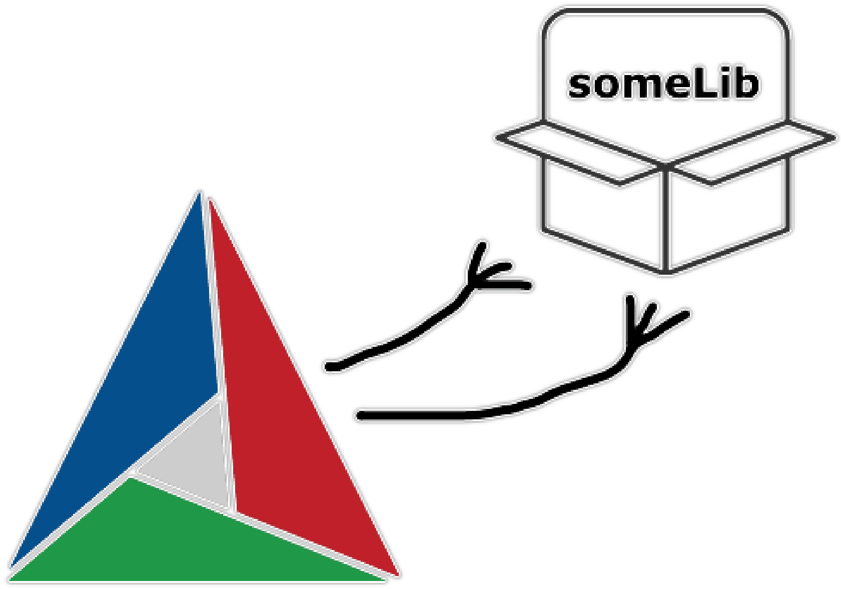
Creating a C++ library with CMake

2021-03-08 20:41:56 +0100
2024-11-15 10:53:36 +0100
26 min read

All of the sudden I found myself in a situation that I have been successfully avoiding so far - I needed to make a C++ library with CMake.



Search in titles and summ

Sponsor

Feeds

Social networks

To clarify, this will be about so-called normal kind of library. And right away, the picture above shows a bad example of naming a library, because there should be no `lib` neither in the beginning nor in the end of the library name.

Zuck: *Yeah so if you ever need info about anyone at Harvard*
Zuck: *Just ask*
Zuck: *I have over 4,000 emails, pictures, addresses, SNS*
smb: *What? How'd you manage that one?*
Zuck: *People just submitted it.*
Zuck: *I don't know why.*
Zuck: *They "trust me"*
Zuck: *Dumb fucks*

Tags

The library

Folder structure and sources

For the sake of focusing on CMake side of things, the library itself is very trivial:

```
├── CMakeLists.txt
├── include
│   └── some.h
└── src
    ├── some.cpp
    └── things.h
```

This particular folder structure is not enforced, but I've seen it being used around and I think it serves nicely for the purpose of keeping library files organized. Following this structure, you put internal library sources and headers to `src` folder, and public headers go to `include` folder. This article, however, lists a few disadvantages of such approach.

#privacy (2)
#steam-deck (2)
#tvos (2) #unsolved (2)
#books (1) #ffmpeg (1)
#home-automation (1)
#ldap (1) #music (1)
#science (1) #vr (1)

Public headers is something other projects will use to interface with your library. That is how they will know what functions are available in it and what is their signature (*parameters names and types*). So, `include/some.h` is a public header, and here are its contents:

```
#ifndef SOME_H
#define SOME_H

namespace sm
{
    namespace lbr
    {
        void printSomething();
    }
}

#endif // SOME_H
```

So our library has just one function. Its definition is in `src/some.cpp`:

```
#include <iostream>
```

```
#include "things.h"

namespace sm
{
    namespace lbr
    {
        void printSomething()
        {
            std::cout << "olol
        }
    }
}
```

As you can see, this function just prints a message to standard output. The `someString` variable comes from internal header `src/things.h`:

```
#include <string>

const std::string someString =
```

CMakeLists

Making a library with CMake is not that different from making an application - instead of add_executable you call add_library. But doing just that would be too easy, wouldn't it.

Here are some of the things you need to take care of:

- what artifacts should the library produce at install step
- where install artifacts should be placed
- how other applications can find the library
  - when they are using it pre-built as an external dependency
  - when its sources are nested in their source tree
- will it be static or shared library
  - will you need to have it as DLL on Windows

Everything from this list is handled by CMake. So let's gradually create a `CMakeLists.txt` for the library project.

