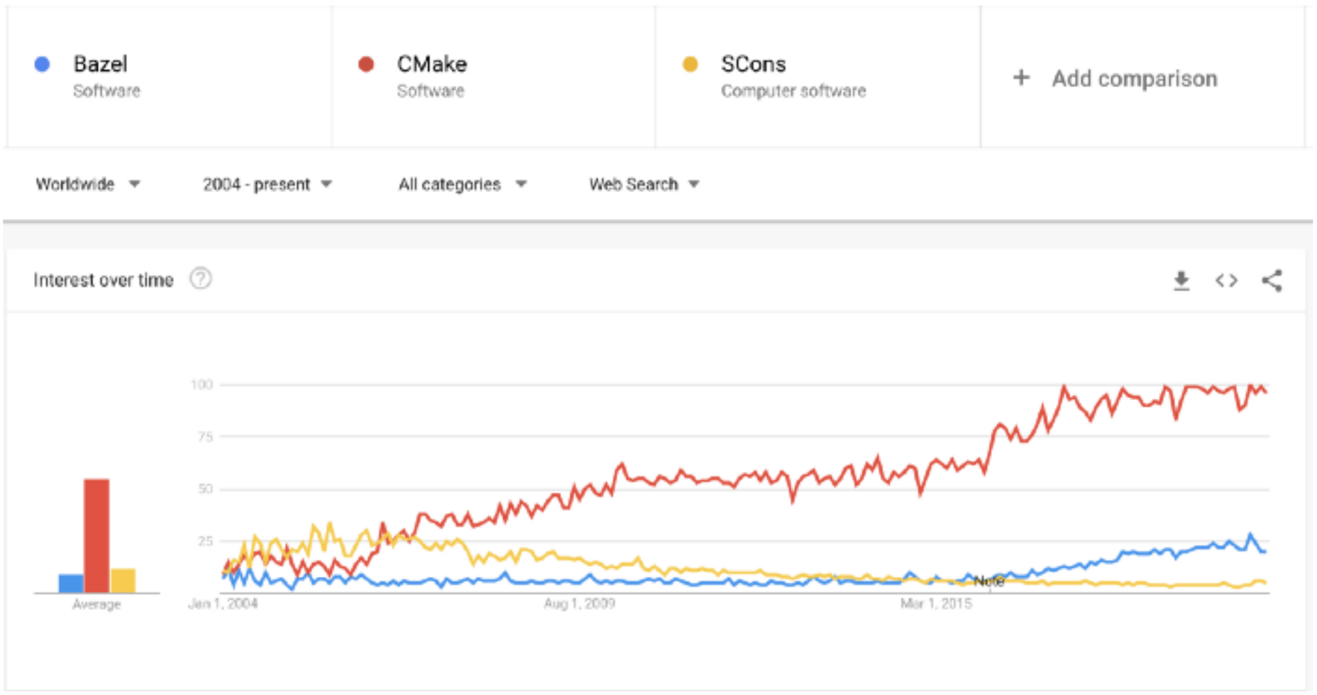
Simple Modern CMake Tutorial Part 1

Short introduction of “Object oriented” Modern CMake

Kohei Otsuka Follow
May 24, 2020 · 7 min read ★

Photo by Artem Sapegin on Unsplash

CMake[1] is an open-source tool for managing the build process of software. CMake is a popular tool especially among the C++ community. Many C++ projects are currently using CMake. Also as you can see the Google trend result below, more people are interested in CMake, compared to other build systems such as SCons[2], Bazel[3] (So it is certainly a good idea to get familiar with it!).



Google trend on different build systems (Bazel, CMake, SCons).

Actually, CMake was not as popular as now before, due to some syntax and the less elegant features it had. However, since CMake 3.x (released on 06.2014), it added new features that make CMake more powerful, clean, and elegant. The newer version of CMake (3.1 or late) is sometimes called "Modern CMake"[3].

In this post, I will explain about certain aspects of the Modern CMake with some simple examples (Here is the link to the Github repository).
Especially, the points I would like to cover are,

- The Concept of treating the "build target" (i.e. executable or library you are building with CMake) as an "object" in Object-oriented programming in order to better structure your projects and the idea of the inheritance of the build dependencies transitively (Transitive dependency).
- How to export/install CMake package and import it from another CMake project.

Effective CMake[4] talk from Daniel Pfeifer gave me a lot of useful insights to understand the Modern CMake concept[5] and helped me to write this post.

