Prog C — td7

version **#0.0.1-dirty**

Assistants C/Unix 2024 <assistants@tickets.forge.epita.fr>

Copyright

This document is for internal use at EPITA (website) only.

Contents

*https://intra.forge.epita.fr

1 Makefile

1.1 Context

For now, in order to compile your program you have been using something like:

```
42sh$ gcc -std=c99 -Wall -Wextra -Werror -pedantic hello_world.c -o hello_world
```

We can all agree that this is inconvenient and fastidious to type[1].

When working on actual projects, you will also have more than one source file, and quickly, your simple one line command can become very long and if you want to move a file it will be complex to edit[2].

Yet another problem that you will experience soon enough: compiling a big project can take a lot of time and be boring. You do not want to recompile the whole project each time you edit a single file.

To summarize, we need something that can solve these problems:

- We do not want to type a long command.
- We want to be able to edit our compilation command easily.
- We do not want to recompile our project each time we edit one single file.
- We want other people to be able to easily compile our project.

1.2 What is make?

In response to those issues, the POSIX standard contains a tool used to handle project builds: `make`. The implementation of `make` that we will use this semester is GNU make which was released by the FSF as part of the GNU project. `GNU make` implements all the features defined in the POSIX standard for `make` and has some nice extensions as well.

`make` is a tool used to simplify the task of building programs composed of many source files. It can be configured through a file named `Makefile`, `makefile`, or `GNUmakefile`. When ran, `make` will first search for a `GNUmakefile`, then `makefile`, and finally `Makefile` if it did not find the previous ones. However, it is recommended in the official GNU/Make documentation to call your makefile `Makefile`, and we expect you to follow this convention for your submissions.

`make` parses rules and variables from the `Makefile` which are mainly used to build the project. `make` will only re-build things that need to be re-built by comparing modification dates of targets with their dependencies.

A `Makefile` typically starts with a few variable definitions, followed by a set of target entries. Each variable from the `Makefile` is used with `$(VARIABLE)` and can be declared at the top of the file as:

```
VARIABLE = VALUE
```

Finally, each rule abides by the following syntax:

[1] And do not talk to us about your `.bash_history`, it is equally fastidious to search the command line you want.
[2] `fc(1)` is out of the discussion as well.

```
target: dependency1 dependency2 ...
    command
    ...
```

A target is usually the name of the file generated by the Makefile rule. The most common examples of targets are executables or object files. It can also be the name of an action to carry out, for example `all`, `clean`, ...

A dependency (or "prerequisite") is a file that needs to exist in order to create the target. A target can also be a dependency for another target. When evaluating a rule, `make` will analyze its dependencies and if one of them is a target of another rule, `make` will evaluate it too before the one it depends on.

A command (or "recipe") is an action that `make` carries out. Be careful, each command is preceded by a **tabulation** (`\t`). Otherwise, your `Makefile` will not work.

1.3 Invocation

1.3.1 Basics

Let us start easy. First, you need a `hello_world.c` like the following one:

```c
#include <stdio.h>

int main(void)
{
    puts("Hello World !");
    return 0;
}
```

And a `Makefile`:

```
CC=gcc
CFLAGS=-std=c99 -Wall -Wextra -Wvla -Werror -pedantic

exec: hello_world
    ./hello_world

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

Be careful!

Remember that the `command` part of a `rule` **must** start with a tabulation. If you do not, make will raise the following error:

```
Makefile:5: *** missing separator.  Stop.
```

Now try:

```
42sh$ make exec
gcc -std=c99 -Wall -Wextra -Wvla -Werror -pedantic -o hello_world hello_world.c
./hello_world
```

So what is going on here?

First you have two `variables`:

- `CC`, for the *C* Compiler
- `CFLAGS`, for the *C* Flags

Going further...

Some variables are predefined by make. `CC` and `CFLAGS` belong to them and are generally used for the *C* compiler and *C* flags respectively.

Then you have two `rules`. Those are the core of a `Makefile`. Let us look at the first one.

First we have the target `exec` that will execute a binary called `hello_world` using bash.

In order to execute our binary, we need it. We say that our target `exec` depends on the existence of the binary `hello_world`. Thus, `hello_world` is a **dependency** of the target `exec`.

But the binary `hello_world` does not exist! This is why we have the second rule! If a dependency does not exist, `make` will search for a rule that can produce it and execute it before the first.

The second rule follows the same principle. This time the target is `hello_world` and that is the file produced by this rule. This rule has as dependency `hello_world.c`.

Now that our `rule` specifies the target it produces and what it depends on, we need to specify how it should be produced. Remember our `variables`? We will need them now. In order to expand[3] a Makefile `variable`, you need to put it between parentheses[4] with a dollar before it.

So if we summarize this rule, it will produce the target `hello_world` that depends on the file `hello_world.c` using the following command line:

```
gcc -o hello_world hello_world.c
```

Going further...

The default `rule` called when typing `make` is the first one of the file. Thus, here, typing `make` or `make exec` will have exactly the same behavior.

If you run `make hello_world` without updating any file, and after running `make` or `make hello_world` once, you will have a message like this:

```
42sh$ make hello_world
make: 'hello_world' is up to date.
```

If a target already exists, `make` will rebuild it if its dependencies do not exist or if they have been updated since the last build.

Going further...

You can force `make` to rebuild all targets with the option `-B`.

If you modify the `hello_world.c` and retry `make hello_world`, it will rebuild everything.

[3] The `expansion` is the concept of replacing a variable by its value before interpreting the line. Remember you shell courses. When you needed to access a variable's value you also used this syntax.

[4] You might see braces instead of parentheses, they work the same. It is as you prefer.

Another interesting thing with `make` is the `-n` option which asks `make` to simply print the commands it would have run instead of actually running them.

```
42sh$ cat Makefile
all:
    echo toto
42sh$ make
echo toto
toto
42sh$ make -n
echo toto
```

Tips

Generally the first rule in a Makefile is named `all`. Note that this is only a convention and you are free to not respect it.

I must not fear. Fear is the mind-killer.