Multi-Directory Projects in Qt with CMake

Jeffrey K. Bienstadt

10 February, 2004

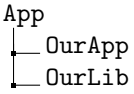
Abstract

The Qt framework and the Qt Creator IDE support projects that are split across multiple directories. Unfortunately, the IDE does not provide mechanisms for managing such projects. This article will address this and demonstrate how to create and manage multi-directory projects with Qt.

Write something introductory here.

This article assumes some previous exposure to Qt, both the framework itself and the Qt Creator integrated development environment (IDE). We will be using version 6.6.1 of the Qt framework, and Qt Creator version 12.0.2.

We will create a multi-directory project with a main application and a helpful library with a directory structure like this:



The main application will be in the directory `App/OurApp`, and the supporting library will be in the directory `App/OurLib`.

We will start by creating a directory named `App` to contain our application. Create this directory wherever you like.

Next we will use Qt Creator to generate our main application project and our library project, both within the new `App` directory. We will start with the main application. Using Qt Creator, create a new project, either from the **File** menu (**New Project...**) or by selecting **Create Project...** from the **Welcome** page. In the **New Project** dialog, select **Application (Qt)** in the left-most pane, and **Qt Widgets Application** in the center pane, as shown in Figure 1, then press the **Choose...** button.

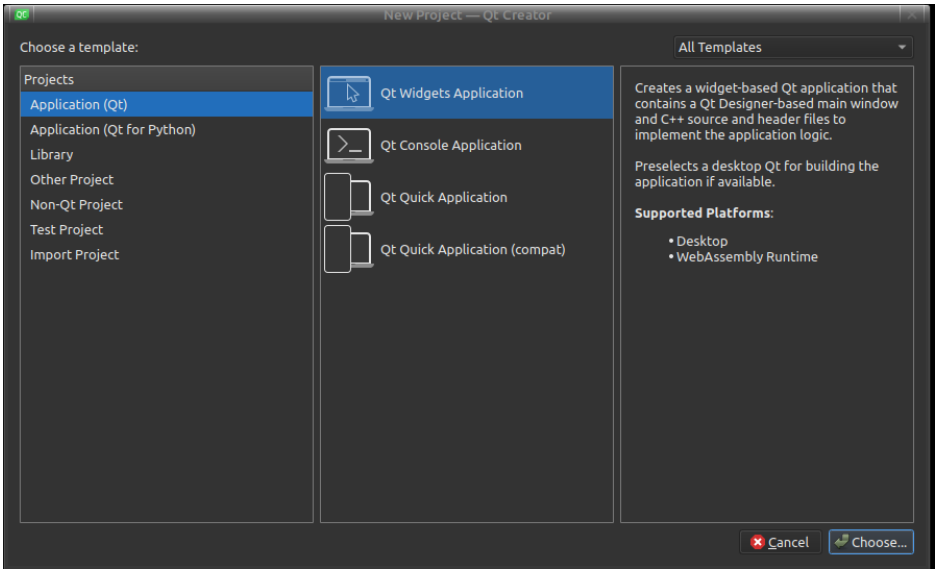


Figure 1: Create the OurApp application.

On the **Project Location** dialog, name the project **OurApp**, and create the project in the newly-created **App** directory, as shown in Figure 2, then click the **Next** button.

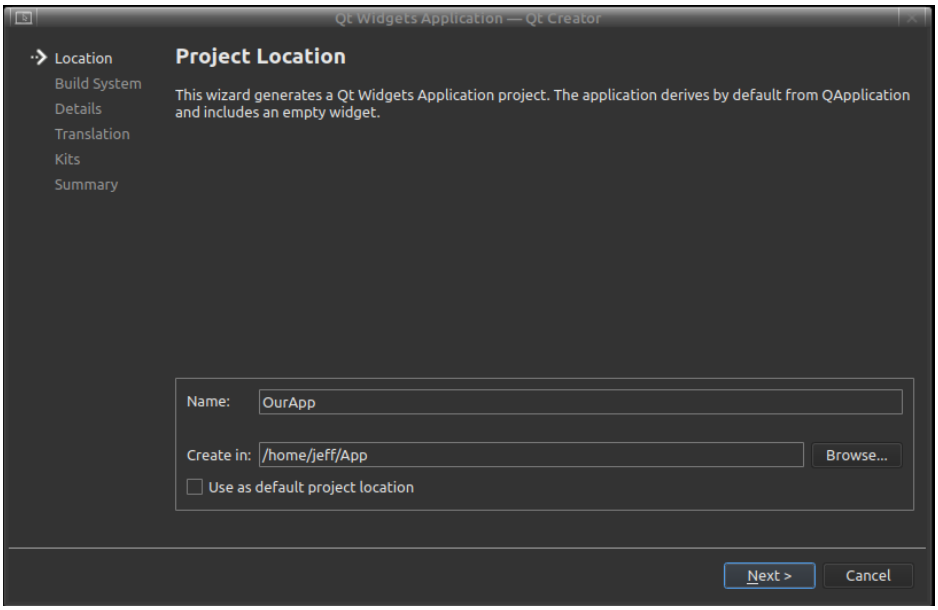


Figure 2: Name and location of the OurApp application.

