

Qt 6.4 > Build with CMake > [Getting started with CMake](#)

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

A project is defined by files written in the CMake language. The main file is called `CMakeLists.txt`, and is usually placed in the same directory as the actual program sources.

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

```
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 argument tells CMake that the program is written in C++. `LANGUAGES`

```
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` variable will let CMake print an error if the compiler is too old.

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

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

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

```
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 variable to , which instructs CMake to automatically set up rules so that Qt's [Meta-Object Compiler \(moc\)](#) is called transparently, when required. `CMAKE_AUTOMOC`

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 as a target. The target should be built from the C++ source file `helloworldmain.cpp`

Note that you typically do not list header files here. This is different from [qmake](#), where header files need to be 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 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. `add_executable()`

For creating libraries, see [qt_add_library](#).

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

Finally, tells CMake that the executable makes use of [Qt Core](#) by referencing the target imported by the 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 keyword is not strictly necessary for an executable target, but it is good practice to specify it. If was a library rather than an executable, then either or should be specified (if the library mentions anything from in its headers, otherwise). `target_link_librarieshelloworldQt6::Corefind_package()PRIVATEhelloworldPRIVATEPUBLICPUBLICQt6::CorePRIVATE`

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
```

Let's walk through the changes we have made.

```
find_package(Qt6 REQUIRED COMPONENTS Widgets)
```

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

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

```
qt_standard_project_setup()
```

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

```
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. `mainwindow.ui` `mainwindow.cpp`

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

In the 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.

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

The top-level contains the overall project setup, and calls `add_subdirectory` to build the subdirectory.

```
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 contains the executable target: `src/app/CMakeLists.txt`

```
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.

Our application currently contains business logic directly in `main.cpp`. We extract the code into a new static library called in the subdirectory as explained in the [previous section](#). `main.cpp` becomes `src/businesslogic/main.cpp`

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

```
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 include paths and preprocessor defines.

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

The library API is defined in the header file `businesslogic.h`. By calling `target_include_directories`, we make sure that the absolute path to the directory is automatically added as an include path to all targets using our library.

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

```
#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 executable `helloworld`.

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

Adding translations

Translations of strings in a Qt project are encoded in files. See [Internationalization with Qt](#) for details.

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

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

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

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

要手动触发文件的生成，请构建目标: `.qmrelease_translations`

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

有关如何影响文件处理和嵌入到资源中的详细信息，请参阅[qt_add_translations文档](#)。 `.ts`

`qt_add_translations`命令是一个方便的包装器。若要更精细地控制文件和文件的处理方式，请使用[基础命令](#)`qt_add_lupdate`和[qt_add_lrelease](#)。 `.ts.qm`

进一步阅读

官方的 [CMake 文档](#)是使用 CMake 的宝贵来源。

官方的 [CMake 教程](#)涵盖了常见的构建系统任务。


《[专业 CMake：实用指南](#)》一书很好地介绍了最相关的 CMake 功能。

[◀ 使用“聚合”进行构建](#)

[在命令行上生成项目 >](#)



The Qt Company



联系我们

公司

关于我们

投资者

编辑部

职业

办公地点

支持

支持服务

专业服务

合作伙伴

训练

发牌

条款及细则

开源

常见问题

对于客户

支持中心

下载

秦特登录

联系我们

客户成功案例

社区

论坛
维基
下载
市场

© 2022 Qt公司

[反馈](#) [登录](#)