Example CMake packages

First, Let me go through the project structure we are going to use for this tutorial.

```
cmake_tutorial
|__ app                    # MyApplication package
|   |__ CMakeLists.txt
|   |__ src
|       |__ main.cpp
|
|
|__ my_libraries          # MyLibraries package
|   |__ CMakeLists.txt
|   |__ cmake
|   |   |__ MyLibrariesConfig.cmake
|   |
|   |__ include
|   |   |__ my_libraries
|   |      |__ my_library_a.hpp
|   |
|   |__ src
|       |__ my_library_a.cpp
|
|
|__ some_libraries        # SomeLibraries package
   |__ CMakeLists.txt
   |__ cmake
   |   |__ SomeLibrariesConfig.cmake
   |
   |__ include
   |   |__ some_libraries
   |     |__ some_library_a.hpp
   |     |__ some_library_b.hpp
   |__ src
       |__ some_library_a.cpp
       |__ some_library_b.cpp
```

The *app* directory contains MyApplication executable package. The *my_libraries* directory contains MyLibraries libraries package. The *some_libraries* contains SomeLibraries libraries package. MyApplication depends on MyLibraries, which in turn depends on SomeLibraries. So the relationship between the 3 packages is like below.

```
MyApplication --depends--> MyLibraries --depends--> SomeLibraries
```

In general, there are three ways to access a CMake project from another CMake project: subdirectory, exported build directories, and importing prebuilt package which is already installed. In this post, I want to explain the third option; how to install a CMake package and import it from another CMake project (So, Let's assume MyApplication, MyLibraries and SomeLibraries are developed separately as separate packages). Hence, the three packages are **not** in the same CMake tree.

Install SomeLibraries package

Since everyone depends on SomeLibraries (MyApplication indirectly and MyLibraries directly), we need to build and install so that it becomes available for others. CMakeLists.txt in some_libraries directory looks like below,

```
## CMakeLists.txt for SomeLibraries ##

cmake_minimum_required(VERSION 3.5)
project(SomeLibraries VERSION 1.0 LANGUAGES CXX)

add_library(SomeLibraryA src/some_library_a.cpp)

target_include_directories(SomeLibraryA
PUBLIC
$<INSTALL_INTERFACE:include>
$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
)

add_library(SomeLibraryB src/some_library_b.cpp)

target_include_directories(SomeLibraryB
PUBLIC
$<INSTALL_INTERFACE:include>
$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
)

install(TARGETS SomeLibraryA SomeLibraryB
EXPORT SomeLibraries-export
LIBRARY DESTINATION lib
ARCHIVE DESTINATION lib
)

install(EXPORT SomeLibraries-export
FILE
SomeLibrariesTargets.cmake
NAMESPACE
SomeLibraries::
DESTINATION
lib/cmake/SomeLibraries
)

install(FILES
${CMAKE_CURRENT_SOURCE_DIR}/include/SomeLibraries/some_library_a.hpp
${CMAKE_CURRENT_SOURCE_DIR}/include/SomeLibraries/some_library_b.hpp
DESTINATION "include/SomeLibraries")

install(FILES
${CMAKE_CURRENT_SOURCE_DIR}/cmake/SomeLibrariesConfig.cmake
DESTINATION "lib/cmake/SomeLibraries" )
```

First, you might have noticed that SomeLibraries package defines 2 targets, SomeLibraryA and SomeLibraryB. They both have public headers some_library_a.hpp and some_library_b.hpp that need to be installed (See the directory tree shown before).

The interesting part starts from the part below.

```
target_include_directories(SomeLibraryA
PUBLIC
$<INSTALL_INTERFACE:include>
$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
)
```

With *target_include_directories*, it is specifying include path for the target, SomeLibraryA. Actually, you can see this as if the object "SomeLibraryA" is calling its member function *target_include_directories*, like Object-oriented programming paradigm to set the data (include path, in this case) to the object. Like a normal Setter function of some class, it is setting a property of the object. The specified include path is used to build the SomeLibraryA target, but also, since It has the *PUBLIC* keyword, the include path will be transitively applied to the user of SomeLibraryA when it adds the dependency to SomeLibraryA via *target_link_libraries (explained later)*. In this example, SomeLibraryA has a header which it publicly exposes.

