Using Makefiles

Luca Tornatore - I.N.A.F.

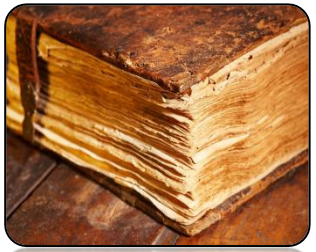
"**Foundation of HPC" course**

DATA SCIENCE &
SCIENTIFIC COMPUTING

2020-2021 @ Università di Trieste

Outline

Introduction



Basic rules

(some) Advanced Topics

The annoying compilation workflow

You know how to compile either a single source file project:

```
cc ${CFLAGS} ${OPTIONS} -o my_exec my_source.c
```

or a multiple source files project:

```
cc ${CFLAGS} ${OPTIONS} -o my_exec my_source_1.c my_source_2.c … \
my_source_n.c
```

Which may be effective enough as long as you have up to few files and compilation options (hidden in `${CFLAGS}` and `${OPTIONS}` in the previous examples) that do not vary in a complicated way. Even in this case it may be an annoying process, especially so when you get back to your project after days or weeks.

The annoying compilation workflow

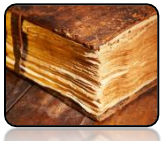
The **`make`** and **`makefile`** tools (*) have been developed exactly to automate the building of your projects, either small or tremendously large .

Using make it is possible to define rules and different “compilation paths” that depend on the value of some variables.

The basic idea is to link different set of commands to different *targets*. Targets are mnemonic names that are used to invoke the associated set o command. A target + its commands set is called a *rule*.

A commonly present rule, for instance, is **`clean`**, which is associated to removing object files and executables

(*) there are more advanced and flexible tools, like **`cmake`** and **`meson`**; however, **`make`** is still widely used and is extremely effective for small- and medium- size projects due to its simplicity



How to invoke make

To invoke make, you simply type at command line

```
:> make
```

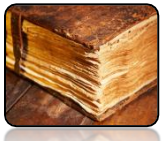
By default, `make` looks to a file named `Makefile` as its input file and executes the first specified rules as default rules.
You can change this behaviour by invoking `make` with additional arguments:

```
:> make target
```

with this, make will execute *target*.

```
:> make -f filename
```

with this, make will use *`filename`* as its input file.



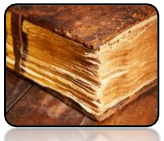
Rules

As we said, a makefile is a collection of rules.
A rule is defined as

These are tab characters, not blank spaces!

```
target: prerequisites
    command1
    command2
    …
    commandN
```

- The `targets` are filenames, separated by space. Routinely, only one file is present in a rule
- The `command`... are a number of instructions whose aim is to build the `target`; each command line *must* begin with a `tab`.
- The `prerequisites` are also filenames, separated by spaces. As the name itself explains, they must exist before the `target` is built.



Rules

Let's have a first example (in the folder `./simple`)

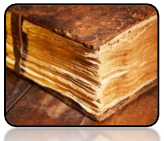
```
try: try.o
	cc try.o -o try

try.o: try.c
	cc -c -o try.o try.c

try.c:
	echo "int main(void) {return 1;}" > try.c

clean:
	rm -f try.c try
```

So, simply invoking `make` will execute the first rule, because that is the default behaviour. The first rule, `try`, depends on `try.o` which does not exist. Hence, make looks for it and finds that it depends on `try.c`, which does not exist either.
`try.c` has its own rule, that actually amounts to create it.
Rewinding, the `make` compiles `try.c` into `try.o` and finally it links the object file into the `try` executable.



Rules

Let's now consider a slightly more complex situation, that you find in the folder `./example`

The 3 files `main.c`, `welcome.c` and `check_args.c` should be compiled and linked into the executable `testmake`.

You would normally do:

```
gcc –o testmake main.c check_args.c welcome.c
```

However, we can automate that a bit by a *rule*:

```
testmake: main.o welcome.o check_args.o
        gcc -o testmake main.c welcome.c check_args.c
```

You find that in the `Makefile.0` file.

Rules

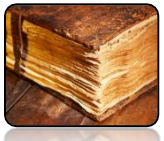
```
testmake: main.o welcome.o check_args.o
        gcc -o testmake main.c welcome.c check_args.c
```

We can generalize this a bit by using variables, that in a makefile are always strings:

```
CC = gcc
CFLAGS = -g –fno-omit-frame-pointer

testmake: main.o welcome.o check_args.o
        ${CC} -o testmake main.c welcome.c check_args.c
```

You find that in the `Makefile.1`file.



Rules

Now we want to generalize, making the .o files dependent on the header file and the `Makefile` itself; in this way, if we change either of the two, automatically `make` will rebuild the object files (see `Makefile.2`)

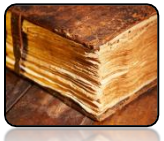
```
CC = gcc
CFLAGS = -g -fno-omit-frame-pointer

HEADERS = myheader.h

%.o: %.c $(HEADERS) Makefile
	$(CC) $(CFLAGS) -c $< -o $@

testmake: main.o welcome.o check_args.o
	${CC} $(CFLAGS) -o testmake main.c welcome.c
check_args.c

clean:
	@rm -f  main.o welcome.o check_args.o testmake *~
```



Rules

Now we want to generalize, making the .o files dependent on the header file and the `Makefile` itself; in this way, if we change either of the two, automatically `make` will rebuild the object files

```
CC = gcc
CFLAGS = -g -fno-omit-frame-pointer

HEADERS = myheader.h

%.o: %.c $(HEADERS) Makefile
        $(CC) $(CFLAGS) -c $< -o $@

testmake: main.o welcome.o check_args.o
        ${CC} $(CFLAGS) -o testmake main.c welcome.c
check_args.c

clean:
        @rm -f  main.o welcome.o check_args.o testmake *~
```

This indicates the first term in the dependency list

This indicates the left side of the :

With this the line is executed but not printed

that's all, have fun