Make and Cmake: what are they?!

An introduction to tools for C and C++ compilation

Matteo Dalle Vedove

June 17-18, 2024

Dept. of Industrial Engineering, University of Trento

Overview

Introduction to the course

Environment setup

Compiling the code

Make and Makefiles

CMake

Advanced CMake

Introduction to the course

What is programming?

When we program, we want to write *code*, i.e. instructions, that can *run* on some hardware.

To do so, we can use lots of different *programming languages*. Each programming language has its own *syntax* and *semantics*, and they might be target to different type of use (e.g., Web Applications, Mobile Applications, Embedded Systems, etc.).

Compiled or interpreted?

The main categorization of programming languages is between

- *Interpreted languages*, like Python, Matlab and Julia, in which the code is translated (at runtime) into a *byte-code* (an intermediate representation) that is then executed by an *interpreter* on the machine.
- *Compiled languages*, like C, C++ and Rust, instead employ a *compiler* to directly translate the *source code* into a *binary* that can be executed by the machine.

Why a compiler?

A compiler enables an efficient translation of the source code into a binary that can be directly executed by the machine. W.r.t. running an interpreted language, this has several advantages:

- the binary is generally *faster* to execute, as it is directly understood by the machine;
- the compiler can perform *optimizations* on the code, to make it faster or smaller;
- the compiler can perform *static analysis* on the code, to catch errors before running it.

Why not a compiler?

Compiled languages are generally more complex to use than interpreted languages, since they involve multiple steps to have a program that can run (e.g., compilation, linking, etc.).

In addition, the compiler depends on the *platform* (e.g., the operating system, the hardware, etc.), so the same code might need to be recompiled to run on different platforms.

Environment setup

The preliminaries

Before actually starting the course we need some setup...

We need:

- a *compiler*;
- the *make* and *cmake* programs;
- an *IDE* to modify our code (VSCode).

Unix systems

The *make* program just works on unix-like system, i.e., Linux and MacOs. Windows is not ufficially supported, and thus is not covered in this course, so please use *WSL*, a *docker container* or a *virtual machine*.

On Ubuntu:

```
sudo apt update
sudo apt install -y build-essential git
```

On MacOs, make sure `xcode` and `homebrew` are installed, then

```
brew install make
```

Compilers

Generally speaking, there exists multiple compilers. For unix systems, the most famous ones are the *GNU Compiler Collection* (GCC) and the *LLVM*-based CLANG.

Other known solutions are the Intel C++, and for windows the *Microsoft Visual Studio Compiler* (MVSC).

We will mainly work with GCC.

Compiling the code

Hello world!

The first thing we can do is the basic *hello world* executable in plain C.

First let's create a `hello.c` file:

```c
#include <stdio.h>

int main() {
    printf("Hello, world!");
    return 0;
}
```

Hello world!

We can now switch to the terminal to actually compile the code:

```
gcc hello.c -o hello
```

In this command we:

- call the `gcc` compiler;
- specify the source file `hello.c`;
- specify the output (`-o`) object `hello`.

Hello world!

```
gcc hello.c -o hello
```

Under the hood, this command performs lots of operations:

- it *pre-process* the *.c, .cpp* source code (e.g., header and macro expansion);
- it *compiles* the code into assembly code that's specific to the target architecture;
- the *assembler* converts the assembly code into machine code (*.o, .obj*);
- the *linker* links the object files into a final executable (*.out, .exe*).

A more complex example

Let's create a simple project that involves multiple sources: a library (*lib.c*, *lib.h*) and a main program *main.c*.

There are two *equivalent* ways to compile the code: either

```
gcc lib.c main.c -o main
```

or

```
gcc -c lib.c
gcc -c main.c
gcc lib.o main.o -o main
```

Some compilation flags

```
gcc -c lib.c
```

In this command, the `-c` flag tells the compiler to compile and assembly the *lib.c* source code, creating *lib.o* (since we didn't specify the output with `-o`).

Some compilation flags

A complete list of compiler flags can be found by running

```
gcc --help
```

Some relevant flags are:

- `-E`: perform only the pre-processing;
- `-S`: perform only the compilation to assembly *.s*;

Best compilation method

What's better between the two methods?

```
gcc lib.c main.c -o main
```

```
gcc -c lib.c
gcc -c main.c
gcc lib.o main.o -o main
```

Simplifying the compilation

The work of compilation is tedious and error-prone. Each time we modify our source code, we need to update our objects and link the final executable.

How can we simplify this process?

Let's put all the commands into a *shell script*!
For example, we can create a `build.sh` file with the following content:

```
gcc -c lib.c
gcc -c main.c
gcc lib.o main.o -o main
```

Shell scripts for compilation

Using a shell script is a good idea, but it's not the best solution.

