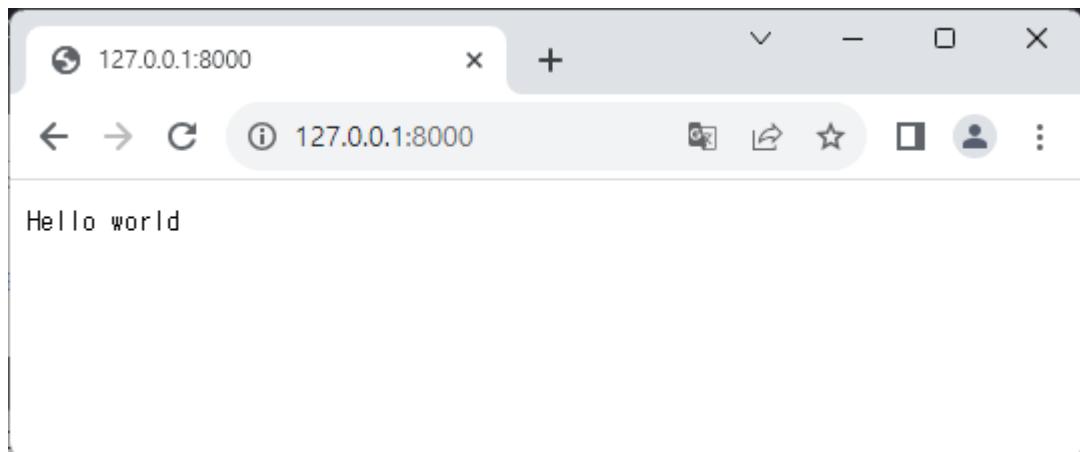
```
QHttpServer m_server;
m_server.listen(QHostAddress::Any, 8000);
```

The first argument is an IP address. You can use an IP address directly in the first argument. However, QHostAddress::Any can use any IP on the system. The second argument is a port number. So a web browser should use `http://[IP Address]:[Port]` when making a request to a web server.

For example, when a Web browser makes a request to a Web server, it receives "Hello world" from the Web server, as shown in the figure below, when the request is made to

Jesus loves you.

the address <http://127.0.0.1:8000/>.

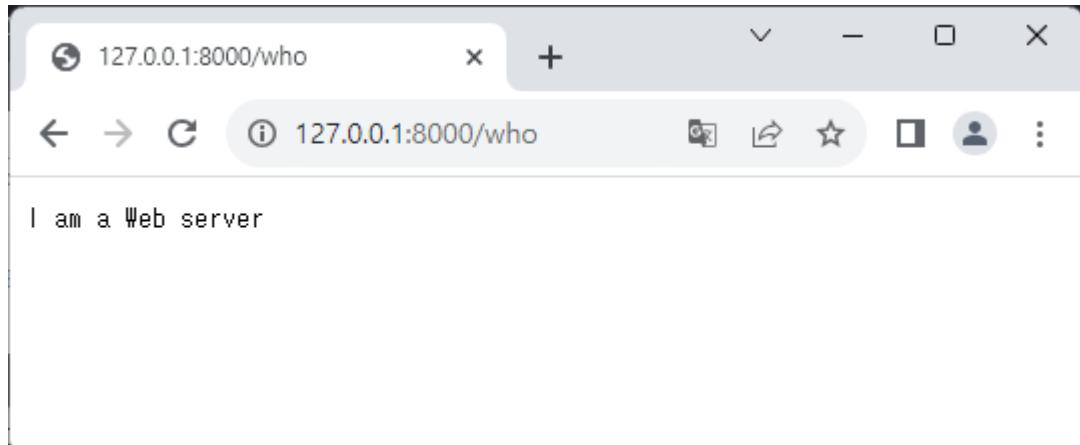


Next, we can use the `route( )` function to get the "Hello world" response from the web server, as shown in the image above.

```
m_server.route("/", [] () {
    return "Hello world";
});
```

If you want to use a specific filter, you can use the `Filter` option in the first argument of the `route( )` function.

```
// [ URL ] http://127.0.0.1:8000/who
m_server.route("/who", [] () {
    return "I am a boy";
});
```



To respond with a JSON type, you can use the following methods

```
// [ URL ] http://127.0.0.1:8000/hello_json
m_server.route("/hello_json", [] () {
    QJsonObject jsonObj;
    jsonObj[ "key1" ] = "value1";
    jsonObj[ "key2" ] = "value2";
    jsonObj[ "key3" ] = "value3";

    return jsonObj;
});
```



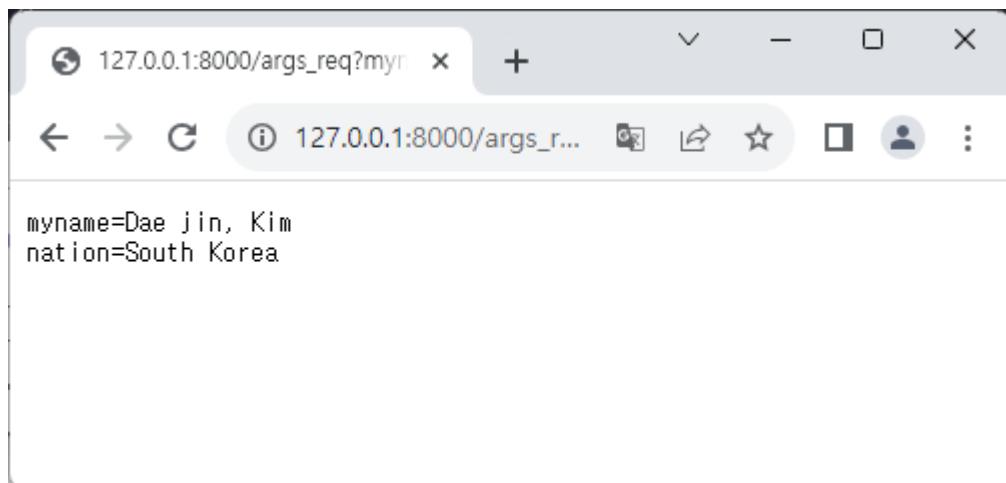
In order for a web browser to pass multiple values to a web server, it can pass the values in the following ways

```
http://127.0.0.1:8000/args_req?myname=Dae jin, Kim&nation=South Korea
```

As shown in the example above, you can use the following method to pass the values myname and nation to the web server.

```
m_server.route("/args_req", [] (const QHttpServerRequest &req) {
    QString filter;
    QTextStream result(&filter);
    for (auto pair : req.query().queryItems()) {
        if (!filter.isEmpty())
            result << "\n";
```

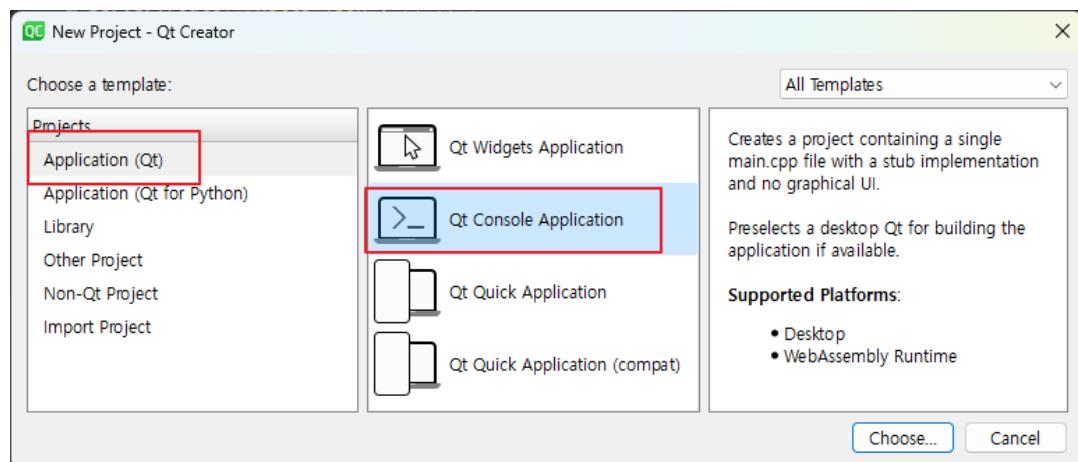
```
    result << pair.first << "=" << pair.second;
}
return filter;
});
```



So far, we've seen how to implement a Web server using the QHttpServer class. Next, let's see how to implement a simple Web server with an example.

- ✓ Simple Web server implementation example

When creating a project, select an application based on the Qt Console.



After creating the project, add the following to your project file to enable the HttpServer module.

```
cmake_minimum_required(VERSION 3.14)

project(00_WebServer LANGUAGES CXX)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Qt NAMES Qt6 Qt5 REQUIRED COMPONENTS Core)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Core HttpServer)

add_executable(00_WebServer
    main.cpp
)

target_link_libraries(00_WebServer Qt${QT_VERSION_MAJOR}::Core)
target_link_libraries(00_WebServer Qt${QT_VERSION_MAJOR}::HttpServer)

include(GNUInstallDirs)
install(TARGETS 00_WebServer
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

Next, add the WebServer class, and write the WebServer.h header file like this

```
#ifndef WEBSERVER_H
#define WEBSERVER_H

#include <QObject>
#include <QtHttpServer/QHttpServer>

class WebServer : public QObject
{
```

```
Q_OBJECT
public:
    explicit WebServer(QObject *parent = nullptr);
    ~WebServer();

private:
    QHttpServer m_server;
    QString hostName(const QHttpRequest &request);

};

#endif // WEBSERVER_H
```

Next, write the WebServer.cpp source code.

```
#include "WebServer.h"
#include <qloggingcategory.h>
#include <QJsonObject>
#include <QDebug>

using namespace Qt::StringLiterals;

WebServer::WebServer(QObject *parent)
    : QObject{parent}
{
    // [ URL ] http://127.0.0.1:8000
    m_server.route("/", [] () {
        return "Hello world";
    });

    // [ URL ] http://127.0.0.1:8000/who/
    m_server.route("/who", [] () {
        return "I am a Web server";
    });

    // [ URL ] http://127.0.0.1:8000/hello/2/
```

```
m_server.route("/hello/<arg>/", [=] (qint32 id, const QHttpServerRequest &req) {
    QString hostStr = hostName(req) + u"/hello id ---> %1"_s.arg(id);
    return hostStr;
});

// [ URL ] http://127.0.0.1:8000/hello_json
m_server.route("/hello_json", [] () {
    QJsonObject jsonObj;
    jsonObj["key1"] = "value1";
    jsonObj["key2"] = "value2";
    jsonObj["key3"] = "value3";

    return jsonObj;
});

//http://127.0.0.1:8000/args_req?myname=Dae jin, Kim&nation=South Korea
m_server.route("/args_req", [] (const QHttpServerRequest &req) {
    QString filter;
    QTextStream result(&filter);
    for (auto pair : req.query().queryItems()) {
        if (!filter.isEmpty())
            result << "\n";
        result << pair.first << "=" << pair.second;
    }
    QLoggingCategory::setFilterRules(filter);

    return filter;
});

//m_server.listen(QHostAddress::LocalHost, 8000);
m_server.listen(QHostAddress::Any, 8000);
}

QString WebServer::hostName(const QHttpServerRequest &request)
```

```
{
    return QString::fromLatin1(request.value("Host"));
}

WebServer::~WebServer()
{
    deleteLater();
}
```

As shown in the example source code above, the route( ) function of the QHttpServer class handles requests from a web browser (client). And the listen( ) function specifies the IP and Port to listen for requests from the web browser.

So far, we've seen how to implement a simple Web server. You can refer to the 00\_WebServer directory for the complete example source code.

- ✓ Example of a Web server that supports SSL

QHttpServer 클래스는 SSL(Secure Socket Layer) 을 지원한다. SSL을 지원하는 Web server를 구현하기 위해서는 Key 와 CSR 가 필요하다. 이 2개의 파일을 생성하기 위해서 여러가지 방법이 있으나 여기서는 OpenSSL을 이용하는 방법에 대해서 살펴보도록 하자.

```
# openssl genrsa -out test.key 2048
```

위와 같이 Key 생성 한다. 만약 Public Key를 사용한다면 아래와 같이 사용하면 된다.

```
# openssl rsa -in test.key -pubout -out test_pub.key
```

다음은 CSR 을 생성하는 방법이다.

```
# openssl req -new -key test.key -out test.csr
```

다음은 CRT를 생성하는 방법이다.

```
# openssl x509 -req -days 365 -in test.csr -signkey test.key -out test.crt
```

Once you have generated the private key and CRT as shown above, you will need to use these two files to implement a web server that supports SSL.

Next, create a Qt Console-based project. Then add the HttpServer module to CMake.

Jesus loves you.

Then add the two SSL files you created to the Resource. After adding the resource, add it to the project. Add the SecureWebServer class to the project. Below is the SecureWebServer.h header file.

```
#ifndef SECUREWEBSERVER_H
#define SECUREWEBSERVER_H

#include <QObject>
#include <QtHttpServer/QHttpServer>

class SecureWebServer : public QObject
{
    Q_OBJECT
public:
    explicit SecureWebServer(QObject *parent = nullptr);
    ~SecureWebServer();

private:
    QHttpServer m_server;

};

#endif // SECUREWEBSERVER_H
```

The example source code below is the SecureWebServer.cpp source code.

```
#include "SecureWebServer.h"
#include <QCoreApplication>
#include <QFile>
#include <QDebug>

#if QT_CONFIG(ssl)
# include <QSslCertificate>
# include <QSslKey>
#endif

SecureWebServer::SecureWebServer(QObject *parent)
```

```
: QObject{parent}

{
    m_server.route("/", [] () {
        return "hello world";
    });
}

#if QT_CONFIG(ssl)
    const auto sslCertificateChain =
        QSslCertificate::fromPath(QStringLiteral(":/test.crt"));

    QFile privateKeyFile(QStringLiteral(":/test.key"));
    if (!privateKeyFile.open(QIODevice::ReadOnly)) {
        qDebug() << "Couldn't open file for reading: %1"
            << privateKeyFile.errorString();
        return;
    }

    m_server.sslSetup(sslCertificateChain.front(),
                      QSslKey(&privateKeyFile, QSsl::Rsa));
    privateKeyFile.close();

    // [ URL ] https://127.0.0.1:8001
    const auto sslPort = m_server.listen(QHostAddress::Any, 8001);
    if (!sslPort)
        qDebug() << "Server failed to listen on a port.";

#else
    qDebug() << "This Web Server does not support SSL.";
#endif

}

SecureWebServer::~SecureWebServer()
{
    deleteLater();
}
```

Jesus loves you.

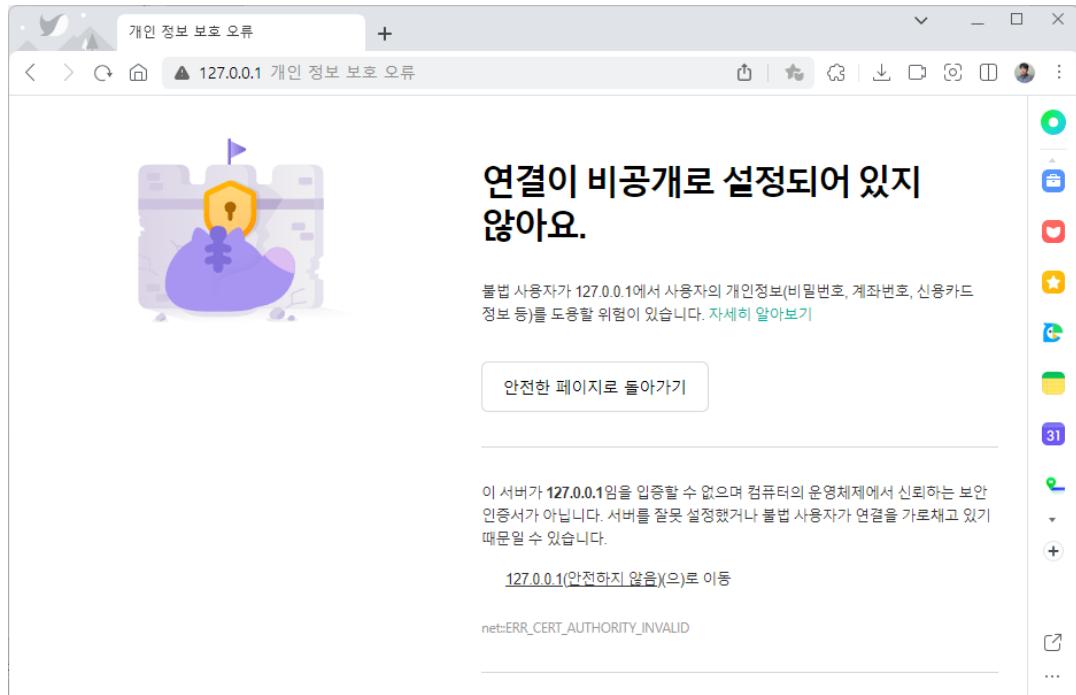
```
}
```

As shown in the example above, you can register the KEY using the `sslSetup()` member function provided by the `QHttpServer` class.

After writing the source code as above, let's build and run it, and then connect to the web server at the address below in a web browser.

```
https://127.0.0.1:8001
```

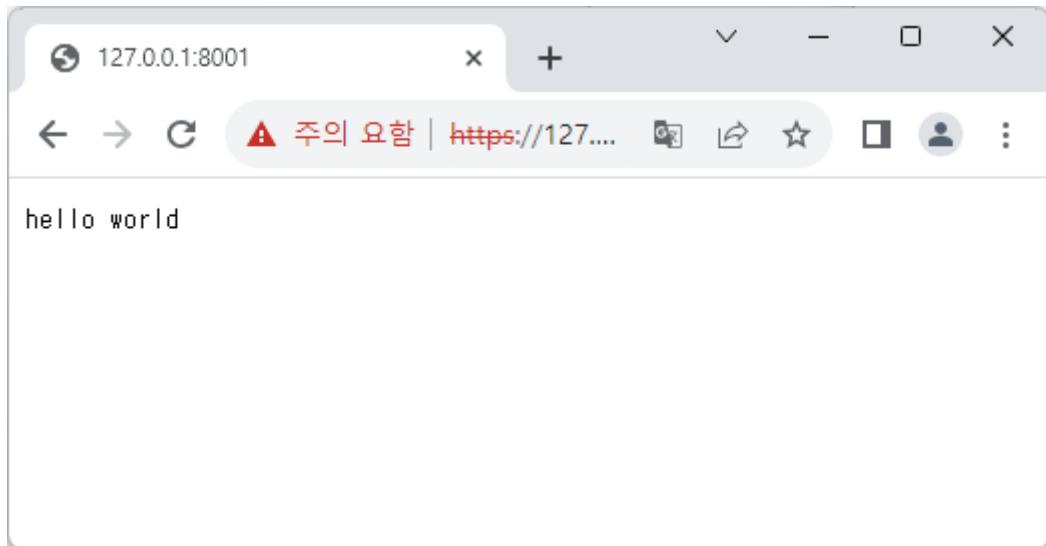
If you go to the address above, you should see a screen like the one below.



As you can see in the screen above, you can see that it's not safe. The reason this screen loads first is because the web browser thinks the URL is a hijacked site.

To fix this problem, you can get an SSL certificate for your domain address. Since we're using a temporary URL, click "Go to 127.0.0.1 (insecure)" at the bottom of the screen. You'll see a message from the web server like the one below.

Jesus loves you.



You can refer to the 01\_Secure\_WebServer directory for this example source code.

## 32. Deployment

In this chapter, you will learn how to distribute applications implemented with Qt to users. Qt provides the Qt Install Framework tool for deploying to users. This tool provides the ability to create an executable and the necessary libraries into an installation package that can be distributed to users in the same way that you would distribute an installation file.

This tool can be installed as an option when installing Qt. It can also be installed separately and supports MS Windows, Linux, and MacOS platforms. The usage is the same.

- ✓ Download and install the Qt Install Framework

[https://download.qt.io/official\\_releases/qt-installer-framework/](https://download.qt.io/official_releases/qt-installer-framework/)

You can download the Qt Install Framework by going to the URL above.

The screenshot shows a web browser window with the address bar containing "download.qt.io/official\_releases/qt-installer-framework/". The page title is "Qt Downloads". Below the title, there is a navigation menu with links to "Qt Home", "Bug Tracker", "Code Review", "Planet Qt", and "Get Qt Extensions". The main content area displays a table of file listings:

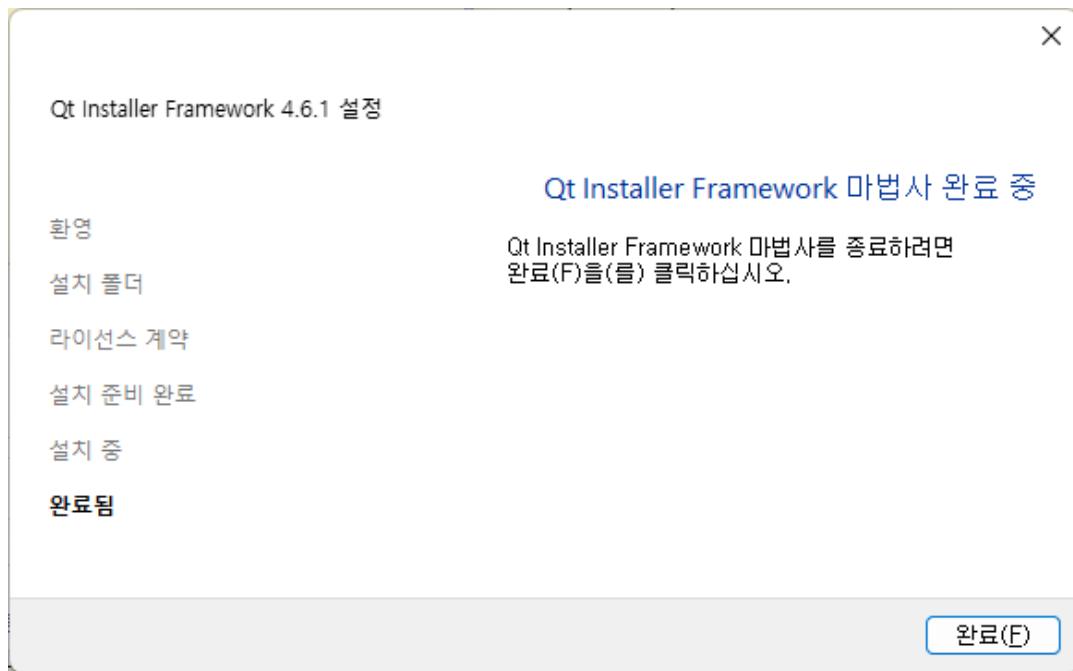
Name	Last modified	Size	Metadata
Parent Directory		-	
4.6.1/	29-Aug-2023 11:37	-	
4.6.0/	07-Jun-2023 12:53	-	
4.5.2/	13-Mar-2023 10:26	-	

Click the most recent version.

Name	Last modified	Size	Metadata
Parent Directory	-	-	-
md5sums.txt	29-Aug-2023 11:37	388	<a href="#">Details</a>
installer-framework-everywhere-src-4.6.1.zip	29-Aug-2023 11:35	5.9M	<a href="#">Details</a>
installer-framework-everywhere-src-4.6.1.tar.xz	29-Aug-2023 11:35	3.2M	<a href="#">Details</a>
QtInstallerFramework-windows-x64-4.6.1.exe	29-Aug-2023 11:34	66M	<a href="#">Details</a>
QtInstallerFramework-macOS-x64-4.6.1.dmg	29-Aug-2023 11:34	46M	<a href="#">Details</a>
QtInstallerFramework-linux-x64-4.6.1.run	29-Aug-2023 11:34	73M	<a href="#">Details</a>

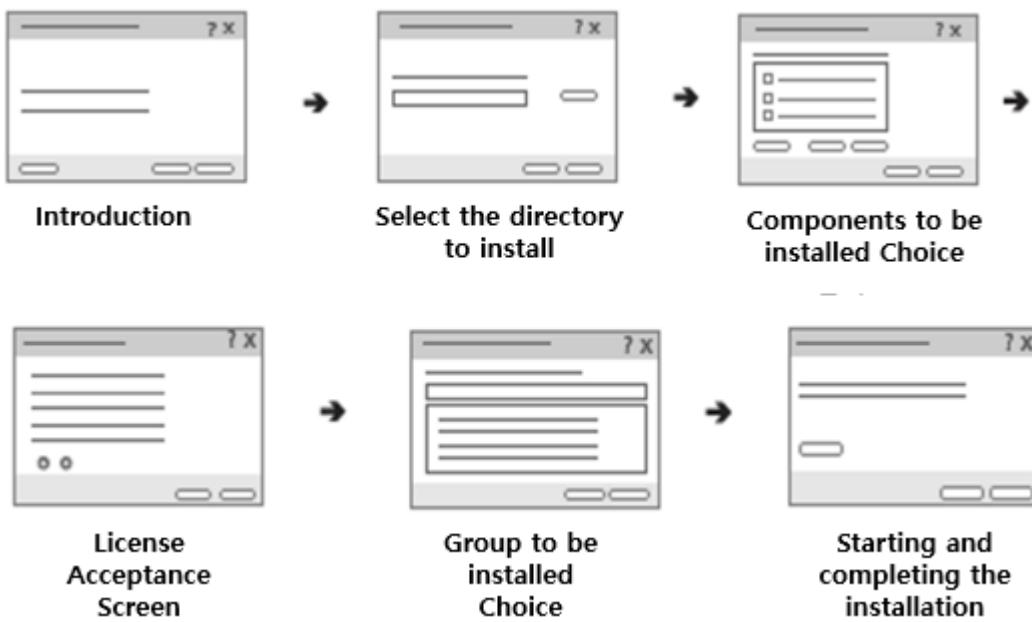
The "exe" extension is available for MS Windows platforms. And the "run" extension is for Linux and "dmg" is for MacOS platforms. In this example, we will download and install the "exe" file.





✓ How to Deploy the Qt Install Framework

Qt Install Framework 는 Online 방식과 Offline 방식을 제공한다. Online 방식은 특정 웹 서버 Repository에 접속하여 자동 다운로드 및 설치하는 방식이다. Offline 방식은 설치 파일을 다운로드 받아 설치하는 방식이다. 그리고 어플리케이션을 배포하기 위한 방법으로 다이얼로그 형태의 Setup Wizard 형식으로 배포판을 제작할 수 있다.



You can use the Qt Install Framework to create installation files in the form of templates in the form of workflows, as shown in the figure above. Let's use the following example to create an installation distribution.

- ✓ Create an offline, installable distribution using the Qt Install Framework

Create a directory called 00\_Installer\_Example, as shown in the example source code below. Then, create the 00\_Installer\_Example.pro file in the created directory like below.

```
TEMPLATE = aux

INSTALLER = 00_Installer_Example

INPUT = $$PWD/config/config.xml $$PWD/packages

myexam.input = INPUT
myexam.output = $$INSTALLER
myexam.commands = C:/Qt/QtIFW-4.6.1/bin/binarycreator \
                  -c $$PWD/config/config.xml \
                  -p $$PWD/packages ${QMAKE_FILE_OUT}

myexam.CONFIG += target_predeps no_link combine

QMAKE_EXTRA_COMPILERS += myexam
```

The TEMPLATE keyword is used to specify whether the installation file is a library or an application. In this case, aux is entered.

The INSTALLER keyword specifies the name of the installation file. When you build later, the name of the generated installation file will be the name you specify with the INSTALLER keyword.

The next INPUT keyword must specify two items. The first is the location and filename of the config.xml file. The second item specifies the packages directory.

The config.xml file specifies the name of the project, version, provider, and, in the case of MS Windows, the name of the Start menu.

The packages directory contains the executables and libraries that will be installed; copy them here.

The myexam.input keyword specifies the INPUT keyword. The second myexam.output

specifies the name of the installation file specified by the INSTALLER keyword.

myexam.commands enters the directory and file name where the binarycreator provided by the Qt Install Framework is located to create the installation file. The "-c" option specifies the location and name of the config.xml file.

"-p" is the directory and filename of the package file. Create a project file as shown above, create a config directory, and write config.xml as shown below.

```
<?xml version="1.0" encoding="UTF-8"?>

<Installer>

    <Name>Example Software</Name>
    <Version>1.0.0</Version>
    <Title>Example Software</Title>

    <Publisher>Qt 개발자</Publisher>
    <StartMenuDir>Qt 개발자 메뉴</StartMenuDir>
    <TargetDir>C:/Example_Software</TargetDir>

</Installer>
```

This time, create a packages directory and inside the packages directory, create a com.vendor.product directory. Inside the com.vendor.product directory, create the data and meta directories.

The data directory is where you copy the executable of the application you want to install, the libraries it uses, and the resource files it uses.

In this example, we will build the analog example executable of the Qt example with the MinGW compiler in Release mode and copy it into the data directory. You should also copy any libraries referenced by the executable. In our example, the source code references 00\_AnalogClock.

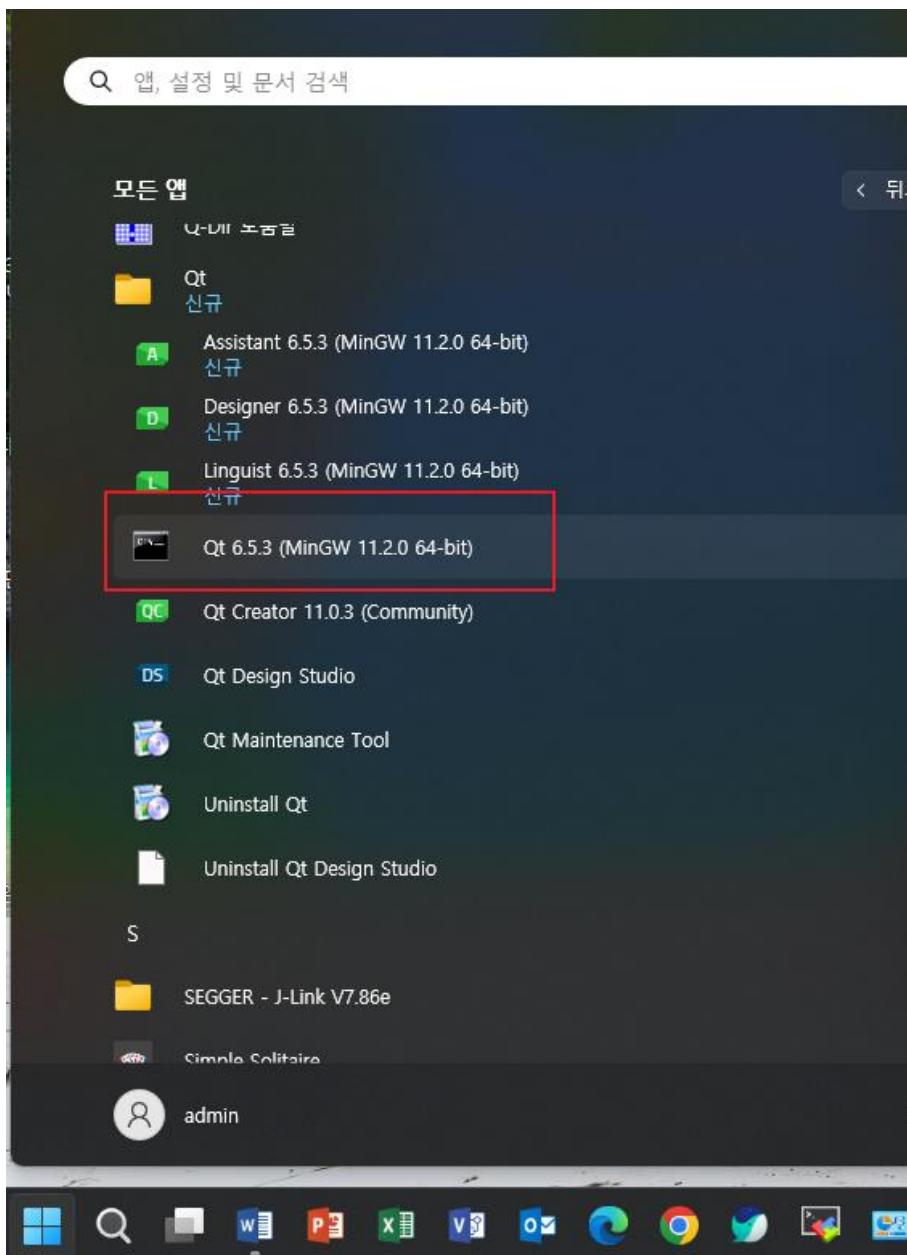
To find out which libraries are referenced by the executable, you can utilize the windeployqt.exe or windeployqt6.exe files on MS Windows platforms. The location of this tool may vary for different compilers. For example, if you are using the MinGW compiler, it might look like this

```
C:[ Qt Installed directory ]\mingw_64\bin
```

In the above directory, `windeployqt.exe` is available for Qt 5 and `windeployqt6.exe` is available for Qt 6.

You can also use the command prompt provided by MS Windows. However, in this example, we will use a command prompt with the Qt installation directory set to PATH.

Run the command prompt provided by Qt, as shown in the figure below.



Then navigate to the location of the executable file and run it as shown below.

```
> windeployqt6.exe 00_AnalogClock.exe
```

If you type and run it as above, 00\_AnalogClock.exe will automatically copy the necessary files for you.

```
D:\QtProjects\Examples\ch33\00_ExecutFiles>dir/w
D 드라이브의 볼륨: LocalDisk
볼륨 일련 번호: 1C0C-8749

D:\QtProjects\Examples\ch33\00_ExecutFiles 디렉터리

[.]          [...]          00_AnalogClock.exe
D3Dcompiler_47.dll    [generic]      [iconengines]
[imageformats]        libgcc_s_seh-1.dll   libstdc++-6.dll
libwinpthread-1.dll   [networkinformation]  opengl32sw.dll
[platforms]           Qt6Core.dll     Qt6Gui.dll
Qt6Network.dll        Qt6Svg.dll     Qt6Widgets.dll
[styles]              [tls]          [translations]

11개 파일      54,189,240 바이트
10개 디렉터리  302,038,573,056 바이트 남음

D:\QtProjects\Examples\ch33\00_ExecutFiles>
```

This tool is currently only available for MS Windows.

In addition to this tool, you can use a tool called Dependency Walker to see which libraries are referenced by MS Windows. (This tool is not provided by Qt).

This tool can show you what libraries are referenced. However, it does not automatically find and copy the libraries for you.

Jesus loves you.

The screenshot shows the official website for Dependency Walker 2.2. At the top, there's a browser header with the title "Dependency Walker (depends.e)" and the URL "dependencywalker.com". Below the header is the software's logo and the title "Dependency Walker 2.2". The main content area contains a brief description of the tool, highlighting its ability to scan Windows modules and build hierarchical dependency trees. It also mentions troubleshooting capabilities for system errors like missing modules and import/export mismatches. A note at the bottom states that the software is completely free but may not be bundled with other products.

Dependency Walker is a free utility that scans any 32-bit or 64-bit Windows module (exe, dll, ocx, sys, etc.) and builds a hierarchical tree diagram of all dependent modules. For each module found, it lists all the functions that are exported by that module, and which of those functions are actually being called by other modules. Another view displays the minimum set of required files, along with detailed information about each file including a full path to the file, base address, version numbers, machine type, debug information, and more.

Dependency Walker is also very useful for troubleshooting system errors related to loading and executing modules. Dependency Walker detects many common application problems such as missing modules, invalid modules, import/export mismatches, circular dependency errors, mismatched machine types of modules, and module initialization failures.

Dependency Walker runs on Windows 95, 98, Me, NT, 2000, XP, 2003, Vista, 7, and 8. It can process any 32-bit or 64-bit Windows module, including ones designed for Windows CE. It can be run as graphical application or as a console application. Dependency Walker handles all types of module dependencies, including implicit, explicit (dynamic / runtime), forwarded, delay-loaded, and injected. A detailed help is included.

Dependency Walker is completely free to use. However, you may not profit from the distribution of it, nor may you bundle it with another product.

The screenshot shows the Dependency Walker application interface. The title bar reads "Dependency Walker - [00\_AnalogClock.exe]". The menu bar includes File, Edit, View, Options, Profile, Window, and Help. The toolbar contains various icons for file operations. The main window has two panes. The left pane displays a hierarchical dependency tree for the executable "00\_ANALOGCLOCK.EXE", which depends on "QT6CORE.DLL". "QT6CORE.DLL" in turn depends on "ADVAPI32.DLL", "MSVCRT.DLL", "NTDLL.DLL", and several entries marked with question marks ("API-MS-WIN-EVENTING-CON", "KERNELBASE.DLL", "API-MS-WIN-EVENTING-", "API-MS-WIN-CORE-API", "EXT-MS-WIN-ADVAPI32"). The right pane shows two tables: one for "PI" and one for "Entry P". The bottom pane is a table showing the loaded modules, their file time stamps, link time stamps, and file sizes. One module, "API-MS-WIN-CORE-APIQUERY-L1-1-0.DLL", is listed with an error message in red: "Error opening file. 지정된 파일을 찾을 수 없습니다" (File not found).

Module

File Time Stamp

Link Time Stamp

File Size

API-MS-WIN-CORE-APIQUERY-L1-1-0.DLL

Error opening file. 지정된 파일을 찾을 수 없습니다

Error opening file. 지정된 파일은 차트에 아직尚未 등록되었습니다.

For Help, press F1

If you are on Linux, you can use the ldd command to see what libraries the executable

is referencing.

```
# ldd /usr/sbin/qtApplication
    linux-vdso.so.1 => (0x00007fff85bff000)
    libpcre.so.3 => /lib/libpcre.so.3 (0x00007ff366142000)
    libapr-1.so.0 => /usr/lib/libapr-1.so.0 (0x00007ff30)
    libpthread.so.0 => /lib/libpthread.so.0 (0x00007ff00)
    libexpat.so.1 => /lib/libexpat.so.1 (0x00007ff3600000)
    ...
```

If your platform is MacOS, you can use the otool command to see the libraries referenced by the executable.

```
otool -L MyApp.app/Contents/MacOS/MyApp
```

Create a meta directory under the com.vendor.product directory with the 00\_AnalogClock.exe executable and the necessary libraries, and create a file like this

In this directory, I have copied and placed main\_icon.ico. This file will be used as the icon in the menu.

Next, create a meta directory under the com.vendor.product directory and create files like the following.

파일명	설명
package.xml	Package configuration file
license.txt	License information file
installscript.qs	Installation scripts

c The package.xml file specifies the name to display, a description, the filename where the license information is written to be displayed in the installation dialog, and the script files required for installation.

```
<?xml version="1.0" encoding="UTF-8"?>
<Package>
    <DisplayName>Example 소프트웨어</DisplayName>
    <Description>Example 아날로그 시계 예제입니다.</Description>
    <Version>1.0.0-1</Version>
    <ReleaseDate>2030-05-18</ReleaseDate>
    <Licenses>
        <License name="My License Agreement" file="license.txt"/>
    </Licenses>
```

```
<Default>script</Default>
<Script>installscript.qs</Script>
</Package>
```

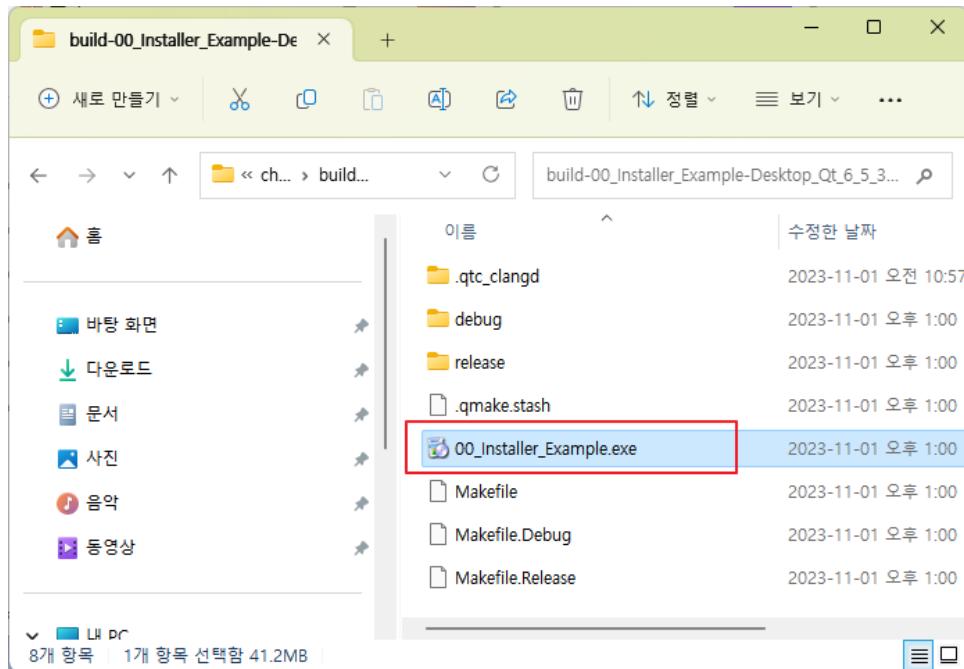
Installscript.qs specifies the location of the Shortcut menu, the Lnk file on the Start menu, and the launch icon for the package to be installed. Installscript.qs looks like this

```
function Component()
{
    // default constructor
}

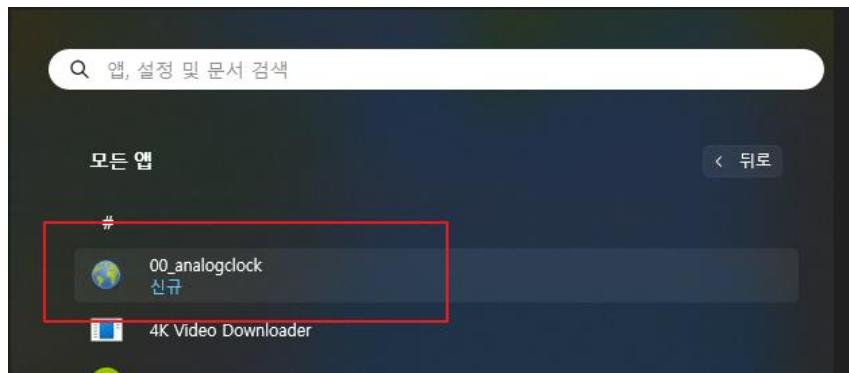
Component.prototype.createOperations = function()
{
    component.createOperations();

    if (systemInfo.productType === "windows")
    {
        component.addOperation(
            "CreateShortcut",
            "@TargetDir@/00_analogclock",
            "@StartMenuDir@/00_analogclock.lnk",
            "workingDirectory=@TargetDir@",
            "iconPath=@TargetDir@/main_icon.ico");
    }
}
```

After writing the above, you are all set to create an installer distribution. Open the 00\_Installer\_Example.pro file in the Qt Creator tool and build it. When the build is complete, you will see that the 00\_Installer\_Example.exe installer distribution file has been created in the build directory.

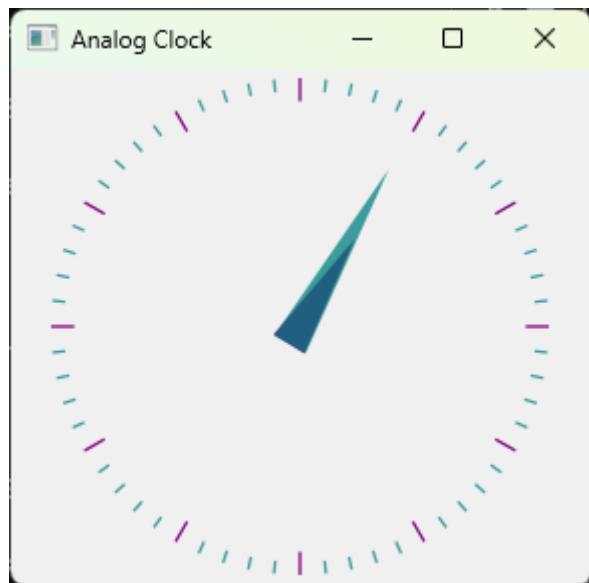


Let's run the file `00_Installer_Example.exe`. After the installation is complete, you can see that the menu is registered in the menu.



As shown in the image above, if you run `00_analogclock` from the menu, you can see the example application loading as shown below.

Jesus loves you.



You can refer to the 00\_Installer\_Example directory for the examples we've covered so far.

## 33. Qt Graphics View Framework

Sometimes you need to use a window area larger than the monitor to develop your application. For example, suppose you need to implement a map-related application.

What if you need to display tens of thousands of objects, such as buildings, on a map?

No matter how big your monitor is, you won't be able to display the entire map on it.

For example, would you be able to display a map of the entire world on your monitor?

You might be able to if you scale it down to fit the size of your monitor, but not otherwise.

Therefore, Qt provides the Qt Graphics View Framework to facilitate the implementation of CAD applications such as maps, architectural drawings, semiconductor design drawings, etc.

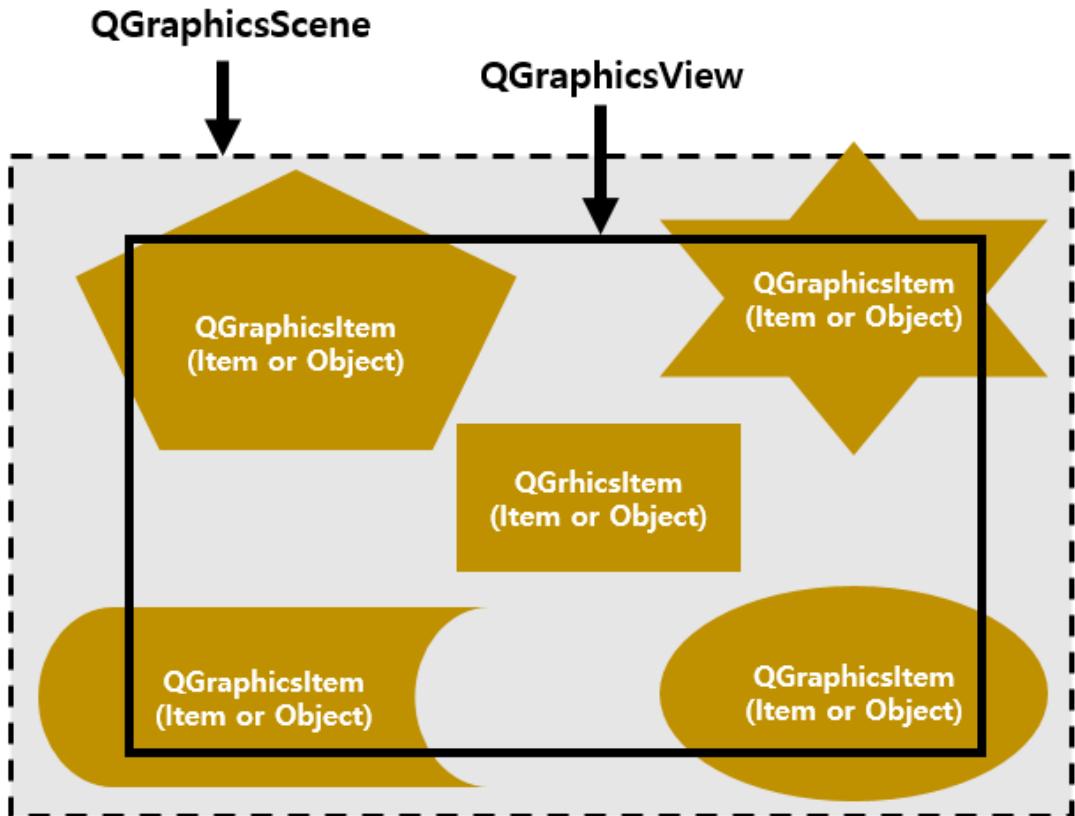
The Qt Graphics View Framework can be easily implemented to display specific objects (e.g., buildings on a map) or to zoom in or out, including zooming.

The Qt Graphics View Framework uses the Binary Space Partitioning (BSP) Tree algorithm as its internal implementation algorithm. This makes it possible to display a large number of objects quickly.

The Graphics View Framework provides three elements

Element	Description
QGraphicsView	An area that can be visually displayed. For example, a specific area in a GUI. This class is a full-sized area on the GUI.
QGraphicisScene	The QGraphicisScene class is a logical region. It is displayed inside a QGraphicsView region.
QGraphicsItem	They are displayed on objects of class QGraphicisScene. Used in the same way as objects displayed on a map, for example.

The following figure shows a schematic representation of the three components of the Qt Graphics View Framework.



Suppose you have a GUI-based application on the bottom of the monitor. The GUI of the GUI-based application has a size of QGraphicsView. The QGraphicsScene is a virtual area. And QGraphicsItems are the items placed in the QGraphicsScene.

For example, a QGraphicsItem is a building (Item or Object) on a map. The QGraphicsItem class provides a `paint( )` virtual function, like QPainter, that allows you to display text, image grid shapes, etc. in the QGraphicsItem area. The following is example source code for an implementation that inherits from QGraphicsItem.