Top-level and nested projects

In CMake projects there is a variable called CMAKE_PROJECT_NAME. It stores the top-level project name that you set with project command. This variable persists across all the nested projects, and so calling `project` command from nested projects will not change `CMAKE_PROJECT_NAME`, but will set another variable called PROJECT_NAME.

Knowing that, here's how you can check if you are in the top-level project or not:

```
project("SomeLibrary"
    # ...
)

if (NOT CMAKE_PROJECT_NAME STR
    message(STATUS "This proje
else()
    message(STATUS "This proje
endif()
```

Later CMake versions added a PROJECT_IS_TOP_LEVEL variable, which might be more convenient.

Why even bother with this? Because later we will be setting certain properties for the target (*our library*). And I saw in lots of places how people copy-paste project name value to every command, which I believe is just a bad idea - it is much better to use already defined `PROJECT_NAME` variable instead, innit.

Target

Here goes the library target and its sources:

```
# here you can see how instead
# we can just use the PROJECT_
add_library(${PROJECT_NAME} ST

# no need to add headers here,
target_sources(${PROJECT_NAME}
    PRIVATE
        src/some.cpp
)
```

In this case the library is declared as `STATIC`, but actually it is not a good idea to hardcode libraries type like that in their project files, because CMake has a global flag for this exact purpose - BUILD_SHARED_LIBS - and in general it's better to rely on that flag instead of setting libraries type inline.

There will be also a section about shared libraries later, but in this particular case hardcoding the library type to `STATIC` somewhat makes sense, because I do not export its symbols for making a DLL, so on Windows it wouldn't build a proper library as `SHARED`.

Name without lib prefix/postfix

Like I said in the beginning, the `someLib` name on the picture is a bad example of naming a library, as there should be no `lib` anywhere in the name (`/(^[lL][iI][bB])|([lL][iI][bB]$)/g`), so `libSome` would also be a bad name. The `SomeLibrary` name, on the other hand, is okay-ish, because it neither starts with `lib` nor ends with `lib` (*it only has `lib` in the middle*).

The reason why one should not have `lib` prefix/postfix in one's library name is because it will be set in an appropriate manner (*depending on the target platform*) by CMake automatically. So, if you have your library named `someLib`, then on Unix platforms it will become `libsomeLib.a` (*and `someLib.lib` on Windows*), which already doesn't look great, and if you have it named `libSome`, then the resulting binary will be `liblibSome.a`, which is even worse.

It is of course possible to manipulate the output names explicitly, but there is already enough things to worry about, so I would just leave this to CMake and rely on the default behaviour by simply not having `lib` in the library name.

Include directories

Setting include directories correctly with target_include_directories is very important:

```
target_include_directories(${P
    PRIVATE
        # where the library it
        ${CMAKE_CURRENT_SOURCE
    PUBLIC
        # where top-level proj
        $<BUILD_INTERFACE:${CM
        # where external proje
        $<INSTALL_INTERFACE:${
)
```

Paths in `PRIVATE` section are used by the library to find its own internal headers. So if you would place `things.h` to `./src/hdrs/things.h`, then you will need to set this path to

`${CMAKE_CURRENT_SOURCE_DIR}/src/hdrs`.

Paths in `PUBLIC` section are used by projects that link to this library. That’s where they will look for its public headers:

- `BUILD_INTERFACE` path is meant for projects that will build the library from their source tree, and here you need to add `include`, because that’s where public headers are in the library’s source folder
- `INSTALL_INTERFACE` is meant for external projects, and here you don’t need to add `include`, because CMake config will do that for you

Install instructions

We need to declare what artifacts should be put to installation directory after building the library. You also need to specify the path of installation directory (*where you would like it to be*).

Certainly, just building the library is already enough to be able to link to it, but we want to do it in the most comfortable way: not by providing paths to its binaries and headers from both build and sources directories, but by installing just the artifacts we need and using find_package command.

With `find_package` you let CMake to worry about finding the library, its public headers and configuring all that. Here's a more detailed documentation about how `find_package` works, and here's how you can create a CMake config of your own.

Installation path

First thing to think about it is the installation path. If you will not set it during configuration step via CMAKE_INSTALL_PREFIX (`-DCMAKE_INSTALL_PREFIX="/some/path"`), then CMake will set it to some system libraries path, which you might not want to use, especially if you are building your library for distribution.