On the **Define Build System** dialog, shown in Figure 3, select **CMake** as the build system, and click **Next**.

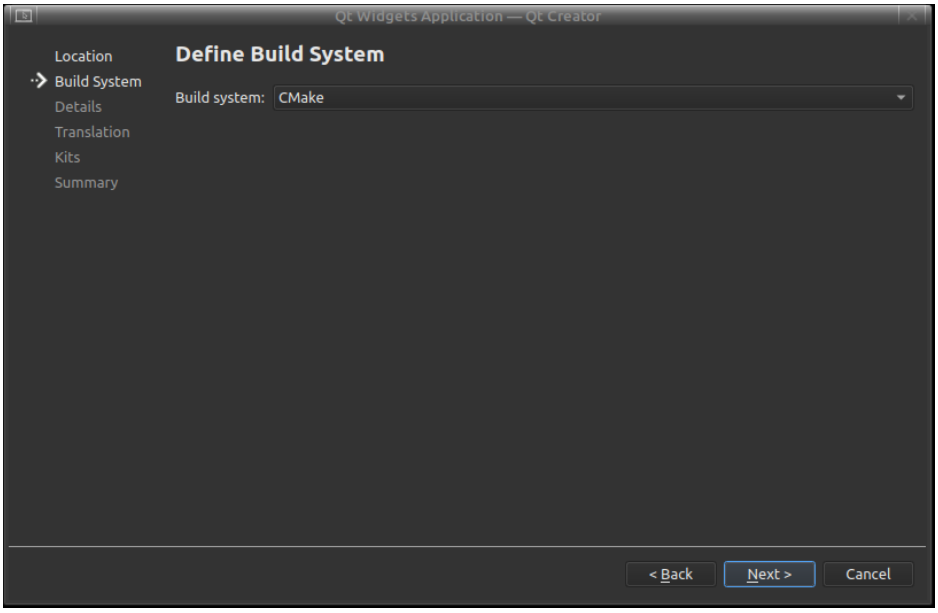


Figure 3: Select CMake as the build system.

On the **Class Information** dialog you can just leave the default values for everything and click the **Next** button (Figure 4).

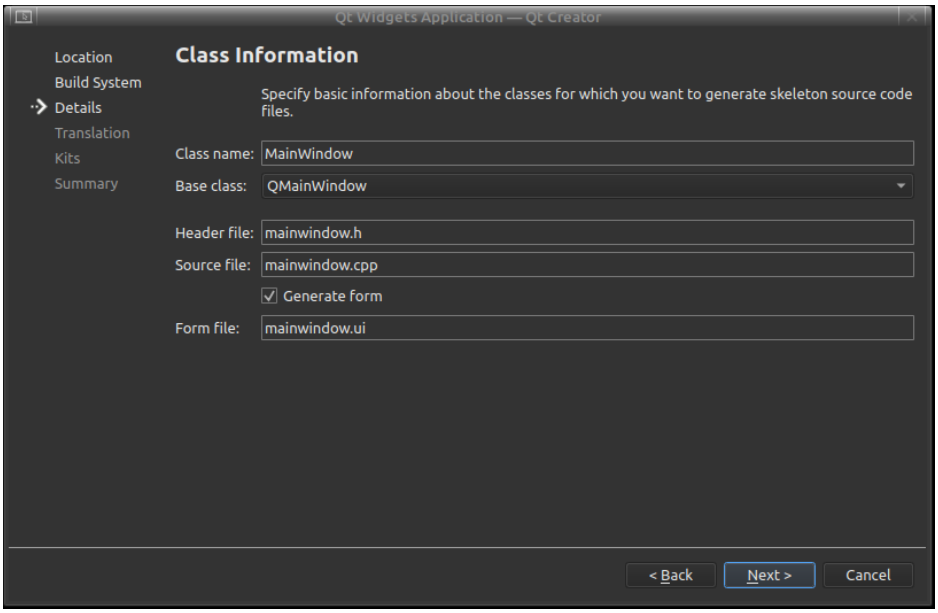


Figure 4: Class Information dialog.

Click **Next** on the **Translation File** dialog (Figure 5), leaving the default values.