```
...
class SimpleItem : public QGraphicsItem
{
public:
    QRectF boundingRect() const override
    {
        qreal penWidth = 1;
        return QRectF(-10 - penWidth / 2, -10 - penWidth / 2,
                     20 + penWidth, 20 + penWidth);
    }
}
```

```
}

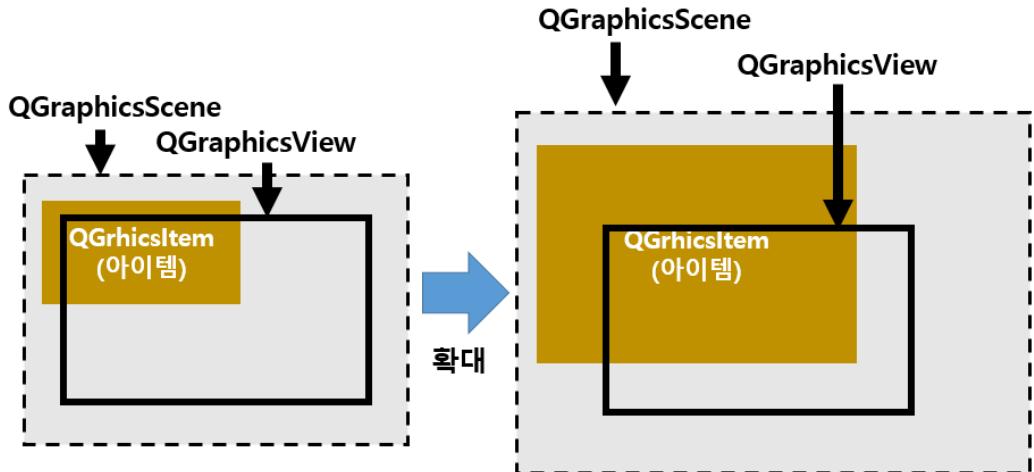
void paint(QPainter *painter, const QStyleOptionGraphicsItem *option,
           QWidget *widget) override
{
    painter->drawRoundedRect(-10, -10, 20, 20, 5, 5);
}
};

...
```

In the example source code above, the `paint( )` function provides the same functionality as the `paintEvent( )` function of a `QWidget`.

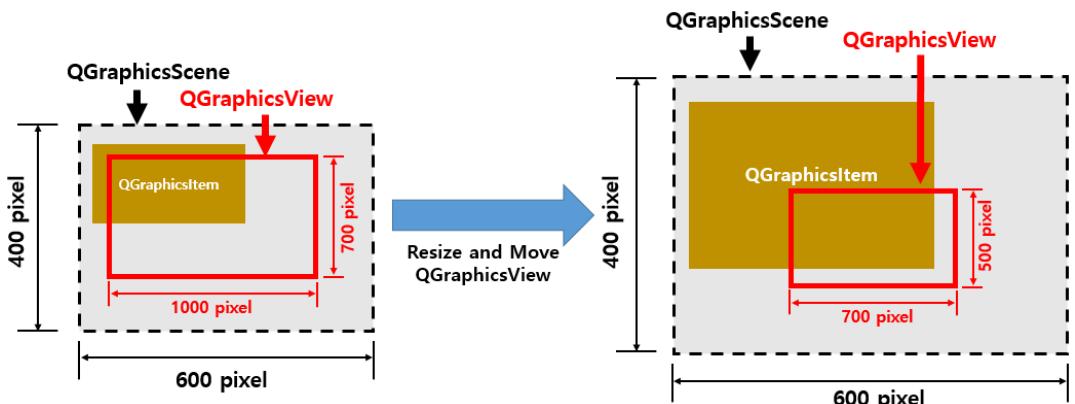
In addition to `QGraphicsItem`, Qt provides several other item classes, including `QGraphicsRectItem`, `QGraphicsTextItem`, and `QGraphicsPixmapItem`.

Qt provides the ability to zoom a `QGraphicsScene` provided by the Qt Graphics View Framework to fit the `QGraphicsView` area.



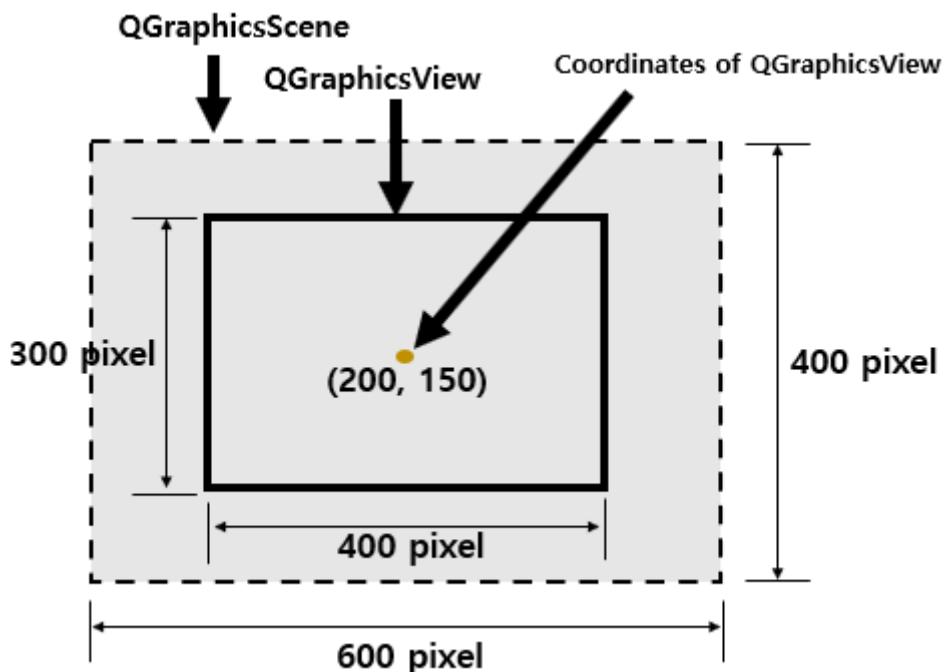
As shown in the figure above, the Qt Graphics View Framework logically increases the size of the `QGraphicsScene` when zoomed in, while leaving the size of the `QGraphicsView` unchanged.

However, this does not increase the actual horizontal and vertical dimensions of the `QGraphicsScene`. The same is true for `QGraphicsItem` registered in the `QGraphicsScene`. As shown in the following figure, the original size of the `QGraphicsScene` remains unchanged when zoomed in, and the same is true when zoomed out.



As shown in the figure above, zooming in or out does not change the size of the actual `QGraphicsScene`.

Also, the coordinate system of the `QGraphicsView` and the virtual `QGraphicsScene` may be different. For example, suppose the X and Y coordinates of the `QGraphicsView` are 200 and 150, as shown in the following figure.



As shown in the figure above, suppose that the horizontal and vertical dimensions of a `QGraphicsView` are 400 and 300, and the coordinates at the center of the `QGraphicsView` are 200 and 150.

As shown in the figure above, the 200 and 150 coordinates are the coordinates of the

Jesus loves you.

QGraphicsView, not the coordinates of the QGraphicsScene. The coordinate points of 200, 150 in the QGraphicsView would be approximately 300, 200 in the QGraphicsScene.

The Qt Graphics View Framework provides functions to convert coordinates. As shown in the figure above, it provides functions to change the coordinates of various QGraphicsView to QGraphicsScene coordinates, and vice versa, to change the coordinates of a QGraphicsScene to the coordinates of a QGraphicsView.

For example, it provides various functions such as the mapFromScene( ) member function and mapToScene( ). It also provides a function to change the coordinates of items registered in a QGraphicsScene to the coordinates of a QGraphicsView. As shown in the following example source code, to map a QGraphicsView to a QGraphicsScene, you can use the following code.

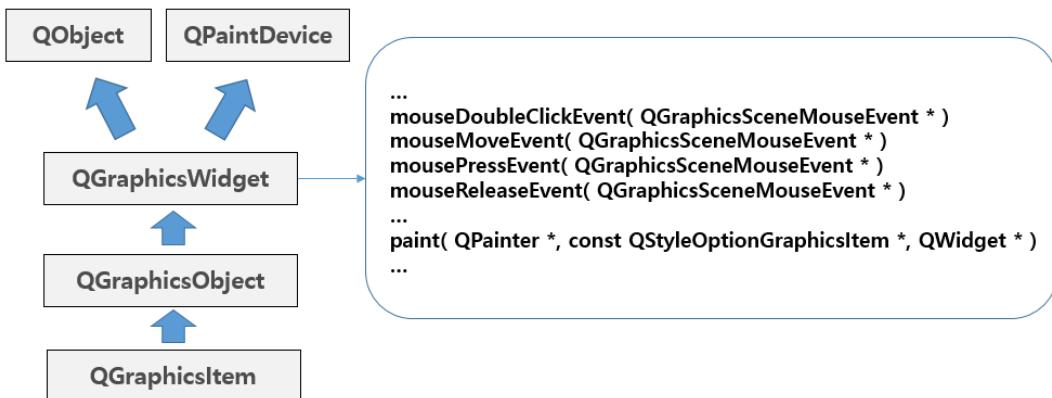
```
...
QGraphicsScene scene;
myPopulateScene(&scene);

QGraphicsItem *item;
scene.addItem(&item);

QGraphicsView view(&scene);
view.show();
...
```

To register a QGraphicsItem on a QGraphicsScene, you can use the addItem( ) member function provided by the QGraphicsScene class.

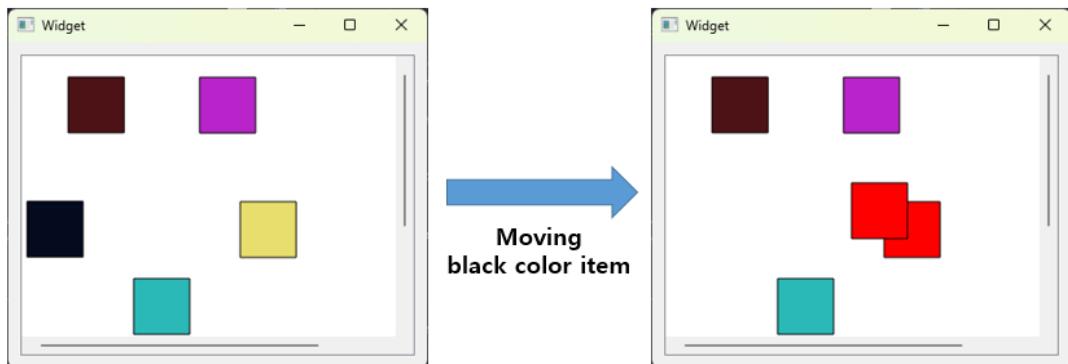
The QGraphicsItem class can draw shapes or display images using 2D graphics elements in the area using the paint( ) virtual function, as described earlier. You can also handle user events in the QGraphicsItem area.



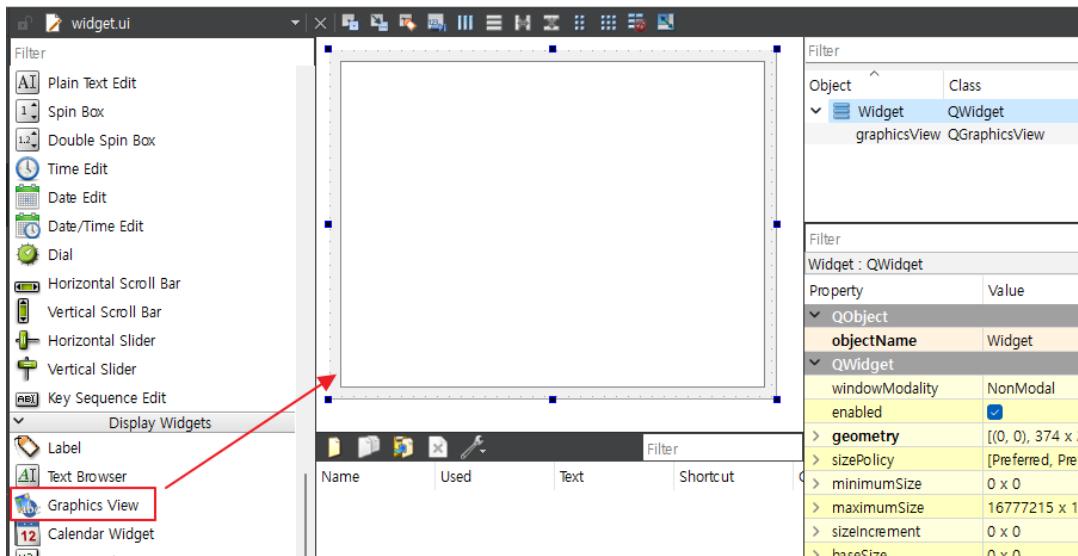
- ✓ Example collision detection implementation for `QGraphicsItem`

In this example, we implement a feature that detects when Shape items created with the `QGraphicsItem` class collide. Items registered in a `QGraphicsScene` can be dragged with the mouse.

If the dragged item collides with another item, the color of the collided item changes to red.



When creating a project in Qt Creator, create a new project based on `QWidget`. Then place the widget in the GUI form as follows.



Place the [Graphics View] item in the left widget list on the GUI as shown in the image above. Then, create a new class called Shape in the project and write the following in the shape.h header file.

```
#ifndef SHAPE_H
#define SHAPE_H

#include <QObject>
#include <QGraphicsItem>

class Shape : public QGraphicsItem
{
public:
    Shape();

    QRectF boundingRect() const override;
    QPainterPath shape() const override;
    void paint(QPainter *painter,
               const QStyleOptionGraphicsItem *option,
               QWidget *widget) override;

protected:
    void advance(int step) override;
    void mouseMoveEvent(QGraphicsSceneMouseEvent *event) override;

private:
    QColor color;
}
```

```
};

#endif // SHAPE_H
```

Class Shape inherits from and implements class QGraphicsItem. The boundingRect( ) override function is used inside QGraphicsScene. In this function, you can pass in the starting point x, y, width, and height values for this item.

The shape( ) override function is a collision detection function. In this function, you can specify the area where collisions are detected. If an item registered within a QGraphicsScene collides with another registered item, this function will detect the registered area and detect the collision. The paint( ) function can be used to draw shapes or other desired elements in the QGraphicsItem area.

The Advance( ) function automatically calls this function when the QGraphicsScene is updated.

We used a timer in the Widget class. This timer is called every 30 milliseconds to update the QGraphicsScene.

Updating the QGraphicsScene calls the advanced( ) virtual function of the Shape class registered on this QGraphicsScene, which in turn calls the paint( ) virtual function of the Shape class.

The mouseMoveEvent( ) function provides functionality for dragging a specific QGraphicsItem with the mouse within the QGraphicsScene. The following example source code is from the Shape class. It is written as follows

```
#include "shape.h"

#include <QGraphicsScene>
#include <QPainter>
#include <QRandomeGenerator>
#include <QStyleOption>
#include <QGraphicsSceneMouseEvent>
#include <QDebug>

Shape::Shape()
{
    setFlags(QGraphicsItem::ItemIsSelectable|
             QGraphicsItem::ItemIsMovable);
```

```
color = QColor(QRandomGenerator::global()->bounded(256),
               QRandomGenerator::global()->bounded(256),
               QRandomGenerator::global()->bounded(256));
}

QRectF Shape::boundingRect() const
{
    return QRectF(0, 0, 50, 50);
}

QPainterPath Shape::shape() const
{
    QPainterPath path;
    path.addRect(0, 0, 50, 50);

    return path;
}

void Shape::paint(QPainter *painter,
                  const QStyleOptionGraphicsItem *,
                  QWidget *)
{
    if(scene()->collidingItems(this).isEmpty()) {
        painter->setBrush(color);
    } else {
        painter->setBrush(QColor(Qt::red));
    }

    painter->drawRect(0, 0, 50, 50);
}

void Shape::advance(int step)
{
    update();
}

void Shape::mouseMoveEvent(QGraphicsSceneMouseEvent *event)
{
    QPointF eventpos = event->pos();
    QPointF shapePos = this->pos();
```

```
QPointF wPos(eventpos.x() + shapePos.x() - 25,  
             eventpos.y() + shapePos.y() - 25);  
  
setPos(wPos);  
setPos(wPos);  
  
update();  
  
QGraphicsItem::mouseMoveEvent(event);  
}
```

In the constructor, the `setFlags( )` function allows the constructor to set values to enable the item to move. The color variable is used as the color of the Brush inside the Shape.

In the `paint( )` function, `scene( )->collidingItems(this).isEmpty( )` can tell if an item is currently colliding.

It uses the value of the color variable set in the class constructor. Otherwise, it changes the item Color to red when a collision is detected.

The following example source code is a Widget class for using the Shape class. Create a header file as shown in the following example.

```
#ifndef WIDGET_H  
#define WIDGET_H  
  
#include <QWidget>  
#include <QGraphicsScene>  
#include <QTimer>  
  
QT_BEGIN_NAMESPACE  
namespace Ui { class Widget; }  
QT_END_NAMESPACE  
  
class Widget : public QWidget  
{  
    Q_OBJECT  
  
public:  
    Widget(QWidget *parent = nullptr);  
    ~Widget();  
  
private:
```

```
Ui::Widget *ui;  
  
QGraphicsScene *m_scene;  
QTimer m_tiamer;  
};  
#endif // WIDGET_H
```

In the constructor function of this class, we create a QGraphicsScene and register a QGraphicsItem on the QGraphicsScene.

Run the example we have created so far and then try dragging the QGraphicsItem with the mouse. If a collision occurs, the collided Shape will change its internal color to red. When the collision is released, it changes back to its original color. For the full source, see the 01\_ShapeDetection directory.

## 34. Animation Framework and State Machine

To use animated elements in the GUI, such as smooth screen transitions, Qt provides the Animation Framework.

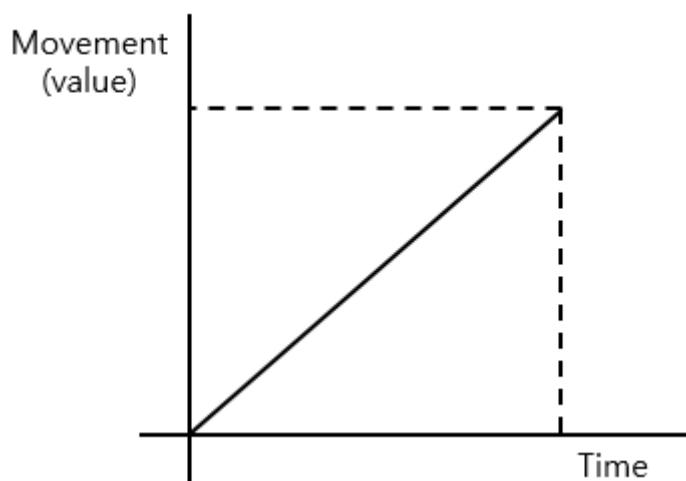
Animation elements can be used on the windowed screen, and widgets placed on the GUI can use their own animation elements.

The Animation Framework is often used in QML. However, you can also use the Animation Framework in C++.

You can use animation elements such as transparency, movement, zooming, and so on. For example, if your application needs to run with a window that is 600 x 400 pixels across and down the screen, you can animate the window to increase in size from 100 x 100 to 600 x 400 for a specified amount of time.

At the same time, you can use Opacity to animate the transparency to change from 0.0 to 1.0 for a specified amount of time. 0.0 is completely transparent and 1.0 is all transparency.

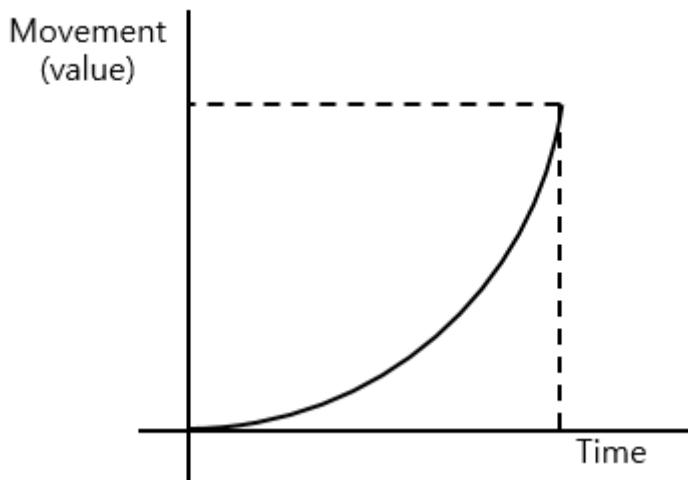
Additionally, you can add Easing Curves to the animation. For example, suppose you want the position (X, Y) of a GUI button to move from coordinates (100, 100) to (200, 200) over a period of 1.0 seconds, with a travel time of 1.0 seconds. It will travel at a constant speed from the start coordinate to the destination coordinate.



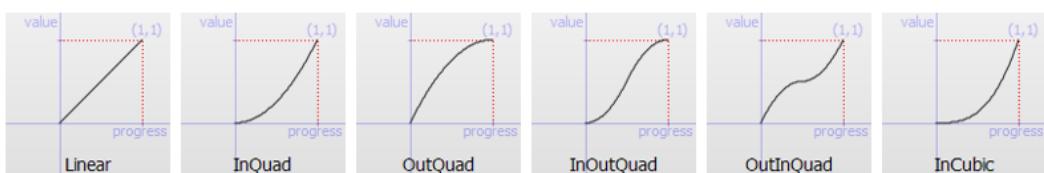
Jesus loves you.

As shown in the figure above, the y-axis, called Move (Value), is a graph that shows that it takes a constant 1.0 second time for the x,y position coordinates of the GUI button to move from the value 100,100 to the value 200,200. In other words, the time increases linearly with the movement value.

However, if you use Easing Curves for Time, you can change the value of time for the travel value, as shown in the following figure.



The value of the time to travel to the X, Y coordinates of the final destination can be applied as shown in the figure above.



There are over 40 different types of Easing Curves available in addition to the ones shown above.

Qt also allows you to use a technique called State Machine for Animation elements. For example, consider an ON/OFF switch.

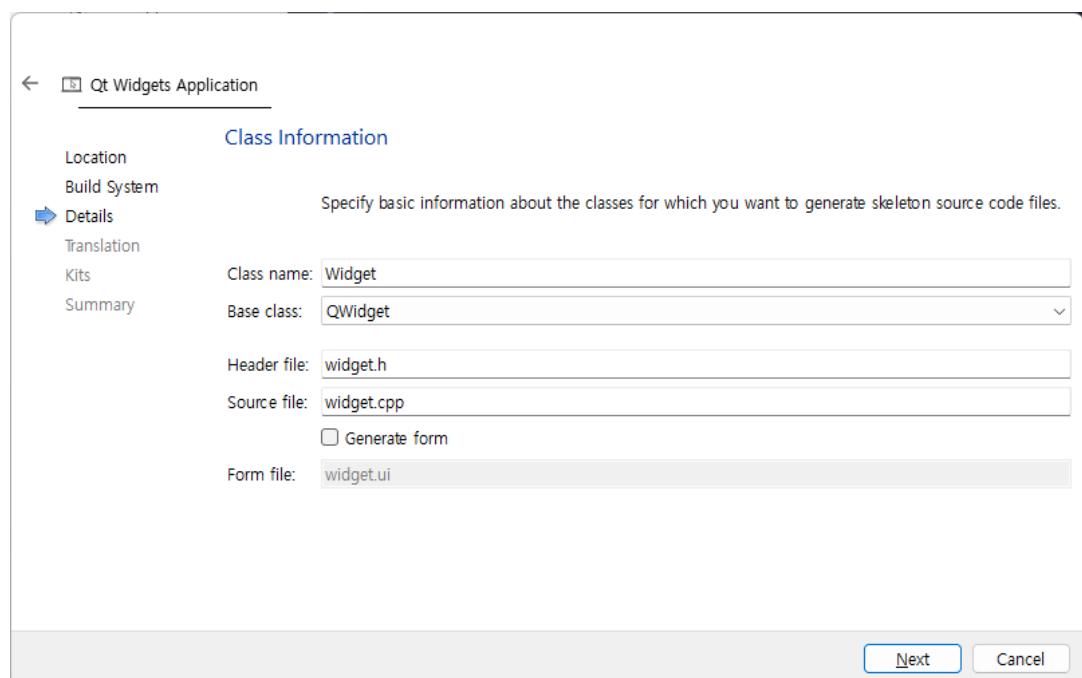
Not only can you use ON/OFF to change a single value, but you can use multiple options together. For example, if you want the fluorescent lights in your foyer and the fluorescent lights in your living room to turn on together, you can use a technique called a State Machine to turn them on or off together. In other words, you can use it to have multiple values apply at the same time, depending on the situation.

Let's say you have a GUI with a button A, a button B, and a button C. If you want to move the position of the C button on the GUI from 100, 100 to 200, 200 and change the transparency from 1.0 to 0.0 at the same time when the A button is pressed, you can use a State Machine.

Of course, Animation provides the ability to run in parallel, but if you need to change dozens of states, it may be more efficient to use the State Machine technique.

✓ Examples of using Animation in the GUI

When creating a project, create it based on Qt Widget. Also, widget.ui is not used when creating a project.



In this example, we will place a QPushButton on the GUI. To move the placed QPushButton to a specific position, we use an Animation element.

When the button is clicked, the x, y position moves from 10, 10 to 200, 150. The time to move is 3000 milliseconds. Here is the widget.h header file. It looks like this

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
```

```
#include <QPropertyAnimation>

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    QPropertyAnimation *animation;

public slots:
    void btnClicked();

};

#endif // WIDGET_H
```

Declare an object of class QPropertyAnimation, as shown in the example source code above. The btnClicked( ) Slot function is called when the button is clicked. The following is the widget.cpp source code.

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

Widget::Widget(QWidget *parent) : QWidget(parent)
{
    this->resize(350, 200);

    QPushButton *btn = new QPushButton("Button", this);
    connect(btn, &QPushButton::pressed, this, &Widget::btnClicked);
    btn->setGeometry(10, 10, 100, 30);

    animation = new QPropertyAnimation(btn, "geometry", this);

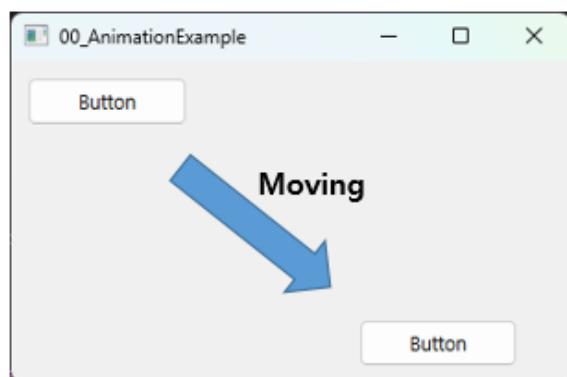
    animation->setDuration(3000); // 3 seconds
    animation->setStartValue(QRect(10, 10, 100, 30)); // Start position
    animation->setEndValue(QRect(200, 150, 100, 30)); // End position
}

void Widget::btnClicked()
```

```
{  
    animation->start();  
}  
  
Widget::~Widget()  
{  
}
```

Animation is an Object of class QPropertyAnimation.

setDuration( ) can specify the total time to move. setStartValue( ) specifies the values of x, y, width, and height for the start coordinates. setEndValue( ) specifies the position to move to and the horizontal and vertical dimensions. The btnClicked( ) Slot function is called when a button is clicked.



As shown in the image above, clicking the button moves the x and y coordinates to 200 and 150 for 3 seconds. Next, let's use the Easing Curve in this example as follows.

Add the source code to the end of the Widget class constructor function as shown below.

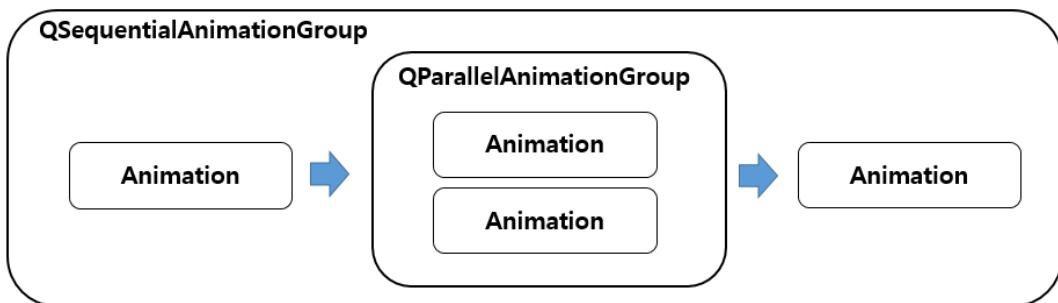
```
...  
animation->setEasingCurve(QEasingCurve::OutInQuart);  
...
```

You can find the source code for this example in the 00\_AnimationExample directory.

- ✓ Example using the Animation group

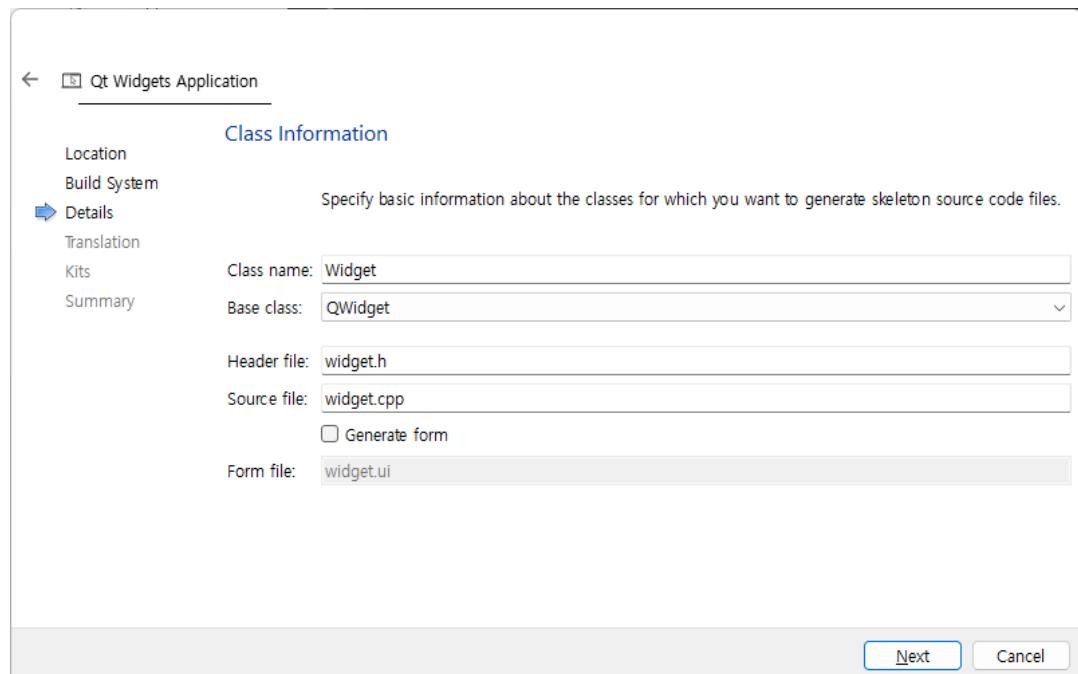
In this example, we will show how to use Grouping to run Animation elements simultaneously or sequentially. If you have many Animation elements that you want to use in the Animation Framework, you can use the grouping feature to group them

together, as shown in the following figure.



A **QSequentialAnimationGroup** class can run Animations sequentially. In contrast, the **QParallelAnimationGroup** class allows Animations to be performed simultaneously (in parallel).

Let's write an example that uses a grouping of Animations as shown in the figure above. Create a project based on Qt Widget. And in this project, we will not use `widget.ui` as in the previous example.



Below is the `widget.h` header file. Let's write it like this

```
#ifndef WIDGET_H  
#define WIDGET_H  
  
#include <QWidget>
```

```
#include <QPropertyAnimation>
#include <QSequentialAnimationGroup>

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private slots:
    void slot_sequential();

private:
    QPropertyAnimation *anim1;
    QPropertyAnimation *anim2;
    QSequentialAnimationGroup *sGroup;

};

#endif // WIDGET_H
```

The following example source code is widget.cpp.

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

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    resize(320, 270);
    QPushButton *btn1 = new QPushButton("First", this);
    btn1->setGeometry(10, 10, 100, 30);

    QPushButton *btn2 = new QPushButton("Second", this);
    btn2->setGeometry(10, 45, 100, 30);

    anim1 = new QPropertyAnimation(btn1, "geometry");
    anim1->setDuration(2000); // 2 Seconds (Unit: Milliseconds)
    anim1->setStartValue(QRect(10, 10, 100, 30)); // Start Position
    anim1->setEndValue(QRect(200, 150, 100, 30)); // End Position

    anim2 = new QPropertyAnimation(btn2, "geometry");
```

```
anim2->setDuration(2000); // 2 Seconds (Unit: Milliseconds)
anim2->setStartValue(QRect(10, 45, 100, 30)); // Start Position
anim2->setEndValue(QRect(200, 195, 100, 30)); // End Position

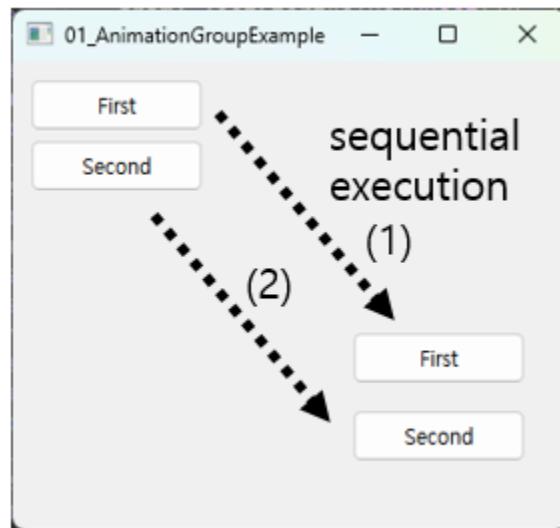
sGroup = new QSequentialAnimationGroup; // Sequential Group
sGroup->addAnimation(anim1);
sGroup->addAnimation(anim2);

connect(btn1, SIGNAL(clicked()), this, SLOT(slot_sequential()));

void Widget::slot_sequential()
{
    sGroup->start(QPropertyAnimation::DeleteWhenStopped);
}

Widget::~Widget()
```

The example source code above places two buttons. When the [First] button is clicked, the [First] button is moved first and then the [Second] button is moved. The following figure shows the example running.



This time, let's modify it to move the buttons at the same time. Let's modify the source code of the Widget class we created above like below. First, change the widget.h header file like below.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QPropertyAnimation>
#include <QParallelAnimationGroup>

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private slots:
    void slot_parallel();

private:
    QPropertyAnimation *anim1;
    QPropertyAnimation *anim2;
    QParallelAnimationGroup *pGroup;

};

#endif // WIDGET_H
```

Next, modify the widget.cpp source code as shown below.

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

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    resize(320, 270);
    QPushButton *btn1 = new QPushButton("First", this);
    btn1->setGeometry(10, 10, 100, 30);

    QPushButton *btn2 = new QPushButton("Second", this);
    btn2->setGeometry(10, 45, 100, 30);

    anim1 = new QPropertyAnimation(btn1, "geometry");
    anim1->setDuration(2000); // 2 Seconds (Unit: Milliseconds)
```

```
anim1->setStartValue(QRect(10, 10, 100, 30)); // Start Position
anim1->setEndValue(QRect(200, 150, 100, 30)); // End Position

anim2 = new QPropertyAnimation(btn2, "geometry");
anim2->setDuration(2000); // 2 Seconds (Unit: Milliseconds)
anim2->setStartValue(QRect(10, 45, 100, 30)); // Start Position
anim2->setEndValue(QRect(200, 195, 100, 30)); // End Position

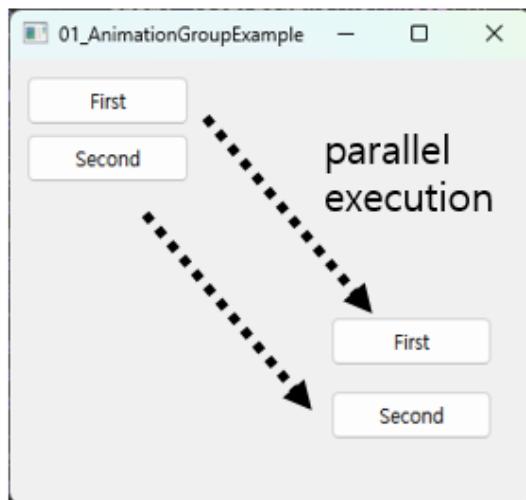
pGroup = new QParallelAnimationGroup; // Parallel Group
pGroup->addAnimation(anim1);
pGroup->addAnimation(anim2);

connect(btn2, SIGNAL(clicked()), this, SLOT(slot_parallel()));
}

void Widget::slot_parallel()
{
    pGroup->start(QPropertyAnimation::DeleteWhenStopped);
}

Widget::~Widget()
{
```

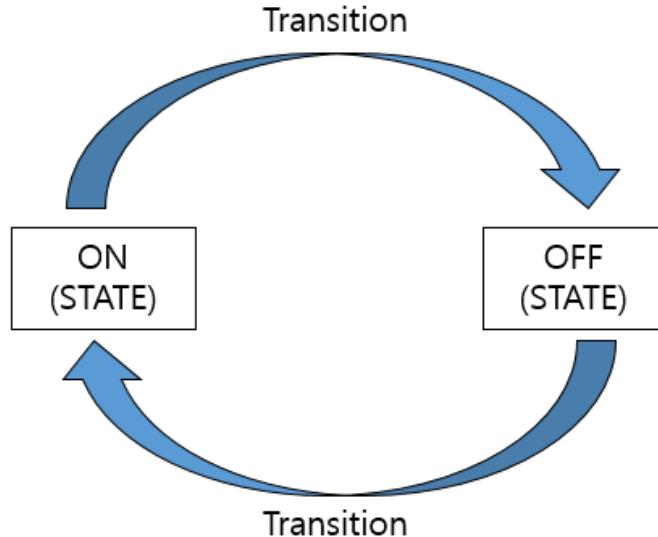
Build and run this example source code. Then click the [First] button. You should now see the animations for the [First] and [Second] buttons running at the same time.



You can find the source code for this example in the `01_AnimationGroupExample` directory.

- ✓ Example using Animation and State Machine

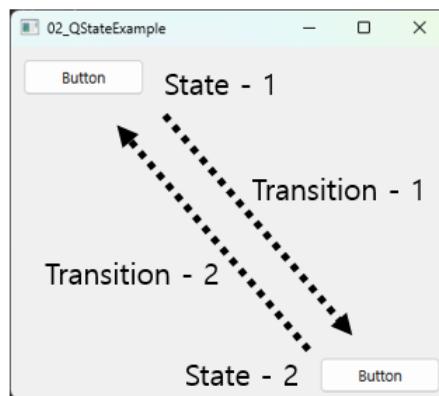
An action or behavior that changes from an ON state to an OFF state is called a Transition.



The reason we use the State Machine technique in Animation is that it allows us to handle many Animations at once.

For example, if you want to perform dozens of animations, you could group them together, but this would complicate your source code. However, this can be easily accomplished by applying a State Machine.

The State Machine technique is divided into States and Transitions.

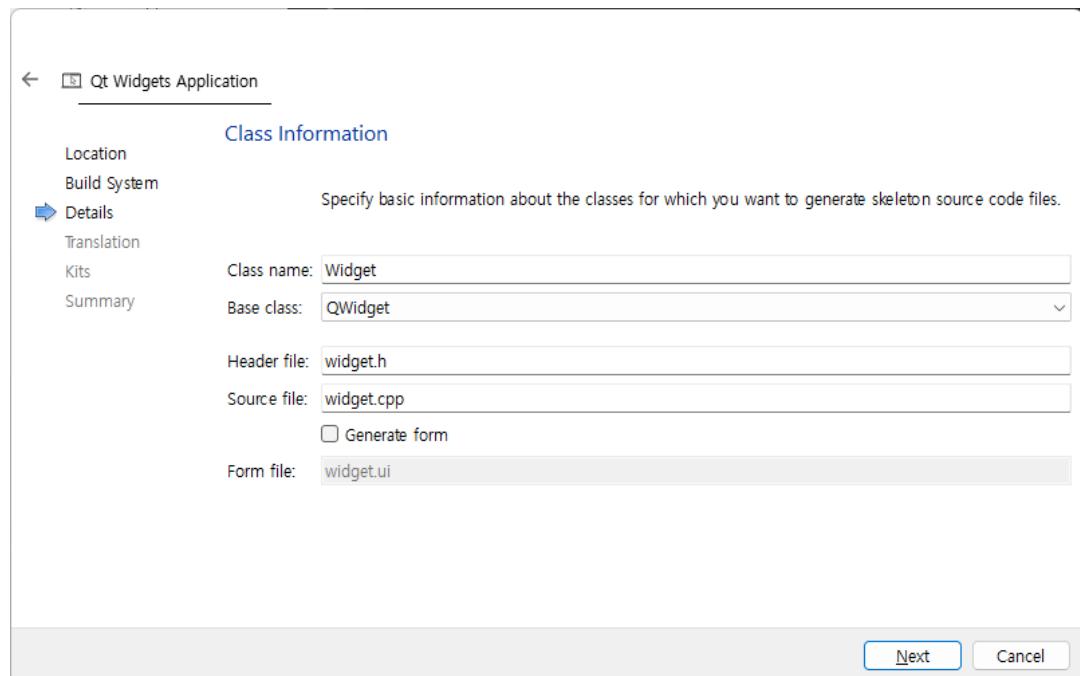


The figure above is a screen shot of the State Machine example. State - 1 is the x, y coordinates of the button at 10, 10. State - 2 is the state where the x, y coordinates are 250, 250.

Moving from State - 1 to State - 2 is called Transition - 1. Moving from State - 2 to State - 1 is called Transition - 2.

So, if you click the button while in State - 1, Transition - 2 is performed to enter State - 2.

Conversely, clicking the button while in State - 2 performs Transition - 2, which leads to State - 1. When creating a project, create a Qt Widget-based project. And do not use `widget.ui` as in the previous example.



After creating the project, write the `widget.cpp` source code like below.

```
#include "widget.h"
#include <QPushButton>
#include <QtStateMachine/QStateMachine>
#include <QSignalTransition>
#include <QStateMachine>
#include <QPropertyAnimation>
#include <QSignalTransition>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    resize(360, 290);
```

```
QPushButton *button = new QPushButton("Button", this);
button->setGeometry(10, 10, 100, 30);

QStateMachine *machine = new QStateMachine;

QState *state1 = new QState(machine);
state1->assignProperty(button, "geometry", QRect(10, 10, 100, 30));
machine->setInitialState(state1);

QState *state2 = new QState(machine);
state2->assignProperty(button, "geometry", QRect(250, 250, 100, 30));

QSignalTransition *t1 =
    state1->addTransition(button, SIGNAL(clicked()), state2);
t1->addAnimation(new QPropertyAnimation(button, "geometry"));

QSignalTransition *t2 =
    state2->addTransition(button, SIGNAL(clicked()), state1);
t2->addAnimation(new QPropertyAnimation(button, "geometry"));

machine->start();
}

Widget::~Widget()
{
```

You can refer to the 02\_QStateExample directory for examples.

## 35. Qt Chart

Qt provides the Qt Chart module to make it easy to use the Chart Component. The internals of the Qt Chart module are available in QWidget and QGraphicsWidget. It is also available in QML.

The Qt Chart module can be used with CMake and qmake. If you are using CMake, you can add it to your project file as follows.

```
find_package(Qt6 REQUIRED COMPONENTS Charts)
target_link_libraries(mytarget PRIVATE Qt6::Charts)
```

If you are using qmake, you can add the following to your project file like below.

```
QT += charts
```

The Qt Chart module requires the Qt Commercial License to be used. Otherwise, it must be used under the GPLv3 License.

The Qt Chart module provides the ability to represent various graphs such as Line, Pie, and Bar. For example, to provide a line chart, you can use the QLineSeries class.

```
QLineSeries* series = new QLineSeries();
series->setName("Line");
series->append(0, 6);
series->append(2, 4);
series->append(3, 8);
series->append(7, 4);
series->append(10, 5);
*series << QPointF(11, 1) << QPointF(13, 3)
    << QPointF(17, 6) << QPointF(18, 3)<< QPointF(20, 2);
```

The setName( ) member function can set the title of the Legend. The first argument to the Append( ) member function is a value on the x-axis and the second argument is a value on the y-axis. In addition to Append( ), you can use Operator to insert data.

Next, you can use a QChart class object to display the objects of class QLineSeries as shown below.

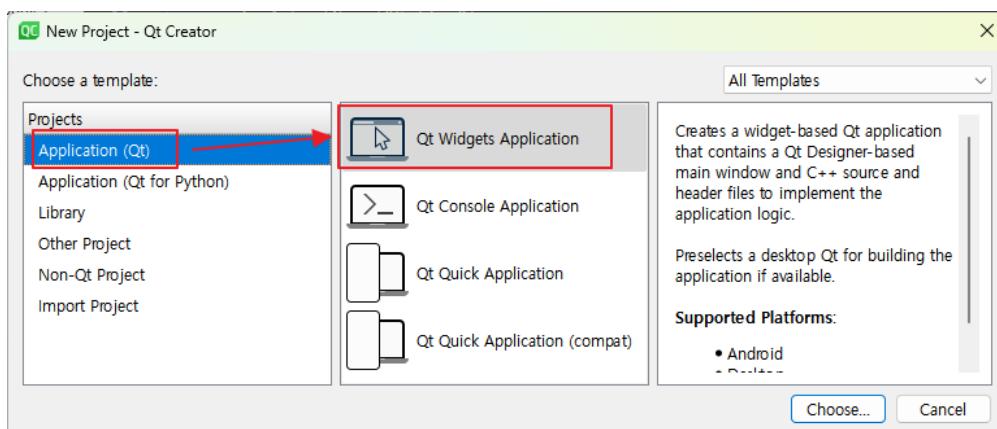
```
QChart *chart = new QChart();
chart->addSeries(series);
chart->createDefaultAxes();
chart->setTitle("Line");
```



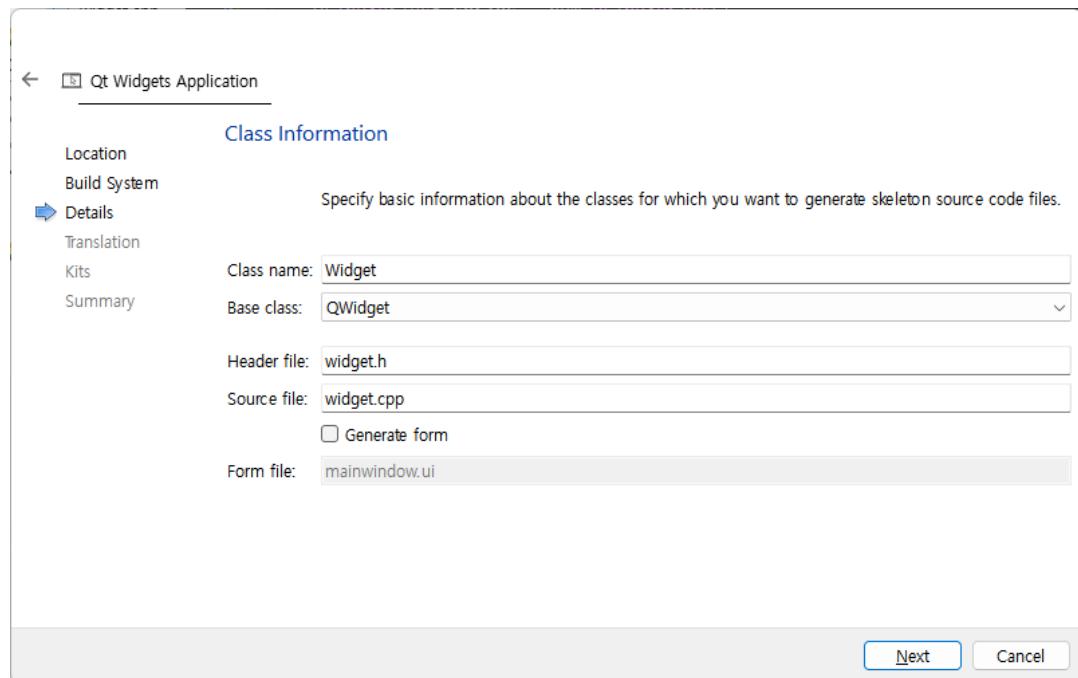
The QChart class can add multiple objects of the QLineSeries class. Let's take a closer look at how to utilize the Qt Chart module with the following example.

- ✓ Implement the Line and Scatter Chart example

Create a project based on Qt Widgets when creating a project.



This project does not use UI forms, so do not select the checkbox for the Generate form item in the Details dialog during the project creation dialog.



After creating the project, open the CMakeList.txt file and add the Charts module as shown below.

```
cmake_minimum_required(VERSION 3.5)
...
find_package(Qt NAMES Qt6 Qt5 REQUIRED COMPONENTS Widgets)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets Charts)
...
target_link_libraries(00_LineAndScatter PRIVATE Qt${QT_VERSION_MAJOR}::Widgets)
target_link_libraries(00_LineAndScatter PRIVATE Qt${QT_VERSION_MAJOR}::Charts)
...
if(QT_VERSION_MAJOR EQUAL 6)
    qt_finalize_executable(00_LineAndScatter)
endif()
```

Next, create the widget.cpp source code file as shown below.

```
#include "widget.h"
#include <QtCharts>
```

```
Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    window()->setMinimumSize(600, 400);

    QLineSeries* series = new QLineSeries();
    series->setName("Line");
    series->append(0, 6);
    series->append(2, 4);
    series->append(3, 8);
    series->append(7, 4);
    series->append(10, 5);
    *series << QPointF(11, 1) << QPointF(13, 3)
        << QPointF(17, 6) << QPointF(18, 3)<< QPointF(20, 2);

    QSplineSeries* series1 = new QSplineSeries();
    series1->setName("Spline");
    series1->append(0, 5);
    series1->append(2, 3);
    series1->append(3, 7);
    series1->append(7, 3);
    series1->append(10, 4);
    *series1 << QPointF(11, 2) << QPointF(13, 4)
        << QPointF(17, 7) << QPointF(18, 4)<< QPointF(20, 3);

    QScatterSeries* series2 = new QScatterSeries();
    series2->setName("Scatter");
    *series2 << QPointF(1,5) << QPointF(6,6)
        << QPointF(12,3) << QPointF(17,5);

    QChart *chart = new QChart();
    chart->addSeries(series);
    chart->addSeries(series1);
    chart->addSeries(series2);
    chart->createDefaultAxes();
```

```
chart->setTitle("Line and scatter chart");

QChartView *chartView = new QChartView(chart);
chartView->setRenderHint(QPainter::Antialiasing);

QHBoxLayout *hLay = new QHBoxLayout();
hLay->addWidget(chartView);
setLayout(hLay);

}

Widget::~Widget()
{
}
```

QLineSeries can display a line chart. QSplineSeries can display a graph chart in the form of a curve. QScatterSeries can display data in a scatter plot.

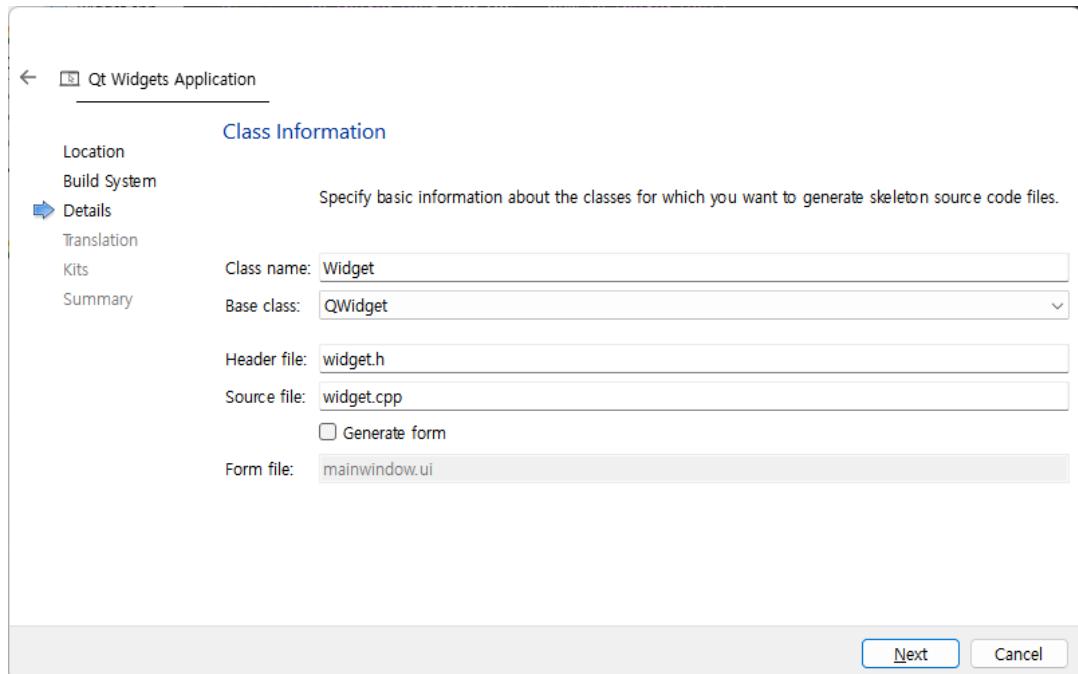
Finally, to display the above graphs in a chart, you can use the addSeries( ) member function of the QChart class. You can specify objects of QLineSeries, QSplineSeries, and QScatterSeries in the first argument of the addSeries( ) member function.



For this example, you can refer to the 00\_LineAndScatter directory.

✓ Implementing Area Chart

In this example, we will implement an Area Chart. Create a project based on Qt Widget. As in the previous example, we will not use the Form UI, so uncheck the [Generate form] box when creating the project.