Here’s how you can overwrite default installation path to install the artifacts into `install` folder in your source tree:

```
# note that it is not CMAKE_IN
if(DEFINED CMAKE_INSTALL_PREFI
    message(
        STATUS
        "CMAKE_INSTALL_PREFIX
        "Default value: ${CMAK
        "Will set it to ${CMAK
    )
    set(CMAKE_INSTALL_PREFIX
        "${CMAKE_SOURCE_DIR}/i
        CACHE PATH "Where the
    )
else()
    message(
        STATUS
        "CMAKE_INSTALL_PREFIX
        "Current value: ${CMAK
    )
endif()
```

Public headers

Next thing you need to do is to declare PUBLIC_HEADER property:

```
# without it public headers wo
set(public_headers
    include/some.h
)
# note that ${public_headers}
set_target_properties(${PROJEC
```

Note, however, that this doesn’t preserve the folder structure, and so for more complex projects that won’t give the desired result. For instance, here are the public headers of the glad library:

```
├── include
│   ├── KHR
│   │   └── khrplatform.h
│   └── glad
│       └── glad.h
└── src
    └── glad.c
```

Trying to install it, you'll get the following result in the installation path:

```
├── cmake
│     ├── gladConfig-release.c
│     └── gladConfig.cmake
├── include
│     └── glad
│           ├── glad.h
│           └── khrplatform.h
└── lib
    └── libglad.a
```

which won't work for `khrplatform.h` header, as it is expected to be included like this:

```
#include <KHR/khrplatform.h>
```

In order to preserve the folder structure, you can use the following trickery:

```
# for CMAKE_INSTALL_INCLUDEDIR
include(GNUInstallDirs)
```

```
# the variant with PUBLIC_HEAD
#set_target_properties(${PROJE
# so instead we iterate throug
foreach(header ${public_header
    file(RELATIVE_PATH header_
    get_filename_component(hea
    install(
        FILES ${header}
        DESTINATION "${CMAKE_I
    )
endforeach()
```

Debug suffix

It might be a good idea to add `d` suffix to debug binaries - that way you'll get `libSomeLibraryd.a` with Debug configuration and `libSomeLibrary.a` with Release. To do that you need to set the DEBUG_POSTFIX property:

```
set_target_properties(${PROJEC
```

Or you can set it globally for the entire project on configuration with `-DCMAKE_DEBUG_POSTFIX="d"`.

Destinations

Here come the actual installation instructions:

```
# if you haven't included it a
# definitions of CMAKE_INSTALL
include(GNUInstallDirs)

# install the target and creat
install(TARGETS ${PROJECT_NAME
    EXPORT "${PROJECT_NAME}Tar
    # these get default values
    #RUNTIME DESTINATION ${CMA
    #LIBRARY DESTINATION ${CMA
    #ARCHIVE DESTINATION ${CMA
    # except for public header
    PUBLIC_HEADER DESTINATION 
    INCLUDES DESTINATION ${CMA
)
```

Important to note here that `INCLUDES` is not part of the `RUNTIME`/`LIBRARY`/`ARCHIVE`/`PUBLIC_HEADER` group. See for yourself in the install() signature:

```
install(
    TARGETS targets... [EXPORT
    [RUNTIME_DEPENDENCIES args
    [
        [
            ARCHIVE|LIBRARY|RU
        ]
        [DESTINATION <dir>]
        [PERMISSIONS permissio
        [CONFIGURATIONS [Debug
        [COMPONENT <component>
        [NAMELINK_COMPONENT <c
        [OPTIONAL] [EXCLUDE_FR
        [NAMELINK_ONLY|NAMELIN
    ] [...]
    [INCLUDES DESTINATION [<di
)
```

If you won’t have `INCLUDES` in the `install()`, then `SomeLibraryTargets.cmake` won’t have these lines:

```
set_target_properties(some::So
  INTERFACE_INCLUDE_DIRECTORIE
)
```

and when you'll try to use this (*installed*) library in an external project, it will configure fine, but it will fail to build:

```
$ cmake --build .
[ 50%] Building CXX object CMa
/path/to/cmake-library-example
#include <SomeLibrary/some.h>
         ^~~~~~~~~~~~~~~~~~~~
1 error generated.
make[2]: *** [CMakeFiles/anoth
make[1]: *** [CMakeFiles/anoth
make: *** [all] Error 2
```

Configs

Create `Config.cmake.in` file:

```
@PACKAGE_INIT@

include("${CMAKE_CURRENT_LIST_

check_required_components(@PRO
```

CMake documentation doesn't mention it in a clear way, but you can still use the `PROJECT_NAME` variable here too - just wrap it in `@@`.

