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

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).