Next, open the `widget.cpp` source code file and write something like this

```
#include "widget.h"
#include <QtCharts>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    window()->setMinimumSize(600, 400);

    QLineSeries *series1 = new QLineSeries();

    *series1 << QPointF(1, 5) << QPointF(3, 7) << QPointF(7, 6)
        << QPointF(9, 7) << QPointF(12, 6) << QPointF(16, 7)
        << QPointF(18, 5);
```

```
QLineSeries *series2 = new QLineSeries();

*series2 << QPointF(1, 3) << QPointF(3, 4) << QPointF(7, 3)
    << QPointF(12, 3) << QPointF(16, 4);

QAreaSeries *series = new QAreaSeries(series1, series2);
series->setName("Area Data");

QPen pen(Qt::blue); // Applying a outline
pen.setWidth(3);
series->setPen(pen);
QLinearGradient gradient(QPointF(0, 0), QPointF(0, 1));
gradient.setColorAt(0.0, 0x3cc63c);
gradient.setColorAt(1.0, 0x26f626);
series->setBrush(gradient); // Applying a gradient

QChart *chart = new QChart();
chart->legend()->hide(); // Hiding legend
chart->addSeries(series);
chart->createDefaultAxes();
chart->setTitle("Area chart");

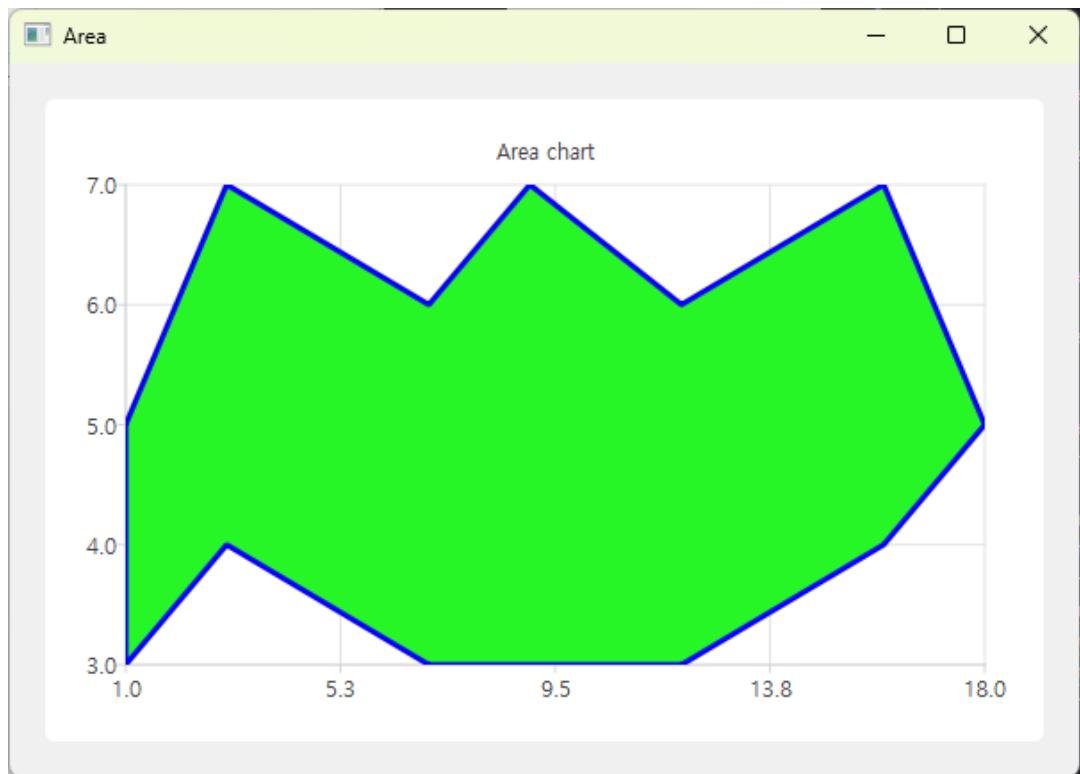
QChartView *chartView = new QChartView(chart);
chartView->setRenderHint(QPainter::Antialiasing);

QHBoxLayout *hLay = new QHBoxLayout();
hLay->addWidget(chartView);
setLayout(hLay);

}

Widget::~Widget()
{
```

To display the Area Chart, we use the QAreaSeries class. And to display data, you can use the QAreaSeries class for QLineSeries. You can pass a QLineSeries class object as an argument when declaring a QAreaSeries class object.



You can find the source code for this example in the 01\_Area directory.

- ✓ Implement the Bar Chart example

This time, let's implement a Bar Chart. Create a Qt-based project. And as in the previous example, we don't want to use UI Form, so uncheck the [Generate form] checkbox in the Detail dialog when creating the project.

Once the project is created, add the Chart module to CMakeList.txt. Then, open the widget.cpp source code and write the following code.

```
#include "widget.h"
#include <QtCharts>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    window()->setMinimumSize(600, 300);
```

```
QBarSet *set1 = new QBarSet("John");
QBarSet *set2 = new QBarSet("Jane");
QBarSet *set3 = new QBarSet("Eddy");

*set1 << 1 << 2 << 3 << 4 << 5 << 6;
*set2 << 5 << 7 << 2 << 4 << 0 << 7;
*set3 << 8 << 4 << 7 << 3 << 7 << 1;

QBarSeries *series = new QBarSeries();
series->append(set1);
series->append(set2);
series->append(set3);

QStringList categories;
categories << "Jan" << "Feb" << "Mar" << "Apr" << "May" << "Jun";
QBarCategoryAxis *axis = new QBarCategoryAxis;
axis->append(categories);

QChart *chart = new QChart();
chart->addSeries(series);
chart->setTitle("Bar chart");
chart->legend()->setVisible(true);
chart->legend()->setAlignment(Qt::AlignBottom);

chart->addAxis(axis, Qt::AlignBottom);
series->attachAxis(axis);

QChartView *chartView = new QChartView(chart);
chartView->setRenderHint(QPainter::Antialiasing);

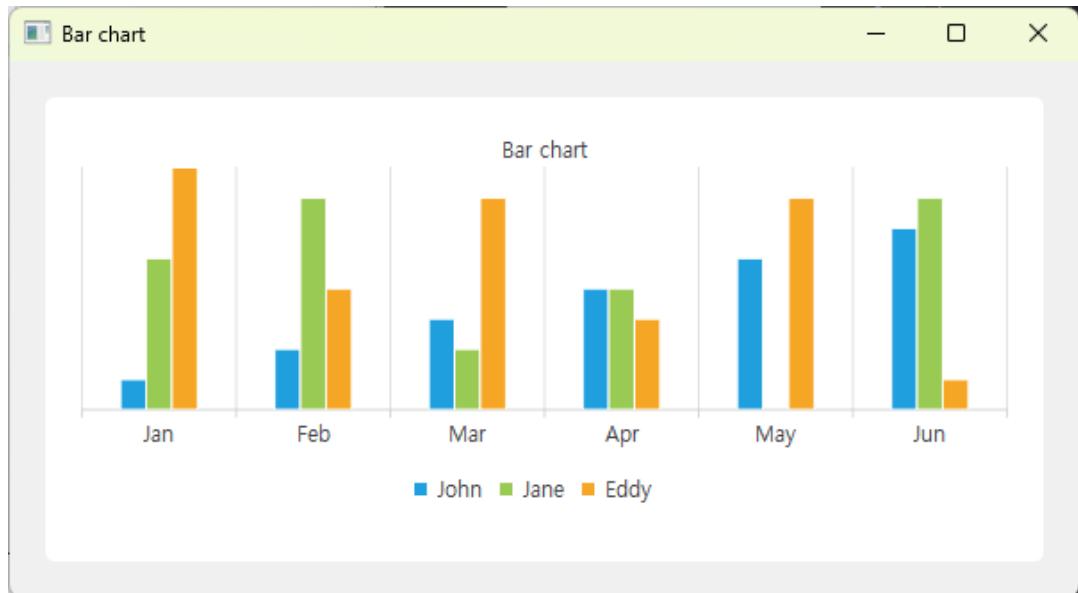
QHBoxLayout *hLay = new QHBoxLayout();
hLay->addWidget(chartView);
setLayout(hLay);

}

Widget::~Widget()
```

```
{  
}
```

To display a bar-shaped chart, we use the QBarSeries class.



You can find the source code for this example in the 02\_Bar directory.

- ✓ Implement the Pie Chart example

When creating the project, create a Qt Widget-based project in the same way as in the previous example, but without the UI Form. Once the project is created, add the Chart module to the CMakeList.txt file. Then, create the widget.cpp source code file like below.

```
#include "widget.h"  
#include <QtCharts>  
  
Widget::Widget(QWidget *parent)  
    : QWidget(parent)  
{  
    window()->setMinimumSize(600, 400);  
  
    QPieSeries *series = new QPieSeries();  
    series->append("Jane", 1); // Inserting name and ratio  
    series->append("Joe", 2);
```

```
series->append("Andy", 3);
series->append("Barbara", 4);
series->append("Axel", 2);

QPieSlice *slice = series->slices().at(1); // Selecting second item
slice->setExploded(); // Separating item
slice->setLabelVisible();
slice->setPen(QPen(Qt::darkGreen, 2));
slice->setBrush(Qt::green);

QChart *chart = new QChart();
chart->legend()->hide();
chart->addSeries(series);
chart->createDefaultAxes();
chart->setTitle("Pie chart");

QChartView *chartView = new QChartView(chart);
chartView->setRenderHint(QPainter::Antialiasing);

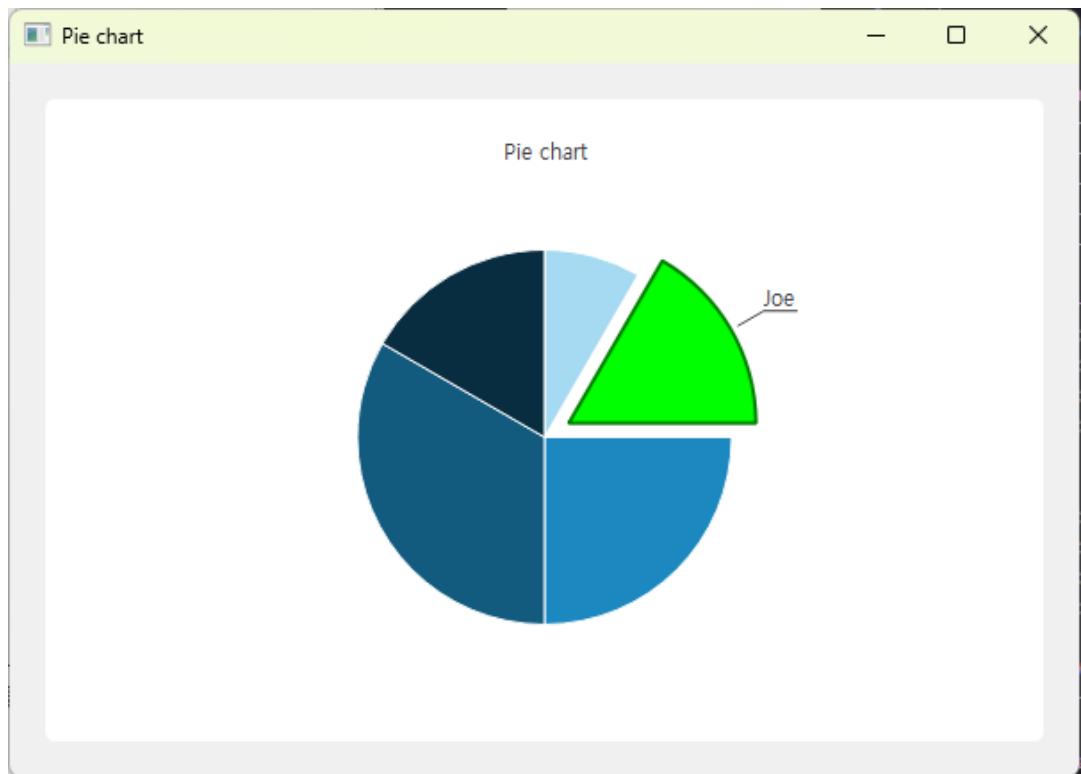
QHBoxLayout *hLay = new QHBoxLayout();
hLay->addWidget(chartView);
setLayout(hLay);

}

Widget::~Widget()
{
}
```

To display a Pie graph, we use the QPieSeries class. And to highlight a specific region of the pie, you can use the QPieSlice class.

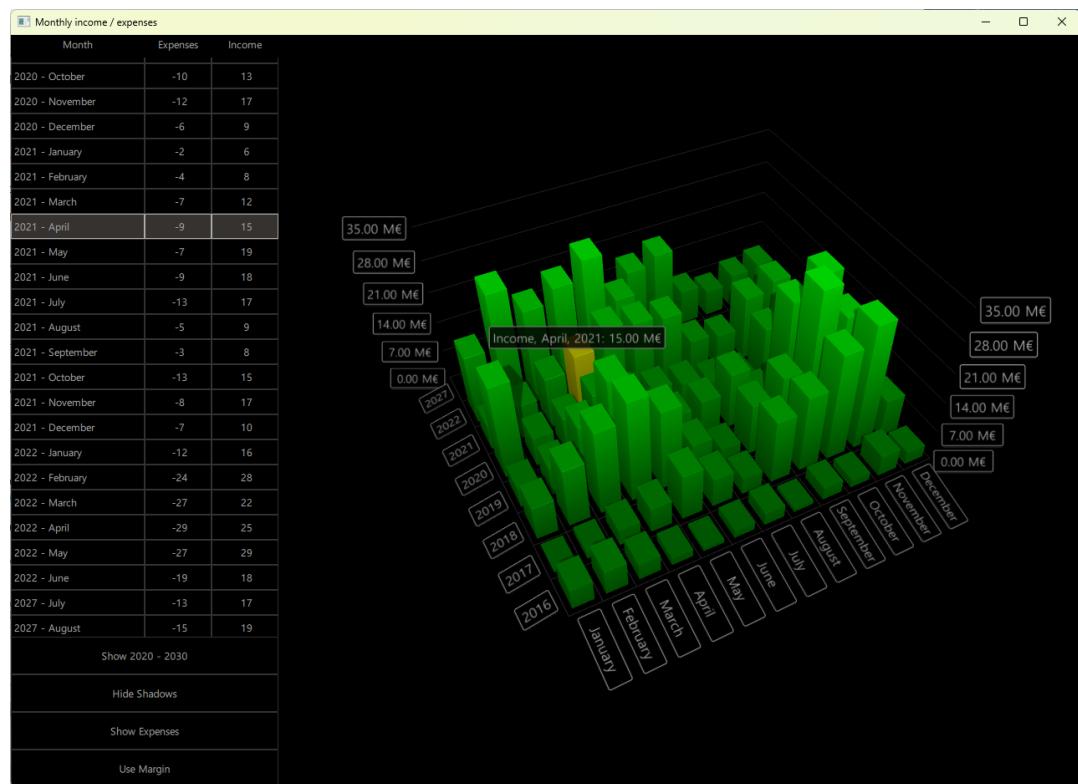
Jesus loves you.



You can find the source code for this example in the 03\_Pie directory.

## 36. Qt Data Visualization

The Qt Data Visualization module provides the ability to visualize data in 3D. You can visualize data in 3D, such as bars, scatters, surfaces (for example, displaying sea levels on land and sea), etc. The Qt Data Visualization module is based on OpenGL and uses hardware acceleration for fast rendering.



The Qt Data Visualization module uses the concept of Camera to provide the user with the ability to pan the graph up/down/left/right, zoom in/out, and click on specific data to visualize it.

For example, you can change the color of data in a particular legend.

If you are using CMake for your project, in order to use the Qt Data Visualization module, you need to add the following to your project files

```
find_package(Qt6 REQUIRED COMPONENTS DataVisualization)
target_link_libraries(mytarget PRIVATE Qt6::DataVisualization)
```

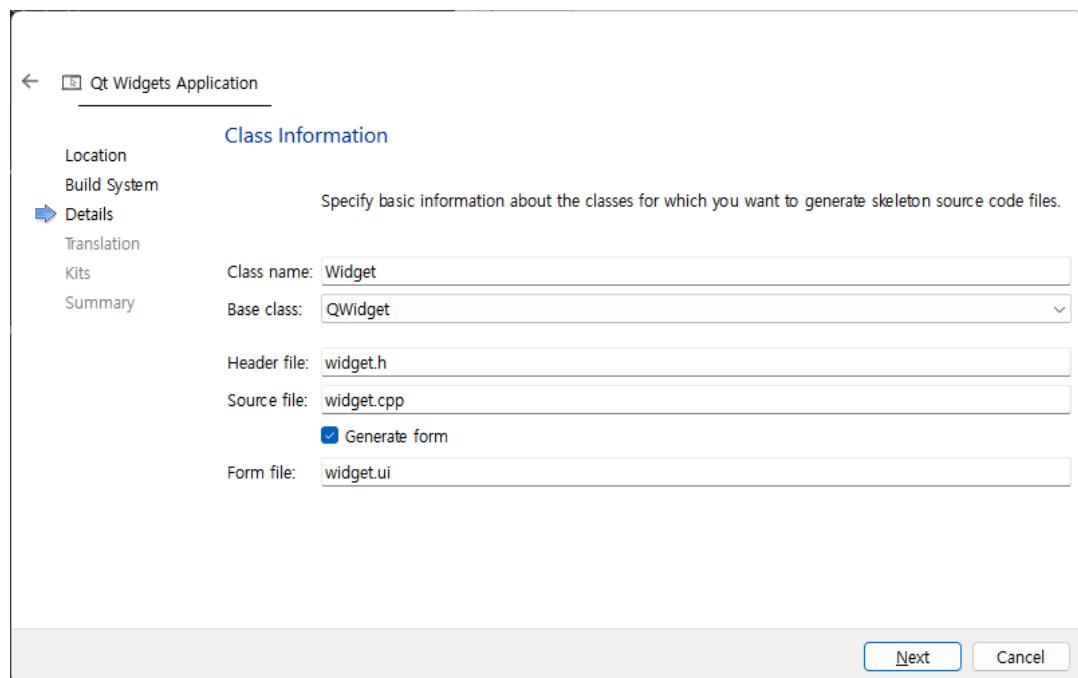
If you're using qmake, you'll need to add the following to your project file

```
QT += datavisualization
```

- ✓ Implement the Bar 3D example

The Q3DBars class allows you to display 3D Bar 3D graphs. The Q3DBars class allows you to pan the screen up/down, left/right, and zoom freely.

When creating the project, we will create a project based on Qt Widget. We don't use UI Forms in this project, so uncheck the [Generate form] checkbox in the Class Information dialog during the project creation dialog.



Once the project is created, in the CMakeList.txt file, add the DataVisualization as shown below.

```
cmake_minimum_required(VERSION 3.5)

project(00_BarExample VERSION 0.1 LANGUAGES CXX)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTOMOC ON)
```

```
set(CMAKE_AUTORCC ON)
...
find_package(Qt NAMES Qt6 Qt5 REQUIRED COMPONENTS Widgets)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS DataVisualization)
...
target_link_libraries(00_BaExample PRIVATE Qt6::Widgets)
target_link_libraries(00_BaExample PRIVATE Qt6::DataVisualization)
...
if(QT_VERSION_MAJOR EQUAL 6)
    qt_finalize_executable(00_BaExample)
endif()
```

다음으로 widget.cpp 소스코드 파일을 열어서 아래와 같이 추가한다.

```
#include "widget.h"
#include <QtDataVisualization>
#include <QHBoxLayout>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    window()->setMinimumSize(600, 400);

    QBarDataRow *data1 = new QBarDataRow;
    *data1 << 1.0f << 3.0f << 7.5f << 5.0f << 2.2f;
    QBarDataRow *data2 = new QBarDataRow;
    *data2 << 5.0f << 2.0f << 9.5f << 1.0f << 7.2f;

    QBar3DSeries *series = new QBar3DSeries;
    series->dataProxy()->addRow(data1);
    series->dataProxy()->addRow(data2);

    Q3DBars *bars = new Q3DBars;
    bars->scene()->activeCamera()->setCameraPreset(
        Q3DCamera::CameraPresetFront);
    bars->rowAxis()->setRange(0,4);
```

```
bars->columnAxis()->setRange(0,4);
bars->addSeries(series);

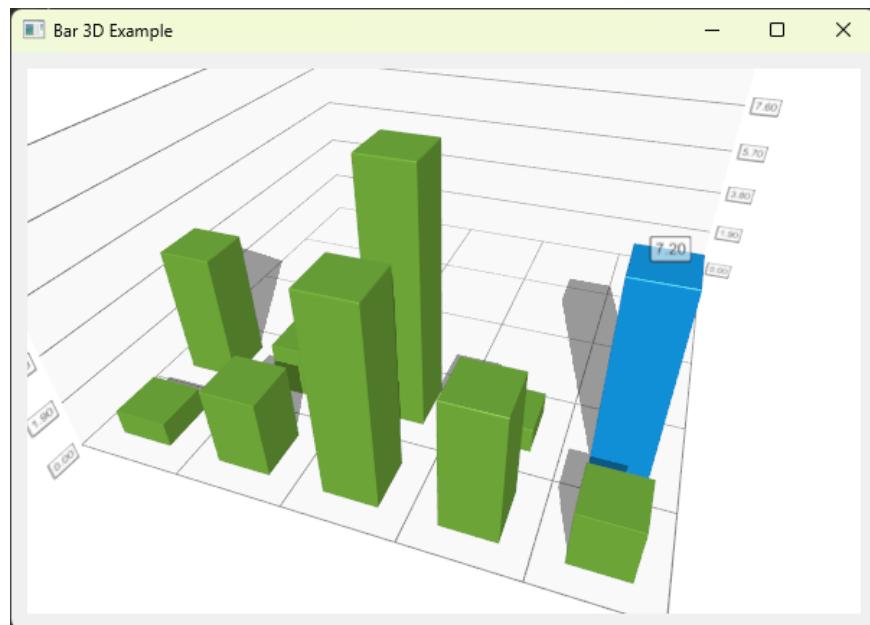
QHBoxLayout *hLay = new QHBoxLayout();
hLay->addWidget(QWidget::createWindowContainer(bars));
setLayout(hLay);
}

Widget::~Widget()
{
}
```

To add data to a 3D Bar, you can use the QBarDataRow class and QBar3DSeries.

You can add an object of the QBarDataRow class with data inserted by using the addRow() member function provided by the QBar3DSeries class.

To set the range of a row, the Q3DBars class can use the rowAxis()->setRange( ) member function. To set the range of a column, you can use the columnAxis()->setRange( ) member function.



You can find the source code for this example in the 00\_BarExample directory.

- ✓ Implement the Scatter 3D example

A scatter graph is a type of graph that is scattered, such as displaying points at specific x and y coordinates. Qt provides the Q3DScatter class to represent scatter graphs in 3D.

When you create a project, you create a project based on the Qt Widget. As in the previous example, we do not use the UI Form. And once the project is created, you need to add a DataVisualization in the CMakeList.txt file.

Then open the widget.cpp source code file and add the following code.

```
#include "widget.h"
#include <QtDataVisualization>
#include <QHBoxLayout>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    window()->setMinimumSize(600, 400);

    QScatter3DSeries *series = new QScatter3DSeries;
    QScatterdataArray data;
    data << QVector3D(0.5f, 0.5f, 0.5f)
        << QVector3D(-0.3f, -0.5f, -0.4f)
        << QVector3D(0.0f, -0.3f, 0.2f);

    Q3DScatter *scatter = new Q3DScatter;

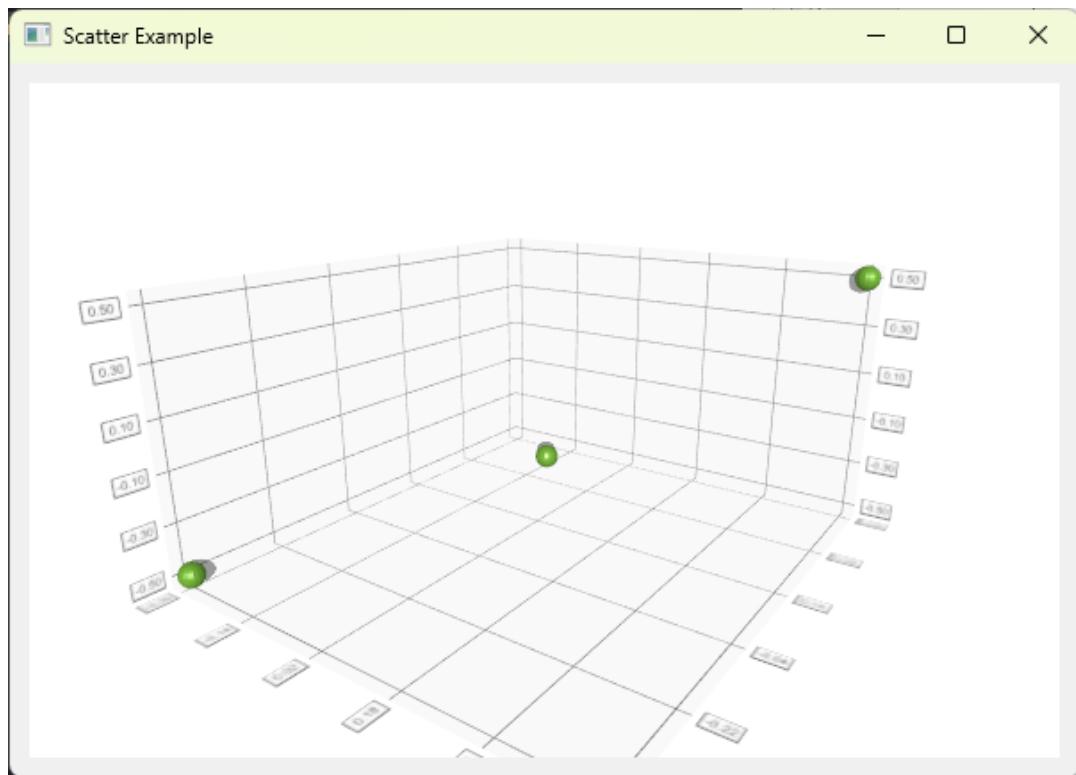
    scatter->scene()->activeCamera()->setCameraPreset(
        Q3DCamera::CameraPresetFront);
    series->dataProxy()->addItems(data);
    scatter->addSeries(series);

    QHBoxLayout *hLay = new QHBoxLayout();
    hLay->addWidget(QWidget::createWindowContainer(scatter));
    setLayout(hLay);
}

Widget::~Widget()
{}
```

In the Q3DScatter class, you can add data using the QScatter3DSeries and

QScatterdataArray class.



The source code for this example can be found in the 01\_ScatterExample directory.

#### ✓ Surface 3D Example

The Q3DSurface class allows you to display surfaces (cross sections). For example, you can visualize the surface of land when displaying terrain. In this example, we will cover a simple way to use the Q3DSurface class.

Like the previous example, we will create a project that does not use UI Forms. Once the project is created, add the DataVisualization module to the CMakeList.txt file.

Then, open the widget.cpp source code file and write the following lines

```
#include "widget.h"
#include <QtDataVisualization>
#include <QHBoxLayout>

Widget::Widget(QWidget *parent): QWidget(parent)
{
    window()->setMinimumSize(600, 400);
```

```
Q3DSurface *surface = new Q3DSurface();
QSurfacedataArray *data = new QSurfacedataArray;

QSurfaceDataRow *dataRow1 = new QSurfaceDataRow;
*dataRow1 << QVector3D(0.0f, 0.1f, 0.5f) << QVector3D(1.0f, 0.5f, 0.5f);
QSurfaceDataRow *dataRow2 = new QSurfaceDataRow;
*dataRow2 << QVector3D(0.0f, 1.8f, 1.0f) << QVector3D(1.0f, 1.2f, 1.0f);

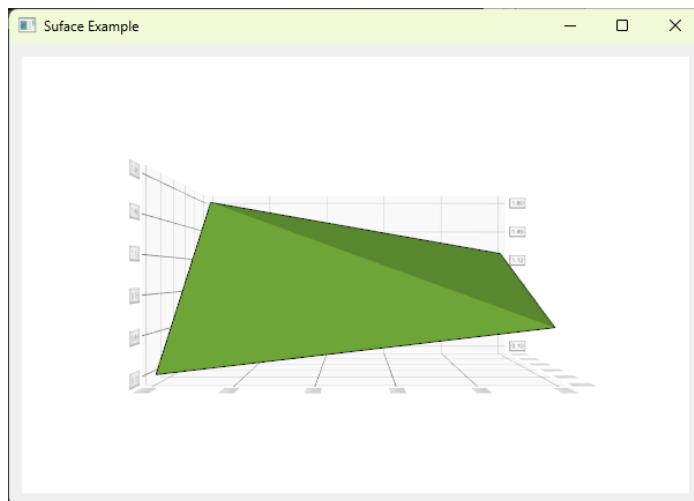
*data << dataRow1 << dataRow2;

QSurface3DSeries *series = new QSurface3DSeries;
series->dataProxy()->resetArray(data);
surface->addSeries(series);

QHBoxLayout *hLay = new QHBoxLayout();
hLay->addWidget(QWidget::createWindowContainer(surface));
setLayout(hLay);
}

Widget::~Widget()
{
```

To display the graph in the form of a Surface, you can use the Q3DSurface class. And to add data, you can use the QSurfacedataArray and QSurfaceDataRow classes.



You can find the source code for this example in the 02\_SurfaceExample directory.

- ✓ Example of displaying a contour 3D Surface using image file

In this example, we'll implement an example of displaying contour lines using a Height Map, as shown in the image below.

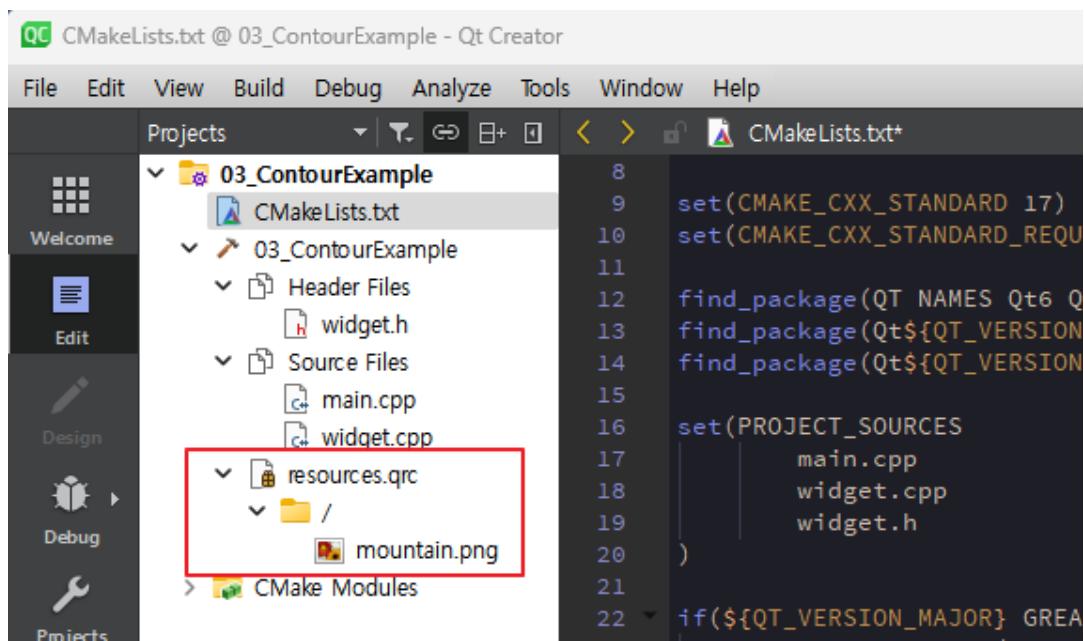


A Height Map is a Texture that stores the height values of a terrain while looking at it from above. In the field of geography, photographs taken from satellites are converted into Height Map images and made available to the public.

The Height Map is labeled with the lowest height value of 0 and the highest value of 255.

When creating the project, we will create a project based on the Qt Widget as in the previous example. Once the project is created, add the DataVisualization module to CMakeList.txt.

Then, register the Height Map image that we will use in the example as a resource. The Height Map image is located in the 03\_CondourExample directory and is named mountain.png. Add the resource to your project, and then register the mountain.png file to the resource.



Next, in your widget.cpp source code file, write the following

```
#include "widget.h"
#include <QtDataVisualization>
#include <QHBoxLayout>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    window()->setMinimumSize(800, 700);

    Q3DSurface *surface = new Q3DSurface;
    surface->scene()->activeCamera()->setCameraPreset(
        Q3DCamera::CameraPresetFront);

    QImage heightMapImage(":/mountain.png");
    QHeightMapSurfaceDataProxy *heightMapProxy
        = new QHeightMapSurfaceDataProxy(heightMapImage);

    QSurface3DSeries *series = new QSurface3DSeries(heightMapProxy);

    series->setItemLabelFormat(QStringLiteral("@xLabel, @zLabel): @yLabel"));
    heightMapProxy->setValueRanges(34.0f, 40.0f, 18.0f, 24.0f);
```

```
surface->axisX()->setLabelFormat("0.1f N");
surface->axisZ()->setLabelFormat("0.1f E");

surface->axisX()->setRange(34.0f, 40.0f);
surface->axisY()->setAutoAdjustRange(true);
surface->axisZ()->setRange(18.0f, 24.0f);

surface->addSeries(series);
surface->setGeometry(50,50,600,400);

QHBoxLayout *hLay = new QHBoxLayout();
hLay->addWidget(QWidget::createWindowContainer(surface));
setLayout(hLay);
}

Widget::~Widget()
{
}
```

This example is for displaying a 3D Surface using an image taken for contour extraction, as shown in the figure below.

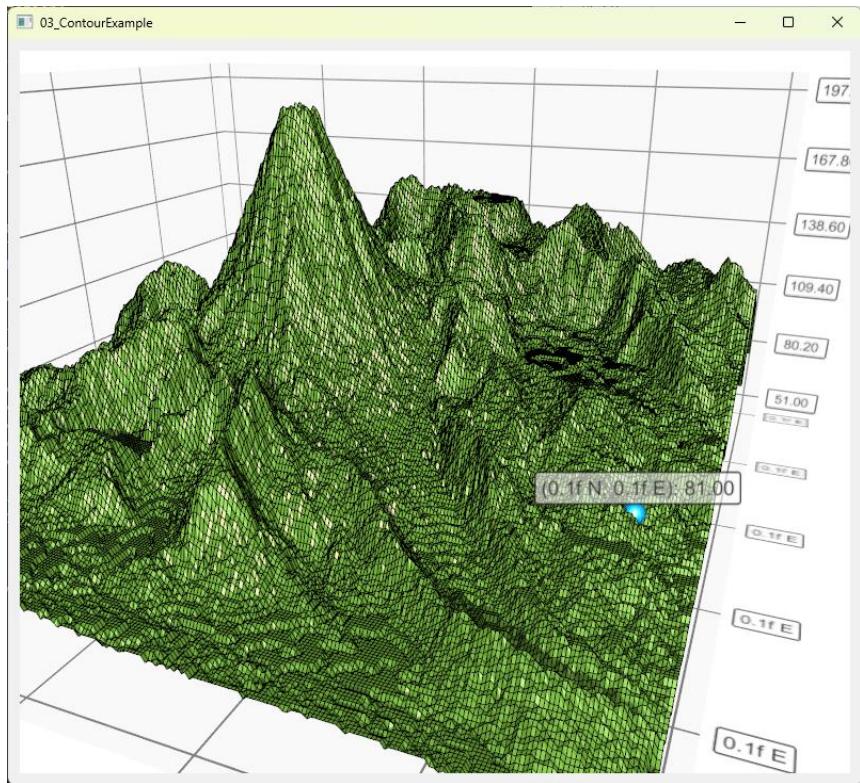
The figure on the left is an image taken for contour extraction. The higher the white color, the higher the height, i.e., higher height means higher elevation. Let's visualize a photo like this as a 3D graph using Q3DSurface. The following figure shows the example running.

It loads the Height map image that you have registered as a resource in a QImage class object. It then passes the QImage object to a QHeightMapSurfaceDataProxy class object.

This class handles the height data for the Surface in order to visualize the image as a Surface.

The QSurface3DSeries class can specify how to display the data when the user clicks on specific data in the graph. And you can specify the ranges through the setValueRanges( ) function.

Jesus loves you.



The source code for the example above can be found in the 03\_ContourExample directory.

## 37. Inter Process Communication

Inter Process Communication (IPC) is a method for communicating between processes that exist within the same system. In this chapter, we'll look at the following IPC methods

### ① Unix Domain Socket and Named pipe

Inter process communication using Unix Domain Sockets (UDS) in Linux and Named pipe names in MS Windows.

### ② Communication with external processes using the QProcess class.

Used to run an external process or get a result

### ③ Share memory with external programs using Shared Memory.

Use shared memory between external programs using the QSharedMemory class.

## 37.1. Unix Domain Socket and Named pipe

UDS (Unix Domain Socket) is one of the interprocess communication methods used by the Linux platform. Name pipe is an extension of UDS and is an interprocess communication method used on the MS Windows platform. The purpose is the same, but the name is different.

However, in Qt, you can use QLocalSocket and QLocalServer classes regardless of platform, whether you are using Linux or MS Windows. They are also available on MacOS.

To use UDS or Name pipe, you need to use the Network module in your project files. If you're using a CMake project, you'll need to add it like this

```
find_package(Qt6 REQUIRED COMPONENTS Network)
target_link_libraries(mytarget PRIVATE Qt6::Network)
```

If you're using qmake, you'll need to add the following to your project file

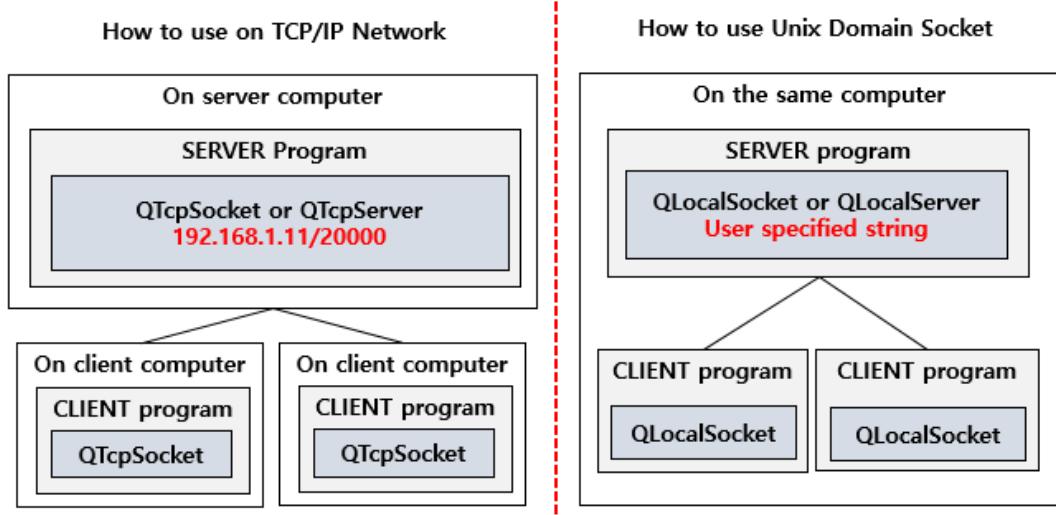
```
QT += network
```

The QLocalSocket and QLocalServer classes use a specific string specified by the user as a way to identify the other party.

QLocalSocket implements functionality for interprocess communication. In UDS, it is possible to send/receive with the concept of Server / Client.

This allows 1:N communication between Server and Client. QLocalServer provides functions that make it easy to implement the functions required for server implementation.

To communicate with a specific server in the network, IP and PORT are used as identifiers. However, a user-defined string (word) can be used as an identifier.



The QLocalSocket and QLocalServer classes make it easy to implement asynchronous data receiving modules using Signal and Slot. This makes it easy to implement a receiving module without using Threads.

Just as servers on TCP/IP networks use the `listen()` member function provided by the `QTcpServer` class to listen for client connections, the `QLocalServer` class can use the `listen()` member function to listen for client connections.

```
...
server = new QLocalServer(this);

if (!server->listen("MyServer")) {
    qDebug() << "Server error : " << server->errorString();
    ...
}

connect(server, SIGNAL(newConnection()), this, SLOT(clientConnection()));
...
```

In the example above, the `connect()` member function fires the `newConnection()` signal when a new client connects. It is associated with the `clientConnection()` Slot function so that it can be called when this signal is raised.

The example source code below uses the `QLocalSocket` class to signal when a client connects to the server, receives a message from the server, or when an unexpected error occurs.

The `connectToServer()` function at the end of the code below provides the ability to

connect to the server. The first argument to this function is the unique name of the server.

Here, we've named it "MyServer", but you can change it to any string you want.

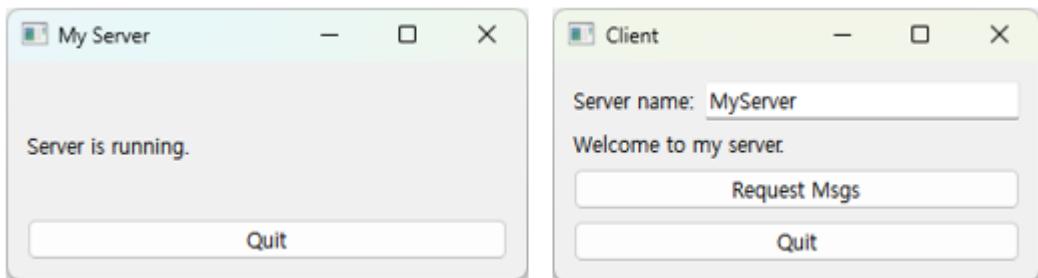
```
...
socket = new QLocalSocket(this);

connect(socket, SIGNAL(readyRead()), this, SLOT(readData()));

connect(socket, SIGNAL(error(QLocalSocket::LocalSocketError)),
        this, SLOT(sockError(QLocalSocket::LocalSocketError)));

socket->connectToServer("MyServer");
...
```

Next, let's take a closer look at the actual Server and Client on the UDS.



The example on the left is the Server and the example on the right is the Client.

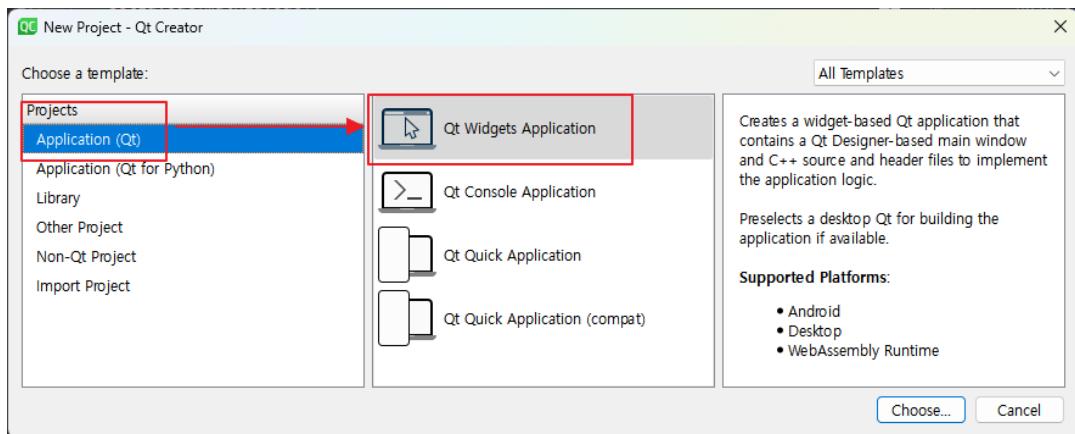
Run the Server first and then run the Client. After Client is executed, click the [Request Msgs] button to request a connection with Server.

The server then connects to the client and sends a message to the client. The server then connects to the client and sends a message to the client.

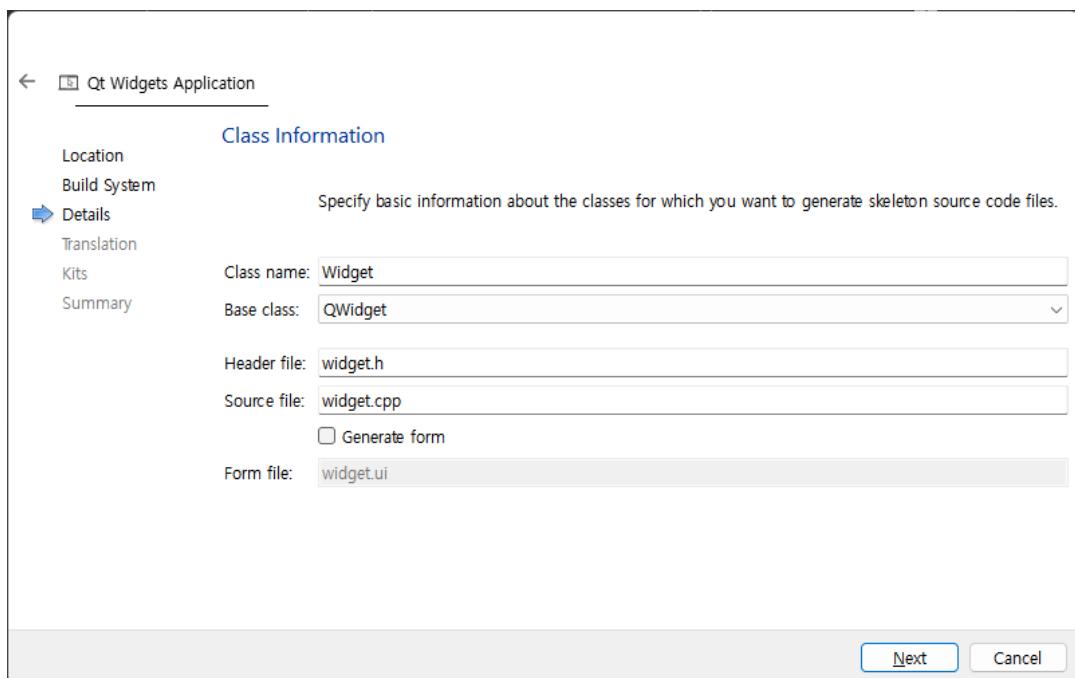
In this chapter, we will implement two examples as shown in the figure above. Let's implement the Server example first, then the Client example.

- ✓ Implement the Server example

When creating a project, select a Qt Widget-based application.



We will not be using UI Forms in this project, so if the [Generate form] item is checked in the Class Information dialog during project creation, it should be unchecked.



Once the project is created, open the CmakeLists.txt file and add the Network module as shown below.

```
cmake_minimum_required(VERSION 3.5)
...
find_package(Qt NAMES Qt6 Qt5 REQUIRED COMPONENTS Widgets)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Widgets)
find_package(Qt${QT_VERSION_MAJOR} REQUIRED COMPONENTS Network)
```

```
set(PROJECT_SOURCES
    main.cpp
    widget.cpp
    widget.h
)
...
target_link_libraries(00_LocalServer PRIVATE Qt${QT_VERSION_MAJOR}::Widgets)
target_link_libraries(00_LocalServer PRIVATE Qt${QT_VERSION_MAJOR}::Network)
...
if(QT_VERSION_MAJOR EQUAL 6)
    qt_finalize_executable(00_LocalServer)
endif()
```

Next, open the widget.h header file and write something like the following.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QLabel>
#include <QPushButton>
#include <QLocalServer>

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private slots:
    void clientConnection();

private:
    QLabel *statusLabel;
```

```
QPushButton *quitButton;
QLocalServer *server;
QStringList sendMsgs;
bool msgKind;

};

#endif // WIDGET_H
```

The clientConnection( ) Slot function is called when the newConnection( ) Signal is fired, so when a new Client connects, the clientConnection( ) Slot function is called.

Next, open the widget.cpp source code file and write the following code.

```
#include "widget.h"
#include <QLocalSocket>
#include <QVBoxLayout>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    window()->setMinimumSize(300, 100);

    statusLabel = new QLabel;
    statusLabel->setWordWrap(true);
    quitButton = new QPushButton(tr("Quit"));
    quitButton->setAutoDefault(false);

    server = new QLocalServer(this);
    if (!server->listen("MyServer")) {
        qDebug() << "Server error : "
              << server->errorString();
        close();
        return;
    }

    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
    connect(server, SIGNAL(newConnection()), this, SLOT(clientConnection()));
}
```

```
statusLabel->setText(tr("Server is running."));  
QVBoxLayout *mainLayout = new QVBoxLayout;  
mainLayout->addWidget(statusLabel);  
mainLayout->addWidget(quitButton);  
setLayout(mainLayout);  
  
setWindowTitle(tr("My Server"));  
}  
  
void Widget::clientConnection()  
{  
    QByteArray writeData;  
    if(msgKind) {  
        writeData.append("Welcome to my server.");  
        msgKind = false;  
    } else {  
        writeData.append("Who are you");  
        msgKind = true;  
    }  
  
    QLocalSocket *clientConnection =  
        server->nextPendingConnection();  
  
    connect(clientConnection, SIGNAL(disconnected()),  
            clientConnection, SLOT(deleteLater()));  
  
    clientConnection->write(writeData, writeData.size());  
    clientConnection->flush();  
    clientConnection->disconnectFromServer();  
}  
  
Widget::~Widget()  
{  
}
```

In the constructor of this class, we declare a QLocalServer class object and give the server a unique name of "MyServer".

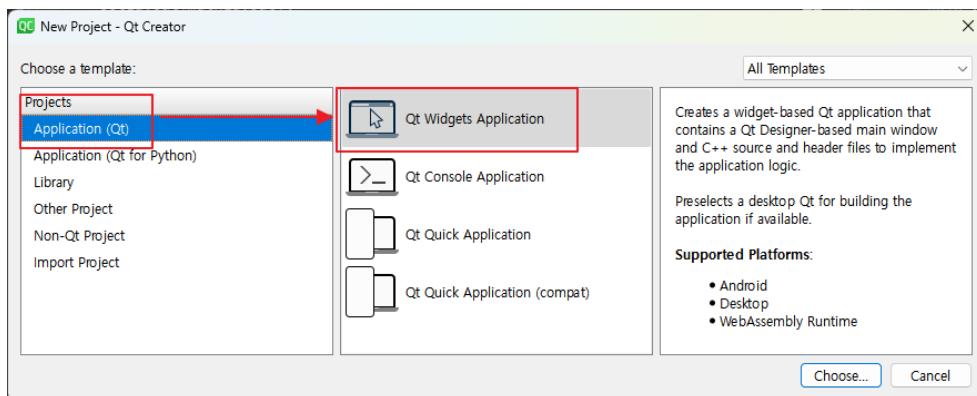
Jesus loves you.

In the clientConnection( ) Slot function above, when a client connects, we send a message to the client and then disconnect the connection.

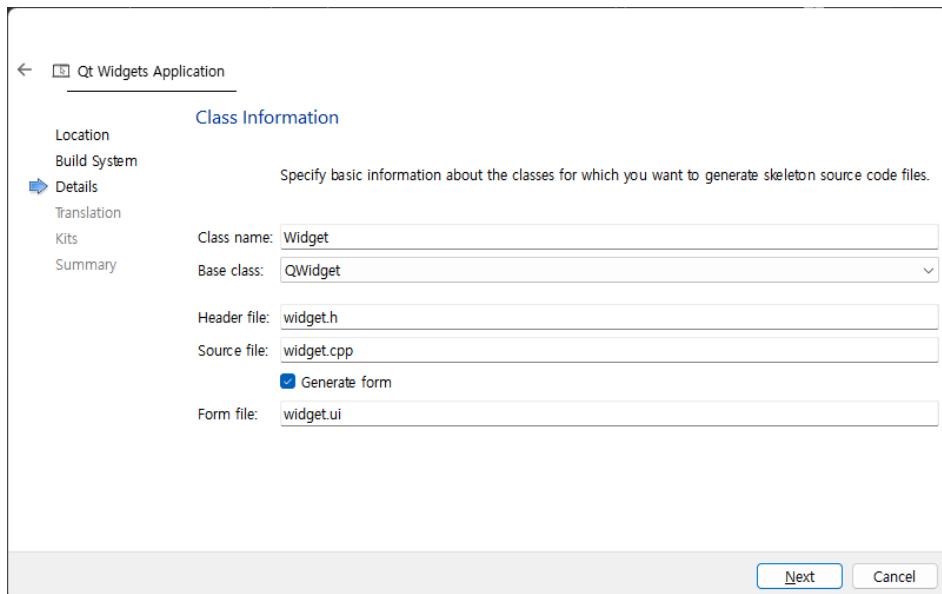
That's it for the Server example. You can find the complete source code for this example in the 00\_LocalServer directory. Next, let's implement the Client example.

## ✓ Client Example

When creating a project, select a Qt Widget-based application.



Client projects do not use UI Forms, so if the [Generate form] item is checked in the Class Information dialog during project creation, uncheck it.



Once the project is created, open the CmakeLists.txt file and add the Network module as shown below.

Next, open the widget.h header file and write the following code.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QLabel>
#include <QLineEdit>
#include <QPushButton>
#include <QLocalSocket>

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private slots:
    void requestNewMsg();
    void readData();
    void sockError(QLocalSocket::LocalSocketError socketError);

private:
    QLabel *hostLabel;
    QLineEdit *hostLineEdit;
    QLabel *statusLabel;
    QPushButton *reqButton;
    QPushButton *quitButton;

    QLocalSocket *socket;
    quint16 blockSize;

};

#endif // WIDGET_H
```

Next, open the widget.cpp file and write something like this

```
#include "widget.h"
#include <QGridLayout>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
{
    hostLabel = new QLabel(tr("Server name:"));
    hostLineEdit = new QLineEdit("MyServer");
    statusLabel = new QLabel(tr(""));
    reqButton = new QPushButton(tr("Request Msgs"));
    quitButton = new QPushButton(tr("Quit"));

    socket = new QLocalSocket(this);
    connect(reqButton, SIGNAL(clicked()), this, SLOT(requestNewMsg()));
    connect(quitButton, SIGNAL(clicked()), this, SLOT(close()));
    connect(socket, SIGNAL(readyRead()), this, SLOT(readData()));
    connect(socket, SIGNAL(error(QLocalSocket::LocalSocketError)), this, SLOT(sockError(QLocalSocket::LocalSocketError)));

    QGridLayout *mainLayout = new QGridLayout;
    mainLayout->addWidget(hostLabel, 0, 0);
    mainLayout->addWidget(hostLineEdit, 0, 1);
    mainLayout->addWidget(statusLabel, 2, 0, 1, 2);
    mainLayout->addWidget(reqButton, 3, 0, 1, 2);
    mainLayout->addWidget(quitButton, 4, 0, 1, 2);
    setLayout(mainLayout);

    setWindowTitle(tr("Client"));
    hostLineEdit->setFocus();
}

void Widget::requestNewMsg()
{
    reqButton->setEnabled(false);
    socket->connectToServer(hostLineEdit->text());
}

void Widget::readData()
{
    statusLabel->setText(socket->readAll());
```

```
    reqButton->setEnabled(true);
}

void Widget::sockError(QLocalSocket::LocalSocketError socketError)
{
    switch (socketError) {
        case QLocalSocket::ServerNotFoundError:
            qDebug() << "ServerNotFoundError";
            break;
        case QLocalSocket::ConnectionRefusedError:
            qDebug() << "ConnectionRefusedError";
            break;
        case QLocalSocket::PeerClosedError:
            qDebug() << "PeerClosedError";
            break;
        default:
            qDebug() << "Error : " << socket->errorString();
    }

    reqButton->setEnabled(true);
}

Widget::~Widget()
{
```

The requestNewMsg( ) Slot function is called when the [Request Msgs] button is clicked.

