Getting started with CMake

CMake is a group of tools that allow to build, test, and package applications. Just like Qt, it is available on all major development platforms. It is also supported by various IDE's, including Qt Creator.

In this section we will show the most basic way to use Qt in a CMake project. First, we create a basic console application. Then, we extend the project into a GUI application that uses Qt Widgets.

If you want to know how to build an existing CMake project with Qt, see the documentation on how to build projects with CMake on the command line.

Building a C++ console application

Here is a typical `CMakeLists.txt` file for a console application written in C++ using Qt:

```
cmake_minimum_required(VERSION 3.16)

project(helloworld VERSION 1.0.0 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Qt6 REQUIRED COMPONENTS Core)
qt_standard_project_setup()

add_executable(helloworld
    main.cpp
)

target_link_libraries(helloworld PRIVATE Qt6::Core)
```

Let's go through the content.

```
cmake_minimum_required(VERSION 3.16)
```

cmake_minimum_required() specifies the minimum CMake version that the application requires. Qt itself requires at least CMake version 3.16. If you use a Qt that was built statically - the default in Qt for iOS and Qt for WebAssembly - you need CMake 3.21.1 or newer.

```
project(helloworld VERSION 1.0.0 LANGUAGES CXX)
```

`project()` sets a project name and the default project version. The `LANGUAGES` argument tells CMake that the program is written in C++.

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

Qt 6 requires a compiler supporting C++ version 17 or newer. Enforcing this by setting the `CMAKE_CXX_STANDARD`, `CMAKE_CXX_STANDARD_REQUIRED` variables will let CMake print an error if the compiler is too old.

```
find_package(Qt6 REQUIRED COMPONENTS Core)
```

This tells CMake to look up Qt 6, and import the `Core` module. There is no point in continuing if `CMake` cannot locate the module, so we do set the `REQUIRED` flag to let CMake abort in this case.

If successful, the module will set some CMake variables documented in Module variables. It furthermore imports the `Qt6::Core` target that we use below.

For `find_package` to be successful, `CMake` must find the Qt installation. There are different ways you can tell `CMake` about Qt, but the most common and recommended approach is to set the CMake cache variable `CMAKE_PREFIX_PATH` to include the Qt 6 installation prefix. Note that Qt Creator will handle this transparently for you.

```
qt_standard_project_setup()
```

The qt_standard_project_setup command sets project-wide defaults for a typical Qt application.

Among other things, this command sets the `CMAKE_AUTOMOC` variable to `ON`, which instructs CMake to automatically set up rules so that Qt's Meta-Object Compiler (moc) is called transparently, when required.

See qt_standard_project_setup's reference for details.

```
add_executable(helloworld
    main.cpp
)
```

add_executable() tells CMake that we want to build an executable (so not a library) called `helloworld` as a

explicitly listed so that they are processed by the Meta-Object Compiler (moc).

For less trivial projects, you may want to call qt_add_executable() instead. It is a wrapper around the built-in `add_executable()` command, providing additional logic to automatically handle things like linking of Qt plugins in static Qt builds, platform-specific customization of library names and so on.

For creating libraries, see qt_add_library.

```
target_link_libraries(helloworld PRIVATE Qt6::Core)
```

Finally, `target_link_libraries` tells CMake that the `helloworld` executable makes use of Qt Core by referencing the `Qt6::Core` target imported by the `find_package()` call above. This will not only add the right arguments to the linker, but also makes sure that the right include directories, compiler definitions are passed to the C++ compiler. The `PRIVATE` keyword is not strictly necessary for an executable target, but it is good practice to specify it. If `helloworld` was a library rather than an executable, then either `PRIVATE` or `PUBLIC` should be specified (`PUBLIC` if the library mentions anything from `Qt6::Core` in its headers, `PRIVATE` otherwise).

Building a C++ GUI application

In the last section we showed the CMakeLists.txt file for a simple console application. We will now extend it to create a GUI application that uses the Qt Widgets module.

