Introduction to CMake

Contents

[1. Build System 3](#_Toc188643111)

[Build Files 3](#_Toc188643112)

[2. CMake 7](#_Toc188643113)

[Introduction 7](#_Toc188643114)

[Steps to Build a C++ Project with CMake 8](#_Toc188643115)

[CMake Project with Multiple Files 14](#_Toc188643116)

[CMake Project with Directories 18](#_Toc188643117)

[3. Variables in CMake 22](#_Toc188643118)

[Introduction 22](#_Toc188643119)

[Types of Variables in CMake: 22](#_Toc188643120)

[Setting Language Standards 25](#_Toc188643121)

[Options in CMake 26](#_Toc188643122)

[4. CMake Cache 28](#_Toc188643123)

[Benefits of Using CMake Cache 29](#_Toc188643124)

[Use Cases for CMake Cache Editing 30](#_Toc188643125)

[Tools for Cache Editing 30](#_Toc188643126)

[5. Automatic Generation of Header Files 32](#_Toc188643127)

[Steps to Create an Auto-Generated Header File 32](#_Toc188643128)

[6. Integrate an external library 34](#_Toc188643129)

[Git Submodule 34](#_Toc188643130)

[Adding a Git Submodule 34](#_Toc188643131)

[FetchContent 41](#_Toc188643132)

[CPM (CMake Package Manager) 44](#_Toc188643133)

# Build System

* In software development, a build system is a collection of **tools** and **scripts** that automate the process of transforming source code into executable programs or libraries.

### Key Functions of Build System:

* **Compilation**: Translating source code (written in languages like C++, Java, Python) into machine code that the computer can understand.
* **Linking**: Combining compiled object files with libraries (collections of pre-compiled code) to create the final executable.
* **Dependency Management**: Tracking dependencies between files (if one file changes, which other files need to be recompiled).
* **Building for Different Platforms**: Adapting the build process to different operating systems (Windows, Linux, macOS) and hardware architectures.
* **Automation**: Automating repetitive tasks, such as compiling, linking, and running tests, saving developers time and effort.

### Examples of Build Systems:

* **Make**: A classic and widely used build system, often defined using a **Makefile**.
* **Ninja**: A fast and lightweight build system often used in conjunction with **CMake**.
* **Bazel**: A modern build system designed for large-scale projects, known for its speed and efficiency.
* **Gradle**: A build system commonly used for Java and Android projects.
* **MSBuild**: A build system used by Visual Studio for .NET, C++ and other projects.

## Build Files

* Build files are text files that contain instructions for a build system.
* Build files are the files that actually drive the compilation, linking, and other build-related processes. They contain the specific commands and rules that direct the build system on how to produce the final executable or library from the source code.

### Key Components of Build Files:

* **Source Files**:
  + **List of source files**: This explicitly states which source code files (e.g., .cpp, .c, .java) need to be compiled into object files.
  + **Dependencies**: Specify which source files depend on others. This allows the build system to only recompile files that have been modified or whose dependencies have changed.
* **Include Directories**:
  + **Header file locations**: Tell the compiler where to look for header files (.h, .hpp) that are included in the source code.
    - Example: -I/path/to/includes (using the -I flag, common in C/C++)
* **Libraries**:
  + **External library dependencies**: Specify which external libraries (static or dynamic) the project needs to link against.
  + **Link flags**: Indicate how to link with the libraries (e.g., -lmylibrary in many cases).
* **Compiler and Linker Flags**:
  + **Compiler flags**: Options passed to the compiler to control compilation behavior (e.g., -Wall for warnings, -O2 for optimization, -g for debugging symbols).
  + **Linker flags**: Options passed to the linker to control the linking process (e.g., -shared for creating shared libraries).
* **Custom Build Steps**:
  + **Additional commands**: Define any custom commands that need to be executed during the build process.
  + This could include:
    - Generating files (e.g., using a code generator)
    - Running tests
    - Copying files to specific locations (Pre/Post build)
    - Cleaning up temporary files
* **Build Targets**:
  + **Define the final products**: Specify what the build system should produce (e.g., executable files, libraries).
  + **Dependencies between targets**: Define dependencies between different targets within the project.

### Example:

* Visual Studio Project File (.vcxproj):
  + An XML file specifically used by **MSBuild**, the build system within Visual Studio.
  + These XML files contain all the necessary information for MSBuild to: Compile, Link, generate output and Manage dependencies.

|  |
| --- |
| <Project DefaultTargets="Build"  xmlns="http://schemas.microsoft.com/developer/msbuild/2003">    <ItemGroup>      <ClCompile Include="main.cpp" />      <ClCompile Include="utils.cpp" />    </ItemGroup>    <ItemGroup>      <ClInclude Include="main.h" />      <ClInclude Include="utils.h" />    </ItemGroup>    <PropertyGroup Label="Configuration">      <ConfigurationType>Application</ConfigurationType>      <PlatformToolset>v142</PlatformToolset>    </PropertyGroup>  </Project> |

* Makefile (GNU Make):
  + A file used by the make build automation tool to control the build process.
  + The Makefile specifies how to compile and link the program.
  + It contains rules and dependencies that instruct make on how to build the project.

|  |
| --- |
| CC = gcc  CFLAGS = -Wall -O2  SOURCES = main.c utils.c  OBJECTS = $(SOURCES:.c=.o)  TARGET = myprogram  all: $(TARGET)  $(TARGET): $(OBJECTS)      $(CC) $(CFLAGS) -o $@ $^  clean:      rm -f $(OBJECTS) $(TARGET) |

# CMake

## Introduction

* Imagine you're building a C++ program. You've got your code files (like main.cpp, my\_functions.cpp), and you want to compile them into an executable file.
* The Problem:
  + **Different Operating Systems**: Your code might need to be built on Windows, macOS, or Linux. Each operating system has its own way of compiling and linking code.
  + **Complex Projects**: If your project has many files and dependencies, managing the compilation process manually can become a nightmare.

### What is CMake?

* **CMake** is a tool that helps you build your C++ projects (It is commonly used in C++ projects) easily and consistently, no matter what platform (Windows, macOS, Linux) or compiler (GCC, Clang, MSVC) you are using. It takes care of generating the files that tell your build system how to compile and link your code.
* CMake is **not a compiler** or a build system. Instead, it is a **build system generator**.
* Think of it as a middleman between your project's source code and the build system that turns it into an executable program or library.

### Why Do You Need CMake?

* When you're working on a small C++ program with one or two files, you can compile it using simple commands like…

g++ file1.cpp file2.cpp -o myprogram

* But when your project grows to have many files, external libraries, or needs to work on different platforms, manually managing the build process becomes complex.
* CMake helps you by:
  + Automatically detecting your compiler and platform.
  + Managing dependencies and linking libraries.
  + Generating build files for different tools (e.g., Makefile for Make, .sln for Visual Studio, or files for Ninja).
  + Supporting cross-platform development (Write your CMakeLists.txt once, and build your project on different systems with minimal changes.).

### How Does CMake Work?

* CMake uses a file called CMakeLists.txt where you define your project's structure and how it should be built.
* You write this once, and CMake can then generate the appropriate build files for any platform or tool.
* Imagine you're baking a cake.
  + **Recipe:** Your CMakeLists.txt is like the recipe.
  + **Chef (CMake):** CMake reads the recipe and prepares the ingredients (compiles your code) according to the instructions.
  + **Kitchen (Build System):** The build system (like make) is the kitchen where the actual baking (compiling) happens.

## Steps to Build a C++ Project with CMake

1. **Create a** CMakeLists.txt **file:** This file defines your project configuration, including source files, libraries, and target definitions.
   * File name: CMakeLists.txt (case-sensitive, uppercase C, M, and L).

|  |
| --- |
| cmake\_minimum\_required(VERSION 3.22)  project(CppProject VERSION 1.0.0 LANGUAGES C CXX)  add\_executable(Executable main.cc) |

cmake\_minimum\_required(VERSION 3.22)

* Breakdown:
  + cmake\_minimum\_required: This **command** ensures that the CMake version used to process the CMakeLists.txt file is at least the version specified.
  + VERSION 3.22: This indicates that CMake version 3.22 or newer is required.
* Why It Matters:
  + **Compatibility**: By setting the minimum required version, you ensure that your project will only be built with versions of CMake that support all the features and commands you use in your CMakeLists.txt. This helps avoid issues where older versions of CMake might not recognize certain commands or features.
  + **Clarity**: It provides clarity to anyone building your project about the minimum tool version required, preventing confusion and build failures due to version incompatibility.

project(CppProject VERSION 1.0.0 LANGUAGES C CXX)

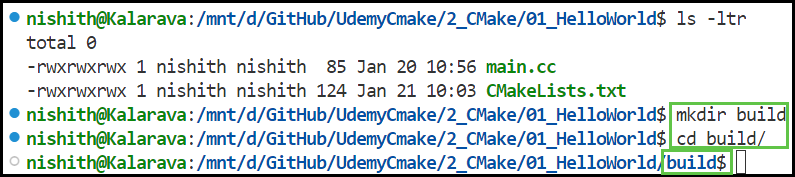
* Breakdown:
  + project: This command sets the name, version, and languages for the project.
  + CppProject: This is the name of your project.
  + VERSION 1.0.0: This specifies the version of your project. In this case, it's set to version 1.0.0.
  + LANGUAGES C CXX: This specifies the programming languages used in your project. Here, it's indicating that the project will use both C and C++.

add\_executable(Executable main.cc)

* Breakdown:
  + add\_executable: This CMake **command** is used to define an executable target. It tells CMake to compile the provided source files into an executable binary.
  + Executable: This is the name of the executable target. You can choose any name for your executable.
  + main.cc: This is the source file that will be compiled to create the executable. You can include multiple source files if needed.