The generator expressions below are used to selectively choose the include path depends on the situation. The include path specified in *$<BUILD_INTERFACE:>* is used when building the SomeLibraryA target itself. On the other hand, Let's say we built the

SomeLibraryA target, and we successfully installed it so that it becomes available to another CMake project(We will dive into the installing part later). Then, If we use *find_pacakge(*SomeLibraryA*)* and import the SomeLibraryA target in another CMake project that depends on SomeLibraryA, the include path specified in *$<INSTALL_INTERFACE:>* is used. The same thing applies to SomeLibraryB.

Next, we need to specify installation locations for the build artifacts of SomeLibraryA(B).

```
install(TARGETS SomeLibraryA SomeLibraryB
EXPORT SomeLibraries-export
LIBRARY DESTINATION lib
ARCHIVE DESTINATION lib
)

install(EXPORT SomeLibraries-export
FILE
SomeLibrariesTargets.cmake
NAMESPACE
SomeLibraries::
DESTINATION
lib/cmake/SomeLibraries
)
```

At the 1st *install* above, it is setting the installation locations for each target and putting the information out to SomeLibraries-export. At the 2nd *install,* it is writing out the information that was set to SomeLibraries-export to SomeLibrariesTargets.cmake file and copying the SomeLibrariesTargets.cmake file to lib/cmake/SomeLibraries directory. NAMESPACE can be specified for each exported target. You can think this like the namespace from a programming language like C++. (Beside the advantage normal namespace concept brings, it also prevents some minor issues specific to CMake context but I will skip it now not to diverge. In general it is a good practice to use namespace).

Then, it is copying the public headers into the installation location.

```
install(FILES
${CMAKE_CURRENT_SOURCE_DIR}/include/SomeLibraries/some_library_a.hpp
${CMAKE_CURRENT_SOURCE_DIR}/include/SomeLibraries/some_library_b.hpp
DESTINATION "include/SomeLibraries")
```

This installation location should correspond to where we specified before with *target_include_directories,* $<INSTALL_INTERFACE:include>. Otherwise, the public headers cannot be found when building another package that depends on SomeLibraryA(B) since the imported SomeLibraryA(B) property doesn't match with the actual installation.

```
install(FILES
${CMAKE_CURRENT_SOURCE_DIR}/cmake/SomeLibrariesConfig.cmake
DESTINATION "lib/cmake/SomeLibraries" )
```

Here, it is copying the SomeLibrariesConfig.cmake in cmake/ to the conventional location "lib/cmake/SomeLibraries" (The same location where we copy SomeLibrariesTargets.cmake, see a couple of paragraphs before.

SomeLibrariesConfig.cmake is like below,

```
include(CMakeFindDependencyMacro)

include("${CMAKE_CURRENT_LIST_DIR}/SomeLibrariesTargets.cmake")
```

CMake search for this SomeLibrariesConfig.cmake file when another package uses *find_package*(SomeLibraries). SomeLibrariesConfig.cmake includes SomeLibrariesTargets.cmake file which contains the exported information of targets SomeLibraryA and SomeLibraryB, this is how the targets and their properties are imported into another project.

That's it. This is the minimum you need to specify in CMakeLists.txt and SomeLibrariesConfig.cmake to install the SomeLibraryA(B) targets from SomeLibraries package so that another packages can import and use them. Now, you just need to run cmake and execute make install like below.

```
cd some_libraries
mkdir build
cd build
cmake ..
make install
```

Import the SomeLibraryA(B) targets from SomeLibraries package into MyLibraries project

Now, we have built and installed SomeLibraries package, we are ready to build MyLibraries package that depends on it. The CMakeLists.txt file for the MyLibraries package looks like below,

```
## CMakeLists.txt for MyLibraries ##

cmake_minimum_required(VERSION 3.5)
project(MyLibraries VERSION 1.0 LANGUAGES CXX)

find_package(SomeLibraries REQUIRED)

add_library(MyLibraryA src/my_library_a.cpp)

target_include_directories(MyLibraryA
PUBLIC
$<INSTALL_INTERFACE:include>
$<BUILD_INTERFACE:${CMAKE_CURRENT_SOURCE_DIR}/include>
)

target_link_libraries(MyLibraryA
PUBLIC
SomeLibraries::SomeLibraryA
PRIVATE
SomeLibraries::SomeLibraryB
)

install(TARGETS MyLibraryA
EXPORT  MyLibraries-export
LIBRARY DESTINATION lib
ARCHIVE DESTINATION lib
)
```

```
install(EXPORT MyLibraries-export
FILE
MyLibrariesTargets.cmake
NAMESPACE
MyLibraries::
DESTINATION
lib/cmake/MyLibraries}
)

install(FILES
${CMAKE_CURRENT_SOURCE_DIR}/include/MyLibraries/my_library_a.hpp
DESTINATION "include/MyLibraries")

install(FILES
${CMAKE_CURRENT_SOURCE_DIR}/cmake/MyLibrariesConfig.cmake
DESTINATION "lib/cmake/MyLibraries" )
```

