

# Cmake

Mikael J. Gonsalves

July 21, 2023

# Contents

<b>1 Cmake</b>	<b>5</b>
1.1 Overview . . . . .	5
1.2 Project Structure . . . . .	5
1.2.1 App Directory . . . . .	6
1.2.2 Source Directory - src . . . . .	6
1.2.3 CMake Default Paths . . . . .	6
1.2.4 Template for Projects is Intermediate project . . . . .	7
1.3 Variables . . . . .	7
1.3.1 Standard for cpp . . . . .	7
1.3.2 Standard Required . . . . .	7
1.3.3 Compiler Extensions . . . . .	7
1.3.4 If Statement . . . . .	8
1.3.5 Options . . . . .	8
1.4 Makefile Shell Scripting . . . . .	9
1.5 Project Software Toolkit . . . . .	9
1.6 Installation Commands . . . . .	9
1.7 Command Line Options . . . . .	10
1.7.1 Generating a Project . . . . .	10
1.7.2 Generator for GCC and Clang . . . . .	10
1.7.3 Generator for MSVC . . . . .	10
1.7.4 Specify the Build Type . . . . .	10
1.7.5 Passing Options . . . . .	10
1.7.6 Specify the Build Target (Option 1) . . . . .	11
1.7.7 Specify the Build Target (Option 2) . . . . .	11
1.8 Run the Executable . . . . .	11
1.9 Configuration - Precompiling Information . . . . .	11
1.9.1 Project Version Number . . . . .	12
1.9.2 Sementic Versioning . . . . .	12
1.10 Sources and Headers . . . . .	13
1.10.1 Executables . . . . .	13
1.11 Using External Libraries . . . . .	13
1.11.1 Git Submodule - nlohman json . . . . .	13
1.11.2 Custom Cmake Functions . . . . .	14
1.11.3 Log example . . . . .	14
1.11.4 Call Your Own Functions . . . . .	15
1.11.5 Fetch Content - External Libraries . . . . .	15
1.11.6 FMT . . . . .	16
1.11.7 spdlog . . . . .	16
1.11.8 Cxxopts . . . . .	17

1.11.9	Catch2 . . . . .	17
1.11.10	Include All Libraries in root CMakeListstxt . . . . .	17
1.11.11	Include all Libraries in App Main.cpp . . . . .	19
1.11.12	Git Submodules vs Fetch Content . . . . .	19
<b>2</b>	<b>CMake Package Managers</b>	<b>20</b>
2.1	CPM - Cmake Package Manager . . . . .	20
2.1.1	CPM configuration in root CMakeLists.txt . . . . .	20
2.2	Conan . . . . .	21
2.2.1	Conan Installation . . . . .	21
2.2.2	Conan Configuration in root CMakeLists.txt . . . . .	22
2.2.3	Conanfile.py . . . . .	22
2.2.4	Conan Debug Build . . . . .	23
2.2.5	Conan Generated File . . . . .	24
2.3	VCPKG . . . . .	24
2.3.1	VCPKG Installation . . . . .	24
2.3.2	VCPKG.json file in project root . . . . .	25
2.3.3	VCPKG configuration in root CMakeLists.txt . . . . .	26
2.4	Package Managers Summary . . . . .	26
<b>3</b>	<b>CMake Tooling</b>	<b>28</b>
3.1	Dependency Graphs . . . . .	28
3.2	Doxygen Documentation . . . . .	29
3.2.1	Vscode Extension . . . . .	29
3.2.2	Doxygen Command Line . . . . .	29
3.2.3	Document Custom Target . . . . .	31
3.2.4	Doc.cmake . . . . .	31
3.3	Catch2 - Unit Testing . . . . .	32
3.3.1	Function testing example . . . . .	32
3.3.2	Test Directory . . . . .	32
3.3.3	Tests Set-up, CMakeLists.txt . . . . .	33
3.3.4	Defining Tests . . . . .	33
3.3.5	Command Line Testing Option . . . . .	33
3.3.6	On Testing Libraries in General . . . . .	34
3.4	Public, Interface and Private . . . . .	34
3.4.1	Different Linking Types . . . . .	34
3.4.2	Public . . . . .	34
3.4.3	Private . . . . .	35
3.4.4	Interface . . . . .	35
3.5	Different Library Types . . . . .	35
3.5.1	Library . . . . .	35
3.5.2	Shared . . . . .	35
3.5.3	Static . . . . .	35
3.6	Adding Compiler Warnings . . . . .	36
3.6.1	Check user's Compiler . . . . .	37
3.6.2	Warnings Root CMakeLists Options . . . . .	37
3.6.3	Executable Target Enable Warnings . . . . .	38
3.7	Sanitizers . . . . .	38
3.7.1	Sanitizers.cmake . . . . .	39
3.7.2	Activate Sanitizers in Root CMakeLists.txt . . . . .	40
3.7.3	Sanitizer bug example . . . . .	40
3.8	IPO LTO . . . . .	40

3.8.1	LTO Example - Clang . . . . .	41
3.8.2	IPO LTO in CMake . . . . .	42
3.8.3	LTO.cmake . . . . .	42
3.8.4	LTO.cmake Refactoring . . . . .	42
3.8.5	In Our Target CMakeLists.txt . . . . .	43
<b>4</b>	<b>Cpp Project Template</b>	<b>45</b>
4.1	Clang Tools . . . . .	45
4.1.1	Tools.cmake . . . . .	45
4.1.2	Tool Directory . . . . .	48
4.1.3	Clang-Tidy . . . . .	48
4.1.4	.clang-tidy file . . . . .	48
4.1.5	Clang-tidy Target . . . . .	49
4.1.6	Clang-tidy in root CMakeLists.txt . . . . .	49
4.1.7	Clang-tidy on Microsoft MSVC . . . . .	49
4.1.8	Clang-Format . . . . .	49
4.1.9	Clang-Format Configuration in Tools.cmake . . . . .	50
4.1.10	CMake-Format . . . . .	51
4.1.11	CMake-Format Configuration in Tools.cmake . . . . .	51
4.1.12	Glob and Exclude Cmake Functions . . . . .	52
4.1.13	Github Pages . . . . .	52
4.2	Code Coverage . . . . .	53
4.2.1	Code Coverage Configuration in root CMakeLists.txt . . . . .	53
4.2.2	Code Coverage for target library in my lib CMakeLists.txt . . . . .	53
4.2.3	Code Coverage Directory . . . . .	54
4.2.4	CodeCoverage.cmake . . . . .	54
4.3	Continuous Integration . . . . .	58
4.3.1	Github Actions . . . . .	58
4.3.2	Github Actions Workflows . . . . .	58
4.3.3	Codecov . . . . .	59
4.4	Pre-Commit . . . . .	59
4.4.1	Pre-Commit Installation . . . . .	60
4.4.2	Pre-Commit Configuration . . . . .	60
4.4.3	Pre-Commit in GitHub Actions Workflow Directory . . . . .	60
4.5	Install Command for your Program . . . . .	61
4.5.1	Install in root CMakeLists.txt . . . . .	61
4.5.2	Install any CMake Program on the Internet . . . . .	61
4.6	Shared Libraries . . . . .	62
4.7	Static Libraries . . . . .	62
4.8	Cmake GUI . . . . .	62
4.9	ImGui . . . . .	62
4.10	Jason Turner's Template . . . . .	62
4.10.1	Lefticus Defaults - ProjectOptions.cmake . . . . .	62
4.10.2	Hardening - Hardening.cmake . . . . .	62
4.11	Simple Cmake (Modern) . . . . .	62
4.11.1	Context . . . . .	62
4.11.2	CMakeLists.txt . . . . .	63
4.11.3	Cmake . . . . .	63
4.11.4	Make . . . . .	63
4.11.5	Build folder . . . . .	64
4.11.6	Sick CMake Vim plugins combos . . . . .	64
4.11.7	COC - for code completion in nvim . . . . .	64

4.11.8	Include Header File - CMake Continued . . . . .	64
4.11.9	Pragma Once . . . . .	65
4.11.10	Glob - Include Many files with CMake . . . . .	65
4.11.11	src Directory - source files . . . . .	65
4.11.12	CMake Custom Libraries . . . . .	66
4.11.13	Custom Library Implementation - Blah example . . . . .	66
4.12	Jason Turner's CMake Template - Options . . . . .	67

# Chapter 1

## Cmake

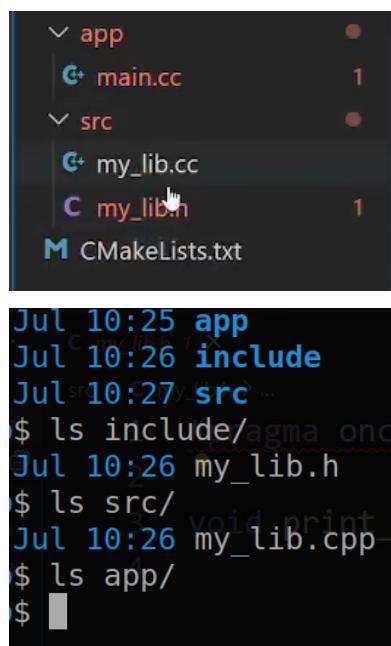
### 1.1 Overview

Cmake is super old. It has more than 300 functions to use, but 250 should not be used in modern cmake projects. This makes it particularly difficult to learn. Many older tutorials are confusing, out-of-date.

Moreover, modern cmake makes it as readable as possible with modern functions. Thus, way easier to read than older functions.

### 1.2 Project Structure

See basic project and intermediate project examples, in cmake udemy.



The image shows a terminal window with two parts. The top part displays a file tree for a project named 'app'. It contains a 'src' directory which holds 'my\_lib.cc' and 'my\_lib.h'. The bottom part shows a command-line session where the user lists files in 'include', 'src', and 'app' directories. The session starts with 'Jul 10:25 app', followed by 'Jul 10:26 include', 'Jul 10:27 src', and then a series of 'ls' commands for each of these three directories. The output for 'ls include' shows 'agma\_onc'. The output for 'ls src' shows 'void print'. The output for 'ls app' shows nothing.

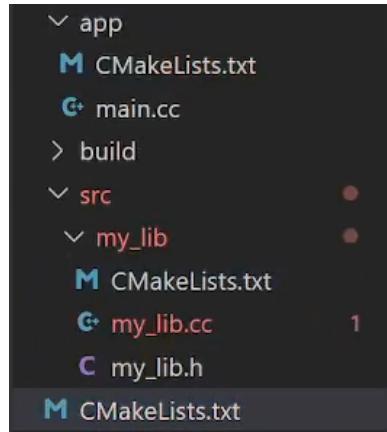
```
Jul 10:25 app
Jul 10:26 include
Jul 10:27 src ...
$ ls include/agma_onc
Jul 10:26 my_lib.h
$ ls src/
Jul 10:26 void print
$ ls app/
$
```

### 1.2.1 App Directory

Define everything important for the executable target, including its own CMakeLists.txt.

### 1.2.2 Source Directory - src

Define everything important for our library. When you have multiple libraries, create multiple directories in src. As a naming convention, the subdirectory should have the same name as the library.



Don't forget to add a CMakeList in the src directory.

```
src > M CMakeLists.txt
1 add_subdirectory(my_lib)
2
```

### 1.2.3 CMake Default Paths

Cmake has built-in paths to use. We use many in these notes.

```
# CMAKE_SOURCE_DIR is always the directory of the
# root CMakeLists.txt file
# our root directory

# CMAKE_BINARY_DIR is always our build directory
```

#### 1.2.4 Template for Projects is Intermediate project

Create a project builder based on the architecture of the intermediate project.

### 1.3 Variables

You can create variables for your executable or your libraries in the root CMakeLists.txt. The variables will be usable in all add\_subdirectory chain at the end of the same document.

```
# set a variable for the library you want to use
# common syntax is capital letters.

# we reference Library in other CMakeLists files
# we change the Library name for LIBA

set(LIBA Library)

# where we used Library, write ${LIBA}

# we have used Hello for our executable name so far.
# we can change it as well.
set(HE Hello)

# where we used Hello, write ${HE}

add_subdirectory(src)
add_subdirectory(app)
```

#### 1.3.1 Standard for cpp

This is essential to have in your program. Otherwise, the compiler's default config will take the lead, with huge variability between compiler versions! In your root CMakeLists.txt file as well. This defines a variable and sets the standard to 17.

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_C_STANDARD 98)
```

#### 1.3.2 Standard Required

Indicate that the compiler 100 pourcent implemented the language's standard.

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
```

#### 1.3.3 Compiler Extensions

Some compilers have features that are not implemented in the Cpp standard. These features are extensions. Some compilers allow you to use non-standard c code into a cpp program, even if they

are not in the cpp standard.

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)
```

### 1.3.4 If Statement

Cmake has if statement. It even has string comparaison functions, to compare variable and create conditions.

```
option(compile_executable "Whether to compile the executable" OFF)

add_subdirectory(src)

if (COMPILE_EXECUTABLE)
    add_subdirectory(app)
else()
    message("Without executable compiling")
endif()
```

### 1.3.5 Options

You can set an option, the second argument is a simple comment for the reader (with no impact).

```
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)
set(CMAKE_CXX_EXTENSIONS OFF)

set(LIBA Library)
set(HE Hello)

option(compile_executable "Whether to compile the executable" OFF)

add_subdirectory(src)

if (COMPILE_EXECUTABLE)
    add_subdirectory(app)
endif()

# to call the option from the command line

## cmake .. -DCOMPILE_EXECUTABLE=ON

# then you can switch it back OFF later

## cmake .. -DCOMPILE_EXECUTABLE=OFF
```

## 1.4 Makefile Shell Scripting

Make is Cmake's ancestor. You can automate folder creation and files, just like any shell script would do.

Make do not support space, use tabs.

```
prepare:  
    rm -rf build  
    mkdir build  
    cd build  
  
# to execute it  
  
# $ make prepare
```

## 1.5 Project Software Toolkit

```
Doxygen – create html documentation based on your codebase  
Conan/VCPKG Packaging – How to install and use external libraries  
Unit Testing  
Code Coverage  
CI Testing -- use all these tools in continuous integration, Github actions
```

We'll see how to create an html documentation based on your codebase

## 1.6 Installation Commands

```
sudo apt-get update  
sudo apt-get upgrade  
  
# Mandatory  
sudo apt-get install gcc g++ gdb  
sudo apt-get install make cmake  
sudo apt-get install git  
sudo apt-get install doxygen  
sudo apt-get install python3 python3-pip  
  
# Optional  
sudo apt-get install lcov gcovr  
sudo apt-get install ccache  
sudo apt-get install cppcheck  
sudo apt-get install llvm clang-format clang-tidy  
sudo apt-get install curl zip unzip tar  
  
# for VS CODE  
  
in extension, download franneck94 c/c++ extension pack
```

And the coding tools extension pack  
Then, with command palette (in view), Config: generate C config file. It configures all tools the same way as the teacher's

## 1.7 Command Line Options

### 1.7.1 Generating a Project

```
cmake [<options>] -S <path-to-source> -B <path-to-build>
```

Assuming that a CMakeLists.txt is in the root directory, you can generate a project like the following.