The readData( ) Slot function is called when receiving a message sent by the server, and the sockError( ) Slot function is called when an error occurs. You can find the source code for this example in the 01\_LocalSocket directory.

## 37.2. QProcess

The QProcess class provides the ability to run external programs and get results from within the application you implement.

For example, in Linux, the "/bin/ls" command prints directory and file information. To use "/bin/ls" to print detailed directory and file information in the "/usr" directory, rather than the current directory, you can use the following arguments

```
/bin/ls -ltr /usr
```

As shown in the example above, you can use the QProcess class to print directory and file information inside the /usr directory from the Linux terminal, as shown below.

```
QString program = "/bin/ls";
QStringList arguments;
arguments << "-ltr" << "/usr";

QProcess *myProcess = new QProcess(parent);
myProcess->start(program, arguments);
```

As shown in the example above, you can run an external program by executing the start() member function of the QProcess class.

The first argument is the name of the program (or filename) to run. The second argument accepts any options that may be available after the program; if there are no options, only the first argument can be used. The result of the execution by QProcess can be obtained by Signal and Slot. The following example shows how to use them in detail.

- ✓ Example using the QProcess class

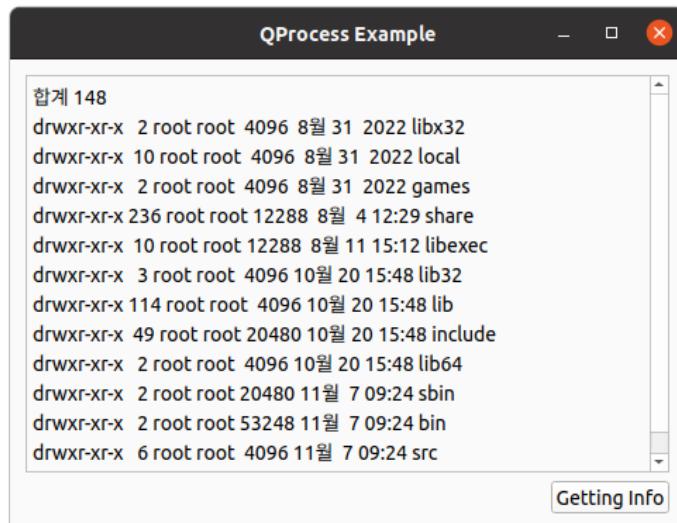
In this example, we'll use the QProcess class to run an external program, get the results, and display the results in a GUI widget. On Linux, run the command "/bin/ls -ltr /usr" to get the following result.

```
qtdev@linux1:~$ ls -ltr /usr
합계 120
drwxr-xr-x 10 root root 4096 2월 10 09:12 local
```

Jesus loves you.

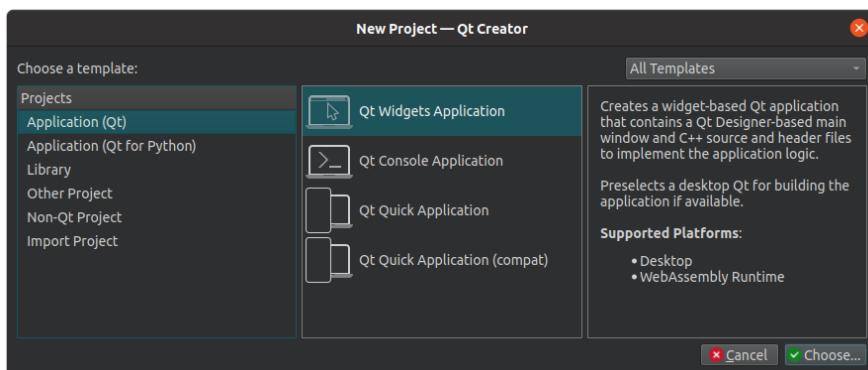
```
drwxr-xr-x  2 root root  4096  2월 10 09:14 games
drwxr-xr-x  3 root root  4096  2월 22 13:14 lib32
drwxr-xr-x 137 root root  4096  2월 22 13:23 lib
...
...
```

As shown above, the "-ltr" option used with the "/bin/ls" command prints detailed file and directory information. The /usr option is the name of the directory to be printed. Here is a screen shot of the example

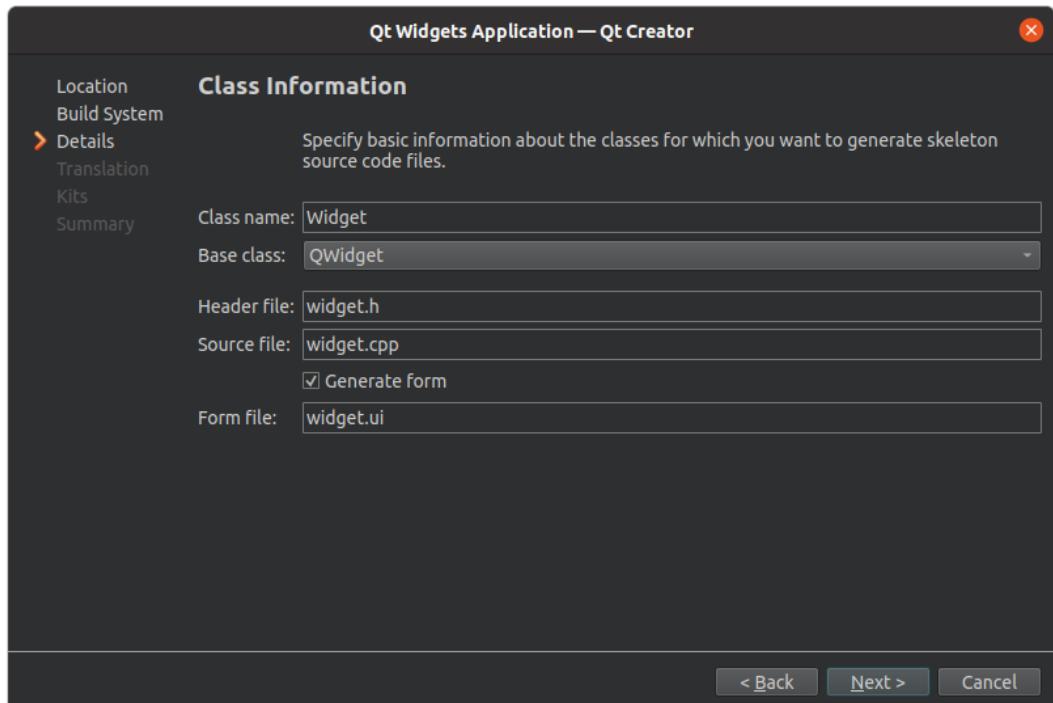


As shown in the figure above, clicking the [Getting Info] button at the bottom executes the external command and gets the result, and then displays the result in the QTextEdit window.

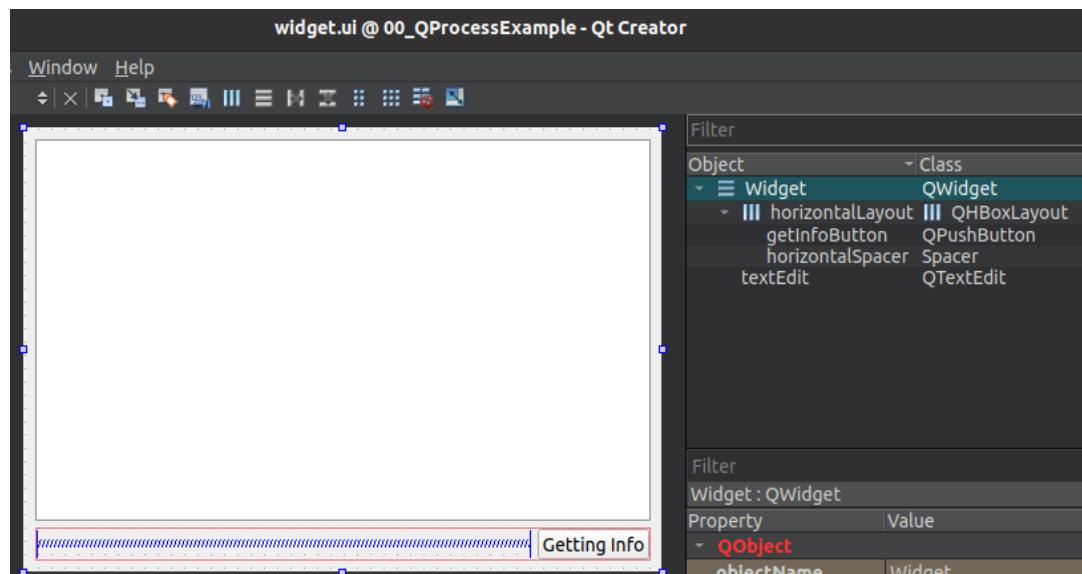
When creating a project, create a class that inherits from QWidget.



In this project, we'll be using a UI Form, so check the [Generate form] box.



Once you have finished creating the project, open the `widget.ui` file and place the widgets as shown in the image below.



Next, open the `widget.h` header file and write something like this

```
#ifndef WIDGET_H  
#define WIDGET_H
```

```
#include <QWidget>
#include <QProcess>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
    QProcess *m_process;

private slots:
    void getInfoButton();
    void finished(int exitCode, QProcess::ExitStatus exitStatus);
    void readyReadStandardError();
    void readyReadStandardOutput();
    void started();
};

#endif // WIDGET_H
```

The finished( ) slot function is called when the external program thread is completed using the QProcess class. The readyReadStandardError( ) slot function is called when an error occurs after executing the external program using the QProcess class.

The readyReadStandardOutput( ) slot function gets the result of the execution, and the started( ) slot function is called when the external program is executed by QProcess. The following is the source code of widget.cpp.

```
#include "widget.h"
#include "./ui_widget.h"

Widget::Widget(QWidget *parent)
    : QWidget(parent)
    , ui(new Ui::Widget)
```

```
{  
    ui->setupUi(this);  
    connect(ui->getInfoButton, &QPushButton::pressed,  
           this,                 &Widget::getInfoButton);  
  
    m_process = new QProcess();  
    connect(m_process, SIGNAL(finished(int,QProcess::ExitStatus)),  
            this,      SLOT(finished(int,QProcess::ExitStatus)));  
    connect(m_process, SIGNAL(readyReadStandardError()),  
            this,      SLOT(readyReadStandardError()));  
    connect(m_process, SIGNAL(readyReadStandardOutput()),  
            this,      SLOT(readyReadStandardOutput()));  
    connect(m_process, SIGNAL(started()),  
            this,      SLOT(started()));  
}  
  
void Widget::getInfoButton()  
{  
    QString program = "/bin/ls";  
    QStringList arguments;  
    arguments << "-ltr" << "/usr";  
  
    m_process->start(program, arguments);  
}  
  
void Widget::finished(int exitCode,  
                      QProcess::ExitStatus exitStatus)  
{  
    qDebug() << "Exit Code :" << exitCode;  
    qDebug() << "Exit Status :" << exitStatus;  
}  
  
void Widget::readyReadStandardError()  
{  
    qDebug() << Q_FUNC_INFO << "ReadyError";  
}  
  
void Widget::readyReadStandardOutput()  
{  
    QByteArray buf = m_process->readAllStandardOutput();  
  
    ui->textEdit->setText(buf);  
}
```

```
{  
  
void Widget::started()  
{  
    qDebug() << "Proc Started";  
}  
  
Widget::~Widget()  
{  
    delete ui;  
}
```

As shown in the example above, we declare an object of class QProcess in the class constructor and connect the Signal and Slot provided by QProcess.

When the [Getting Info] button is clicked on the GUI, the getInfoButton( ) Slot function is called and the external program is executed using the start( ) member function of the QProcess class.

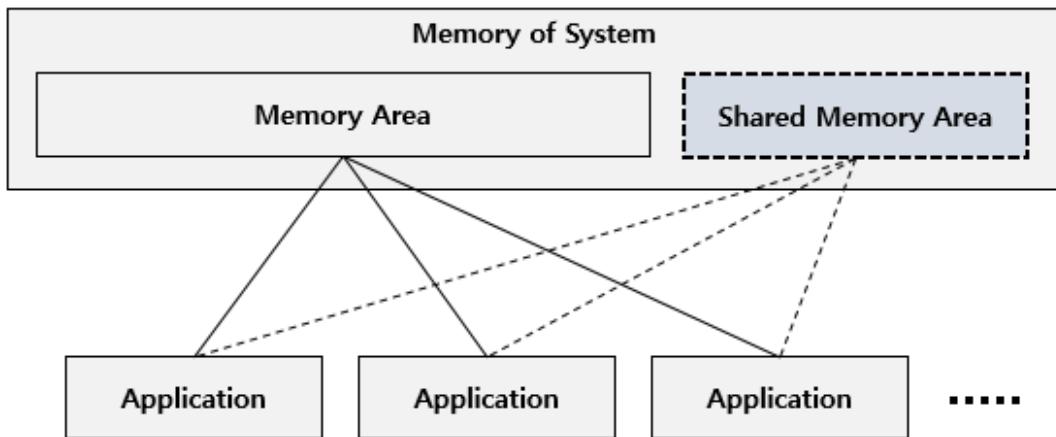
In the readyReadStandardOutput( ) function, the readAllStandardOutput( ) member function of the QProcess class gets the result and saves the result to buf. It then outputs the contents of the buf to the QTextEdit widget.

The source code for this example can be found in the 00\_QProcessExample directory.

### 37.3. Shared Memory

Shared Memory allows you to use a shared memory area for the purpose of exchanging data between programs on the same system.

Qt provides the `QSharedMemory` class for using shared memory areas between applications.



As shown in the figure above, each application uses a medium to access a specific memory area in order to read/write to the shared memory area.

In Qt, a unique KEY value can be used to read or write data to a shared memory area. The following example source code shows how to set a key value.

```
QString key = QString("qt-dev.com");
QSharedMemory *m_sharedMemory = new QSharedMemory(key);
```

As shown in the example above, you can specify the shared KEY value as an argument when declaring an object of class QSharedMemory.

To refer to the set KEY value, you can use the key( ) member function of the QSharedMemory class. The following is how to write data to the shared memory area.

```
QString key = QString("qt-dev.com");
QSharedMemory *m_sharedMemory = new QSharedMemory(key);

QBuffer buffer;
...
m_sharedMemory->lock();
```

```
char *to = (char*)m_sharedMemory->data();
const char *from = buffer.data().data();
memcpy(to, from, qMin(m_sharedMemory->size(), size));

m_sharedMemory->unlock();
```

As shown in the example above, before writing data to a shared memory area, we can lock( ) to prevent external references to it.

Then, we can use the data( ) function to get the address of the shared memory area and use memcpy( ) to write the data.

Next, to read data from the shared memory area, you can use the following functions.

```
...
QBuffer buffer;
QDataStream in(&buffer);
QImage image;

m_sharedMemory->lock();
buffer.setData((char*)m_sharedMemory->constData(),
               m_sharedMemory->size());

buffer.open(QBuffer::ReadOnly);
in >> image;
m_sharedMemory->unlock();

m_sharedMemory->detach();
ui->label->setPixmap(QPixmap::fromImage(image));
...
```

The QSharedMemory class is very easy to use, as shown above.

Let's implement it with a real-world example of writing/reading data to a shared memory area.

- ✓ Example of writing to shared memory

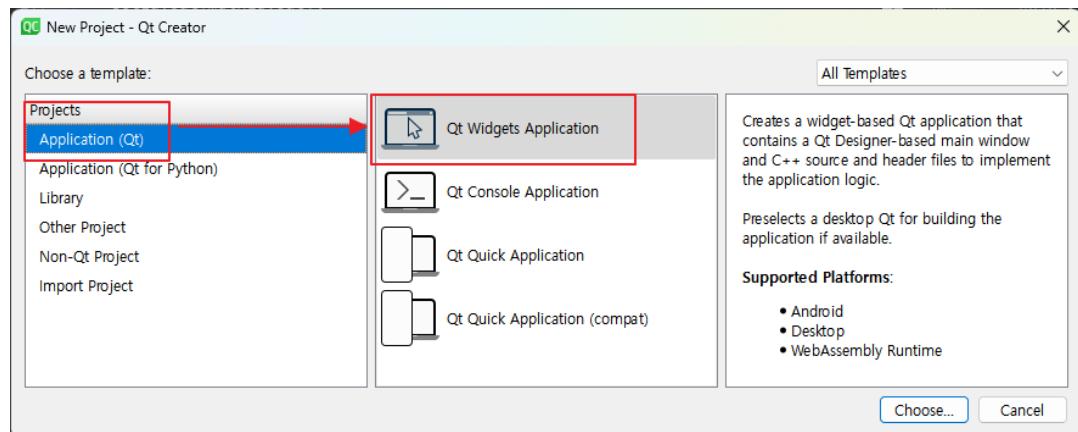
We will implement two separate applications. The first application will set the KEY value to "qt-dev.com", read the image file and display it on the GUI. It will then write the binary data of this image to a shared memory area.

Jesus loves you.

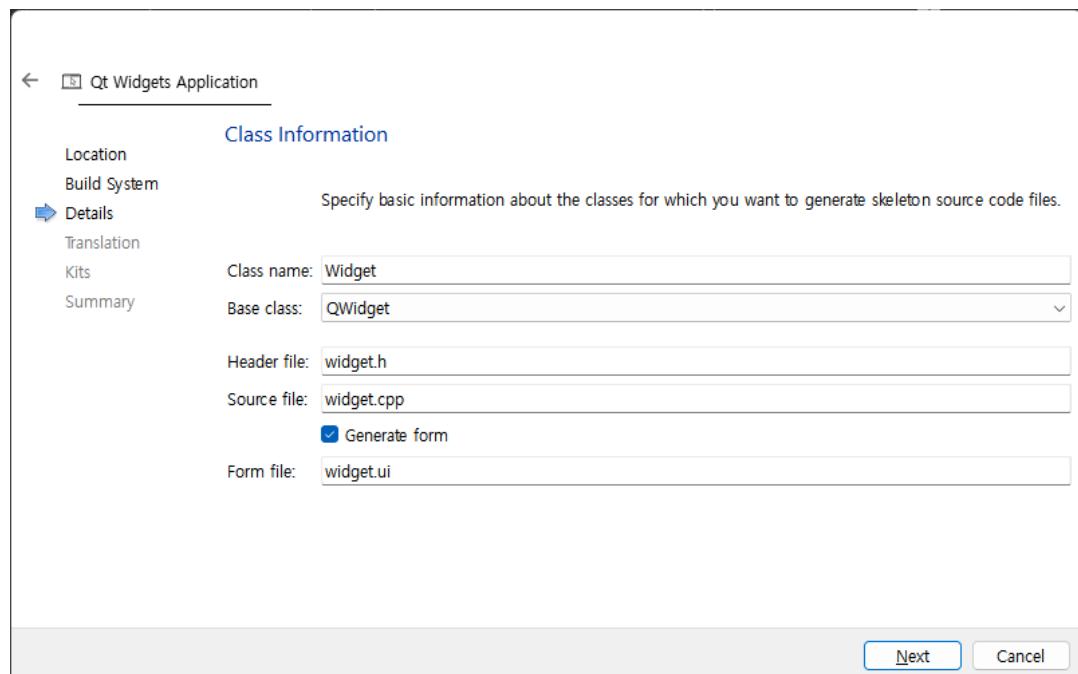
In the second example, we will read data from a shared memory area, convert it to a QPixmap image, and display it on the GUI.

Click the [Writing to the shared memory] button in the first example, as shown in the figure below. This will display the image on the GUI and read the Binary data from the image file and write it to the Shared Memory area.

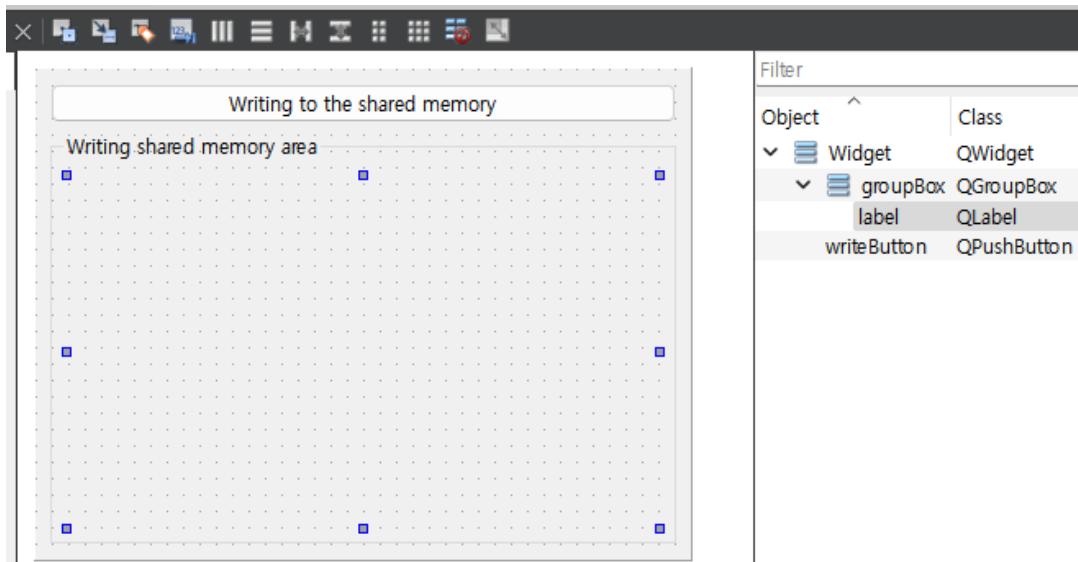
To write the first example, we will create a project based on Qt Widget when creating the project.



And in our first example, we'll be using a UI Form, so we'll check the [Generate form] box.



Once the project is created, open widget.ui and place the widgets as shown below.



Next, open the widget.h header file and write something like this

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QSharedMemory>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
```

```
QSharedMemory *m_sharedMemory;

private slots:
    void writeButton();
};

#endif // WIDGET_H
```

Next, open the widget.cpp source code file and write something like this

```
#include "widget.h"
#include "./ui_widget.h"
#include <QFileDialog>
#include <QBuffer>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
    , ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->writeButton, &QPushButton::pressed,
            this, &Widget::writeButton);

    QString key = QString("qt-dev.com");
    m_sharedMemory = new QSharedMemory(key);
}

void Widget::writeButton()
{
    if (m_sharedMemory->isAttached()) {
        if (!m_sharedMemory->detach())
            ui->label->setText(tr("Shared memory detach failed."));
    }

    QString fileName;
    fileName = QFileDialog::getOpenFileName(
        0, QString(), QString(),
```

```
tr("Images (*.png *.xpm *.jpg)");  
QImage image;  
if (!image.load(fileName)) {  
    ui->label->setText(tr("Could you select a image file?"));  
    return;  
}  
ui->label->setPixmap(QPixmap::fromImage(image));  
  
QBuffer buffer;  
buffer.open(QBuffer::ReadWrite);  
QDataStream out(&buffer);  
out << image;  
int size = buffer.size();  
  
if (!m_sharedMemory->create(size)) {  
    ui->label->setText(tr("Shared memory Segment create failed."));  
    return;  
}  
m_sharedMemory->lock();  
char *to = (char*)m_sharedMemory->data();  
const char *from = buffer.data().data();  
memcpy(to, from, qMin(m_sharedMemory->size(), size));  
m_sharedMemory->unlock();  
}  
  
Widget::~Widget()  
{  
    delete ui;  
}
```

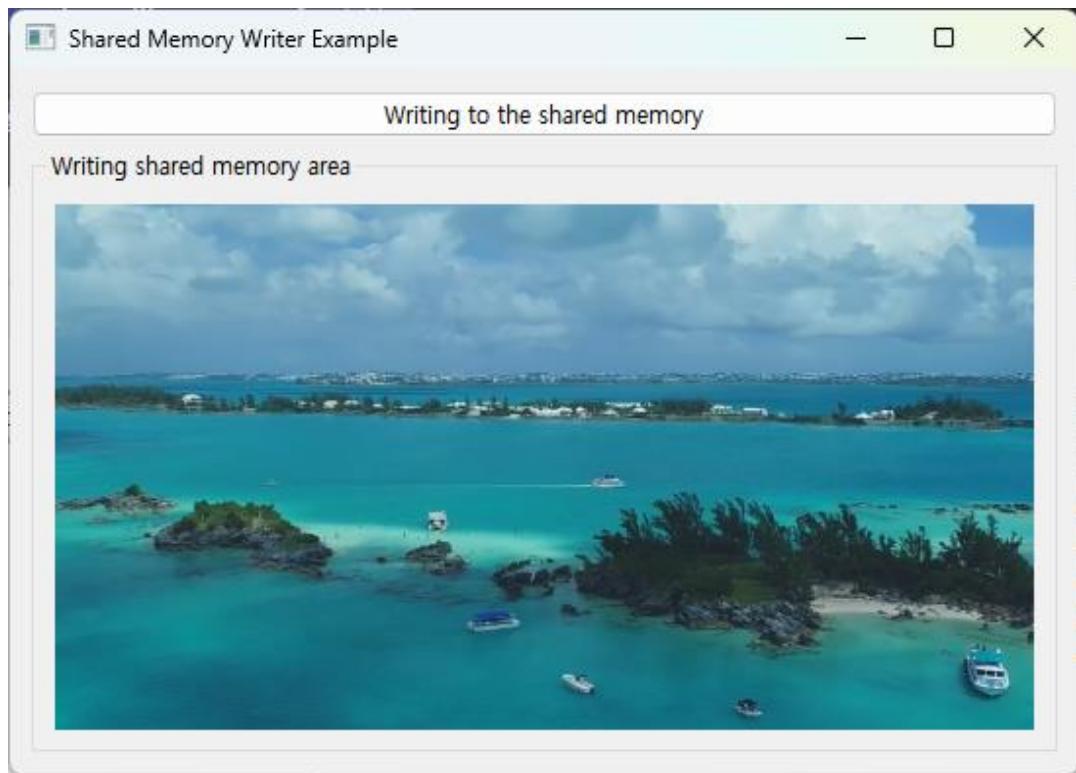
In the constructor function of this class, we declare an object of class QSharedMemory and specify the value of the key to be used in the shared memory. The key uses a unique value and the type is QString.

The writeButton( ) Slot function is called when the [Write Shared Memory] button is clicked. This slot function loads a dialog where you can select a file.

If an image file is selected from the files, the selected image is displayed on the GUI and

Jesus loves you.

the binary data of the image file is read and written to the shared memory area.

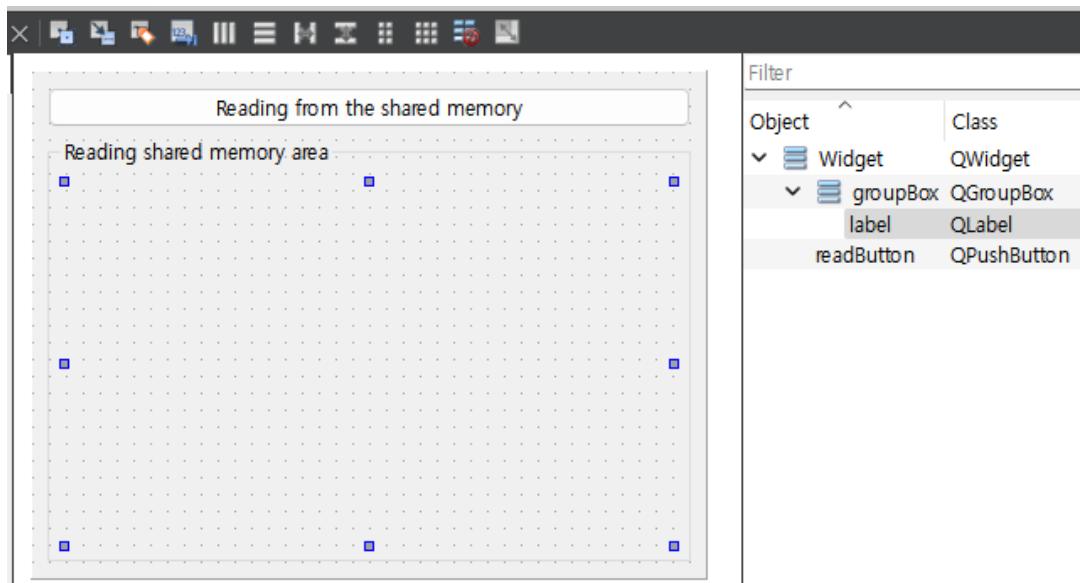


You can find the source code for this example in the `00_SharedMemory_Writer` directory.

- ✓ Example of reading data from shared memory

The second example reads data from the shared memory area written in the first example, converts it to a QPixmap image, and outputs the image on the GUI. When creating a project, we create a project based on Qt Widget.

And in this example, we will use UI Form, so check [Generate form]. When the project is created, open `widget.ui` and place the widgets like below.



Next, open the `widget.h` header file and write something like this

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QSharedMemory>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
    QSharedMemory *m_sharedMemory;

private slots:
    void readButton();
};


```

```
#endif // WIDGET_H
```

Next, open the widget.cpp source code file and write something like this

```
#include "widget.h"
#include "./ui_widget.h"
#include <QBuffer>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
    , ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->readButton, &QPushButton::pressed,
            this, &Widget::readButton);

    QString key = QString("qt-dev.com");
    m_sharedMemory = new QSharedMemory(key);
}

void Widget::readButton()
{
    if (!m_sharedMemory->attach()) {
        ui->label->setText("Reading Failed from shared memory");
        return;
    }

    QBuffer buffer;
    QDataStream in(&buffer);
    QImage image;

    m_sharedMemory->lock();
    buffer.setData((char*)m_sharedMemory->constData(),
                  m_sharedMemory->size());

    buffer.open(QBuffer::ReadOnly);
    in >> image;
```

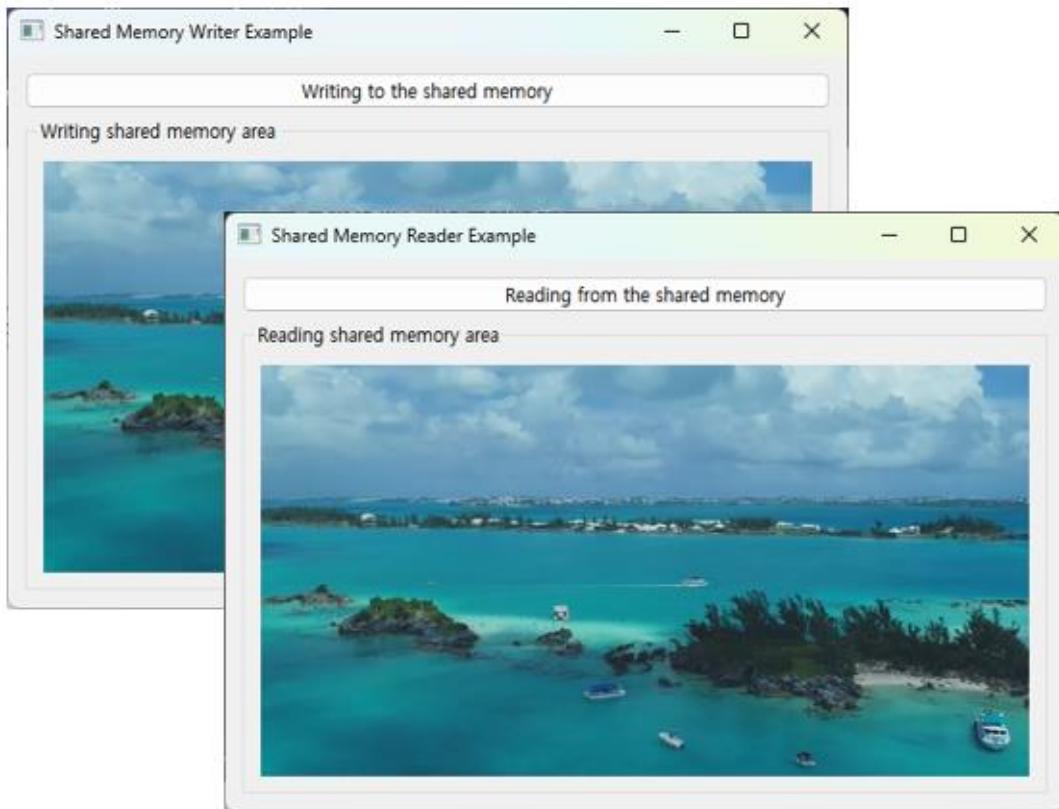
```
m_sharedMemory->unlock();

m_sharedMemory->detach();
ui->label->setPixmap(QPixmap::fromImage(image));
}

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

The `readButton()` Slot function is called when the [Reading from the shared memory] button is clicked on the GUI.

This Slot function reads data from the shared memory area and stores it in a `QImage`. Then, to display the image in the `QLabel` widget on the GUI, the `QImage` is converted back to a `QPixmap` image.



You can find the source code for this example in the `01_SharedMemory_Reader` directory.

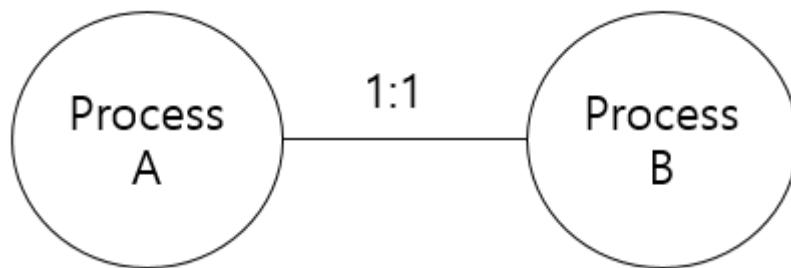
Jesus loves you.

Note that the Writing example is performed first. Then, with the Writing example loaded, you can perform the Reading example.

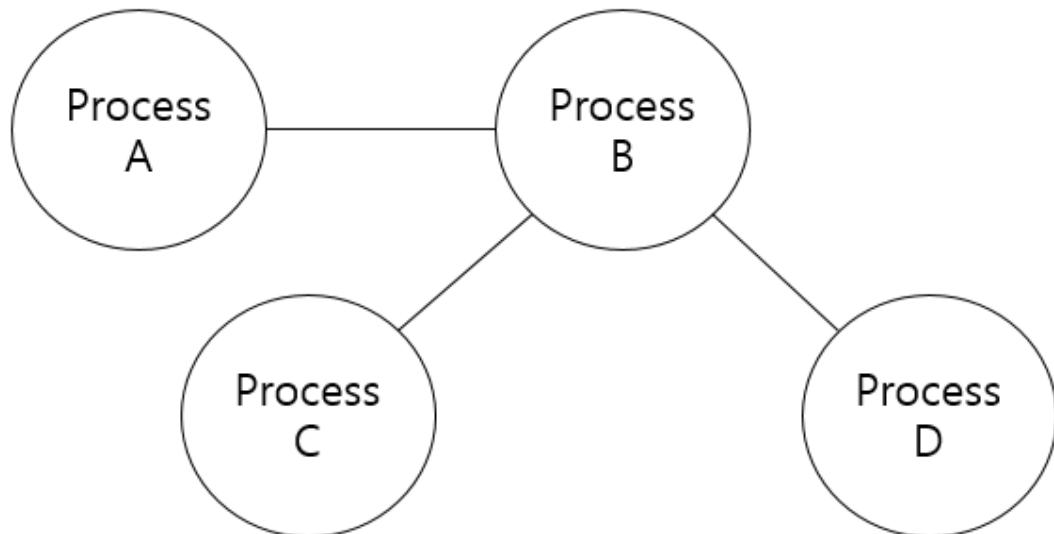
### 37.4. Qt D-Bus

D-Bus is an inter process communication (IPC) and remote procedure call (RPC) mechanism originally developed for Linux to replace the existing IPCs in Linux with a unified protocol called D-Bus.

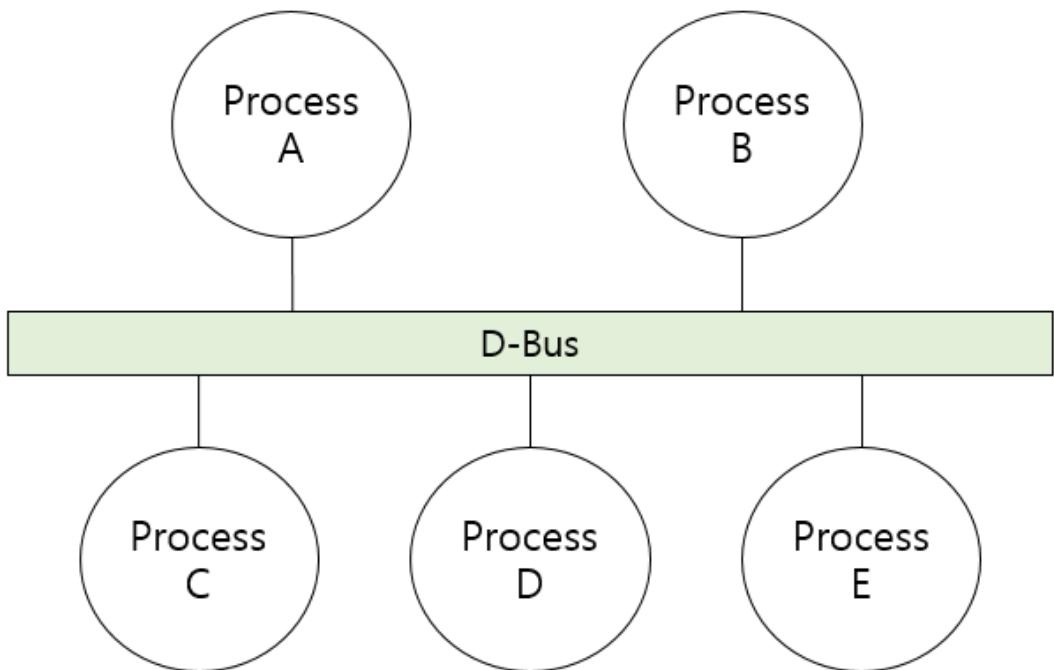
Traditional IPCs communicate in a 1:1 or 1:N fashion.



As shown in the figure above, IPCs are connected in a 1:1 fashion. Inter-process communication using IPC can also be done in a 1:N fashion, as shown in the figure below.



Therefore, in the traditional IPC method, there is always a counterpart to communicate with. However, D-Bus has a different structure than IPC. In D-Bus, communication is not done between processes, but through D-Bus services, as shown in the figure below.



The D-Bus service is used to communicate between processes, and the D-Bus manages the communication. For example, when Process A sends a message to a D-Bus, it can forward the message to all Processes associated with that D-Bus.

In this illustration, there are five Processes, but there may or may not be only one. There can also be six or more Processes.

This means that Process A can send/receive messages from all Processes connected to the D-Bus service because it is not connected to a specific Process to send/receive, but to the D-Bus service.

Note that there is not just one D-Bus service. Each unique name of D-Bus is divided into D-Bus Service Name and Object Path.

The Service Name can be customized, for example, "org.freedesktop.DBus". And the Object Path is separated by "pub/something/".

#### ✓ Types of D-Bus

D-Bus is categorized into System Bus and Session Bus. The System Bus is used to communicate with processes such as the Linux kernel, and the Session Bus is mainly provided to communicate with applications.

✓ Method and Signal

The D-Bus is divided into Method and Signal. Method can be understood as Slot in Qt. And Signal is the same as Signal in Qt.

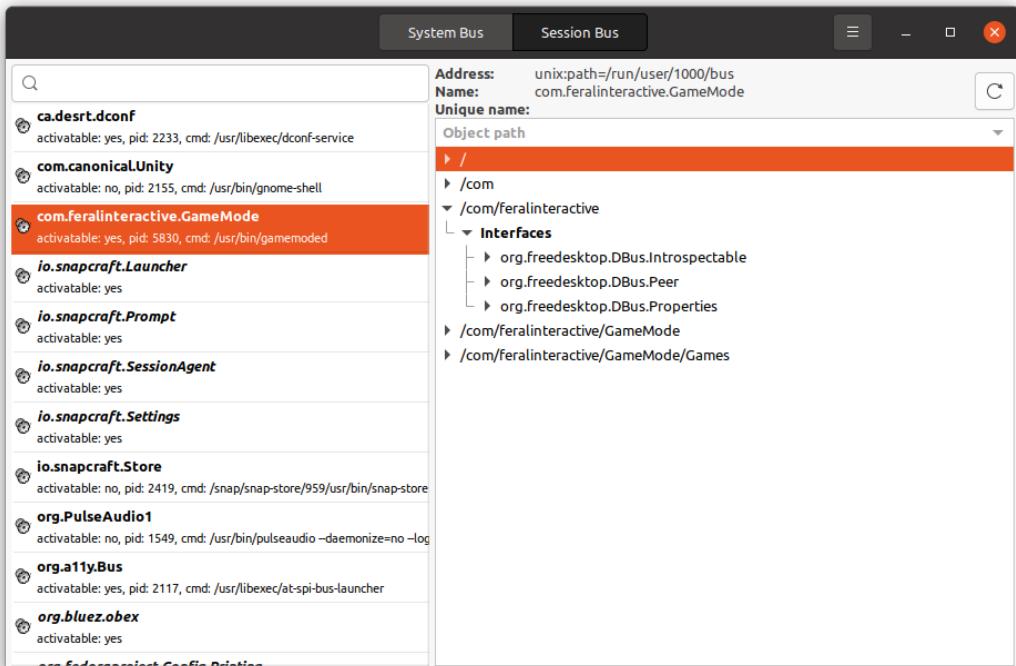
In order to use D-Bus in Qt, we need to add the following to CMake's project file, CMakeList.txt file.

```
find_package(Qt6 REQUIRED COMPONENTS DBus)
target_link_libraries(mytarget PRIVATE Qt6::DBus)
```

✓ Useful D-Bus debugging tools

Linux provides d-feet as a tool for debugging D-Bus. To install d-feet, you can install it using apt like below.

```
$ sudo apt install d-feet
```



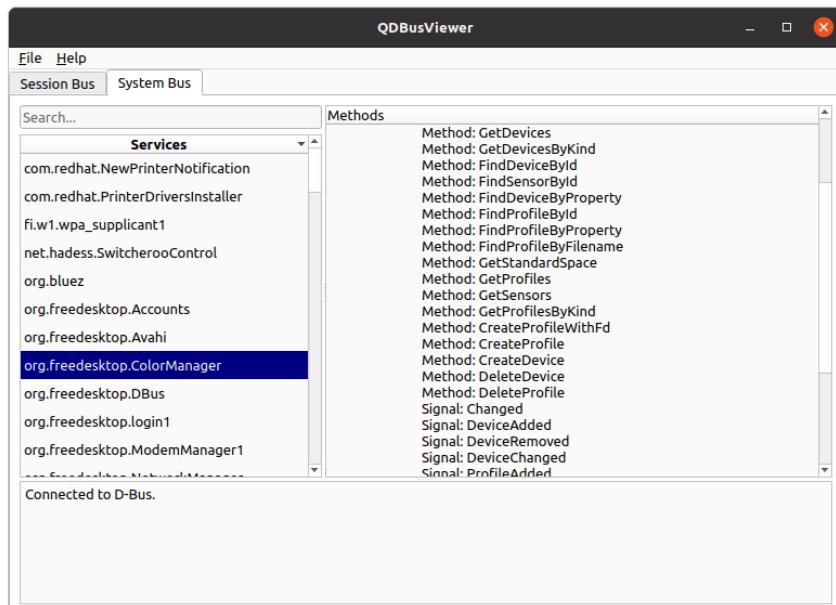
It provides a command tool called "dbus-monitor" as a way to debug messages sent/received on the D-Bus in real time.

```
dev@ubuntu:~/QtProjects/examples$ dbus-monitor
signal time=1699861043.056620 sender=org.freedesktop.DBus -> destination=:1.101 serial=2 path=/org/freedesktop/DBus; interface=org.freedesktop.DBus; member=NameAcquired
    string ":1.101"
signal time=1699861043.056645 sender=org.freedesktop.DBus -> destination=:1.101 serial=4 path=/org/freedesktop/DBus; interface=org.freedesktop.DBus; member=NameLost
    string ":1.101"
```

Qt provides the "qdbusviewer" tool as a tool for debugging D-Bus.

```
dev@ubuntu:~/Qt/6.5.3/gcc_64/bin$ pwd
/home/dev/Qt/6.5.3/gcc_64/bin
dev@ubuntu:~/Qt/6.5.3/gcc_64/bin$ ls
androiddeployqt           lrelease
androiddeployqt.debug      lrelease.debug
androiddeployqt6            lupdate
androidtestrunner          lupdate.debug
androidtestrunner.debug    materialeditor
assistant                  materialeditor.debug
assistant.debug             meshdebug
balsam                     meshdebug.debug
balsam.debug               pixeltool
balsamui                   pixeltool.debug
balsamui.debug             qdbus
canbusutil                 qdbus.debug
canbusutil.debug           qdbuscpp2xml
cooker                     qdbuscpp2xml.debug
cooker.debug               qdbusviewer
designer                   qdbusviewer.debug
```

You can use the QDBusViewer tool to debug the D-Bus.



✓ Interface of D-Bus

To create your own D-Bus Service, you must first create an Interface file in XML format.

```
<node>
  <interface name="local.DBusChat">
    <method name="accelerate"/>
    <signal name="message">
      <arg name="nickname" type="s" direction="out"/>
      <arg name="text" type="s" direction="out"/>
    </signal>
    <signal name="action">
      <arg name="nickname" type="s" direction="out"/>
      <arg name="text" type="s" direction="out"/>
    </signal>
  </interface>
</node>
```

The example above is the Interface of the D-Bus service. Here, method is the same as Slot function in Qt. For example, the method above is the accelerate( ) Slot function in Qt, and when this method is called in D-Bus, the accelerate( ) Slot function is called in Qt.

