Modern C/C++ with Make

Riccardo Agazzotti

2023

Contents

1 Installing Requirements

Windows

The following examples can be run on windows by installing the following packages.
All the required packages can be found in "chocolatey package manager" which can be installed from https://chocolatey.org/install.
Once installed the package manager, the commands to be run are:

```
choco install cmake
choco install git
choco install doxygen.install
choco install python
choco install make
choco install llvm
```

Linux

On linux (Debian-Ubuntu) the commands that needs to be run are:

```
sudo apt-get update
sudo apt-get upgrade

# Mandatory
sudo apt-get install gcc
sudo apt-get install g++
sudo apt-get install gdb
sudo apt-get install make
sudo apt-get install cmake
sudo apt-get install git
sudo apt-get install doxygen
sudo apt-get install python3
sudo apt-get install python3-pip

# Optional
sudo apt-get install lcov
sudo apt-get install gcovr
sudo apt-get install ccache
sudo apt-get install cppcheck
sudo apt-get install clang-format
sudo apt-get install clang-tidy
```

Visual Studio Code Setup

To setup visual studio code the required extensions are:

```
C/C++ Extension Pack
Coding tool extensions
```

To also have access to vscode json configurations available you need to run the command(in VS command line):

```
C/C++ config: Generate C++ Config files.
```

Please also consider the following aspects you might have to deal with:

- **Optional**: You can install remote WLS to enable visual studio code to talk directly with windows linux sub-system.
- **COMPILER**: With the provided instructions, on windows, you are expected to compile with Visual Studio compiler, so if you have no compiler installed you should consider installing VS Community.
- **Cmake Path**: Cmake path, if not added to system Path environment variable it can be added to the in the setting.json file of VSCODE under the keyword cmake.cmakePath

2 What is CMake

CMake can be seen as a language used for giving the compiler some instruction on how to compile your projects. The results you are obtaining with CMake can also be achieved by typing instructions to the compiler in the command line.

3 How To Use CMake

In this section a possible sequence of steps that can be followed to manage a C/C++ project with CMake is presented.

Initialize CMAKE in VS Code

- **Generate Instruction File** Instruction File should always be called CMakeLists.txt and should be placed inside the project folder.
- **Select Machine** You need to choose which machine you want to use for building your code. Cmake extension in VS code allows you to compile your project either with the compilers that are available on the machine or to open your project inside a remote machine(WSL is treated as a remote machine).
- **Select Compiler** Once the machine has been selected you need to select the compiler by typing:

  ```
  >CMake: Select Kit
  ```

You will be shown a window in which you can choose a compiler among the ones that have been found in your machine.

Please note that choosing WSL as subsystem is equivalent as running CMAKE on a Linux VM.

Generic Steps To Build a Project With CMake

- Create code and CMake File
- mkdir build
- cd build
- cmake .. -> Initializing the main directory of the project on linux and on windows check path variables
- cmake –build . -> The compiler will build the project
- Call the executable you have build to run the code

If you want to use the CMake extension you can just use the play and debug buttons that are on the bottom of VS interface.

Sections of A CMakeLists.txt file

- Version
- Naming The project
- Adding libraries
- Adding Executables
- Linking Executables and libraries
- Install

4 Project: Organizing Folders

Project should be organized in folders, a possible choice would be the following one:

- Project_path: Main project folder
- Project_path-src : folder that contains all the source code.
- Project_path-src-lib_name : Each library should have its own folder in which you can insert their source file, header and c++.

- Project_path-app : The folder that contains the code on of which the executable is compiled

if using a directory structured project, a good practice, is to write a CMakeLists.txt for each folder in which you reference folders with cmake instruction add_subdirectory().

5 CMake: Variables and Options

Variables

CMAKE allows you to declare variables and names around the CMakeLists.txt hierarchy. The way to declare variables is through the syntax:

```
set(varName varValue)
```

Options

Options can be considered variables for branching the code compilation rather than carrying around values. Options can be declared through the syntax:

```
options(varName <help> varValue)
```

Code can be branched with the canonical if-else-endif statement. Options can be changed from cmake command line according to the following syntax:

```
cmake --build . -DMY_VARNAME=[VarValue]
```

MakeFile

Makefiles can be used for doing preprocessing and postprocessing on your working folder.

6 Configure File

Configure files are files used for keeping track of options, version and configurations of your C/C++ code. By defining the version information inside the CMakeLists.txt you can make CMAKE modify some files while it compiles the project by adding those information.
To achieve this result these are the steps:

- Create a configuration folder
- Add a configuration file inside the folder, possibly with the extension *.hpp.in.
- The file must be written in C/C++, you can leave @VAR_NAME@ inside the code, those variables will be filled by CMAKE.

- Add a CMakeLists.txt file to that folder and declare the existence of the configure file, in such way:

```
configure_file(
    "config.hpp.in"
    "${CMAKE_BINARY_DIR}/configured_files/include/config.hpp"
)
```

- Be sure of adding the latest added CMakeLists to the project directory hierarchy and link also the generated *.hpp file as an header to access its content from the project.

Please remind CMAKE_BINARY_DIR contains the main path of the project and its a variable available from CMake scope.

7 Using External Libraries in CMake

Adding and external library as a Git Submodule

CMake allows you to add external libraries. A possible way to do it is by cloning the external library inside your project as a git submodule. A suggested work-flow to follow to use this feature:

- Create a subdirectory (like external)
- Clone the external dependency inside that folder
- Create a cmake folder to hold cmake functions
- Define a cmake function that configures the dependency for us. A possible function definition can be:

```
function(add_git_submodule dir)
    find_package(Git REQUIRED)
    if(NOT EXISTS ${dir}/CMakeLists.txt)
        execute_process(COMMAND ${GIT_EXECUTABLE}
        submodule update --init --recursive -- ${dir}
        WORKING_DIRECTORY ${PROJECT_SOURCE_DIR})
    endif()
    add_subdirectory(${dir})
endfunction(add_git_submodule)
```

  Please not the procedure is less straight forward if the submodule is not a cmake project itself.

- Edit the main CMakeLists.txt by adding a varaible called CMAKE_MODULE_PATH in which holds the path where functions are stored.
- include the file that holds the function definition.

- Link the added library in the App executable CMakeLists.txt .
- Build and Run.

Adding external libraries using Fetch Content

Another way to add an external library to your project is using the already implemented cmake FetchContent function. In the following example the code to include "fmt c++ library" for string formatting.

```
include(FetchContent)
# Cloning the repo
FetchContent_Declare(
    fmt
    GIT_REPOSITORY https://github.com/fmtlib/fmt.git
    GIT_TAG 8.1.1 # Version as git tag
)
# builds the repo and makes it a target for out project
FetchContent_MakeAvailable(fmt)
```

Obviously to use the library in your code you must tell the linker that you want to link the compiled library to your project.

Adding External Library using Conan

Conan is a package manager which hosts many C++ libraries. To include those library in a project is as easy as follows.

- Install conan using the instructions listed in the dependencies section.
- Add a file to the project path named conanfile.txt . The file should have the following structure:

  ```
  [requires]
  <Package name here> (i.e catch2/3.2.1)

  [generators]
  cmake
  ```

- Run: conan install (folder path that holds conanfile.txt) . Please note this command will generate cmake files for building the required dependencies, and place them in the path where the command is performed. A good practice is to run this command from build folder. A possible Makefile instruction to do it:

  ```
  cd build && conan install .. && cd ..
  ```

- Dependencies have to be linked to your project to be used. The name of the libraries can be found in the conan auto-generated cmake files.

External libraries with VCPKG

Using VCPKG rather than conan has some versatility the advantages. Unlike conan that gives you a pre-compiled version of the required library, vcpkg compiles the library locally. Compiling the library on local machine is an advantage because allows you to use an arbitrary compiler (i.e conan compiles windows libraries with MSVS compiler, if you want to use another compiler on windows the pre-compiled library will not link anymore). On the other hand vcpkg installs library somewhere outside the project which is not ideal, the work around to this problem would be adding vcpkg as a git submodule of out project, which is still not ideal.

Dependency Graph

Using graphviz (Unix environment) plots the project dependencies in a graph. The following shell code can be used for the mentioned purpose:

```
cd build
cmake .. --graphviz=graph.dot
dot -Tpng graph.dot -o graphimage.png
```

Shared vs Static Libraries

Library

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

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.

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.

Installing Libraries

Libraries and programs can be installed system wise using cmake. An Example:

```
install(TARGETS ${EXECUTABLE_NAME}
        EXPORT ${LIBRARY_NAME}
        ARCHIVE DESTINATION lib
        LIBRARY DESTINATION lib
        RUNTIME DESTINATION bin)

install(TARGETS ${LIBRARY_NAME}
        ARCHIVE DESTINATION lib
        LIBRARY DESTINATION lib)
```

Using Cmake to install the depenencies requires admin rights,thus this procedure should not be performed from VS Code cmake extension but with the following command in the command line:

```
sudo cmake --build <build path> --target install
```

8 Documentation

Cmake can build the documentation using doxygen each time you rebuild the project just by adding a cmake instruction. To automatically build the documentation you need to:

- Write docstrings
- Create a folder for the documentation
- Create a Doxyfile
- Create a function that adds a custom target. For example:

  ```
  add_custom_target(
          docs
          ${DOXYGEN_EXECUTABLE} ${CMAKE_CURRENT_BIANRY_DIR}/Doxyfile
          WORKING_DIRECTORY ${CMAKE_CURRENT_SOURCE_DIR}/docs)
  ```

- Call the function in the cmake file.