```
mkdir build
cd build
cmake -S .. -B . # Option 1
cmake .. # Option 2
```

Assuming that you have already built the CMake project, you can update the generated project.

```
cd build
cmake .
```

### 1.7.2 Generator for GCC and Clang

```
cd build
cmake -S .. -B . -G "Unix Makefiles" # Option 1
cmake .. -G "Unix Makefiles" # Option 2
```

### 1.7.3 Generator for MSVC

```
cd build
cmake -S .. -B . -G "Visual Studio 16 2019" # Option 1
cmake .. -G "Visual Studio 16 2019" # Option 2
```

### 1.7.4 Specify the Build Type

Per default, the standard type is in most cases the debug type. If you want to generate the project, for example, in release mode you have to set the build type.

```
cd build
cmake -DCMAKE_BUILD_TYPE=Release ..
```

### 1.7.5 Passing Options

If you have set some options in the CMakeLists, you can pass values in the command line.

```
cd build
cmake -DMY_OPTION=[ON|OFF] ..
```

### 1.7.6 Specify the Build Target (Option 1)

The standard build command would build all created targets within the CMakeLists. If you want to build a specific target, you can do so.

```
cd build  
cmake --build . --target ExternalLibraries_Executable
```

The target ExternalLibraries\_Executable is just an example of a possible target name. Note: All dependent targets will be built beforehand.

### 1.7.7 Specify the Build Target (Option 2)

Besides setting the target within the cmake build command, you could also run the previously generated Makefile (from the generating step). If you want to build the ExternalLibraries\_Executable, you could do the following.

```
cd build  
make ExternalLibraries_Executable
```

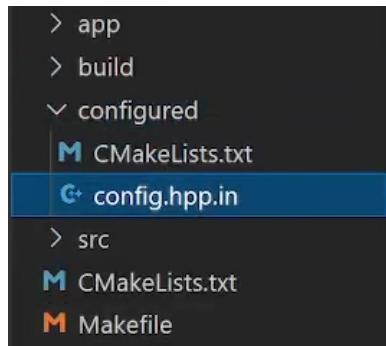
## 1.8 Run the Executable

After generating the project and building a specific target you might want to run the executable. In the default case, the executable is stored in build/5\_ExternalLibraries/app/ExternalLibraries\_Executable, assuming that you are building the project 5\_ExternalLibraries and the main file of the executable is in the app dir.

```
cd build  
.bin/ExternalLibraries_Executable
```

## 1.9 Configuration - Precompiling Information

Create a configuration directory. With a config.hpp.in file.



In the CMakeLists.txt of this new configured directory.

```
configure_file(  
  "config.hpp.in"  
  # "${CMAKE_BINARY_DIR}" this is our build directory
```

```

# it is one of prebuilt directories in CMAKE
# thus, we reference it with CMAKE_BINARY_DIR
# we create an output for a config file,
# in our build directory.

# "${PROJECT_SOURCE_DIR}" # stores absolute path to project's root directory
"${CMAKE_BINARY_DIR}/configured_files/include/config.hpp" ESCAPE_QUOTES

```

### 1.9.1 Project Version Number

In config.hpp.in

```

# @ cmake looks for and replaces text between @@ this text @

#include <cstdint>
#include <string_view>
static constexpr std::string_view project_name = "@PROJECT_NAME@";
static constexpr std::string_view project_version = "@PROJECT_VERSION@";

```

### 1.9.2 Sementic Versioning

In version names: 1.0.0 . This is the first Major version (1). Incrementing the major version means that the previous and the new codebase are not compatible at all. 2.0.0 have breaking changes with the 1.0.0 version.

The minor version 1.6.0, number 6 here, indicate that new features are available. Yet, nothing breaks between minor versions.

The patches are the last number, the last small fixes.

```

static constexpr std::int32_t project_version_major{@PROJECT_VERSION_MAJOR@};
static constexpr std::int32_t project_version_minor{@PROJECT_VERSION_MINOR@};
static constexpr std::int32_t project_version_patch{@PROJECT_VERSION_PATCH@};

```

#### Resulting file looks like.

```

#pragma once

#include <cstdint>
#include <string_view>

static constexpr std::string_view project_name = "CppProjectTemplate";
static constexpr std::string_view project_version = "1.0.0";

static constexpr std::int32_t project_version_major{1};
static constexpr std::int32_t project_version_minor{0};
static constexpr std::int32_t project_version_patch{0};

```

## 1.10 Sources and Headers

When you have many libraries and many headers, create a variable for all of your headers. Then, reference this variable to include everything. List all of your source files in my\_lib directory.

```
set(LIBRARY_SOURCES
    "my_lib.cpp"
    "my_lib2.cpp"
    ) # quotes are not mandatory, just preference.
set(LIBRARY_HEADERS
    "my_lib.h")

add_library(${LIBRARY_NAME} STATIC
    "./"
    "${CMAKE_BINARY_DIR}/configured_files/include")
```

### 1.10.1 Executables

You can do the same thing if you generate many executables, I think. This would be in your app directory.

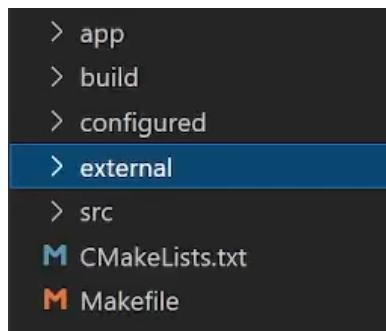
```
set(EXE_SOURCES
    "main.cpp")

add_executable(${EXECUTABLE_NAME} ${EXE_SOURCES})
target_link_libraries(${EXECUTABLE_NAME} PUBLIC ${LIBRARY_NAME})
```

## 1.11 Using External Libraries

### 1.11.1 Git Submodule - nlohman json

Create an external directory. Our project looks like this:



```
// make sure that your root folder is a git repo, git init
git submodule add https://github.com/nlogmann/json external/json
// it creates the json directory, when cloning the module in it.
```

### 1.11.2 Custom Cmake Functions

In a cpp project, when you create your own cmake functions, they are stored in a Cmake directory. We just cloned a git repo in our external directory.



The screenshot shows a terminal window with a file tree on the left and a code editor on the right. The file tree shows a directory structure with 'gitmodules' and 'EXTERNAL...' at the top, followed by 'vscode', 'app', 'build', 'cmake', and 'AddGitSubmodule.cmake'. The code editor displays a file named '.gitmodules' containing the following content:

```
1 [submodule "external/json"]
2   path = external/json
3   url = https://github.com/nlohmann/json
4
```

Now we create a AddGitSubmodule.cmake file. Inside, we define our function.

```
function(add_git_submodule dir) # our function takes one argument called dir
    # this dir, is where the submodule is located
    # making the add_ action possible.
    find_package(Git REQUIRED) # Git must be on your computer, or it
    # Error's out.
    if (NOT EXISTS ${dir}/CMakeLists.txt)
        execute_process (COMMAND ${GIT_EXECUTABLE})
        submodule update --init --recursive -- ${dir}
            # recursive is essential here
            # If someone clones your project,
            # It automatically clones this submodules,
            # It clones auto repos, inside your repo.

        WORKING_DIRECTORY ${PROJECT_SOURCE_DIR}) # cmake saves some variable
        # including the source dir
        # of our project.

    endif()

    if (EXISTS ${dir}/CMakeLists.txt)
        message("Adding ${dir}/CMakeLists.txt")
        add_subdirectory(${dir})
    else
        message("Could not add ${dir}/CMakeLists.txt")
    endif()
endfunction(add_git_submodule)
```

### 1.11.3 Log example

Here we are using an external library called log, with only two files, log.c and log.h . We have the library in our external directory. Plus, we have a CMakeLists.txt at the root of the external folder, taking care of the log library.

```

set(LOG_SOURCES "${CMAKE_CURRENT_SOURCE_DIR}/log/src/log.c")
set(LOG_HEADERS "${CMAKE_CURRENT_SOURCE_DIR}/log/src/log.h")
set(LOG_INCLUDES "${CMAKE_CURRENT_SOURCE_DIR}/log/src/")
add_library(log STATIC ${LOG_SOURCES} ${LOG_HEADERS})
target_include_directories(log PUBLIC ${LOG_INCLUDES})
|
```

#### 1.11.4 Call Your Own Functions

In the source CMakeFiles.txt, knowing that we have a new git submodule. We add:

```

set(CMAKE_MODULE_PATH "${PROJECT_SOURCE_DIR}/cmake")
include(AddGitSubmodule) # includes indicates that it a cmake module file.
                        # it will look where cmake modules are defined
                        # in our cmake dir

add_git_submodule(external/json) # this calls our custom function.

# In our app, we add this as well

set(EXE_SOURCES
    "main.cpp")

add_executable(${EXECUTABLE_NAME} ${EXE_SOURCES})
target_link_libraries(${EXECUTABLE_NAME} PUBLIC
    ${LIBRARY_NAME}
    nlohmann_json) # this links the submodule with the executable

# In main.cpp, we add

#include <nlohman/json.hpp>
```

#### 1.11.5 Fetch Content - External Libraries

We have the gitmodule example, but modern CMake has a great fetching feature. Our root CMakeLists.txt will look like this,

```

...
set(CMAKE_MODULE_PATH "${PROJECT_SOURCE_DIR}/cmake/")
include(AddGitSubmodule)

include(FetchContent) # built-in library or file
                      # including it gives access to features

FetchContent_Declare() # Declare which github repository we would like to use
```

```

FetchContent_MakeAvailable() # will load this library in our cmake project.

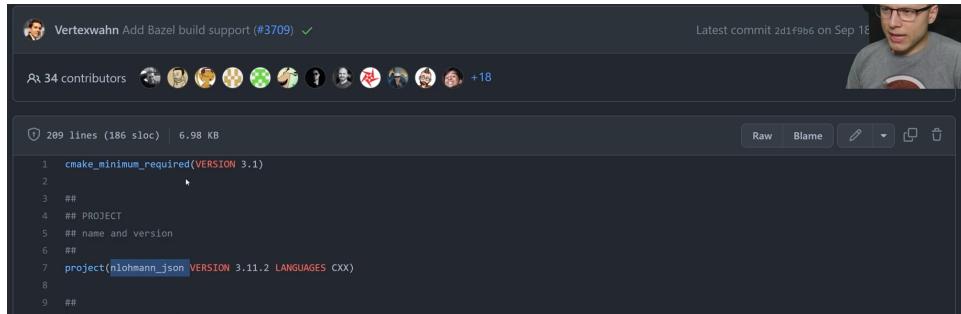
# I want to use github.com/nlohmann/json
# any Gitlab is also possible
# I can do:
FetchContent_Declare(
    nlohmann_json      # since this repo is a cmake project,
                        # look at the project's root CMakeLists.txt file
                        # you will the name of the project, to enter here
                        # see next image

    GIT_REPOSITORY https://github.com/nlohmann/json
    GIT_TAG v3.11.2    # the version I want to use
    GIT_SHALLOW TRUE) # The function won't clone the repo recursively

                        # With this function, the git repository will be cloned in
                        # our cloned repository.
                        # And it needs to be a cmake project.

                        # if it is not a cmake project, use the AddGitSubmodule method shown.
FetchContent_MakeAvailable(nlohmann_json) # will load this library in our cmake project.

```



## 1.11.6 FMT

The best library to easily format strings in cpp.

```

FetchContent_Declare(
    fmt
    GIT_REPOSITORY https://github.com/fmtlib/fmt
    GIT_TAG 9.1.0
    GIT_SHALLOW TRUE)
FetchContent_MakeAvailable(fmt)

```

## 1.11.7 spdlog

The best fast logging library for cpp.

```

FetchContent_Declare(
    spdlog

```

```

GIT_REPOSITORY https://github.com/gabime spdlog
GIT_TAG v1.11.0
GIT_SHALLOW TRUE)
FetchContent_MakeAvailable(spdlog)

```

### 1.11.8 Cxxopts

The best library to work with command line arguments in cpp. From the received arguments, to any other type. The equivalent of the argument parser in python.

```

FetchContent_Declare(
    cxxopts
    GIT_REPOSITORY https://github.com/jaroo2783/cxxopts
    GIT_TAG v3.0.0
    GIT_SHALLOW TRUE)
FetchContent_MakeAvailable(cxxopts)

```

### 1.11.9 Catch2

The best Unit Testing library, seen further below.

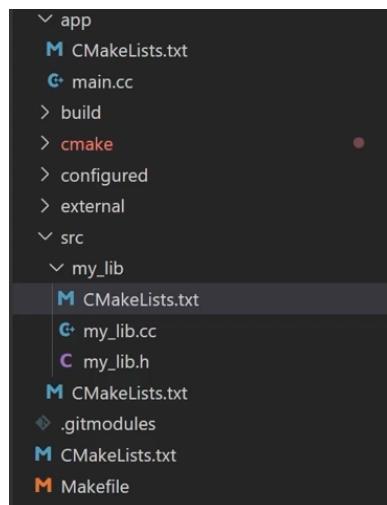
```

FetchContent_Declare(
    catch2
    GIT_REPOSITORY https://github.com/catchorg/Catch2
    GIT_TAG v2.13.9 # teacher recommended this version, not the latest.
    GIT_SHALLOW TRUE)
FetchContent_MakeAvailable(catch2)

```

### 1.11.10 Include All Libraries in root CMakeListstxt

Refer to Final project Template seen in the course. Modifying the CMakeLists.txt in our my-lib directory.



```

set(LIBRARY_SOURCES
    "my_lib.cpp")
set(LIBRARY_HEADERS
    "my_lib.h")
set(LIBRARY_INCLUDES
    "./"
    "${CMAKE_BINARY_DIR}/configured_files/include")

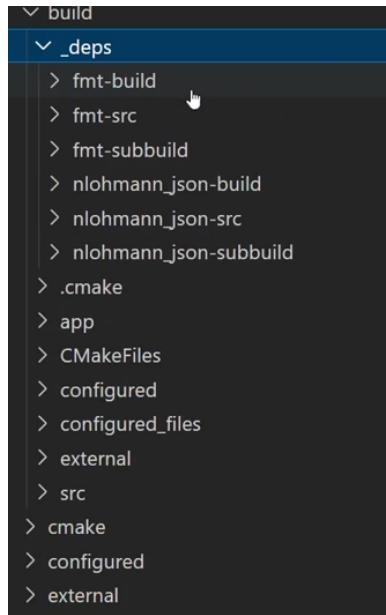
add_library(${LIBA} STATIC
    ${LIBRARY_SOURCES}
    ${LIBRARY_HEADERS})
target_include_directories(${LIBA} PUBLIC
    ${LIBRARY_INCLUDES})
target_link_libraries(${LIBA} PUBLIC
    # Naming convention is project_name::library_name
    # see next image, to find library_name in CMake project
    # on github

    nlohmann_json::nlohmann_json
    fmt::fmt
    spdlog::spdlog
    catch2::catch2
    cxxopts::cxxopts           # not always the same

)

```

When this is set-up, and we reconfigure our cmake project, the repository will be cloned in a \_deps directory. This includes a build, subbuild and src directory for all dependencies. Don't worry about it for now.



### 1.11.11 Include all Libraries in App Main.cpp

It is tricky to include the libraries files in main. Some have directories, some do not.

