How to use `make` – a short guide

ARQCP – DEI/ISEP 2023/2024

September 17, 2023

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 deeper study of `make` please read the manual or the references.

In order to build a software project `make` must possess a list of the dependencies between the various files. The list of files and associated prerequisites 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 it's name, the list of targets that it depends upon (prerequisites), and the list of commands that produce the required target. Each set of target, prerequisites and command list is also known as a *rule*.

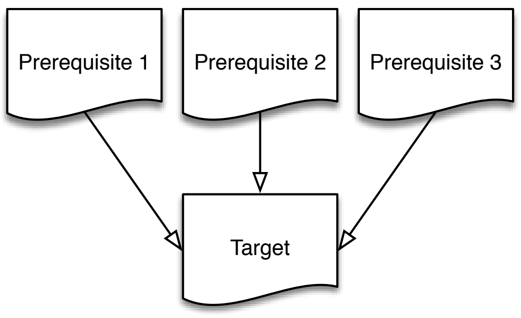

Figure 1: Target and Prerequisites

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...    :    prerequisite ...
                  command
                  ...
```

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

The next lines that begin 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 features of `make`, for a beginner.

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:

```
# ex01/Makefile
test.txt:
	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]Some text editors like `vi` or `emacs` avoid those errors when editing a `Makefile`.

```
# ex02/Makefile
test:
        cd /tmp
        pwd

```

```
# ex03/Makefile
test:
        cd /tmp ; pwd

```

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:

```
# ex04/Makefile

hello: hello.c
        gcc -Wall -Wextra -fanalyzer  hello.c -o hello
```

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

Up to now, we had only makefiles with only one target. When makefiles have more than one target, `make` only will try to build the first target on the makefile, and will ignore the others.

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*):

```
# ex05/Makefile

hello: hello.o
        gcc -Wall -Wextra hello.o -o hello

hello.o: hello.c
        gcc -Wall -Wextra -fanalyzer -c hello.c
```

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

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

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

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

```
ex06# ls
hello.c  hello.h  main.c  Makefile
guest@antix-dev:/media/sf_antiX/make_eng/ex06
ex06# make
gcc -Wall -Wextra -fanalyzer -c hello.c
gcc -Wall -Wextra -fanalyzer -c main.c
gcc -Wall -Wextra hello.o main.o -o hello
ex06# make
make: 'hello' is up to date.
ex06# touch hello.c
ex06# make
gcc -Wall -Wextra -fanalyzer -c hello.c
gcc -Wall -Wextra hello.o main.o -o hello
ex06# make
make: 'hello' is up to date.
```

```
ex06# touch main.c
ex06# make
gcc -Wall -Wextra -fanalyzer -c main.c
gcc -Wall -Wextra hello.o main.o -o hello
ex06# touch hello.h
ex06# make
gcc -Wall -Wextra -fanalyzer -c hello.c
gcc -Wall -Wextra -fanalyzer -c main.c
gcc -Wall -Wextra hello.o main.o -o hello
ex06# ls
hello  hello.c  hello.h  hello.o  main.c  main.o  Makefile
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 -Wextra hello.o  main.o -o hello

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

main.o: main.c  hello.h
	gcc -Wall -Wextra -fanalyzer -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
hello.c  hello.h  main.c  Makefile
ex07# make
gcc -Wall -Wextra -fanalyzer -c hello.c
gcc -Wall -Wextra -fanalyzer -c main.c
gcc -Wall -Wextra hello.o  main.o -o hello
guest@antix-dev:/media/sf_antiX/make_eng/ex07
ex07# make run
./hello
Hello, world
ex07# touch hello.h
ex07# make run
gcc -Wall -Wextra -fanalyzer -c hello.c
gcc -Wall -Wextra -fanalyzer -c main.c
gcc -Wall -Wextra 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.

```
# ex08/Makefile
# make variables
OBJFILES = hello.o main.o
PROGRAM = hello

# suffix rules
.SUFFIXES : .c .o
.c.o:
	gcc -Wall -Wextra -fanalyzer -c $<
```

```
# how to transform a .c file into a .o file ;   $<   -- filename

hello:         ${OBJFILES}
        gcc -Wall -Wextra ${OBJFILES} -o ${PROGRAM}
# how to create the executable

hello.o: hello.c hello.h

main.o: main.c hello.h

# individual dependencies

run:        ${PROGRAM}
        ./${PROGRAM}

clean:
        rm ${PROGRAM} ${OBJFILES}

```

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

[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).

5 Assembly Language

To show how *makefiles* can be used in *assembly* we will use 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.

```
#  ex09/Makefile

cpuid:  cpuid.o
        gcc -Wall -Wextra cpuid.o -o cpuid

cpuid.o: cpuid.s
        gcc -Wall -Wextra -c cpuid.s -o cpuid.o


```

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

```
# ex10/Makefile

# with suffixes
.SUFFIXES : .s .o

.s.o:
        gcc -Wall -c $<

cpuid:  cpuid.o
        gcc -Wall 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`.

```
# ex11/Makefile

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


mainprog.o : mainprog.c asmfunc.h
        gcc -Wall -Wextra -fanalyze -c mainprog.c


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

run: mainprog
        ./mainprog
```

Once again the *Makefile* can be simplified using suffix rules, as 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 -Wextra -fanalyzer  -c $<

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


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

run:  mainprog
	./mainprog


clean:
	rm -f mainprog *.o

```

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