```
public slots:
  void accelerate();
```

And signal is the same as Signal in Qt. For example, in the XML above, when the message signal is fired, a Signal is fired in Qt. In this case, the arguments of Signal are nickname and text. Therefore, the following signal is called

```
signals:
  void message(const QString &nickname, const QString &text);
```

Qt provides a tool to automatically create XML files. The "dbuscpp2xml" command will automatically create an XML file for you. You can learn more about this in the example.

✓ Interface Classes and Adaptor Classes

In addition to the Interface XML file, Qt requires you to create an Interface class and an

Adaptor class. Qt creates them automatically for the developer's convenience. To create the Interface class and Adaptor class, add the following to the CMakeList.txt file to automatically create the necessary classes.

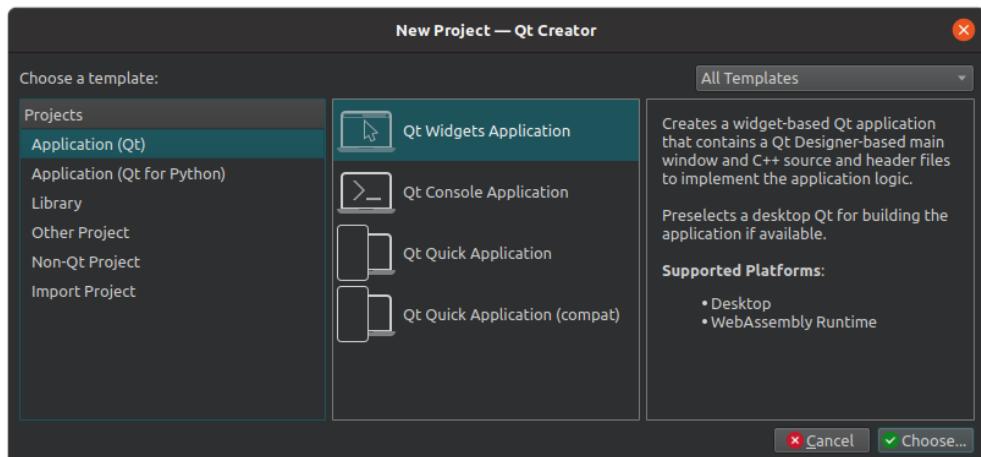
```
set(dbuschat_SRCS)
qt_add_dbus_interface(dbuschat_SRCS
    org.example dbuschat.xml
    dbuschat_interface
)

qt_add_dbus_adaptor(dbuschat_SRCS
    org.example dbuschat.xml
    qobject.h
    QObject
    dbuschat_adaptor
)
```

To learn how to create Interface Classes and Adaptor Classes automatically, we'll cover the method used in the example in more detail.

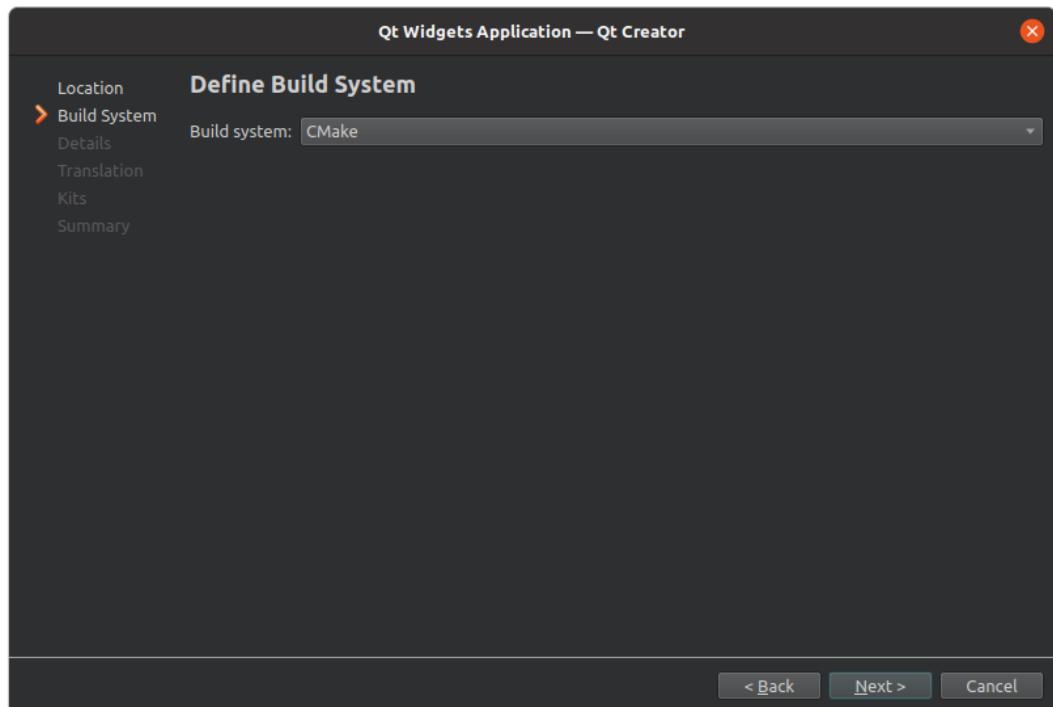
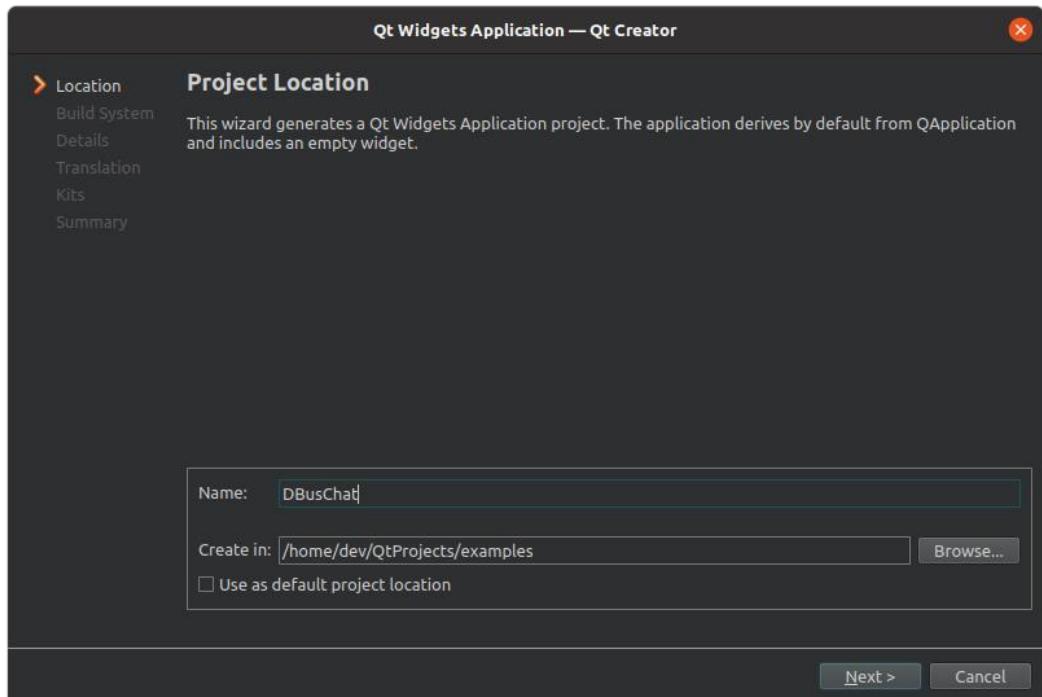
- ✓ Simple Chatting example using D-Bus

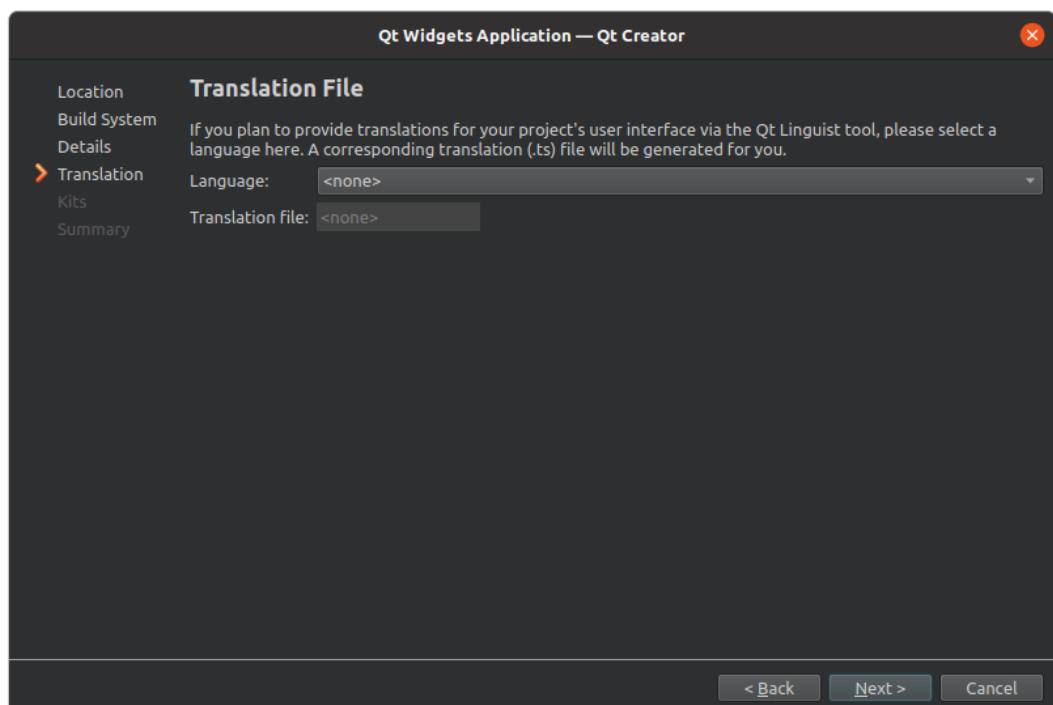
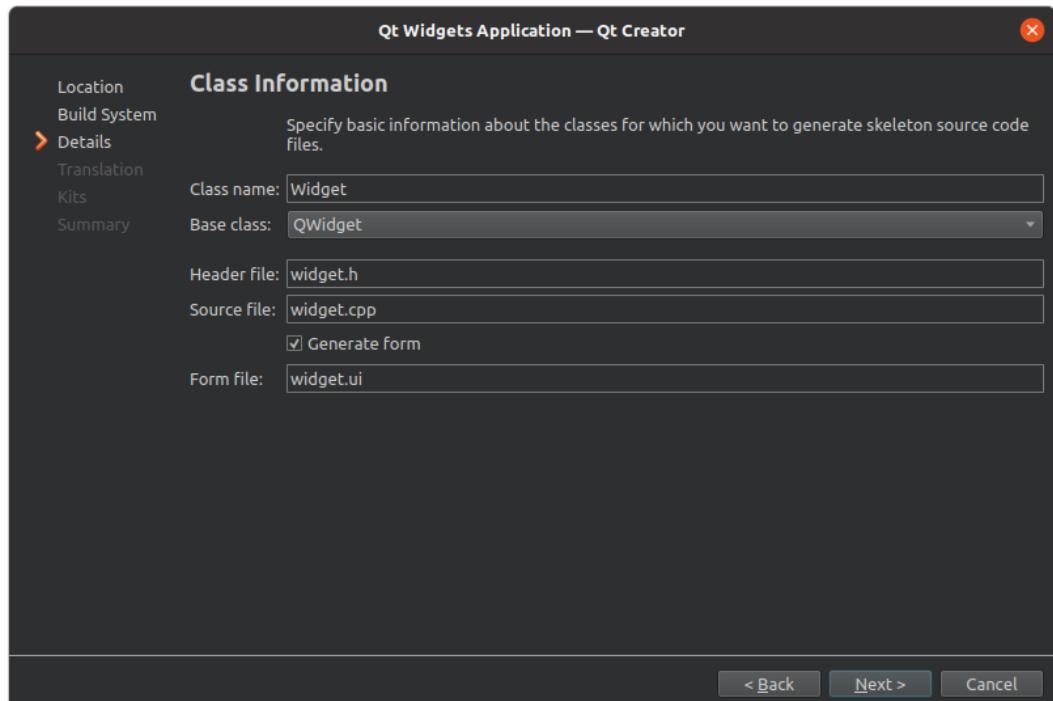
Create a Qt Widget-based project when creating a project.

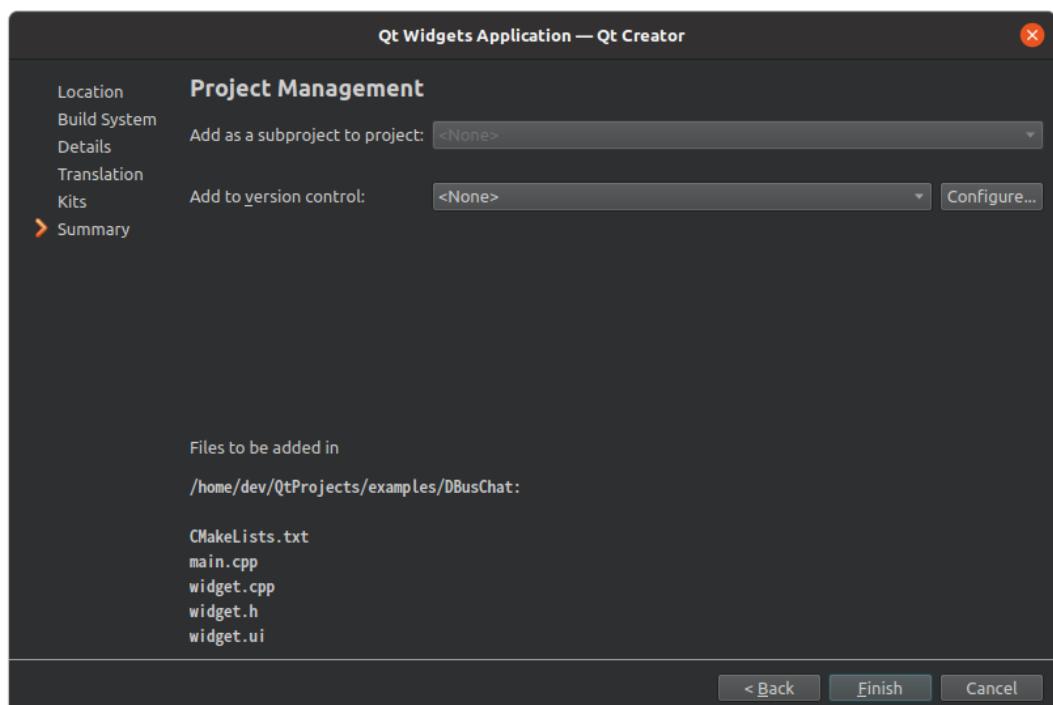
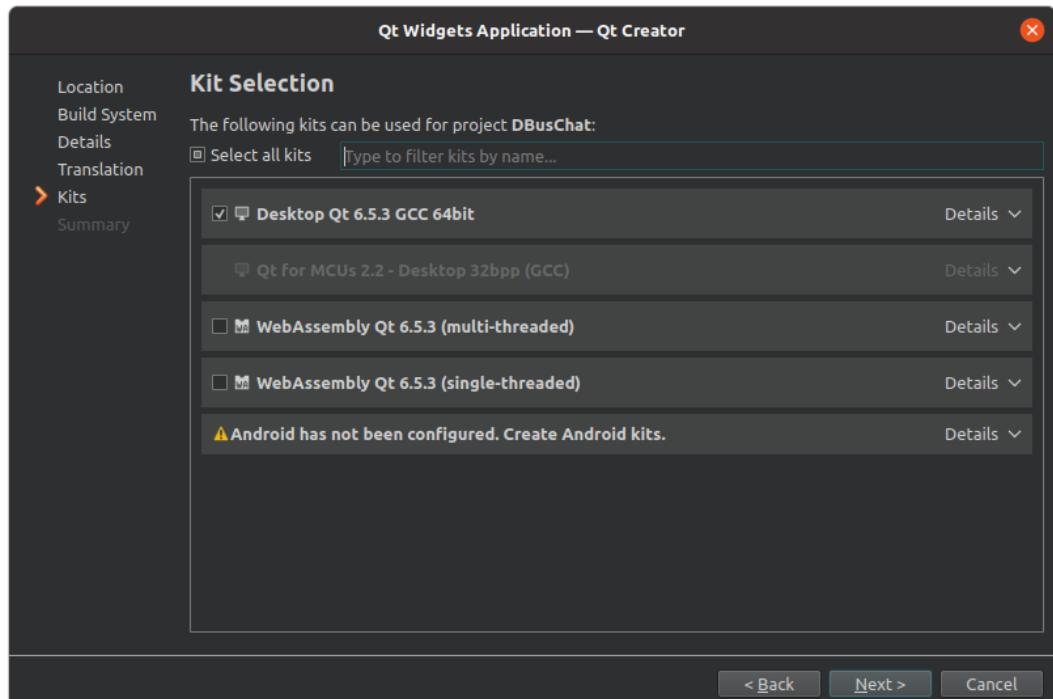


Enter "DBusChat" as the project name.

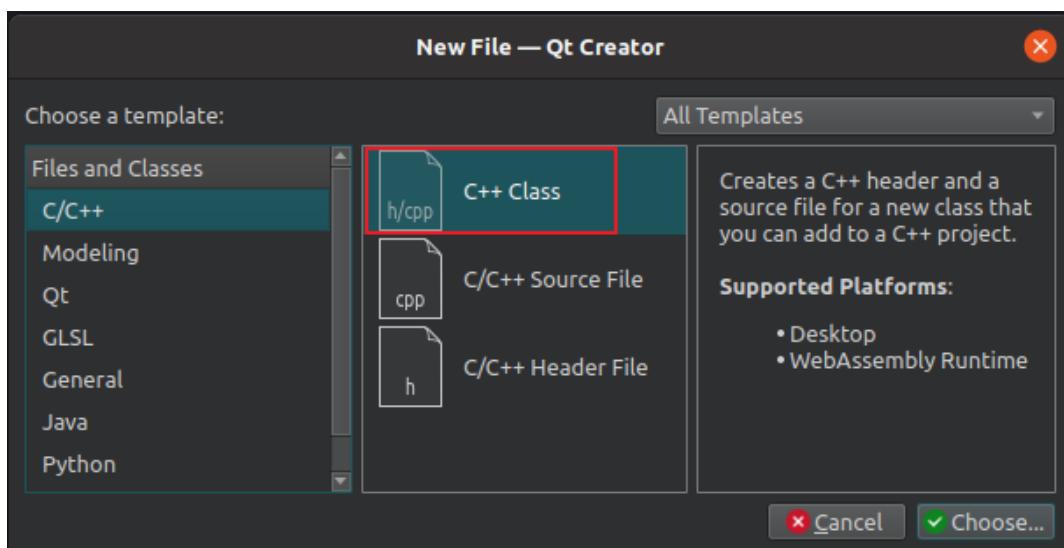
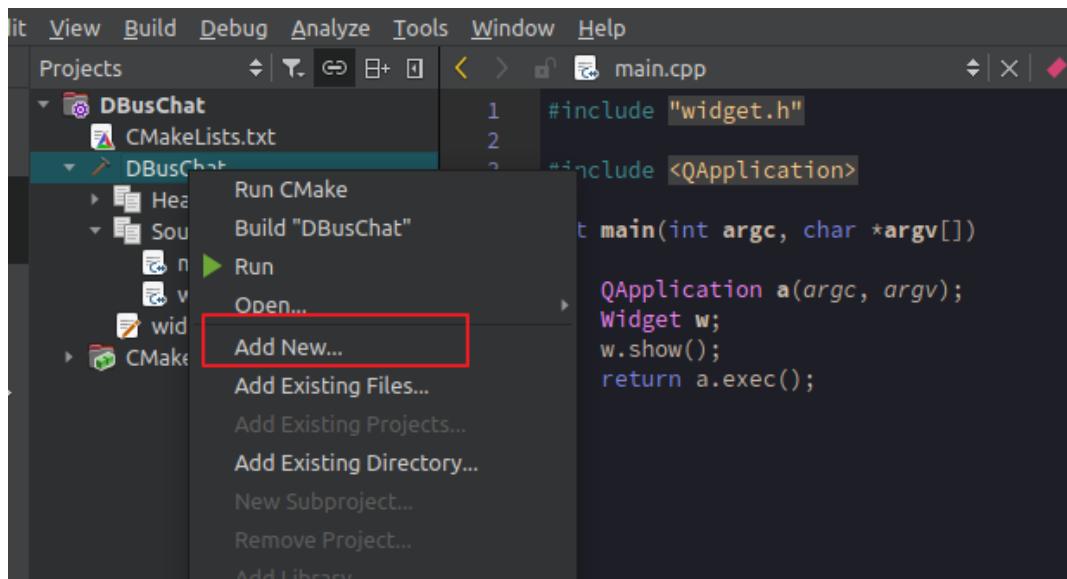
Jesus loves you.

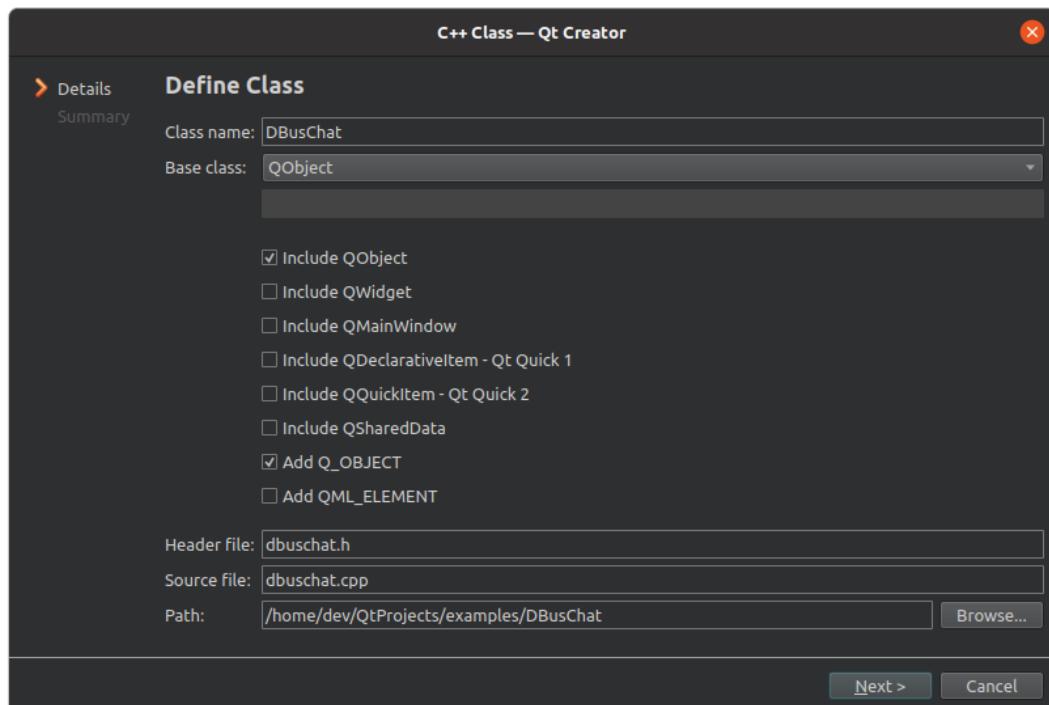






After the project creation is complete, add the DBusChat Class to the project.





Once the project creation is complete, open the CMakeLists.txt file and add the DBus module as shown below.

```
cmake_minimum_required(VERSION 3.5)
...
find_package(Qt6 REQUIRED COMPONENTS Widgets DBus)
...
target_link_libraries(DBusChat PRIVATE
    Qt6::DBus
    Qt6::Widgets
)
...
```

c Next, open the dbuschat.h header file and write the following

```
#ifndef DBUSCHAT_H
#define DBUSCHAT_H

#include <QObject>

class DBusChat : public QObject
```

```
{  
    Q_OBJECT  
public:  
    explicit DBusChat(QObject *parent = nullptr);  
    void newConnection(const QString &nickname, const QString &text);  
    void newMessage(const QString &text);  
  
signals:  
    void message(const QString &nickname, const QString &text);  
    void action(const QString &nickname, const QString &text);  
  
    void uiDisplayMessage(const QString &text);  
  
private:  
    QString m_nickname;  
    void displayMessage(const QString &message);  
};  
  
#endif // DBUSCHAT_H
```

다음으로 dbuschat.cpp 소스코드 파일을 열어서 아래와 같이 작성한다.

```
#include <QApplication>  
#include <QDebug>  
  
#include "dbuschat.h"  
  
DBusChat::DBusChat(QObject *parent)  
    : QObject{parent}  
{  
}  
  
void DBusChat::displayMessage(const QString &message)  
{  
}  
  
void DBusChat::newConnection(const QString &nickname, const QString &text)
```

```
{  
}  
  
void DBusChat::newMessage(const QString &text)  
{  
}
```

Once you've created the above, you need to create an interface file (in XML format) for the DBus service. You can create it manually as described before. However, Qt provides the ability to create it automatically.

You can create the interface file automatically by extracting the Signal and Slot functions registered in the DBusChat Class.

Open a terminal and navigate to the directory where the source code is located, and enter the following command in the terminal window.

```
$ ~/Qt/6.x.x/gcc_64/bin/qdbuscpp2xml ./dbuschat.h  
<!DOCTYPE node PUBLIC "-//freedesktop//DTD D-BUS Object Introspection 1.0//EN"  
"http://www.freedesktop.org/standards/dbus/1.0/introspect.dtd">  
<node>  
  <interface name="local.DBusChat">  
    <signal name="message">  
      <arg name="nickname" type="s" direction="out"/>  
      <arg name="text" type="s" direction="out"/>  
    </signal>  
    <signal name="action">  
      <arg name="nickname" type="s" direction="out"/>  
      <arg name="text" type="s" direction="out"/>  
    </signal>  
    <signal name="uiDisplayMessage">  
      <arg name="text" type="s" direction="out"/>  
    </signal>  
  </interface>  
</node>  
dev@ubuntu:~/QtProjects/examples/DBusChat$
```

If it is displayed like above, copy the contents of `<node>` Tag and save it as "local.DbusChat.xml" file name like below.

And in the above XML, uiDisplayMessage Tag is not used in DBus, but we will use GUI, so delete this tag and save it like below.

```
<node>
  <interface name="local.DBusChat">
    <signal name="message">
      <arg name="nickname" type="s" direction="out"/>
      <arg name="text" type="s" direction="out"/>
    </signal>
    <signal name="action">
      <arg name="nickname" type="s" direction="out"/>
      <arg name="text" type="s" direction="out"/>
    </signal>
  </node>
```

For reference, Qt provides the "qdbusxml2cpp" command. This command also provides a way to extract classes from the interface file.

Coming back to the content, save the above XML-formatted content as a local.DBusChat.xml file. In the previous section, we discussed the Adaptor class and Interface class in Qt.

In this section, we will create the Adaptor class and Interface class automatically. Open the project file and add the following lines.

```
cmake_minimum_required(VERSION 3.5)
project(DBusChat VERSION 0.1 LANGUAGES CXX)
...
find_package(Qt6 REQUIRED COMPONENTS Widgets DBus)

set(dbuschat_SRCS)
qt_add_dbus_interface(dbuschat_SRCS
  local dbuschat.xml
  dbuschat_interface
)
qt_add_dbus_adaptor(dbuschat_SRCS
  local dbuschat.xml
```

```
qobject.h
QObject
dbuschat_adaptor
)

qt_add_executable(DBusChat
    main.cpp
    widget.cpp
    widget.h
    widget.ui
    dbuschat.h dbuschat.cpp
    ${dbuschat_SRCS}
)

target_link_libraries(DBusChat PRIVATE
    Qt6::DBus
    Qt6::Widgets
)

INCLUDE_DIRECTORIES(
    "../build-DBusChat-Desktop_Qt_6_5_3_GCC_64bit-Debug"
)

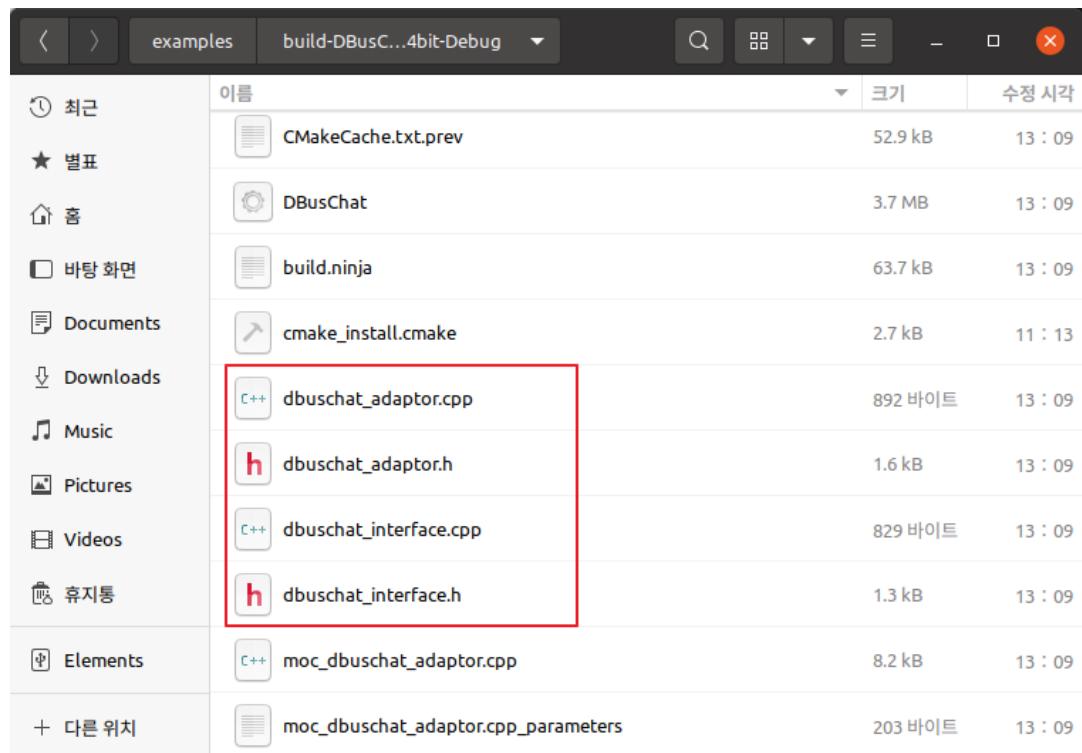
install(TARGETS DBusChat
    BUNDLE DESTINATION .
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

qt\_add\_dbus\_interface( ) automatically creates an Interface class from an XML file. And qt\_add\_dbus\_adaptor( ) automatically creates an Adaptor class from an XML file.

And INCLUDE\_DIRECTORIES( ) at the bottom specifies the build directories for this project. The reason is that Interface class and Adaptor class are created in the built directory, so they can be referenced by DBusChat class.

Write and build as shown above. Once the build is complete, navigate to the built directory and verify that the Interface class and Adapter class are actually created in this

directory.



You should see that the Interface class and Adaptor class have been created, as shown in the image above.

Open the dbuschat.cpp source file again and add the source code as shown below.

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

#include "dbuschat.h"
#include "dbuschat_adaptor.h"
#include "dbuschat_interface.h"

DBusChat::DBusChat(QObject *parent)
    : QObject{parent}
{
    qDebug() << Q_FUNC_INFO;

    new DBusChatAdaptor(this);
}
```

```
auto connection = QDBusConnection::sessionBus();
connection.registerObject("/", this);

using local::DBusChat;

auto *iface = new DBusChat({}, {}, connection, this);

connect(iface, &DBusChat::message, this, [this](const QString &nickname,
const QString &text) {
    displayMessage(tr("<%1> %2").arg(nickname, text));
});

connect(iface, &DBusChat::action, this, [this](const QString &nickname, const
QString &text) {
    displayMessage(tr("* %1 %2").arg(nickname, text));
});
}

void DBusChat::displayMessage(const QString &message)
{
    emit uiDisplayMessage(message);
}

void DBusChat::newConnection(const QString &nickname, const QString &text)
{
    m_nickname = nickname;

    emit action(nickname, text);
}

void DBusChat::newMessage(const QString &text)
{
    emit message(m_nickname, text);
}
```

In this class constructor function, we initialize the D-Bus. Then, when a signal is generated

Jesus loves you.

from the D-Bus service, we execute the `displayMessage( )` function. In the `Connect( )` function, we use a C++ lambda.

When the `newConnection( )` member function is called, it calls the `action( )` Signal to the D-Bus service. Therefore, the Processes connected to the D-Bus service receive the `action( )` Signal.

And when the `newMessage( )` member function is called, the `message( )` Signal of the D-Bus service is called. Therefore, the Processes connected to the D-Bus service receive the `message( )` Signal.

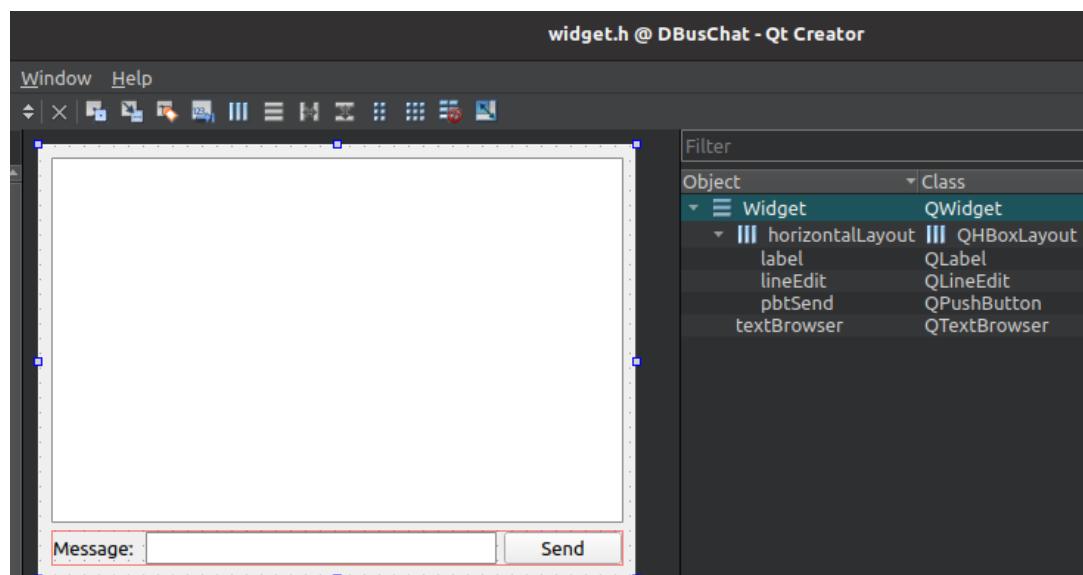
Next, remove the `w.show( )` function from `main.cpp` and write the following code.

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

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

    return a.exec();
}
```

Next, open the `widget.ui` file and place the widgets on the GUI as shown below.



Jesus loves you.

Next, open the widget.h header file and write something like this

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include "dbuschat.h"

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

signals:
    void message(const QString &nickname, const QString &text);
    void action(const QString &nickname, const QString &text);

private:
    Ui::Widget *ui;
    DBusChat *m_dbusChat;

    QStringList m_msgList;
};

#endif // WIDGET_H
```

Next, open the widget.cpp source code file and write something like the following

```
#include "widget.h"
#include "./ui_widget.h"
#include <QInputDialog>
#include <QDebug>
```

```
Widget::Widget(QWidget *parent)
    : QWidget(parent), ui(new Ui::Widget)
{
    ui->setupUi(this);
    auto newNickname = QInputDialog::getText(this,
                                              tr("Set nickname"),
                                              tr("New nickname:"));

    if(newNickname.isEmpty())
    {
        deleteLater();
    }
    else
    {
        this->show();
        m_dbusChat = new DBusChat(this);
        m_dbusChat->newConnection(newNickname, "joins the chat.");

        connect(m_dbusChat, &DBusChat::uiDisplayMessage,
                this, [this](const QString &text) {

            m_msgList.append(text);
            auto history = m_msgList.join(QLatin1String("\n"));
            ui->textBrowser->setPlainText(history);
        });

        connect(ui->pbtSend, &QPushButton::clicked, this, [this]() {
            m_dbusChat->newMessage(ui->lineEdit->text().trimmed());
        });
    }
}

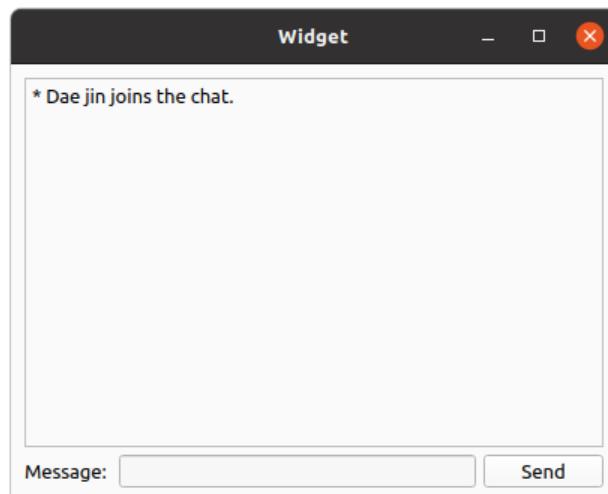
Widget::~Widget()
{
    delete ui;
}
```

Jesus loves you.

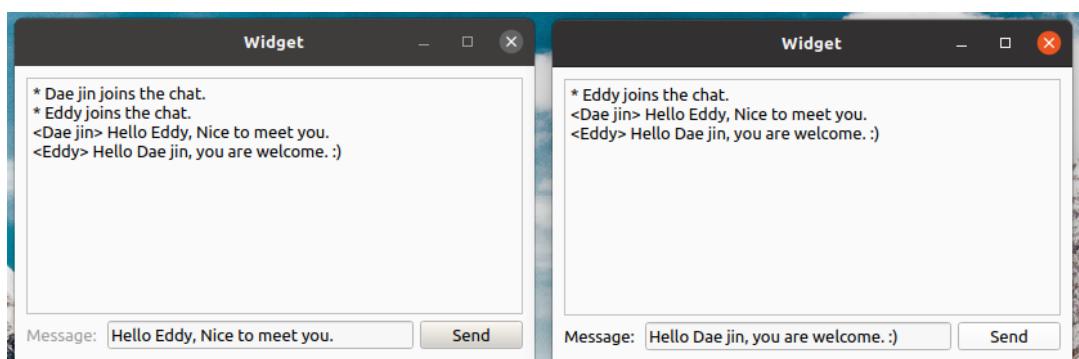
In the Widget class constructor function, a dialog is first loaded to accept input from the user using the QInputDialog class. Here we enter Nick Name.



This will load the Widget below.



If it works like above, let's check if the message is delivered to the other Process via D-Bus normally. Let's run this example further like below and then try to deliver the message.



The complete source code for this example can be found in the 00\_DBusChat directory.

## 38. Multimedia

The Multimedia module provided by Qt allows you to develop applications using audio, video, and camera.

As a cautionary note, there are many changes between Qt 5.x.x and Qt 6.5.x and later versions. Therefore, source code implemented using the Multimedia module provided by Qt 5 is not compatible with Qt 6.

Therefore, it is not possible to use the source code implemented in Qt 5.x.x for Qt 6 and later versions. We will use Qt 6.5.x and later. If you are using Qt 5.x, you will need to reinstall Qt 6.5.x or later.

So let's get back to the point and continue with Qt Multimedia.

To use the Multimedia module in Qt, you need to add the following to your project files

```
find_package(Qt6 REQUIRED COMPONENTS Multimedia)
target_link_libraries(my_project PRIVATE Qt6::Multimedia)
```

In addition, if you need to use multimedia GUI widgets such as QVideoWidget, you will need to add MultimediaWidgets as shown below.

```
find_package(Qt6 REQUIRED COMPONENTS Multimedia MultimediaWidgets)
target_link_libraries(my_project PRIVATE
    Qt6::Multimedia
    Qt6::MultimediaWidgets
)
```

If you're using qmake, you can add the following to your project file

```
QT += multimedia multimediawidgets
```

The Multimedia module events are very easy to implement because they use Signals and Slots.

Of course, it's simple to implement the ability to simply play a music file or a movie file. However, when you need to implement more sophisticated functionality, Signal and Slot make it very easy to do so.

Jesus loves you.

For example, let's take the example of implementing an internet radio station. Suppose you need to implement an application that sends music input from a microphone across a network of computers every 100 milliseconds, and plays it back on the receiving computer.

This is difficult to accomplish with an API that simply plays a music file. Therefore, the Multimedia module provided by Qt has the advantage of making it easier to implement simple to more complex multimedia applications.

In addition, the Multimedia module provided by Qt is platform-independent. For example, an application implemented using the Qt Multimedia module on the MS Windows platform can be used equally well on Linux or MacOS, as well as on mobile platforms.

In this chapter, we will introduce the Multimedia modules provided by Qt in three chapters.

### ① Audio

We'll look at how to manipulate sound sources at a low level, starting with the ability to simply play music files.

### ② Video

Let's take a look at how to play a video file.

### ③ Camera

Let's take a look at how to scan the system for available camera devices and output the video from the camera to the GUI.

## 38.1. Audio

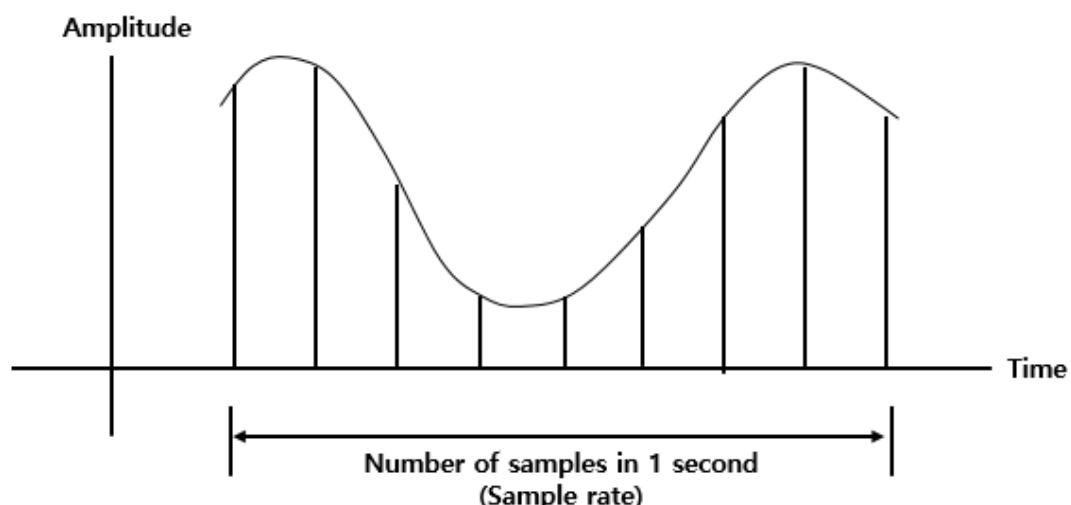
Before we get into how to implement applications using the Qt Multimedia module, let's take a look at some essential Audio basics. If you don't need a basic understanding of Audio, you can skip the Audio basics section.

✓ **Audio basics**

When a person speaks, sound waves are transmitted to another person's ear in the form of an analog waveform, but in order for a computer to play the sound source on a digital device, the analog waveform must be converted to binary. In order to control the sound source converted to binary, you must first understand the concepts of sample rate and bit rate.

✓ **Sample rate**

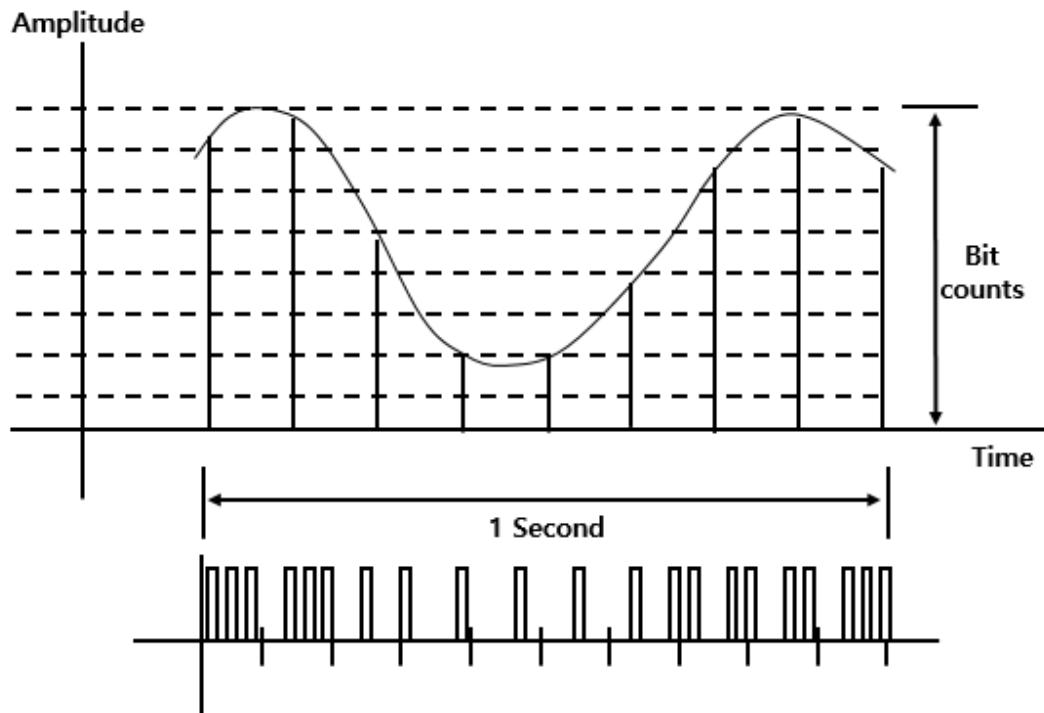
Sample rate is a specific value extracted from a continuous waveform, such as an analog signal, by converting it to binary digital. In other words, sample rate is the number of samples extracted per second, or the frequency of samples. In audio terms, it is expressed in units of 52.3 KHz (52,300 Hz) or 22.4 KHz (22,400 Hz). This means the number of samples repeated per second.



For example, 52.1 KHz means that 52,100 samples are taken in one second. You can see that the higher the sample count, the more detailed it becomes. The quality we use for CD quality is 44.1 KHz.

- ✓ Bit rate

The number of bits used in one second, in bps. When we talk about the quality of a sound source, we say bps, such as 96 Kbps, 128 Kbps, 192 Kbps, etc.



As you can see in the figure above, the analog to digital conversion signal is represented as 1 and 0, and only two states can be displayed. And digitizing analog into 1 and 0 is called PCM (Pulse Code Modulation).

Once we have the sample rate and bit rate, we can decode an MP3 file and calculate the number of bytes in the entire sound source.

For example, if the sample.mp3 file has a playback time of 299 seconds (4 minutes 59 seconds), a bit rate of 56 Kbps, and a sample rate of 22,050 Hz, the number of bytes in the sample.mp3 file is calculated as "playback time \* bit rate / 8".

And you can get the PCM as follows. The PCM file size can be calculated as "PCM data

throughput per second \* playback time / 8".

### Example calculation using duration, sample rate and bit rate

#### About playback

Duration: 299.277 seconds (4 minutes, 59 seconds)  
Bitrate: 56 Kbps (CBR)  
Channels: 2  
Bit: 16-bit  
Sample Rate: 22050 Hz

#### Sample.mp3 – Calculate file size for MP3 format

File size = Duration \* Bit rate / 8  
 $299.277 \times 56,000(\text{Hz}) / 8 = 2,094,939 \text{ Bytes}$

#### Sample.pcm – Calculate file size for PCM format

PCM File size = PCM data throughput per second \* Duration / 8  
 $705,600 \times 299.277 / 8 = 26,394,624 \text{ Bytes}$

PCM data throughput per second = Sample rate \* Channel \* Bit  
 $22,050(\text{Hz}) \times 2 \times 16 \text{ (bit)} = 705,600 \text{ Bit (88,200 Kbytes)}$

In audio, a channel is referred to as mono if it sends sound in one direction, and stereo if it sends sound in two channels, or two directions.

For example, it refers to how many directions the sound is divided into during the recording process. In the MP3 file calculation, Channel refers to the channel just mentioned. And Bit refers to the number of bits per sample. The higher the number of bits, the smaller the analog waveform can be broken down.

So far, we've covered the basic concepts we need before implementing audio.

Next, let's see how to use the Qt Multimedia module to play a simple MP3 file.

To simply play MP3s, we can use the QMediaPlayer class to implement easy playback functionality.

```
m_player = new QMediaPlayer();
m_audioOutput = new QAudioOutput;
m_player->setAudioOutput(m_audioOutput);
...
float volValue = ui->volSlider->value() / 100.0;
m_audioOutput->setVolume(volValue);
```

The QMediaPlayer class can be thought of as a container. In addition to audio, QMediaPlayer class provides container functions for video and camera.

To play a music file, you need to connect an object of QAudioOutput class to QMediaPlayer using the setAudioOutput( ) member function.

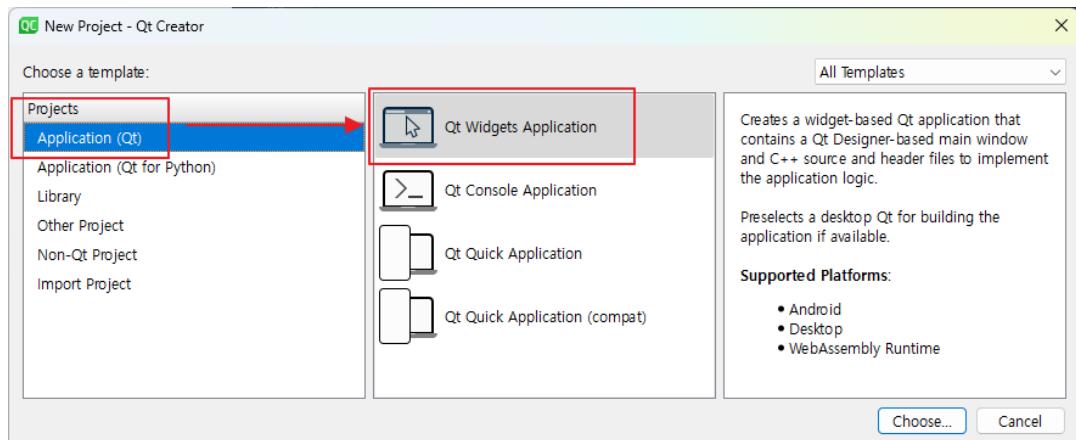
Next, you can use the setSource( ) member function of the QMediaPlayer class to specify the music file. Then, to play the music file, you can use the play( ) member function.

```
m_player->setSource(QUrl::fromLocalFile(m_fName));
ui->sliderPosition->setEnabled(true);
m_player->play();
```

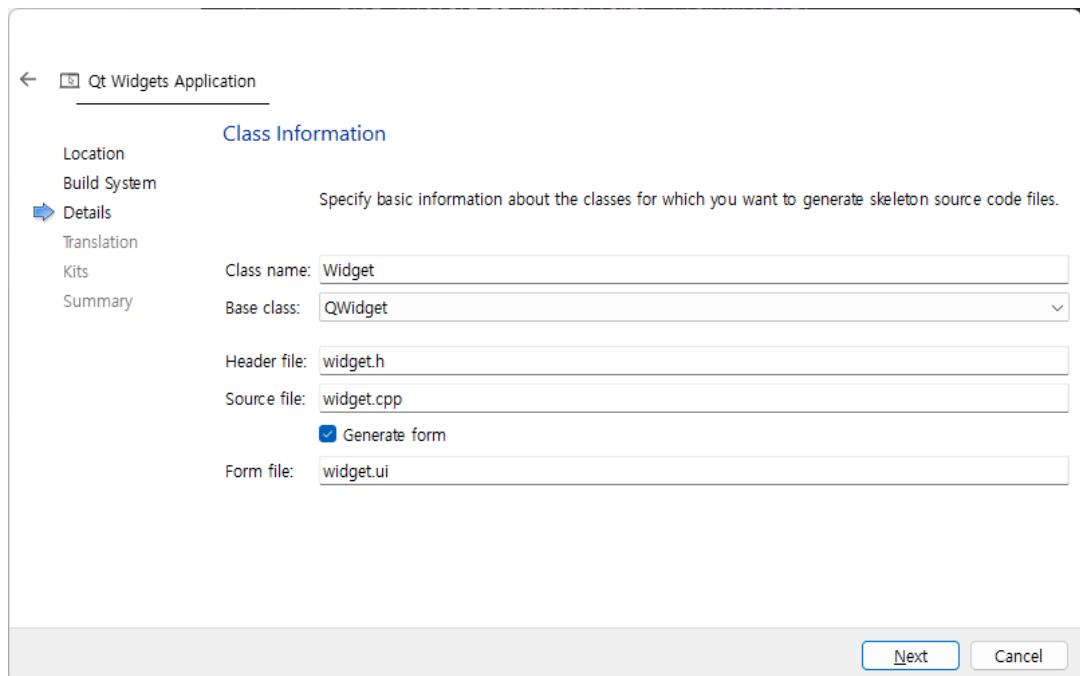
We've briefly covered how to play music files. Next, let's implement a simple music file playback example with a real-world example.

- ✓ Simple Audio playback example

When creating a project, select an Application based on Qt Widgets.



Since we will use UI Form, check [Generation form] as shown below.



Once the project is created, open the CMakeLists.txt file and add the Multimedia module as shown below.