```
#include <iostream>

#include <cxxopts.hpp>
#include <nlohman/json.hpp>
#include <fmt/format.h>
#include <spdlog/spdlog.h>
#include <catch2>

#include "my_lib.h"
#include "config.hpp"

int main() {
    ...
    std::cout << "CXXOPTS: # chose any included lib, to prove you have access to their info
    << CXXOPTS__VERSION_MAJOR << "."
    << CXXOPTS__VERSION_MINOR << "."
    << CXXOPTS__VERSION_PATCH << "." # if you have access to these variable,
                                         # you have successfully imported and configure the lib
                                         # for your project.

    ...
}
```

### 1.11.12 Git Submodules vs Fetch Content

If the repo is not a CMakeProject, you should use Git Submodules. Valid for GitHub and GitLab. In this case, define its own library target, I think. Not explained in detail.

If it is a CMake project on GitHub or Gitlab, use FetchContent. It is easier to use, you don't need to mess with header libraries or anything. We can simply use the makeAvailable and prepare function.

The instructor highly recommends FetchContent!

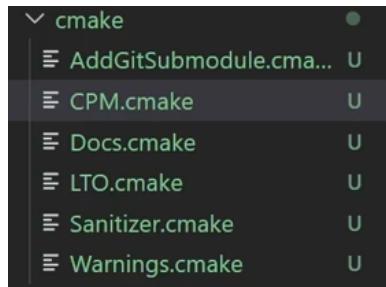
## Chapter 2

# CMake Package Managers

### 2.1 CPM - Cmake Package Manager

There are a few ways to include external libraries in your project. Git submodules, fetch content and packages managers such as Cmake Package Manager.

On CPM's github page, click Releases. Pick the latest release, download the CPM.cmake file and copy it to your project's cmake directory.



#### 2.1.1 CPM configuration in root CMakeLists.txt

Add an option in your root CMakeLists.txt. Since the course's project has two different ways to import external libraries, we will have an option for cpm and for fetch content. We shouldn't use multiple tools at the same time. Thus, code an if statement to use one or the other fetching method.

**Every external libraries used under CPM need to be github CMake projects, which 95 pourcent are. It shouldn't be a problem.**

**Under the hood, CPM uses fetchcontent.** Therefore, the linking, in the src/my\_lib/CMakeLists.txt file target\_link\_libraries function, can keep the fetchcontent synthax of nlohman\_json::nlohman\_json.

```
option(USE_CPM "Whether to use CPM" ON)

if(USE_CPM)
    message(STATUS "Using Cmake Package Manager")
    include(CPM)          # this includes the cpm.cmake file
```

```

# "gh" cpm will look at github
# "gh:nholmann" username
# "gh:nholmann/json" repository name
# "gh:nholmann/json#v3.11.2" version number

cpmaddpackage("gh:nholman/json#v3.11.2")      # This is CPM's defined function
cpmaddpackage("gh:fmtlib/fmt#9.1.0")
cpmaddpackage("gh:gabime/spdlog#v1.11.0")
cpmaddpackage("gh:jarro2783/cxxopts#v3.1.1")
cpmaddpackage("gh:cathorg/Catch2#v2.13.9")

else()
    message(STATUS "Using FetchContent")

FetchContent_Declare(
    nlohmann_json
    GIT_REPOSITORY https://github.com/nlohmann/json
    GIT_TAG v3.11.2
    GIT_SHALLOW TRUE)
FetchContent_MakeAvailable(nlohmann_json) # will load this library in our cmake project.

FetchContent_Declare(
    fmt
    GIT_REPOSITORY ...

endif()

```

## 2.2 Conan

A package manager for cmake projects, alternatives to CPM and fetchcontent. CPM and fetchcontent locally clones github repositories in your build directory. Then they compile the library locally, on your machine.

Conan has a different approach. In an online database, pre-compiled repositories are available. Conan downloads these. It depends on the compiler, but it generally saves compilation time. No need to compile locally.

Conan binaries are compiled as release builds, not as debug builds.

A drawback for Conan is the binary updates. There are many configuration possible, and many updates to keep track of. Therefore, many popular libraries won't have a compiled version for our machine's configuration. Too many possibilities, Conan can't keep up with updates.

### 2.2.1 Conan Installation

Official installation guide is [here](<https://docs.conan.io/2/>).

The conan database is [here](<https://conan.io/center/>).

1. Install Python (3.7+)
2. Type ‘‘pip install --user -U conan’’ into the terminal
  1. Unix: Append conan to the PATH by: ‘‘source ~/.profile’’
  3. Run: \$ conan
4. \$ conan profile detect --force
5. \$ conan profile path default

## 2.2.2 Conan Configuration in root CMakeLists.txt

There is a pattern here, when you want to use something new in your cmake project, set an option in the CMakeLists.txt file first.

Since the course shared examples for GitSubmodules, FetchContent, CPM and Conan, we have a large if statement in our root CMakeLists.txt. In a regular project, select the tool you want, you wouldn't need the if.

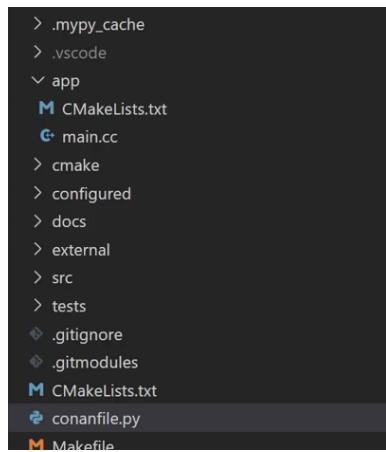
```
option(USE_CONAN "Whether to use CPM" ON)

if(USE_CPM)
  ...
elseif(USE_CONAN)
  message(STATUS "Using Conan")
  include(${CMAKE_BINARY_DIR}/conan_toolchain.cmake
    # This is generated by a conan command
    # in the conanfile.py (generate)
    # including it in advance here.
  find_package(nlohmann_json REQUIRED)
  find_package(fmtlib REQUIRED)
  find_package(spdlog REQUIRED)
  find_package(cxxopts REQUIRED)
  find_package(Catch2 REQUIRED)

else()
  FetchContent_Declare(
    ...
  )
```

## 2.2.3 Conanfile.py

To configure conan, create a conanfile.py in your root directory.



```

from conan import ConanFile
from conan.tools.cmake import CMakeToolchain

class CompressorRecipe(ConanFile):
    settings = "os", "compiler", "build_type", "arch"
    generators = "CMakeDeps"

    def requirements(self):
        self.requires("nlohman_json/3.11.2")      # look at conan's database
                                                    # to see which pre-compiled
                                                    # versions are available
        self.requires("fmt/9.1.0")
        self.requires("spdlog/1.11.0")
        self.requires("catch2/2.13.9")
        self.requires("cxxopts/3.1.1")

    def generate(self):                          # this function generates the
                                                # conan_toolchain.cmake file
        tc = CMakeToolchain(self)
        tc.user_presets_path = False
        tc.generate()

```

#### 2.2.4 Conan Debug Build

Conan has release binaries by default, you have to tweak settings for your debug build. To automatize conan's debug management, the instructor changed its makefile script.

```

conan_d:      # for conan debug
    rm -rf build
    mkdir build
    cd build && conan install .. -s build_type=Debug =s compiler.cppstd=17 --output-folder=. --build
                -s                      # for settings change
                -- build missing
                # build binaries where no compiled version are available

conan_r:      # for conan release
    rm -rf build

```

```

mkdir build
cd build && conan install .. -s build_type=Release -s compiler.cppstd=17 --output-folder=. --buil

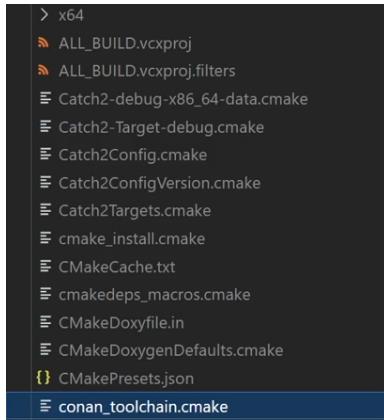
### With this, run

$make conan_d
$make conan_r

```

### 2.2.5 Conan Generated File

Conan generates many files in your build directory, based on our configurations. Yet, no need to worry about them, conan will deal with them.



## 2.3 VCPKG

A C/C++ package manager for Microsoft. To use it, clone the VCPKG repository in your project's external library directory.

VCPKG database for available packages.

[vcpkg.io/en/packages.html](http://vcpkg.io/en/packages.html)

Better is

[vcpkg.link](http://vcpkg.link)

### 2.3.1 VCPKG Installation

Move to you project's external directory and clone the repo. Then, execute the .sh script.

Official Link: <<https://vcpkg.io/en/index.html>>

cd external

```

git clone https://github.com/Microsoft/vcpkg.git

.\vcpkg\bootstrap-vcpkg.bat # windows
./vcpkg/bootstrap-vcpkg.sh # Unix

cd vcpkg
vcpkg --help

```

### 2.3.2 VCPKG.json file in project root

List dependencies of your cmake project in a JSON file. VCPKG's synthax is very tricky, plus its requirements are complicated for no reason.

It automatically downloads the latest versions of all libraries. Thus, if we want one particular version, we need this overrides keyword.

```

{
  "name": "cpptemplateproject",
  "version-string": "1.0.0",
  "dependencies": [
    {
      "name": "cxxopts",
      "version>=": "3.1.1"
    },
    {
      "name": "fmt",
      "version>=": "9.1.0"
    },
    {
      "name": "nlohmann-json",
      "version>=": "3.11.2"
    },
    {
      "name": "spdlog",
      "version>=": "1.11.0"
    },
    {
      "name": "catch2",
      "version>=": "2.13.9"
    }
  ],
  "overrides": [
    {
      "name": "catch2",
      "version": "2.13.9"
    }
  ],
  "builtin-baseline": "40619a55c3e76dc4005c8d1b7395071471bb8b96"

  # this baseline is the hexadecimal value
  # of a certain VCPKG git commit
}

```

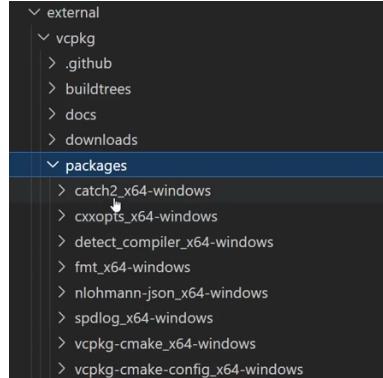
```

# this is ridiculously complicated,
# in this examples here, we git logged the external/vcpkg directory of our project.
}

```

\external\vcpkg>git log

After configuration, libraries are downloaded and compiled on your system.



### 2.3.3 VCPKG configuration in root CMakeLists.txt

Include vcpkg.cmake file in our project.

```

option(USE_CONAN "Whether to use CPM" ON)

if(USE_CPM)
    ...
elseif(USE_CONAN)
    message(STATUS "Using Conan")
    include(${CMAKE_BINARY_DIR}/conan_toolchain.cmake
        # This is generated by a conan command
elseif(USE_VCPKG)
    message(STATUS "Using VCPKG")

    include(${CMAKE_SOURCE_DIR}/external/vcpkg/scripts/buildsystems/vcpkg.cmake)
    find_package(nlohmann_json REQUIRED)
    find_package(fmt REQUIRED)
    find_package(.. REQUIRED)

else()
    FetchContent_Declare(
        ...

```

## 2.4 Package Managers Summary

There are many ways to add and include libraries to our cmake project: Git Submodules, FetchContent, Cmake Package Manager (CPM), Conan and VCPKG.

Use Git Submodules if you github repo you want to use is not a cmake project. It is a rare case, but it is the best solution. Do not use this approach if they are cmake projects.

**CPM is the instructors' recommendation** for all gitlab and github cmake projects. However, it is important to understand the FetchContent functions of CMake. CPM uses these same fetch content methods under the hood.

VCPKG is hard to configure and version download complications. Not recommended.

Conan's pre-built binaries are a great idea, but they do not update these binaries often enough. However, if you have huge libraries that take 30 minutes to compile. Conan is the way to go.

# Chapter 3

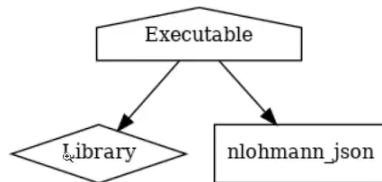
## CMake Tooling

### 3.1 Dependency Graphs

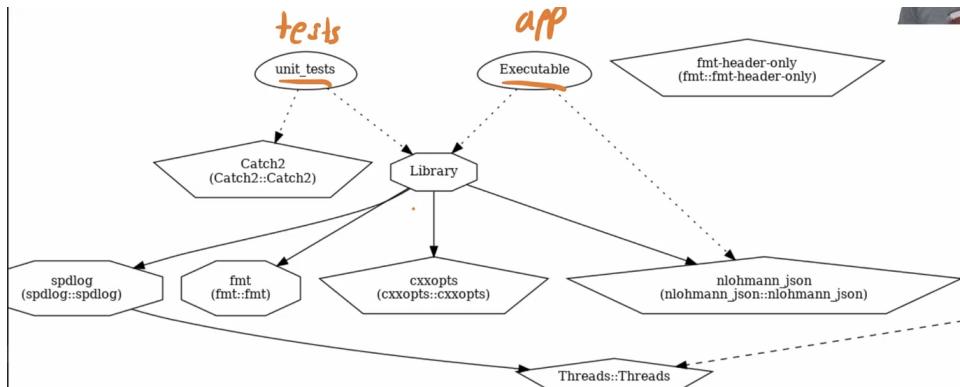
In a makefile, or any script, use the command `--graphviz`. Dependency and prepare are the flag needed to run the command.

```
dependency:  
    cd build && cmake .. --graphviz=graph.dot && dot -Tpng graph.dot -o graphImage.png  
        // not yet an image when .dot  
        // but easy to transform  
  
prepare:  
    rm -rf build  
    mkdir build  
        // unrelated to dependency graph.
```

The house symbol is the executable. Rectangle are external libraries. Losanges shapes are internal libraries.



Towards the end of the course, the graph was more complex.



## 3.2 Doxygen Documentation

Generate html documentation for our code. For example, for our library. The course has a vscode extention: Doxygen Documentation Generator.

### 3.2.1 Vscode Extension

The extention generates documentation base on this synthax.

