Applause from you and 157 others

Mirko Kiefer Follow
Traveler, developer and drone pilot.
Feb 22, 2017 · 4 min read

CMake by Example

When trying to learn CMake I could not find any good introduction. The CMake documentation is quite comprehensive but not suitable for a beginner. There are some useful tutorials linked on the CMake Wiki but most of them only cover very specific problems or are too basic. So I wrote this short CMake introduction as a distilled version of what I found out after working through the docs and following stackoverflow questions.

Its a work in progress and I will try to continuously improve it.

CMake is a meta build tool that allows you to generate native build scripts for a range of platforms:

- Unix Makefiles
- Xcode
- Visual Studio
- CodeBlocks

- Eclipse
- and more…

See the full list of CMake generators.

Using CMake with executables

Lets assume we have a simple app with a single .c file.

We by creating a `CMakeLists.txt` file in the root of our project.

```
cmake_minimum_required(VERSION 2.8)

project(app_project)

add_executable(myapp main.c)

install(TARGETS myapp DESTINATION bin)
```

Thats all we need to be able to build our app with any of the available generators.

`add_executable` defines our binary with all linked source files.

`install` tells cmake to install our binary into the `bin` directory of the install directory.

Building

CMake supports out-of-source builds—so all our compiled code goes into a directory separate to the sources.

To start a build we create a new folder:

```
mkdir _build
cd _build
```

And call cmake with the path to the project’s root (in this case the parent folder):

```
cmake ..
```

This will generate build scripts using the default generator—on Linux/OSX this should be Makefiles.

By default cmake will install our build into the system directories. To define a custom install directory we simply pass it to cmake:

```
cmake .. -DCMAKE_INSTALL_PREFIX=../_install
```

To run the build script you can simply use the Makefile:

```
make
make install
```

We can now run our binary from the install directory:

```
../_install/bin/myapp
```

If we wanted to use a different generator we pass it to cmake using the `-G` parameter:

```
cmake .. -GXcode
```

This will output a readily configured Xcode project to build our app.

Using CMake with libraries

To build a library we use a similar script:

```
cmake_minimum_required(VERSION 2.8)

project(libtest_project)

add_library(test STATIC test.c)

install(TARGETS test DESTINATION lib)
install(FILES test.h DESTINATION include)
```

CMake will build the library as `libtest.a` and install it into lib folder of the install directory.
We also include our public header file into the install step and tell cmake to put it into `include`.

Instead of a static library we can build a shared lib as well:

```
add_library(test SHARED test.c)
```

Linking libraries to executables with CMake

We can extend our executable from above by linking it to our libray `libtest.a`.

Let's start by adding the library's directory as a subdirectory to our myapp project.

Now, we can use the library defined in CMakeLists.txt of libtest_project in myapp's CMakeLists.txt:

```
cmake_minimum_required(VERSION 2.8)
```

```
project(myapp)

add_subdirectory(libtest_project)

add_executable(myapp main.c)

target_link_libraries(myapp test)

install(TARGETS myapp DESTINATION bin)
```

`add_subdirectory` makes the library `test` defined in libtest*project available to the build.*
In `target_link_libraries` we tell CMake to link it to our executable. CMake will make sure to first build test before linking it to myapp.

Including external libraries using other build systems

While CMake enjoys increasing interest, there are still plenty of libraries using native build systems like Unix Makefiles. You can make use of them in your CMake project without having to re-write their build scripts.

All we need is CMake’s support for external projects and imported libraries:

```
ExternalProject_Add(project_luajit
  URL http://luajit.org/download/LuaJIT-2.0.1.tar.gz
  PREFIX ${CMAKE_CURRENT_BINARY_DIR}/luajit-2.0.1
  CONFIGURE_COMMAND ""
  BUILD_COMMAND make
  INSTALL_COMMAND make install
  PREFIX=${CMAKE_CURRENT_BINARY_DIR}/luajit-2.0.1
)

ExternalProject_Get_Property(project_luajit install_dir)

add_library(luajit STATIC IMPORTED)

set_property(TARGET luajit PROPERTY IMPORTED_LOCATION
${install_dir}/lib/libluajit-5.1.a)

add_dependencies(luajit project_luajit)
```

```
add_executable(myapp main.c)

include_directories(${install_dir}/include/luajit-2.0)

target_link_libraries(myapp luajit)
```

`ExternalProject_Add` allows us to add an external project as a target to our project. If you don't specify commands like `BUILD_COMMAND` or `INSTALL_COMMAND`, CMake will look for a `CMakeLists.txt` in the external project and execute it.
In our case we want to make use of the luajit library which is built using a Makefile.

`add_library` supports the import of already built libraries as well - we just have to set its `IMPORTED_LOCATION` property.
Calling `ExternalProject_Add` only specifies the external project as a target but does not automatically build it. It will only be built if we add a dependency to it. We therefore call `add_dependencies` to make our imported library dependent on the external project.

Finally we can link our imported library just like a “normal” library with `target_link_libraries`.