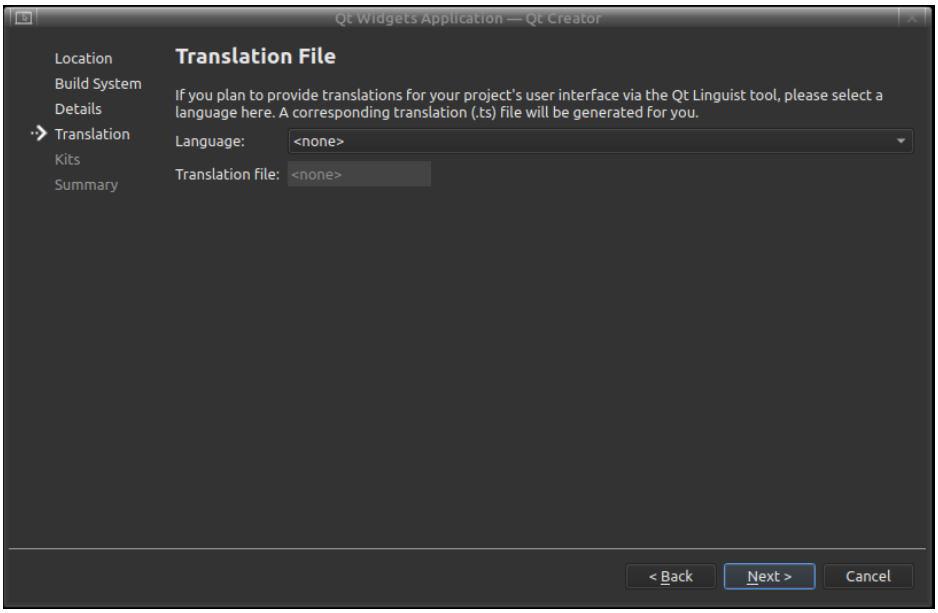


Figure 5: Translation File dialog.

On the **Kit Selection** dialog, select a **Desktop** kit as shown in Figure 6 and click the **Next** button.

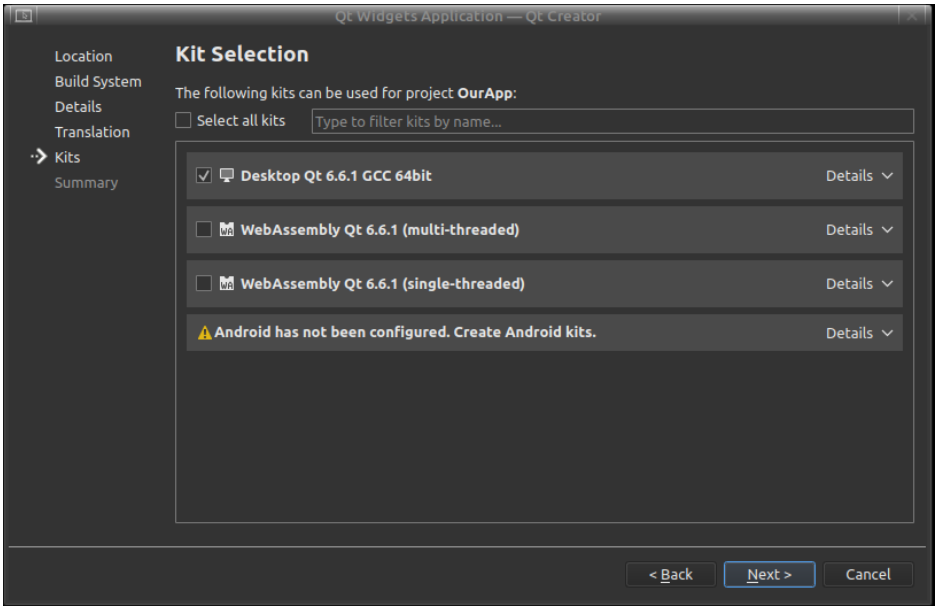


Figure 6: Select a Desktop kit.

Finally, on the **Project Management** dialog (Figure 7), leave the default values and click the **Finish** button. Qt Creator will generate a GUI project in the `App/OurApp` directory.

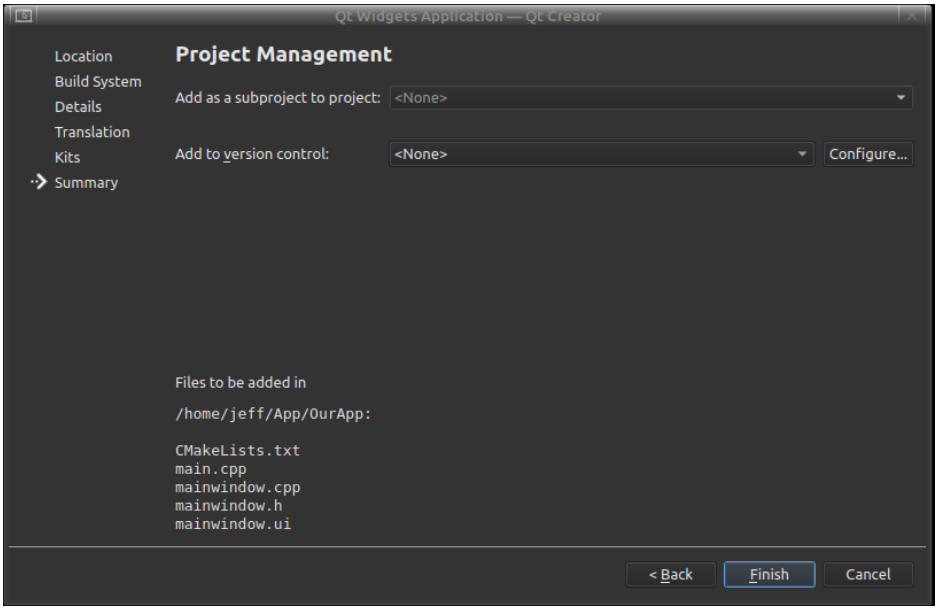


Figure 7: Project Management dialog.