```
#include <iostream>

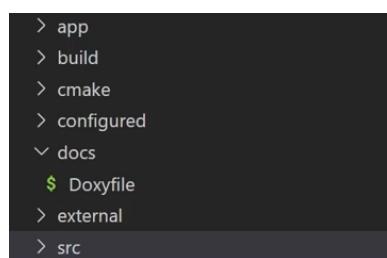
#include <nlohmann/json.hpp>
#include "my_lib.h"

/***
 * @brief Prints out hello world and tests the JSON lib.
 *
 */

```

### 3.2.2 Doxygen Command Line

Doxygen is looking for a doxy file, a config file. Generate a doxy file with doxygen -g. In it, fill information on the project, name, version, path to source files, etc. It will generate better html file with the information. In the docs directory.



```
# Configuration for Doxygen for use with CMake
# Only options that deviate from the default are included
```

```

# To create a new Doxyfile containing all available options, call 'doxygen -g'

#-----
# Project related configuration options
#-----
DOXYFILE_ENCODING      = UTF-8
PROJECT_NAME           = "C++ Project Template"
PROJECT_NUMBER          = 1.0
PROJECT_BRIEF          =
PROJECT_LOGO           =
OUTPUT_DIRECTORY        = ../
OUTPUT_LANGUAGE         = English
MARKDOWN_SUPPORT        = YES

#-----
# Build related configuration options
#-----
EXTRACT_ALL            = YES
RECURSIVE               = YES
GENERATE_HTML           = YES
GENERATE_LATEX          = NO

#-----
# Configuration options related to the input files
#-----
INPUT                  =     ../src \
INPUT                  =     ../include
INPUT_ENCODING          = UTF-8
FILE_PATTERNS          = *.c \
                           *.cc \
                           *.cpp \
                           *.c++ \
                           *.h \
                           *.hpp \
                           *.h++ \
                           *.md \
                           *.dox \
                           *.doc \
                           *.txt

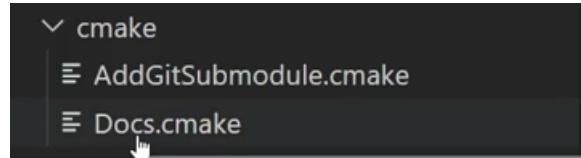
```

When ready, generate the html with the command Doxygen, inside the docs folder (where the doxyfile is located). It creates an html dir. it has the index.html automatically. The webpage looks like this:



### 3.2.3 Document Custom Target

We need to add the documentation to our cmake project.



In our root CMakeLists.txt, we add.

```
include(AddGitSubmodule)
include(FetchContent)
include(Docs)           # this is our new dir.
```

### 3.2.4 Doc.cmake

Cmake needs to find Doxygen, because we are using it for our docs. In Docs.cmake,

```
find_package(Doxygen)
if (DOXYGEN_FOUND)
    add_custom_target( # This is just an utility target
                      # With it, we can interact with it,
                      # in the terminal
        docs
        ${DOXYGEN_EXECUTABLE}
        WORKING_DIRECTORY ${CMAKE_SOURCE_DIR}/docs

                      # CMAKE_SOURCE_DIR is always the directory of the
                      # root CMakeLists.txt file
                      # our root directory

        # CMAKE_BINARY_DIR is always our build directory
endif()
```

Now Documentation can be built seperately, independently of the main project app built process.

## 3.3 Catch2 - Unit Testing

### 3.3.1 Function testing example

Adding Unit test to our codebase, with the catch2 library. Unit tests are useful to test functions from the library.

There is a tutorial on the github catch2 page. As an example, it provides this factorial function example, to try.

```
unsigned int factorial( unsigned int number ) {
    return number <= 1 ? number : factorial(number-1)*number;
}

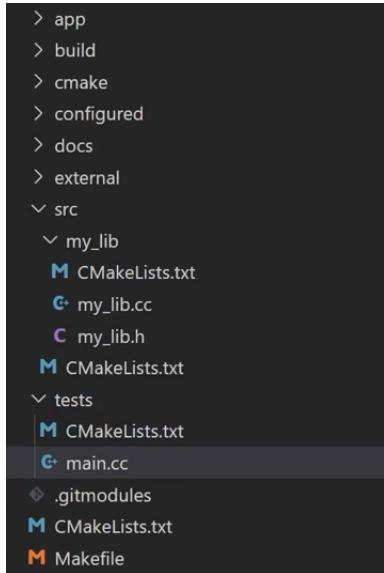
// the instructor changes it to

std::uint32_t factorial(std::uint32_t number)
{

    return number <= 1 ? number : factorial(number-1) * number;
}
```

### 3.3.2 Test Directory

To introduce unit testing, we create a new directory: tests.



In our root CMakeLists.txt.

```
add_subdirectory(configured)
add_subdirectory(external)
add_subdirectory(src)
add_subdirectory(app)
```

```
add_subdirectory(tests) # adding tests
```

The idea is to have our main executable, our library(lib) and our unit test executable. The main and the Unit executable we'll both use our library, but we will test with Unit. Unit will test if all our implemented functions work without bugs.

### 3.3.3 Tests Set-up, CMakeLists.txt

```
set(TEST_MAIN "unit_tests")           # this is the name of our testing executable
set(TEST_SOURCES "main.cpp")          # Here, the tests are in one file only
                                      # It could be divided into more files
                                      # Header files for example
set(TEST_INCLUDES "./")
add_executable(${TEST_MAIN} ${TEST_SOURCES})
target_include_directories(${TEST_MAIN} PUBLIC ${TEST_INCLUDES})
target_link_libraries(${TEST_MAIN} PUBLIC ${LIBA} Catch2::Catch2)

                                      # Our library is linked, LIBA
                                      # This is how we will test it
```

### 3.3.4 Defining Tests

To test the factorial function, this is the test definition given as example.

```
#define CATCH_CONFIG_MAIN // This tells Catch to provide a main() - only do this in one file
                           // No need to write int main() {}, this does it.
#include "catch2/catch.hpp"

#include "my_lib.h"      // Will be called in my_lib.h

TEST_CASE( "Factorials are computed", "[factorial]" ) {
    REQUIRE( Factorial(1) == 1);
    REQUIRE( Factorial(2) == 2);
    REQUIRE( Factorial(3 == 6);
    REQUIRE( Factorial(10 == 362880 );
}

This function needs to be called in my_lib.h
std::uint32_t factorial(std::uint32_t number);
```

### 3.3.5 Command Line Testing Option

It is convenient to have a command line option to activate or deactivate our testing build. In our root CMakeLists.txt file, we add an option. In our CMakeLists.txt in the tests directory, we add an if statement.

```
option(ENABLE_TESTING "Enable a Unit Testing Build" ON)

# in tests directory's CMakeLists.txt
```

```

if (ENABLE_TESTING)
    set(TEST_MAIN "unit_tests")
    set(TEST_SOURCES "main.cpp")
    set(TEST_INCLUDES "./")

    add_executable(${TEST_MAIN} ${TEST_SOURCES})
    target_include_directories(${TEST_MAIN} PUBLIC ${TEST_INCLUDES})
    target_link_libraries(${TEST_MAIN} PUBLIC ${LIBA} Catch2::Catch2)
endif()

```

### 3.3.6 On Testing Libraries in General

The specific words (Test\_case, require, etc.) may vary a bit, but they all have the same logic. When you are familiar with one, you will be able to use another testing library.

You have a test case,

```
TEST_CASE()
```

You can give it a name and a short-name(abbreviated)

```
TEST_CASE( "Factorials are computed", "[factorial]" )
```

you can test a function with a keyword like require, or require\_equal. Giving an input, the result should be equal to the output.

```
REQUIRE( factorial(0) == 0 )
```

## 3.4 Public, Interface and Private

Public and private is similar to OOP public and private keywords in classes.

### 3.4.1 Different Linking Types

```

add_library(A ...)
add_library(B ...)
add_library(C ...)

```

### 3.4.2 Public

Here, fmt can be used in the library of A.

```

target_link_libraries(A PUBLIC fmt)

target_link_libraries(C PUBLIC/PRIVATE A)
target_link_libraries(C PUBLIC/PRIVATE A)

```

When A links fmt as PUBLIC, it says that A uses fmt in its implementation, and fmt is also used in A's public API. Hence, C can use fmt since it is part of the public API of A.

### 3.4.3 Private

Using PRIVATE does not make the library available in the target's public API. Instead, it is part of a private API. When B links in spdlog as PRIVATE, it is saying that B uses spdlog in its implementation, but spdlog is not used in any part of B's public API.

```
target_link_libraries(B PRIVATE spdlog)

target_link_libraries(C PUBLIC/PRIVATE B)
```

Any code that makes calls into B would not need to refer directly to anything from spdlog.

In a professional setting, you want to keep certain aspect of your project private. Hence, this option to consider.

### 3.4.4 Interface

In general, used for header-only libraries. That is, libraries where you don't need to compile anything. They do not have any compilation logic in them.

```
add_library(D INTERFACE)
target_include_directories(D INTERFACE {CMAKE_CURRENT_SOURCE_DIR}/include)

# this only links something to the executable, I think. No compilation logic.
```

## 3.5 Different Library Types

### 3.5.1 Library

A binary file that contains information about code. A library cannot be executed on its own. An application utilizes a library.

A library must be built too, if it is used by an executable.

```
cmake --build . --target Library
cmake --build . --target Executable // it is dependent to the Library build!
```

### 3.5.2 Shared

Linux: \*.so  
MacOS: \*.dylib  
Windows: \*.dll

Shared libraries reduce the amount of code that is duplicated in each program that makes use of the library, keeping the binaries small. Shared libraries will however have a small additional cost for the execution. In general the shared library is in the same directory as the executable.

### 3.5.3 Static

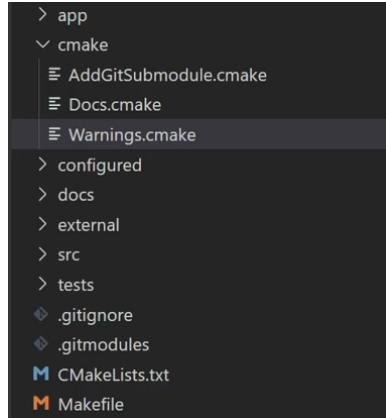
Linux/MacOS: \*.a  
Windows: \*.lib

Static libraries increase the overall size of the binary, but it means that you don't need to carry along a copy of the library that is being used. As the code is connected at compile time there are not any additional run-time loading costs.

## 3.6 Adding Compiler Warnings

You can pull certain warnings for a particular target. We can activate them based on the operating system and on the compiler. We can trigger certain set of compiler checks. In this course, we had two targets: the library target and the executable target.

In the cmake directory, add Warnings.cmake.



```
function(target_set_warnings)
    set(oneValueArgs TARGET ENABLE AS_ERRORS)
    cmake_parse_arguments(
        TARGET_SET_WARNINGS
            # every option and oneValueArgs
            # will be prefixed by
            # TARGET_SET_WARNINGS

        # this function was updated at the end of the course.
        # refactoring at the end of the course was hard to follow.

        "${options}"
        "${oneValueArgs}"
        "${multiValueArgs}"
        ${ARGN})

    if (NOT ${TARGET_SET_WARNINGS_ENABLE})
        message(STATUS "Warnings disabled for: ${TARGET_SET_WARNINGS_TARGET}")
        return()
    endif()
    message(STATUS "Warnings Active for: ${TARGET_SET_WARNINGS_TARGET}")
    message(STATUS "Warnings as Errors: ${TARGET_SET_WARNINGS_AS_ERRORS}")

    set(MSCV_COMPILER
        /WA4
        /permissive-)
```

```

set(CLANG_COMPILER
    -Wall
    -Wextra
    -Wpedantic)

set(GCC_WARNINGS ${CLANG_WARNINGS})

if(${ENABLED_AS_ERRORS})
    set(MSCV_WARNINGS ${MSVC_WARNINGS} /WX) # We need to append to our MSVC compiler
                                                # We append /WX

    set(CLANG_WARNINGS ${CLANG_WARNINGS} -Werror)
                                                # We append -Werror
    set(GCC_WARNINGS ${GCC_WARNINGS} -Werror)
endif()

if(CMAKE_CXX_COMPILER_ID MATCHES "MSVC")
    set(WARNINGS ${MSVC_WARNINGS})
elseif(CMAKE_CXX_COMPILER_ID MATCHES "CLANG")
    set(WARNINGS ${CLANG_WARNINGS})
elseif(CMAKE_CXX_COMPILER_ID MATCHES "GNU")
    set(WARNINGS ${GCC_WARNINGS})
endif()

target_compile_options(${TARGET} PRIVATE ${WARNINGS})
message(STATUS ${WARNINGS})

endfunction(target_set_warnings TARGET)

```

### 3.6.1 Check user's Compiler

```

if(CMAKE_CXX_COMPILER_ID MATCHES "MSVC")
    set(WARNINGS ${MSVC_WARNINGS})
elseif(CMAKE_CXX_COMPILER_ID MATCHES "CLANG")
    set(WARNINGS ${CLANG_WARNINGS})
elseif(CMAKE_CXX_COMPILER_ID MATCHES "GNU")
    set(WARNINGS ${GCC_WARNINGS})
endif()

```

### 3.6.2 Warnings Root CMakeLists Options

We just created a function to check the user's compiler and set compilation warnings. Now create options in the root CMakeFilelists.txt.

```

option(ENABLE_WARNINGS "Enable warnings" ON)
option(ENABLE_WARNINGS_AS_ERRORS "Enable warnings as errors" ON)

```

```

...

```

```

if(ENABLE_WARNINGS)
    include(Warnings) # include the newly created Warnings.cmake file
endif()

```

### 3.6.3 Executable Target Enable Warnings

There are two targets that can have compilation warnings, our library and our executable (our app). We have to add warning conditionals in both of their CMakeLists.txt file.

```

if(${ENABLE_WARNINGS})
    target_set_warnings(
        ${HE}      # this is our executable / app name,
                    # passed as argument in our created warnings function
        ${ENABLE_WARNINGS}
        ${ENABLE_WARNINGS_AS_ERRORS})
endif()

# for the lib CMakeLists.txt

if(${ENABLE_WARNINGS})
    target_set_warnings(
        ${LIBA}      # this is our library target name,
                    # passed as argument in our created warnings function
        ${ENABLE_WARNINGS}
        ${ENABLE_WARNINGS_AS_ERRORS})
endif()

```

You don't have to have warnings as error for all targets, but you should have compilation warnings for all of them.

## 3.7 Sanitizers

Use sanitizers to find memory links or memory problems in your code. Sanitizers are used at runtime. Thus, it happens after compilation. Clang-tidy is a static linter, it finds problems before compilation!

In order, you have clang-tidy before compilation, compiler warnings during compilation and sanitizers at runtime (after compilation).

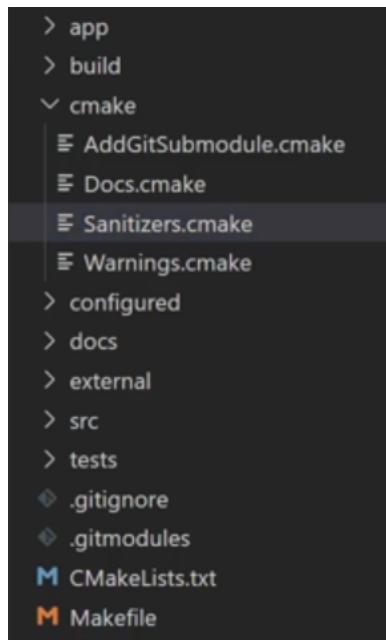
In our root CMakeLists.txt, we add

```

option(ENABLE_SANITIZE_ADDR "Enable warnings" ON)
option(ENABLE_SANITIZE_UNDEF "Enable warnings" ON)

if(ENABLE_SANITIZE_UNDUF OR ENABLE_SANITIZE_ADDR)
    include(Sanitizers) # include a Sanitizers.cmake file
                        # same as Warnings.cmake file
endif()

```



### 3.7.1 Sanitizers.cmake

In cmake directory, we have this file.