```
cmake_minimum_required(VERSION 3.5)

project(00_SimpleAudioPlayer VERSION 0.1 LANGUAGES CXX)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)

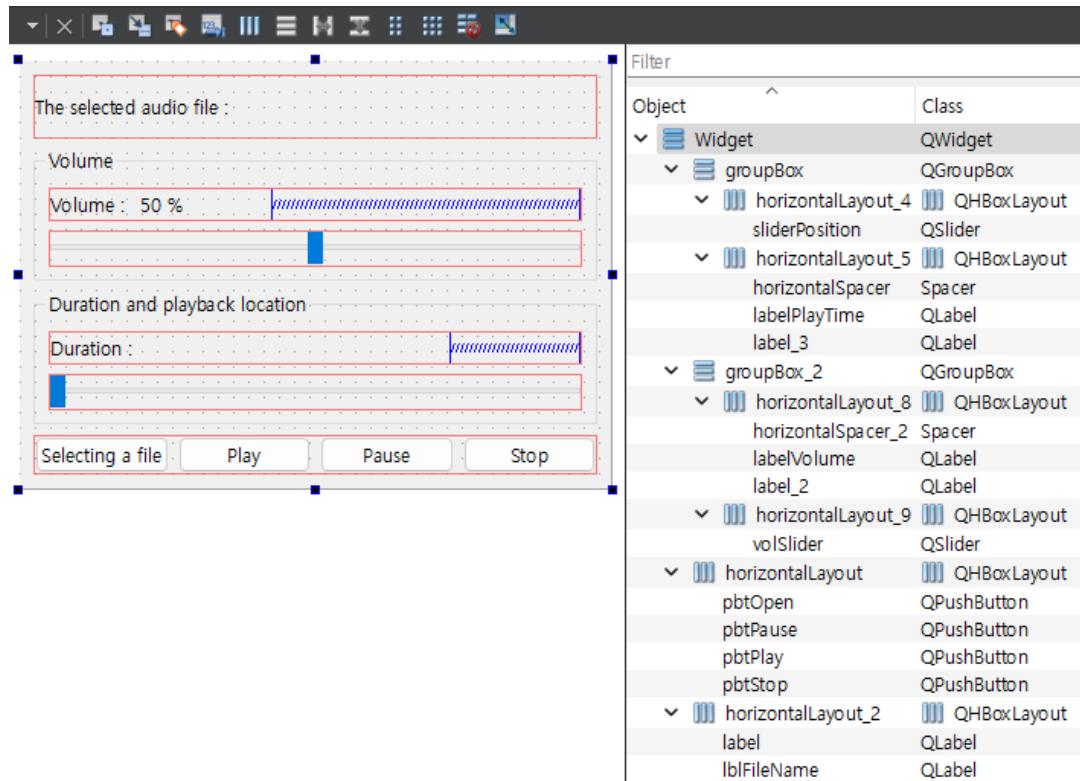
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Qt6 REQUIRED COMPONENTS Widgets Multimedia)

set(PROJECT_SOURCES
    main.cpp
    widget.cpp
    widget.h
    widget.ui)
```

```
)  
  
qt_add_executable(00_SimpleAudioPlayer  
    ${PROJECT_SOURCES}  
)  
  
target_link_libraries(00_SimpleAudioPlayer PRIVATE  
    Qt6::Widgets  
    Qt6::Multimedia  
)  
  
install(TARGETS 00_SimpleAudioPlayer  
    BUNDLE DESTINATION .  
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}  
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}  
)
```

Next, open the widget.ui file and place the GUI widgets as shown below.



Next, open the widget.h header file and write something like this

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QMediaPlayer>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;

    QString      fName;
    QMediaPlayer *m_player;
    QAudioOutput *m_audioOutput;
    qint64        m_duration;

private slots:
    void onOpenBtn();
    void onPlayBtn();
    void onPauseBtn();
    void onStopBtn();

    void sliderValueChange(int val);
    void durationChanged(qint64 duration);
    void positionChanged(qint64 progress);
```

```
void seek(int seconds);
};

#endif // WIDGET_H
```

Next, create the widget.cpp source code file as shown below.

```
#include "widget.h"
#include "./ui_widget.h"
#include <QFileDialog>
#include <QTime>
#include <QAudioOutput>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
    , ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->volSlider, SIGNAL(valueChanged(int)),
            this, SLOT(sliderValueChange(int)));

    ui->sliderPosition->setEnabled(false);

    connect(ui->pbtOpen, SIGNAL(clicked()), this, SLOT(onOpenBtn()));
    connect(ui->pbtPlay, SIGNAL(clicked()), this, SLOT(onPlayBtn()));
    connect(ui->pbtPause, SIGNAL(clicked()), this, SLOT(onPauseBtn()));
    connect(ui->pbtStop, SIGNAL(clicked()), this, SLOT(onStopBtn()));

    m_player = new QMediaPlayer();
    m_audioOutput = new QAudioOutput;
    m_player->setAudioOutput(m_audioOutput);

    float volValue = ui->volSlider->value() / 100.0;
    m_audioOutput->setVolume(volValue);

    connect(m_player, &QMediaPlayer::durationChanged,
            this,     &Widget::durationChanged);
```

```
connect(m_player, &QMediaPlayer::positionChanged,
        this,      &Widget::positionChanged);

connect(ui->sliderPosition, &QSlider::sliderMoved,
        this,          &Widget::seek);

}

void Widget::sliderValueChange(int val)
{
    ui->labelVolume->setText(QString("%1 %").arg(val));
    float volValue = val / 100.0;
    m_audioOutput->setVolume(volValue);
}

void Widget::onOpenBtn()
{
    m_fName = QFileDialog::getOpenFileName(this,
                                           tr("Open File"),
                                           QDir::homePath(),
                                           tr("MP3 files (*.mp3);"));

    if(!m_fName.isNull())
        ui->lblFileName->setText(m_fName);
}

void Widget::onPlayBtn()
{
    if(!m_fName.isNull())
    {
        m_player->setSource(QUrl::fromLocalFile(m_fName));
        ui->sliderPosition->setEnabled(true);
        m_player->play();
    }
}
```

```
void Widget::onPauseBtn()
{
    int state = m_player->playbackState();
    if(state == QMediaPlayer::PausedState)
    {
        m_player->play();
    }
    else if(state == QMediaPlayer::PlayingState)
    {
        m_player->pause();
    }
}

void Widget::onStopBtn()
{
    int state = m_player->playbackState();

    if(state == QMediaPlayer::PlayingState)
    {
        m_player->stop();
    }
}

void Widget::durationChanged(qint64 duration)
{
    m_duration = duration / 1000;
    ui->sliderPosition->setMaximum(m_duration);
}

void Widget::positionChanged(qint64 progress)
{
    if (!ui->sliderPosition->isSliderDown())
        ui->sliderPosition->setValue(progress / 1000);

    qint64 currentInfo = progress / 1000;
```

```
QString playTimeStr;
if (currentInfo || m_duration) {
    QTime currentTime((currentInfo / 3600) % 60,
                      (currentInfo / 60) % 60,
                      currentInfo % 60,
                      (currentInfo * 1000) % 1000);

    QTime totalTime((m_duration / 3600) % 60,
                    (m_duration / 60) % 60,
                    m_duration % 60,
                    (m_duration * 1000) % 1000);

    QString format = "mm:ss";
    if (m_duration > 3600)
        format = "hh:mm:ss";
    playTimeStr = currentTime.toString(format) + " / "
                  + totalTime.toString(format);
}

ui->labelPlayTime->setText(playTimeStr);
}

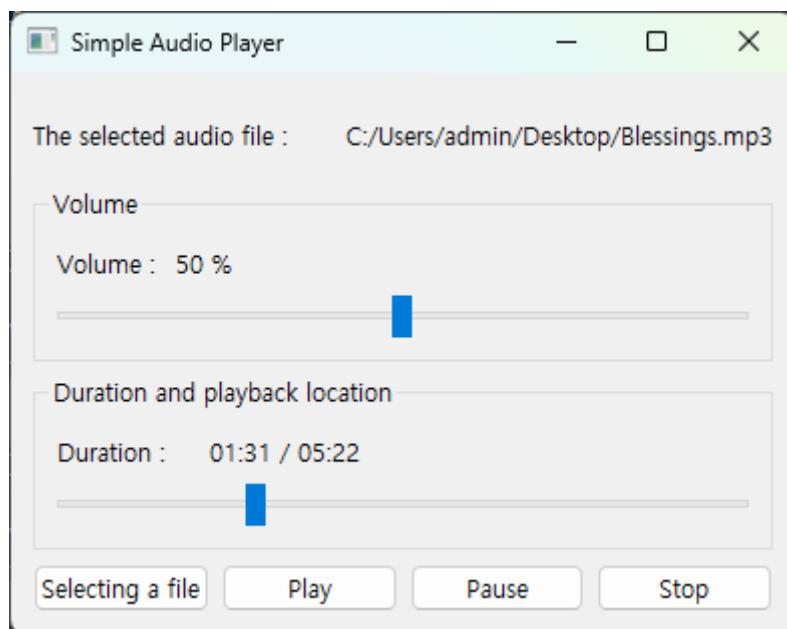
void Widget::seek(int seconds)
{
    m_player->setPosition(seconds * 1000);
}

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

The constructor function connects the Signal and Slot of each button on the GUI and creates a QMediaPlayer class and a QAudioOutput class object. We then pass the QAudioOutput class object as an argument to the setAudioOutput( ) member function of the QMediaPlayer class.

Jesus loves you.

The durationChanged( ) signal is emitted when the playback time changes in the QMediaPlayer class, so we associate this signal with the durationChanged( ) Slot function. The positionChanged signal in the QMediaPlayer class is used to notify you when the position of the playback has changed. This Signal is associated with the positionChanged( ) Slot function. As a note, when adjusting the Volume, the minimum value is between 0.0 and 1.0. If you have finished writing the source code up to this point, let's build and run it.



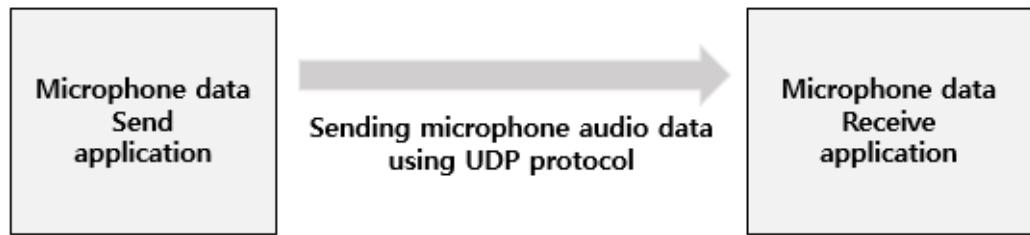
The complete source code for this example can be found in the 00\_SimpleAudioPlayer directory.

- ✓ Example of sending a microphone input signal to the network (UDP)

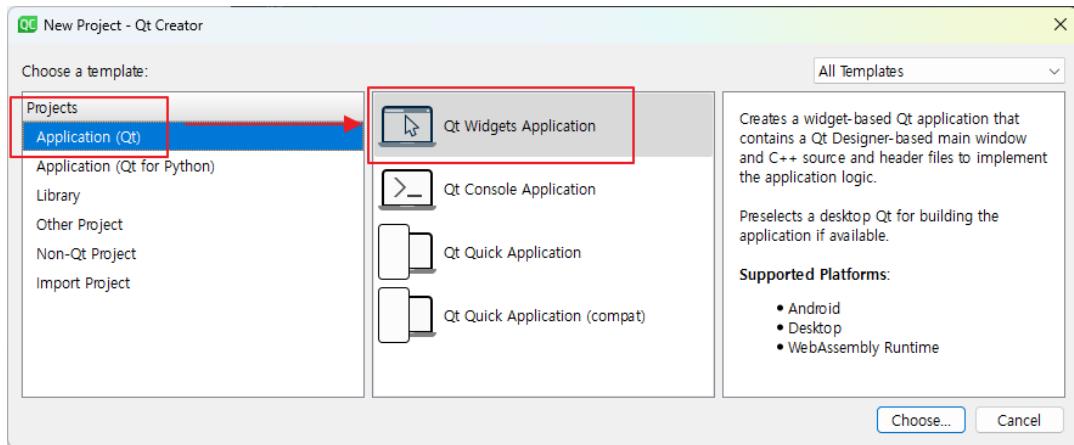
This time we will implement two examples. The first is an example that extracts Audio data from a microphone and sends it to the network.

The second example is to output the data received over the network to the speaker.

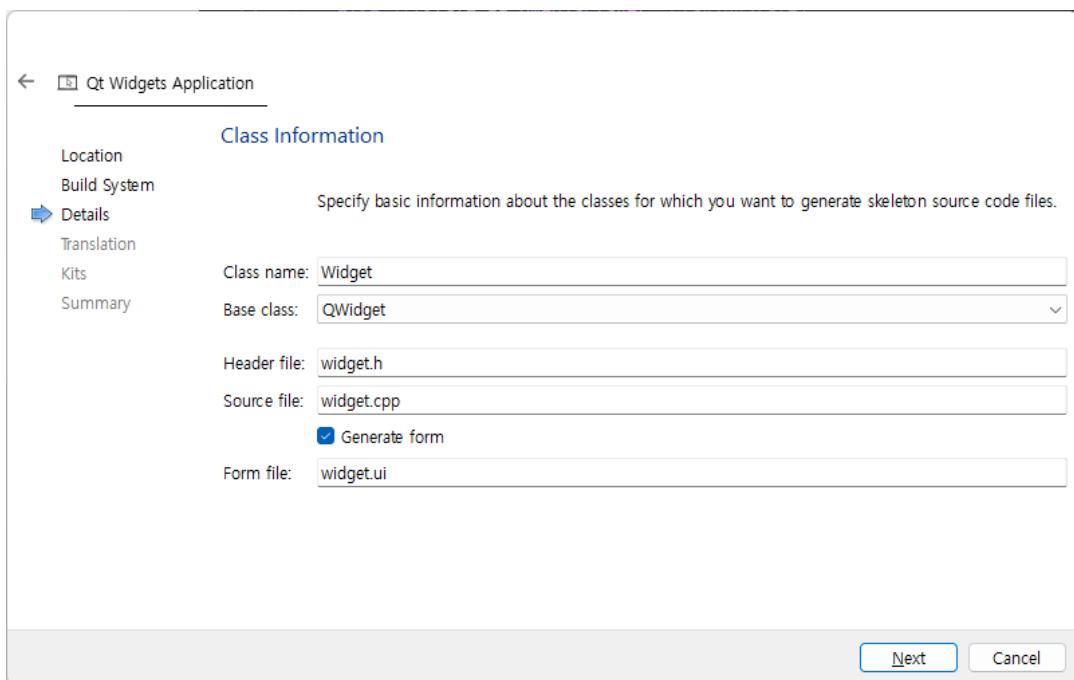
Jesus loves you.



Let's implement our first application first. When creating a project, create an Application based on the Qt Widget.



In this project, we will use UI Form, so check [Generation form] as shown below.



Once the project is created, add the Multimedia module and Network module to the

CMakeLists.txt file as shown below.

```
cmake_minimum_required(VERSION 3.5)

project(01_AudioSender VERSION 0.1 LANGUAGES CXX)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Qt6 REQUIRED COMPONENTS
    Widgets
    Multimedia
    Network
)

set(PROPERTY_SOURCES
    main.cpp
    widget.cpp
    widget.h
    widget.ui
)

qt_add_executable(01_AudioSender
    ${PROPERTY_SOURCES}
)

target_link_libraries(01_AudioSender PRIVATE
    Qt6::Widgets
    Qt6::Multimedia
    Qt6::Network
)

install(TARGETS 01_AudioSender
```

```
BUNDLE DESTINATION .
LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

Next, create the widget.h header file as shown below.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QtMultimedia>
#include <QUdpSocket>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;

    QMediaDevices      *m_devices = nullptr;
    QList<QAudioDevice> devList;
    QAudioFormat        mFormat;
    QAudioDevice        mDeviceInfo;

    QAudioInput         *mAudioInput = nullptr;
    Q AudioSource       *m_audioSource = nullptr;
```

```
QIODevice           *mInput;
QByteArray         mBuffer;
int                mInputVolume;

QUdpSocket         mUdpSocket;
void audioInitialize();

private slots:
    void volSliderChanged(int val);
    void sendStartBtn();
    void sendStopBtn();

    void stateChanged(QAudio::State state);
    void readMore();
};

#endif // WIDGET_H
```

Next, open the widget.cpp source code file and write something like the following

```
#include "widget.h"
#include "./ui_widget.h"

Widget::Widget(QWidget *parent)
    : QWidget(parent)
    , ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->volSlider,      &QSlider::valueChanged,
            this,                  &Widget::volSliderChanged);
    connect(ui->pbtSendStart,   &QPushButton::clicked,
            this,                  &Widget::sendStartBtn);
    connect(ui->pbtSendStop,    &QPushButton::clicked,
            this,                  &Widget::sendStopBtn);

    ui->volSlider->setEnabled(false);
    ui->pbtSendStop->setEnabled(false);
```

```
    audioInitialize();
}

void Widget::audioInitialize()
{
    m_devices = new QMediaDevices(this);

    devList = m_devices->audioInputs();
    for(int i = 0; i < devList.size(); ++i)
        ui->comboDevList->addItem(devList.at(i).description());

    mDevInfo = devList.at(ui->comboDevList->currentIndex());

    mFormat.setSampleRate(8000);
    mFormat.setChannelCount(1);

    mFormat.setSampleFormat(QAudioFormat::Int16);

    mAudioInput = new QAudioInput(mDevInfo, this);
    mInputVolume = ui->volSlider->value();
    ui->volLabel->setText(QString("%1 %").arg(mInputVolume));

    mBuffer = QByteArray(14096, 0);
}

void Widget::volSliderChanged(int val)
{
    mInputVolume = val;
    ui->volLabel->setText(QString("%1 %").arg(mInputVolume));
    mAudioInput->setVolume(qreal(mInputVolume) / 100);
}

void Widget::sendStartBtn()
{
    mDevInfo = devList.at(ui->comboDevList->currentIndex());
```

```
mAudioInput = new QAudioInput(mDeviceInfo, this);
mAudioInput->setVolume(qreal(mInputVolume) / 100);
m_audioSource = new QAudioSource(mDeviceInfo, mFormat);
m_audioSource->setBufferSize(200);

connect(m_audioSource, SIGNAL(stateChanged(QAudio::State)),
        this,           SLOT(stateChanged(QAudio::State)));

mInput = m_audioSource->start();
connect(mInput, SIGNAL(readyRead()), this, SLOT(readMore()));
}

void Widget::sendStopBtn()
{
    if(mInput != nullptr) {
        disconnect(mInput, nullptr, this, nullptr);
        mInput = nullptr;
    }

    m_audioSource->stop();
}

void Widget::stateChanged(QAudio::State state)
{
    if( state == QAudio::IdleState || state == QAudio::ActiveState ) {
        ui->volSlider->setEnabled(true);
        ui->pbtSendStart->setEnabled(false);
        ui->pbtSendStop->setEnabled(true);
    } else if(state == QAudio::StoppedState) {
        ui->volSlider->setEnabled(false);
        ui->pbtSendStart->setEnabled(true);
        ui->pbtSendStop->setEnabled(false);
    }
}

void Widget::readMore()
```

```
{  
    if(!mAudioInput)  
        return;  
  
    qint64 len = m_audioSource->bytesAvailable();  
    if(len > 4096) len = 4096;  
  
    qint64 readLen = mInput->read(mBuffer.data(), len);  
    if(readLen > 0) {  
        mUdpSocket.writeDatagram(mBuffer.data(),  
            len,  
            QHostAddress::LocalHost,  
            15000);  
    }  
}  
  
Widget::~Widget()  
{  
    delete ui;  
}
```

The audioInitialize( ) member function is called in the constructor function. In this function, we get a list of Audio Input Devices and register them in the Combo Box. To get the list of Audio Input Devices, we use the QMediaDevices class.

The QAudioInput class is provided for the purpose of extracting incoming audio data from an input device such as a Microphone.

It is the class that reads the actual audio data from the QIODevice. So, the QIODevice class stores the microphone signal in a QByteArray.

It then sends the data stored in the QByteArray over the network (UDP).

The volSliderChanged( ) Slot function provides the ability to adjust the volume on the GUI. The sendStartBtn( ) Slot function is called by clicking the [Start sending] button on the GUI. In this function, the Microphone device selected in the Combo box is passed as the first argument when declaring an object of QAudioInput class.

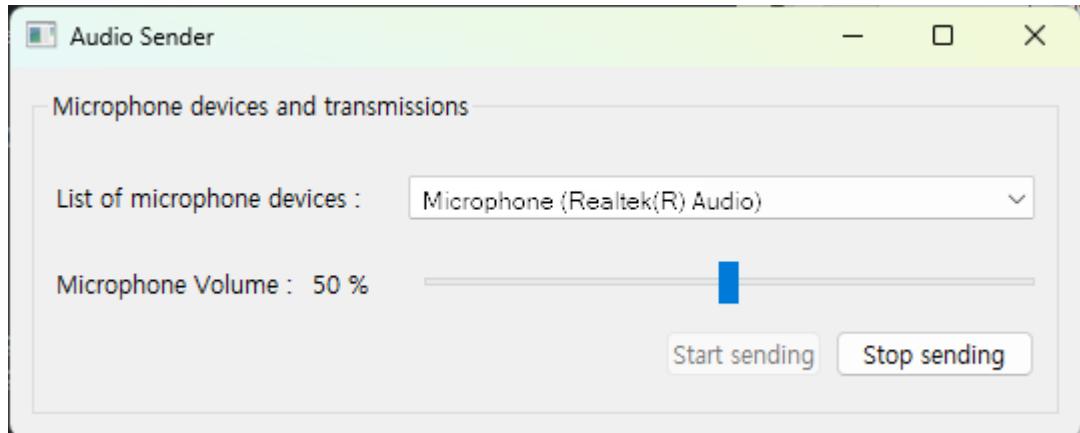
The QAudioInput class object then extracts data from the Microphone device. The data

Jesus loves you.

is extracted by Signal and Slot.

For example, when extracted from a Microphone device, the readyRead( ) Signal is called, which in turn calls the readMore( ) Slot function associated with this Signal.

The readMore( ) Slot function sends the data extracted from the Microphone device over the network.

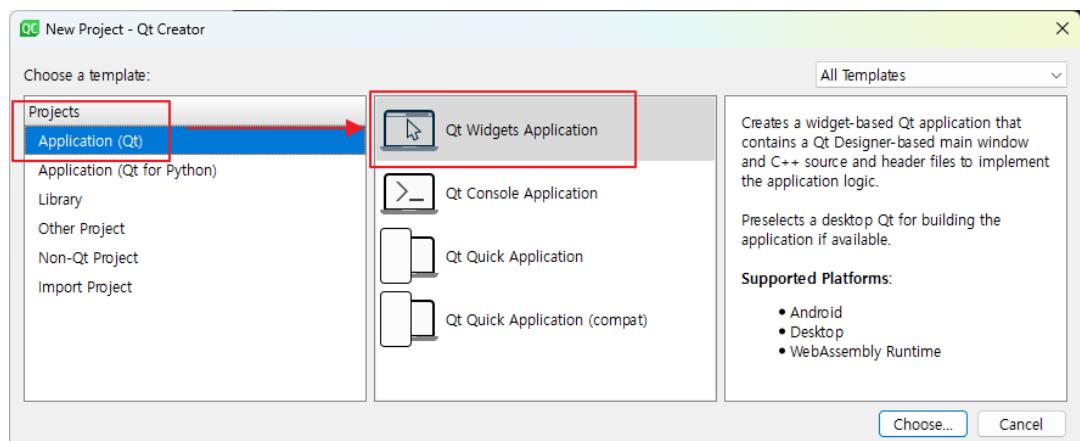


You can find the source code for this example in the 01\_AudioSender directory.

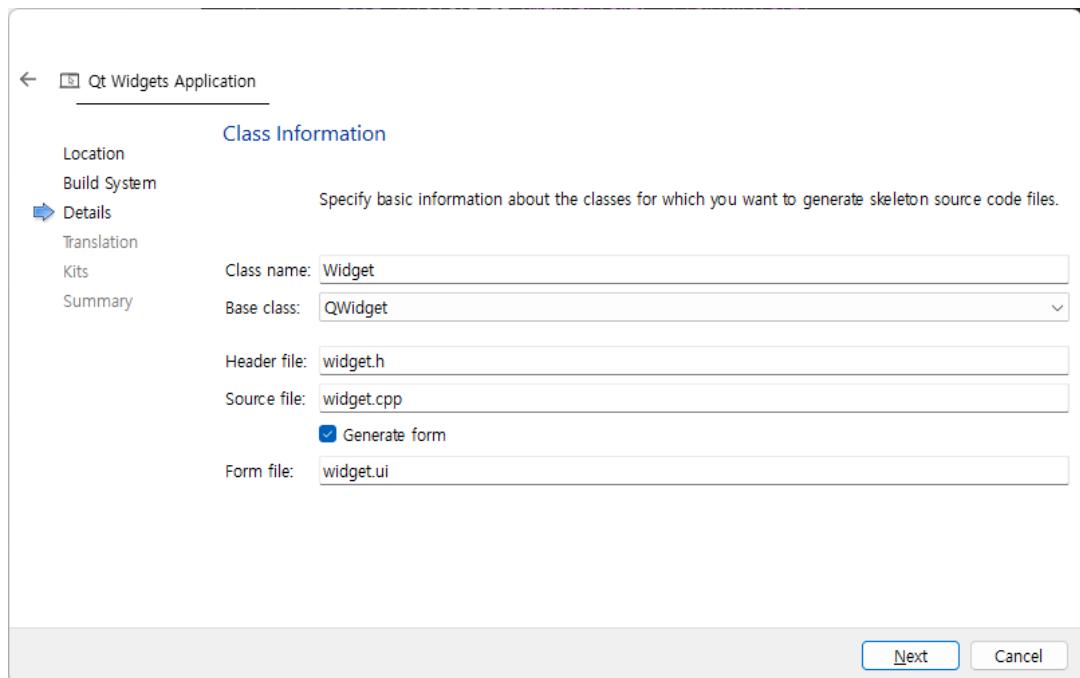
- ✓ Example of outputting data received over the network to a speaker

This time, let's write an example that outputs the data received from the sending example to the speaker.

Create a project based on Qt Widget.



In this project, we will use UI Form, so check [Generation form] as shown below.



Once the project is created, add the Multimedia module and Network module to the CMakeLists.txt file as shown below.

```
cmake_minimum_required(VERSION 3.5)

project(02_AudioReceiver VERSION 0.1 LANGUAGES CXX)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Qt6 REQUIRED COMPONENTS
    Widgets
    Multimedia
    Network
)

set(PROJECT_SOURCES
```

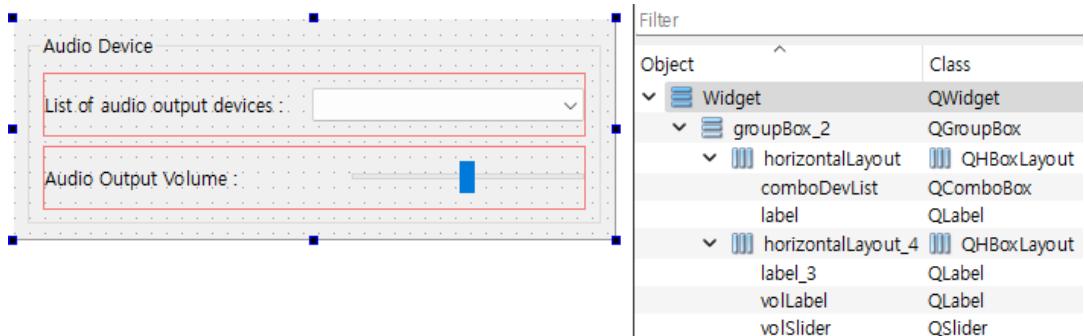
```
main.cpp
widget.cpp
widget.h
widget.ui
)

qt_add_executable(02_AudioReceiver
    ${PROJECT_SOURCES}
)

target_link_libraries(02_AudioReceiver PRIVATE
    Qt6::Widgets
    Qt6::Multimedia
    Qt6::Network
)

install(TARGETS 02_AudioReceiver
    BUNDLE DESTINATION .
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

Next, open the widget.ui file and place the GUI widget as shown below.



Next, create the widget.h header file as shown below.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
```

```
#include <QtMultimedia>
#include <QtNetwork>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;

    QMediaDevices      *m_devices = nullptr;
    QList<QAudioDevice> mDevList;
    QAudioDevice       mDeviceInfo;
    float              mOutputVolume;
    QAudioFormat        mFormat;
    QAudioSink          *mAudioOutput;
    QIODevice          *mOutput;

    void audioInitialize();
    void audioOutputProcess(const QByteArray &ba);

    QUdpSocket *mUdpSocket;
    void networkInitialize();

private slots:
    void devListIndexChanged(int index);
    void volSliderChanged(int val);
    void readUdpData();
```

```
};  
#endif // WIDGET_H
```

Next, create the widget.cpp source code file as shown below.

```
#include "widget.h"  
#include "./ui_widget.h"  
  
Widget::Widget(QWidget *parent)  
    : QWidget(parent)  
    , ui(new Ui::Widget)  
{  
    ui->setupUi(this);  
  
    audioInitialize();  
    networkInitialize();  
  
    connect(ui->comboDevList, SIGNAL(currentIndexChanged(int)),  
            this,           SLOT(devListIndexChanged(int)));  
    connect(ui->volSlider,   &QSlider::valueChanged,  
            this,           &Widget::volSliderChanged);  
}  
  
void Widget::audioInitialize()  
{  
    m_devices = new QMediaDevices(this);  
    mDevList = m_devices->audioOutputs();  
  
    for(int i = 0; i < mDevList.size(); ++i)  
        ui->comboDevList->addItem(mDevList.at(i).description());  
  
    mDevInfo = mDevList.at(ui->comboDevList->currentIndex());  
  
    mFormat.setSampleRate(8000);  
    mFormat.setChannelCount(1);  
    mFormat.setSampleFormat(QAudioFormat::Int16);
```

```
mAudioOutput = new QAudioSink(mDevInfo, mFormat, this);

mOutput = mAudioOutput->start();

mOutputVolume = ui->volSlider->value() / 100.0;
ui->volLabel->setText(QString("%1 %")
                        .arg(ui->volSlider->value()));

mAudioOutput->setVolume(mOutputVolume);
}

void Widget::networkInitialize()
{
    mUdpSocket = new QUdpSocket(this);
    mUdpSocket->bind(QHostAddress::LocalHost, 15000);

    connect(mUdpSocket, SIGNAL(readyRead()),
            this,           SLOT(readUdpData()));
}

void Widget::devListIndexChanged(int index)
{
    mDevInfo = mDevList.at(index);

    if(mOutput != nullptr) {
        disconnect(mOutput, nullptr, this, nullptr);
        mOutput = nullptr;
    }

    mAudioOutput->stop();
    mAudioOutput->disconnect(this);
    delete mAudioOutput;

    mAudioOutput = new QAudioSink(mDevInfo, mFormat, this);
    mAudioOutput->setVolume(mOutputVolume);
```

```
mOutput = mAudioOutput->start();

}

void Widget::volSliderChanged(int val)
{
    mOutputVolume = val / 100.0;
    mAudioOutput->setVolume(mOutputVolume);

    ui->volLabel->setText(QString("%1 %").arg(val));
}

void Widget::readUdpData()
{
    while (mUdpSocket->hasPendingDatagrams())
    {
        QNetworkDatagram datagram;
        datagram = mUdpSocket->receiveDatagram();

        QByteArray audioData = datagram.data();
        audioOutputProcess(audioData);
    }
}

void Widget::audioOutputProcess(const QByteArray &ba)
{
    qint64 len = ba.size();
    if(len > 0) {
        mOutput->write(ba.data(), ba.size());
    }
}

Widget::~Widget()
{
```

Jesus loves you.

The audiointialize( ) member function gets a list of devices to be output and registers them in the Combo box.

Among the registered items in the Combo box, the device currently selected in the Combo box is set as the device to be output.

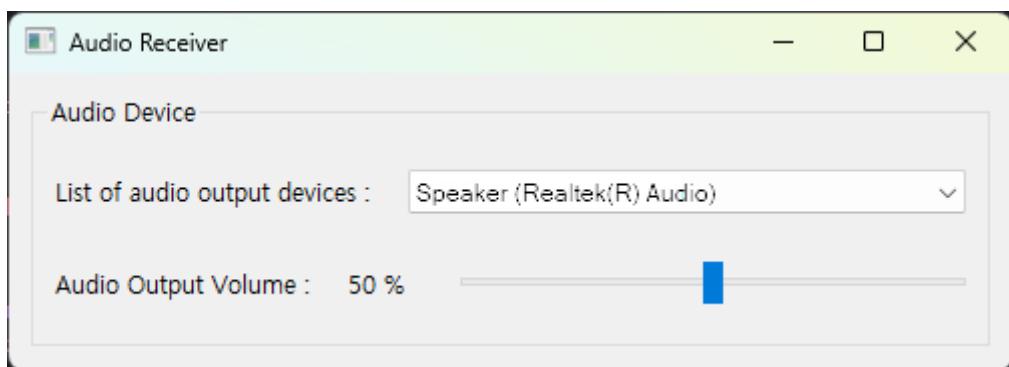
The QAudioFormat class sets the sample rate, channel counts, and number of bytes per sample for the device to be output.

Next, the QAudioSink class sets up the device to output to.

It passes an object of class QAudioOutput as its first argument. The second argument is an object of class QAudioFormat.

Finally, to output the QByteArray data read from the network to the Speaker, we use the QIODevice class. Therefore, at the end of the audiointialize( ) member function, an object pointer of class QIODevice is obtained from class QAudioSink and stored.

The readUdpData( ) Slot function is called when data is received from the UDP protocol. It passes the received data as the first argument to the write( ) member function provided by the QIODevice class. The data is then output to the Speaker.



Let's run the application and check if the data received from the Microphone device is output to the Speaker correctly, as shown in the figure below. You can find the source code for this example in the 02\_AudioReceiver directory.

## 38.2. Video

Before we look at the multimedia features for video provided by Qt, let's take a look at the codecs needed to play video.

Video files are compressed. When you play a video file, the application playing the video does real-time decoding internally to render it on the screen. Conversely, when recording, the video is encoded (i.e. compressed).

Therefore, to decode or encode a video, you need a codec. There are many different types of codecs, and even now, improved codecs are being developed to further increase file compression.

Qt does not provide many video codecs with Qt due to licensing issues. This is also true for other development frameworks.

If a video playback application implemented in Qt cannot decode a video file, it is because your platform (operating system) does not have the necessary codecs to play the video.

There are several video codecs available for free. You can use the K-Lite Codec Pack ([www.codecguide.com](http://www.codecguide.com)) for free as a collection of integrated codecs.

It offers Basic, Standard, and Full versions, and we'll download and install the Full version.

Coming back to Qt, it provides a Multimedia GUI widget to display the video playback on the GUI. To use the Multimedia GUI, we need to use the Multimedia module and the MultimediaWidgets module together in the project file as shown below.

```
find_package(Qt6 REQUIRED COMPONENTS
    Widgets
    Multimedia
    MultimediaWidgets
)

target_link_libraries(00_VideoPlayer PRIVATE
    Qt6::Widgets
    Qt6::Multimedia
```

```
Qt6::MultimediaWidgets  
)
```

To implement an application that plays videos in Qt, we need to use the QMediaPlayer class.

And when we play a movie, we need a video screen and sound. Therefore, we need to set up a QVideoWidget class object in the QMediaPlayer class to render the video screen.

You can use the setVideoOutput( ) member function of the QMediaPlayer class to connect an object of the QVideoWidget class.

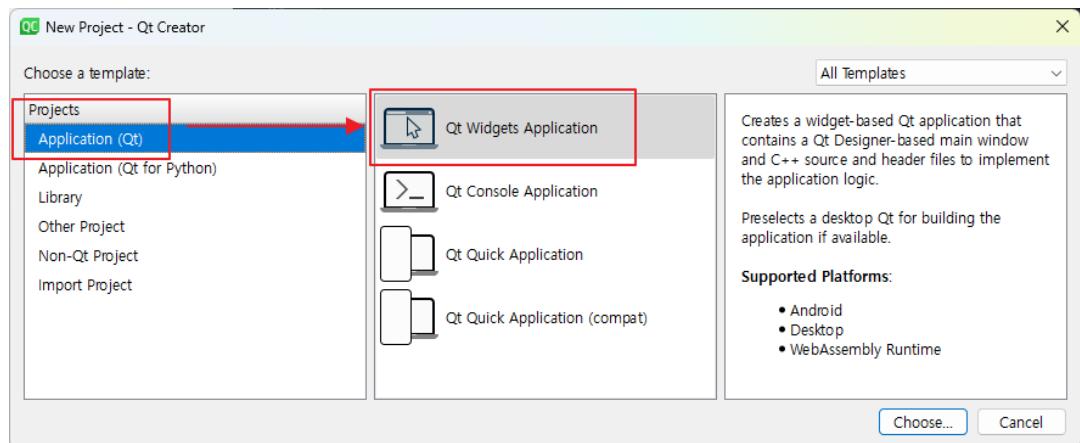
And to play sound when the video plays, you can use the QAudioOutput class. To play sound, you can pass an object of class QAudioOutput as the first argument to the setAudioOutput( ) member function provided by class QMediaPlayer.

```
player = new QMediaPlayer;  
  
vWidget = new QVideoWidget();  
player->setVideoOutput(vWidget);  
  
audioOutput = new QAudioOutput;  
player->setAudioOutput(audioOutput);  
...  
audioOutput->setVolume(50);  
...  
player->setSource(QUrl::fromLocalFile("d:/sample.mp4"));  
player->play();
```

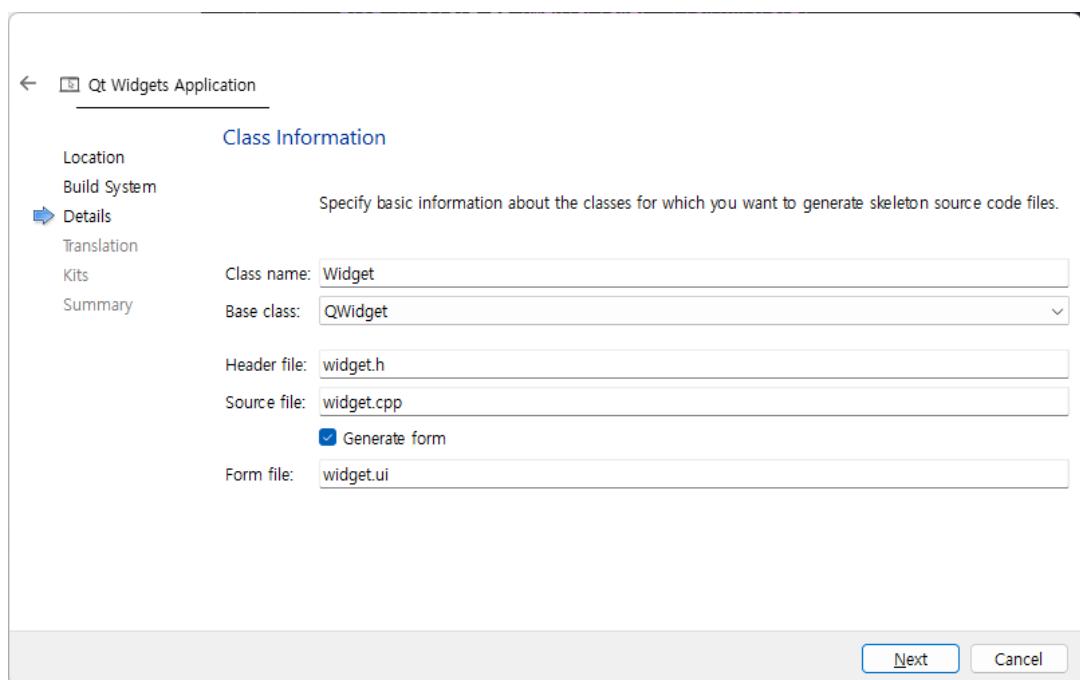
#### ✓ Video Player Example

In this example, we will create an example that uses the QMediaPlayer class to play a video file.

Create a project based on Qt Widget.



In this project, we will use UI Form, so check [Generation form] as shown below.



Once the project is created, add the Multimedia module and MultimediaWidgets module to the project file as shown below.

```
cmake_minimum_required(VERSION 3.5)

project(00_VideoPlayer VERSION 0.1 LANGUAGES CXX)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTOMOC ON)
```

```
set(CMAKE_AUTORCC ON)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Qt6 REQUIRED COMPONENTS
    Widgets
    Multimedia
    MultimediaWidgets
)

set(PROJECT_SOURCES
    main.cpp
    widget.cpp
    widget.h
    widget.ui
)

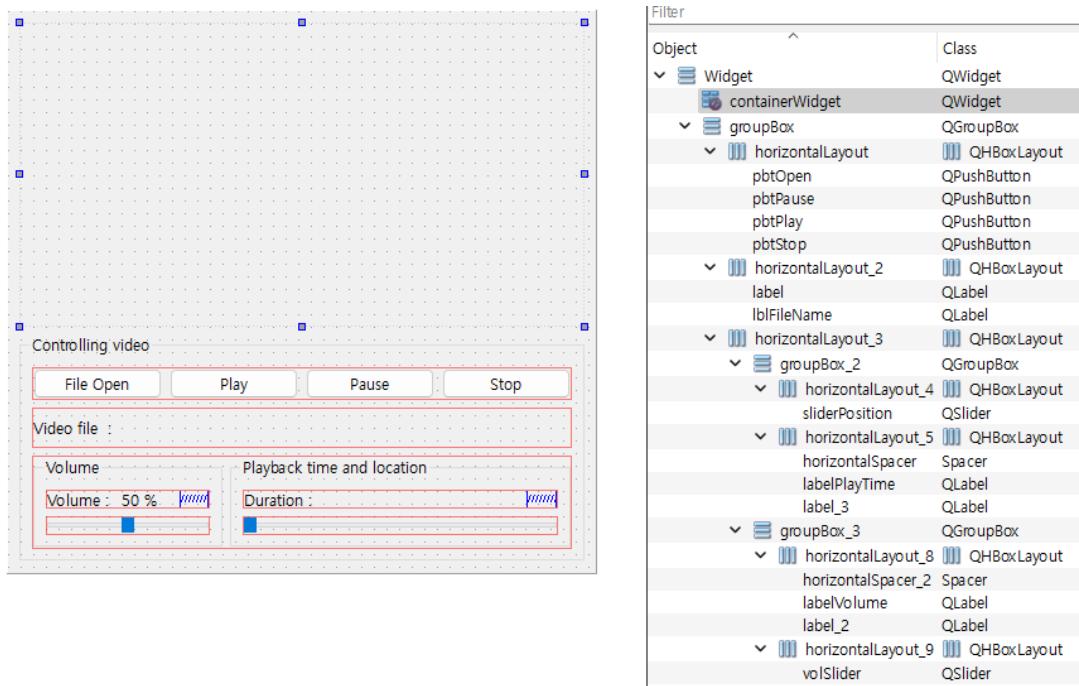
qt_add_executable(00_VideoPlayer
    ${PROJECT_SOURCES}
)

target_link_libraries(00_VideoPlayer PRIVATE
    Qt6::Widgets
    Qt6::Multimedia
    Qt6::MultimediaWidgets
)

install(TARGETS 00_VideoPlayer
    BUNDLE DESTINATION .
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

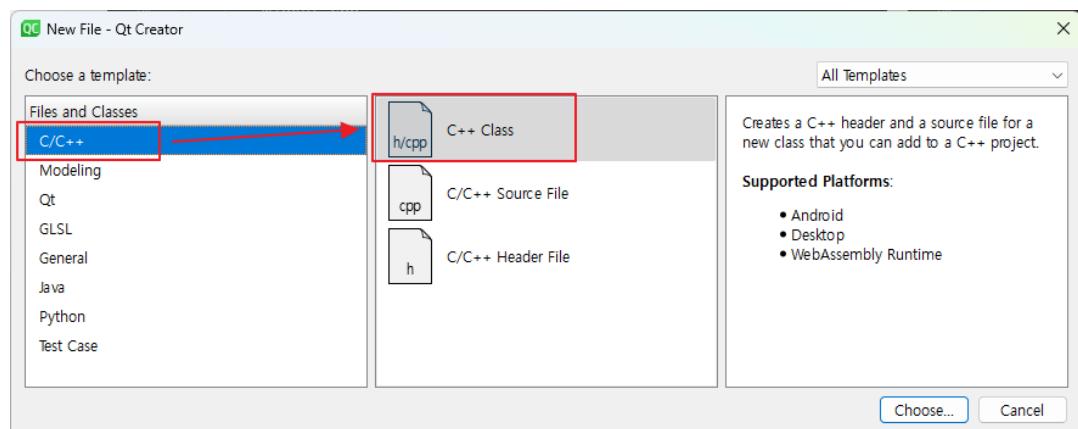
Next, open the `widget.ui` file and place the GUI widgets as shown below.

Jesus loves you.



To display the video on the GUI, we used the QVideoWidget. And if you double-click on the part where the video is displayed, the area where the video is displayed changes to full screen.

And if you click the ESC key in the full screen state, it will return to the original size screen. To implement these features, we implemented the DisplayWidget class, which inherits from the QVideoWidget class. Add the DisplayWidget class to your project like below.



Add the DisplayWidget class and write the following in the displaywidget.h header file

```
#ifndef DISPLAYWIDGET_H
```

```
#define DISPLAYWIDGET_H

#include <QVideoWidget>

class DisplayWidget : public QVideoWidget
{
    Q_OBJECT
public:
    DisplayWidget();

protected:
    void keyPressEvent(QKeyEvent *event) override;
    void mouseDoubleClickEvent(QMouseEvent *event) override;
    void mousePressEvent(QMouseEvent *event) override;
};

#endif // DISPLAYWIDGET_H
```

The mouseDoubleClickEvent( ) virtual function is called when the mouse is double-clicked.

When the mouse is double-clicked, the video rendering widget is switched to full screen.

Create the displaywidget.cpp source code file as shown below.

```
#include "displaywidget.h"
#include <QMouseEvent>

DisplayWidget::DisplayWidget()
{
    setSizePolicy(QSizePolicy::Ignored, QSizePolicy::Ignored);

    QPalette p = palette();
    p.setColor(QPalette::Window, Qt::black);
    setPalette(p);

    setAttribute(Qt::WA_OpaquePaintEvent);
}

void DisplayWidget::keyPressEvent(QKeyEvent *event)
{
    if (event->key() == Qt::Key_Escape && isFullScreen()) {
        setFullScreen(false);
        event->accept();
    }
    else if (event->key() == Qt::Key_Enter &&
```

```
        event->modifiers() & Qt::Key_Alt)
{
    setFullScreen(!isFullScreen());
    event->accept();
} else {
    QVideoWidget::keyPressEvent(event);
}
}

void DisplayWidget::mouseDoubleClickEvent(QMouseEvent *event)
{
    setFullScreen(!isFullScreen());
    event->accept();
}

void DisplayWidget::mousePressEvent(QMouseEvent *event)
{
    QVideoWidget::mousePressEvent(event);
}
```

The keyPressEvent( ) is called when a keyboard event occurs and will switch to the original screen size when the ESC (Escape) key or ALT + Enter key is clicked.

Next, create the widget.h header file as shown below.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QMediaPlayer>
#include <QAudioOutput>
#include "displaywidget.h"

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
```

```

~Widget();

private:
    Ui::Widget *ui;

    QString      m_fName;
    QMediaPlayer *m_player;
    DisplayWidget *m_displayWidget;
    QAudioOutput *m_audioOutput;
    qint64        m_duration;

private slots:
    void onOpenBtn();
    void onPlayBtn();
    void onPauseBtn();
    void onStopBtn();

    void sliderValueChange(int val);

    void durationChanged(qint64 duration);
    void positionChanged(qint64 progress);

    void seek(int seconds);
};

#endif // WIDGET_H

```

The `onOpenBtn()` Slot function is a Slot function that is called when the [File Open] button is clicked. This function uses the `QFileDialog` class to select a movie file in a file selection dialog.

The `onPlayBtn()` function plays the movie file. `onPauseBtn()` pauses the movie playback and `onStopBtn()` stops the movie. Next, write the `widget.cpp` source code as shown below.

