1 Compiling with GNU make

In spite of its simplicity, the shell script `mymake` used for building the executable program has two disavantages. First, each source file is compiled when the script is launched. For big project, compiling all files could take a lot of time and this time could be an issue if only one source file has been changed since the last compilation. Secondly, the compilation command must be repeated in the script as many time as there are source files. Besides, new compilation commands must be added if new source files are created. The manual writing and management of this script should be painful in the context of big project.

To tackle these two disavantages, project building can be performed by using an advanced configuration file containing generic and compact compilation instructions. This kind of configuration file is called `makefile`. Numerous programs allow to interpret the makefile and to launch automatically the compilation sequence: `GNU make` (called also `gmake`), `nmake`, `tmake` ... and, unfortunately, each corresponding makefile has a specific syntax. The following explanations are based on the example of the most popular tool: `GNU make`.

1.1 Some words about the program `GNU make`

The `GNU make` tool is usually included in every LINUX distributions and it is fully operational on LXPLUS session of the students. The corresponding executable program is called `gmake` or simply MAKE. To check the presence of this program, you can issue the command below at the shell prompt: the release version must be displayed at the screen.

```
bash$make␣-v
```

By default, `GNU make` will look for a makefile called `Makefile` or `makefile`. The next sections of this document are devoted to the syntax of this file.

1.2 Minimal makefile

Here is explained the simplest way to write a makefile. For explaining the syntax, let has consider the example of a project made up of a main source file called `main.cpp` which use two classes described in the header/source files `file1.h`, `file1.cpp`, `file2.h` and `file2.cpp`. Building an executable program called `main` can be performed with the following makefile:

```
# Makefile example

all: main

main.o: main.cpp
	g++ -W -Wall -ansi -pedantic -o main.o -c main.cpp

file1.o: file1.cpp file1.h
	g++ -W -Wall -ansi -pedantic -o file1.o -c file1.cpp

file2.o: file2.cpp file2.h
	g++ -W -Wall -ansi -pedantic -o file2.o -c file2.cpp

main: main.o file1.o file1.h file2.o file2.h
	g++ -o main main.o file1.o file2.o

clean:
	rm -rf *.o main
```

Listing 1: A simple makefile

Like for shell script, lines begun with # are interpreted as comment lines. The file is made up of several instruction blocks called *rules*. Each *rule* targets to compile a source file or to link the object files. The generic syntax for a *rule* is the following:

```
target: dependency1 dependency2 [...]
	instructions1
	instructions2
	[...]
```

When `GNU make` treats a *target*, it analyzes first the *dependencies*. If a *dependency* is a file, the program determines if this file has been changed since the previous compilation. If a *dependency* is a target specified in the makefile, the program checks if the target has been treated. In the case of one dependency has changed or has to be rebuilt, `GNU make` treats the *target* before and execute the *instructions*. In the other case, the instructions are skipped. **Beware: instructions are preceded by a tabulation character (and not by space characters).**

To launch `GNU make` and to interpret the makefile, just type the following command at the shell prompt:

```
bash$make
```

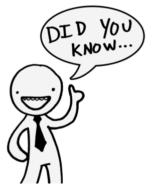
`GNU make` looks for the makefile and treats the first rule specified in the makefile. Usually it is called *all*. Of course, a given target could be specified to the program by set the target name as an argument of `make` command. This is the example of application to the target `main`:

```
bash$make␣main
```

We focus the user that the following makefile contains a specific rules called `clean`. It is very useful to remove files produced by *g++* (object files *.o and executable program) in order to back to the original source.

```
bash$make␣clean
```

- **By analyzing the example above, write a makefile adapted to the programming project.**
- **Clean your project with the makefile rule `clean` (a copy of the source file must be saved in a safe place in case of an unexpected deletion due to a bug in the makefile).**
- **Compile the project with the makefile.**
- **Check that if only one file is changed, only source depending on this file are treated in the next `GNU make` run.**



Some developers prefer splitting the `clean` target into two targets: `clean` for removing only temporary files (object files) and `mrproper` for removing all compiler produced files (object files and executable).

1.3 Enriched makefile

The previous makefile example is not really automated. In the next example, internal variables are used and allow to write compact and generic rules.

```makefile
# Makefile example using variable

CC=g++

CFLAGS  = -W -Wall -ansi -pedantic
SRCS = $(wildcard *.cpp)
HDRS = $(wildcard *.h)
OBJS = $(SRCS:.cpp=.o)
EXEC=main


all: $(SRCS) $(EXEC)

$(EXEC): $(OBJS) $(HDRS)
	$(CC) $(LDFLAGS) $(OBJS) -o $@

%.o: %.cpp %.h
	$(CC) $(CFLAGS) -c $< -o $@

clean:
	rm -f *.o $(EXEC)
```

Listing 2: an automated makefile

Definition of variables follows the scheme *VARIABLE* = *value*. The list of variables used in the analysed makefile can be found below. Of course, the user can define his/her own variables.

- *CC*: compiler command
- *CFLAGS*: compiler options
- *SRCS*: list of source files (*.cpp).
- *HDRS*: list of header files (*.h).
- *OBJS*: list of object files (*.o).
- *EXEC*: name of the executable program to create.

To access the content of a variable, the syntax is: `$(VARIABLE)`. For information, the special value `$(wildcard *)` is very useful because it allows to extract a list of files from the local folder safisfying a given criterion.

Then there are also some special variables, internal to `GNU make` which can be used in the different *rules*. The two such variables used in the example are very powerful:

- *$@*: name of the *target*.

- $¡: name of the first dependency.

Finally, repeating rule definition could be avoided by using automated rules. Thus, the following rule is applied to every file ended with '.o'. The character % replace the name of the files.

```
%.o: %.cpp %.h
	commands1
	commands2
	[...]
```

- **Adapt (if necessary) the automated makefile to the programming project.**
- **Compile your program with the obtained makefile**