How to use `make` – a short guide

ARQCP – DEI/ISEP 2015/2016

September 13, 2015

Contents

1 Introduction

The `make` program has as purpose the streamlining of a program's compilation, avoiding unnecessary recompilations but can be used to automate many other tasks. In alternative one can always use a custom script for building a software project, but the script will blindly recompile all the source files even if only one source file has been altered, or it will be very complex.

In this document several examples of how `make` can be used will be given, with usage simplicity as a main target. Some of the examples may not be the most concise or elegant way of using `make` but they are (in the author's opinion) the easiest way for a beginner. For a deepest study of `make` please read the manual or the references.

In order to build a software project `make` must posses a list of the dependencies between the various files. The list of files and asociated dependencies is stored in a file called `Makefile` and each software project has it's own folder, and on that folder a corresponding `Makefile`.

To know if a file is need of an update, `make` uses the file modification date. If a file does not exist, `make` considers it as in need of an update.

As the date of a file is important, one may have the need of changing the date of a file. In order to do that, one can use the `touch` command that updates the date of a file (or changes it – see manual). If the file given as an argument to the `touch` command does not exist, the file is created (without content – with a size of zero bytes).

2 `Makefile` anatomy

One can assume that a `Makefile` is a recipe. On the recipe we have the steps need to build our project, and `make` is smart enough to execute only the steps needed to (re)build our project.

Each file to build or step to be executed receives the name of *target*. Each target will have an entry on the `Makefile` with its name, the list of targets that it depends upon, and the list of commands that produce the required target. The command list is also known as rule.

If the command list does not produces a file, `make` executes it considering that the target has been updated.

The basic format of a `Makefile` entry is:

target… **:** *dependency …*
command
…

On the first line, the colon (**:**) ends the target list (one or more targets). After we have the list of dependencies (the files that our target depends upon).


The next lines that beging with a Tab are considered as command lines for the target building rule. The use of spaces instead of a Tab is a very common error[1] and one of the most infuriating feature of `make`, for someone leaning `make`.

Lines starting with # are considered comments, and do not end a rule. The end of a rule is marked by an empty line, that does not start with Tab or #.

3 Basic Examples

A very simple `Makefile` is the following:

```
1 # ex01/Makefile
2 test.txt:
3         cat Makefile > test.txt
```

The first time `make` runs the file creates the file `test.txt` and on the second time it runs, will tell us the file is up to date:

```
ex01# make
cat Makefile > test.txt
ex01# make
make: 'test.txt' is up to date.
```

Each line of the building rule will be executed by a different shell (`make` starts a new shell for every line). As an example we have the following two files:

```
1 # ex02/Makefile
2 test:
3         cd /tmp
4         pwd
5
```

```
1 # ex03/Makefile
2 test:
3         cd /tmp ; pwd
4
```

[1] Some text editors like `vi` or `emacs` avoid those errors when editing a `Makefile`.

While in the first example, the `pwd` is executed in the current folder, in the second case `pwd` will run on the `/tmp` folder.

```
ex02# make
cd /tmp
pwd
/Users/paf/unix/make/ex02
ex02#


ex03# make
cd /tmp ; pwd
/tmp
ex03#
```

The next example is a very simple *makefile* for the common `hello.c` program:

```
1 # ex04/Makefile
2
3 hello: hello.c
4         gcc -Wall hello.c -o hello
```

```
ex04# make
gcc -Wall hello.c -o hello
ex04# make
make: 'hello' is up to date.
ex04# touch hello.c
ex04# make
gcc -Wall hello.c -o hello
ex04#
```

4 Advanced examples

The creation of an executable by a C compiler can be split in two phases: the compilation phase including the object code generation (`.o`) and the creation of the desired executable file (made by the *linker*):

```
1 # ex05/Makefile
2
3 hello: hello.o
4         gcc -Wall  hello.o -o hello
5
6 hello.o: hello.c
7         gcc -Wall -c hello.c
```

```
ex05# ls
Makefile        hello.c
ex05# make
gcc -Wall -c hello.c
gcc -Wall  hello.o -o hello
ex05# ls
Makefile        hello               hello.c             hello.o
ex05#
```

The linking (joining the object code) phase is much faster than the proper compilation phase. This can used, if we split our source code in different modules and compile them separately, only joining them on the linking phase. This way, when working only in a small part of our source code, we will not need to recompile the full source code. In a simple example we will use a source file withe the `hello` function and another with the `main` function.

```
# ex06/Makefile

hello: hello.o  main.o
	gcc -Wall hello.o main.o -o hello

hello.o: hello.c hello.h
	gcc -Wall -c hello.c

main.o: main.c  hello.h
	gcc -Wall -c main.c
```

