Makefile

How to really make files ?

Workshop Makefile

Corentin COUTRET-ROZET

Patricia MONFA-MATAS

Contents

History and definition ?

Make is a software that automatically builds files, often executable, or libraries from basic elements such as source code. It uses files called **Makefile** that specify how to build target files. Unlike a simple shell script, make executes commands only if they are needed. The goal is to arrive at a result (software compiled or installed, documentation created, etc.) without necessarily redoing all the steps. make is particularly used on UNIX platforms.

Did you know ?

In the **1970s**, the compilation of programs became increasingly long and complex, requiring many interdependent steps. Most of the systems then used are based on shell scripts, requiring repeating all the steps at the slightest correction. It is in this context that Make was developed by **Dr.Stuart Feldman**, in **1977** while working for Bell Labs. By managing dependencies between source and compiled files, Make allows you to compile only what is needed when editing a source file.

Later, other tools appeared allowing the automatic generation of configuration files (Makefile) used by Make. These tools allow to analyze dependencies or configuration

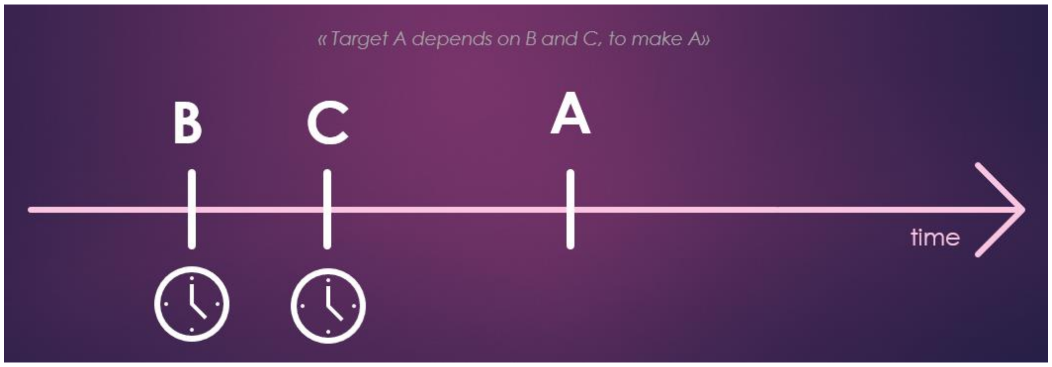
In **2003**, Dr. Feldman was awarded the **ACM** Award for Make Development.

Understand its functioning

General usage

The compilation process is broken down into elementary rules of the ***type "Target A depends on B and C, to make A you have to execute this sequence of commands"***. All these rules are placed in a file commonly called Makefile. Commands consist of elementary actions of compiling, editing links, generating code. Dependencies correspond to files, these can be source files, or intermediate results in the construction process. The target is usually a file but can also be abstract.

Make will then check that the B and C files exist and are up to date, that is, they do not have any modified dependencies after their creation, and otherwise **Make will start with the recursive construction of B and C. Make will then apply the creation commands of A as soon as no file of this name exists, or that file A is older than B or C.**



It is usual to use an all target summarizing the entire project, this abstract target having as dependency all the files to be built. Other targets are common: install to execute the project's installation commands, clean to erase all the generated files made, etc.

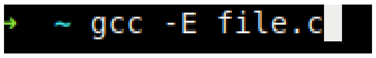
C/C++ usage

For compilation in C or C++ you will have to use a compiler such as gcc for C and a lot of other compilers. It is important to understand how it work and interacts with your Makefile.

A standard compilation in gcc is divided into 4 steps: **preprocessing**, **compiling**, **assembling**, and **linking**. To explain further, to compile and run a code in C, all the source code must be translated into an instruction sequence understandable by the machine.

To start with, during compilation, the **compiler starts preprocessing the directives**; that is #include #defines… In concrete terms, **gcc** will read and **insert the definitions within the code file**. The pre-processed file then ends up in a .i file. For example, a C code that prints using the printf function will need to include the stdio.h. header. The compiler will then read the entire file and insert it into your source code to generate **a .i file**.