1. **Create a build directory:** This directory will store the build files generated by CMake.

* Create a build directory and cd into the build director.



1. **Configure the project:** Use the cmake <source\_directory> command to configure the project based on your CMakeLists.txt file.

### Configuration Process

* **Reading** CMakeLists.txt:
  + CMake reads the CMakeLists.txt file in the specified source directory (<source\_directory>).
  + This file contains the project's build configuration, including the project name, version, source files, dependencies, and other build instructions.
* **Checking CMake Version**:
  + CMake checks if the version specified by cmake\_minimum\_required is met. If the installed CMake version is lower than required, an error is thrown.
* **Setting Up the Build Environment**:
  + CMake processes the commands in the CMakeLists.txt file to determine the configuration of the build system.
  + This includes defining variables, locating dependencies, setting compiler and linker flags, and determining the output targets (executables, libraries, etc.).
* **Generating Build Files**:
  + Based on the specified generator (e.g., Makefiles, Ninja files, Visual Studio project files), CMake generates the necessary build files in the build directory.
  + These build files contain the exact commands required to compile and link the project.
* **Creating Cache**:
  + CMake creates a CMakeCache.txt file in the build directory. This cache file stores configuration options, paths to dependencies, and other information needed for subsequent builds.
  + The cache helps speed up the build process by reusing information from previous configurations.
* **Setting Up Project Structure**:
  + CMake sets up the build directory with the necessary files and directories needed for the build process.
  + This includes creating directories for object files, binaries, and other intermediate files.
* Example File Structure After Running cmake ..

build/

├── CMakeCache.txt

├── CMakeFiles/

│   ├── CMakeOutput.log

│   ├── CMakeError.log

│   └── (other internal files)

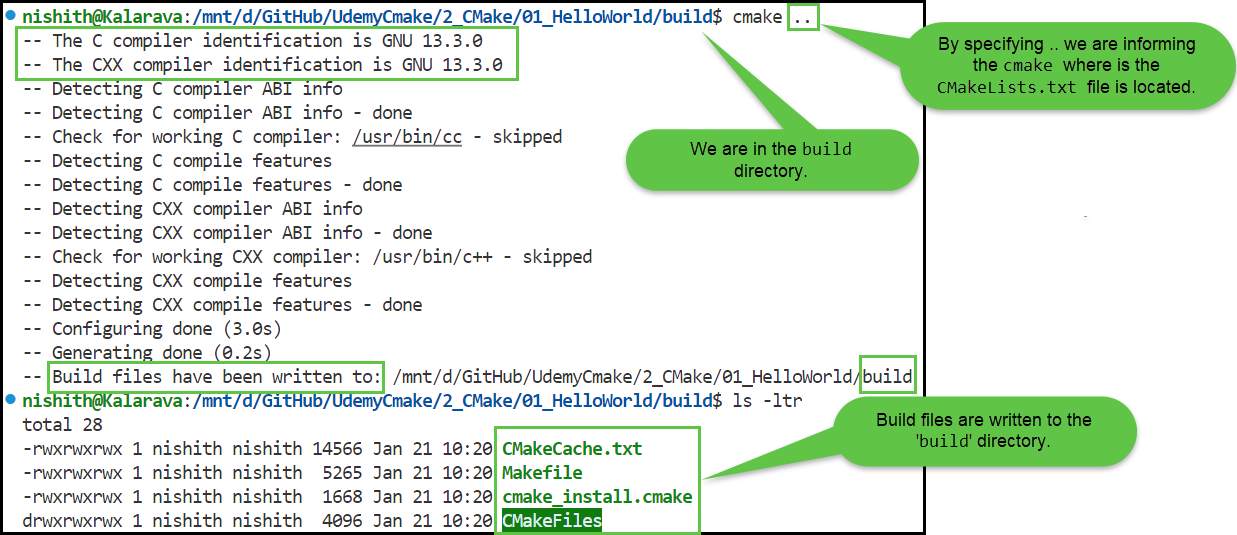
├── Makefile              # For Makefile-based generators

├── cmake\_install.cmake   # Optional install script

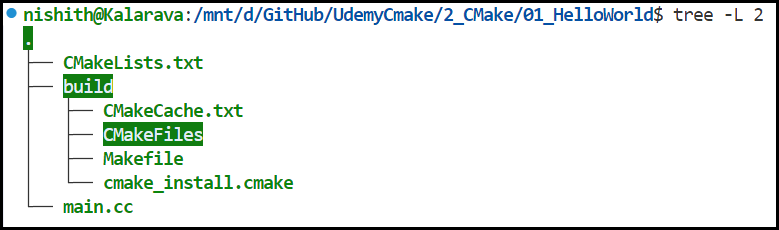
├── build.ninja           # For Ninja-based generators (if Ninja is used)

└── my\_project.sln        # For Visual Studio (if Visual Studio is used)

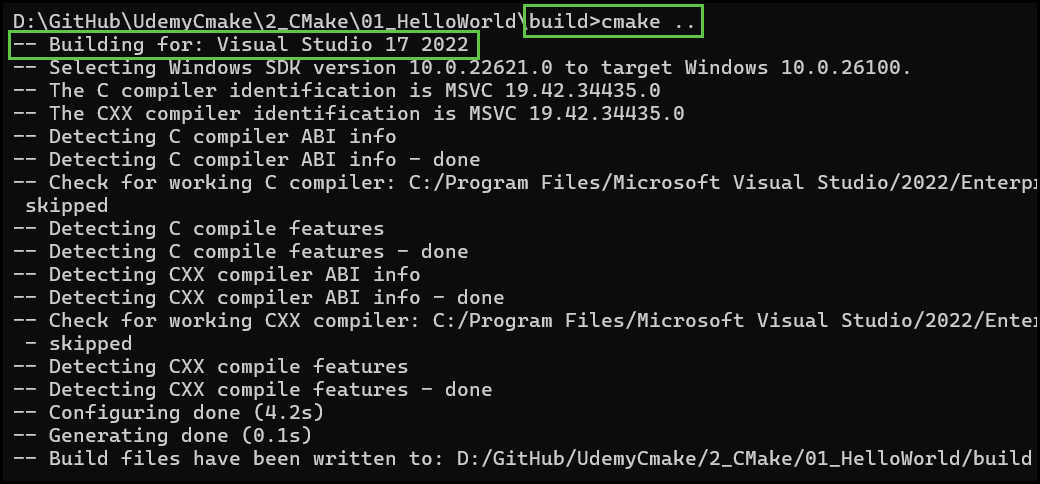
1. For Makefile-based generators.



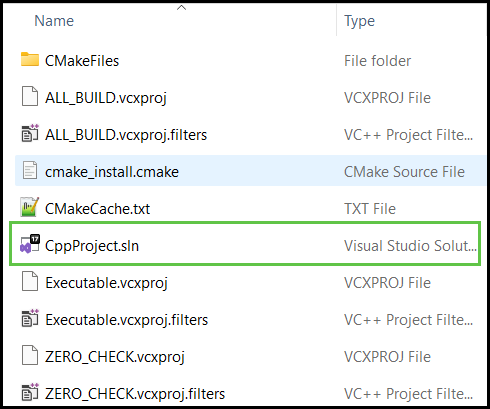
1. The content of build folder for Makefile-based generators.



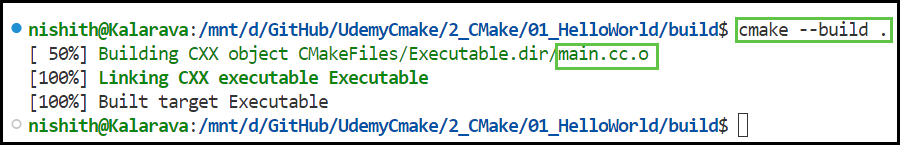
1. For Visual Studio based generators.



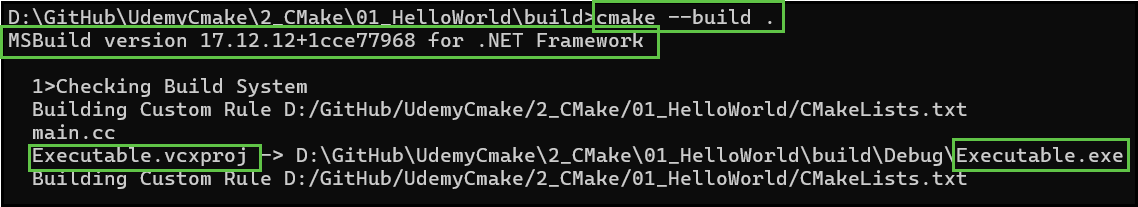
1. The content of build folder for Visual Studio based generators.



1. **Build the project:** Use the cmake --build <build\_directory> command to build the project using the generated build files.
2. For Makefile-based generators…

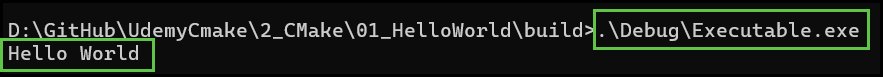


1. For Visual Studio based generators



1. **Run the executable:** Once the build is successful, the executable file will be located in the build directory.