One can see that `make` executes only the required commands to create the final program:

```
ex06# ls
Makefile        hello.c                hello.h                main.c
ex06# make
gcc -Wall -c hello.c
gcc -Wall -c main.c
gcc -Wall hello.o main.o -o hello
ex06# make
make: 'hello' is up to date.
ex06# touch hello.c
ex06# make
gcc -Wall -c hello.c
gcc -Wall hello.o main.o -o hello
ex06# make
make: 'hello' is up to date.
ex06# touch main.c
ex06# make
gcc -Wall -c main.c
gcc -Wall hello.o main.o -o hello
ex06# touch hello.h
ex06# make
gcc -Wall -c hello.c
gcc -Wall -c main.c
gcc -Wall hello.o main.o -o hello
ex06# ls
Makefile        hello                hello.c                hello.h                hello.o
ex06#
```

On this example, we have an header file (.h) with the function declarations and the source code files.

```
/* hello.h */

#ifndef HELLO_H

#define HELLO_H
void hello(void);
#endif
```

```
/*  hello.c */
#include <stdio.h>
#include "hello.h"

void hello(void)
     {
        printf("Hello, world\n");
         }

```

```
/* main.c */
#include "hello.h"

int main(void)
     {
        hello();
        return 0;
         }
```

Is is not needed to include the hello.h on the hello.c source, but if that is done, the complier will detect possible mismatches between the two. The header file is *protected* against accidental multiple inclusion by the use a symbol (HELLO_H). If the symbol has not been defined (#ifndef), then the symbol is defined (#define) and the function definitions (only one in this example) are made. If the symbol has already been defined, it means that this header file has already been included so no further action is needed.

On the next example make is used to run programs, and that is very simple to do.

```
# ex07/Makefile

hello: hello.o  main.o
        gcc -Wall hello.o  main.o -o hello

hello.o: hello.c hello.h
        gcc -Wall -c hello.c

main.o: main.c  hello.h
        gcc -Wall -c main.c

run:        hello
        ./hello
```

One can see the behavior of make, compiling the program, in order to run it, if that is needed:

```
ex07# ls
Makefile        hello.c                 hello.h                 main.c
ex07# make
gcc -Wall -c hello.c
gcc -Wall -c main.c
gcc -Wall hello.o  main.o -o hello
ex07# make run
./hello
Hello, world
ex07# touch hello.h
ex07# make run
gcc -Wall -c hello.c
gcc -Wall -c main.c
gcc -Wall hello.o  main.o -o hello
./hello
Hello, world
ex07#
```

On the next example we have more samples of the versatility of make, variables and suffix definitions. Variables are esay to understand for someone used to programming languages and are used between brackets preceded by a dollar sign. Suffix definitions allows one to tell make how to obtain a certain file of a certain type from another file of a different type. First we tell make the desired suffixes and after we write the rules for their transformation.

```
1  # ex08/Makefile
2  # make variables
3  OBJFILES = hello.o main.o
4  PROGRAM = hello
5
6  # suffix rules
7  .SUFFIXES : .c .o
8  .c.o:
9          gcc -Wall -c $<
10 # how to transform a .c file into a .o file ;   $<   -- filename
11
12 hello:          ${OBJFILES}
13         gcc -Wall ${OBJFILES} -o ${PROGRAM}
14 # how to create the executable
15
16 hello.o: hello.c hello.h
17
18 main.o: main.c hello.h
19
20 # individual dependencies
21
22 run:         ${PROGRAM}
23         ./${PROGRAM}
24
25 clean:
26         rm ${PROGRAM} ${OBJFILES}
27
```

In the make rules we can use (among others) two special variables $< and $*, where the first is equal to the filename and the second one is the filename without the extension[2].

```
ex08# ls
Makefile        hello.c                 hello.h                 main.c
ex08# make
gcc -Wall -c hello.c
gcc -Wall -c main.c
gcc -Wall hello.o main.o   -o hello
ex08# make run
./hello
Hello, world
ex08# touch hello.h
ex08# make run
gcc -Wall -c hello.c
gcc -Wall -c main.c
gcc -Wall hello.o main.o   -o hello
./hello
Hello, world
ex08#
```

5 Flex

On the next example we have a program made with the help of flex. The rules.l has the flex rules, the file count.c has our source code in C. The program's objective is to count lines that that start with the login name of a student (an 'i' followed by several digits) or a teacher's login (at least two lower case letters). On the *makefile* we have an additional rule to run the program in a file called list.txt, to test easily our program.

```c
/*  ex09/count.c */
#include <stdio.h>

extern int      students, teachers;
extern int      yylex(void);

int main(int argc, char **argv)
{
        yylex();
        printf("Students: %d Teachers: %d \n", students, teachers);
        return(0);
}
```

```
#  ex09 Makefile

count:  count.o rules.o
        gcc -Wall count.o rules.o -lfl -o count

count.o: count.c
        gcc -Wall -c count.c
```

[2]This terminology is not 100% correct, but has been used for a better understanding. One can say that $* gets the filename, and erases the last characters up to the first dot that it finds (starting from the end of the string).

```
8
9
10 rules.o: rules.c
11         gcc -Wall -c rules.c
12
13
14 rules.c: rules.l
15         flex -t rules.l > rules.c
16
17
18 run: count
19         ./count <list.txt
```

```
1    /* ex09/rules.l  */
2           int students = 0;
3           int teachers = 0;
4
5  %%
6  ^i[0-9]+         students++;
7  ^[a-z][a-z]+     teachers++;
8  .
9  \n
```

