

Cross-Compiling the C++ Example Project

Pieter P

The Greeter Library

For this example, we'll create a very simple library with a single function that just takes a name and an output stream as arguments, and that prints a greeting message to this stream. It's basically a "Hello, World!" example, but as a library for demonstration purposes.

The structure of the library will be as follows:

```
greeter
├── CMakeLists.txt
├── include
│   └── greeter
│       └── greeter.hpp
├── src
│   └── greeter.cpp
└── test
    ├── CMakeLists.txt
    └── greeter.test.cpp
```

This structure is very common for C++ libraries: the function prototypes/declarations will be in the header file `greeter.hpp`. The implementations for these functions are in the implementation file `greeter.cpp`.

The `CMakeLists.txt` file in the `greeter` directory specifies how the library should be compiled, and where to find the headers. Additionally, there's a `test` folder with unit tests in `greeter.test.cpp`. The `CMakeLists.txt` file in this folder specifies how to compile and link the tests executable.

greeter.hpp

```
1 #pragma once
2
3 #include <iosfwd> // std::ostream
4 #include <string> // std::string
5
6 namespace greeter {
7
8 /**
9  * @brief Function that greets a given person.
10  *
11  * @param name
12  *       The name of the person to greet.
13  * @param os
14  *       The output stream to print the greetings to.
15  */
16 void sayHello(const std::string &name, std::ostream &os);
17
18 } // namespace greeter
```

greeter.cpp

```
1 #include <greeter/greeter.hpp>
2 #include <iostream> // std::endl, <<
3
4 namespace greeter {
5
6 void sayHello(const std::string &name, std::ostream &os) {
7     os << "Hello, " << name << "!" << std::endl;
8 }
9
10 } // namespace greeter
```

CMakeLists.txt

```

1 # Add a new library with the name "greeter" that is compiled from the source
2 # file "src/greeter.cpp".
3 add_library(greeter
4     src/greeter.cpp
5 )
6
7 # The public header files for greeter can be found in the "include" folder, and
8 # they have to be passed to the compiler, both for compiling the library itself
9 # and for using the library in a other implementation files (such as
10 # applications/hello-world/hello-world.cpp). Therefore the "include" folder is a
11 # public include directory for the "greeter" library. The paths are different
12 # when building the library and when installing it, so generator expressions are
13 # used to distinguish between these two cases.
14 # See https://cmake.org/cmake/help/v3.17/command/target_include_directories.html
15 # for more information.
16 # If you have private headers in the "src" folder, these have to be added as
17 # well. They are private because they are only needed when building the library,
18 # not when using it from a different implementation file.
19 target_include_directories(greeter
20     PUBLIC
21         ${INSTALL_INTERFACE:include}
22         ${BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include}
23     PRIVATE
24         ${BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/src}
25 )
26
27 # Include the tests in the "test" folder.
28 add_subdirectory(test)

```

The unit tests

The test file only contains a single unit test, and just serves as an example. It uses the [Google Test framework](#).

The tests can only be run on the build computer if we're not cross-compiling.

greeter.test.cpp

```

1 #include <greeter/greeter.hpp>
2 #include <gtest/gtest.h>
3 #include <sstream>
4
5 /**
6  * @test
7  *
8  * Check that the output of the greeter::sayHello function matches the
9  * documentation.
10 */
11 TEST(greeter, sayHello) {
12     std::ostringstream ss;
13     greeter::sayHello("John Doe", ss);
14     EXPECT_EQ(ss.str(), "Hello, John Doe!\n");
15 }

```

test/CMakeLists.txt

```

1 # Add a new test executable with the name "greeter.test" that is compiled from
2 # the source file "greeter.test.cpp".
3 add_executable(greeter.test
4     greeter.test.cpp
5 )
6
7 # The test executable requires the "greeter" library (it's the library under
8 # test), as well as the Google Test main function to actually run all tests.
9 target_link_libraries(greeter.test
10     greeter
11     gtest_main
12 )
13
14 # Only look for tests if we're not cross-compiling. When cross-compiling, it's
15 # not possible to run the test executable on the computer that's performing the
16 # build.
17 if (NOT CMAKE_CROSSCOMPILING)
18     include(GoogleTest)
19     gtest_discover_tests(greeter.test)
20 endif()

```

The main Hello World program

Finally, the Greeter library can be used to create a simple Hello World program.

hello-world.cpp

```

1 #include <greeter/greeter.hpp> // Our own custom library
2
3 #include <iostream> // std::cout, std::cin
4 #include <string> // std::getline
5
6 int main(int argc, char *argv[]) {
7     std::string name;
8     if (argc > 1) { // If the user passed arguments to our program
9         name = argv[1]; // The name is the first argument
10    } else { // If not, ask the user for his name
11        std::cout << "Please enter your name: ";
12        std::getline(std::cin, name);
13    }
14    greeter::sayHello(name, std::cout); // Greet the user
15 }

```

CMakeLists.txt

```

1 # Add a new executable with the name "hello-world" that is compiled from the
2 # source file "hello-world.cpp".
3 add_executable(hello-world
4     hello-world.cpp
5 )
6
7 # The "hello-world" program requires the "greeter" library.
8 # The target_link_libraries command ensures that all compiler options such as
9 # include paths are set correctly, and that the executable is linked with the
10 # library as well.
11 target_link_libraries(hello-world
12     greeter
13 )

```