tuto-make-main-getopt

November 21, 2024

1 Build with gcc, Makefile and main arguments

C programming - Muriel Pressigout - V2024-07-10

Ceci est un notebook, les cellules sont éditables. Une cellule commençant par $ est un texte qui correspond à une commande à tester dans le terminal. Vous pouvez copier-coller facilement les commandes demandées dans un terminal et y écrire vos résultats. C'est un sujet-feuille de réponse.

Pour exporter ce document en pdf : File -> Export Notebook as… -> PDF.

NB : vous pouver ouvrir ce fichier dans un navigateur sous Google colab, ce qui permet de lire et écrire dans ce fichier sans installation au préalable. Cependant vous n'aurez pas l'export en pdf.

Table of contents

Open a terminal in the same directory than this notebook (File->New->Terminal). Let note that any cell beginning with :

```
$
```

means that the text following the `$` character is a command to be run in this terminal you have just opened.

For example, run in this terminal to check the content of your current directory:

```
$ ls
```

Check you have : * tuto-make-main-getopt.ipynb * README-Jupyter-Markdown.md * README-Jupyter-Markdown.pdf * images * prod-cons * main-arg

1.1 1 Build with GCC

The GNU Compiler Collection provides different tools to build projects in C, C++,... For the C language, it provides the `gcc` command that compiles code and builds the program. It is widely used on Unix-like systems.

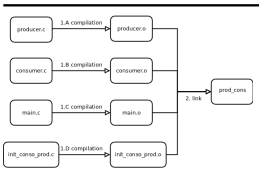
1.1.1 1 Example

Let's go into `prod-cons` directory :

```
$ cd prod-cons
```

There is C code source with a main function. To get an executable file, we need two steps : 1. **compilation** (C text -> assembly) 2. **linking** (put together all assembly files)

Only one has a main function : `main.c`. The project can be synthetized in this simple graph :



[Figure 1 : Build graph for `prod_cons`]

So the **compilation step** is :

```
$ gcc -c producer.c
```

```
$ gcc -c consumer.c
```

```
$ gcc -c main.c
```

```
$ gcc -c init_conso_prod.c
```

Remember that the order of these compilation commands is not important since a source file is compiled independantly from the other ones.

Now the **link step** to get the binary program `prod_cons` :

```
$ gcc init_conso_prod.o main.o consumer.o producer.o -o prod_cons
```

Run the program :

```
$ ./prod_cons
```

1.2 2 Make and Makefile

2.1 First Makefile **make** is a program that manages the build process of **targets**. A target can be : * an object file (file.o) * an executable file (as `prod_cons` in the previous example) * a library (this case will be studied in `S6 - operating systems`) * ans so many cases you will discover latter

Therefore, **make** is very useful to manage a code project where the objective is to build a program. For example, we have 5 targets in Figure build_graph : the 4 objects files .o and the binary program `prod_cons`.

To build each target, we know what we need : * for `producer.o` we need `producer.c` * for `consumer.o` we need `consumer.c` * for `main.o` we need `main.c` * for `init_conso_prod.o` we need `init_conso_prod.c` * for `prod_cons` we need `producer.o`, `producer.o`, `consumer.o`, `main.o` and `init_conso_prod.o`

These "needs" all called **dependencies** : `producer.o` depends from `producer.c` and so on. There are illustrated by the arrows in Figure build_graph

Once we know what we need as data, we need the command that builds the target. For example, to build `producer.o` , we need the command `gcc -c producer.c`

So, for a target, we need to know the dependencies (inputs) and the command using the dependencies to build it. The target name + dependencies + command is called a **rule** and is always in this form:

```
targetname: dependency1 dependency2
    command
```

Dependencies or command can be empty.

Let note that the line with the command **starts ALWAYS with a tabulation** and only a tabulation

All the rules are written in a file which is called **Makefile** by default.

For example, the Makefile corresponding to the Figure build_graph is :

```
producer.o: producer.c
    gcc -c producer.c

consumer.o: consumer.c
    gcc -c consumer.c

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

init_prod_conso.o: init_prod_conso.c
    gcc -c init_prod_conso.c
```

```
prod_conso: producer.o consumer.o main.o init_prod_conso.o
    gcc producer.o consumer.o main.o init_prod_conso.o -o prod_conso
```

It is provided in the `prod-cons` directory.

To test it :

```
$ rm *.o prod_cons
```

```
$ make prod_cons
```

```
$ ./prod_cons
```

In a single make call, the binary is produced. It has detected that it requires `producer.o`, `producer.o`, `consumer.o`, `main.o` and `init_conso_prod.o`. Since they were not present, **make** tried to find a rule to build them. Since a rule for each of them is defined, it calls them successively.