## CMake Project with Multiple Files

### Code Files

1. Create a New Source File and Header File
   * Files:
     + my\_lib.cpp (source file).
     + my\_lib.h (header file).
     + Move the print\_hello\_world function from main.cpp into my\_lib.cpp.
2. Declare and Define the Function
   * Header File (my\_lib.h):
     + Function declaration: void print\_hello\_world();.
   * Source File (my\_lib.cpp):
     + Function definition: The function prints " Hello World from Library " to the console.
3. Include the Library in the Executable
   * In main.cpp:
     + Include the header file using #include "my\_lib.h".
     + Call print\_hello\_world() from the main function.

### **Configuring CMake**

1. **Defining Targets**

* Use the add\_library command to define a new library target.
  + Syntax: add\_library(<target\_name> STATIC <source\_files>).
  + Breakdown:
    - add\_library: This is a CMake command used to define a new library target.
    - <target\_name>: This is the name you assign to the library target. It can be any valid name you choose, and this name will be used to reference the library in other parts of your CMake configuration.
    - STATIC: This keyword specifies that the library being created is a static library. A static library is a collection of object files that are linked into the executable at compile time. It means that all code and data from the library are included in the final executable.
    - <source\_files>: These are the source files that make up the library. You can list one or more source files here. CMake will compile these source files and combine them into a static library.
    - Example: add\_library(my\_lib STATIC my\_lib.cpp).
* Define the executable target using add\_executable.
  + Syntax: add\_executable(<target\_name> <source\_files>).
  + Example: add\_executable(executable main.cpp).

1. **Linking the Library to the Executable**

* Use target\_link\_libraries to link the library to the executable.
  + Syntax: target\_link\_libraries(<target> PUBLIC <library>).
  + Breakdown:
    - target\_link\_libraries: This is a CMake command used to specify libraries or targets that should be **linked** to a given target.
    - <target>: This is the name of the target you are specifying the link libraries for. This could be an **executable** or another **library** target.
    - PUBLIC: This keyword defines the visibility of the linked library. The PUBLIC keyword means that the library will be linked to <target>, and this linkage information will be propagated to any other targets that link to <target>.
    - <library>: This is the name of the library target you are linking to <target>. This can be a **library** defined in the same project or an **external library**.
  + Example: target\_link\_libraries(executable PUBLIC my\_lib).
* Linking ensures the linker combines object files (my\_lib.cpp and main.cpp) during the build process.

### Content of CMakeLists.txt

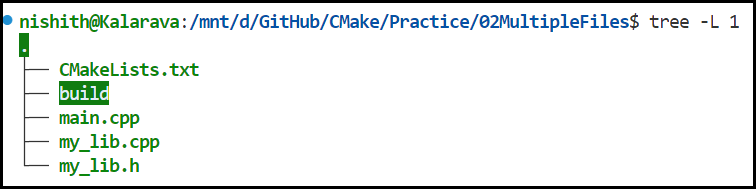
cmake\_minimum\_required(VERSION 3.22)

project(CppProject VERSION 1.0.0 LANGUAGES C CXX)

add\_library(my\_lib STATIC my\_lib.cpp)

add\_executable(Executable main.cpp)

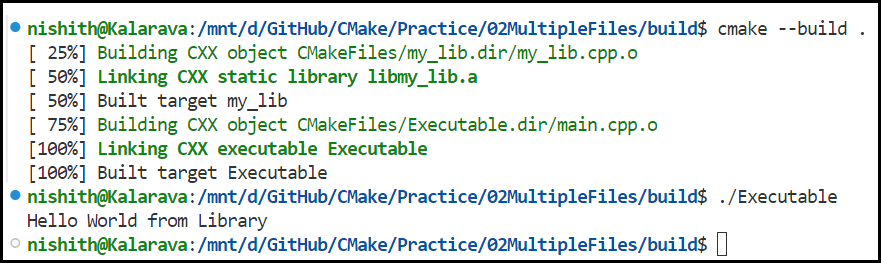
target\_link\_libraries(Executable PUBLIC my\_lib)



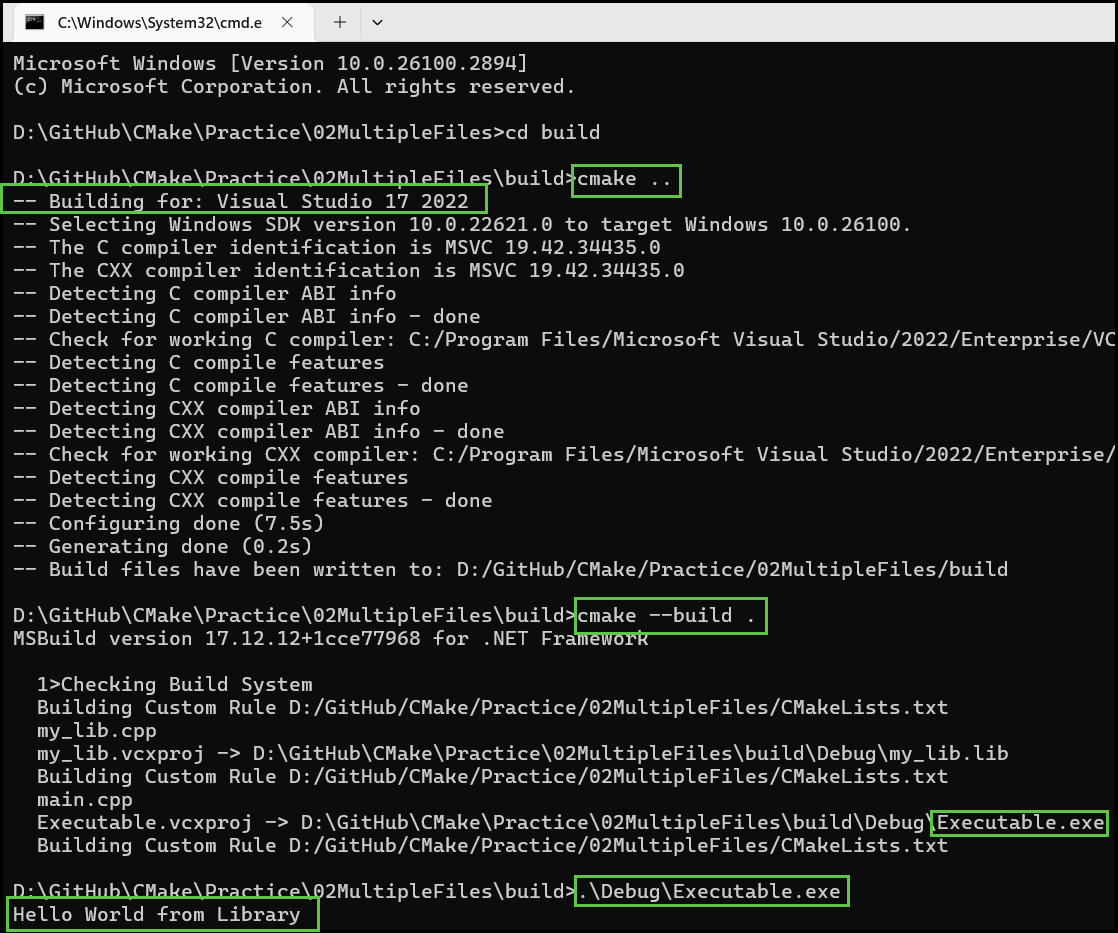
### Building and running the Project with Multiple Files

1. **Configure** and Build
   * Run cmake .. from the build directory to configure the project.
   * Use cmake --build . to build all targets.
2. Run **the** Executable
   * Use the play button in the IDE or execute the binary directly to run the program.

* On Ubuntu



* On Windows



## CMake Project with Directories

* We organize source files (.cpp) and header files (.h) into separate subdirectories like src and include. This improves…
  + Readability and Maintainability.
  + Separation of Concerns or Logical Separation.
  + Easier Navigation.
  + Reduced Clutter.
* Example of a repository with this structure…

|  |
| --- |
| 03MultipleFolders       # Root Directory      ├───Application     # Executable source directory      └───Libraries       # Library source directory          ├───MyLib01     # Subdirectory for a specific library          │   ├───include # Header files          │   └───src     # Source files          └───MyLib02     # Subdirectory for a specific library              ├───include # Header files              └───src     # Source files |

### General Rule for CMakeLists.txt Placement

1. Each Directory That Contributes to the Build Should Have a CMakeLists.txt File:

* The root directory must have a CMakeLists.txt file to define the overall project and include subdirectories.
* Each subdirectory with additional libraries, executables, or targets should have its own CMakeLists.txt file to define its specific logic.
* The general rule is to place a CMakeLists.txt in every directory that represents a buildable unit (library or executable), but not directly inside src or include directories. The parent directory's CMakeLists.txt will handle the source files and header files within those subdirectories.

1. Hierarchy and Inclusion:

* The root CMakeLists.txt includes subdirectories using add\_subdirectory() to delegate build rules to those directories' CMakeLists.txt files.

