Introduction to CMake

Contents

[1. Build System 2](#_Toc188391142)

[Build Files 2](#_Toc188391143)

[2. CMake 6](#_Toc188391144)

[Introduction 6](#_Toc188391145)

[Steps to Build a C++ Project with CMake 7](#_Toc188391146)

[Creating a Simple C++ Project with Multiple Files 13](#_Toc188391147)

# 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)

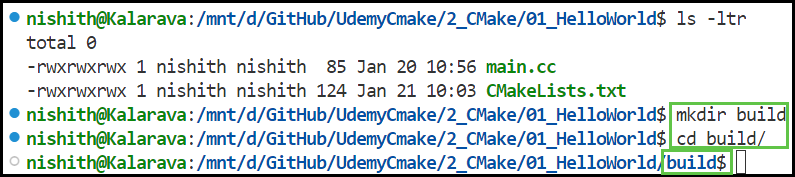
* Breakdown:
  + project: This command sets the name, version, and languages for the project.
  + CppProject: This is the name of your project.
  + VERSION 1.0.0: This specifies the version of your project. In this case, it's set to version 1.0.0.
  + LANGUAGES C CXX: This specifies the programming languages 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)

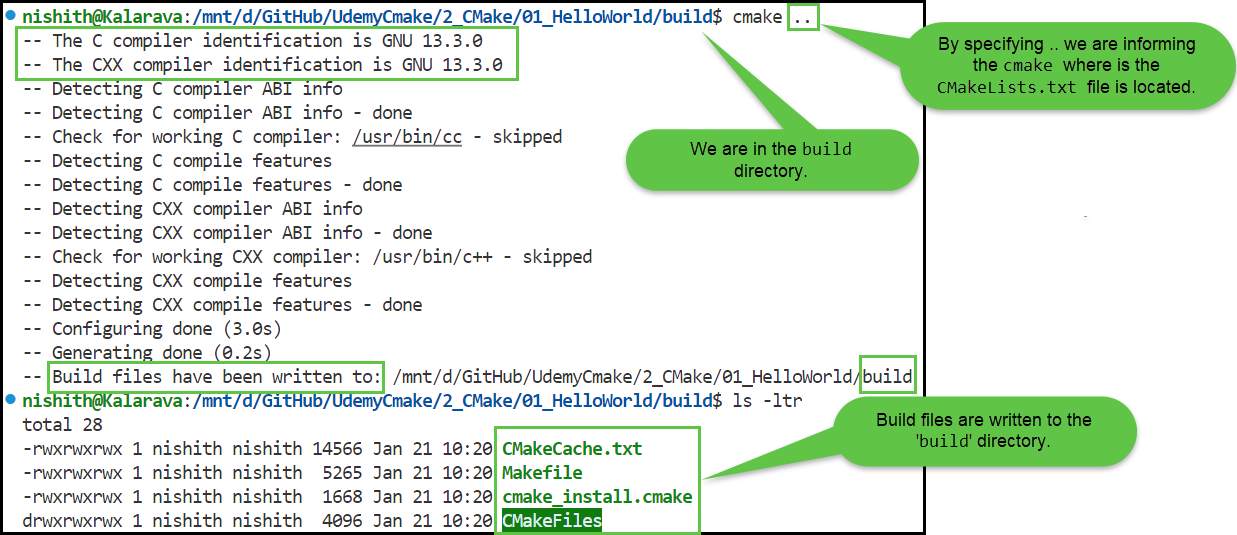
├── Makefile              # For Makefile-based generators

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

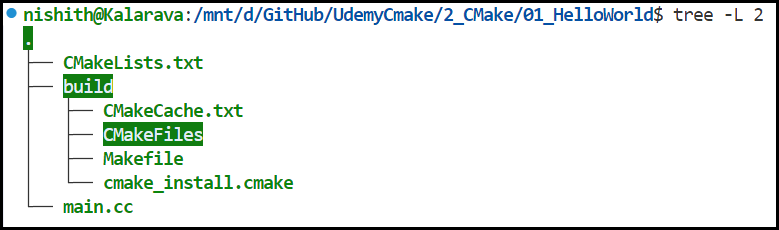
├── build.ninja           # For Ninja-based generators (if Ninja is used)

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

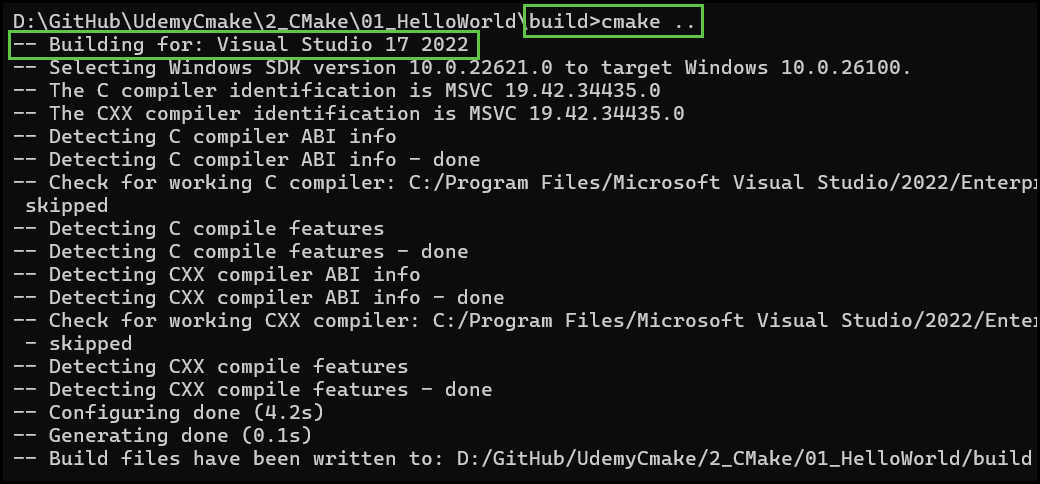
1. For Makefile-based generators.



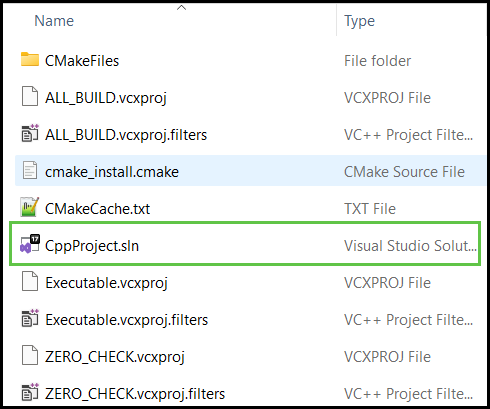
1. The content of build folder for Makefile-based generators.



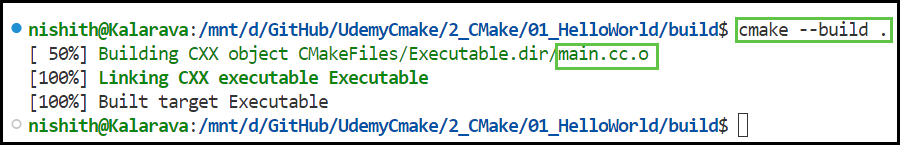
1. For Visual Studio based generators.



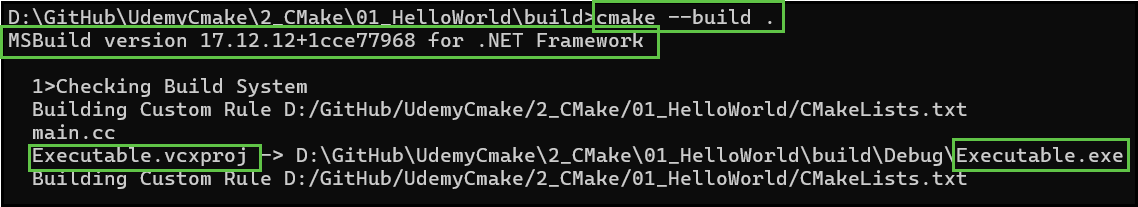
1. The content of build folder for Visual Studio based generators.



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…

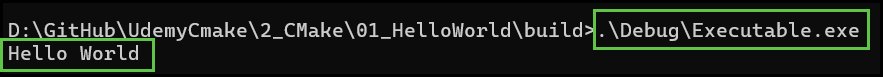


1. For Visual Studio based generators



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





## Creating a Simple C++ 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

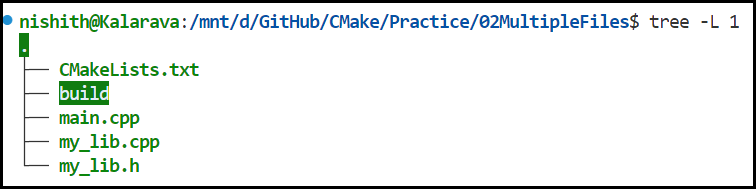
cmake\_minimum\_required(VERSION 3.22)

project(CppProject VERSION 1.0.0 LANGUAGES C CXX)

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

add\_executable(Executable main.cpp)

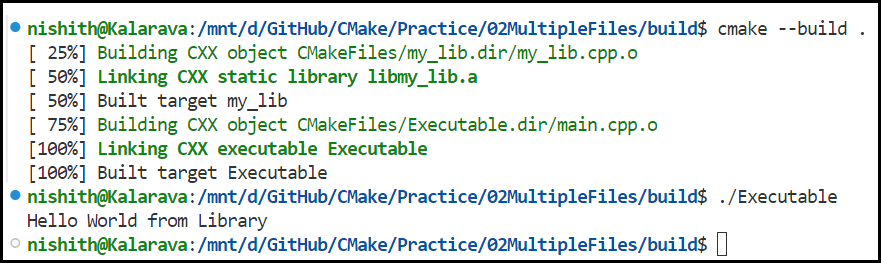
target\_link\_libraries(Executable PUBLIC my\_lib)



### 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