This project contains a `CMakeLists.txt` file with the following content:

```
cmake_minimum_required(VERSION 3.5)

project(OurApp 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
        mainwindow.cpp
        mainwindow.h
        mainwindow.ui
)

if(${QT_VERSION_MAJOR} GREATER_EQUAL 6)
    qt_add_executable(OurApp
        MANUAL_FINALIZATION
        ${PROJECT_SOURCES}
    )
# Define target properties for Android with Qt 6 as:
#    set_property(TARGET OurApp APPEND PROPERTY QT_ANDROID_PACKAGE_SOURCE_DIR
#                 ${CMAKE_CURRENT_SOURCE_DIR}/android)
# For more information, see https://doc.qt.io/qt-6/qt-add-executable.html#target-creation
else()
    if(ANDROID)
        add_library(OurApp SHARED
            ${PROJECT_SOURCES}
        )
# Define properties for Android with Qt 5 after find_package() calls as:
#    set(ANDROID_PACKAGE_SOURCE_DIR "${CMAKE_CURRENT_SOURCE_DIR}/android")
    else()
        add_executable(OurApp
            ${PROJECT_SOURCES}
        )
    endif()
endif()

target_link_libraries(OurApp PRIVATE Qt${QT_VERSION_MAJOR}::Widgets)

# Qt for iOS sets MACOSX_BUNDLE_GUI_IDENTIFIER automatically since Qt 6.1.
# If you are developing for iOS or macOS you should consider setting an
# explicit, fixed bundle identifier manually though.
if(${QT_VERSION} VERSION_LESS 6.1.0)
  set(BUNDLE_ID_OPTION MACOSX_BUNDLE_GUI_IDENTIFIER com.example.OurApp)
endif()
set_target_properties(OurApp PROPERTIES
    ${BUNDLE_ID_OPTION}
    MACOSX_BUNDLE_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 OurApp
    BUNDLE DESTINATION .
    LIBRARY DESTINATION ${CMAKE_INSTALL_LIBDIR}
    RUNTIME DESTINATION ${CMAKE_INSTALL_BINDIR}
)

if(QT_VERSION_MAJOR EQUAL 6)
    qt_finalize_executable(OurApp)
endif()
```

Now that we have our main application, we can augment it with a custom library. Create a new project, this time select **Library** in the left-most pane and **C++ Library** in the center pane, as shown in Figure 8, and click the **Choose...** button.

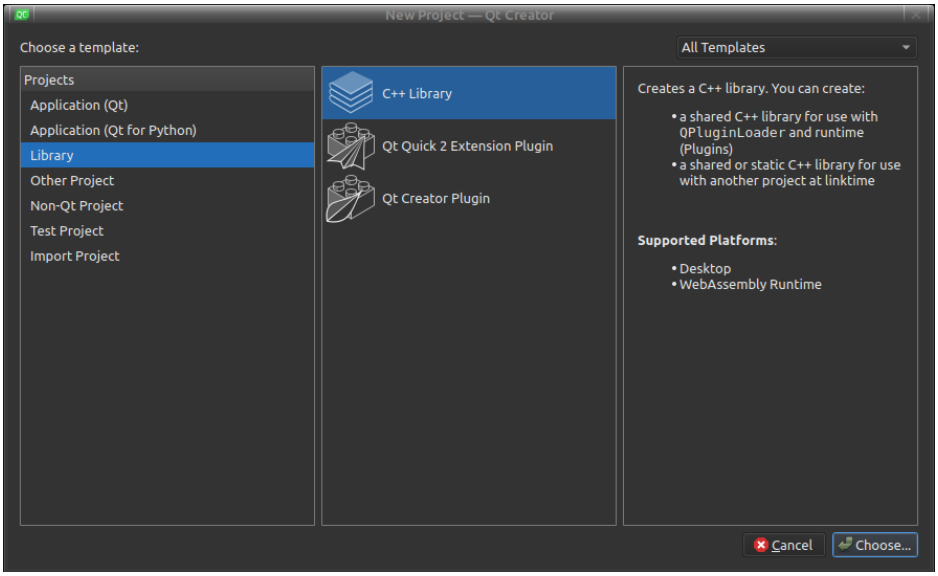


Figure 8: Create the C++ library dialog.

On the Project Location dialog (Figure 9), name the project **OurLib** and create the project within the **App** directory along side the **OurApp** project directory.