1. Final Directory and CMakeLists.txt Structure

|  |
| --- |
| 03MultipleFolders/          # Root Directory  ├───CMakeLists.txt          # Main entry point for the project  ├───Application/            # Application directory  │   ├───CMakeLists.txt      # Defines the executable logic  │   └───main.cpp            # Main application code  └───Libraries/              # Libraries directory      ├───CMakeLists.txt      # Includes all library subdirectories      ├───MyLib01/            # Subdirectory for a specific library      │   ├───CMakeLists.txt  # Defines build logic for MyLib01      │   ├───include/        # Header files (no CMakeLists.txt here)      │   │   └───mylib01.h   # Header of MyLib01      │   └───src/            # Source files (no CMakeLists.txt here)      │       └───mylib01.cpp # Source code of MyLib01      └───MyLib02/            # Subdirectory for a specific library          ├───CMakeLists.txt  # Defines build logic for MyLib02          ├───include/        # Header files (no CMakeLists.txt here)          │   └───mylib02.h   # Header of MyLib02          └───src/            # Source files (no CMakeLists.txt here)              └───mylib02.cpp # Source code of MyLib02 |

1. **Best** Practice

* **Do Not Place** CMakeLists.txt **in** src **or** include unless absolutely necessary.
* Instead, handle the build configuration in the CMakeLists.txt file located in the **parent directory** (e.g., MyLib01 or MyLib02).

### Write the Root CMakeLists.txt Files

* Having placed the CMakeLists.txt files in the correct directories (those containing buildable targets), let's write the commands.

1. The content of the CMakeLists.txt file present in the root of the project directory…

cmake\_minimum\_required(VERSION 3.20)

project(CMakeMultipleFolders)

# Add Sub-directories

add\_subdirectory(Application)

add\_subdirectory(Libraries)

* Configure the Libraries Directory

add\_subdirectory(MyLib01)

add\_subdirectory(MyLib02)

1. Set Up Individual Libraries

MyLib01/CMakeLists.txt

add\_library(MyLib01 src/MyLib01.cpp)

target\_include\_directories(MyLib01 PUBLIC include)

MyLib02/CMakeLists.txt

add\_library(MyLib02 src/MyLib02.cpp)

target\_include\_directories(MyLib02 PUBLIC include)

1. Configure the Application Directory

add\_executable(MyApp main.cpp)

target\_include\_directories( MyApp

                            PRIVATE

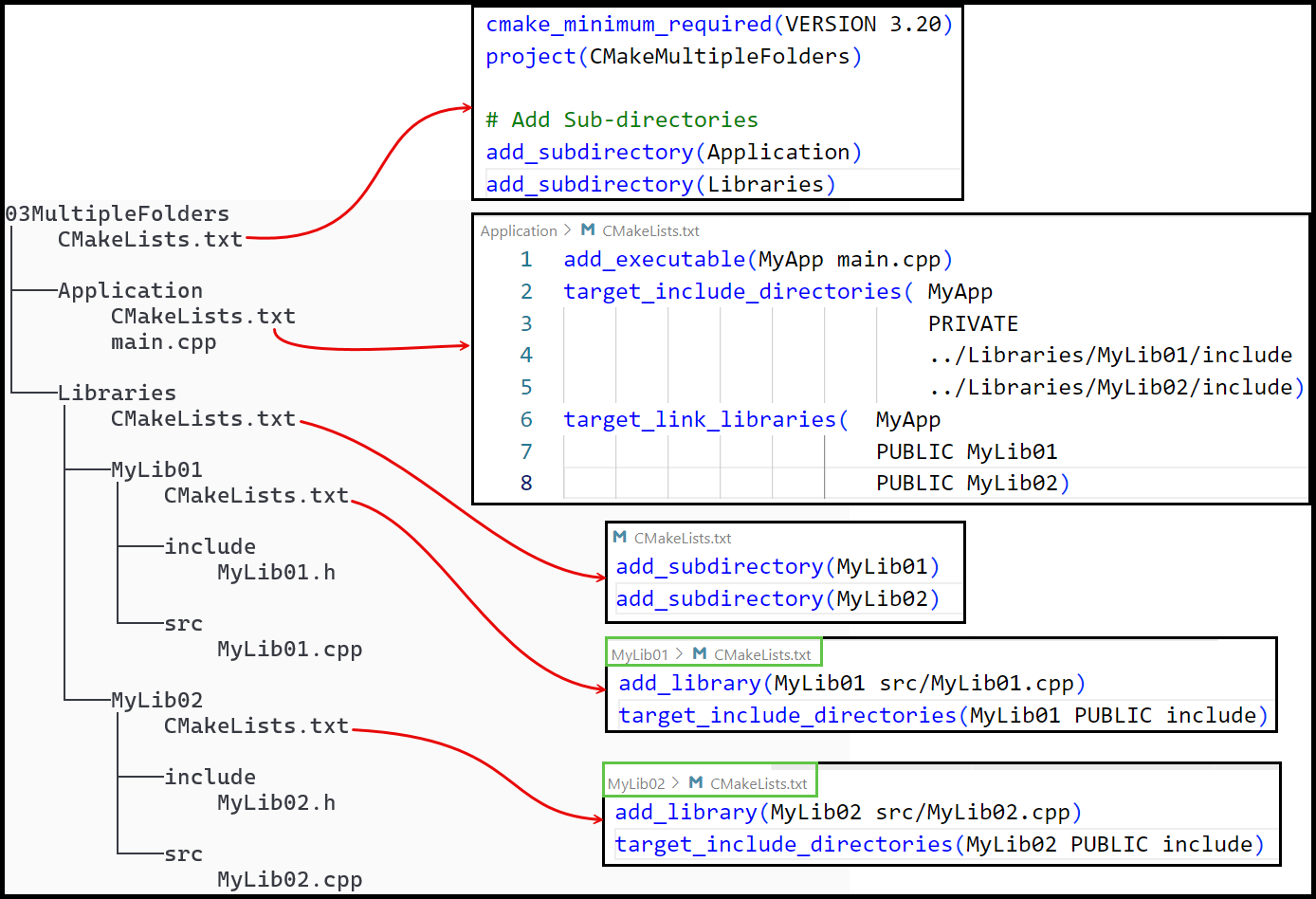
                            ../Libraries/MyLib01/include

                            ../Libraries/MyLib02/include)

target\_link\_libraries(  MyApp

                        PUBLIC MyLib01

                        PUBLIC MyLib02)

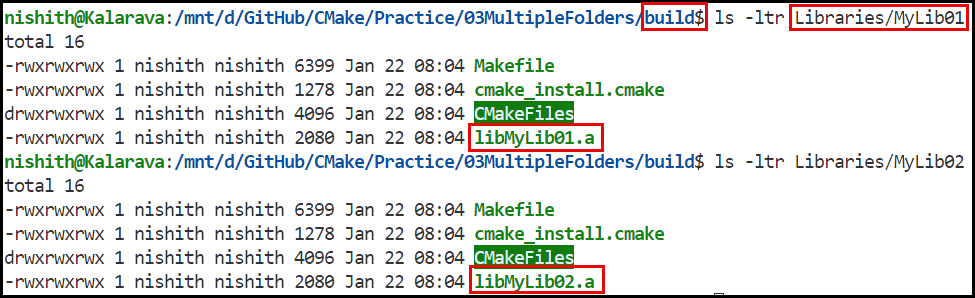


### Build the Project

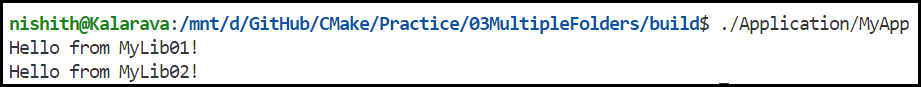
* Create a build directory for the build files and cd into build directory.
* Run the CMake configuration by cmake ..
* Build the project using cmake –build .

### **Outcome**

* The libraries (MyLib01 and MyLib02) will be compiled and are placed under respective folders.



* The application will link with the libraries and run, displaying output as:



# Variables in CMake

## Introduction

* In CMake, variables are used to store and manipulate data that can be referenced throughout your CMakeLists.txt files.
* Variables help manage configuration options, paths, and other parameters, making the build process more flexible and easier to maintain.
* Variables are created using the set function, which takes the variable name and value as arguments.
* Variable names should be uppercase and cannot contain spaces. They can start with underscores.
* Common practice is to define variables in the top-most CMakeLists.txt for better maintainability. Reference them in subdirectories using ${VARIABLE\_NAME} syntax.
* Values can be strings or integers. Booleans are handled differently.

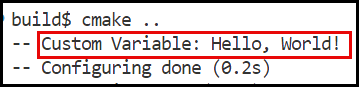
## Types of Variables in CMake:

### **Standard Variables**

* **Definition**: Custom variables you define in your CMakeLists.txt files for use within your project. These variables are not persistent and exist only during the current CMake run.
* **Purpose**: To store and reuse values specific to your project's configuration.
* **Example**:

set(MY\_VAR "Hello, World!")

message(STATUS "Custom Variable: ${MY\_VAR}")

* **Key Points**:
  + These variables are useful for project-specific logic or temporary calculations.
  + Can be redefined at any point during the execution of the CMakeLists.txt.