Indeed there are few drawbacks:

- *.sh* scripts are not scalable;
- *.sh* scripts are less configurable;
- each time, we need to recompile all the sources. This is no problem for small projects, but it's a big issue for large ones (e.g. the Linux kernel takes hours to compile).

Make and Makefiles

What is Make?

By definition,

> *GNU Make is a tool which controls the generation of executables and other non-source files of a program from the program's source files.*

In this regard, make is a *build system*, a tool that automates the action of *building* an application, i.e. it collects dependencies, creates objects, and it links them.

Make is a programming language that enables to abstract the build procedure, providing simpler way to compile executables using variables and rules.

Why Make?

Make addresses all the limitations of the manual/scripted compilation process:

- *reproducibility*: the build process is defined in a single file, so it can be easily shared and executed on different machines;
- *efficiency*: make **only recompiles** the files that have changed;
- *maintainability*: using variables and rules, the build process is more readable and maintainable.

How to use Make?

To use make to build a project, it is necessary to create a *Makefile* containing all the rules to build the project.

After that, when inside the project directory, we can run the command `make` to build the project.

Makefiles

Let us consider the compile script we have seen before and the corresponding *Makefile*:

```
gcc -c lib.c
gcc -c main.c
gcc lib.o main.o -o main
```

```
main: lib.o main.o
    gcc lib.o main.o -o main

lib.o: lib.c
    gcc -c lib.c

main.o: main.c
    gcc -c main.c
```

Makefiles

Each Makefile contains a set of *rules* of the form

```
<target>:<dependencies>
    <steps>
```

- *target* is the output of the rule;
- *dependencies* are the required rules to build the target;
- *steps* are all commands required to generate the target.

```
main: lib.o main.o
    gcc lib.o main.o -o main

lib.o: lib.c
    gcc -c lib.c

main.o: main.c
    gcc -c main.c
```

Makefiles: so what?

At this stage, the only advantage bring by make is that only source code that will be actually modified is re-compiled.

To improve, we will start making use of *regular expressions* and *variables*.

```
main: lib.o main.o
    gcc lib.o main.o -o main

lib.o: lib.c
    gcc -c lib.c

main.o: main.c
    gcc -c main.c
```

Makefiles: variables

In Makefiles we can define *variables* that can be used in the code as:

```
<NAME>=<VALUE>
```

`$(<NAME>)` denotes the access of a variable.

Two *special* variables:

- `$@`: current target;
- `$^`: target dependencies;
- `$<`: first target dependency;
- `$?`: *updated* target dependencies;

```
CC=gcc
CFLAGS=-Wall -Wextra
OBJS=lib.o main.o
BINARY=main

$(BINARY): $(OBJS)
    $(CC) -o $@ $^

lib.o: lib.c
    $(CC) $(CFLAGS) -c lib.c

main.o: main.c
    $(CC) $(CFLAGS) -c main.c
```

Makefiles: regular expressions

By using the *wildcard* % (or, in general, any regular expression), we can create a more *general* makefile.

```
CC=gcc
CFLAGS=-Wall -Wextra

main: lib.o main.o
    $(CC) -o $@ $^

%.o: %.c
    $(CC) $(CFLAGS) -c $^ -o $@
```

Makefiles: targets

Within the same makefile, it is possible to create multiple targets.

Running the plain `make` command will build, by default, the top-most target (`all` in this case).

But we can also specify the target to generate, like `make clean`.

```
CC=gcc

all: main

clean:
    rm -f *.o main

main: lib.o main.o
    $(CC) -o $@ $^

%.o: %.c
    $(CC) -c $^ -o $@
```

Makefiles: suppressing commands

When running, make will display the actual shell command that's executed (e.g. *gcc -o lib lib.c ...*).

To avoid this behavior, it is simply necessary to prepend a `@` character at the beginning of the line, e.g.:

```
clean:
    @echo "Removing .o and .s files"
    @rm -f *.o *.s
```

Makefiles: target dependencies

In practice, make decides to rebuild a rule when the provided dependencies have *newer* updates w.r.t. to the target itself.

This, however, doesn't capture all the semantic dependencies between files. For instance, changing a header *.h, .hpp* won't trigger the compilation of all objects that depends from them (since they are not explicit dependencies).

Makefiles: embed header dependencies

To embed relationship between source code and header:

- add the `-MP -MD` compilation flag to generate *.d* dependency files;
- append `-include <files...>.d` at the end of the makefile.

A Makefile example

Let's build a makefile for the project in the git repository

Why not to use Make?