And then in `CMakeLists.txt`:

```
# generate and install export
install(EXPORT "${PROJECT_NAME
    FILE "${PROJECT_NAME}Targe
    NAMESPACE ${namespace}::
    DESTINATION cmake
)

include(CMakePackageConfigHelp

# generate the version file fo
write_basic_package_version_fi
    "${CMAKE_CURRENT_BINARY_DI
    VERSION "${version}"
    COMPATIBILITY AnyNewerVers
)
# create config file
configure_package_config_file(
    "${CMAKE_CURRENT_BINARY_DI
    INSTALL_DESTINATION cmake
)
# install config files
install(FILES
    "${CMAKE_CURRENT_BINARY_DI
    "${CMAKE_CURRENT_BINARY_DI
    DESTINATION cmake
)
# generate the export targets
# (can't say what this one is
# so I stopped adding it to pr
# export(EXPORT "${PROJECT_NAM
#     FILE "${CMAKE_CURRENT_BI
#     NAMESPACE ${namespace}::
#)
```

The `write_basic_package_version_file()` function from above will create `SomeLibraryConfigVersion.cmake` file in the `install` folder. Having it, if you now try to find your package in external project (`cmake-library-example/external-project/CMakeLists.txt`) like this:

```
find_package(SomeLibrary 0.9.2
```

then you will get the following error on configuration:

```
CMake Error at CMakeLists.txt:
  Could not find a configurati
  compatible with requested ve

  The following configuration

    /Users/YOURNAME/code/cpp/c
```

Finally, the `NAMESPACE` property is exactly what is looks like - a namespace of your library. I reckon, that will help to avoid names collision and also allow to group related stuff in one namespace, similar to how Qt does it:

```
target_link_libraries(hellowor
```

Building and installing

Go to library source tree root and run the usual:

```
$ mkdir build && cd $_

$ cmake ..
-- The C compiler identificati
-- The CXX compiler identifica
-- Detecting C compiler ABI in
-- Detecting C compiler ABI in
-- Check for working C compile
-- Detecting C compile feature
-- Detecting C compile feature
-- Detecting CXX compiler ABI
-- Detecting CXX compiler ABI
-- Check for working CXX compi
-- Detecting CXX compile featu
-- Detecting CXX compile featu
-- This project is a top-level
-- CMAKE_INSTALL_PREFIX is not
Default value: /usr/local
Will set it to /Users/YOURNAME
-- Configuring done
-- Generating done
-- Build files have been writt

$ cmake --build . --target ins
Scanning dependencies of targe
[ 50%] Building CXX object CMa
```

```
[100%] Linking CXX static libr
[100%] Built target SomeLibrar
Install the project...
-- Install configuration: ""
-- Installing: /Users/YOURNAME
-- Installing: /Users/YOURNAME
-- Installing: /Users/YOURNAME
-- Installing: /Users/YOURNAME

$ tree ../install/
├── cmake
│   ├── SomeLibraryConfig-noco
│   └── SomeLibraryConfig.cmak
├── include
│   └── SomeLibrary
│       └── some.h
└── lib
    └── libSomeLibrary.a
```

Note that `SomeLibraryConfig-noconfig.cmake` has this weird `noconfig` suffix. This is because we ran configuration without specifying the build type - better to explicitly set it then, both Debug and Release:

```
$ rm -r ../install/* && rm -r

$ cmake --build . --target ins
Scanning dependencies of targe
[ 50%] Building CXX object CMa
[100%] Linking CXX static libr
[100%] Built target SomeLibrar
Install the project...
-- Install configuration: "Deb
-- Installing: /Users/YOURNAME
-- Installing: /Users/YOURNAME
-- Installing: /Users/YOURNAME
-- Installing: /Users/YOURNAME

$ rm -r ./* && cmake -DCMAKE_B

$ cmake --build . --target ins
Scanning dependencies of targe
[ 50%] Building CXX object CMa
[100%] Linking CXX static libr
[100%] Built target SomeLibrar
Install the project...
-- Install configuration: "Rel
-- Installing: /Users/YOURNAME
-- Up-to-date: /Users/YOURNAME
-- Installing: /Users/YOURNAME
-- Installing: /Users/YOURNAME

$ tree ../install/
├── cmake
│   ├── SomeLibraryConfig-debu
│   ├── SomeLibraryConfig-rele
│   └── SomeLibraryConfig.cmak
├── include
│   └── SomeLibrary
│       └── some.h
└── lib
    ├── libSomeLibrary.a
    └── libSomeLibraryd.a
```