```
function(add_sanitizer_flags)
    if(NOT ${ENABLE_SANITIZE_UNDEF} AND NOT ${ENABLE_SANITIZE_ADDR})
        message(STATUS "Sanitizers deactivated")
        return()
    endif()

    if(CMAKE_CXX_COMPILER_ID MATCHES "CLANG" OR CMAKE_CXX_COMPILER_ID MATCHES "GNU")
        add_compile_options("-fno-omit-frame-pointer")
            # This functions adds compiler flags for every target
            # Sanitizers need to run on all of the application
        add_link_options("-fno-omit-frame-pointer")

        if (${ENABLE_SANITIZE_ADDR})
            add_compile_options("-fsanitize=address")
            add_link_options("-fsanitize=address")
        endif()

        if (${ENABLE_SANITIZE_UNDEF})
            add_compile_options("-fsanitize=undefined")
            add_link_options("-fsanitize=undefined")
        endif()

    elseif(CMAKE_CXX_COMPILER_ID MATCHES "MSVC")
        if (${ENABLE_SANITIZE_ADDR})
            add_compile_options("-fsanitize=address")
            add_link_options("-fsanitize=address")
```

```

endif()

if (${ENABLE_SANITIZE_UNDEF})
    message(STATUS "Undefined sanitizer is not implemented for MVSC")
endif()

else()
    message(ERROR "Compiler not supported for Sanitizers")
endif()
endfunction()

```

### 3.7.2 Activate Sanitizers in Root CMakeLists.txt

Call the add\_sanitizer\_flags function from root.

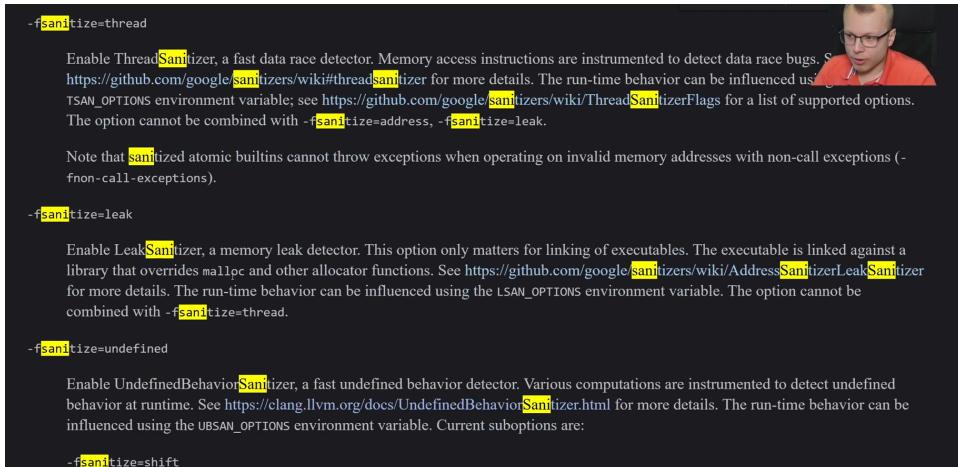
```

if(ENABLE_SANITIZE_ADDR OR ENABLE_SANITIZE_UNDEF)
    include(Sanitizers)
    add_sanitizer_flags()
endif()

```

### 3.7.3 Sanitizer bug example

Going out of bounds, like this, would be caught by clang-tidy, but not by compilers. Thus, Sanitizers help big time. See documentation at [gcc.gnu.org/onlinedocs/Instrumentation-Options.html](http://gcc.gnu.org/onlinedocs/Instrumentation-Options.html).



```

int main() {
    int x[2];
    x[2] = 1337;
}

```

## 3.8 IPO LTO

In release mode, the release build, optimizations are activated to create the best product. C++ has many types of builds for a project, debug modes, performance modes, etc.

In this case, the release mode optimizations are related to the compiler for a better runtime. Yet, these optimizations look at functions separately, not as a chain of functions so to speak (subsequent calls of different functions).

Link Time Optimization (LTO) or Interprocedural Optimization (IPO) [synonyms] is a response to these limits. Using functions from different translation units, the compiler with optimizations, will be able to analyse them subsequently. In other words, the optimized compiler will be able to evaluate if some operations can be cancelled in the function chain.

To use it, we will use a CMake function. All compilers (MSVC, CLANG and GCC) have LTO implemented.

### 3.8.1 LTO Example - Clang

```
--- a.h --- // .h for header file
             // this is genius note-taking!!!

extern int foo1(void);
extern void foo2(void);
extern void foo4(void);

--- a.c --- // .c for source file
             // this is genius note-taking!!!

#include "a.h"

static signed int i = 0;

void foo2(void {
    i = -1;
}

static int foo3(void {
    i = -1;
}

int foo1(void) {
    int data = 0;

    if (i < 0)
        data = foo3();

    data = data + 42;
    return data;
}

--- main.c ---
#include <stdio.h>
#include "a.h"

void foo4(void){
    printf("Hi\n");
```

```

}

int main() {
    return foo1()
}

```

In this example, it is impossible for `i` to be reduced under zero. Since it is impossible, the `foo3` function will never be called. Thus, the compiler cancels an impossible chain and does not generate code for these logically uncallable functions. This is the optimization.

### 3.8.2 IPO LTO in CMake

In your `cmake` directory, create a `LTO.cmake` file. Plus, create a new option in your root `CMakeLists.txt`

```

option(ENABLE_LTO "Enable the link time optimization" ON)

...
if(ENABLE_LTO)
    include(LTO)
endif()

```

### 3.8.3 LTO.cmake

Configuring link time optimization in the new `lto.cmake` file, in the `cmake` directory.

```

function(target_enable_lto TARGET ENABLE)
    if(NOT ${ENABLE})
        endif()

    include(CheckIPOSupported)
    check_ipo_supported(RESULT result OUTPUT output)

    if(result)
        message(STATUS "IPO/LTO is supported!")
        set_property(TARGET ${TARGET} PROPERTY_INTERPROCEDURAL_OPTIMIZATION ${ENABLE})
    else()
        message("WARNING "IPO/LTO is not supported!")

        #PROPERTY... is a predefined variable in modern cmake
    endif()
endfunction(target_enable_lto)

```

### 3.8.4 LTO.cmake Refactoring

The instructor revised the `lto.cmake` file at the end of the course.

```

function(target_enable_lto)          # he passes less arguments here
    set(oneValueArgs TARGET ENABLE)
    cmake_parse_arguments()           # every options and arguments in oneValueArgs
                                    # will be prefixed by LTO_

```

```

# thus, TARGET becomes LTO_TARGET
LTO
"${options}"
"${oneValueArgs}"
"${multiValueArgs}"
${ARGN}

include(CheckIPOSupported)
check_ipo_supported(RESULT result OUTPUT output)

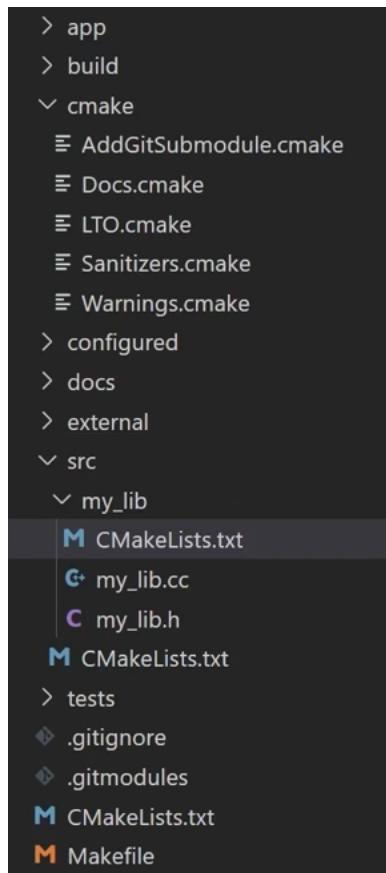
if(result)
    message(STATUS "IPO/LTO is supported: ${LTO_TARGET}")
    set_property(TARGET ${LTO_TARGET} PROPERTY_INTERPROCEDURAL_OPTIMIZATION ${LTO_ENABLE})
else()
    message("WARNING IPO/LTO is not supported! ${LTO_TARGET}")

#PROPERTY... is a predefined variable in modern cmake
endif()
endfunction(target_enable_lto)

```

### 3.8.5 In Our Target CMakeLists.txt

Enabling LTO for our target library, means configuring it in its CMakeLists.txt file.



```
if(${ENABLE_LTO})
    target_enable_lto(${LIBRARY_NAME})
endif()

----- in app -----
----- CMakeLists.txt -----  
  
if(${ENABLE_LTO})
    target_enable_lto(${EXECUTABLE_NAME} ${ENABLE_LTO})
endif()
```

# Chapter 4

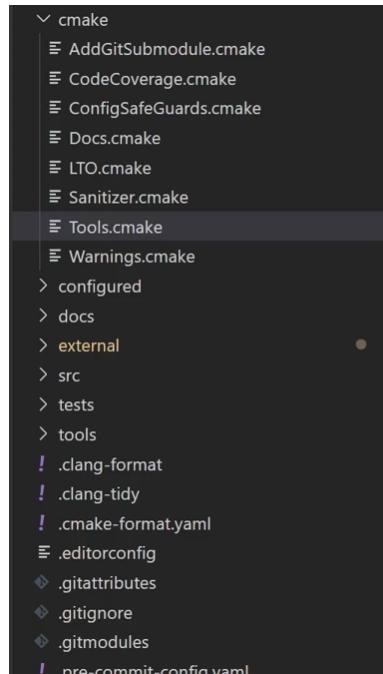
## Cpp Project Template

A project template was included in the course.

### 4.1 Clang Tools

#### 4.1.1 Tools.cmake

This content was seen at the end of the course. The instructor configured the tools in a new Tools.cmake file.



```
function(add_cmake_format_target)
if(NOT ${ENABLE_CMAKE_FORMAT})
    return()
endif()
```

```

endif()
set(ROOT_CMAKE_FILES "${CMAKE_SOURCE_DIR}/CMakeLists.txt")
file(GLOB_RECURSE CMAKE_FILES_TXT "*/CMakeLists.txt")
file(GLOB_RECURSE CMAKE_FILES_C "cmake/*.cmake")
list(
    FILTER
    CMAKE_FILES_TXT
    EXCLUDE
    REGEX
    "${CMAKE_SOURCE_DIR}/(build|external)/*")
set(CMAKE_FILES ${ROOT_CMAKE_FILES} ${CMAKE_FILES_TXT} ${CMAKE_FILES_C})
find_program(CMAKE_FORMAT cmake-format)
if(CMAKE_FORMAT)
    message(STATUS "Added Cmake Format")
    set(FORMATTING_COMMANDS)
    foreach(cmake_file ${CMAKE_FILES})
        list(
            APPEND
            FORMATTING_COMMANDS
            COMMAND
            cmake-format
            -c
            ${CMAKE_SOURCE_DIR}/.cmake-format.yaml
            -i
            ${cmake_file})
    endforeach()
    add_custom_target(
        run_cmake_format
        COMMAND ${FORMATTING_COMMANDS}
        WORKING_DIRECTORY ${CMAKE_SOURCE_DIR})
else()
    message(WARNING "CMAKE_FORMAT NOT FOUND")
endif()
endfunction()

function(add_clang_format_target)
    if(NOT ${ENABLE_CLANG_FORMAT})
        return()
    endif()
    find_package(Python3 COMPONENTS Interpreter)
    if(NOT ${Python_FOUND})
        return()
    endif()
    file(GLOB_RECURSE CMAKE_FILES_CC "*/*.cc")
    file(GLOB_RECURSE CMAKE_FILES_CPP "*/*.cpp")
    file(GLOB_RECURSE CMAKE_FILES_H "*/*.h")
    file(GLOB_RECURSE CMAKE_FILES_HPP "*/*.hpp")
    set(CPP_FILES
        ${CMAKE_FILES_CC}
        ${CMAKE_FILES_CPP}
        ${CMAKE_FILES_H})

```

```

${CMAKE_FILES_HPP})
list(
    FILTER
    CPP_FILES
    EXCLUDE
    REGEX
    "${CMAKE_SOURCE_DIR}/(build|external)/.*")
find_program(CLANGFORMAT clang-format)
if(CLANGFORMAT)
    message(STATUS "Added Clang Format")
    add_custom_target(
        run_clang_format
        COMMAND
        ${Python3_EXECUTABLE}
        ${CMAKE_SOURCE_DIR}/tools/run-clang-format.py ${CPP_FILES}
        --in-place
        WORKING_DIRECTORY ${CMAKE_SOURCE_DIR}
        USES_TERMINAL)
else()
    message(WARNING "CLANGFORMAT NOT FOUND")
endif()
endfunction()

function(add_clang_tidy_to_target target)
get_target_property(TARGET_SOURCES ${target} SOURCES)
list(
    FILTER
    TARGET_SOURCES
    INCLUDE
    REGEX
    ".*.(cc|h|cpp|hpp)")

find_package(Python3 COMPONENTS Interpreter)
if(NOT ${Python_FOUND})
    message(WARNING "Python3 needed for Clang-Tidy")
    return()
endif()

find_program(CLANGTIDY clang-tidy)
if(CLANGTIDY)
    if(CMAKE_CXX_COMPILER_ID MATCHES "MSVC")
        message(STATUS "Added MSVC ClangTidy (VS GUI only) for: ${target}")
        set_target_properties(
            ${target} PROPERTIES VS_GLOBAL_EnableMicrosoftCodeAnalysis
                false)
        set_target_properties(
            ${target} PROPERTIES VS_GLOBAL_EnableClangTidyCodeAnalysis true)
    else()
        message(STATUS "Added Clang Tidy for Target: ${target}")
        add_custom_target(
            ${target}_clangtidy

```

```

COMMAND
    ${Python3_EXECUTABLE}
    ${CMAKE_SOURCE_DIR}/tools/run-clang-tidy.py
    ${TARGET_SOURCES}
    -config-file=${CMAKE_SOURCE_DIR}/.clang-tidy
    -extra-arg-before=-std=${CMAKE_CXX_STANDARD}
    -header-filter="^(src|app)/*.(h|hpp)"
    -p=${CMAKE_BINARY_DIR}
    WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}
    USES_TERMINAL)

endif()
else()
    message(WARNING "CLANGTIDY NOT FOUND")
endif()
endfunction()

```

#### 4.1.2 Tool Directory

The previous tool.cmake file references two python files, one for clang-format and one for clang-tidy. Both are part of the LLVM package, no need to inspect those, it is part of the programs.



#### 4.1.3 Clang-Tidy

A static linting tool for cpp, also known as static analyzers. It checks your code according to set verifications. It is similar to compile warnings, but a linter does not require any compilation to run. It checks to code whenever needed.

You do not need to use the clang compiler to use it.

It is nice to use, because it is the earliest form of warnings. It saves a ton of time if you do not have to compile to have warnings. Early on is best.

Documentation for Clang-Tidy: <https://clang.llvm.org/extra/clang-tidy/>

#### 4.1.4 .clang-tidy file

Specify the checks clang-tidy executes in a .clang-tidy file at the root of your project.