It uses *find_package(SomeLibraries REQUIRED)* to import SomeLibraries package.

```
find_package(SomeLibraries REQUIRED)
```

This triggers CMake to search for SomeLibrariesConfig.cmake file. This eventually imports the SomeLibraryA(B) targets in *SomeLibraries* package with all their properties into MyLibraries CMake project.

Then, here is the interesting part, using *target_link_libraries*

```
target_link_libraries(MyLibraryA
PUBLIC
SomeLibraries::SomeLibraryA
PRIVATE
SomeLibraries::SomeLibraryB
)
```

The basic concepts of “target and property” and the analogy of “object and member function” are equally applied to *target_link_libraries* as they were applied to *target_include_directories* we discussed before.

As you might have noticed, *SomeLibraries* is a namespace we specified with *install* function when we were generating SomeLibrariesTargets.cmake file. By specifying SomeLibraries::SomeLibraryA(B) as dependencies, MyLibraryA target inherits all PUBLIC properties of the SomeLibraryA(B) targets. Also, by adding keyword PUBLIC, MyLibraryA will pass the inherited properties from SomeLibraryA to any other targets that link MyLibraryA as dependency with *target_link_libraries* in their CMakeLists.txt. On the other hand, by adding PRIVATE keyword, it is specifying that the MyLibraryA's dependency to SomeLibraryB as "private", and the inherited properties from SomeLibraryB will **not** be passed to other targets that link MyLibraryA as dependency. This is the basic of how the transitive dependency in Modern CMake works.

Lastly, in order to use the MyLibraries package from MyApplication package, we need to install the MyLibraries package. Basically, we need to do the same things as what we did to install SomeLibraries package.

We also need to write MyLibrariesConfig.cmake to install MyLibraries package like when we did for SomeLibraries package. One thing to note is, inside of MyLibrariesConfig.cmake, we need to specify dependency to SomeLibraries with *find_dependency(SomeLibraries REQUIRED)* like below,

```
include(CMakeFindDependencyMacro)

find_dependency(SomeLibraries REQUIRED)

include("${CMAKE_CURRENT_LIST_DIR}/MyLibrariesTargets.cmake")
```

Then, again we just need to execute command below,

```
cd my_libraries
mkdir build
cd build
cmake ..
make install
```

Import the MyLibraryA target from MyLibraries package into

MyApplication project

Finally, we are ready to build MyApplication executable package. The CMakeLists.txt for the MyApplication executable package looks like below,

```
## CMakeLists.txt for the MyApplication executable package ##

cmake_minimum_required(VERSION 3.5)
project(MyApplication VERSION 1.0 LANGUAGES CXX)

find_package(MyLibraries REQUIRED)

add_executable(MyApplication src/main.cpp)

target_link_libraries(MyApplication PRIVATE MyLibraries::MyLibraryA)
```

Basically, as you can see, nothing new. Since the MyLibraryA target passes PUBLIC property of SomeLibraryA target(in this case, the include path where the public header is installed), so some_library_a.hpp can be found during the build of the MyApplication target.

Summary

In this tutorial, I explained the following aspects of Modern CMake.

- The Concept of treating "build target" (i.e. executable or library you are building with CMake) as "object" in Object-oriented programming in order to better structure your projects and the idea of the inheritance of the build dependencies transitively (Transitive dependency).
- How to export/install CMake package and import it from another CMake project.

Simple Modern CMake Tutorial Part 2

[1]: https://cmake.org/

[2]: https://scons.org/

[3]: https://bazel.build/

[4]: C++Now 2017: Daniel Pfeifer "Effective CMake" https://www.youtube.com/watch?v=bsXLMQ6WgIk

[5]: https://cliutils.gitlab.io/modern-cmake/

Sign up for Top Stories

By Level Up Coding

Programming Software Development Software Engineering Cmake Cplusplus