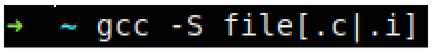
- You can stop the compilation process at the preprocessing step by using the -E flag such as:



```
➜  ~ gcc -E file.c
```

Then comes the **compilation**, which is often a bad term. The compiler will translate the file. i previously generated in file .s files. This file contains the **code translated into assembler**.

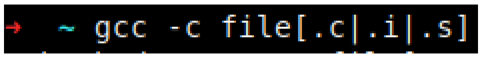
- You can stop the compilation process at the compilation stage by using the -S flag such as:



```
➜  ~ gcc -S file[.c|.i]
```

Then, the compiler goes to the assembly step. It will then translate the compiled file (.s) into an **object file (.o).**

- You can stop the compilation process at the assembly stage by using the -c flag such as:



```
➜  ~ gcc -c file[.c|.i|.s]
```

Finally, the compilation process ends on **linkage**. It will then **link all objects to create the expected executable**.

Implement a Makefile

Rules

A Makefile consists of rules. The simplest rule form is:

```
target [target ...]: [component ...]
[tab] command 1
.
.
.
[tab] command n
```

The "**target**" is most often a file to build, but it can also define an action (**delete**, **compile**...). The "**components**" are the **prerequisites** necessary for carrying out the action defined by the rule. In other words, the "components" are the targets of other rules that must be implemented before this rule can be implemented. The rule defines an action by a series of "**commands**". These commands define how to use the components to produce the target. Each command must be preceded by a tab character.
Components are not always targets for other rules, they can also be files needed to build the target file:
To note :

- **A minus sign (-),** specifying that errors should be ignored.
- **An arobase (@),** specifying that the command must not be displayed in the standard output before being executed.
- **A plus sign (+),** the command is executed even if make is called in a "do not execute" mode.

Variables and Macros

Definition

A Makefile can contain **macro definitions**. Macros are traditionally defined in uppercase letters:

```
MAKE        =  DEFINITION
```

Macros are most often referred to as variables when they contain only simple string definitions, such as CC = gcc. Environment variables are also available as macros. Macros in a Makefile can be overwritten by arguments passed to make. The command is then:

```
→  ~  make MACRO="value" [MARCRO="value" ...] TARGET[TARGET...]
```

Macros can be composed of **shell commands** using the low (‘): `PATH        =  'pwd'`

There are also 'internal macros' to make:

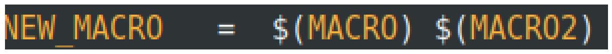
- **$@** refers to the target.
- **$?** : contains the names of all components more recent than the target.
- **$<** : contains the first component of a rule.
- **$^** : contains all the components of a rule.

Macros allow users to specify which programs to use or some custom options during the construction process. For example, the CC macro is frequently used in Makefile to specify a C compiler to use.

Expantion

To use a macro, you need to expand it by **encapsulating** it in $(). For example, to use the **CC macro**, you need to write $(CC). Note that it is also possible to use ${}.