```

Checks: 'clang-analyzer-*,cppcoreguidelines-*,  

modernize-*,bugprone-*,performance-*,readability-non-const-parameter,  

misc-const-correctness,misc-use-anonymous-namespace,  

google-explicit-constructor,-modernize-use-trailing-return-type,  

-bugprone-exception-escape,-cppcoreguidelines-pro-bounds-constant-array-index,  

-cppcoreguidelines-avoid-magic-numbers,-bugprone-easily-swappable-parameters'

```

```

# modernize-* ----- includes all warnings in this category  

# -bugprone=exception-escape # - with a '-' char,

```

```

# specifies one omission in the group

WarningsAsErrors: ''
HeaderFilterRegex: '\(src\|app\)\/*.\.(h\|hpp\)'
AnalyzeTemporaryDtors: false
FormatStyle: none

```

#### 4.1.5 Clang-tidy Target

Configuring a target for clang-tidy checks is made possible in the target's CMakeLists.txt. Here, in my.lib dir for library, our target.

```

if(${ENABLE_CLANG_TIDY})
    add_clang_tidy_to_target(${LIBA}) # this is defined in the Tools.cmake file
endif()

```

#### 4.1.6 Clang-tidy in root CMakeLists.txt

Create an option for clang-tidy in your projects' root CMakeLists.txt

```
option(ENABLE_CLANG_TIDY "Enable to add clang tidy." ON)
```

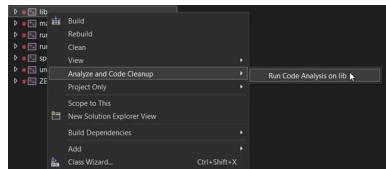
```

set(CMAKE_EXPORT_COMPILE_COMMANDS ON) # this cmake function creates a json file
                                    # clang-tidy needs this json to work
                                    # to differentiate your code from dependencies.

```

#### 4.1.7 Clang-tidy on Microsoft MSVC

To activate it with the microsoft compiler, you need to use the Visual Studio Code GUI only. Some filters, defined in .clang-tidy config file may not work on Windows, but will work on Unix systems.



#### 4.1.8 Clang-Format

An essential professional formatting tool for cpp. Use version 14 at least, 16+ is best. The clang-tidy and clang-format .py files expects at least clang-format 14. Command line arguments change with versions. Check cpp project template .clang-format file in root directory.

Plus, these tools were configured in the root CMakeLists.txt file.

Documentation for Clang-Format: <https://clang.llvm.org/docs/ClangFormat.html>

Install Clang Tools

It's included in the LLVM toolchain, but also installable by apt, brew, winget etc.

<https://github.com/llvm/llvm-project/releases/tag/llvmorg-16.0.0>

#### 4.1.9 Clang-Format Configuration in Tools.cmake

```
function(add_clang_format_target)
    if(NOT ${ENABLE_CLANG_FORMAT})
        return()
    endif()
    find_package(Python3 COMPONENTS Interpreter)
    if(NOT ${Python_FOUND})
        return()
    endif()
    file(GLOB_RECURSE CMAKE_FILES_CC "*/*.cc")
    file(GLOB_RECURSE CMAKE_FILES_CPP "*/*.cpp")
    file(GLOB_RECURSE CMAKE_FILES_H "*/*.h")
    file(GLOB_RECURSE CMAKE_FILES_HPP "*/*.hpp")
    set(CPP_FILES
        ${CMAKE_FILES_CC}
        ${CMAKE_FILES_CPP}
        ${CMAKE_FILES_H}
        ${CMAKE_FILES_HPP})
    list(
        FILTER
        CPP_FILES
        EXCLUDE
        REGEX
        "${CMAKE_SOURCE_DIR}/(build|external)/*"
        # exclude build and external files

    find_program(CLANGFORMAT clang-format)
    if(CLANGFORMAT)
        message(STATUS "Added Clang Format")
        add_custom_target(
            run_clang_format
            COMMAND
                ${Python3_EXECUTABLE}
                ${CMAKE_SOURCE_DIR}/tools/run-clang-format.py ${CPP_FILES}
                --in-place
                WORKING_DIRECTORY ${CMAKE_SOURCE_DIR}
                USES_TERMINAL)
    else()
        message(WARNING "CLANGFORMAT NOT FOUND")
    endif()
endfunction()
```

#### 4.1.10 CMake-Format

A nice-to-have .cmake file formater. It may make your project easier to read, since it standardizes all of your CMakeLists.txt files and . Check .cmake-format.yaml file in root directory.

Plus, these tools were configured in the root CMakeLists.txt file.

```
git clone https://github.com/Kitware/cmake-format.git
cd cmake-format
cmake .
make
make install
```

Install CMake-format,

```
pip install cmake-format # python 3.7+
```

#### 4.1.11 CMake-Format Configuration in Tools.cmake

```
function(add_cmake_format_target)
    if(NOT ${ENABLE_CMAKE_FORMAT})
        return()
    endif()
    set(ROOT_CMAKE_FILES "${CMAKE_SOURCE_DIR}/CMakeLists.txt")
    file(GLOB_RECURSE CMAKE_FILES_TXT "*/CMakeLists.txt")
    file(GLOB_RECURSE CMAKE_FILES_C "cmake/*.cmake")
    list(
        FILTER
        CMAKE_FILES_TXT
        EXCLUDE
        REGEX
        "${CMAKE_SOURCE_DIR}/(build|external)/*")
    set(CMAKE_FILES ${ROOT_CMAKE_FILES} ${CMAKE_FILES_TXT} ${CMAKE_FILES_C})
    find_program(CMAKE_FORMAT cmake-format)
    if(CMAKE_FORMAT)
        message(STATUS "Added Cmake Format")
        set(FORMATTING_COMMANDS)
        foreach(cmake_file ${CMAKE_FILES})
            list(
                APPEND
                FORMATTING_COMMANDS
                COMMAND
                cmake-format
                -c                      # this flag finds the config flag.
                # this is the file.
                ${CMAKE_SOURCE_DIR}/.cmake-format.yaml

                -i                      # specifies to replace the file in-place.
                # Otherwise, like sed command,
                # it only outputs the file
                ${cmake_file})
        endforeach()
        add_custom_target(
            run_cmake_format
            COMMAND ${FORMATTING_COMMANDS}
        )
    endif()
endfunction()
```

```

WORKING_DIRECTORY ${CMAKE_SOURCE_DIR})
else()
    message(WARNING "CMAKE_FORMAT NOT FOUND")
endif()
endfunction()

```

#### 4.1.12 Glob and Exclude Cmake Functions

For clang and cmake-format tools, the tools.cmake file contains globing functions to find all .cmake and CMakeLists.txt files.

```

function(add_cmake_format_target)
    if(NOT ${ENABLE_CMAKE_FORMAT})
        return()
    endif()
    set(ROOT_CMAKE_FILES "${CMAKE_SOURCE_DIR}/CMakeLists.txt")
    file(GLOB_RECURSE CMAKE_FILES_TXT "${Root_CmakeFiles}")
    file(GLOB_RECURSE CMAKE_FILES_C "cmake/*_cmake/")
    list(
        FILTER CMAKE_FILES_TXT
        EXCLUDE REGEX ".+${CMAKE_SOURCE_DIR}/(build|externals)/.*")
}

```

#### 4.1.13 Github Pages

Since Doxygen creates an html you can have a github-pages branch of your project and use github pages to create a webpage your project.

In your .github directory of you project, in a workflows directory, create a documentation.yaml file. This will configure github to update your doc webpage automatically.

```

name: Documentation

on:
  push:
    tags:
      - "*"
    branches: [ main, master ]

jobs:
  build:
    name: Build and publish documentation
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup-python@v2
      - name: Install Docs
        run:
          make install_doc
      - name: prepare
        run:
          make prepare
      - name: configure
        run:
          cmake -H -B build -G "Unix Makefiles" -DCMAKE_BUILD_TYPE=Debug
      - name: building

```

Then, when we push to master, all the jobs will be run by github.

```

name: Documentation

on:
  push:
    tags:
      - "*"
    branches: [ main, master ]

jobs:
  build:
    name: Build and publish documentation
    runs-on: ubuntu-latest

```

```

steps:
- uses: actions/checkout@v2
- uses: actions/setup-python@v2
- name: Install Docs
  run: |
    sudo apt-get install doxygen
    pip install jinja2 Pygments
- name: prepare
  run: |
    make prepare
- name: configure
  run: |
    cmake -H. -Bbuild -G "Unix Makefiles" -DCMAKE_BUILD_TYPE="Debug"
- name: building
  run: |
    cmake --build build --config Debug --target docs -j4
- name: Deploy to GitHub Pages
  uses: Cecillapp/GitHub-Pages-deploy@v3
  env:
    GITHUB_TOKEN: ${{ secrets.GITHUB_TOKEN }}
  with:
    build_dir: ./docs/html  # where is the doxygen documentation

```

It hosts the website on MikaelJG.github.io/repo-name/. Public documentation uploaded online.

## 4.2 Code Coverage

A metric to know how many lines of code were tested by our unit tests. These tools are only available in linux.

```
sudo apt-get install gcovr lcov
```

### 4.2.1 Code Coverage Configuration in root CMakeLists.txt

```

option(ENABLE_COVERAGE "Enable a Code COverage build." ON)

...
if(ENABLE_COVERAGE)
  include(CodeCoverage)
  append_coverage_compiler_flags() # defined in CodeCoverage.cmake file
endif()

```

### 4.2.2 Code Coverage for target library in my lib CMakeLists.txt

Configure code coverage for a specific target in its CMakeLists.txt file.

```

if(ENABLE_COVERAGE)
  set(COVERAGE_MAIN "coverage") # the name of this coverage to run

```

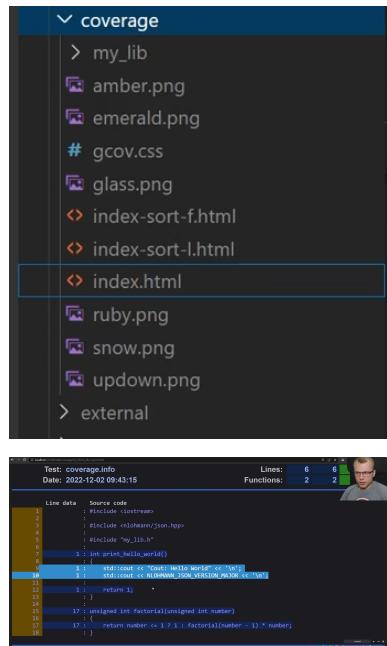
```

set(COVERAGE_EXCLUDES
    "${PROJECT_SOURCE_DIR}/app/*"
    "${PROJECT_SOURCE_DIR}/cmake/*"
    "${PROJECT_SOURCE_DIR}/docs/*"
    "${PROJECT_SOURCE_DIR}/external/*"
    "${PROJECT_SOURCE_DIR}/tests/*"
    "${PROJECT_SOURCE_DIR}/build/*"
    "$usr/include/*")
setup_target_for_coverage_lcov(
    NAME
    ${COVERAGE_MAIN}
    EXECUTABLE
    ${UNIT_TEST_NAME}
    DEPENDENCIES
    ${UNIT_TEST_NAME})
endif()

```

#### 4.2.3 Code Coverage Directory

lcov generates a code Coverage directory, with information in an html.



#### 4.2.4 CodeCoverage.cmake

```

include(CMakeParseArguments)

if(ENABLE_COVERAGE)
    # Check prereqs
    find_program(GCOV_PATH gcov)
    find_program(

```

```

LCOV_PATH
NAMES lcov
    lcov.bat
    lcov.exe
    lcov.perl)
find_program(GENHTML_PATH NAMES genhtml genhtml.perl genhtml.bat)
find_program(GCOVR_PATH gcovr PATHS ${CMAKE_SOURCE_DIR}/scripts/test)
find_program(CPPFILT_PATH NAMES c++filt)

if(NOT GCOV_PATH)
    message(FATAL_ERROR "gcov not found! Aborting...")
endif() # NOT GCOV_PATH

if(CMAKE_C_COMPILER_ID MATCHES "Clang" OR CMAKE_CXX_COMPILER_ID MATCHES
    "Clang")
    set(IS_CLANG TRUE)
else()
    set(IS_CLANG FALSE)
endif()
if(CMAKE_C_COMPILER_ID MATCHES "GNU" OR CMAKE_CXX_COMPILER_ID MATCHES "GNU")
    set(IS_GCC TRUE)
else()
    set(IS_GCC FALSE)
endif()

if(NOT ${IS_CLANG} AND NOT ${IS_GCC})
    message(FATAL_ERROR "Compiler is not gcc/clang! Aborting...")
endif()

set(COVERAGE_COMPILER_FLAGS "-g -O0 -fprofile-arcs -ftest-coverage")
set(CMAKE_CXX_FLAGS_COVERAGE ${COVERAGE_COMPILER_FLAGS} FORCE)
set(CMAKE_C_FLAGS_COVERAGE ${COVERAGE_COMPILER_FLAGS} FORCE)
set(CMAKE_EXE_LINKER_FLAGS_COVERAGE "-lgcov" FORCE)
set(CMAKE_SHARED_LINKER_FLAGS_COVERAGE "" FORCE)
mark_as_advanced(
    CMAKE_CXX_FLAGS_COVERAGE
    CMAKE_C_FLAGS_COVERAGE
    CMAKE_EXE_LINKER_FLAGS_COVERAGE
    CMAKE_SHARED_LINKER_FLAGS_COVERAGE)

if(NOT
    CMAKE_BUILD_TYPE
    STREQUAL
    "Debug")
    message(WARNING "Cov results with non-Debug build may be misleading")
endif()

if(${IS_GCC})
    link_libraries(gcov)
endif()
endif()

```

```

# Defines a target for running and collection code coverage information Builds
# dependencies, runs the given executable and outputs reports. NOTE! The
# executable should always have a ZERO as exit code otherwise the coverage
# generation will not complete.
#
function(setup_target_for_coverage_lcov)
    set(options NO_DEMANGLE)
    set(oneValueArgs BASE_DIRECTORY NAME)
    set(multiValueArgs
        EXCLUDE
        EXECUTABLE
        EXECUTABLE_ARGS
        DEPENDENCIES
        LCOV_ARGS
        GENHTML_ARGS)
    cmake_parse_arguments(
        Coverage
        "${options}"
        "${oneValueArgs}"
        "${multiValueArgs}"
        ${ARGN})

    if(NOT LCOV_PATH)
        message(FATAL_ERROR "lcov not found! Aborting...")
    endif()
    if(NOT GENHTML_PATH)
        message(FATAL_ERROR "genhtml not found! Aborting...")
    endif()