The result of running the program will be the following:

```
ex09# make
gcc -Wall -c count.c
flex -t rules.l > rules.c
gcc -Wall -c rules.c
<stdout>:1084:17: warning: ‘yyunput’ defined but not used [-Wunused-function]
<stdout>:1125:16: warning: ‘input’ defined but not used [-Wunused-function]
gcc -Wall count.o rules.o -lfl -o count
ex09# make run
./count <list.txt
Students: 5 Teachers: 9
ex09#
```

on this case we can change our program or change the rules, and `make` will take care of everything. We will simulate filechanges with the `touch` command.

```
ex09# make
make: ‘count’ is up to date.
ex09# touch count.c
ex09# make run
gcc -Wall -c count.c
gcc -Wall count.o rules.o -lfl -o count
./count <list.txt
Students: 5 Teachers: 9
ex09# make
make: ‘count’ is up to date.
ex09# touch rules.l
ex09# make run
```

```
flex -t rules.l > rules.c
gcc -Wall -c rules.c
<stdout>:1084:17: warning: ‘yyunput’ defined but not used [-Wunused-function]
<stdout>:1125:16: warning: ‘input’ defined but not used [-Wunused-function]
gcc -Wall count.o rules.o -lfl -o count
./count <list.txt
Students: 5 Teachers: 9
ex09#
```

We can also use suffix rules for generating the `.c` files from the `.l` files. On thi case the dependencies are eliminated, because suffix rules also specify the dependencies.

```
# ex10/Makefile

# make variables
OBJFILES = count.o rules.o

#  suffix rules
.SUFFIXES : .c .o .l
.c.o:
        gcc -c -Wall $<

.l.c:
        flex -t $< > $*.c

# how to transform a .c into .o ;   $<   -- filename
# how to transform a .l into .c ;   $*   -- filename ( up to . )

count:        ${OBJFILES}
        gcc -o count ${OBJFILES} -lfl

# run the program

run:        count
        ./count <list.txt
```

6 Assembly Language

To show how *makefiles* can be used *assembly* we will some very small examples taken from (Blum, 2005).

On the first example we have only one source file (`cpuid.s`) generating an object file (`cpuid.o`) and finally an executable program.

```
#  ex11/Makefile

cpuid:  cpuid.o
        ld cpuid.o -o cpuid

cpuid.o: cpuid.s
        as cpuid.s -o cpuid.o


```

We can do the same thing with the help of suffix rules:

```
# ex12/Makefile

# with suffixes
.SUFFIXES : .s .o

.s.o:
        as $< -o $*.o

cpuid:  cpuid.o
        ld cpuid.o -o cpuid


```

Many times *assembly* langauge is used associated to a C language program. As an example we have an executable (`mainprog`) that results from two source code files, one in C (`mainprog.c`) and another in *assembly* language (`asmfunc.s`). While one is processed by `gcc`, the other is processed by `as`.

```
# ex13/Makefile

mainprog: mainprog.o asmfunc.o
        gcc -Wall mainprog.o asmfunc.o -o mainprog


mainprog.o : mainprog.c
        gcc -Wall -c mainprog.c


asmfunc.o : asmfunc.s
        as asmfunc.s -o asmfunc.o

```

Once again the *Makefile* can be simplified using suffix rules, like we can see in the next example.

```
# ex14/Makefile

# the suffixes that are important
.SUFFIXES :  .o .c .s


# how to transform a .c file into a .o file
.c.o:
        gcc -Wall -c $<

# how to transform a .s file into a .o file
.s.o:
        as $< -o $*.o


mainprog: mainprog.o asmfunc.o
        gcc -Wall mainprog.o asmfunc.o -o mainprog

```

References

Blum, Richard – *Professional Assembly Language*. Indianapolis, IN: Wiley Publishing, Inc., 2005. ISBN 0–7645–7901–0

Lehey, Greg – *Porting UNIX Software: from download to debug*. Sebastopol, CA: O'Reilly & Associates, 1995. ISBN 1–56592–126–7

Matthew, Neil; Stones, Richard – *Beginning Linux programming*. 4th edition. Wrox, 2007 ⟨URL: http://www.wrox.com/WileyCDA/⟩. ISBN 978–0–470–14762–7

Mecklenburg, Robert William; Oram, Andrew – *Managing Projects with GNU Make*. 3rd edition. Sebastopol, CA: O'Reilly, 2005, p. 280. ISBN 0596006101

——— ; Talbot, Steve – *Managing Projects with Make*. 2nd edition. Sebastopol, CA: O'Reilly & Associates, 1991, p. 150. ISBN 0–937175–90–0

Oram, Andy; Loukides, Mike – *Programming with GNU Software*. Sebastopol, CA: O'Reilly & Associates, 1996. ISBN 1–56592–112–7

Rehman, Rafeeq Ur; Paul, Christopher – *The Linux Development Platform*. Upper Saddle River, NJ: Prentice Hall PTR, November 2002 ⟨URL: http://www.informit.com/promotions/promotion.aspx?promo=135563⟩. ISBN 0130091154