Search

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 , and is usually placed in the same directory as the actual program sources.CMakeCMakeLists.txt

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 , variables will let CMake print an error if the compiler is too old.CMAKE_CXX_STANDARDCMAKE_CXX_STANDARD_REQUIRED

Qt DOCUMENTATION

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_AUTOMOCON`

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 . This will locate the module and provide the targets we later link against.find_packageCoreWidgetsQt6WidgetsQt6::Widgets

Note that the application will still link against , because depends on it.Qt6::CoreQt6::Widgets

```
qt_standard_project_setup()
```

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

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

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

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

In the command, we link against instead of .target_link_librariesQt6::WidgetsQt6::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 in there.CMakeLists.txt

```
<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:CMakeLists.txtfind_packageadd_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 . We extract the code into a new static library called in the subdirectory as explained in the previous section.`main.cppbusinesslogic"src/businesslogic"`

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
```

让我们看一下库的项目文件（）。`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})
```

让我们来看看内容。

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

关键字表示静态库。如果我们想创建一个共享或动态库，我们将使用关键字。STATICSHARED

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

器定义。QtCoreQt6::CoreQtCore

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

库 API 在头文件中定义。通过调用target_include_directories，我们确保使用我们的库自动将目录的绝对路径添加为所有目标的包含路径。businesslogic/businesslogic.hbusinesslogic

这使我们免于使用相对路径来定位。相反，我们可以只写main.cppbusinesslogic.h

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

最后，我们必须将库的子目录添加到顶级项目文件中：

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

使用库

要使用我们在上一节中创建的库，我们指示 CMake 针对它进行链接：

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

这可确保在编译 main.cpp 时找到它。此外，商业静态库将成为可执行文件的一部分。businesslogic.hhelloworld

在 CMake 术语中，库指定了我们库（应用程序）的每个使用者都必须满足*的使用要求*（包含路径）。该命令负责处理。businesslogictarget_link_libraries

添加资源

我们希望在应用程序中显示一些图像，因此我们使用Qt资源系统添加它们。

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

qt_add_resources命令会自动创建包含引用图像的 Qt 资源。从源代码C++，您可以通过在指定的资源前缀之前访问图像：

```
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..ts

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

```
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
```

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

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

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

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

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 ›

Forum
Wiki
Downloads
Marketplace

Feedback Sign In