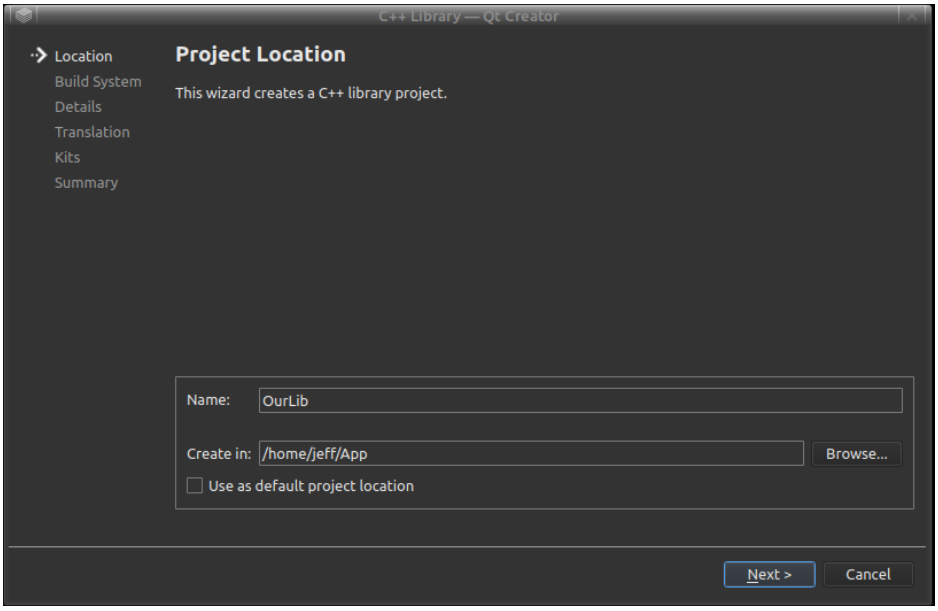


Figure 9: Provide the name and location of the library.

As before, in the **Define Build System** dialog choose **CMake** as the build system and click **Next**.

On the **Define Project Details** dialog, select **Statically Linked Library** as the type, and select **Widgets** as the Qt module (Figure 10). Leave the rest to their default values and click **Next**.

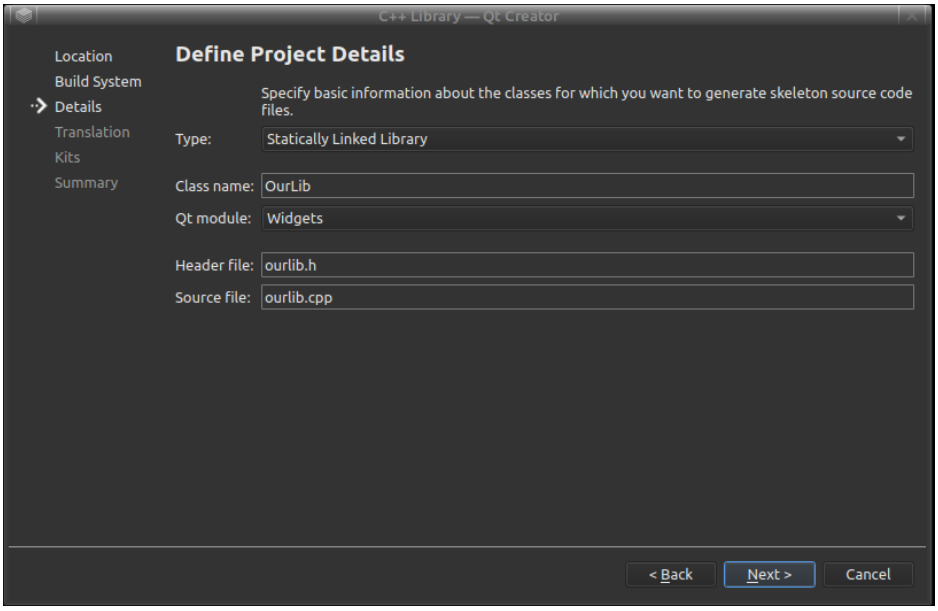


Figure 10: Provide the library type and the required Qt module.

As we did before, click **Next** to leave the **Translation File** values to their defaults, and select a **Desktop** kit on the **Kit Selection** dialog and click **Next**. Again, leave the **Project Management** dialog to itself default values and click **Finish** to generate the library project.

Among the files in the library project is a `CMakeLists.txt` file. This one is a bit simpler than the one for the `OurApp` project:

```
cmake_minimum_required(VERSION 3.14)

project(OurLib 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)

add_library(OurLib STATIC
  ourlib.cpp
  ourlib.h
)

target_link_libraries(OurLib PRIVATE Qt${QT_VERSION_MAJOR}::Widgets)

target_compile_definitions(OurLib PRIVATE OURLIB_LIBRARY)
```

We now have two separate projects, an application and a library, as shown in Figure 11.