### Predefined Variables

* **Definition**: Built-in variables provided by CMake that convey information about the system, compiler, and other build-related configurations.
* **Purpose**: To simplify build configurations by exposing relevant system and compiler details.
* **Example**:

# CMAKE\_SOURCE\_DIR: The root directory of the project source tree.

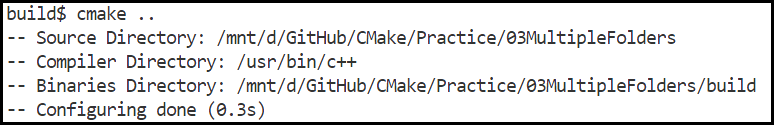
message(STATUS "Source Directory: ${CMAKE\_SOURCE\_DIR}")

# CMAKE\_CXX\_COMPILER: Path to the C++ compiler being used.

message(STATUS "Compiler Directory: ${CMAKE\_CXX\_COMPILER}")

# CMAKE\_BINARY\_DIR: The directory where CMake is generating the build files.

message(STATUS "Binaries Directory: ${CMAKE\_BINARY\_DIR}")



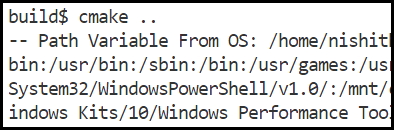
* **Key Points**:
  + These are predefined by CMake for use in build scripts.
  + You cannot redefine these variables (though you can override some of their effects by changing other settings).

### Environment Variables

* **Definition**: Variables inherited from the system environment where CMake is executed.
* **Purpose**: To access or manipulate system-wide information during the build process.
* Example:

# Accesses the system's PATH environment variable.

message(STATUS "Path Variable From OS: $ENV{PATH}")



* **Key Points**:
  + These are not defined in CMake but are instead passed to CMake from the operating system.
  + You can set environment variables within CMake using set(ENV{VAR\_NAME} value).

### Cache Variables

* **Definition**: Persistent variables stored in the CMakeCache.txt file. These retain their values across CMake runs unless explicitly modified.
* **Purpose**: To manage configuration options and parameters for the build system.
* Example:

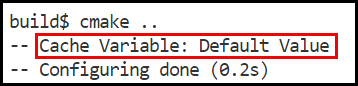
# Creates a cache variable MY\_CACHE\_VAR

# with a default value of "Default Value".

set(MY\_CACHE\_VAR "Default Value" CACHE STRING

                 "Description of the variable")

message(STATUS "Cache Variable: ${MY\_CACHE\_VAR}")



* **Key Points**:
  + Useful for passing options and settings between multiple CMake runs or for user-specified options.
  + Typically used for options that affect build behavior, e.g., CMAKE\_BUILD\_TYPE, CMAKE\_INSTALL\_PREFIX.

### **Summary Table**

| **Type** | **Defined By** | **Scope** | **Persistence** | **Example** |
| --- | --- | --- | --- | --- |
| Standard Variables | User in CMakeLists.txt | Current CMake run | No | set(MY\_VAR "Hello") |
| Predefined Variables | CMake itself | Current CMake run | No | ${CMAKE\_SOURCE\_DIR} |
| Environment Variables | System OS | Current CMake run | External | $ENV{PATH} |
| Cache Variables | User/CMake in cache file | Across multiple CMake runs | Yes | set(MY\_CACHE\_VAR "Value" CACHE) |

## Setting Language Standards

* There are CMake commands used to configure the C++ standard and its behaviour in a CMake-based build system.

set(CMAKE\_CXX\_STANDARD 17)

* This sets the C++ standard to C++17. CMake will attempt to configure the compiler to use the specified C++ standard.
* If the compiler does not support the specified standard, it **might still proceed without errors or warnings**, potentially using an earlier standard instead.

set(CMAKE\_CXX\_STANDARD\_REQUIRED ON)

* This makes the C++ standard requirement mandatory. If the specified C++ standard (C++17 in this case) is not supported by the compiler, CMake will terminate with an error.
* **If** OFF **(default)**: The compiler may fall back to an older C++ standard if C++17 is not supported, without stopping the build.
* **If** ON: The **build will fail** if C++17 support is unavailable, enforcing strict adherence to the specified standard.

set(CMAKE\_CXX\_EXTENSIONS OFF)

* This disables compiler-specific language extensions (often non-standard or proprietary) and ensures that the compiler uses strictly standard C++ features.
* **If** OFF: Non-standard extensions like -std=gnu++17 might be enabled. For example, GNU compilers default to gnu++17 instead of c++17, which allows some GCC-specific extensions.
* **If** ON: Only strictly standard C++ is allowed (e.g., -std=c++17).

## Options in CMake

* **Options** are a way to define configurable variables that users can set when configuring a project.
* Options are typically used for enabling or disabling features, selecting build types, or configuring other project-specific settings. They are Boolean-like variables (ON or OFF) that can be toggled by the user.

### Defining Options

* Options in CMake are defined using the option() command.
* Syntax:

option(<variable> "Description" [initial value])

* + <variable>: The name of the variable.
  + "Description": A description of what the option controls, displayed in CMake GUIs or command-line messages.
  + [initial value]: The default value, either ON or OFF. If omitted, the default is OFF.

### Example

option(BUILD\_TESTS "Enable building of tests" ON)

* Name: BUILD\_TESTS
* Description: "Enable building of tests"
* Default Value: ON
* If a user runs CMake with this project, they can toggle the BUILD\_TESTS option in the command line: Using -D flag, e.g., cmake -DBUILD\_TESTS=OFF, its value can be used as a variable in the CMake script.

if(BUILD\_TESTS)

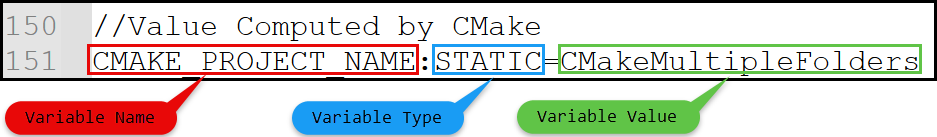
    add\_subdirectory(tests)

endif()

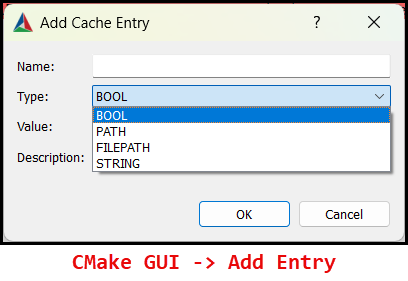
# CMake Cache

* The CMake Cache is an important feature in CMake that stores configuration settings and values across runs, making the build process more efficient and consistent.

### What is CMake Cache?

* **Purpose**:
  + The CMake Cache is used to store persistent configuration values and variables that are set during the **initial configuration process**.
  + These values can be reused in subsequent runs, ensuring consistency and reducing the need for reconfiguration.
  + Any change to CMakeLists.txt requires a reconfiguration step.
  + Reconfiguring large projects can take several minutes.
* **Location**:
  + The CMake Cache is stored in a file named CMakeCache.txt, which is located in the build directory.
* **Content**:
  + The CMakeCache.txt file contains key-value pairs, where each entry consists of a variable name, type, and value.
* Example entry in CMakeCache.txt:
  + Explanation of Each Component:
    - **Variable Name**: CMAKE\_PROJECT\_NAME
      * This is the name of the variable, which in this case represents the project name defined in your CMakeLists.txt file.
    - **Type**: STATIC
      * This indicates the **scope** or **persistence** of the variable in the CMake cache.
      * STATIC means that the variable is a **cache variable** and its value is stored persistently in CMakeCache.txt. It will not be automatically overwritten unless explicitly changed or the cache is cleared.
    - **Value**: CMakeMultipleFolders
      * This is the actual value assigned to the variable. In this case, the project name is CMakeMultipleFolders.

### Other Types in CMake Cache:

* Apart from STATIC, other types you might encounter are:
  + BOOL: For Boolean values (ON/OFF).
  + PATH: For directory or file paths.
  + FILEPATH: Specifically for file paths.
  + STRING: For general string values.
  + INTERNAL: Internal variables used by CMake that are **not meant for direct user modification**.

## Benefits of Using CMake Cache

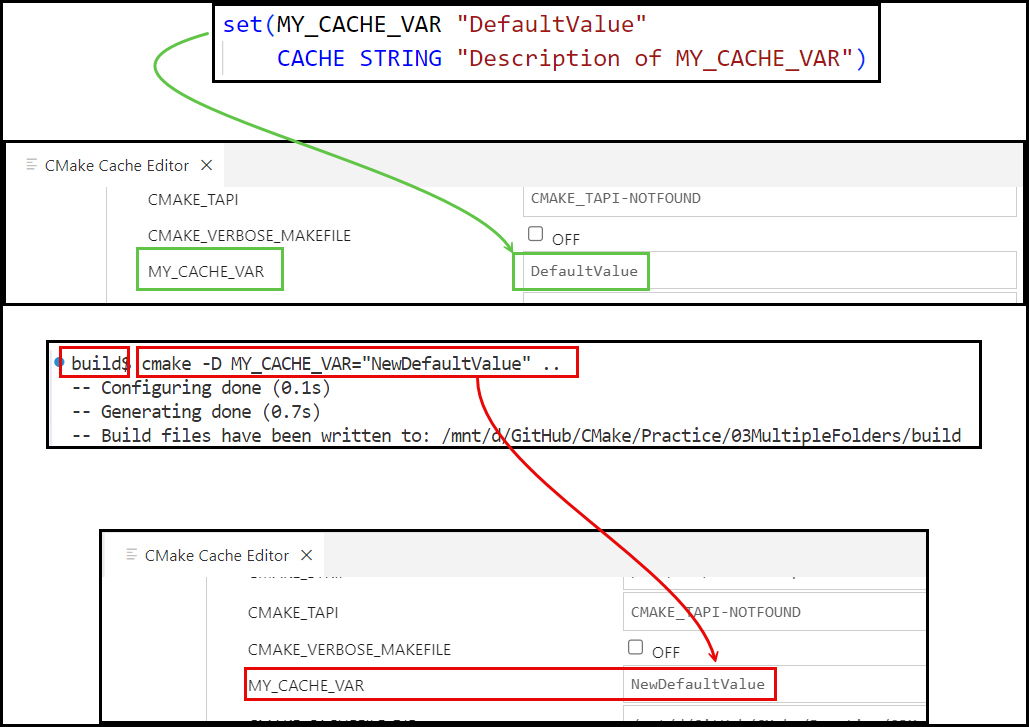
* **Efficiency**:
  + Speeds up the configuration process by reusing previously determined values and settings.
  + Reduces the need to repeatedly search for dependencies or configure paths.
* **Consistency**:
  + Ensures that the same configuration values are used across multiple builds, minimizing the risk of configuration drift.
  + Helps maintain a stable and reproducible build environment.
* **Customization**:
  + Allows users to easily modify configuration settings without altering the CMakeLists.txt files.
  + Users can edit CMakeCache.txt directly or use tools like ccmake or cmake-gui to make changes.