The dependencies are used to know if it is necessary to run the command associated to the target. If at least one dependency is newest that the target, the command is run. Retry :

```
$ make prod_cons
```

make should have told you there is nothing to do : `prod_cons`is built and none of its dependencies content have been modified since it has been built.

Now modify `init_prod_cons.c` to add a test in the function `isFull` to return -1 if the pointer p is NULL. Save and retry :

```
$ make prod_cons
```

Q1: How many object(s) file(s) have been rebuilt? Why? How many target(s) have been rebuilt? Save this Makefile as `Makefile.Q1`

R1: 4 objects files have been rebuilt, because we have made a change in init_conso_prod.h, which has dependencies on all 4 objects files. Similarly, all 5 target files have been rebuilt

You have studied your first Makefile. But it is not correct yet for at least 2 reasons : * not all dependencies are described -> the program is not rebuilt in some cases it should have been rebuilt (The Makefile does not account for header file dependencies. For example, if producer.c includes producer.h, modifying producer.h will not trigger a recompilation of producer.o) * good practices are not applied -> Makefile and C program prone to errors, hard to maintain

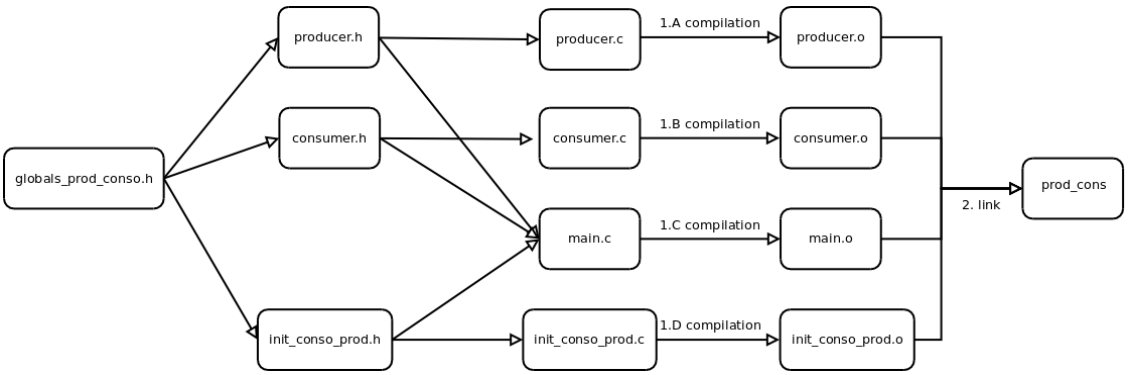
We will fix these issues in the following exercises

1.2.1 2.1 Second Makefile : add header dependencies

2.1.1 Second Makefile : add header dependencies : first version Change the value of NBPRODUCT in `globals_prod_conso.h` and call make to rebuild the program:

```
$ make prod_cons
```

Nothing is rebuilt :'(We need to explain that object files depends from the header files included in C source files. In fact, the complete dependency graph of this program is :



[Figure 2 : Complete dependency graph `prod_cons`]

Q2 : Change your Makefile to add the missing header dependencies. **Always add what is strictly necessary (during the exam, you will be penalized if not so)!** There are two ways to check it is ok: 1. when trying to rebuild `prod_cons`, a change in the NBPRODUCT value entails rebuilding every object files and the binary program 2. Ask a teacher. Mandatory for your first Makefile however always proceed to 1. before

Save this Makefile as `Makefile.Q2`

2.1.2 Second Makefile : add header dependencies using -MMD option When using **gcc** to build a program, one can use the compiler option **-MMD** that computes automatically all dependencies for object files. Try :

```
$ gcc -c -MMD producer.c
```

A file named `producer.d` has been created and you can check it contains the dependencies for `producer.o`

To include this kind of file, you have to add in your Makefile

```
include $(wildcard *.d)
```

Q3 : Change your Makefile to replace all the header dependencies by the use of **-MMD** compiler option 1. when trying to rebuild `prod_cons`, a change in the NBPRODUCT value entails rebuilding every object files and the binary program 2. Ask a teacher. Mandatory for your first Makfile however always proceed to 1. before

This way to proceed with header dependencies is better when using **gcc** Save this Makefile as `Makefile.Q3`

Now you have a Makefile that is complete but that does not respect the good practises in Makefile writing and in C programming. We will fix this. **Let note that good practises are taken into account when evaluating a Makefile**

1.2.2 2.3 Good practices in Makefile for C programming

When compiling and linking to build a program, it is good to have the warnings the compiler and linker send. In most of the cases, it points to an error :-)

Q4 : Modify your Makefile to add **-Wall** and **-Wextra** compiler options. Delete all object files and `prod_cons` program : when rebuilding the program, a C compiler warning should occur.

Let note that if you want to debug, you should also add **-g** compiler option.

Save this Makefile as `Makefile.Q4`

1.2.3 2.4 Good practices in Makefile : use common variables

To have a Makefile that is easy to maintain and to deploy on different configurations (change OS, change compiler,…), some variables are defined at the beginning of the Makefile: * CC stores the command for the compilation step (here : gcc -c) * CFLAGS stores the compiler options (here : -Wall -Wextra -MMD) * LD stores the command for the linking step (here : gcc) * LDFLAGS stores the link options (none here) : for example, path to a library * LDLIBS stores the third party library names (none here), for example if linking with the math library : -lm

As a consequence, your Makefile should begin with :

```
CC=gcc -c
CFLAGS= -Wall -Wextra -MMD
LD=gcc
LDFLAGS=
LDLIBS=
```

Q5 Add this text to your Makefile and modify the rules to use them. For example, producer.o rule becomes:

```
producer.o: producer.c
    $(CC) $(CFLAGS) producer.c
```

Save this Makefile as `Makefile.Q5`

1.2.4 2.5 Good practices in Makefile : use make variables

These variables are specific to Makefile writing and are used to make Makefile more generic (and therefore more copy-pasteable):

Variable	Meaning
`$@`	The target name
`$<`	The first dependency name
`$^`	The dependencies list
`$?`	The list of dependencies that have been more recently modified than the target
`$*`	The target name without its suffix

Q6 Add this text to your Makefile and modify the rules to use them. For example, producer.o rule becomes:

```
producer.o: producer.c
    $(CC) $(CFLAGS) $<
```

Save this Makefile as `Makefile.Q6`

1.2.5 2.6 Good practices in Makefile : use pattern

Now you should see that all rules for object files look similar.

Q7 Replace all rules for object files by this one that uses a pattern. It says that every target finishing with.o has a dependency with the same name with the suffix .c:

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

Save this Makefile as `Makefile.Q7`

2.7 Good practices in Makefile : use phony rules all and clean

In fact, when you type

```
$ make
```

make try to build the first rule found in the Makefile. As a consequence, the first rule of a Makefile is ALWAYS a rule named `all` whose dependencies are the list of binary programs and libraries handled by the Makefile. Here we have only one program so the rule is:

```
    all: prod_cons
```

It is called a phony rule because no file is directly produced by this rule.

Q8 Add this rule at the begining of your Makefile. Change the NBPRODUCT value and build your program with only this command:

```
$ make
```

Q9 Add a `clean` rule that deletes all files created by the Makefile: dependency files .d, object files .o and the program(s).

[Figure 3 : clean rule]

The command to delete files is **rm** so it should be like this:

```
    clean :
            rm *.d *.o
            rm prod_cons
```

Test it :

```
$ make clean
```

```
$ make
```

Save this Makefile as `Makefile.Q8`. Your Makefile should perfect now :-)

1.3 3. The arguments of the main function

When typing

```
$ gcc -c myecho.c -o myecho.o
```

we say that we run `gcc` with **arguments** : `-c myecho.c -o myecho.o`

1.3.1 3.1 Full prototype of the main function

The full prototype is of the main function is:

```
int main ( int argc , char * argv [] , char ** arge ) ;
```

with: * `argc` : number of arguments of the program, included the program name : argv[0] is the name of the program. As a consequence argc >=1 * `argv` : array of the arguments. An argument is stored as a string (char*) and there are delimited by at least a space* arge : array of the inherited environment variables. A NULL pointer ends the list. It will be studied in `S6-operating systems`

Example : myecho Go into `main-arg` directory and study the code of `myecho.c` that displays the arguments of the program. Compile the program `myecho` from `myecho.c` and test it:

`$ gcc -c myecho.c` //this compiles the source code (myecho.c) into an object file (myecho.o) without linking

`$ gcc myecho.o -o myecho` //This links the object file (myecho.o) to create an executable binary named myecho (explicitely by applying -o)

`$ ./myecho` //runs the compiled executable myecho without any command-line arguments

`$ ./myecho hi there! I am here !`//This runs the myecho program and passes command-line arguments to it.

Exercise : mysum **Q10**: Write a program called `mysum` that sums all the integers that are given as arguments to the program. As arguments are given a a string you will need to transform a string as an integer. atoi (ASCII to Integer) is not recommanded by ANSSI (cf cybersecurity rules for C) so use long strtol(const char *str, char **endptr, int base) as in:

```
#include <stdlib.h>

...

    char * endstring;
    int nb = strtotl(argv[i],&endstring,10);
```

Here are some expected results:

```
$ ./mysum

Sum = 0

$ ./mysum 2 3 8

Sum = 13

$ ./mysum 30 3 70 1000
```

```
Sum = 1003
```

1.3.2 3.2 Parsing the main arguments with getopt

When typing

```
$ gcc -c myecho.c -o myecho.o
```

we say that we run `gcc` with arguments : `-c myecho.c -o myecho.o` where `-c` and `-o myecho.o` are special arguments: they are **options**, *ie* arguments that defines the behavior of the command. `-c` is a simple option and says to `gcc` to act as a compiler. `-o` is an option with a value: `myecho.o` to change the default value of the output filename.

Most of the times the order of the options is not significant. It means that if we want to parse the arguments, there are so many combinations of options that it is not relevant to code it oneself. The function `getopt` is provided in the standard C library to do it. It is a good practice (aka mandatory :p) to use it as soon as a function has at least one option.

Example Build `test-getopt` from `from test-getopt.c` and test it:

```
$ gcc -c test-getopt.c -o test-getopt.o

$ gcc  test-getopt.o -o test-getopt

$ ./test-getopt hi there !

$ ./test-getopt hi there ! -a

$ ./test-getopt  -a hi there ! -b

$ ./test-getopt  -a hi -c here there ! -b
```

Explanation Stydy the code and let note that : * As usually, `getopt` is called in a **loop** (while) as shown in the example. When `getopt` returns -1, indicating no more options are present, the loop terminates. * In the loop, a switch statement is used to dispatch on the return value from `getopt`. In typical use, each case just sets a variable that is used later in the program. * As `getopt` sorts the array `argv` in order to put first the option parameters then non-option parameters, you should **never** launch any data processing in this loop. You should have only code related to the option detection in the switch statements.

- This process is based on some variables associated to the use of `getopt` :
  - char * optarg : set by `getopt` to point at the value the option value, for those options that accept values.
  - int optind : once `getopt` has found all of the option arguments, you can use this variable to determine where the remaining non-option arguments begin.
  - int opterr : If the value of this variable is nonzero, then `getopt` prints an error message to the standard error stream if it encounters an unknown option character or an option with a missing required argument. This is the default behavior. If you set this variable to zero, `getopt` does not print any messages, but it still returns the character ? to indicate an error.
  - int optopt : when `getopt` encounters an unknown option character or an option with a missing required argument, it stores that option character in this variable. You can use this for providing your own diagnostic messages.

In the provided example, the loop calling `getopt` is here followed by a second loop (for) to process the **remaining non-option arguments**.

Exercise Change the code of `test-getopt` to display the `printf ("aflag = %d, bflag = %d, cvalue = %s\n", aflag, bflag, cvalue);` only if the option -v (for verbose) is activated

Exercise Save your `mysum.c` as `myop.c`. Modify `myop.c` using`getoptso : * by default, it makes the sum of all arguments as previsouly * if option`-p‘ is used, it displays the product of all arguments

1.4 4. Summary of knowledge and skills after this tutorial and related lessons

1.4.1 4.1 Build process in C programming

- I know how to build an executable file from C source code using **gcc** command in a terminal
- I know that building a program from C code requires two main steps, **compilation** and **linking** and I understand what a compiler does and what a linker does.

1.4.2 4.2 Makefile

- I know what is a **rule**, a **target** and a **dependency** in a **Makefile**
- I know how to find the strictly necessary dependencies of a target and add them in a Makefile
- I know how to use variables to fulfill good practices in C programming and Makefile writing

1.4.3 4.2 Main arguments

- I know what is an **argument** to a program
- I know what the full prototype of the main function, what is **argc** and **argv** and how to use them to get the arguments of a program
- I know how to use **getopt** to parse the arguments of a program and detect automtically the **options**.

1.5 5. Perspectives

make is old. It is still often used to install simple third party librairies, for example for github. However its successors **cmake** or **autotools** are easier to use to describe a large project that can deploy on several systems. They rely on the same principles: identify targets and dependencies, apply the relevant commands when necessary. They will generate the Makefile adapted to your system.

Go back to the beginning