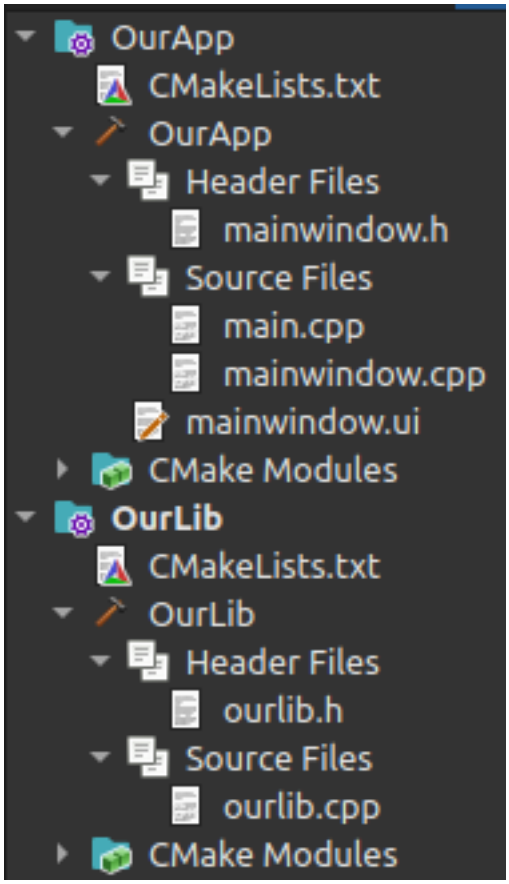


Figure 11: Layout for two projects.

However, they are just that, two separate projects that have no real connection. We can build each project, but they have nothing to do with each other. We need to make sure that Qt Creator knows that the application OurApp requires the static library OurLib.

Before we go any further, let's modify the library's code so that it does something useful for the application. The library header file `outlib.h` contains a simple class definition:

```
#ifndef OURLIB_H
#define OURLIB_H

class OurLib
{
public:
    OurLib();
};

#endif // OURLIB_H
```

The library souce file `ourlib.cpp` contains just an empty definition of the class' constructor:

```
#include "ourlib.h"

OurLib::OurLib() {}
```

Let's get rid of the class and provide a simple free function that greets the user. Our new `ourlib.h` looks like

```
#ifndef OURLIB_H
#define OURLIB_H

#include <QWidget>

void sayHello(QWidget *parent);

#endif // OURLIB_H
```

while our new `ourlib.cpp` file defines the `sayHello` function:

```
  #include "ourlib.h"

  #include <QMessageBox>

void sayHello(QWidget *parent)
{
    QMessageBox::information(parent, "Hello", "Hello from OurLib!");
}
```

The Qt-generated `main.cpp` file in the OurApp application is very simple:

```
#include "mainwindow.h"

#include <QApplication>

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

We will modify this file to make use of our library by adding two lines of code:

```
#include "mainwindow.h"
#include "../OurLib/ourlib.h"

#include <QApplication>

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

Here we added the `#include` directive to bring in our library's header file from the `OurLib` project directory, and added a call to the `sayHello` function provided by the library. When we build our projects, the `OurLib` project compiles and links, producing a library file. However, the `OurApp` project compiles but fails to link, with an error

saying that there is an undefined reference to the `sayHello` function. This is because the `OurApp` project knows nothing about the `OurLib` library and isn't linking to it.

It's time to change our project model a bit. We currently have two separate projects, each with its own "top-level" `CMakeLists.txt` file. We are going to change this so that there is a single top-level `CMakeLists.txt` file, and slightly modified `CMakeLists.txt` files in each of the project directories.

To make it easier to edit the `CMakeLists.txt` files, we'll close both of our projects. From the **File** menu, select **Close All Projects and Editors**.

Edit the `CMakeLists.txt` file in the `OurLib` directory. In Qt Creator's **File** menu, select **Open File With...**, select the `CMakeLists.txt` file, then select **CMake Editor** in the dialog shown in Figure 12, then click OK.