So there you have it! The library has been successfully built and nicely installed, so now you can just pack the `install` folder contents (*you might want to use CPack for that*) and distribute it to your users.

CMake generators

While we are here, you should probably know about such thing as CMake generators. Trying to put it simply, I would say that generator is what transforms CMake project files/scripts into specific build instructions for a particular build tool.

As you hopefully are aware, there are several build tools available. We have GNU make, Microsoft NMAKE, Ninja, xcodebuild and so on. And depending on which one you would like to use, you need to choose a particular CMake generator for that.

To list all the available generators for the current platform you can run CMake with an “empty” `-G` argument, for example here’s what I have on Mac OS:

```
$ cmake -G
CMake Error: No generator spec

Generators
* Unix Makefiles
  Ninja
  Ninja Multi-Config
  Watcom WMake
  Xcode
  CodeBlocks - Ninja

  CodeBlocks - Unix Makefiles

  CodeLite - Ninja

  CodeLite - Unix Makefiles

  Eclipse CDT4 - Ninja

  Eclipse CDT4 - Unix Makefile

  Kate - Ninja
  Kate - Ninja Multi-Config
  Kate - Unix Makefiles
  Sublime Text 2 - Ninja

  Sublime Text 2 - Unix Makefi
```

All the builds in this article are done with Unix Makefiles generator, because that is the default one on Mac OS, and back then I simply didn’t know anything about CMake generators, so I was just obliviously using what was set by default.

Now, when I know a little bit more, I would recommend to use Ninja any day of the week (*for it's the fastest of them all*). But if you need to generate a project for your IDE, then you'd need to use an IDE-specific generator, such as Xcode or Visual Studio 17 2022 (*there are several versions of that one*).

One other thing to mention here is that specifying Release or Debug build configuration is different between generators. In particular, for `Unix Makefiles` and `Ninja` you would do it with `CMAKE_BUILD_TYPE` on configuration:

```
$ cd /path/to/project
$ mkdir build && cd $_
$ cmake -G Ninja -DCMAKE_BUILD
$ cmake --build . --target ins
```

while for `Xcode` and `Visual Studio 17 2022` that would be via `config` argument on build:

```
$ cd /path/to/project
$ mkdir build && cd $_
$ cmake -G "Visual Studio 17 2
$ cmake --build . --target ins
```

Linking to the library

Now let’s see how your users or yourself can link to the library.

From external project

Let’s take a simple project:

```
$ cd ~/code/cpp/external-proje

$ tree .
├── CMakeLists.txt
└── main.cpp
```

The source:

```
#include <iostream>
// note that you need to prepe
// because that is how it is i
// install/include/SomeLibrary
#include <SomeLibrary/some.h>

int main(int argc, char *argv[
{
    std::cout << "base applica
    // here we call a function
    sm::lbr::printSomething();
}
```

The project file:

```
project("another-application"

# provide the path to library'
#
# for simplicity it is set rig
# to hardcode these paths in p
# for example: -DCMAKE_PREFIX_
list(APPEND CMAKE_PREFIX_PATH 
# the rest will be taken care 
find_package(SomeLibrary CONFI


# it is an application
add_executable(${PROJECT_NAME}

target_sources(${PROJECT_NAME}
    PRIVATE
        main.cpp
)

# linking to the library, here
target_link_libraries(${PROJEC
```

The `PRIVATE` keyword means that the library will be used by this project, but it will not be available to other projects via this project's interface. For example, if this project is in turn a library itself, then yet another external/parent project linking to it won't be able to use `printSomething()` function from the underlying library (*this is not exactly true, here are some more details about that*).

Let's now try to build and run the application:

```
$ mkdir build && cd $_

$ cmake -DCMAKE_BUILD_TYPE=Rel

$ cmake --build .
Scanning dependencies of targe
[ 50%] Building CXX object CMa
[100%] Linking CXX executable 
[100%] Built target another-ap

$ ./another-application
base application message
ololo, some string
```

It works!