    # Set base directory (as absolute path), or default to PROJECT_SOURCE_DIR
    if(${Coverage_BASE_DIRECTORY})
        get_filename_component(BASEDIR ${Coverage_BASE_DIRECTORY} ABSOLUTE)
    else()
        set(BASEDIR ${PROJECT_SOURCE_DIR})
    endif()

    # Collect excludes (CMake 3.4+: Also compute absolute paths)
    set(LCOV_EXCLUDES "")
    foreach(EXCLUDE ${Coverage_EXCLUDE} ${COVERAGE_EXCLUDES}
          ${COVERAGE_LCOV_EXCLUDES})
        if(CMAKE_VERSION VERSION_GREATER 3.4)
            get_filename_component(
                EXCLUDE
                ${EXCLUDE}
                ABSOLUTE
                BASE_DIR
                ${BASEDIR})
        endif()
        list(APPEND LCOV_EXCLUDES "${EXCLUDE}")
    endforeach()
endfunction()

```

```

list(REMOVE_DUPLICATES LCOV_EXCLUDES)

# Conditional arguments
if(CPPFILT_PATH AND NOT ${Coverage_NO_DEMANGLE})
    set(GENHTML_EXTRA_ARGS "--demangle-cpp")
endif()

# Setup target
add_custom_target(
    ${Coverage_NAME}
    # Cleanup lcov
    COMMAND ${LCOV_PATH} ${Coverage_LCOV_ARGS} --gcov-tool ${GCOV_PATH}
        -directory .
        -b ${BASEDIR} --zerocounters
    # Create baseline to make sure untouched files show up in the report
    COMMAND ${LCOV_PATH} ${Coverage_LCOV_ARGS} --gcov-tool ${GCOV_PATH} -c
        -i -d .
        -b ${BASEDIR} -o ${Coverage_NAME}.base
    # Run tests
    COMMAND ${Coverage_EXECUTABLE} ${Coverage_EXECUTABLE_ARGS}
    # Capturing lcov counters and generating report
    COMMAND
        ${LCOV_PATH} ${Coverage_LCOV_ARGS} --gcov-tool ${GCOV_PATH}
        -directory .
        -b ${BASEDIR} --capture --output-file
        ${Coverage_NAME}.capture
    # add baseline counters
    COMMAND
        ${LCOV_PATH} ${Coverage_LCOV_ARGS} --gcov-tool ${GCOV_PATH} -a
        ${Coverage_NAME}.base -a ${Coverage_NAME}.capture --output-file
        ${Coverage_NAME}.total
    # filter collected data to final coverage report
    COMMAND
        ${LCOV_PATH} ${Coverage_LCOV_ARGS} --gcov-tool ${GCOV_PATH} --remove
        ${Coverage_NAME}.total ${LCOV_EXCLUDES} --output-file
        ${Coverage_NAME}.info
    # Generate HTML output
    COMMAND ${GENHTML_PATH} ${GENHTML_EXTRA_ARGS} ${Coverage_GENHTML_ARGS}
        -o ${Coverage_NAME} ${Coverage_NAME}.info
    # Set output files as GENERATED (will be removed on 'make clean')
    BYPRODUCTS ${Coverage_NAME}.base
        ${Coverage_NAME}.capture
        ${Coverage_NAME}.total
        ${Coverage_NAME}.info
        ${Coverage_NAME} # report directory
    WORKING_DIRECTORY ${PROJECT_BINARY_DIR}
    DEPENDS ${Coverage_DEPENDENCIES}
    VERBATIM # Protect arguments to commands
)

# Show where to find the lcov info report
add_custom_command(
    TARGET ${Coverage_NAME}
    POST_BUILD

```

```

COMMAND ;)

# Show info where to find the report
add_custom_command(
    TARGET ${Coverage_NAME}
    POST_BUILD
    COMMAND ;)
endfunction()

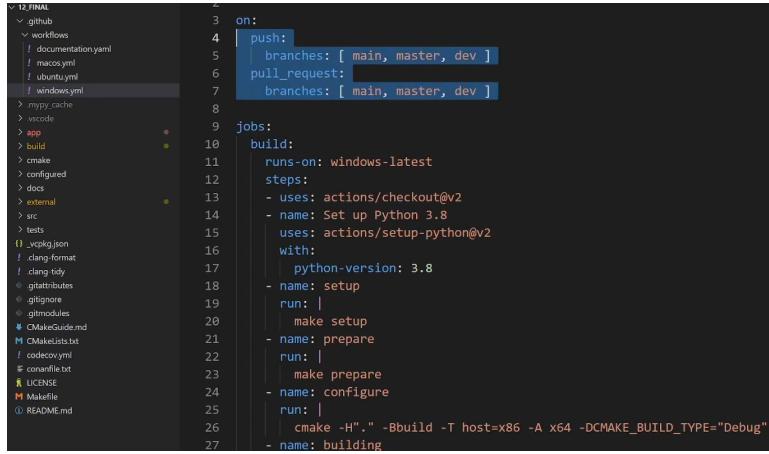
function(append_coverage_compiler_flags)
    set(CMAKE_C_FLAGS
        "${CMAKE_C_FLAGS} ${COVERAGE_COMPILER_FLAGS}"
        PARENT_SCOPE)
    set(CMAKE_CXX_FLAGS
        "${CMAKE_CXX_FLAGS} ${COVERAGE_COMPILER_FLAGS}"
        PARENT_SCOPE)
    message(
        STATUS
            "Appending code coverage compiler flags: ${COVERAGE_COMPILER_FLAGS}"
    )
endfunction()

```

## 4.3 Continuous Integration

### 4.3.1 Github Actions

Yml files instruct github to perform some actions when you push to the main branch. You can have unit testing automatically when you push for example.



```

12_FINAL
  - .github
  - workflows
    - documentation.yml
    - macos.yml
    - ubuntu.yml
    - windows.yml
  - mypy_cache
  - vscode
  - app
  - build
  - cmake
  - configured
  - docs
  - external
  - src
  - tests
  - vcpkg.json
  - clang-format
  - clang-tidy
  - gitattributes
  - gitignore
  - gitmodules
  - CMakeGuide.md
  - CMakeLists.txt
  - codecov.yml
  - coverage.txt
  - coverage
  - Makefile
  - README.md

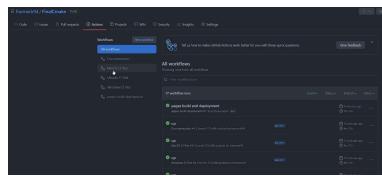
3  on:
4  | push:
5  |   branches: [ main, master, dev ]
6  | pull_request:
7  |   branches: [ main, master, dev ]
8
9  jobs:
10 | build:
11 |   runs-on: windows-latest
12 |   steps:
13 |     - uses: actions/checkout@v2
14 |     - name: Set up Python 3.8
15 |       uses: actions/setup-python@v2
16 |       with:
17 |         python-version: 3.8
18 |     - name: setup
19 |       run: |
20 |         make setup
21 |     - name: prepare
22 |       run: |
23 |         make prepare
24 |     - name: configure
25 |       run: |
26 |         make -H . -Bbuild -T host=x86 -A x64 -DCMAKE_BUILD_TYPE="Debug"
27 |     - name: building

```

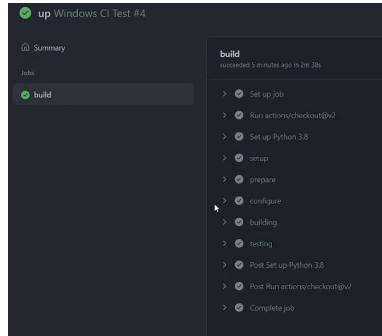
For Ubuntu, you can have automatic code coverage as well. Not possible on macOS and Windows.

### 4.3.2 Github Actions Workflows

Follow your workflows, your different configurated unit tests on Github.



For every job configured, every step is executed on github, with a checkmark if everything has passed.



You can have great icons to see versions of the project passing on your github page.

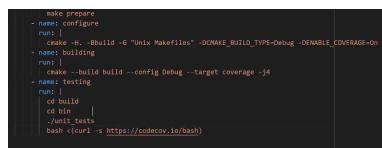


### 4.3.3 Codecov

In Ubuntu, it is possible to upload the code coverage online, for free, automatically with github actions.



It is based on the Ubuntu.yaml configuration file.



## 4.4 Pre-Commit

When you have a github repo with your cpp project, you have many config files. Technically, a user doesn't have to use any of your tools. However, there is a way to force him to do so, with Pre-Commit.

#### 4.4.1 Pre-Commit Installation

Install pre-commit, a python program, with pip.

```
pip install pre-commit
```

#### 4.4.2 Pre-Commit Configuration

In your root directory, create a .pre-commit-config.yaml file.

```
fail_fast: false
repos:
- repo: https://github.com/pre-commit/pre-commit-hooks
  rev: v4.4.0
  hooks:
    - id: check-yaml
    - id: check-json
      exclude: .vscode
    - id: end-of-file-fixer
    - id: trailing whitespace

- repo: https://github.com/pre-commit/mirrors-clang-format
  rev: 'v16.0.3'
  hooks:
    - id: clang-format
      exclude_types: [javascript, json]
```

Then, run \$pre-commit install and \$pre-commit install-hooks commands from the project root.

#### 4.4.3 Pre-Commit in GitHub Actions Workflow Directory

You can run Pre-Commit everytime there is a push on github branches with a pre-commit.yaml in the .github/workflow directory. It will run pre-commit on all pull requests as well!

```
name: pre-commit

on:
  pull_request:
  push:
    branches: [ main, master ]

jobs:
  pre-commit:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v2
      - uses: actions/setup python@v2
      - uses: pre-commit/action@2.0.0
```

## 4.5 Install Command for your Program

We can make our build executable usable on your computer, just like any other tool. To make a transition from our project to our real computer, we will use an install command.

### 4.5.1 Install in root CMakeLists.txt

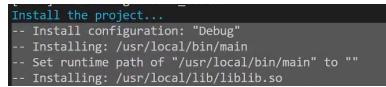
On linux, cmake install automatically on usr/local. On windows, it is in Programs directory.

```
install(TARGETS ${HE}
        EXPORT ${LIBA}
        ARCHIVE DESTINATION lib
        LIBRARY DESTINATION lib
        RUNTIME DESTINATION bin)

install(TARGETS ${LIBA}
        ARCHIVE DESTINATION lib
        LIBRARY DESTINATION lib)

# in build

$ sudo cmake --build . --target install
```



A terminal window showing the output of a cmake install command. The text inside the window reads:

```
Install the project...
-- Install configuration: "Debug"
-- Installing: /usr/local/bin/main
-- Set runtime path of "/usr/local/bin/main" to ""
-- Installing: /usr/local/lib/liblib.so
```

### 4.5.2 Install any CMake Program on the Internet

Logically, you could clone any Cmake project from github, build it and use the executable as a regular program on your machine. This is how pros do it, I guess.

```
Clone the repo
Generate the Cmake project (cmake --build ...)
and
Run the install function we just seen.
```

## 4.6 Shared Libraries

## 4.7 Static Libraries

## 4.8 Cmake GUI

## 4.9 ImGUI

## 4.10 Jason Turner's Template

```
lefticus/cmake_template // Jason Turner 2023 cmake starter pack  
rename "myproject" in the cmake files to use it.
```

### 4.10.1 Lefticus Defaults - ProjectOptions.cmake

```
Address sanitizer  
Undefined behavior sanitizer  
Fuzzing example built  
Procedural optimization IPO (link time optimization)  
Warnings as errors  
Clang-tidy enabled  
CPPcheck enabled  
Options for precompiled headers
```

### 4.10.2 Hardening - Hardening.cmake

```
Hardened compilation // make code safer  
More compilation options / securities.  
  
-fstack-protector  
-fcf-protection  
-fsanitize=undefined // undefined behavior sanitizer  
-fno-sanitize-recover=undefined  
-fsanitize-minimal-runtime  
  
+ debug information
```

## 4.11 Simple Cmake (Modern)

### 4.11.1 Context

Cmake is "a generator of make files", it abstracts away makefile complexity.

```
First, cmake // generate make files  
Second, make // run make files  
Last, ./hello // run the created executable
```

To create an executable of hello.cpp. We usually:

```
:wq // quit vim  
g++ main.cpp -o hello // compile  
../hello // run the executable
```

#### 4.11.2 CMakeLists.txt

With Cmake, we can have:

```
// have cmake installed  
  
cmake_minimum_required(VERSION 3.10)  
set(CMAKE_CXX_STANDARD 17)  
set(CMAKE_CXX_STANDARD_REQUIRED ON)  
  
project(hello VERSION 1.0)  
add_executable(hello main.cpp)
```

Run CMake from the command line, specify a directory.

```
cmake . && make && ./hello
```

#### 4.11.3 Cmake .

Generates all needed make files (Artifacts).