Make is a powerful build system, better then basic shell scripting. However:

- as the project gets bigger, it is *harder* to maintain;
- it requires knowledge of CLI tools to accomplish multiple tasks;
- it is hard to manage external dependencies (e.g. other libraries).

CMake

What is CMake?

CMake might be regarded as an abstraction on top of makefiles.

Indeed, cmake is *not a build system* by itself, but rather a multi-platform *generator for build system*.

CMake can in fact generate configuration files for make, *ninja* or *visual studio projects*.

CMake in 3 lines

CMake projects are configured in *CMakeLists.txt* files.

A simple example is

```
cmake_minimum_required(VERSION 3.11)
project(my_sample_project)

add_executable(hello main.cpp)
```

in which we create an executable *hello* starting from a single source file *main.cpp*.

CMake in 3 lines: and now?

With the *CMakeLists.txt* file in place, we have to generate the file for our build system.

One cool feature of cmake is that it enables to easily *decouple* the *source code* from the *build* (they can live in different directories in the system).

A known standard is to create a *build* folder inside the root directory of the project.

CMake in 3 lines: and now?

With the *CMakeLists.txt* file in place, we have to generate the file for our build system.

To list availables *generators*, run the command `cmake -G`.

From within the *build* directory, we *configure* the project with the command `cmake <path/to/prj> -G makefiles`

If we don't specify the `-G` flag, we use the default generator (most probably *make* or *ninja*).

CMake in 3 lines: and now?

With the *CMakeLists.txt* file in place, we have to generate the file for our build system.

From within the *build* directory, we configure the project with the command `cmake <path/to/prj> -G makefiles`

We can now *call* the build system to build the project, i.e., running `make` (or the chosen generator).

CMake: generate and build

To have access to a full list of arguments, call `cmake -h`. Generally, to generate a project:

```
cmake [options] <path/to/source>
cmake [options] <path/to/existing/build>
cmake [options] -S <path/to/source> -B <path/to/build>
```

Then, to build a project (regardless of the generator), call

```
cmake --build <path/to/build>
```

CMake: executables and libraries

With cmake we can mainly handle 2 kind of *objects*:

- *executables*, so application that *can run* (having a *main* function);

```
add_executable(<exec name> <sources...>)
```

- *libraries*, so binary objects that are not executables but contains implementaiton of functions.

```
add_library(<lib name> <STATIC | SHARED> <srcs...>)
```

CMake: variables

As in make, cmake makes extensive use of *variables*.

A variable is defined with the `set` function and can be accessed with the `${}` notation:

```
set(LIB_SOURCES src/lib1.cpp src/lib2.cpp)
add_library(mylib ${LIB_SOURCES})
```

CMake: variables

CMake provides different variables that simply cmake scripting, like:

- `CFLAGS/CXXFLAGS`: custom compilation flags for C/C++ executables;
- `CMAKE_CXX_STANDARD`: the C++ standard to use;
- `CMAKE_BUILD_TYPE`: the build type:
  - `Debug`: no optimization, debug symbols;
  - `MinSizeRel`: optimization for size.
  - `RelWithDebInfo`: optimization, debug symbols;
  - `Release`: full optimization, no debug symbols.
- `CMAKE_SOURCE_DIR`: the (absolute) path of the source directory;
- `CMAKE_BINARY_DIR`: the (absolute) path of the build directory;
- `CMAKE_CURRENT_SOURCE_DIR`: the directory of the current *CMakeLists.txt*.

CMake: interacting with the variables

The full state of the cmake project is stored in the `CMakeCache.txt` file in the build directory.

Variables can be set, while calling `cmake` command, passing the `-D` flag, e.g.

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

To interact dynamically with the variables, there exists two executables: `ccmake`, a CLI tool, and `cmake-gui`, a graphical interface.

CMake: headers and libraries

The main advantage of using cmake is its ability to easily handle executables headers and dependent libraries.

To specify the location of the headers of a given target, we use the function

```
target_include_directories(<target> <PUBLIC | PRIVATE |
   INTERFACE> <include paths...>)
```

Similarly, libraries are linked as

```
target_link_libraries(<target> <PUBLIC | PRIVATE |
   INTERFACE> <include paths...>)
```

CMake: public, private and interface

For both `target_include_directories` and `target_link_libraries` , we have 3 specifiers for the property usage of the current and dependent targets:

- `PUBLIC`: the property is propagated to both the current target and its dependents;
- `PRIVATE`: the property is only applied to the current target;
- `INTERFACE`: the property is only applied to the dependents of the current target.

CMake: subdirectories

To simplify the management of a project, cmake allows the use of *subdirectories*.

One can write a *CMakeLists.txt* file in a subdirectory and include it in the main one with the `add_subdirectory` function:

```
add_subdirectory(<path/to/subdir>)
```

This enables to have a *modular* project structure, where the knowledge on *how to build a target* is close to the source code.