No need to set include_directories and use magic variables

You might have seen in other projects that they also set `include_directories` for external libraries, and probably you are now wondering why I didn't do it here, and how then it all works.

Indeed, if we take a look at one of such projects, for example SDL2 installed via Homebrew on Mac OS:

```
$ brew info sdl2
sdl2: stable 2.0.16 (bottled),
Low-level access to audio, key
https://www.libsdl.org/
/usr/local/Cellar/sdl2/2.0.16

$ tree /usr/local/Cellar/sdl2/
/usr/local/Cellar/sdl2/2.0.16
├── ...
├── bin
│  └── sdl2-config
├── include
│  └── SDL2
│      ├── SDL.h
│      └── ...
├── lib
│  ├── cmake
│  │  └── SDL2
│  │      ├── sdl2-config-versio
│  │      └── sdl2-config.cmake
│  ├── libSDL2-2.0.0.dylib
│  ├── libSDL2.a
│  ├── libSDL2.dylib -> libSDL2
│  ├── libSDL2_test.a
│  ├── libSDL2main.a
│  └── pkgconfig
│      └── sdl2.pc
└── share
    └── aclocal
        └── sdl2.m4
```

Thanks to Homebrew, it is discoverable even without adding its path to `CMAKE_PREFIX_PATH`:

```
# and once again, it's better
#list(APPEND CMAKE_PREFIX_PATH
find_package(SDL2 REQUIRED)
if(${SDL2_FOUND})
    message(STATUS "Found SDL:
endif()

# ...

target_link_libraries(${CMAKE_
    PRIVATE
        ${SDL2_LIBRARIES}
)
```

But trying to build your application you'll get this error:

```
fatal error: 'SDL.h' file not
#include <SDL.h>
         ^~~~~~~
1 error generated.
```

To fix that you'll also need to add the following to the project file:

```
include_directories(
    ${SDL2_INCLUDE_DIRS}
)
```

That is because SDL2 CMake package was created in a rather obsolete manner, so you have to manually set `include_directories` and also to use these magic variables such as `_INCLUDE_DIRS` and `_LIBRARIES`.

Our library package is written in a more modern way, so including directories is already taken care of, and also there is no need to use any magic variables, just the package name. Nice, innit.

From internal top-level project

But what if we have our library as a part of some other top-level project, so the library lives in its source tree? Do we still need to build it first and add it to the main project via `find_package`? Not exactly - now there is no need to "find" it: the library will be built together with the parent project and then linked to.

Adding nested library to the main project

Here's the full project structure:

```
$ cd /Users/YOURNAME/code/cpp/

$ tree .
├── CMakeLists.txt
├── libraries
│   ├── CMakeLists.txt
│   └── SomeLibrary
│       ├── CMakeLists.txt
│         ├── include
│         │    └── some.h
│         └── src
│              ├── some.cpp
│              └── things.h
└── main.cpp
```

The main `CMakeLists.txt`:

```
project("some-application" VER

add_subdirectory(libraries)

add_executable(${PROJECT_NAME}

target_sources(${PROJECT_NAME}
    PRIVATE
        main.cpp
)

target_link_libraries(${PROJEC
```

Yes, it is all the same as with external project - we just need to link to the library. No crazy relative paths, just the very same `target_link_libraries`.

But this time we don't need `find_package` and also we don't need to provide the namespace. That last part I don't entirely understand, but perhaps that is because everything in the project is supposed to be within the same namespace already?

Following add_subdirectory statement, we get to `libraries` folder, which also has a `CMakeLists.txt`:

```
add_subdirectory(SomeLibrary)
```

And there we get to our library project.

About include paths

Let's start with `main.cpp`:

```
#include <iostream>
#include <some.h>

int main(int argc, char *argv[
{
    std::cout << "base applica
    sm::lbr::printSomething();
}
```

While the code here is the same as the one from external project, there is one notable difference - `some.h` is not prepended with `SomeLibrary/` in the `#include` statement.

Why is that, why is it different from the way in was included in external project? Well, it is because that is how this header is placed in the library’s source `include` folder - there is no `SomeLibrary` folder nested there. But when you install the library, then there is `SomeLibrary` folder inside `include` in the installation folder, so in that case you need to add `SomeLibrary/` to `#include` statement.