## **Use Cases for CMake Cache Editing**

* **When to Use**:
  + To quickly turn options or variables ON/OFF for temporary builds.
  + To avoid reconfiguring projects, which saves time in large builds.
* **Examples**:
  + Temporarily disable a target by setting its variable to OFF.
  + Test changes to variables without modifying CMakeLists.txt or reconfiguring.

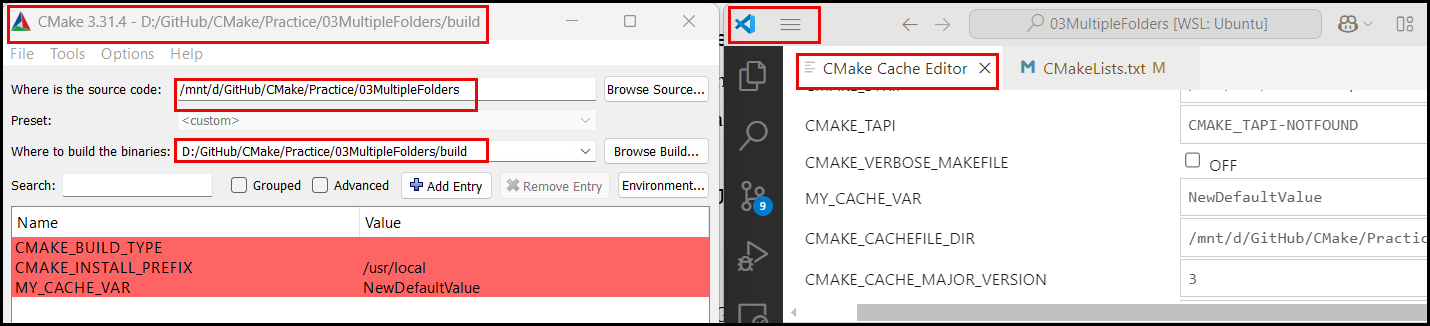
## Tools for Cache Editing

1. Using CMake Command-Line Options
   * You can pass new values for variables when running CMake from the command line.
   * Example: cmake -D<Variable>=<Value> <Path to Source>



1. Manually Editing the CMakeCache.txt File
   * Open the CMakeCache.txt file in a text editor.
   * Locate the variable and modify its value.
   * Example:

CMAKE\_PROJECT\_NAME:STRING=NewProjectName

1. Using CMake GUI
   * If you have CMake GUI installed, you can open the GUI and load your build directory.
   * Modify the cache variables in the user-friendly interface.
   * You can use VS Code's CMake Tools Extension to edit cache.

# Automatic Generation of Header Files

* **Purpose of the Header File**:
  + Stores project information, such as:
    - Project name.
    - Version details (major, minor, patch).
  + These values can be defined as macros or constants for use in the project.
* **CMake Feature**:
  + Automatically generates the header file by replacing placeholders with actual project values.

## **Steps to Create an Auto-Generated Header File**

* **Define Project Details in CMakeLists.txt**:
  + Example:

project(CppProject VERSION 1.0.0 LANGUAGES CXX)

* + - VERSION specifies:
      * Major: Breaking changes. (1)
      * Minor: New features (backward compatible) (0).
      * Patch: Bug fixes (0).
* **Create a Template Header File**:
  + File: config.hpp.in
  + Contains placeholders for project details:

|  |
| --- |
| static constexpr std::string\_view project\_name = "@PROJECT\_NAME@";  static constexpr std::string\_view project\_version = "@PROJECT\_VERSION@";  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@}; |

* + Placeholders:
    - @PROJECT\_NAME@, @PROJECT\_VERSION\_MAJOR@, etc.
  + CMake replaces these placeholders with actual values.
* **Use** configure\_file() **in CMakeLists.txt**:
  + Command:

|  |
| --- |
| configure\_file( "config.hpp.in"                  "${CMAKE\_BINARY\_DIR}/configured\_files/include/config.hpp"                  ESCAPE\_QUOTES) |

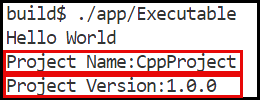
* + Input: config.hpp.in.
  + Output: A processed header file (config.hpp) with actual values.
  + Destination: Inside CMAKE\_BINARY\_DIR.
* **Include the Generated File in Your Project**:
  + Add the generated directory to the include path:

|  |
| --- |
| include\_directories(${CMAKE\_BINARY\_DIR}/configured\_files/include)  add\_executable(${EXECUTABLE\_NAME} "main.cc") |

* + Use the generated header file in the code:

|  |
| --- |
| #include "config.hpp"  int main() {      std::cout << "Hello World\n";      std::cout << "Project Name:" << project\_name << '\n';      std::cout << "Project Version:" << project\_version << '\n';      return 0;  } |

* **Build and Verify**



# Integrate an external library

## Git Submodule

* In Git, a submodule is a way to include another Git repository as a subdirectory within your main project. It's like a nested Git repository within your project.

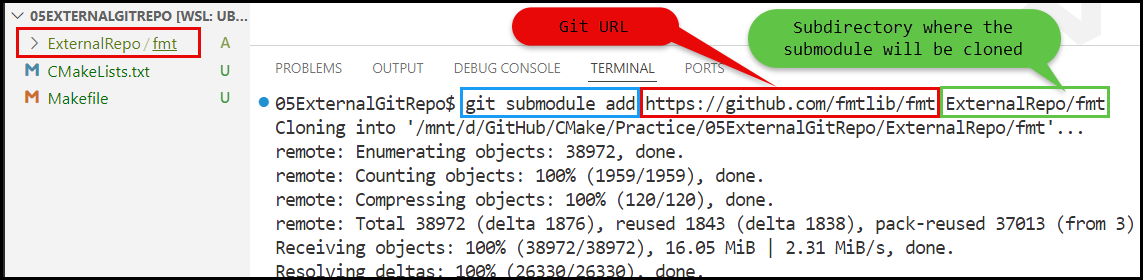
### Key Points:

* **Separate Repositories**: Submodules maintain their own independent version history. Changes made within the submodule are tracked separately from the main project.
* **Specific Commit**: When you add a submodule, you're essentially pinning it to a specific commit of the external repository. This ensures that your project always uses that particular version of the submodule.
* .gitmodules **File**: A special file named .gitmodules is created in your main project to store information about the submodules, such as their URLs and commit hashes.

## Adding a Git Submodule

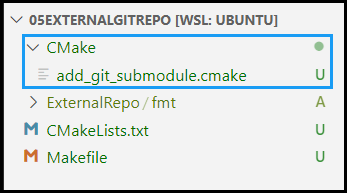
### Clone the external Git submodule

* Since you are adding a Git submodule, it's essential that you already have an existing **Git repository for your main project**.
* To add a sub module, say <https://github.com/fmtlib/fmt> which is an open-source formatting library providing a fast and safe alternative to C stdio and C++ iostreams, use the “git submodule add” command as shown below…

git submodule add https://github.com/fmtlib/fmt ExternalRepo/fmt

* The command “git submodule add” creates a file called **.gitmodules** in the directory where the .git folder is present.