9 Testing

Unit Testing

The unit-test tool used in this project is catch2. It can be set-up as follows:

- Create a test directory.

- Create a cmake file and a main that holds the testing code inside the created folder.
- Assuming to define a cmake variable to enable or disable testing during compilation, cmake has to look like this:

```
if(ENABLE_TESTING)
    set(TEST_MAIN "unit_tests")
    set(TEST_SOURCES "main.cpp")
    add_executable(${TEST_MAIN} ${TEST_SOURCES})
    target_link_libraries(${TEST_MAIN} PUBLIC
                                       ${LIBRARY_NAME}
                                       ${CONAN_CATCH2})

endif()
```

  The code above means that cmake needs to build the code for testing as an executable called unit_test from the code written inside main.cpp, last instruction is to link the required libraries.
- Add the test subdirectory to the main CMakeLists to make sure also this part gets compiled.
- Build the project.

In the example code the code is written to satisfy catch2 framework testing, but the procedure to enable testing is the same for any test framework.

Code Coverage

The tool used in the example works for linux, but the cmake configuration can be considered general. Since this tool works for linux you can tell github to run a ubuntu-VM to produce your output, this can be done if the repo is public otherwise git (up to now) is not giving access to the VMs.

10 Linking types

CMake allows to link libraries using three keyword:

- PUBLIC. Consider the case:

```
target_link_libraires(A PUBLIC B)
target_link_libraries(C PUBLIC A)
```

  In this case C can use B since it is in the public API of A.
- PRIVATE

```
target_link_libraires(A PRIVATE B)
target_link_libraries(C PRIVATE A)
```

In this case C can use just A but not B.

- INTERFACE

  ```
  add_library(D INTERFACE)
  target_include_target_directories(D INTERFACE {CMAKE_CURRETN_SOURCE_DIR}/inc
  ```

  Used for header only libraries, no extra compilation step needed, can be seen as a bunch of data.

11 Package Mangers

The package managers used for installing libraries and dependencies in the examples are:

- chocolatey : install from https://chocolatey.org/install
- conan : install from python pip:

  ```
  /*Windows*/
  pip install conan
  conan user
  conan
  /*Linux*/
  pip install conan
  source ~/.profile
  conan user
  conan
  ```

  N.B you use python pip since conan is written in python but the downloadable libraries are for C and C++.

- vcpkg

12 CMake: Good Practices

- Use variables for library file names and headers.
- CMake allows you to specify external library targets, you do not need to compile everything every time.
- Use header libraries. Some external library allows you to just use header files to run link the library with your project.
- Split Cmake directives into organized files or functions. CMake functions can be stored in files with *.cmake extension. A good practice is to store all the files containing functions in a directory then tell cmake to look there for functions. The Cmake variable that has to be defined is:

```
set(CMAKE_MODULE_PATH "${PROJECT_SOURCE_DIR}/cmake")
```

Please note in the code above functions are stored in a folder called cmake.

- To check the compiler cmake is storing the information in a variable called CMAKE_CXX_COMPILER_ID.
- According to the used compiler, setting the correct variables compiler warnings can be enabled through variable definition.
- If you have some package that are not cross platform you should wrap them in a if-else and these variables can be changed from the command line.

13 Continuos Integration

GitHub Workflows

Github provides some tools for continuous integration from testing pipelines to documentation deployment.

Documentation

In github, if a project is stored as a public repo, it is possible to deploy the documentation in a free-server. The way to deploy into github pages is to add a branch gh-pages than link that branch to the github pages from github repo website thus write a documentation workflow that links the index of your documentation into git hub web server.

Testing

Testing can be performed through github workflows in a on-line VM. To run tests the idea is to build the project and than run the compiled test main.

Code coverage

Code coverage through lcov is done by adding the CMake provided by the lcov library and than run than add the cmake's functions call to into your cmake file. Code coverage can be uploaded to codecov by logging in with a github account.

Extra Tooling

- CLANG Tidy enables certain type of warnings
- CPPCheck enables certain type of warnings
- LTO optimizes linking operations

- CCACHE avoids to re-compile already cached libraries.