CMake Crash Course: From Zero to Hero

Bahey Shalash

February 12, 2025

Contents

1 Introduction

CMake is a cross-platform build system generator that controls the compilation process of software projects by generating native build scripts (such as Makefiles or Visual Studio project files). This crash course will guide youstep by stepfrom a minimal, single-file project (without variables) to a full-featured, multi-directory project that leverages CMakes advanced features.

Note: In the beginning, we keep things simple by avoiding variables. Later sections will introduce variables and other advanced techniques.

2 Basic Concepts in CMake

CMake uses a file called `CMakeLists.txt` in your projects root (and optionally in subdirectories) to describe how your project should be built. The basic components in a `CMakeLists.txt` file are:

- **Minimum Required Version:** Ensures the version of CMake is new enough.
- **Project Declaration:** Names your project (and optionally sets its version).
- **Targets:** Defines what to build (executables or libraries) and which source files to use.

3 A Very Basic Example Without Variables

We begin with a minimal example that avoids using variables to keep the concepts very clear.

3.1 Project Structure

```
project/
├── main.cpp
└── CMakeLists.txt
```

3.2 Minimal `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.10)
project(HelloWorld)

add_executable(hello main.cpp)
```

3.3 Explanation

- `cmake_minimum_required(VERSION 3.10)`
  - Ensures that CMake version 3.10 (or later) is used.
- `project(HelloWorld)`
  - Declares the project name as `HelloWorld`.
- `add_executable(hello main.cpp)`
  - Instructs CMake to build an executable named `hello` from the source file `main.cpp`.

3.4 Building the Project

To build the project:

1. Open a terminal in the `project` directory.
2. Create a build directory:

   ```
   mkdir build
   cd build
   ```

3. Run CMake to generate the build files:

   ```
   cmake ..
   ```

4. Build the project (for example, with `make`):

   ```
   make
   ```

4 Introducing Variables in CMake

Now that you understand the basic structure, we can simplify the `CMakeLists.txt` by using variables. This makes your file more maintainable, especially as the project grows.

4.1 Example with Variables

```
cmake_minimum_required(VERSION 3.10)
project(HelloWorld)

# Define variables for source file and executable name
set(SOURCE_FILES main.cpp)
set(EXECUTABLE_NAME hello)

# Create the executable using the variables
add_executable(${EXECUTABLE_NAME} ${SOURCE_FILES})
```

4.2 Explanation

- `set(SOURCE_FILES main.cpp)` creates a variable holding the source file.
- `set(EXECUTABLE_NAME hello)` defines the name of the executable.
- `add_executable(${EXECUTABLE_NAME} ${SOURCE_FILES})` uses these variables to build the target.

5 Basic CMake Example: A Single-File Project

Below is a complete example for a project with a single source file.

5.1 Project Structure

```
project/
├── main.cpp
└── CMakeLists.txt
```

5.2 `CMakeLists.txt`

```
cmake_minimum_required(VERSION 3.10)
project(SingleFileProject)

# Variables for clarity
set(SOURCE_FILE main.cpp)
set(TARGET_NAME my_program)

# Define the executable target
add_executable(${TARGET_NAME} ${SOURCE_FILE})
```

5.3 How It Works

- The project name is declared and the minimum required version is set.
- Variables `SOURCE_FILE` and `TARGET_NAME` store the source file and target name.
- The executable `my_program` is built from `main.cpp`.

6 Expanding to a Multi-File Project in a Single Folder

When your project contains several source files, you can list them all in a variable.

6.1 Project Structure

```
project/
├── main.cpp
├── helper.cpp
└── helper.h
```

6.2 `CMakeLists.txt` for Multiple Files

```
cmake_minimum_required(VERSION 3.10)
project(MultiFileProject)

# List all source files
set(SOURCE_FILES main.cpp helper.cpp)
set(TARGET_NAME my_program)

# Define the executable target
add_executable(${TARGET_NAME} ${SOURCE_FILES})
```

6.3 Explanation

- `SOURCE_FILES` includes both `main.cpp` and `helper.cpp`.
- The target `my_program` is built by compiling and linking these files.

7 Moving to a Multi-Folder Project

For larger projects, it is common to organize your code into multiple directories (e.g., separating source files and header files).

7.1 Project Structure

```
project/
├── src/
│   ├── main.cpp
│   └── helper.cpp
├── include/
│   └── helper.h
└── CMakeLists.txt
```

7.2 Advanced `CMakeLists.txt` Example

```
cmake_minimum_required(VERSION 3.10)
project(MultiFolderProject)