I wasn’t sure if that is really how things are, so I went and asked about this on StackOverflow, hoping that there is perhaps some way to handle this in a unified way, but the answer was:

> *You think too "magically" about what CMake can. CMake just calls a compiler/linker with proper parameters. A compiler requires for #include that a header file will be in the SomeLibrary directory. CMake cannot overcome this requirement.*

So if you would like to unify the way you include the library’s public headers both in external and internal projects, then you’ll need to create `SomeLibrary` folder inside library’s source `include` folder (*yeah, to get the ugly `SomeLibrary/include/SomeLibrary` path*) and adjust the library’s `CMakeLists.txt` accordingly.

Building

Main project

Do the usual in the project source tree root:

```
$ mkdir build && cd $_
$ cmake -DCMAKE_BUILD_TYPE=Deb

$ cmake --build .
Scanning dependencies of targe
[ 25%] Building CXX object lib
[ 50%] Linking CXX static libr
[ 50%] Built target SomeLibrar
Scanning dependencies of targe
[ 75%] Building CXX object CMa
[100%] Linking CXX executable
[100%] Built target some-appli

$ ./some-application
base application message
ololo, some string
```

Eeeeeeeasy!

Library as a target

We can also build and install just the library, without building the entire project. Aside from just going to the library folder and running CMake from there, you can actually do it from the project root - by setting `--target` option on build:

```
$ rm -r ./* && cmake -DCMAKE_B

$ cmake --build . --target Som
Scanning dependencies of targe
[ 50%] Building CXX object lib
[100%] Linking CXX static libr
[100%] Built target SomeLibrar

$ cmake --install ./libraries/
-- Install configuration: "Deb
-- Installing: /Users/YOURNAME
-- Installing: /Users/YOURNAME
-- Installing: /Users/YOURNAME
-- Installing: /Users/YOURNAME
```

Here you can also see how you can install a single target with `--install` option - by pointing it to the target folder inside `build` folder. Also note that install folder is now on the project’s source tree root level, not in the library’s nested source folder.

STATIC vs SHARED

Hopefully, you already know the difference between static and shared libraries. If not, then, to put it simple, static libraries are “bundled” into your binaries, and shared libraries are separate files which need to be discoverable by your binaries in order for the latter to work.

A little practical example: let’s build our library as static, link to it from external project, then build it as shared and link to that one.

To make our library shared, we need to replace `STATIC` with `SHARED` in `add_library` statement in the library’s `CMakeLists.txt`. And once again, like I already said, the library type should not be hardcoded like that, as it would be better to have `add_library()` without type and instead set `-DBUILD_SHARED_LIBS=1` on project configuration.

Anyway, first thing that will be different about resulting executable (`another-application`) is its size:

- statically linked with SomeLibrary: 51 920 bytes
- dynamically linked with SomeLibrary: 51 616 bytes

Not a very noticeable difference (*remember that our library just prints a line of text*), but it is there: statically linked executable is bigger, because the library is “bundled” into it.

Secondly, if you now rename or delete `libSomeLibrary.dylib` (*`libSomeLibrary.so`*, *`SomeLibrary.dll`*), then trying to run this dynamically linked application you’ll get an error like this on Mac OS:

```
$ ./another-application
dyld: Library not loaded: @rpa
  Referenced from: /Users/YOUR
  Reason: image not found
Abort trap: 6
```

or like this on Linux:

```
$ ./another-application
./another-application: error w
```

So in case of a shared library on Mac OS or Linux it has to either stay available in its installation path, or be placed into the one of the system libraries paths. Be aware that simply copying it to the same folder with executable won't work, unless you set the `LD_LIBRARY_PATH` variable on Linux (*or `DYLD_FALLBACK_LIBRARY_PATH`/`DYLD_LIBRARY_PATH` on Mac OS*) before running the application:

```
$ LD_LIBRARY_PATH="." ./anothe
```

And on Windows it will fail even if you haven't touched the library in its install folder, however you can just copy it to the same folder where executable is, but more on that below.

SHARED DLL on Windows

Shared libraries on Windows are a special thing. There it is not enough just to replace `STATIC` with `SHARED` in `add_library` statement (*or set* `-DBUILD_SHARED_LIBS=1`).