This is the full project file:

```
cmake_minimum_required(VERSION 3.16)

project(helloworld VERSION 1.0.0 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Qt6 REQUIRED COMPONENTS Widgets)
qt_standard_project_setup()

add_executable(helloworld
    mainwindow.ui
    mainwindow.cpp
    main.cpp
)

target_link_libraries(helloworld PRIVATE Qt6::Widgets)

set_target_properties(helloworld PROPERTIES
    WIN32_EXECUTABLE ON
    MACOSX_BUNDLE ON
)
```

Let's walk through the changes we have made.

In the `find_package` call, we replace `Core` with `Widgets`. This will locate the `Qt6Widgets` module and provide the `Qt6::Widgets` targets we later link against.

Note that the application will still link against `Qt6::Core`, because `Qt6::Widgets` depends on it.

```
qt_standard_project_setup()
```

In addition to `CMAKE_AUTOMOC`, qt_standard_project_setup sets the `CMAKE_AUTOUIC` variable to `ON`. This will automatically create rules to invoke Qt's User Interface Compiler (uic) on `.ui` source files.

```
add_executable(helloworld
    mainwindow.ui
    mainwindow.cpp
    main.cpp
)
```

We add a Qt Designer file (`mainwindow.ui`) and its corresponding C++ source file (`mainwindow.cpp`) to the application target's sources.

```
target_link_libraries(helloworld PRIVATE Qt6::Widgets)
```

In the `target_link_libraries` command, we link against `Qt6::Widgets` instead of `Qt6::Core`.

```
set_target_properties(helloworld PROPERTIES
    WIN32_EXECUTABLE ON
    MACOSX_BUNDLE ON
)
```

Finally, we set properties on our application target with the following effects:

- Prevent the creation of a console window on Windows.
- Create an application bundle on macOS.

See the CMake Documentation for more information about these target properties.

Structuring projects

Projects that contain more than just one target will benefit from a clear project file structure. We will use CMake's subdirectory feature.

As we plan to extend the project with more targets, we move the source files of the application into a subdirectory and create a new `CMakeLists.txt` in there.

```
├── CMakeLists.txt
└── src
    └── app
        ├── CMakeLists.txt
        ├── main.cpp
        ├── mainwindow.cpp
        ├── mainwindow.h
        └── mainwindow.ui
```

The top-level `CMakeLists.txt` contains the overall project setup, `find_package` and `add_subdirectory` calls:

```
cmake_minimum_required(VERSION 3.16)

project(helloworld VERSION 1.0.0 LANGUAGES CXX)

set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

find_package(Qt6 REQUIRED COMPONENTS Widgets)
qt_standard_project_setup()

add_subdirectory(src/app)
```

Variables that are set in this file are visible in subdirectory project files.

The application's project file `src/app/CMakeLists.txt` contains the executable target:

```
add_executable(helloworld
    mainwindow.ui
    mainwindow.cpp
    main.cpp
)

target_link_libraries(helloworld PRIVATE Qt6::Widgets)

set_target_properties(helloworld PROPERTIES
    WIN32_EXECUTABLE ON
    MACOSX_BUNDLE ON
)
```

Such a structure will make it easy to add more targets to the project such as libraries or unit tests.

Building libraries

As the project grows, you may want to turn parts of your application code into a library that is used by the application and possibly unit tests. This section shows how to create such a library.

For the sake of simplicity, the library consists of just one C++ source file and its corresponding header file that is included by the application's `main.cpp`:

```
<project root>
├── CMakeLists.txt
└── src
    ├── app
    │   ├── ...
    │   └── main.cpp
    └── businesslogic
        ├── CMakeLists.txt
        ├── businesslogic.cpp
        └── businesslogic.h
```

Let's have a look at the library's project file (`src/businesslogic/CMakeLists.txt`).

```
add_library(businesslogic STATIC
    businesslogic.cpp
)
target_link_libraries(businesslogic PRIVATE Qt6::Core)
target_include_directories(businesslogic INTERFACE ${CMAKE_CURRENT_SOURCE_DIR})
```

Let's go through the content.

```
add_library(businesslogic STATIC
    businesslogic.cpp
)
```

The add_library command creates the library `businesslogic`. Later, we will let the application link against this target.