### **Setting Up** CMake

* CMake Directory Structure:
  + Create a CMake folder in the project.
  + Inside this folder, create a add\_git\_submodule.cmake file to automate submodule handling.
  + Creating a dedicated CMake folder and using a separate add\_git\_submodule.cmake file for submodule management provides a structured, maintainable, and reusable approach to handling Git submodules within your CMake projects. It promotes better organization, reduces the risk of errors, and simplifies the process of managing external dependencies.
* CMake Function Implementation:
  + Define the add\_git\_submodule function.
  + Why Define an add\_git\_submodule Function in CMake?
    - **Centralized Logic**: By defining a function like add\_git\_submodule, you encapsulate all the logic related to adding and managing Git submodules within a single, well-defined unit.
    - **Reusability**: This function can be reused across multiple projects or even within different parts of the same project.

|  |
| --- |
| function(add\_git\_submodule directory)  find\_package(Git REQUIRED)  if (NOT EXISTS ${directory}/CMakeLists.txt)  execute\_process(  COMMAND ${GIT\_EXECUTABLE} submodule update --init --recursive  WORKING\_DIRECTORY ${PROJECT\_SOURCE\_DIR} )  endif()  add\_subdirectory(${directory})  endfunction(add\_git\_submodule) |

* Function Definition:
  + function(add\_git\_submodule directory): This line defines a function named add\_git\_submodule that takes a single argument directory. This argument represents the path to the submodule directory.
* Finding Git:
  + find\_package(Git REQUIRED): This line ensures that the Git executable is available. If Git is not found, the configuration will fail with an error. The REQUIRED keyword makes it mandatory.
* Checking for CMakeLists.txt:
  + if (NOT EXISTS ${directory}/CMakeLists.txt): This condition checks if the CMakeLists.txt file does not exist in the specified submodule directory. If the file does not exist, it proceeds with the commands inside the if block.
* Executing Git Command:

execute\_process(

COMMAND ${GIT\_EXECUTABLE} submodule update --init --recursive

WORKING\_DIRECTORY ${PROJECT\_SOURCE\_DIR} )

* + execute\_process: This command runs a process during the CMake configuration.
  + COMMAND ${GIT\_EXECUTABLE} submodule update --init --recursive: This part of the command uses the Git executable to update and initialize the submodule. The --init option initializes the submodule, and the --recursive option ensures that any nested submodules are also updated and initialized.
  + WORKING\_DIRECTORY ${PROJECT\_SOURCE\_DIR}: This specifies the working directory for the command. Here, it sets the working directory to the project's root directory (PROJECT\_SOURCE\_DIR).
* Adding the Subdirectory:
  + add\_subdirectory(${directory}): This command adds the specified directory as a subdirectory to the project. This means CMake will process the CMakeLists.txt file within that directory as part of the current build.
* Ending the Function:
  + endfunction(add\_git\_submodule): This marks the end of the function definition.

### **Incorporating** Submodules

* Include the add\_git\_submodule function in the root CMakeLists.txt.

set(CMAKE\_MODULE\_PATH "${PROJECT\_SOURCE\_DIR}/cmake/")

include(add\_git\_submodule)

add\_git\_submodule(ExternalRepo/fmt)

* set(CMAKE\_MODULE\_PATH "${PROJECT\_SOURCE\_DIR}/cmake/")
  + This line tells CMake where to look for custom modules (files ending in .cmake).
  + PROJECT\_SOURCE\_DIR is a CMake variable that points to the root directory of your project.
  + CMAKE\_MODULE\_PATH is a variable that specifies the search paths for custom CMake modules.
  + By setting CMAKE\_MODULE\_PATH to ${PROJECT\_SOURCE\_DIR}/cmake, you instruct CMake to search for modules within the cmake subdirectory of your project's root directory.
* include(add\_git\_submodule)
  + This line tells CMake to include the add\_git\_submodule.cmake file, which is assumed to be located in the cmake subdirectory of your project.
  + This inclusion makes the functions and variables defined within add\_git\_submodule.cmake available for use in your main CMakeLists.txt file.
* add\_git\_submodule(ExternalRepo/fmt)
  + This line calls the add\_git\_submodule function (defined in add\_git\_submodule.cmake) to add a Git submodule.
  + ExternalRepo/fmt specifies the relative path within your project where the submodule should be cloned.

### Linking the Library

* If we are adding 3rd party library in Visual Studio, we follow these steps…

1. Add "Additional Include Directories"
   * This tells the compiler where to look for the header files of the library you want to use.
   * **Purpose**: Without this, the compiler will fail to find the necessary headers when you include them in your code (e.g., #include <core.h>).
2. Add "Additional Library Directories"
   * This tells the linker where to look for the library files (e.g., .lib files on Windows).
   * **Purpose**: The linker uses this to find the actual implementation of the library functions during the linking phase.
3. Add "Additional Dependencies"
   * This tells the linker the names of the libraries to link against.
   * **Purpose**: Specifies which specific library files (e.g., fmt.lib) the linker should use.

* To do the same in CMake, we do…

|  |
| --- |
| # Include the fmt library's include directories  target\_include\_directories(${EXECUTABLE\_NAME} PRIVATE ExternalRepo/fmt/include)  # Link the fmt library to the executable  target\_link\_libraries(${EXECUTABLE\_NAME} PRIVATE fmt) |

1. Add "Additional Include Directories"

|  |
| --- |
| target\_include\_directories(${EXECUTABLE\_NAME} PRIVATE ExternalRepo/fmt/include) |

* **CMake**: This specifies include directories for your target. It is **similar to the "Additional Include Directories"** field in Visual Studio.

1. Add "Additional Library Directories"

* In Visual Studio, the "Additional Library Directories" field specifies paths where the linker looks for the library files (e.g., .lib files).
* CMake: In CMake, there is **no direct equivalent** for "Additional Library Directories" because target\_link\_libraries automatically handles both specifying the library and finding its location if the library is properly configured (e.g., using find\_package() or add\_subdirectory()).

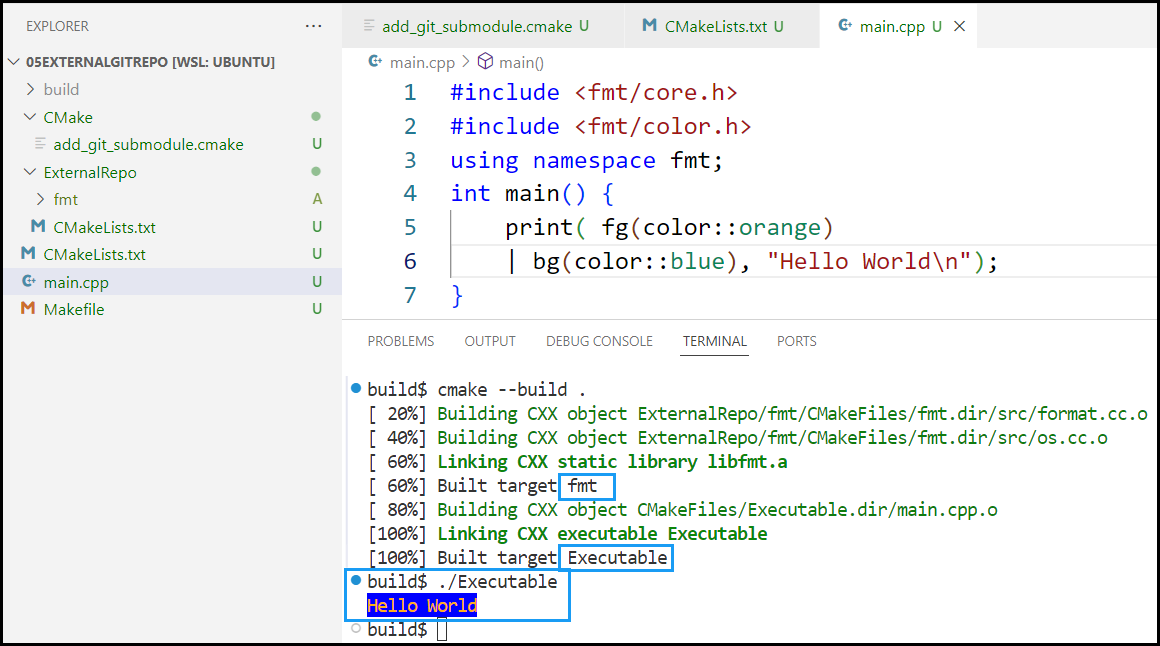
1. Add "Additional Dependencies"

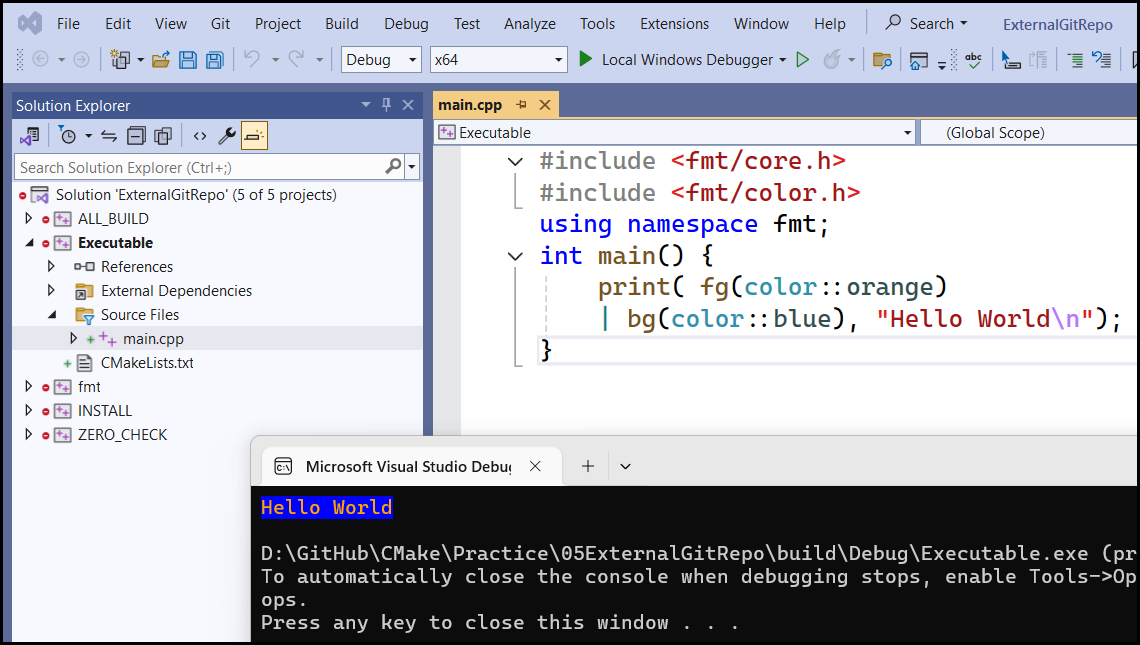
target\_link\_libraries(${EXECUTABLE\_NAME} PRIVATE fmt)

* **CMake**: This specifies the libraries to link against your target (e.g., fmt in this case). It is **similar to the "**Additional Dependencies**"** field in Visual Studio, where you explicitly specify the libraries, your project depends on (e.g., fmt.lib).
* Include the library in your source code:

|  |
| --- |
| #include <fmt/core.h>  #include <fmt/color.h>  using namespace fmt;  int main() {      print( fg(color::orange) | bg(color::blue), "Hello World\n");  } |

### Build and Run





## FetchContent

* FetchContent is a CMake module that simplifies the process of downloading and integrating external dependencies into your CMake projects.
* It provides a convenient way to manage third-party libraries, source code, or other resources that your project relies on.