# Specify the C++ standard (optional)
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)

# Include the header directory
include_directories(include)

# Collect all source files in the src/ directory
file(GLOB SOURCE_FILES src/*.cpp)
set(TARGET_NAME my_program)

# Define the executable target
add_executable(${TARGET_NAME} ${SOURCE_FILES})
```

7.3 Key Points

- `include_directories(include)` tells CMake where to find header files.
- `file(GLOB SOURCE_FILES src/*.cpp)` automatically gathers all `.cpp` files in `src/`.
- The executable is built from the discovered source files.

8 Advanced Topics and Techniques

8.1 Build Types and Compiler Flags

CMake allows you to choose between build types (e.g., `Debug` and `Release`). You can set the build type when running CMake:

```
cmake -DCMAKE_BUILD_TYPE=Debug ..
```

Additionally, you can customize compiler flags by modifying variables such as `CMAKE_CXX_FLAGS`.

8.2 External Libraries and Packages

CMake offers powerful features to find and link external libraries:

```
find_package(SomeLibrary REQUIRED)
if(SomeLibrary_FOUND)
    include_directories(${SomeLibrary_INCLUDE_DIRS})
    target_link_libraries(${TARGET_NAME} ${SomeLibrary_LIBRARIES})
endif()
```

8.3 Adding Subdirectories

For very large projects, you might split the project into subdirectories, each with its own `CMakeLists.txt`. In your top-level `CMakeLists.txt` you can add:

```
add_subdirectory(src)
add_subdirectory(tests)
```

9 Debugging and Common Pitfalls

9.1 Debugging Techniques

- **Verbose Output:** Run `cmake --trace-expand ..` to see detailed output.
- **CMake GUIs:** Tools like `ccmake` or `cmake-gui` let you inspect and modify CMake cache variables.

9.2 Common Pitfalls

- **Directory Structure:** Ensure that the directory layout in your `CMakeLists.txt` matches your project.
- **Case Sensitivity:** File and directory names are case-sensitive on many platforms.
- **CMake Version:** Some features require a newer version of CMake; always verify your version if you run into issues.

10 Comprehensive Example: From Beginner to Advanced

Below is a comprehensive example that combines many features discussed in this crash course. It supports:

- Automatic source discovery (even recursively)
- Inclusion of header directories
- Setting build types and standards
- Linking to an external library (if found)

```
cmake_minimum_required(VERSION 3.10)
project(ComprehensiveProject)

# Set C++ standard and build type
set(CMAKE_CXX_STANDARD 11)
set(CMAKE_CXX_STANDARD_REQUIRED True)
if(NOT CMAKE_BUILD_TYPE)
    set(CMAKE_BUILD_TYPE Release)
endif()

# Include directories
include_directories(${PROJECT_SOURCE_DIR}/include)

# Automatically gather all source files (recursively) in the src/ directory
file(GLOB_RECURSE SOURCE_FILES ${PROJECT_SOURCE_DIR}/src/*.cpp)
set(TARGET_NAME my_program)

# Define the executable target
add_executable(${TARGET_NAME} ${SOURCE_FILES})
```

```
# Example of linking an external library (if found)
find_package(SomeLibrary REQUIRED)
if(SomeLibrary_FOUND)
    include_directories(${SomeLibrary_INCLUDE_DIRS})
    target_link_libraries(${TARGET_NAME} ${SomeLibrary_LIBRARIES})
endif()

# Installation rules (optional)
install(TARGETS ${TARGET_NAME} DESTINATION bin)
```

10.1 How It Works

1. **Build Type:** The build type defaults to `Release` if not specified.
2. **Header Inclusion:** `include_directories` tells CMake where to look for header files.
3. **Source Discovery:** `file(GLOB_RECURSE ...)` collects all `.cpp` files under `src/`.
4. **External Libraries:** `find_package` and `target_link_libraries` handle external dependencies.
5. **Installation:** The `install` command specifies where the built executable should be installed.

11 Conclusion and Further Resources

This crash course has guided you from the basics of CMakebuilding a minimal project without variablesto managing more complex projects with multiple directories, automatic source discovery, and external library linking. By taking the time to understand each concept in depth, youll be well-equipped to configure and manage builds for your own projects.

11.1 Further Reading

- **Official CMake Documentation:** `https://cmake.org/cmake/help/latest/`
- **CMake Tutorial:** Numerous online tutorials provide hands-on exercises.
- **Open-Source Projects:** Study GitHub projects that use CMake to see how these techniques are applied in real-world scenarios.

Happy building and coding!