CMake: file globbing

The good cmake practice requires that all source files are explicitly listed in the target defition (mainly for clarity).

Still, there are few cases in which it is useful to use *file globbing*, e.g. to create executables for each source file in a directory.

To do so, we can use the `file` function:

```
file(GLOB SOURCES src/*.cpp)
```

Doing so, we create the variable `SOURCES` that contains all the `.cpp` files in the `src` directory.

CMake: file globbing

We can also do some cmake scripting to create a target for each source file in a directory, e.g.

```
file(GLOB SOURCES src/*.cpp)
foreach(source ${SOURCES})
    get_filename_component(exec_name ${source} NAME_WE)
    add_executable(${exec_name} ${source})
endforeach()
```

Note

This globbing occurs at *configure* time. If a new file is added to the directory, the project must be reconfigured, otherwise the new file will not be included in the build.

CMake: logging

As scripts become more complex, it is useful to have a way to log messages to the console.

For this purpose, cmake provides the `message` function:

```
message(<mode> "<message>")
```

where mode could be `STATUS`, `WARNING`, `SEND_ERROR`, `FATAL_ERROR` or `AUTHOR_WARNING`.

CMake: functions and macros

To avoid code duplication, cmake allows the definition of *functions* and *macros*.

Their syntax is similar, but the main difference is that functions have their own scope, while macros are expanded in the caller's scope (are *inlined*).

```
function(<name> <args ...>)
    <body>
endfunction()
```

```
macro(<name> <args ...>)
    <body>
endmacro()
```

CMake: a function example

An example of function is the one printing the name of the variable, as well as its value:

```
function(print_var)
   message(STATUS " > ${var} = ${${var}}")
endfunction()
```

The script

```
set(MY_VAR "Hello World")
print(MY_VAR)
print(MY_UNSET_VAR)
```

Produces the output

```
> MY_VAR = Hello World
> MY_UNSET_VAR =
```

CMake: function with variable number of arguments

Function can accept a variable number of arguments, and provide default variable names to access them, particularly:

- `ARGC`: the number of arguments;
- `ARGN`: the full list of arguments;
- `ARGV0`, `ARGV1`, `ARGV2`, ...: the first, second, ... argument.

CMake: function with variable number of arguments

In light of this, we can define a function that prints all its arguments:

```
function(print)
    foreach(arg ${ARGN})
        message(STATUS " > ${arg} = ${${arg}}")
    endforeach()
endfunction()
```

Advanced CMake

Advanced CMake

Until now we used the basic features of CMake, showing the potential over plain Make.

From now, we will see some advanced features that are necessary to handle more complex, real-world projects.

CMake: finding packages

Any decently-complex *project* will *depend on external libraries* (e.g., *Eigen* for linear algebra, *OpenCV* for image processing, etc.).

To ensure that such dependencies are met, we can use the `find_package` command.

Libraries are searched system-wide, or in directories specified by the `CMAKE_PREFIX_PATH` variable. Suggestions on a per-package basis are avaible.

CMake: finding packages

```
find_package(Eigen3 REQUIRED)
find_package(OpenCV)
```

In this example, not finding the *Eigen* library will abort the configuration step, since we set the `REQUIRED` keyword.

The `OpenCV` package is instead optional, i.e., failing to find it will not stop the project build.
The call to `find_package` will set some variables, like `OpenCV_FOUND` which can be queried to assess wethever the package was found (or not).

CMake: using packages

Once the package is found, it can be used as any other target in cmake, e.g. by linking it to a target:

```
add_executable(my_target main.cpp)
target_link_libraries(my_target Eigen3::Eigen)
```

This will link the `my_target` executable with the *Eigen* library.

In this case, we use a *namespaced* target linking, which is a modern (and recommended) way to handle dependencies in CMake, i.e., of the form `<package_name>::<target>`.

CMake: fetching content

To easen development, cmake provides some useful commands to integrate different projects.

In this regard, `FetchContent` is a powerful tool to download external projects and build them along with your application.

```
include(FetchContent)
FetchContent_Declare(
    Eigen3
    URL https://gitlab.com/libeigen/eigen/-/archive/3.4.0/
   eigen-3.4.0.zip
    FIND_PACKAGE_ARGS NAMES Eigen3
)
FetchContent_MakeAvailable(Eigen3)
```

CMake: file configuration

Sometimes it might be necessary to set some variables of the project at configuration time, e.g., the project version or some path.

To this extent, `configure_file` is a function that takes as input a template file and write a new file with cmake variable substituted:

```
configure_file(
    defines.hpp.in
    defines.hpp
)
```