For example:



```
NEW_MACRO   =  $(MACRO) $(MACRO2)
```

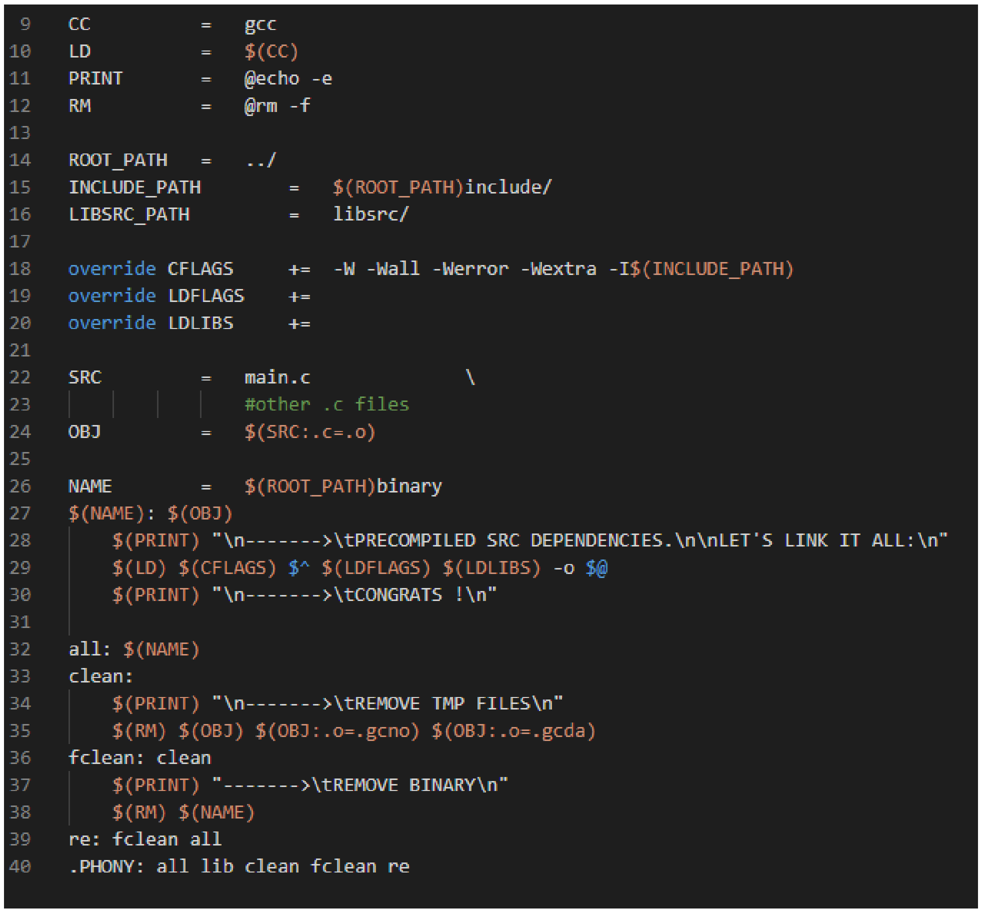
This line creates a new **NEW_MACRO** macro by subtracting the **MACRO2** content from the **MACRO** content.

The types of assignement

There are several ways to define a macro:

- The = is an assignment per reference (recursive expansion)
- The := is an appropriation by value (simple expansion)
- The ?= is a conditional assignment. It only affects the **macro** if the macro is not yet affected.
- The += is a concatenation assignment. It assumes that the macro already exists.

Example



```
CC          =   gcc
LD          =   $(CC)
PRINT       =   @echo -e
RM          =   @rm -f

ROOT_PATH   =   ../
INCLUDE_PATH        =   $(ROOT_PATH)include/
LIBSRC_PATH         =   libsrc/

override CFLAGS     +=  -W -Wall -Werror -Wextra -I$(INCLUDE_PATH)
override LDFLAGS    +=
override LDLIBS     +=

SRC         =   main.c              \
                #other .c files
OBJ         =   $(SRC:.c=.o)

NAME        =   $(ROOT_PATH)binary
$(NAME): $(OBJ)
    $(PRINT) "\n------->\tPRECOMPILED SRC DEPENDENCIES.\n\nLET'S LINK IT ALL:\n"
    $(LD) $(CFLAGS) $^ $(LDFLAGS) $(LDLIBS) -o $@
    $(PRINT) "\n------->\tCONGRATS !\n"

all: $(NAME)
clean:
    $(PRINT) "\n------->\tREMOVE TMP FILES\n"
    $(RM) $(OBJ) $(OBJ:.o=.gcno) $(OBJ:.o=.gcda)
fclean: clean
    $(PRINT) "------->\tREMOVE BINARY\n"
    $(RM) $(NAME)
re: fclean all
.PHONY: all lib clean fclean re
```

Other links…

- Introduction to Makefile: https://gl.developpez.com/tutoriel/outil/makefile/
- GCC usage : https://www.cs-fundamentals.com/c-programming/how-to-compile-c-program-using-gcc