The `STATIC` keyword denotes a static library. If we wanted to create a shared or dynamic library, we would use the `SHARED` keyword.

```
target_link_libraries(businesslogic PRIVATE Qt6::Core)
```

We have a static library and don't actually have to link other libraries. But as our library uses classes from `QtCore`, we add a link dependency to `Qt6::Core`. This pulls in the necessary `QtCore` include paths and preprocessor defines.

```
target_include_directories(businesslogic INTERFACE ${CMAKE_CURRENT_SOURCE_DIR})
```

automatically added as an include path to all targets using our library.

This frees us in `main.cpp` from using relative paths to locate `businesslogic.h`. Instead, we can just write

```
#include <businesslogic.h>
```

Last, we must add the library's subdirectory to the top-level project file:

```
add_subdirectory(src/app)
add_subdirectory(src/businesslogic)
```

Using libraries

To use the library we created in the previous section, we instruct CMake to link against it:

```
target_link_libraries(helloworld PRIVATE
    businesslogic
    Qt6::Widgets)
```

This ensures that `businesslogic.h` is found when main.cpp is compiled. Furthermore, the businesslogic static library will become a part of the `helloworld` executable.

In CMake terms, the library `businesslogic` specifies *usage requirements* (the include path) that every consumer of our library (the application) has to satisfy. The `target_link_libraries` command takes care of that.

Adding resources

We want to display some images in our application, so we add them using the Qt Resource System.

```
qt_add_resources(helloworld imageresources
    PREFIX "/images"
    FILES logo.png splashscreen.png
)
```

The qt_add_resources command automatically creates a Qt resource containing the referenced images. From the C++ source code, you can access the images by prepending the specified resource prefix:

```
logoLabel->setPixmap(QPixmap(":/images/logo.png"));
```

The qt_add_resources command takes as the first argument either a variable name or a target name. We recommend to use the target-based variant of this command as shown in the example above.

Translations of strings in a Qt project are encoded in `.ts` files. See Internationalization with Qt for details.

To add `.ts` files to your project, use the qt_add_translations command.

The following example adds a German and a French translation file to the `helloworld` target:

```
qt_add_translations(helloworld
    TS_FILES helloworld_de.ts helloworld_fr.ts)
```

This creates build system rules to automatically generate `.qm` files from the `.ts` files. By default, the `.qm` files are embedded into a resource and are accessible under the `":/i18n"` resource prefix.

To update the entries in the `.ts` file, build the `update_translations` target:

```
$ cmake --build . --target update_translations
```

To trigger the generation of the `.qm` files manually, build the `release_translations` target:

```
$ cmake --build . --target release_translations
```

For more information about how to influence the handling of `.ts` files and the embedding into a resource, see the qt_add_translations documentation.

The qt_add_translations command is a convenience wrapper. For more fine-grained control of how `.ts` files and `.qm` files are handled, use the underlying commands qt_add_lupdate and qt_add_lrelease.

Further reading

The official CMake Documentation is an invaluable source for working with CMake.

The official CMake Tutorial covers common build system tasks.

The book Professional CMake: A Practical Guide provides a great introduction to the most relevant CMake features.

‹ Build with CMake

Building projects on the command line ›

Contact Us

Company

About Us
Investors
Newsroom
Careers
Office Locations

Licensing

Terms & Conditions
Open Source
FAQ

Support

Support Services
Professional Services
Partners
Training

For Customers

Support Center
Downloads
Qt Login
Contact Us
Customer Success

Community

Contribute to Qt
Forum
Wiki
Downloads
Marketplace

Feedback Sign In