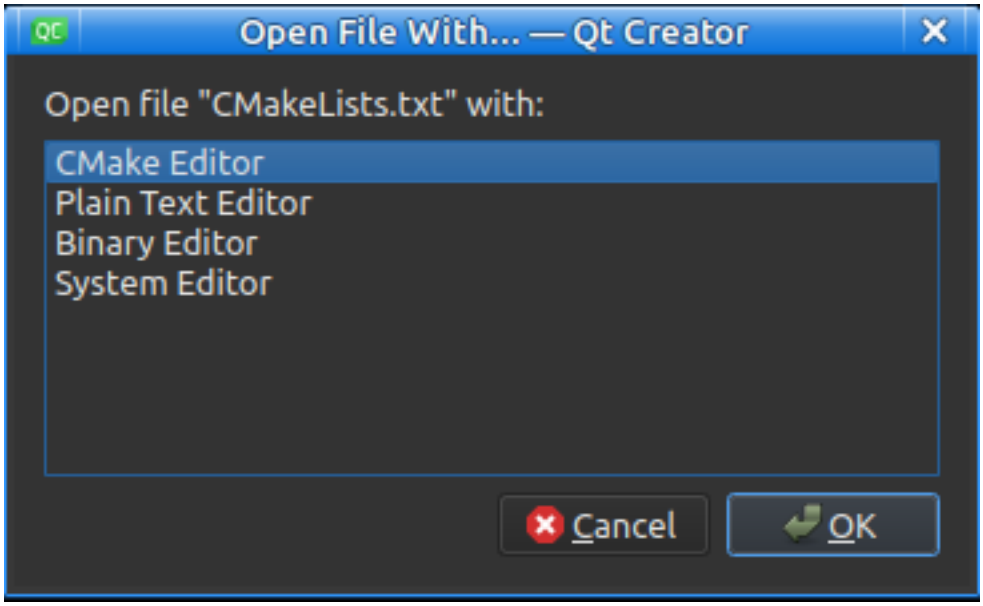


Figure 12: Open with CMake editor.

Delete the first twelve lines, leaving

```
add_library(OurLib STATIC
  ourlib.cpp
  ourlib.h
)

target_link_libraries(OurLib PRIVATE Qt${QT_VERSION_MAJOR}::Widgets)

target_compile_definitions(OurLib PRIVATE OURLIB_LIBRARY)
```

Edit the `CMakeLists.txt` file in the `OurApp` directory and delete the first fourteen lines, so the file begins with

```
set(PROJECT_SOURCES
        main.cpp
        mainwindow.cpp
        mainwindow.h
        mainwindow.ui
)
```

We new have `CMakeLists.txt` files for the `OurApp` and `OurLib` projects that contain only the information requred for each separate project. What we need now is a controlling top-level `CMakeLists.txt` file that contains the information required for both projects.

Create a new `CMakeLists.txt` file. From the Qt Creator **File** menu, select **New File...**. In the **New File** dialog, shown in Figure 13, select **General** in the left-most pane, and **Empty File** in the center pane, then click the **Choose...** button.

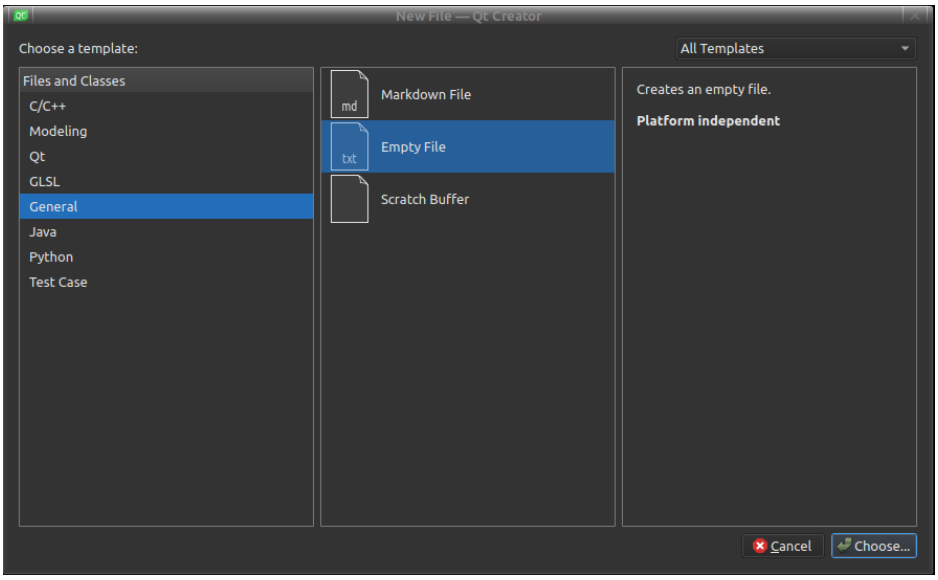


Figure 13: Create a new file.

In the **Location** dialog (see Figure 14), name the file `CMakeLists.txt` and place it in the `App` directory we created earlier, one directory level up from the `OurApp` and `OurLib` directories. Click **Next**.