If you build and install it having done nothing else, then you will get this error trying to configure a project that needs to link to it:

```
CMake Error at C:/code/cmake-l
  The imported target "some::S

    "C:/code/cmake-library-ex

  but this file does not exist

  * The file was deleted, rena

  * An install or uninstall pr

  * The installation package w

    "C:/code/cmake-library-ex
  but not all the files it ref

Call Stack (most recent call f
  CMakeLists.txt:9 (find_packa
```

And indeed, there is no `SomeLibrary.lib` in `install/lib/`, only `SomeLibrary.dll` in `install/bin/`. That is because a DLL on Windows needs an explicit listing of all the symbols that it will export, and apparently this is what `SomeLibrary.lib` is supposed to be.

As I understood, in order to produce it, in past it was required to add `__declspec` compiler directives to every public single class or function declaration in your library sources, which is quite a bummer, especially if you have a lot of those. Here's one example of how this is done.

Fortunately, starting with CMake 3.4, this is no longer required. Instead you can just set the

CMAKE_WINDOWS_EXPORT_ALL_SYN
option when configuring the library:

```
$ cmake -DCMAKE_WINDOWS_EXPORT
```

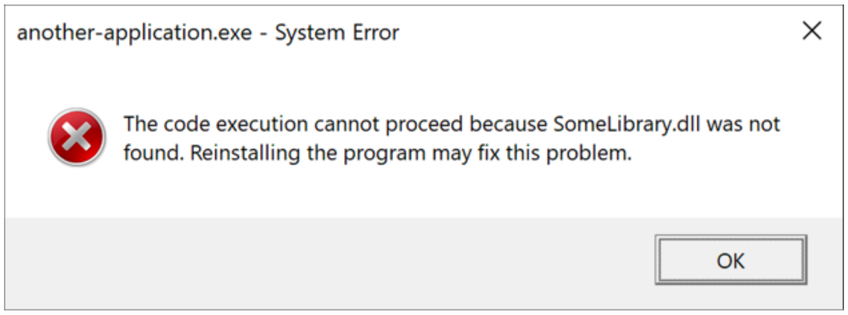
However, I saw somewhere that it is not recommended to export all the symbols like that. I didn't quite get why, but probably there is a reason, so just keep that in mind.

Either way, now you can build and install the library as usual. Note that if you are using Visual Studio generator, then `-DCMAKE_BUILD_TYPE` won't work, because with this generator the configuration type is specified on build:

```
$ cmake --build . --target ins
```

After that configuring and building the external application will also succeed, because now it will get that missing `SomeLibrary.lib`.

However, unlike Mac OS and Linux, trying to run the resulting application will fail even if you haven't touched the `SomeLibrary.dll` in the install folder:


another-application.exe - System Error

The code execution cannot proceed because SomeLibrary.dll was not found. Reinstalling the program may fix this problem.

or, if running from Git BASH:

```
$ ./Release/another-applicatio
C:/code/cmake-library-example/
```

That is because on Windows you need to explicitly put the DLL either to the same folder where the executable is or somewhere in PATH.

Final words and repository

Later I will probably update the article, because, like I said, here I touched only "normal" libraries, but there are also other kinds. Plus, one can go further than just making a CMake config and create a proper package. So there are quite a few things left to talk about.

You might have noticed that most of the stuff I was doing on Mac OS, but actually everything (*library and sample applications*) builds and works just fine also on Linux and Windows.

Full source code of the library, its parent project and external project are available in this repository.

Updates

2022-07-05 | Updated repository

Be aware that I’ve been updating the referenced repository as I was discovering new things, but not everything has been synced-up back to the article, and so by now the article contains a somewhat simpler CMake code, while repository has a bit more advanced things (*such as installation instructions being a separate CMake module*).

You might also want to take a look at the article about using CPack and its example project (*you can ignore the packing parts*), as in particular it has a better organized installation and demonstrates re-using “shared” CMake modules.

2022-09-17 | Dynamic libraries and paths

A useful addition about dynamic libraries and paths.

2023-07-22 | About target_link_libraries() scopes

At some point later I also wrote about `target_link_libraries()` scopes in particular. There you will find yet another repository, which I now consider to be my best example of creating C++ libraries with CMake (*until I learn some more CMake and make an even better one*). Among other improvements, it includes exporting symbols for making a `.lib` file for DLLs on Windows.

cmake cpp

Your comment here

Sign In

Styling with **Markdown** is supported

Sort by **Recently updated**