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
```

让我们来看看我们所做的更改。

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

在调用中，我们将 替换为 。这将找到模块并提供我们稍后链接的目标。find_packageCoreWidgetsQt6WidgetsQt6::Widgets

请注意，应用程序仍将链接到 ，因为依赖于 它。Qt6::CoreQt6::Widgets

```
qt_standard_project_setup()
```

除了，qt_standard_project_setup还将变量设置为 。这将自动创建规则以在源文件上调用Qt的用户界面编译器（uic）。CMAKE_AUTOMOCCMAKE_AUTOUICON.ui

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

我们将 Qt 设计器文件 （） 及其相应的C++源文件 （） 添加到应用程序目标的源中。mainwindow.uimainwindow.cpp

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

在命令中，我们链接针对 而不是 。target_link_librariesQt6::WidgetsQt6::Core

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

最后，我们在应用程序目标上设置属性，并具有以下效果：

- 阻止在窗口上创建控制台窗口。
- 在 macOS 上创建应用程序包。

有关这些目标属性的详细信息，请参阅 CMake 文档。

构建项目

包含多个目标的项目将受益于清晰的项目文件结构。我们将使用 CMake 的子目录功能。

当我们计划使用更多目标扩展项目时，我们将应用程序的源文件移动到子目录中，并在那里创建一个新目录。CMakeLists.txt

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

顶层包含整个项目设置，并调用：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
```

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
)
```

```
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 , we add a link dependency to . This pulls in the necessary include paths and preprocessor defines.`QtCoreQt6::CoreQtCore`

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

The library API is defined in the header file . 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.`businesslogic/businesslogic.hbusinesslogic`

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

```
#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 is found when main.cpp is compiled. Furthermore, the businesslogic static library will become a part of the executable.`businesslogic.hhelloworld`

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.`businesslogictarget_link_libraries`

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.`.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 ›

Contribute to Qt
Forum
Wiki
Downloads
Marketplace

Feedback Sign In