```
~/dev/tutorial/cmake > ls  
CMakeLists.txt  main.cpp  
~/dev/tutorial/cmake > cmake .  
-- The C compiler identification is GNU 10.2.0  
-- The CXX compiler identification is GNU 10.2.0  
-- Detecting C compiler ABI info  
-- Detecting C compiler ABI info - done  
-- Check for working C compiler: /usr/bin/cc - skipped  
-- Detecting C compile features  
-- Detecting C compile features - done  
-- Detecting CXX compiler ABI info  
-- Detecting CXX compiler ABI info - done  
-- Check for working CXX compiler: /usr/bin/c++ - skipped  
-- Detecting CXX compile features  
-- Detecting CXX compile features - done  
-- Configuring done  
-- Generating done  
-- Build files have been written to: /home/ammar/dev/tutorial/cmake  
~/dev/tutorial/cmake > ls  
CMakeFiles  CMakeCache.txt  CMakeLists.txt  Makefile  cmake_install.cmake  main.cpp  
~/dev/tutorial/cmake > █
```

#### 4.11.4 Make

With the MakeFile generated by cmake. Build your binary (the executable) with:

```

~/dev/tutorial/cmake > ls
CMakeFiles CMakeCache.txt CMakeLists.txt Makefile cmake_install.cmake main.cpp
~/dev/tutorial/cmake > vi Makefile
~/dev/tutorial/cmake > make
Scanning dependencies of target hello
[ 50%] Building CXX object CMakeFiles/hello.dir/main.cpp.o
[100%] Linking CXX executable hello
[100%] Built target hello

```

#### 4.11.5 Build folder

```

~/dev/tutorial/cmake > ls
build CMakeLists.txt main.cpp
~/dev/tutorial/cmake > cd build
~/dev/tutorial/cmake/build > ls
~/dev/tutorial/cmake/build > cmake ../
-- The C compiler identification is GNU 10.2.0
-- The CXX compiler identification is GNU 10.2.0
-- Detecting C compiler ABI info
-- Detecting C compiler ABI info - done
-- Check for working C compiler: /usr/bin/cc - skipped
-- Detecting C compile features
-- Detecting C compile features - done
-- Detecting CXX compiler ABI info
-- Detecting CXX compiler ABI info - done
-- Check for working CXX compiler: /usr/bin/c++ - skipped
-- Detecting CXX compile features
-- Detecting CXX compile features - done
-- Configuring done
-- Generating done
-- Build files have been written to: /home/ammar/dev/tutorial/cmake/build
~/dev/tutorial/cmake/build > ls
CMakeFiles CMakeCache.txt Makefile cmake_install.cmake
~/dev/tutorial/cmake/build >

```

#### 4.11.6 Sick CMake Vim plugins combos

See [codevion/cpp2.md](#)

#### 4.11.7 COC - for code completion in nvim

<https://github.com/neoclide/coc.nvim>  
 Jason Turner has this too

#### 4.11.8 Include Header File - CMake Continued

```

target_include_directories(hello PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include)

// Standard is having header files in /include directory!

hello // our target, where to add the stuff from headers
PUBLIC // gives the scope of added stuff from headers.
  // Public, Private or Interface
  // Usage: when you have cmake library, make sure it is seen by #include in files

cmake_minimum_required(VERSION 3.10)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

```

```

project(hello VERSION 1.0)
add_executable(hello main.cpp)
target_include_directories(hello PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include)

```

#### In header file

```

#pragma once
#include <iostream>

class Blah {
public:
    inline void boo() {
        std::cout << "Boo!\n";
    }
};

```

#### 4.11.9 Pragma Once

`#pragma once` is a non-standard directive that serves as an include guard. It ensures that a header file is included only once during the compilation process, regardless of how many times it is referenced.

Placed at the beginning of a header file, it acts as a compiler directive to prevent multiple inclusion. Supported by most compilers, including GCC, Clang, and MSVC.

#### 4.11.10 Glob - Include Many files with CMake

You have at least two options. First, include every files one-by-one in the CMakeList.txt.

```

cmake_minimum_required(VERSION 3.10)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

project(hello VERSION 1.0) // traditional way to include files
add_executable(hello main.cpp Blah.cpp) // added here
target_include_directories(hello PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/include)

```

Cmake discourages this glob method, but it is a more sane options for large projects.

```

file(GLOB_RECURSE SRC_FILES src/*.cpp) // glob everything in src/
add_executable(hello ${SRC_FILES})

```

#### 4.11.11 src Directory - source files

Declaration in the header file `blah.h`.

```

#pragma once

class Blah {
public:

```

```

        void boo(); // declaring function boo in header
}

Definition (implementation) of class Blah in the source files blah.cpp.
#include "blah.h"
#include <iostream>

void Blah::boo() {
    std::cout << "Boo!\n"; // defining function boo in src file
}

```

#### In CMake - Traditionally Added

```
add_executable(hello main.cpp src/Blah.cpp) // added here
```

#### In CMake - Globing

```
file(GLOB_RECURSE SRC_FILES src/*.cpp)
add_executable(hello main.cpp ${SRC_FILES})
```

### 4.11.12 CMake Custom Libraries

Create a lib from some source files:

```
Replace add_executable with add_library
```

```
add_library(mylib STATIC lib/blah.cpp) // create a library
                                         // Statically linked or dynamically linked
```

Then, include it in your main executable

```
target_link_libraries(hello Public mylib)
```

### 4.11.13 Custom Library Implementation - Blah example

```

~/dev/tutorials/cmake > ls
Debug blah src CMakeLists.txt compile_commands.json main.cpp

cmake_minimum_required(VERSION 3.10)
set(CMAKE_CXX_STANDARD 17)
set(CMAKE_CXX_STANDARD_REQUIRED ON)

project(hello VERSION 1.0)

add_library(blah STATIC blah/Blah.cpp)
target_include_directories(blah PUBLIC ${CMAKE_CURRENT_SOURCE_DIR}/blah/include)

// file(GLOB_RECURSE SRC_FILES src/*.cpp)
// now useless, since main.cpp has #include added library

// We are linking our library with our executable directly, with
// target_include_libraries

add_executable(hello main.cpp)
target_link_libraries(hello PUBLIC blah)
```

The target link generates a libblah.a

```
make --build Debug
Scanning dependencies of target blah
[ 25%] Building CXX object CMakeFiles/blah.dir/blah/Blah.cpp.o
[ 50%] Linking CXX static library libblah.a
[ 50%] Built target blah
Scanning dependencies of target hello
[ 75%] Building CXX object CMakeFiles/hello.dir/main.cpp.o
[100%] Linking CXX executable hello
[100%] Built target hello
```

## 4.12 Jason Turner's CMake Template - Options

```
include(cmake/SystemLink.cmake)
include(cmake/LibFuzzer.cmake)
include(CMakeDependentOption)
include(CheckCXXCompilerFlag)

macro(myproject_supports_sanitizers)
    if((CMAKE_CXX_COMPILER_ID MATCHES ".*Clang.*" OR CMAKE_CXX_COMPILER_ID MATCHES ".*GNU.*") AND NOT V
        set(SUPPORTS_UBSAN ON)
    else()
        set(SUPPORTS_UBSAN OFF)
    endif()

    if((CMAKE_CXX_COMPILER_ID MATCHES ".*Clang.*" OR CMAKE_CXX_COMPILER_ID MATCHES ".*GNU.*") AND WIN32
        set(SUPPORTS_ASAN OFF)
    else()
        set(SUPPORTS_ASAN ON)
    endif()
endmacro()

macro(myproject_setup_options)
    option(myproject_ENABLE_HARDENING "Enable hardening" ON)
    option(myproject_ENABLE_COVERAGE "Enable coverage reporting" OFF)
    cmake_dependent_option(
        myproject_ENABLE_GLOBAL_HARDENING
        "Attempt to push hardening options to built dependencies"
        ON
        myproject_ENABLE_HARDENING
        OFF)

    myproject_supports_sanitizers()

    if(NOT PROJECT_IS_TOP_LEVEL OR myproject_PACKAGING_MAINTAINER_MODE)
        option(myproject_ENABLE_IPO "Enable IPO/LTO" OFF)
        option(myproject_WARNINGS_AS_ERRORS "Treat Warnings As Errors" OFF)
        option(myproject_ENABLE_USER_LINKER "Enable user-selected linker" OFF)
        option(myproject_ENABLE_SANITIZER_ADDRESS "Enable address sanitizer" OFF)
        option(myproject_ENABLE_SANITIZER_LEAK "Enable leak sanitizer" OFF)
        option(myproject_ENABLE_SANITIZER_UNDEFINED "Enable undefined sanitizer" OFF)
        option(myproject_ENABLE_SANITIZER_THREAD "Enable thread sanitizer" OFF)
        option(myproject_ENABLE_SANITIZER_MEMORY "Enable memory sanitizer" OFF)
        option(myproject_ENABLE_UNITY_BUILD "Enable unity builds" OFF)
```

```

option(myproject_ENABLE_CLANG_TIDY "Enable clang-tidy" OFF)
option(myproject_ENABLE_CPPCHECK "Enable cpp-check analysis" OFF)
option(myproject_ENABLE_PCH "Enable precompiled headers" OFF)
option(myproject_ENABLE_CACHE "Enable ccache" OFF)
else()
  option(myproject_ENABLE_IPO "Enable IPO/LTO" ON)
  option(myproject_WARNINGS_AS_ERRORS "Treat Warnings As Errors" ON)
  option(myproject_ENABLE_USER_LINKER "Enable user-selected linker" OFF)
  option(myproject_ENABLE_SANITIZER_ADDRESS "Enable address sanitizer" ${SUPPORTS_ASAN})
  option(myproject_ENABLE_SANITIZER_LEAK "Enable leak sanitizer" OFF)
  option(myproject_ENABLE_SANITIZER_UNDEFINED "Enable undefined sanitizer" ${SUPPORTS_UBSAN})
  option(myproject_ENABLE_SANITIZER_THREAD "Enable thread sanitizer" OFF)
  option(myproject_ENABLE_SANITIZER_MEMORY "Enable memory sanitizer" OFF)
  option(myproject_ENABLE_UNITY_BUILD "Enable unity builds" OFF)
  option(myproject_ENABLE_CLANG_TIDY "Enable clang-tidy" ON)
  option(myproject_ENABLE_CPPCHECK "Enable cpp-check analysis" ON)
  option(myproject_ENABLE_PCH "Enable precompiled headers" OFF)
  option(myproject_ENABLE_CACHE "Enable ccache" ON)
endif()

if(NOT PROJECT_IS_TOP_LEVEL)
mark_as_advanced(
  myproject_ENABLE_IPO
  myproject_WARNINGS_AS_ERRORS
  myproject_ENABLE_USER_LINKER
  myproject_ENABLE_SANITIZER_ADDRESS
  myproject_ENABLE_SANITIZER_LEAK
  myproject_ENABLE_SANITIZER_UNDEFINED
  myproject_ENABLE_SANITIZER_THREAD
  myproject_ENABLE_SANITIZER_MEMORY
  myproject_ENABLE_UNITY_BUILD
  myproject_ENABLE_CLANG_TIDY
  myproject_ENABLE_CPPCHECK
  myproject_ENABLE_COVERAGE
  myproject_ENABLE_PCH
  myproject_ENABLE_CACHE)
endif()

myproject_check_libfuzzer_support(LIBFUZZER_SUPPORTED)
if(LIBFUZZER_SUPPORTED AND (myproject_ENABLE_SANITIZER_ADDRESS OR myproject_ENABLE_SANITIZER_THREAD))
  set(DEFAULT_FUZZER ON)
else()
  set(DEFAULT_FUZZER OFF)
endif()

option(myproject_BUILD_FUZZ_TESTS "Enable fuzz testing executable" ${DEFAULT_FUZZER})

endmacro()

macro(myproject_global_options)
  if(myproject_ENABLE_IPO)

```

```

include(cmake/InterproceduralOptimization.cmake)
myproject_enable_ipo()
endif()

myproject_supports_sanitizers()

if(myproject_ENABLE_HARDENING AND myproject_ENABLE_GLOBAL_HARDENING)
    include(cmake/Hardening.cmake)
    if(NOT SUPPORTS_UBSAN
        OR myproject_ENABLE_SANITIZER_UNDEFINED
        OR myproject_ENABLE_SANITIZER_ADDRESS
        OR myproject_ENABLE_SANITIZER_THREAD
        OR myproject_ENABLE_SANITIZER_LEAK)
        set(ENABLE_UBSAN_MINIMAL_RUNTIME FALSE)
    else()
        set(ENABLE_UBSAN_MINIMAL_RUNTIME TRUE)
    endif()
    message("${myproject_ENABLE_HARDENING} ${ENABLE_UBSAN_MINIMAL_RUNTIME} ${myproject_ENABLE_SANITIZER_UNDEFINED}")
    myproject_enable_hardening(myproject_options ON ${ENABLE_UBSAN_MINIMAL_RUNTIME})
    endif()
endmacro()

macro(myproject_local_options)
    if(PROJECT_IS_TOP_LEVEL)
        include(cmake/StandardProjectSettings.cmake)
    endif()

    add_library(myproject_warnings INTERFACE)
    add_library(myproject_options INTERFACE)

    include(cmake/CompilerWarnings.cmake)
    myproject_set_project_warnings(
        myproject_warnings
        ${myproject_WARNINGS_AS_ERRORS}
        ""
        ""
        ""
        "")
    endif()

    if(myproject_ENABLE_USER_LINKER)
        include(cmake/Linker.cmake)
        configure_linker(myproject_options)
    endif()

    include(cmake/Sanitizers.cmake)
    myproject_enable_sanitizers(
        myproject_options
        ${myproject_ENABLE_SANITIZER_ADDRESS}
        ${myproject_ENABLE_SANITIZER_LEAK}
        ${myproject_ENABLE_SANITIZER_UNDEFINED}
        ${myproject_ENABLE_SANITIZER_THREAD})

```

```

${myproject_ENABLE_SANITIZER_MEMORY}

set_target_properties(myproject_options PROPERTIES UNITY_BUILD ${myproject_ENABLE_UNITY_BUILD})

if(myproject_ENABLE_PCH)
  target_precompile_headers(
    myproject_options
    INTERFACE
    <vector>
    <string>
    <utility>
  endif()

  if(myproject_ENABLE_CACHE)
    include(cmake/Cache.cmake)
    myproject_enable_cache()
  endif()

  include(cmake/StaticAnalyzers.cmake)
  if(myproject_ENABLE_CLANG_TIDY)
    myproject_enable_clang_tidy(myproject_options ${myproject_WARNINGS_AS_ERRORS})
  endif()

  if(myproject_ENABLE_CPPCHECK)
    myproject_enable_cppcheck(${myproject_WARNINGS_AS_ERRORS} "" # override cppcheck options
    )
  endif()

  if(myproject_ENABLE_COVERAGE)
    include(cmake/Tests.cmake)
    myproject_enable_coverage(myproject_options)
  endif()

  if(myproject_WARNINGS_AS_ERRORS)
    check_cxx_compiler_flag("-Wl,--fatal-warnings" LINKER_FATAL_WARNINGS)
    if(LINKER_FATAL_WARNINGS)
      # This is not working consistently, so disabling for now
      # target_link_options(myproject_options INTERFACE -Wl,--fatal-warnings)
    endif()
  endif()

  if(myproject_ENABLE_HARDENING AND NOT myproject_ENABLE_GLOBAL_HARDENING)
    include(cmake/Hardening.cmake)
    if(NOT SUPPORTS_UBSAN
      OR myproject_ENABLE_SANITIZER_UNDEFINED
      OR myproject_ENABLE_SANITIZER_ADDRESS
      OR myproject_ENABLE_SANITIZER_THREAD
      OR myproject_ENABLE_SANITIZER_LEAK)
      set(ENABLE_UBSAN_MINIMAL_RUNTIME FALSE)
    else()
      set(ENABLE_UBSAN_MINIMAL_RUNTIME TRUE)
    endif()
  endif()

```

```
endif()
myproject_enable_hardening(myproject_options OFF ${ENABLE_UBSAN_MINIMAL_RUNTIME})
endif()

endmacro()
```