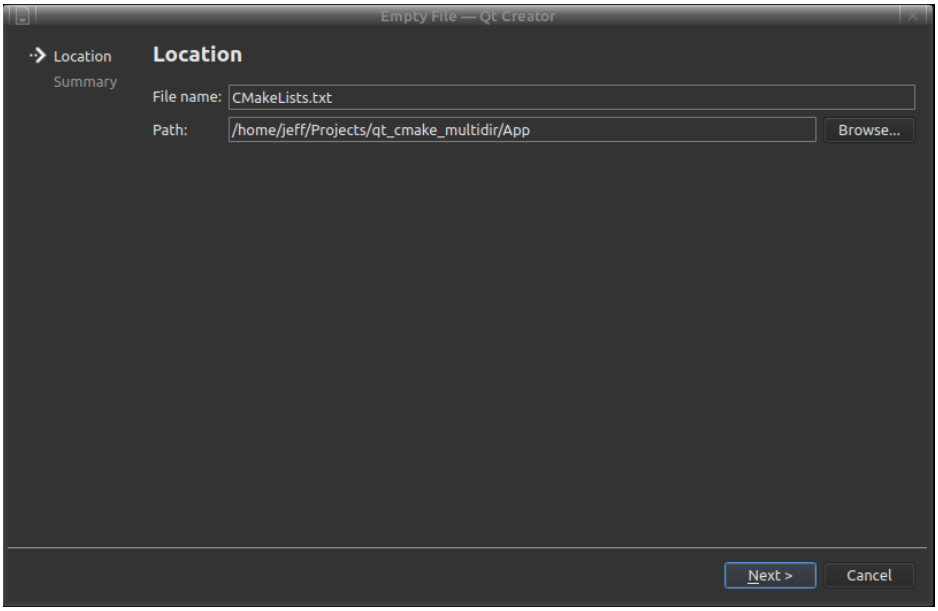


Figure 14: Name and location of new file.

In the **Project Management** dialog, leave everything to the default values and click **Finish**.

Add the following lines to the new `CMakeLists.txt` file:

```
cmake_minimum_required(VERSION 3.14)

project(App 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)

add_subdirectory(OurLib)
add_subdirectory(OurApp)
```

The `cmake_minimum_required` command specifies the minimum version of CMake required to process the project files. The `project` command specifies the name of the project (App) as well as the project version number and the programming languages used.

The `set(CMAKE_AUTOUIC ON)` command specifies that `CMake` will handle the Qt user interface compiler automatically. `set(CMAKE_AUTOMOC ON)` specifies that `CMake` will handle the Qt `moc` preprocessor automatically, while the `set(CMAKE_AUTORCC)` command tells `CMake` to handle the Qt resource compiler automatically.

The `set(CMAKE_CXX_STANDARD 17)` and `set(CMAKE_CXX_STANDARD_REQUIRED)` commands specify that the C++ compiler should use C++17, and that this is a requirement.

The two `find_package` commands locate the necessary Qt libraries that will be required by both the `OurLib` and `OurApp` sub-projects.

Finally, we need to tell `CMake` that the `OurLib` and `OurApp` projects are sub-projects of the root project with the `add_subdirectory(OurLib)` and `add_subdirectory(OurApp)` commands. It is important to issue these commands in this order, with `OurLib` first, because `OurApp` depends on `OurLib` and will require some information about the library.

One final task remains: we must tell the `OurApp` project that it needs to link with the library produced by the `OurLib` project. Modify the `CMakeLists.txt` file in the `OurApp` directory such that the `target_link_libraries` command includes the `OurLib` library project. The updated command should now look like:

```
target_link_libraries(OurApp PRIVATE Qt${QT_VERSION_MAJOR}::Widgets OurLib)
```

Save and close all three `CMakeLists.txt` files. Then from the Qt Creator **File** menu select **Open File or Project...** then select the new top-level `CMakeLists.txt` file in the `App` directory. Select a **Desktop** kit when prompted, then click the **Configure Project** button. You now have a main project with two sub-projects. If you expand the project tree it should look like Figure 15.

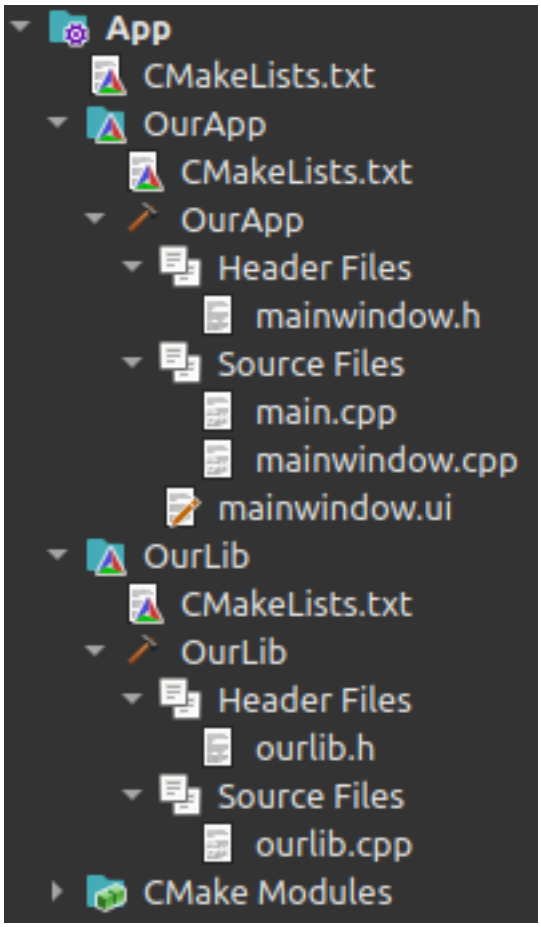


Figure 15: Layout for a main project with two sub-projects.

Once you have a working top-level `CMakeLists.txt` file, you may add as many executable or library subdirectory

projects as you need. Unfortunately, Qt Creator doesn't yet provide a mechanism for directly creating subdirectory projects, so at least for the time being you can use the `CMakeLists.txt` files we created here as templates for new projects.