### Key Features:

* **Downloads dependencies**: FetchContent can download dependencies from various sources, including Git repositories, archives (ZIP, tar.gz), and local directories.
* **Integrates with your project**: Once downloaded, FetchContent makes the dependencies available to your CMake project, allowing you to use their targets, include directories, and libraries.
* **Easy to use**: The module provides a straightforward interface with commands like FetchContent\_Declare() and FetchContent\_MakeAvailable().

### How it Works:

1. Include the FetchContent Module:
   * Use include(FetchContent) in your CMakeLists.txt file to enable the FetchContent module.
   * This step is similar to importing a library or module in programming languages. It allows you to use FetchContent-related commands in your CMake script.
2. Declare the Dependency:
   * Use FetchContent\_Declare() to specify details about the dependency:
     + The source URL or Git repository.
     + The specific version or branch you want to fetch.
     + Any other optional settings like source directory or additional configurations.
   * Example:

FetchContent\_Declare(

    fmt # This declares a dependency named fmt.

    GIT\_REPOSITORY https://github.com/fmtlib/fmt # Git URL

    GIT\_TAG 11.1.2   # Defines the specific to fetch.

    GIT\_SHALLOW TRUE # Fetches only the specified version

    )

* + The GIT\_SHALLOW TRUE fetches only the specified version (or a single commit) instead of the entire Git history, reducing the amount of data downloaded and saving time.

1. Make the Dependency Available:
   * Use FetchContent\_MakeAvailable() to download the dependency and prepare it for integration into your project.
   * This step often ensures the dependency's source directory is added to your include paths and the necessary libraries are prepared for linking.
   * Example:

FetchContent\_MakeAvailable(fmt)

1. Link the Target Library:
   * Use target\_link\_libraries() to link your project's target with the downloaded library.
   * This ensures the dependency is properly linked during the build process.

target\_link\_libraries(${EXECUTABLE\_NAME} PUBLIC fmt)

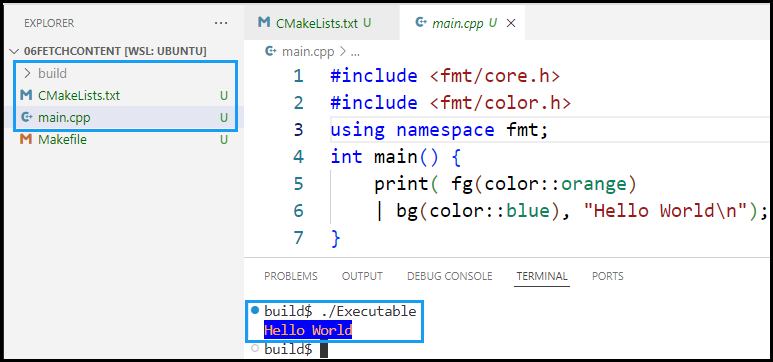
1. Include Headers in Source File:

|  |
| --- |
| #include <fmt/core.h>  #include <fmt/color.h>  using namespace fmt;  int main() {      print( fg(color::orange)      | bg(color::blue), "Hello World\n");  } |

### Key Points

* When you use FetchContent in CMake, the dependency is downloaded to a location managed by CMake, typically inside a build directory.
* When you link the library using target\_link\_libraries, CMake automatically:
  + Adds the appropriate include paths to your target.
  + Links the fmt library to your target for both compilation and linking stages.
  + You don’t need to manually include the downloaded path (<build\_directory>/\_deps/fmt-src/include) because CMake handles it behind the scenes when you link the target. So no need to use “target\_include\_directories()”

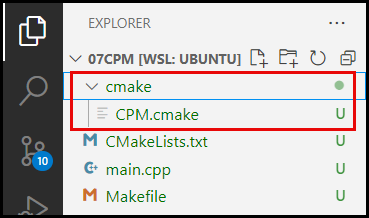
1. Build and Run:



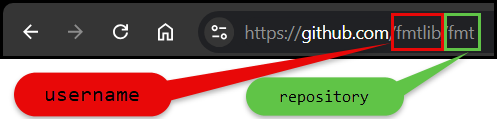
## CPM (CMake Package Manager)

* **CPM** is an unofficial CMake Package Manager. **CPM** is a powerful tool that simplifies the process of finding, downloading, and integrating external C/C++ libraries into your CMake projects.
* It simplifies the process of including external libraries in a C++ project.
* Uses FetchContent under the hood but provides a more user-friendly interface.

### Steps to Use CPM:

1. Download **the** CPM.cmake **File**:
   * Visit the official GitHub repository (<https://github.com/cpm-cmake/CPM.cmake>) of CPM to download the CPM.cmake file.
   * Navigate to the "Releases" section and download the latest CPM.cmake file.
   * Create a folder named ‘cmake’ in your project and copy the downloaded file into the cmake directory of your project.
2. Modify Your CMake Configuration:
   * Add the line include("${CMAKE\_SOURCE\_DIR}/cmake/CPM.cmake").
   * CMAKE\_SOURCE\_DIR is a built-in variable that holds the absolute path to the top-level directory of your CMake project.
3. Define and download a Git repository:
   * Use the CPMAddPackage command as shown…

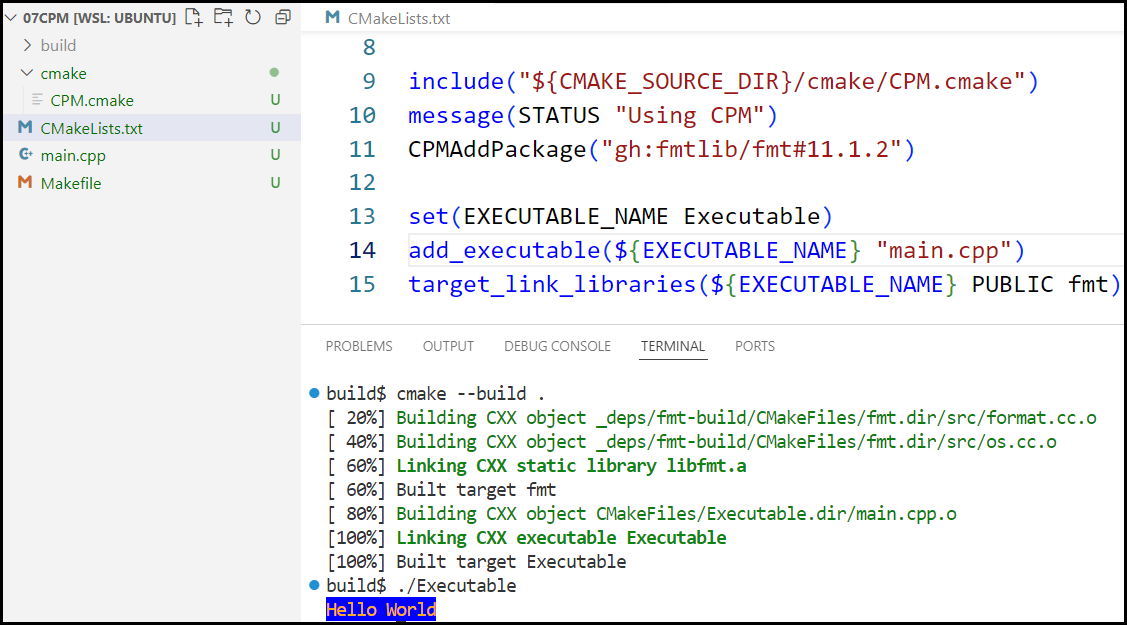
CPMAddPackage("gh:username/repository#version")

* + - CPMAddPackage: This is the main function provided by the CPM module to declare and fetch external dependencies.
      * "gh:username/repository#version": This is a special URL format supported by CPM that:
      * "gh:" Indicates that the dependency is located on GitHub.
      * "username/repository": Specifies the GitHub username and the name of the repository containing the dependency.
      * "#version": (Optional) Specifies the desired version or tag of the repository. If omitted, CPM will typically use the latest commit from the main or master branch.
  + Example:

CPMAddPackage("gh:fmtlib/fmt# 11.1.2")

1. Link the Target Library:
   * Use target\_link\_libraries() to link your project's target with the downloaded library.
   * This ensures the dependency is properly linked during the build process.

target\_link\_libraries(${EXECUTABLE\_NAME} PUBLIC fmt)

1. Build and Run: