CMake

Obolenskiy Arseniy, Nesterov Alexander

ITLab

November 15, 2024

Contents

C++ "Hello, World" example

Listing 1: Hello World example

```
1  #include <iostream>
2
3  int main() {
4      std::cout << "Hello, World!" << std::endl;
5      return 0;
6  }
7
```

Building simple main.cpp on UNIX

- Open a terminal.
- Navigate to the directory containing `main.cpp`.
- Compile the program using g++:

```
g++ -o hello main.cpp
```

- Run the executable:

```
./hello
```

Building simple main.cpp on Windows

- Open the Command Prompt.
- Navigate to the directory containing `main.cpp`.
- If using MinGW:

```
g++ -o hello.exe main.cpp
hello.exe
```

- If using Visual Studio Developer Command Prompt:

```
cl /EHsc main.cpp
main.exe
```

Simple library example

Assuming there is a simple library that contains a function add:

Listing 2: add.h

```h
#ifndef ADD_H
#define ADD_H

int add(int x, int y);

#endif // ADD_H

```

Listing 3: add.cpp

```cpp
  #include "add.h"

  int add(int x, int y) {
      return x + y;
  }

```

Listing 4: main.cpp

```cpp
    #include <iostream>
    #include "add.h"

    int main() {
        int result = add(5, 3);
        std::cout << "5 + 3 = " << result << std::endl;
        return 0;
    }

```

Building simple main.cpp with add library on UNIX

- Compile `add.cpp` into an object file:

```
g++ -c add.cpp
```

- Compile `main.cpp` into an object file:

```
g++ -c main.cpp
```

- Link the object files into an executable:

```
g++ -o program main.o add.o
```

- Run the executable:

```
./program
```

Building simple main.cpp with add library on Windows

Using MinGW:

- Compile `add.cpp`:

```
g++ -c add.cpp
```

- Compile `main.cpp`:

```
g++ -c main.cpp
```

- Link the object files:

```
g++ -o program.exe main.o add.o
```

Using Visual Studio Developer Command Prompt:

- Compile and link:

```
cl /EHsc main.cpp add.cpp
```

- Run the executable:

```
program.exe
```

Build systems

There are various build systems for different environments:

- Make
- Autoconf/Automake
- CMake
- Ninja
- Meson
- Bazel

Make

- Traditional build tool using Makefiles.
- Defines rules and dependencies for compiling code.
- Widely used on UNIX systems.
- Simple but can become complex for large projects.

Autoconf/Automake

Autoconf is the colution to automatically generate Makefiles.

- Part of the GNU build system.
- Generates portable Makefiles.
- Handles platform-specific configurations.
- Useful for open-source projects targeting multiple UNIX-like systems.

CMake

- Cross-platform build system generator.
- Generates native build files (Makefiles, Visual Studio solutions, etc.).
- Supports complex project configurations.
- Widely adopted in both open-source and commercial projects.

Ninja

- Focused on speed and efficiency.
- Uses a simple build file format.
- Often used as a backend by higher-level build systems like CMake and Meson.
- Not intended to be written by hand.

Meson

- High-level build system with a simple syntax.
- Uses Ninja as its default backend.
- Designed for fast and user-friendly builds.
- Supports multiple programming languages.

Bazel

- Developed by Google for large-scale projects.
- Focuses on build correctness and reproducibility.
- Supports multiple languages and platforms.
- Uses a domain-specific language for build definitions.

Why CMake?

- Cross-platform compatibility.
- Generates native build systems.
- Handles complex build requirements.
- Strong community support and documentation.
- Integrates well with IDEs and other tools.

Download CMake

- Official website: `https://cmake.org`
- Available for Windows, macOS, and Linux.
- Installation via package managers:
  - On Ubuntu:

    ```
    sudo apt-get install cmake
    ```

  - On macOS with Homebrew:

    ```
    brew install cmake
    ```

CMake script for main.cpp with add library

CMakeLists.txt

```
cmake_minimum_required(VERSION 3.0)
project(HelloAdd)

add_library(add add.cpp)
add_executable(main main.cpp)
target_link_libraries(main add)
```

CMake configure and CMake build commands

- Create a build directory:

```
mkdir build
cd build
```

- Configure the project:

```
cmake ..
```

- Build the project:

```
cmake --build .
```

- Run the executable:

```
./main        # On UNIX
main.exe      # On Windows
```

CMake configure and CMake build commands

Usage of `-S` and `-B` is encouraged:

- Configure the project:

```
cmake -S . -B build
```

- Build the project:

```
cmake --build build
```

- Run the executable:

```
./build/main      # On UNIX
build/main.exe    # On Windows
```

CMake configure

- Confgure
- Build

CMake configure

In a typical CMake-based build process, there are two primary stages: the configure step and the build step.

- Purpose: To inspect the system, check dependencies, and generate platform-specific build files (like Makefile, ninja files, or Visual Studio project files).
- Outcome: After this step, the necessary files for building the project are ready and platform-specific. The project isn't compiled yet, but CMake has configured everything based on the system environment, the user options, and the project setup.

CMake configure (continued)

What Happens on CMake confguration stage?

- CMakeLists.txt Processing: CMake reads the CMakeLists.txt files in your project. This file describes how the project should be built, which dependencies are needed, which source files to compile, etc.
- Checking Dependencies: CMake will check for external libraries and dependencies, ensuring they are installed and can be found (e.g., checking for required packages, libraries).
- Compiler and Toolchain Discovery: CMake will determine which compilers (e.g., GCC, Clang, MSVC) and toolchains to use based on the environment or user input.
- Configuration Options: During this step, you can also pass configuration options to CMake using the -D flag, which can influence how the project is built (e.g., enabling/disabling certain features or setting paths).
- Build File Generation: CMake generates the actual build system files (like Makefile, Ninja files, or Visual Studio solution files) in the output directory (typically in a build/ folder). These files are specific to the build system chosen by the user (e.g., Make, Ninja, MSBuild).

CMake build

The build step is where the actual compilation and linking take place.

- Purpose: To compile the source code into binary executables, libraries, or other artifacts based on the configuration done in the previous step.
- Outcome: After this step, the output files (executables, libraries, etc.) are created and are ready to be run or installed.

CMake build (continued)

What Happens on CMake build stage?

- Calling Build System: The generated build files from the configure step (like Makefile or ninja.build files) are invoked by CMake to compile the code.
- Compiling Source Code: The build system uses the appropriate compiler to compile the source code into object files.
- Linking: Once the object files are generated, the linker combines them into final binaries (e.g., an executable or shared library).
- Error/Warning Reporting: If there are any syntax errors, missing files, or other issues, they will typically surface during this step, as the code is actively being compiled.
- Rebuilding: CMake build system will track changes to source files. If you make changes to specific files and run the build step again, only the modified files are recompiled (i.e., incremental builds).

CMake demo

Demo

Thank You!

References

1. CMake Download https://cmake.org/download/
2. CMake Tutorial
   https://cmake.org/cmake/help/latest/guide/tutorial/index.html
3. An Introduction to Modern CMake
   https://cliutils.gitlab.io/modern-cmake/README.html