```

#include "widget.h"
#include "./ui_widget.h"
#include <QFileDialog>
#include <QTime>

Widget::Widget(QWidget *parent) : QWidget(parent), ui(new Ui::Widget)
{

```

```
m_duration = 0;

ui->setupUi(this);
connect(ui->volSlider, SIGNAL(valueChanged(int)),
        this,           SLOT(sliderValueChange(int)));

ui->sliderPosition->setEnabled(false);

connect(ui->pbtOpen,   SIGNAL(clicked()), this, SLOT(onOpenBtn()));
connect(ui->pbtPlay,   SIGNAL(clicked()), this, SLOT(onPlayBtn()));
connect(ui->pbtPause,  SIGNAL(clicked()), this, SLOT(onPauseBtn()));
connect(ui->pbtStop,   SIGNAL(clicked()), this, SLOT(onStopBtn()));

m_displayWidget = new DisplayWidget();

QVBoxLayout *vLay = new QVBoxLayout();
vLay->addWidget(m_displayWidget);

ui->containerWidget->setLayout(vLay);

m_player = new QMediaPlayer();
m_audioOutput = new QAudioOutput;
m_player->setAudioOutput(m_audioOutput);
m_audioOutput->setVolume(0.5);

m_player->setVideoOutput(m_displayWidget);

connect(m_player, &QMediaPlayer::durationChanged,
        this,       &Widget::durationChanged);

connect(m_player, &QMediaPlayer::positionChanged,
        this,       &Widget::positionChanged);

connect(ui->sliderPosition, &QSlider::sliderMoved,
        this,           &Widget::seek);
}

void Widget::sliderValueChange(int val)
{
    ui->labelVolume->setText(QString("%1 %").arg(val));
    m_audioOutput->setVolume(val / 100.0);
}
```

```
void Widget::onOpenBtn()
{
    m_fName = QFileDialog::getOpenFileName(this,
                                            tr("Open File"),
                                            QDir::homePath());
    if(!m_fName.isNull())
        ui->lblFileName->setText(m_fName);
}

void Widget::onPlayBtn()
{
    if(!m_fName.isNull())
    {
        m_player->setSource(QUrl::fromLocalFile(m_fName));
        ui->sliderPosition->setEnabled(true);
        m_player->play();
    }
}

void Widget::onPauseBtn()
{
    int state = m_player->playbackState();
    if(state == QMediaPlayer::PausedState)
        m_player->play();
    else if(state == QMediaPlayer::PlayingState)
        m_player->pause();
}

void Widget::onStopBtn()
{
    int state = m_player->playbackState();
    if(state == QMediaPlayer::PlayingState)
    {
        m_player->stop();
    }
}

void Widget::durationChanged(qint64 duration)
{
    m_duration = duration / 1000;
    ui->sliderPosition->setMaximum(m_duration);
```

```
}

void Widget::positionChanged(qint64 progress)
{
    if ( !ui->sliderPosition->isSliderDown() )
        ui->sliderPosition->setValue(progress / 1000);

    qint64 currentInfo = progress / 1000;
    QString playTimeStr;
    if (currentInfo || m_duration) {
        QTime currentTime((currentInfo / 3600) % 60,
                           (currentInfo / 60) % 60,
                           currentInfo % 60,
                           (currentInfo * 1000) % 1000);

        QTime totalTime((m_duration / 3600) % 60,
                        (m_duration / 60) % 60,
                        m_duration % 60,
                        (m_duration * 1000) % 1000);

        QString format = "mm:ss";
        if (m_duration > 3600) format = "hh:mm:ss";

        playTimeStr = currentTime.toString(format) + " / " +
                      totalTime.toString(format);
    }

    ui->labelPlayTime->setText(playTimeStr);
}

void Widget::seek(int seconds)
{
    m_player->setPosition(seconds * 1000);
}

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

In the class constructor, you place the DisplayWidget on the GUI and pass an object of the DisplayWidget class as the first argument to the setVideoOutput( ) function of the

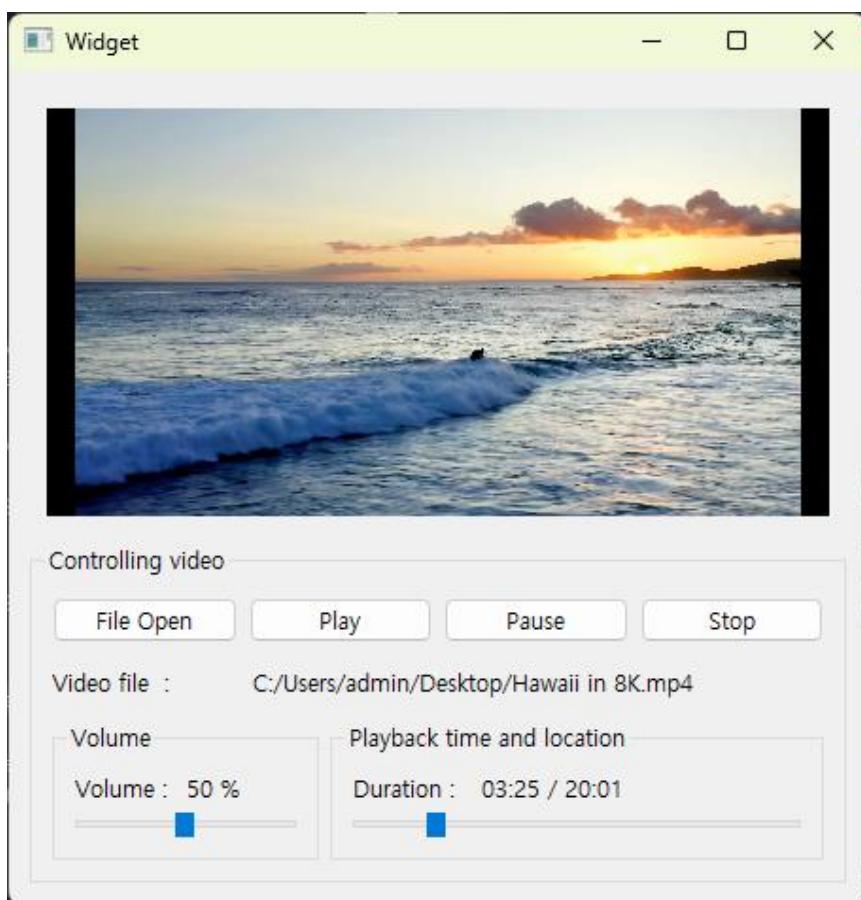
Jesus loves you.

QMediaPlayer class, which causes the QMediaPlayer class to play the video.

When the QMediaPlayer plays the video, it outputs the decoded result to a widget of class DisplayWidget.

The sliderValueChange( ) function allows the volume to be adjusted on the GUI. The durationChanged( ) function is called when the playback position of the movie file changes randomly.

positionChanged( ) provides a function to tell the current playing position. In this Slot function, we change the overall time to "hours:minutes:seconds" and output it to the GUI widget.



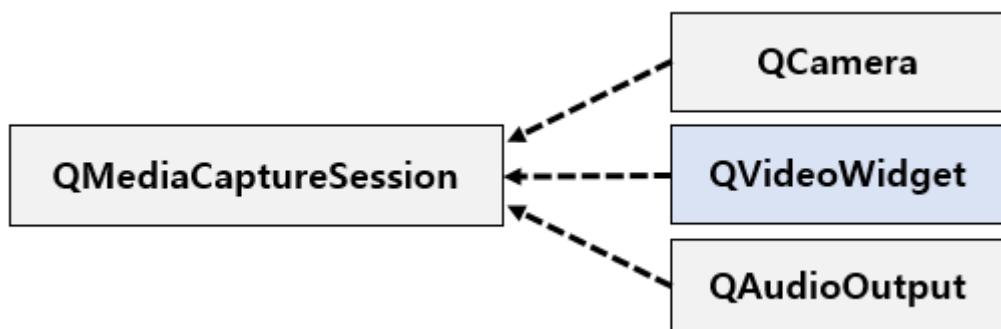
You can find the source code for this example in the 00\_VideoPlayer directory.

### 38.3. Camera

The Qt Multimedia module provides APIs for developing applications using camera devices.

To control the camera device, `QMediaCaptureSession` class is provided. This class uses `QCamera` class to extract the camera image and `QVideoWidget` class to display the extracted image on the GUI. It uses the `QVideoWidget` class to display the extracted video on the GUI.

The Camera device comes with a Microphone device. To process the data input from the Microphone device included in the Camera device, you need to connect a `QAudioInput` class object to the `QMediaCaptureSession`.



Finally, to start the Camera device, you can use the `start( )` member function provided by the `QCamera` class.

```

QMediaCaptureSession          m_captureSession;
QScopedPointer<QAudioInput>   m_audioInput;
QScopedPointer<QCamera>        mCamera;
QVideoWidget                  *mVideoWidget;
...
m_audioInput.reset(new QAudioInput);
mCamera.reset(new QCamera(0));
mVideoWidget = new QVideoWidget();

m_captureSession.setAudioInput(m_audioInput.get());
...
m_captureSession.setCamera(mCamera.data());
  
```

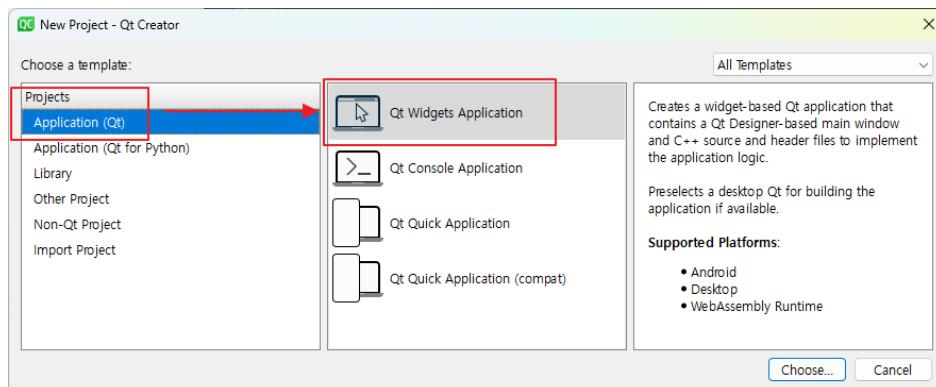
```
m_captureSession.setVideoOutput(mVideoWidget);  
...  
mCamera->start();  
...
```

Next, let's implement an example that deals with video from a real Camera Device.

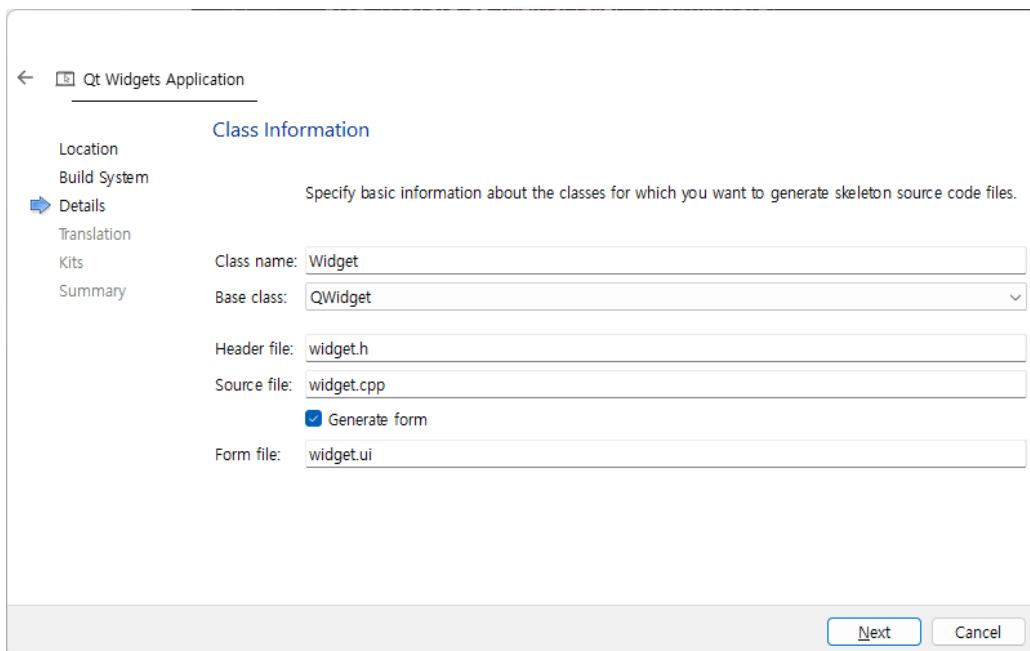
- ✓ Example using the Camera device

In this example, we will implement an example of displaying video from a USB Camera device on the GUI.

Create a project based on Qt Widget.



In this project, we will use UI Form, so check [Generation form] as shown below.



Once the project is created, add the Multimedia module and MultimediaWidgets module to the project file as shown below.

```
cmake_minimum_required(VERSION 3.5)

project(00_CameraCapture VERSION 0.1 LANGUAGES CXX)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Qt6 REQUIRED COMPONENTS
    Widgets
    Multimedia
    MultimediaWidgets
)

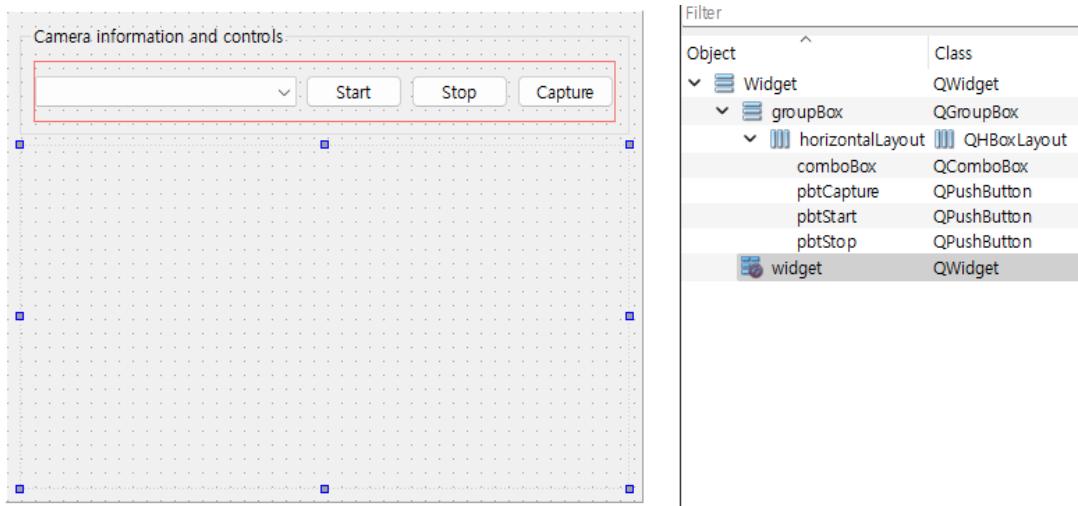
set(PROJECT_SOURCES
    main.cpp
    widget.cpp
    widget.h
    widget.ui
)

qt_add_executable(00_CameraCapture
    ${PROJECT_SOURCES}
)

target_link_libraries(00_CameraCapture PRIVATE
    Qt6::Widgets
    Qt6::Multimedia
    Qt6::MultimediaWidgets
)

install(TARGETS 00_CameraCapture
    BUNDLE DESTINATION .
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

Next, open the `widget.ui` file and place the GUI widgets as shown below.



Next, create the `widget.h` header file as shown below.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QVideoWidget>
#include <QCamera>
#include <QImageCapture>
#include <QMediaCaptureSession>
#include <QAudioInput>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
```

```
QList<QCameraDevice>      mCamList;

QVideoWidget                 *m_videoWidget;

QScopedPointer<QImageCapture>   m_imageCapture;
QMediaCaptureSession           m_captureSession;
QScopedPointer<QCamera>        mCamera;
QScopedPointer<QAudioInput>    m_audioInput;

void cameraDevicesSearch();
void viewCaptureImgDialog(const QImage &img);

private slots:
void onStartBtn();
void onStopBtn();
void onCaptureBtn();
void camError();

void imageCaptured(int requestId, const QImage &img);
};

#endif // WIDGET_H
```

다음으로 widget.cpp 소스코드 파일을 열어서 아래와 같이 작성한다.

```
#include "widget.h"
#include "./ui_widget.h"
#include <QMediaDevices>
#include <QMMessageBox>
#include <QLabel>
#include <QHBoxLayout>

Widget::Widget(QWidget *parent) : QWidget(parent), ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->pbtStart, SIGNAL(clicked()), this, SLOT(onStartBtn()));
    connect(ui->pbtStop,  SIGNAL(clicked()), this, SLOT(onStopBtn()));
    connect(ui->pbtCapture, SIGNAL(clicked()), this, SLOT(onCaptureBtn()));

    cameraDevicesSearch();

    ui->pbtStart->setEnabled(true);
    ui->pbtStop->setEnabled(false);
    ui->pbtCapture->setEnabled(false);
```

```
m_videoWidget = new QVideoWidget();
QHBoxLayout *hLay = new QHBoxLayout;
hLay->addWidget(m_videoWidget);
ui->widget->setLayout(hLay);
}

void Widget::cameraDevicesSearch()
{
    ui->comboBox->clear();
    mCamList = QMediaDevices::videoInputs();
    foreach(const QCameraDevice &camDev, mCamList)
        ui->comboBox->addItem(camDev.description());
}

void Widget::viewCaptureImgDialog(const QImage &img)
{
    QDialog viewDialog(this, Qt::Dialog);
    viewDialog.setWindowTitle("Capture Image");

    QLabel doneLabel("");
    doneLabel.setPixmap(QPixmap::fromImage(img));

    QBoxLayout lay;
    lay.addWidget(&doneLabel);

    viewDialog.setLayout(&lay);
    viewDialog.resize(img.size());
    viewDialog.exec();
}

void Widget::onStartBtn()
{
    m_audioInput.reset(new QAudioInput);
    m_captureSession.setAudioInput(m_audioInput.get());

    int currCamIndex = ui->comboBox->currentIndex();

    mCamera.reset(new QCamera(mCamList.at(currCamIndex)));
    connect(mCamera.data(), &QCamera::errorOccurred,
            this,             &Widget::camError);
}
```

```
m_captureSession.setCamera(mCamera.data());
m_captureSession.setVideoOutput(m_videoWidget);

if (!m_imageCapture) {
    m_imageCapture.reset(new QImageCapture);
    m_captureSession.setImageCapture(m_imageCapture.get());

    connect(m_imageCapture.get(), &QImageCapture::imageCaptured,
            this,                      &Widget::imageCaptured);
}

mCamera->start();

ui->pbtStart->setEnabled(false);
ui->pbtStop->setEnabled(true);
ui->pbtCapture->setEnabled(true);
}

void Widget::onStopBtn()
{
    mCamera->stop();
    m_videoWidget->clearMask();

    ui->pbtStart->setEnabled(true);
    ui->pbtStop->setEnabled(false);
    ui->pbtCapture->setEnabled(false);
}

void Widget::onCaptureBtn()
{
    m_imageCapture->capture();
}

void Widget::camError()
{
    if ( mCamera->error() != QCamera::.NoError ) {
        QMessageBox::warning(this,
                            tr("Camera Error"),
                            mCamera->errorString());
    }
}

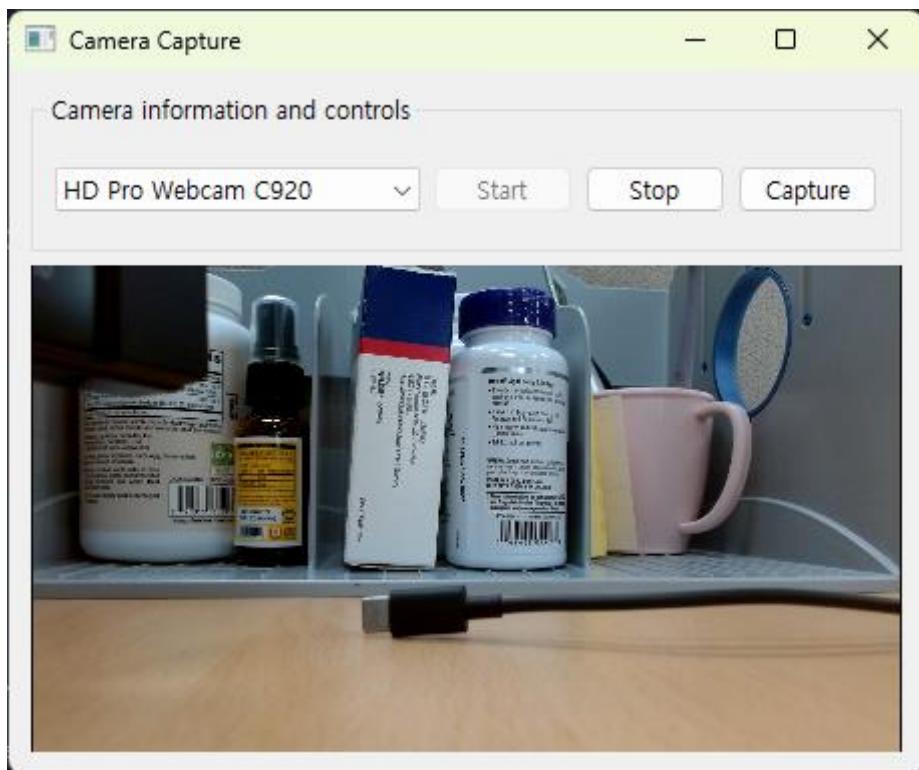
void Widget::imageCaptured(int requestId, const QImage &img)
```

```
{  
    QImage scaledImgImage = img.scaled(m_videoWidget->size(),  
                                         Qt::KeepAspectRatio,  
                                         Qt::SmoothTransformation);  
    viewCaptureImgDialog(scaledImgImage);  
}  
  
Widget::~Widget()  
{  
    delete ui;  
}
```

The cameraDevicesSearch( ) function registers all camera devices found in the system to a widget of class QComboBox.

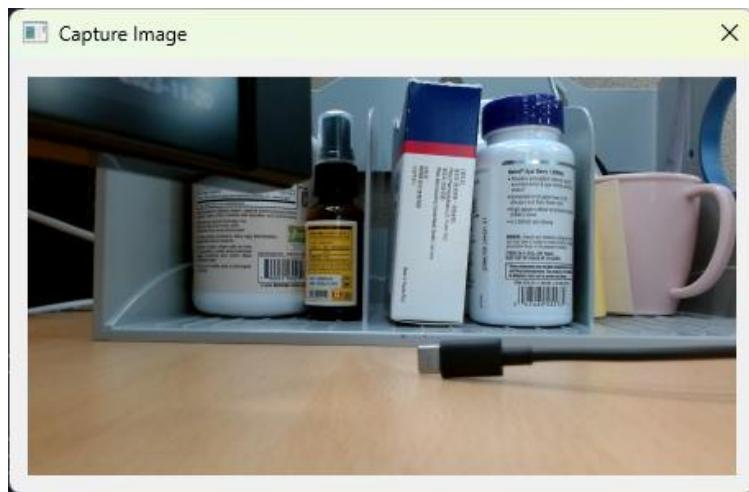
The onStartBtn( ) function is called when the [Start] button is clicked. The onStopBtn( ) function is called when the [Stop] button is clicked.

The viewCaptureImgDialog( ) function is called when the [Capture] button is clicked. This function displays the captured image in a dialog when this function is called while the Camera device is active.



Jesus loves you.

Click the [Capture] button to display the captured image on the dialog as shown below.  
You can refer to the 00\_CameraCapture directory for the source code of this example.



## 39. Serial Communication

The SerialPort module provided by Qt can transmit and receive data via control signals on the RS-232 pinout. Serial communication is only used for communication between physically separated computers or embedded devices.

The main advantage of the Qt SerialPort module is that it is compatible with MS Windows, Linux and MacOS platforms. For example, source code implemented on MS Windows using the classes provided by the Qt SerialPort module is compatible with Linux and MacOS.

To use serials such as the QSerialPort class provided by Qt, you need to add the following to your project file

```
find_package(Qt6 REQUIRED COMPONENTS SerialPort)
target_link_libraries(mytarget PRIVATE Qt6::SerialPort)
```

If you're using qmake, you'll need to add the following

```
QT += serialport
```

QSerialPort also uses Signal and Slot. The following is an example of connecting a device using the QSerialPort class.

```
QSerialPort *m_serial = new QSerialPort(this);

connect(m_serial, &QSerialPort::errorOccurred, this, &MainWindow::handleError);
connect(m_serial, &QSerialPort::readyRead, this, &MainWindow::readData);

QString name      = QString("COM1");
qint32 baudRate = QSerialPort::Baud115200;
QSerialPort::DataBits dataBits  = QSerialPort::Data8;
QSerialPort::Parity parity      = QSerialPort::NoParity;
QSerialPort::StopBits stopBits  = QSerialPort::OneStop;

QSerialPort::FlowControl flowControl = QSerialPort::NoFlowControl;

m_serial->setPortName(name);
m_serial->setBaudRate(baudRate);
```

```
m_serial->setDataBits(dataBits);  
m_serial->setParity(parity);  
m_serial->setStopBits(stopBits);  
m_serial->setFlowControl(flowControl);  
  
m_serial->open(QIODevice::ReadWrite);
```

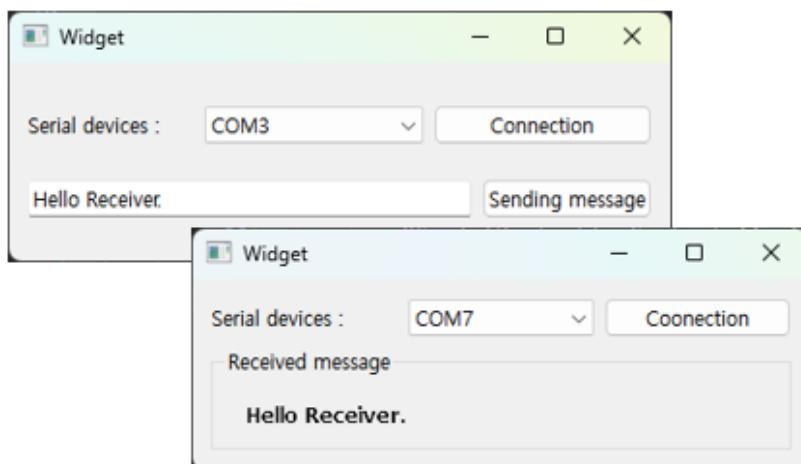
The QSerialPort class raises the errorOccurred( ) signal when an error occurs, as shown in the example source code above.

And the readyRead( ) signal can be used to receive the data transmitted by the other party. The serial must be set to the same as the other side to make a serial connection.

The setPortName( ) member function takes the name of the serial device to use (for example, /dev/USB0 on Linux). The setBaudRate( ) member function allows you to specify the baud rate. The setDataBits( ) member function sets the number of data bits to use per character, setParity( ) sets the parity setting to use, and so on.

To complete the above settings and use serial communication, open the device for sending/receiving data using the open( ) function that provides the QSerialPort class. Let's implement serial communication through the following example.

- ✓ Example of sending and receiving data using the QSerialPort class



In this example, we will use two applications to send and receive data. The first example application sends data over serial. The second application outputs the data received on the serial port to the GUI.

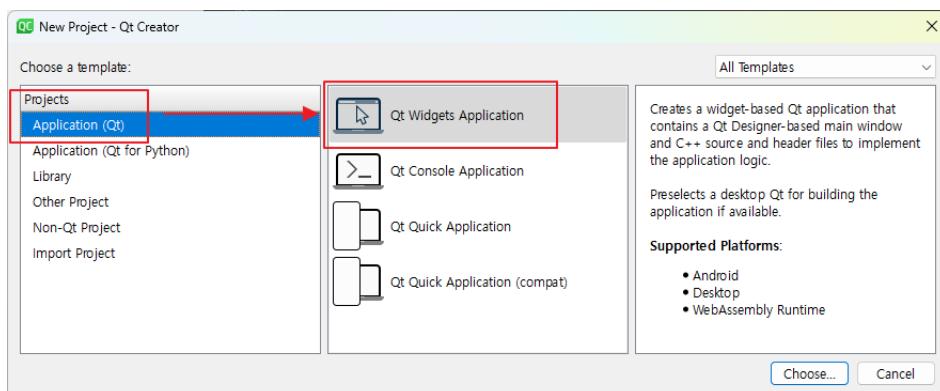
In the figure above, the left side is an example of Sender. Select the device you want to use on the GUI and click the [Connection] button. Similarly, for the Receiver example on the right, select the device to be used on the GUI and click the [Connection] button, and you are ready to send/receive data between the two applications.

After entering the message to be sent in the QLineEdit widget at the bottom of the first Sender application, click the [Sending message] button to send the message to the Receiver as shown in the figure above.

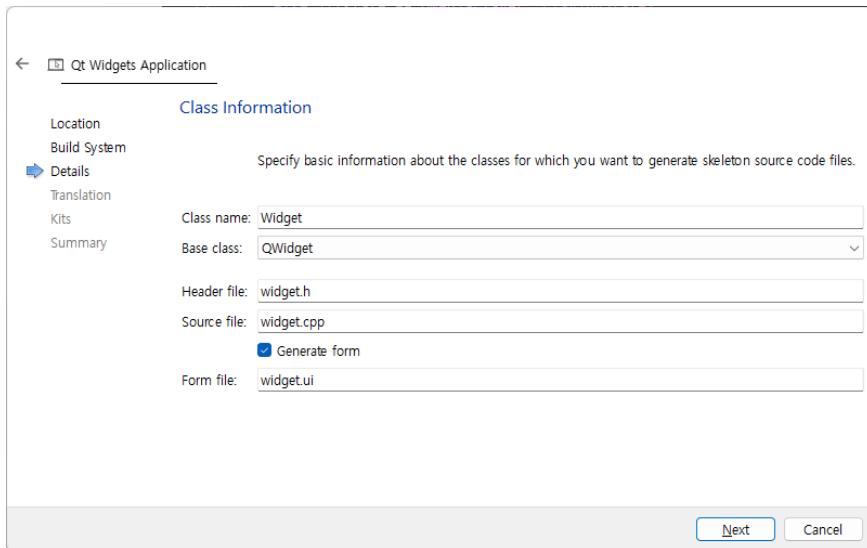
The Receiver outputs the message sent by the Sender on the GUI. Let's take a look at the source code of the Sender example first, then the Receiver example.

#### ✓ Sender example

Create a project based on Qt Widgets when creating a project.



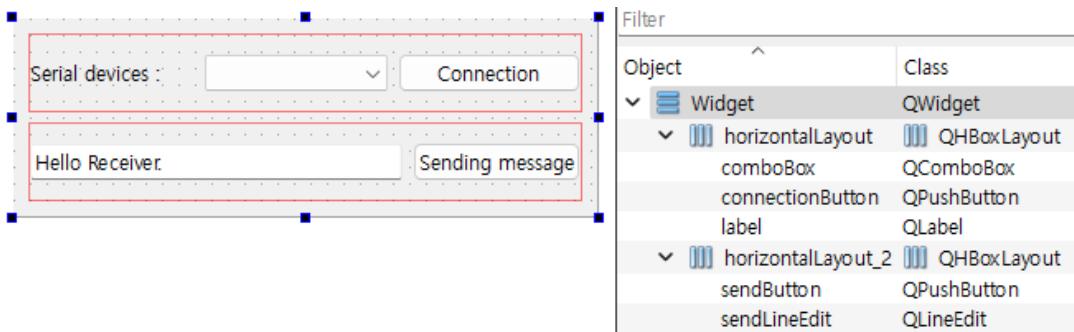
In this project, we will use UI Form, so check [Generation form] as shown below.



Once the project is created, add the SerialPort module to the project file as shown below.

```
cmake_minimum_required(VERSION 3.5)
...
find_package(Qt6 REQUIRED COMPONENTS Widgets SerialPort)
...
target_link_libraries(00_Sender PRIVATE
    Qt6::Widgets
    Qt6::SerialPort
)
...
```

Next, open the widget.ui file and place the GUI widget as shown below.



Next, open the widget.h header file and write something like this

```
#ifndef WIDGET_H
#define WIDGET_H
```

```
#include <QWidget>
#include <QSerialPort>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;

    struct SerialSettings {
        QString portName;
        qint32 baudRate;
        QSerialPort::DataBits dataBits;
        QSerialPort::Parity parity;
        QSerialPort::StopBits stopBits;
        QSerialPort::FlowControl flowControl;
    };

    SerialSettings m_serialSettings;
    QSerialPort *m_serial = nullptr;

private slots:
    void connectButton();
    void sendButton();
};
```

```
#endif // WIDGET_H
```

Next, open the widget.cpp source code file and write something like this

```
#include "widget.h"
#include "./ui_widget.h"
#include <QSerialPortInfo>

Widget::Widget(QWidget *parent)
    : QWidget(parent)
    , ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->connectionButton, SIGNAL(pressed()),
            this, SLOT(connectButton()));
    connect(ui->sendButton, SIGNAL(pressed()),
            this, SLOT(sendButton()));

    m_serialSettings.baudRate      = 115200;
    m_serialSettings.dataBits      = QSerialPort::Data8;
    m_serialSettings.parity        = QSerialPort::NoParity;
    m_serialSettings.stopBits      = QSerialPort::OneStop;
    m_serialSettings.flowControl   = QSerialPort::NoFlowControl;

    const auto infos = QSerialPortInfo::availablePorts();
    for (const QSerialPortInfo &info : infos)
        ui->comboBox->addItem(info.portName());

    m_serial = new QSerialPort(this);
}

void Widget::connectButton()
{
    if(m_serial->isOpen())
        return;

    QString devName = ui->comboBox->currentText().trimmed();
```

```
m_serialSettings.portName = devName;

m_serial->setPortName(m_serialSettings.portName);
m_serial->setBaudRate(m_serialSettings.baudRate);
m_serial->setDataBits(m_serialSettings.dataBits);
m_serial->setParity(m_serialSettings.parity);
m_serial->setStopBits(m_serialSettings.stopBits);
m_serial->setFlowControl(m_serialSettings.flowControl);

if (!m_serial->open(QIODevice::ReadWrite)) {
    qDebug() << "Error : " << m_serial->errorString();
}

void Widget::sendButton()
{
    if(!m_serial->isOpen())
        return;

    QString sendMsg = ui->sendLineEdit->text().trimmed();
    QByteArray msg = sendMsg.toLocal8Bit();

    m_serial->write(msg);
}

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

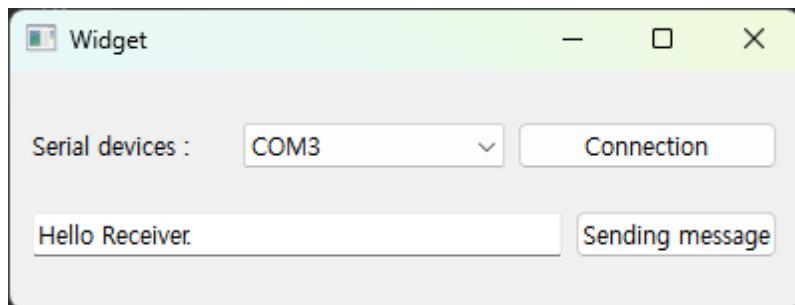
In the class constructor, the `QSerialPortInfo::availablePorts()` member function provides the ability to get information about all available serial devices. The available serial device information is obtained and registered in a combo box on the GUI.

When the `connectButton( )` Slot function is called, it sets the option value for serial connection saved in the class constructor and connects the serial device.

The `sendButton( )` Slot function converts the value of the character variable entered in

Jesus loves you.

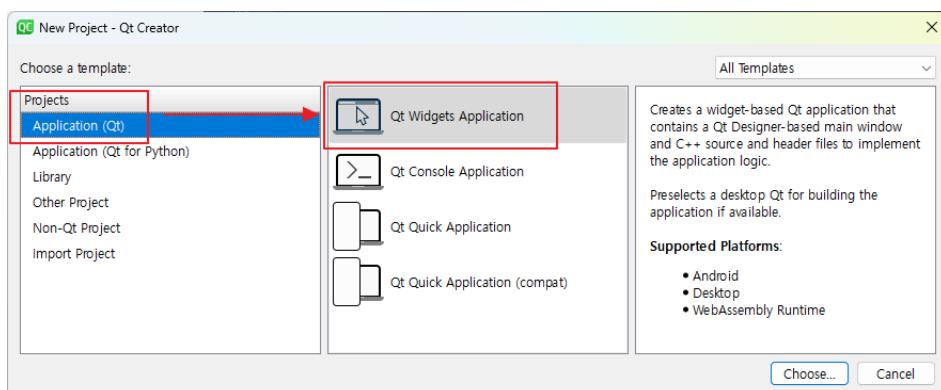
the QLineEdit to a QByteArray and sends a message using the write( ) member function of the QSerialPort class.



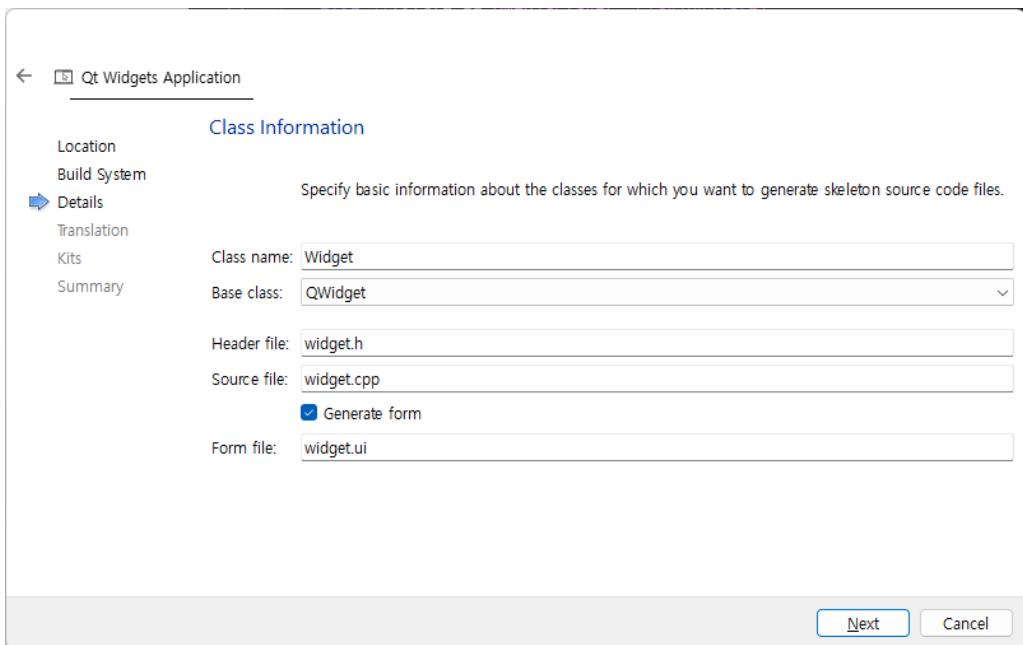
For the source code for this example, see the 00\_Sender example.

✓ Receiver example

Create a project based on Qt Widgets when creating a project.



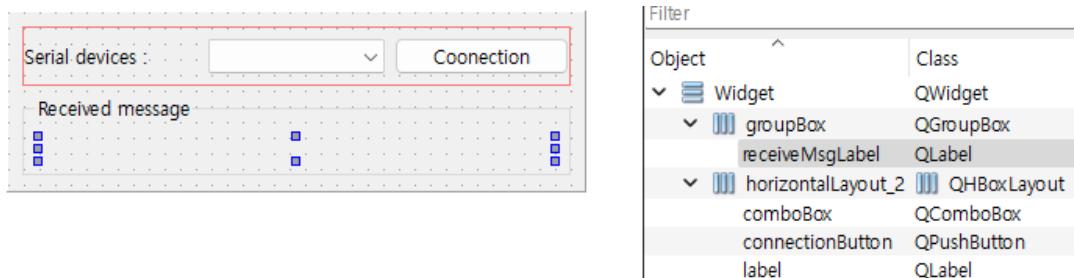
In this project, we will use UI Form, so check [Generation form] as shown below.



Once the project is created, add the SerialPort module to the project file as shown below.

```
cmake_minimum_required(VERSION 3.5)
...
find_package(Qt6 REQUIRED COMPONENTS Widgets SerialPort)
...
target_link_libraries(01_Receiver PRIVATE
    Qt6::Widgets
    Qt6::SerialPort
)
...
```

Next, open the `widget.ui` file and place the GUI widgets as shown below.



Next, open the `widget.h` header file and write something like this

```
#ifndef WIDGET_H
```

```
#define WIDGET_H

#include <QWidget>
#include <QSerialPort>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;

    struct SerialSettings {
        QString portName;
        qint32 baudRate;
        QSerialPort::DataBits dataBits;
        QSerialPort::Parity parity;
        QSerialPort::StopBits stopBits;
        QSerialPort::FlowControl flowControl;
    };

    SerialSettings m_serialSettings;
    QSerialPort *m_serial;

private slots:
    void connectButton();
    void readData();
    void errorOccurred(QSerialPort::SerialPortError err);
```

```
};

#endif // WIDGET_H
```

Next, create the widget.cpp source code file as shown below.

```
#include "widget.h"
#include "./ui_widget.h"
#include <QSerialPortInfo>

Widget::Widget(QWidget *parent) : QWidget(parent), ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->connectionButton, SIGNAL(pressed()),
            this, SLOT(connectButton()));

    m_serialSettings.baudRate      = 115200;
    m_serialSettings.dataBits      = QSerialPort::Data8;
    m_serialSettings.parity        = QSerialPort::NoParity;
    m_serialSettings.stopBits      = QSerialPort::OneStop;
    m_serialSettings.flowControl   = QSerialPort::NoFlowControl;

    const auto infos = QSerialPortInfo::availablePorts();
    for (const QSerialPortInfo &info : infos)
        ui->comboBox->addItem(info.portName());

    m_serial = new QSerialPort(this);
    connect(m_serial, SIGNAL(errorOccurred(QSerialPort::SerialPortError)),
            this,     SLOT(errorOccurred(QSerialPort::SerialPortError)));
    connect(m_serial, SIGNAL(readyRead()),
            this,     SLOT(readData()));
}

void Widget::connectButton()
{
    if(m_serial->isOpen())
        return;
```

```
QString devName = ui->comboBox->currentText().trimmed();
m_serialSettings.portName = devName;

m_serial->setPortName(m_serialSettings.portName);
m_serial->setBaudRate(m_serialSettings.baudRate);
m_serial->setDataBits(m_serialSettings.dataBits);
m_serial->setParity(m_serialSettings.parity);
m_serial->setStopBits(m_serialSettings.stopBits);
m_serial->setFlowControl(m_serialSettings.flowControl);

if (!m_serial->open(QIODevice::ReadWrite)) {
    qDebug() << "Error :" << m_serial->errorString();
}
}

void Widget::errorOccurred(QSerialPort::SerialPortError err)
{
    if (err == QSerialPort::ResourceError) {
        qDebug() << Q_FUNC_INFO
            << "Critical Error : "
            << m_serial->errorString();

        m_serial->close();
    }
}

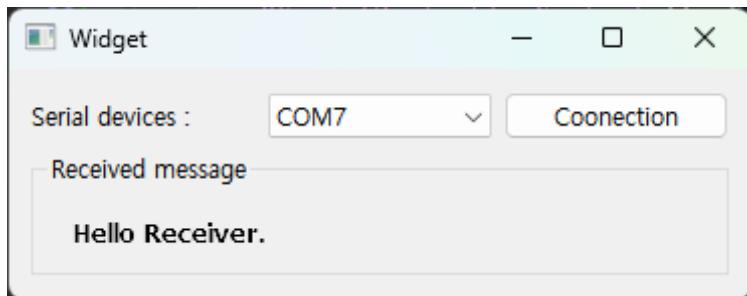
void Widget::readData()
{
    const QByteArray data = m_serial->readAll();
    ui->receiveMsgLabel->setText(data);
}

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

Jesus loves you.

Clicking the [Connection] button calls the `connectButton()` Slot function. This function opens the Serial device and receives the data sent by the Sender example.

When a message is received from the Sender, the `readyRead()` Signal is called. When this signal is called, the `readData()` Slot function is called. This Slot function stores the received data in a `QByteArray`. It then displays it on the GUI.



You can find the source code for this example in the 01\_Receiver directory.

## 40. Qt Positioning

The Qt Position module provides functions for easily calculating distance, azimuth, velocity, etc. using latitude and longitude. In addition, there are DD, DMS, and other ways of representing latitude and longitude. Converting these notations is also easy.

For example, you can use the `distanceTo( )` member function provided by the `QGeoCoordinate` class in conjunction with `Map` to calculate the distance between two points. You can also use the `azimuthTo( )` member function to calculate the azimuth.

The Qt Positioning module can therefore be useful when implementing applications that involve maps.

If you are using CMake for your project file, you will need to add the following to it

```
find_package(Qt6 REQUIRED COMPONENTS Positioning)
target_link_libraries(mytarget PRIVATE Qt6::Positioning)
```

Using the `QGeoCoordinate` class and `QGeoPositionInfo` class, you can display Date/Time, Latitude, and Longitude as shown below.

```
double latitude = -27.572321;
double longitude = 153.090718;

QString timeStr = QString("2009-08-24T22:24:37");
QDateTime timestamp = QDateTime::fromString(timeStr, Qt::ISODate);

QGeoCoordinate coordinate(latitude, longitude);
QGeoPositionInfo info(coordinate, timestamp);

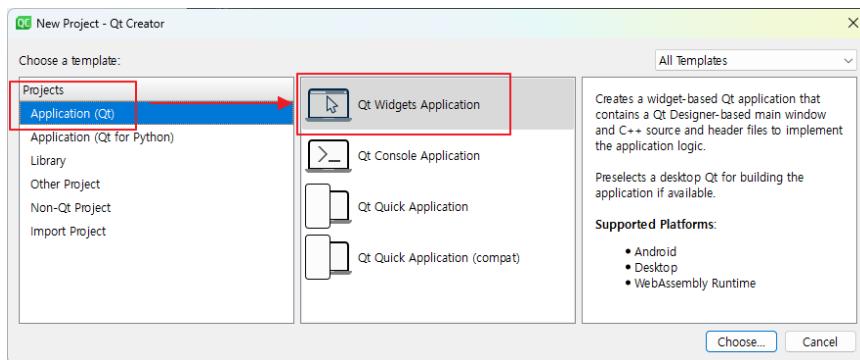
QString posDateTime;
posDateTime = QString("Position Date/time = %1")
               .arg(info.timestamp().toString());
QString posGeo;
posGeo = QString("Coordinate = %1")
               .arg(info.coordinate().toString());
```

```
qDebug() << posDateTime;  
qDebug() << posGeo;  
  
/*  
[ Result ]  
Position Date/time = Mon Aug 24 22:24:37 2009  
Coordinate = 27 ° 34' 20.4" S, 153 ° 5' 26.6" E  
*/
```

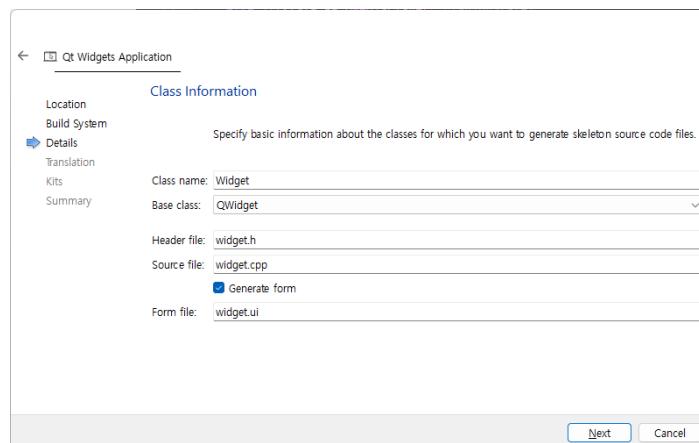
- Distance Example

In this example, we will implement an example to find the direct line between two cities.

Create a project based on Qt Widget.



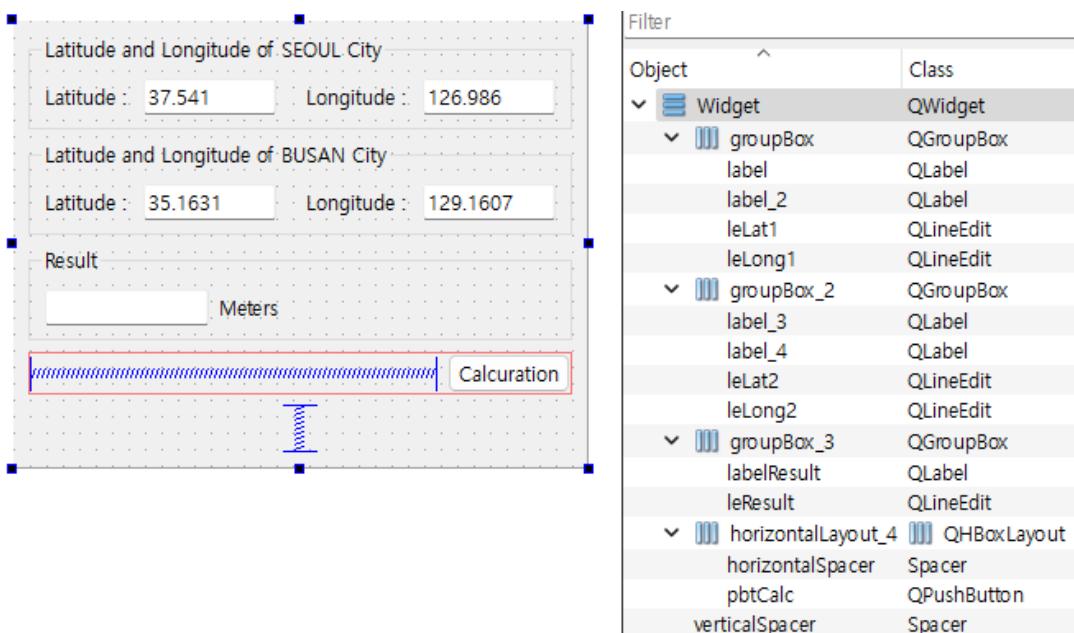
In this project, we will use UI Form, so check [Generation form] as shown below.



When you're done creating your project, add the Positioning module to your project file as shown below.

```
cmake_minimum_required(VERSION 3.5)
...
find_package(Qt6 REQUIRED COMPONENTS Widgets Positioning)
...
target_link_libraries(00_DistanceExample PRIVATE
    Qt6::Widgets
    Qt6::Positioning
)
...
```

Next, open the widget.ui file and place the GUI widgets as shown below.



Next, open the widget.h header file and write something like this

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE
```

```
class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;

private slots:
    void slotCalculation();
};

#endif // WIDGET_H
```

Next, open the widget.cpp source code file and write something like this

```
#include "widget.h"
#include "./ui_widget.h"
#include <QDateTime>
#include <QGeoCoordinate>
#include <QGeoPositionInfo>

Widget::Widget(QWidget *parent) : QWidget(parent), ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->pbtCalc, &QPushButton::clicked,
            this,           &Widget::slotCalculation);
}

void Widget::slotCalculation()
{
    double lat1      = ui->leLat1->text().toDouble();
    double long1     = ui->leLong1->text().toDouble();
    double lat2      = ui->leLat2->text().toDouble();
    double long2     = ui->leLong2->text().toDouble();
```

```
QGeoCoordinate coord1(lat1, long1);
QGeoCoordinate coord2(lat2, long2);

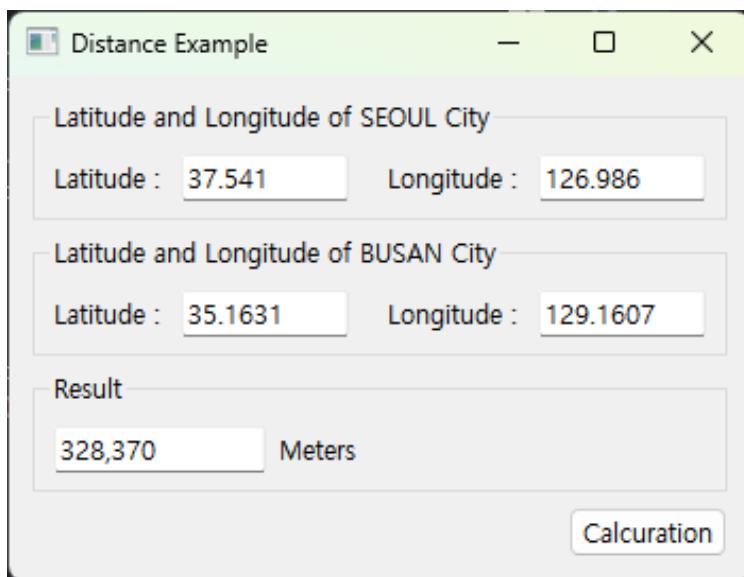
int distance_meter = coord1.distanceTo(coord2);
QString distanceStr = QString("%L1").arg(distance_meter);

ui->leResult->setText(distanceStr);
}

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

Clicking the [Calculation] button on the GUI calls the slotCalculation( ) Slot function. In this function, the distance between SEOUL City and BUSAN City is calculated in meters.

The distance between two cities can be calculated by using the distanceTo( ) member function provided by the QGeoCoordinate class. Here, the distance between the two cities is a straight line distance.



You can find the source code for this example in the 00\_DistanceExample directory.

## 41. Qt PDF

The PDF module provided by Qt provides functions for rendering PDFs on the GUI. In addition to simply displaying PDF documents, it provides various functions required for PDF rendering, such as ZOOM In/Out, Page back, Page Forward, Page Preview, etc. To use the Qt PDF module, you need to add the following to the project file of CMake.

```
find_package(Qt6 REQUIRED COMPONENTS Pdf)
target_link_libraries(mytarget Qt6::Pdf)
```

To use the widget class to display PDF documents on the GUI, you need to add it to your CMake project file as shown below.

```
find_package(Qt6 REQUIRED COMPONENTS PdfWidgets)
target_link_libraries(mytarget Qt6::PdfWidgets)
```

If you're using qmake, you'll need to add the following to your project file

```
QT += pdf
```

It is important to note that the Qt PDF module is not currently supported on MS Windows platforms. As of today, it is not supported in Qt 6.5.x, but it may be supported in future versions.

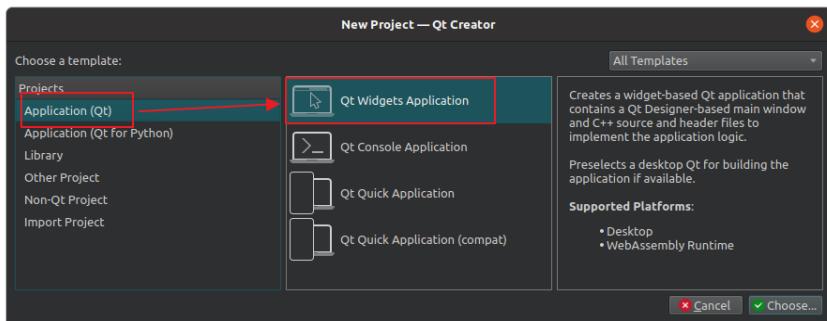
To use Qt PDF, it provides a very easy to use API. Therefore, we will cover how to use it with a simple example of rendering a PDF on the GUI.

- PDF Viewer Example

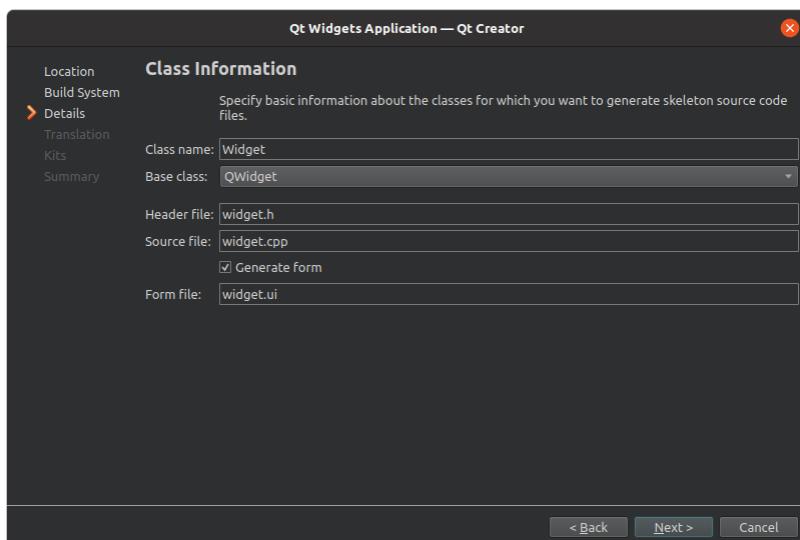
Since it is not supported on the MS Windows platform (Qt 6.5.x), we will cover the example on Linux.

In the project creation, we will write an application based on Qt Widget.

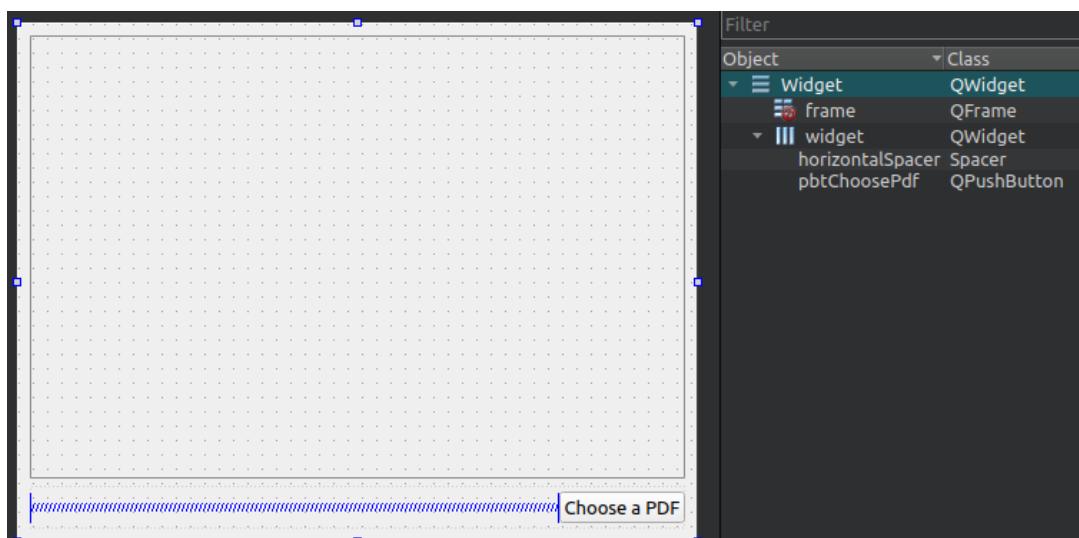
Jesus loves you.



In this project, we will use UI Form, so check [Generation form] as shown below.



Next, open the widget.ui file and place the GUI widgets as shown below.



Next, create the widget.h header file as shown below.

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QPdfView>
#include <QPdfDocument>
#include <QFileDialog>
#include <QPdfPageNavigator>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;
    QFileDialog *m_fileDialog = nullptr;

    QPdfView     *m_pdfView;
    QPdfDocument *m_document;

    void slot_pbtChoosePDF();
};

#endif // WIDGET_H
```

Next, create the widget.cpp source code file as shown below.

```
#include "widget.h"
#include "./ui_widget.h"
#include <QHBoxLayout>
```

```
#include <QStandardPaths>
#include <QDebug>

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

    m_pdfView = new QPdfView;
    m_document = new QPdfDocument(this);

    m_pdfView->setDocument(m_document);

    QBoxLayout *hLay = new QBoxLayout();
    hLay->addWidget(m_pdfView);

    ui->frame->setLayout(hLay);

    connect(ui->pbtChoosePdf, &QPushButton::clicked,
            this,           &Widget::slot_pbtChoosePDF);
}

void Widget::slot_pbtChoosePDF()
{
    qDebug() << Q_FUNC_INFO;

    if (m_fileDialog == nullptr) {
        m_fileDialog = new QFileDialog(this, tr("Choose a PDF"),
                                      QStandardPaths::writableLocation(
                                          QStandardPaths::DocumentsLocation));

        m_fileDialog->setAcceptMode(QFileDialog::AcceptOpen);
        m_fileDialog->setMimeTypeFilters({"application/pdf"});
    }
}
```

```
if (_fileDialog->exec() == QDialog::Accepted)
{
    const QUrl toOpen = _fileDialog->selectedUrls().constFirst();
    _document->load(toOpen.toLocalFile());

    const auto docTitle =
        _document->metaData(QPdfDocument::MetaDataField::Title)
            .toString();

    setWindowTitle(!docTitle.isEmpty() ? docTitle : "PDF Viewer");

    _pdfView->setPageMode(QPdfView::PageMode::MultiPage);
}
}

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

The constructor creates a QPdfView class object. This class provides the ability to render a PDF on the GUI.

And the QPdfDocument class provides the ability to load each page of a PDF document. Therefore, you need to associate an object of class QPdfDocument with QPdfView.

To connect with the QPdfView class, you can use the setDocument( ) member function provided by the QPdfView class.

Clicking the [Choose a PDF] button on the GUI calls the slot\_pbtChoosePDF( ) Slot function. When this function is called, the File Dialog is loaded. Select a PDF file from this dialogue, and the PDF document will be loaded into the QPdfView widget class.

Jesus loves you.



You can find the source code for this example in the `00_PDFViewer` directory.

## 42. Qt Printer Support

The Print Support module provided by Qt can print to a connected printer or to a remote printer over a network.

Qt Print Support modules can also be generated as PDF files. (Note: The Qt Print Support module is not currently supported on iOS.) To use the Qt Printer Support module, if you are using CMake, you need to add the module to your project file as shown below.

```
find_package(Qt6 REQUIRED COMPONENTS PrintSupport)
target_link_libraries(mytarget PRIVATE Qt6::PrintSupport)
```

If you're using qmake, you can add the following to your project file

```
QT += printsupport
```

Among the classes provided in the Qt Print Support module, the QPrinter class can print Qt GUI widgets or QPainter regions.

```
QPrinter printer;
...
QPainter painter;
painter.begin(&printer);

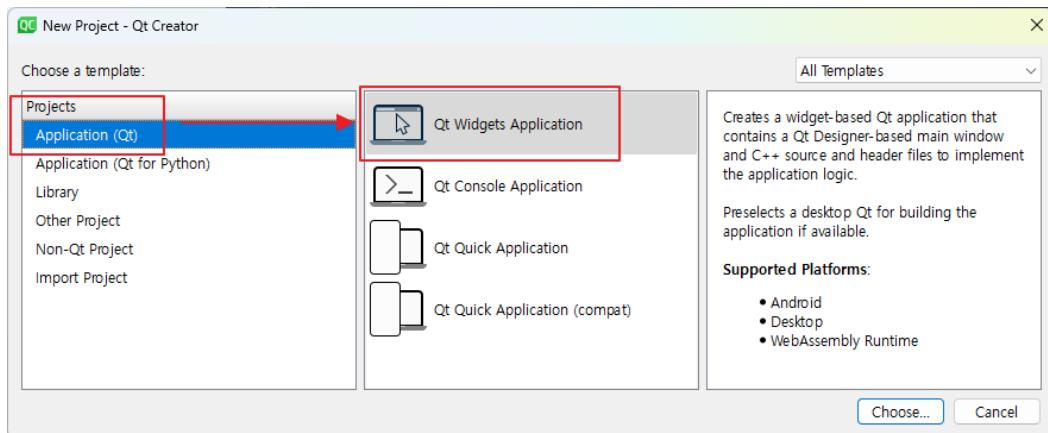
...
myWidget->render(&painter);
...
```

QTextEdit 하는 GUI Widget 상에 표시된 텍스트를 프린트 하기 위해서 아래와 같이 사용할 수 있다.

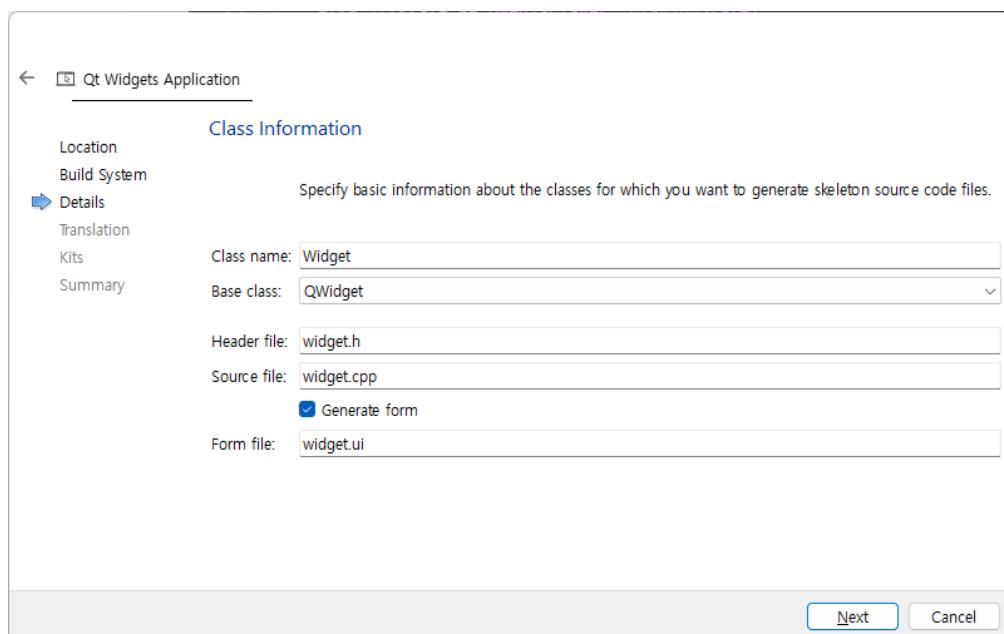
```
QTextEdit *textEdit = new QTextEdit();
...
textEdit.setText("Hello world.");
QPrinter printer(QPrinter::HighResolution);
textEdit->print(&printer);
```

- Print example

In this simple example, we will implement the ability to print text from the GUI widget QTextEdit to a printer connected to the computer. When creating a project, create a project based on Qt Widgets.



In this project, we will use UI Form, so check [Generation form] as shown below.



Once the project creation is complete, add the Printer Support module to the CMakeLists.txt file as shown below.

```
cmake_minimum_required(VERSION 3.5)
```

```
project(00_Printer_Example VERSION 0.1 LANGUAGES CXX)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTOMOC ON)
set(CMAKE_AUTORCC ON)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Qt6 REQUIRED COMPONENTS Widgets PrintSupport)

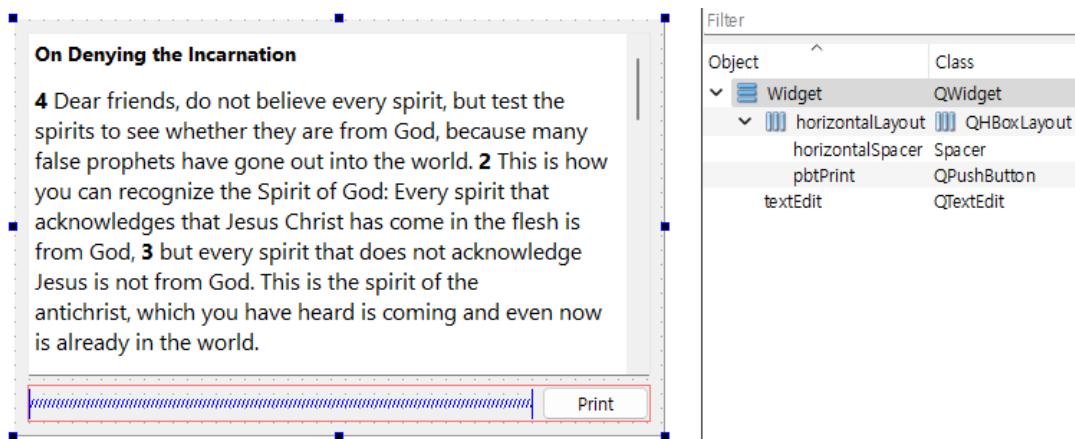
set(PROJECT_SOURCES
    main.cpp
    widget.cpp
    widget.h
    widget.ui
)

qt_add_executable(00_Printer_Example
    ${PROJECT_SOURCES}
)

target_link_libraries(00_Printer_Example PRIVATE
    Qt6::Widgets
    Qt6::PrintSupport
)

install(TARGETS 00_Printer_Example
    BUNDLE DESTINATION .
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

Next, open the `widget.ui` file and place the GUI widgets as shown below.



You can put anything in the QTextEdit widget. In this case, we'll be printing, so we've entered the above text, but you can enter any text you want.

Place the GUI widgets as shown above and create the `widget.h` header file as shown below.

```
#ifndef WIDGET_H
#define WIDGET_H
#include <QWidget>
#include <QPrinter>

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT
public:
    Widget(QWidget *parent = nullptr);
    ~Widget();
private:
    Ui::Widget *ui;
    void slot_pbtPrint();
};

#endif // WIDGET_H
```

Next, open your `widget.cpp` source code file and write the following code

```
#include "widget.h"
#include "./ui_widget.h"
#include <QDebug>
#include <QFileDialog>
#include <QPrintDialog>

Widget::Widget(QWidget *parent)
    : QWidget(parent), ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->pbtPrint, &QPushButton::clicked,
            this,           &Widget::slot_pbtPrint);
}

void Widget::slot_pbtPrint()
{
    QPrinter printer(QPrinter::HighResolution);

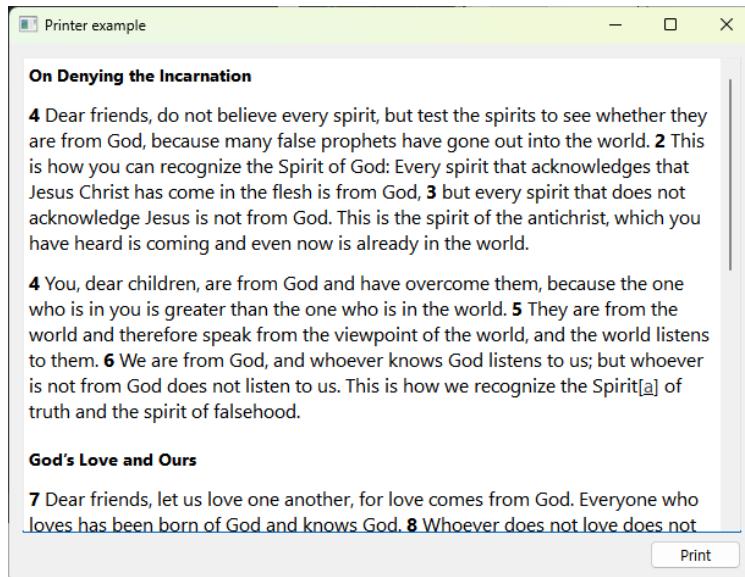
    QPrintDialog dialog(&printer, this);
    dialog.setWindowTitle(tr("Print Document"));

    if (dialog.exec() == QDialog::Accepted) {
        ui->textEdit->print(&printer);
    }
}

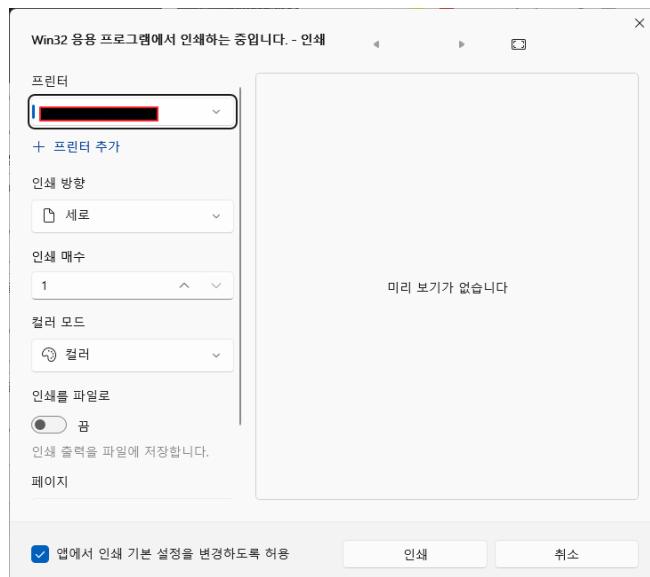
Widget::~Widget()
{
    delete ui;
}
```

Once you've written the above, run it after building and print out the contents of QTextEdit.

Jesus loves you.



To print, click the [Print] button on the GUI. When you click the [Print] button, the Print dialogue is loaded as shown below.



For the above example source code, refer to the 00\_Printer\_Example directory.

## 43. Qt Pprotobuf

Protobuf, provided by Qt, is a module that provides easy access to Protocol Buffers (or Protobuf), provided by Google.

Protocol Buffers makes it easy to serialise structured data. For example, suppose you have the following structure for sending and receiving data between two programs.

```
struct EmployeeInfo {  
    int32 number;  
    string name;  
    string department;  
}
```

The sending program converts (or serialises) the above structure data into a QByteArray and sends it over the network.

Then, suppose the receiving program receives the QByteArray data and inserts the data into the structure above. In this case, Protocol Buffer automatically creates a class to facilitate the serialisation and deserialisation process. Thus, it provides the ability to easily handle data stored in QByteArray.

Protocol Buffers is supported by many programming languages besides C++. Therefore, you can create the same proto file in various programming languages and easily use it even if the language is different.

For more information, you can refer to the official Protocol Buffers website for more information. (<https://protobuf.dev>)

- Protocol Buffers Install

Protocol Buffers provided by Qt can be installed on MS Windows, Linux and MacOS. To install Protocol Buffers on MS Windows, you can download it from GitHub below.

```
https://github.com/protocolbuffers/protobuf/releases
```

Jesus loves you.

Protocol Buffers v25.1

## Announcements

- [Protobuf News](#) may include additional announcements or pre-announcements for upcoming changes.

Python

Let's download and install Protocol Buffers on MS Windows. As you can see in the picture above, at the bottom, you will be downloading the MS Windows 64 bit version.

File	Size	Published
protoc-25.1-linux-x86_32.zip	3.22 MB	2 weeks ago
protoc-25.1-linux-x86_64.zip	2.96 MB	2 weeks ago
protoc-25.1-osx-aarch_64.zip	2.11 MB	2 weeks ago
protoc-25.1-osx-universal_binary.zip	4.23 MB	2 weeks ago
protoc-25.1-osx-x86_64.zip	2.13 MB	2 weeks ago
protoc-25.1-win32.zip	3.03 MB	2 weeks ago
protoc-25.1-win64.zip	3 MB	2 weeks ago
Source code (zip)		2 weeks ago
Source code (tar.gz)		2 weeks ago

https://github.com/protocolbuffers/protobuf/releases/download/v25.1/protoc-25.1-osx-x86\_64.zip

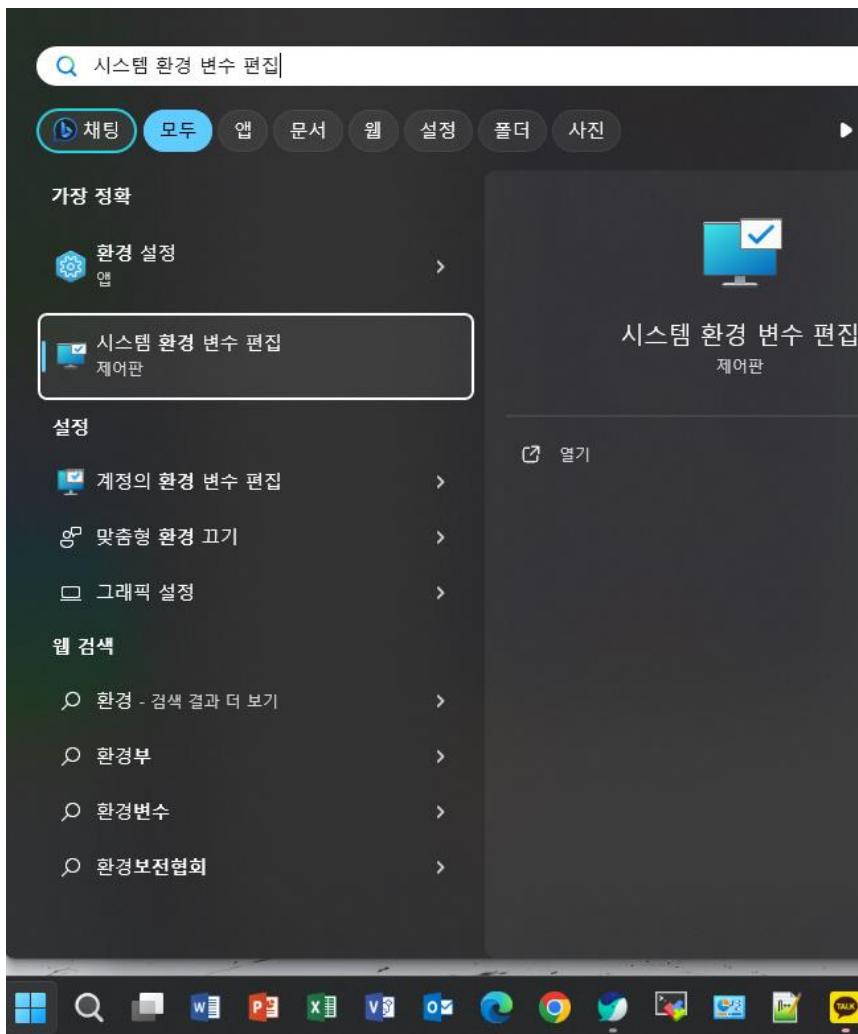
Download the above file, which is not an executable binary and does not require installation.

After unzipping the downloaded file, you can move the unzipped directory to any

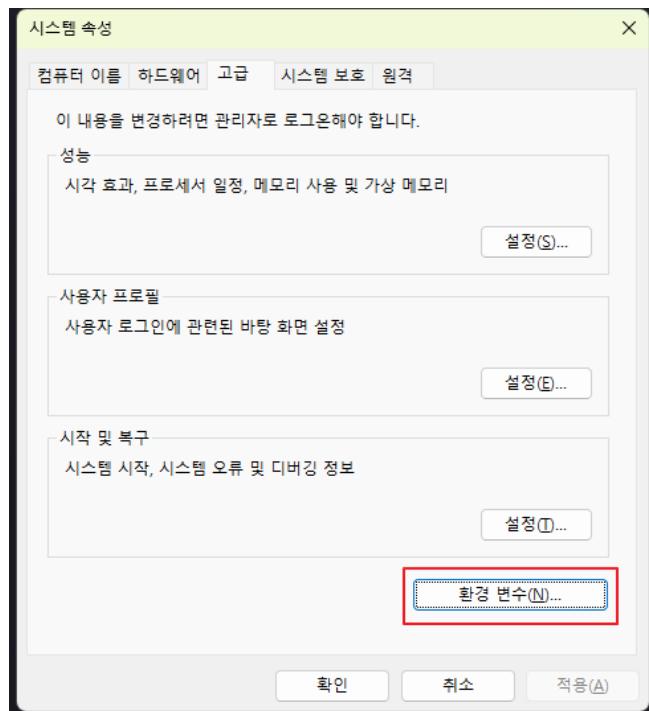
location of your choice. In my case, I moved it to D:\.

```
D:\protoc-25.1-win64
```

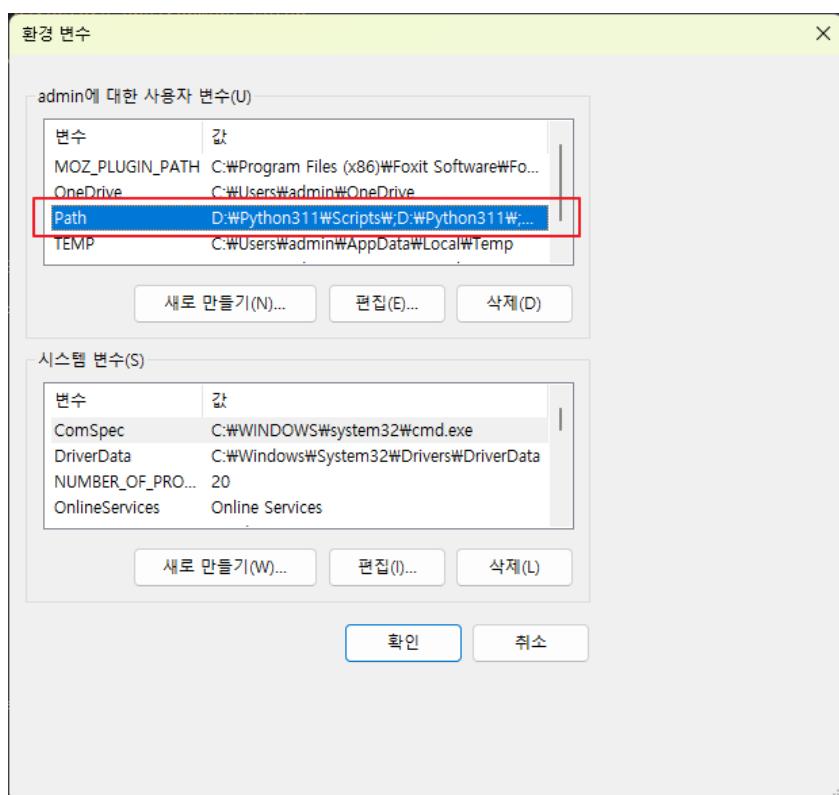
Next, register the above Protocol Buffers directory in a system environment variable so that Qt Creator can find it.

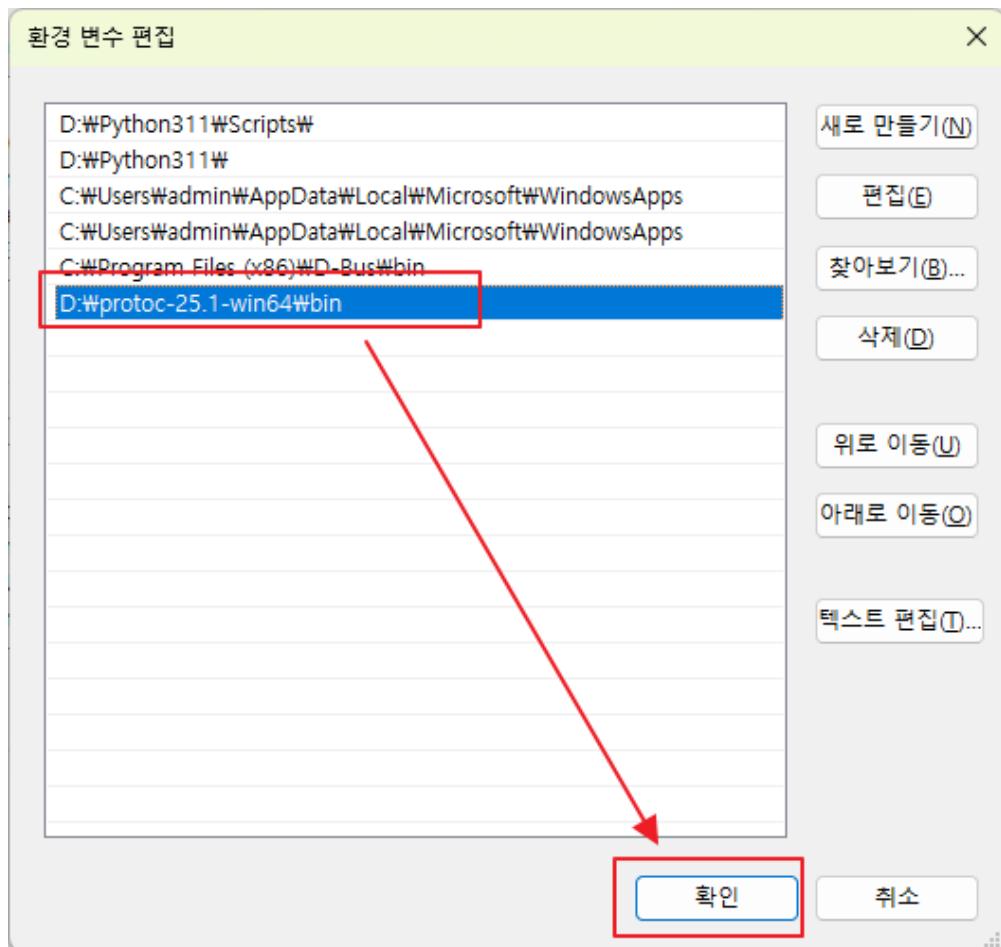


Click [Edit System Environment Variables] as shown in the image above.



Click [Environment Variables], as shown in the image above.





Add it to the environment variable as shown in the image above, and then click the [OK] button. Next, let's check if the environment variable has been added successfully as shown below.

A screenshot of a Windows Command Prompt window titled "명령 프롬프트". The window shows the following text:  
Microsoft Windows [Version 10.0.22621.2715]  
(c) Microsoft Corporation. All rights reserved.  
C:\Users\admin>protoc --version  
libprotoc 25.1  
C:\Users\admin>

Jesus loves you.

Linux and MacOS can be registered in the same way as above. Once you've done that, you're ready to use the Protobuf module provided by Qt. To use the Qt Protobuf module, you need to add the following to your project files

```
find_package(Qt6 REQUIRED COMPONENTS Protobuf)
target_link_libraries(mytarget PRIVATE Qt6::Protobuf)
```

And the .proto file used by the Qt Protobuf module should look like this

```
syntax = "proto3";

package my.employee;

message EmployeeInfo {
    int32 num = 1;
    string name = 2;
    string department = 3;
}
```

The first line is the version information. The second "my.employee" is a namespace in C++. For example, it is used as the following namespace in C++.

```
namespace my::employee {
class EmployeeInfo;
}
```

And message is the same concept as structure in C++. For example, you can think of three variables defined in a structure called EmployeeInfo.

If you're up to this point, you can reference the .proto file to automatically create the EmployeeInfo class. To automatically create this class, you can add the following to your project file

```
qt_add_protobuf(protobuf_employee
    PROTO_FILES
        ../protobuf_files/employee.proto
)
```

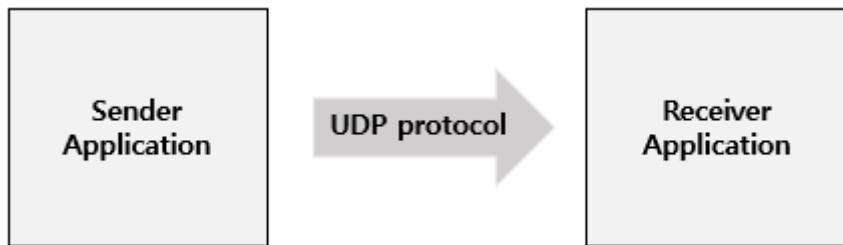
You can do this by registering the protobuf\_employee above with the target\_link\_libraries( ) entry as shown below.

```
target_link_libraries(00_Sender PRIVATE
    Qt6::Core
    Qt6::Widgets
    Qt6::Protobuf
    Qt6::Network
    protobuf_employee
)
```

So far, we've seen how to use the Qt Protobuf module. Let's see how to use it with an example.

- Examples of sending and receiving data using Qt Protobuf

In this example, we will implement two examples. The Sender example uses the Qt Protobuf module to send a serialised QByteArray over the UDP protocol. The Receiver example uses the Qt Protobuf module to deserialise the data received using the UDP protocol.



Firstly, we need to use the same structure for both examples, so create a directory called "protobuf\_files" as shown below and create a file called employee.proto inside this directory.

Next, write the employee.proto file like this

```
syntax = "proto3";

package my.employee;

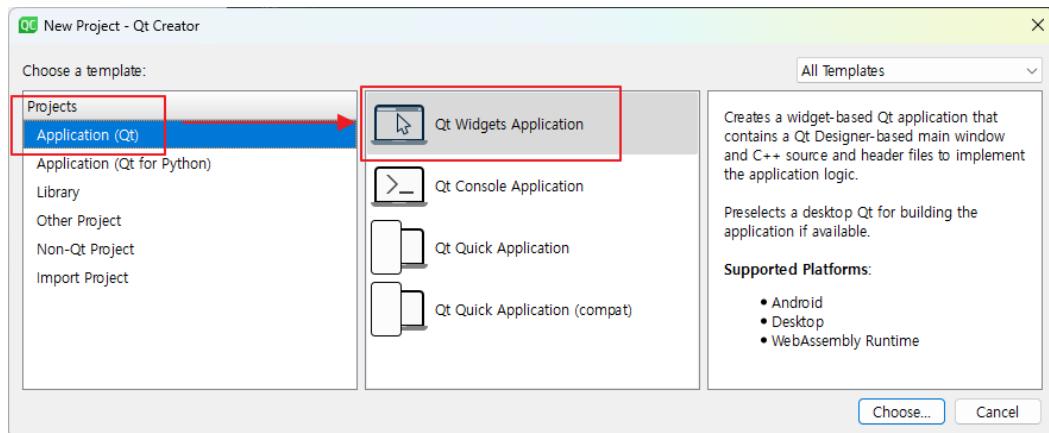
message EmployeeInfo {
    int32 num = 1;
    string name = 2;
```

```
string department = 3;  
}
```

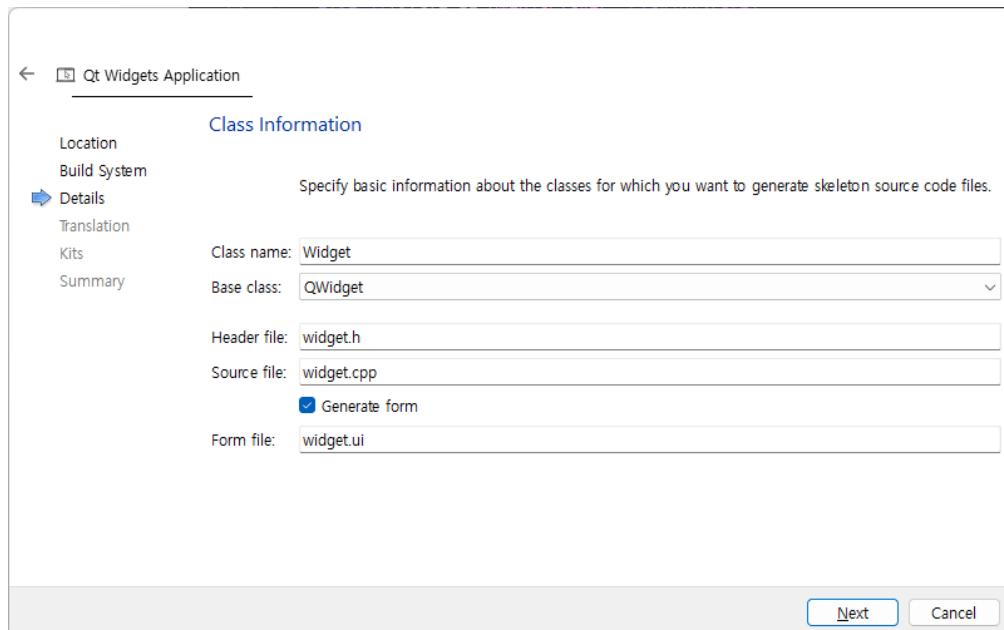
Now that you have the above, let's create a Sender example.

✓ Sender example

Create a project based on Qt Widgets when creating a project.



In this project, we will use UI Form, so check [Generation form] as shown below.



Once the project is created, write the following in the CMakeLists.txt file

```
cmake_minimum_required(VERSION 3.16)
```

```
project(00_Sender VERSION 0.1 LANGUAGES CXX)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTORCC ON)
set(CMAKE_AUTOMOC ON)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Qt6 REQUIRED COMPONENTS
    Core
    Widgets
    Protobuf
    Network
)

qt_standard_project_setup()

set(PROJECT_SOURCES
    main.cpp
    widget.cpp
    widget.h
    widget.ui
)

qt_add_executable(00_Sender
    ${PROJECT_SOURCES}
)

qt_add_protobuf(protobuf_employee
    PROTO_FILES
        ../protobuf_files/employee.proto
)

target_link_libraries(00_Sender PRIVATE
```

```

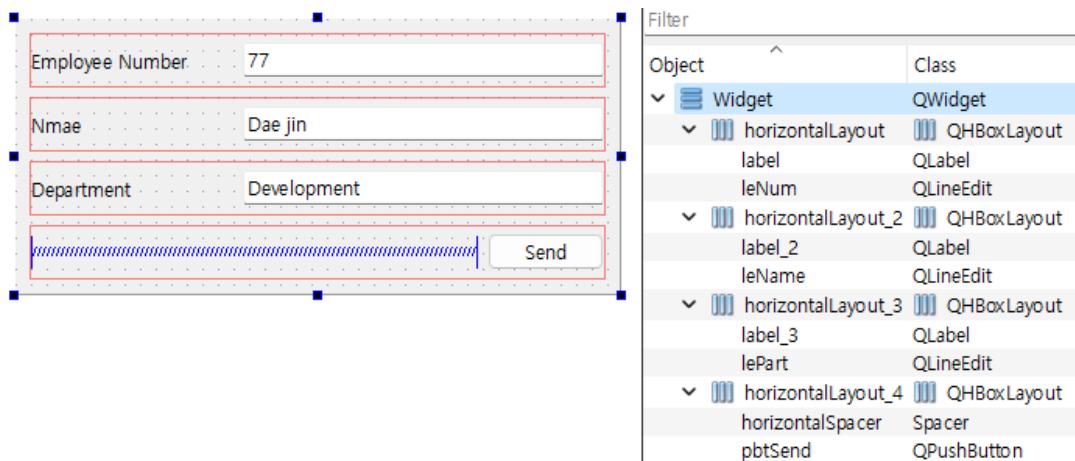
Qt6::Core
Qt6::Widgets
Qt6::Protobuf
Qt6::Network
protobuf_employee

)

install(TARGETS 00_Sender
    BUNDLE DESTINATION .
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)

```

Write it as above. If you get an error that the file is not found for the MOC, it may be caused by the CMake version in the `cmake_minimum_required()` function, as shown above. Therefore, modify the version to 3.16 as shown above. Next, open the `widget.ui` file and place the GUI widget as shown below.



Next, open the `widget.h` header file and write the following code.

```

#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QProtobufSerializer>
#include <QUdpSocket>

```

```
namespace my::employee {
class EmployeeInfo;
}

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
    ~Widget();

private:
    Ui::Widget *ui;

    QUdpSocket m_socket;
    QProtobufSerializer m_serializer;

    void send(const QByteArray &data);

private slots:
    void slot_pbtSend();
};

#endif // WIDGET_H
```

다음으로 widget.cpp 소스코드 파일을 아래와 같이 작성한다.

```
#include "widget.h"
#include "ui_widget.h"
#include <QDebug>

#include "employee.qpb.h"
```

```
Widget::Widget(QWidget *parent)
    : QWidget(parent)
    , ui(new Ui::Widget)
{
    ui->setupUi(this);
    connect(ui->pbtSend, &QPushButton::clicked, this, &Widget::slot_pbtSend);
}

void Widget::send(const QByteArray &data)
{
    if (m_socket.writeDatagram(data, QHostAddress::LocalHost, 61122) == -1) {
        qWarning() << "Unable to send data of size: "
                    << data.size()
                    << "to UDP port 61122";
    }
}

void Widget::slot_pbtSend()
{
    my::employee::EmployeeInfo eInfo;

    QtProtobuf::int32 tNum      = ui->leNum->text().toInt();
    QString          tName     = ui->leName->text();
    QString          tDepart   = ui->lePart->text();

    eInfo.setNum(tNum);
    eInfo.setName(tName);
    eInfo.setDepartment(tDepart);

    QByteArray sendData = eInfo.serialize(&m_serializer);
    send(sendData);
}

Widget::~Widget()
{
```

```
    delete ui;  
}
```

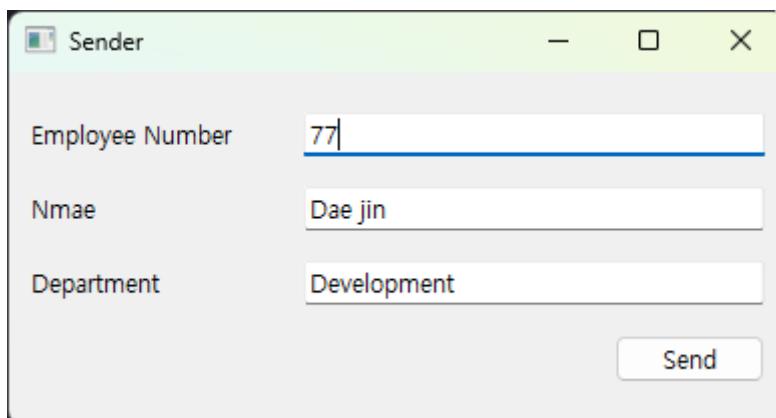
Click the [Send] button on the GUI to call the slot\_pbtSend( ) Slot function. It reads the value from the GUI and stores the value in int32 and QString.

We store the value using the member function provided by the EmployeeInfo class, which is automatically created by the Qt Protobuf module.

And to store the data stored in the EmployeeInfo class object in a QByteArray, we use the serialise( ) member function provided by this class.

This member function provides the ability to store the data stored in the EmployeeInfo class object as a QByteArray.

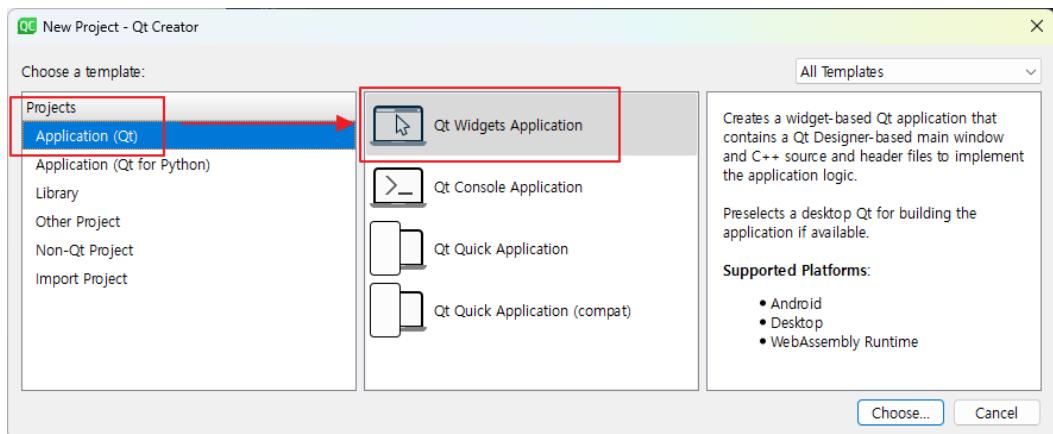
The QByteArray data is then sent using the UDP protocol.



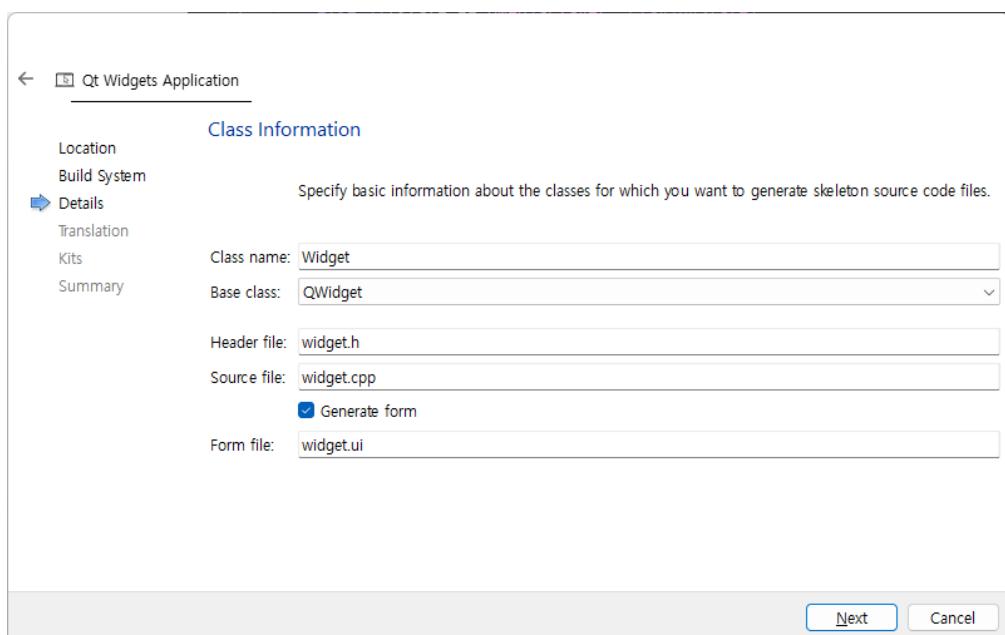
You can find the source code for this example in the 00\_Sender directory.

#### ✓ Receiver example

The Receiver example is an example of displaying data received via UDP Protocol in a GUI. When creating a project, create a project based on Qt Widget.



In this project, we will use UI Form, so check [Generation form] as shown below.



Once the project is created, write the following in the CMakeLists.txt file

```
cmake_minimum_required(VERSION 3.16)
project(01_Receiver VERSION 0.1 LANGUAGES CXX)

set(CMAKE_AUTOUIC ON)
set(CMAKE_AUTORCC ON)
set(CMAKE_AUTOMOC ON)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

```
find_package(Qt6 REQUIRED COMPONENTS
    Core
    Widgets
    Protobuf
    Network
)

qt_standard_project_setup()

set(PROJECT_SOURCES
    main.cpp
    widget.cpp
    widget.h
    widget.ui
)

qt_add_executable(01_Receiver
    ${PROJECT_SOURCES}
)

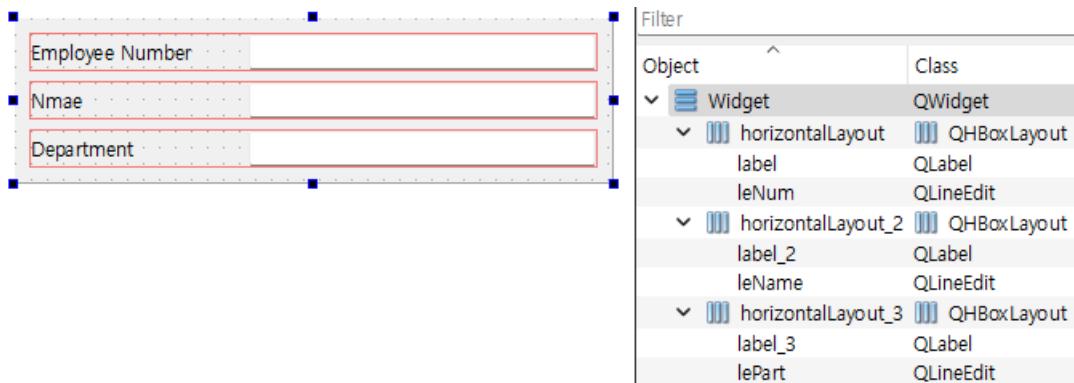
qt_add_protobuf(protobuf_employee
    PROTO_FILES
        ./protobuf_files/employee.proto
)

target_link_libraries(01_Receiver PRIVATE
    Qt6::Core
    Qt6::Widgets
    Qt6::Protobuf
    Qt6::Network
    protobuf_employee
)

install(TARGETS 01_Receiver
    BUNDLE DESTINATION .
```

```
LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)
```

Next, open the widget.ui file and place the GUI widgets as shown below.



Next, open the widget.h header file and write the following code

```
#ifndef WIDGET_H
#define WIDGET_H

#include <QWidget>
#include <QProtobufSerializer>
#include <QUdpSocket>

namespace my::employee {
class EmployeeInfo;
}

QT_BEGIN_NAMESPACE
namespace Ui { class Widget; }
QT_END_NAMESPACE

class Widget : public QWidget
{
    Q_OBJECT

public:
    Widget(QWidget *parent = nullptr);
```

```
~Widget();

void receive();

private:
Ui::Widget *ui;

Q_udpSocket m_client;
QProtobufSerializer m_serializer;

void display(const QtProtobuf::int32,
             const QString &,
             const QString &);

};

#endif // WIDGET_H
```

Next, open your widget.cpp source code file and write the following code

```
#include "widget.h"
#include "ui_widget.h"

#include <QNetworkDatagram>
#include <QDebug>

#include "employee.qpb.h"

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

    Q_ASSERT_X(m_client.bind(QHostAddress::LocalHost, 61122),
               "Receiver",
               "Unable to bind to port 61122");
```

```
QObject::connect(&m_client, &QUdpSocket::readyRead,
                 this,      &Widget::receive);
}

void Widget::receive()
{
    while (m_client.hasPendingDatagrams())
    {
        const auto datagram = m_client.receiveDatagram();

        my::employee::EmployeeInfo eInfo;
        eInfo.deserialize(&m_serializer, datagram.data());

        if(m_serializer.serializationError()
           == QAbstractProtobufSerializer::NoError)
        {
            QtProtobuf::int32 tNum      = eInfo.num();
            QString          tName     = eInfo.name();
            QString          tDepart   = eInfo.department();

            display(tNum, tName, tDepart);
        }
    }
}

void Widget::display(const QtProtobuf::int32 vNum,
                     const QString &vName,
                     const QString &vDepartment)
{
    QString strNum      = QString("%1").arg(vNum);
    QString strName     = QString("%1").arg(vName);
    QString strDepartment = QString("%1").arg(vDepartment);

    ui->leNum->setText(strNum);
    ui->leName->setText(strName);
    ui->lePart->setText(strDepartment);
```

```
}
```

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

When data is received using the UDP protocol, the receive( ) Slot function is called. In this function, the received QByteArray data is stored in the EmployeeInfo class object using the deserialise( ) member function of the automatically generated class using the Qt Protobuf module. We then read the value using the member function provided by the EmployeeInfo class and display the value on the GUI.



You can find the source code for this example in the 01\_Receiver directory.