Makefile

Σέργιος - Ανέστης Κεφαλίδης
Κωνσταντίνος Νικολέτος
Κώστας Πλας

What is a Makefile and why should we use it?

- Imagine having a large C program with many files and hundreds of line of code. Making a minor change and recompiling the program using gcc alone, becomes difficult and needlessly time consuming. To solve this problem in C/C++ problems we use Makefiles.
- Makefiles are used to help decide which parts of large program need to be recompiled.
- Makefiles save us time, energy and help organize our code. Every program you write should be accompanied by a Makefile!

The basics of Makefile: Syntax

Makefiles consist of rules. Rules generally look like this:

```
targets: prerequisites
<tab>command1
        command2
        command3
```

- ➔ targets: target file names, separated by spaces. You should use only one per rule.
- ➔ command: steps that will be executed to make the target. The commands should ALWAYS start with a tab character, not spaces.
- ➔ prerequisites: file names, separated by spaces. These files must exist, before the commands run. They are usually called dependencies.

Using Makefile to compile C programs (1)

All we need to know to create a working Makefile, is writing and using rules. Let's take for a example a c program consisting of 3 source files and 2 header files:

```c
#include <stdio.h>
#include "foo.h"
#include "fun.h"

int main(){
    foo(1,3);
    int f = fun(1,3);
    printf("FUN: %d\n",f);
    return 0;
}
```

main.c

```c
#include <stdio.h>
#include "fun.h"

int fun(int a,int b){
    return a+b;
}
```

fun.c

```c
#ifndef FUN_H
#define FUN_H

int fun(int a,int b);

#endif //FUN_H
```

fun.h

```c
#include <stdio.h>
#include "foo.h"

void foo(int a,int b){
    printf("FOO:
%d",a+b);
}
```

foo.c

```c
#ifndef FOO_H
#define FOO_H

void foo(int a,int b);

#endif //FOO_H
```

foo.h

Using Makefile to compile C programs (2)

Even with a few files, like in this, example compilation by hand (using gcc in a terminal) becomes complicated and messy. Let’s start creating our Makefile to deal with the problem:

- We want main.c to be the executable program.
- main.c needs foo.c and fun.c to be compiled.
- We need to combine all the above to create our Makefile.

Using Makefile to compile C programs (3)

Let's start by compiling the different .c files into objective files (.o files). Remember, that in order to compile a .c file (myprogram.c) into an objective file, we use the -c flag in gcc:

```
>$ gcc -c myprogram.c
```

The rules we need for the objective files are the following:

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

fun.o : fun.c
	gcc -c fun.c

foo.o : foo.c
	gcc -c foo.c
```

Using Makefile to compile C programs (4)

Having obtained the .o files we can now create the executable for our program. Let's name it main.

The rule for the executable will look something like this:

```
main : main.o fun.o foo.o
	gcc main.o fun.o foo.o -o main
```

We can add an additional rule for deleting the objective and executable files when we don't need them anymore:

The rule for the deletion will look something like this (we usually name this rule clean):

```
clean :
	rm -f main main.o fun.o foo.o
```

Using Makefile to compile C programs (5)

We can now combine all the different rules into one file, creating the Makefile for our project (Makefiles should always be named Makefile with a capital M).

```
main : main.o fun.o foo.o
	gcc main.o fun.o foo.o -o main

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

fun.o : fun.c
	gcc -c fun.c

foo.o : foo.c
	gcc -c foo.c

clean :
	rm -f main main.o fun.o foo.o
```

Makefile

Using Makefile to compile C programs (6)

To use the Makefile, all we have to do is type make and our program will be ready for execution:

```
>$ make
```

To remove the objective files and the executable, all we have to do is type:

```
>$ make clean
```

In general, you can use make with every rule you have written:

```
make [rule]
```

Adding More Rules (1)

Let's assume that we want to add a different executable named different_main.c, without changing main.c. Makefile helps us integrate more than one executables in our program, by adding different rules. In our case, all we have to do is add the following two rules:

```
different_main : different_main.o fun.o foo.o
	gcc different_main.o fun.o foo.o -o different_main

different_main.o : different_main.c
	gcc -c different_main.c
```

Adding More Rules (2)

The new Makefile will look like this:

```
main : main.o fun.o foo.o
	gcc main.o fun.o foo.o -o main

different_main : different_main.o fun.o foo.o
	gcc different_main.o fun.o foo.o -o different_main

different_main.o : different_main.c
	gcc -c different_main.c

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

fun.o : fun.c
	gcc -c fun.c

foo.o : foo.c
	gcc -c foo.c

clean :
	rm -f main different_main different_main.o main.o fun.o foo.o
```

Makefile

Adding More Rules (3)

By using:

```
>$ make different_main
```

Now different_main.c will be compiled as an executable. Using make main, will compile main.c (or simply using make, because main is the first rule that creates an executable).

Can we compile all rules for executables together?

Adding More Rules (4)

Yes, by adding a rule that that has as prerequisites all the executable rules:

```
all: main different_main
```

Using the command:

```
>$ make all
```

Will compile both main and different_main executables.

Making Makefiles easier to use (1)

- Is there any way to avoid writing all these rules, over and over again?
  - Yes, by adding variables and flags!
- Makefile provides:
  - flags for the compiler
  - variables
- By using flags and variables we can make the Makefile easier to read, write and understand.

Making Makefiles easier to use (2)

- How and why do variables and flags work?
  - Makefile provides a capability called *implicit rules*
- Through *implicit rules* we can run commands without the need to specify them in detail when we want to use them.
  - For example the C compilation rule that takes a .c file and produces a .o file!
- The built-in *implicit rules* use several variables in their recipes so that, by changing the values of the variables, you can change the way the implicit rule works. For example, the variable CFLAGS controls the flags given to the C compiler by the implicit rule for C compilation.
- For more information on implicit rules you can read the GNU manual

Using flags and variables in rules (1)

Some useful flags that we will use in the course are:

- CC: compiler for the C program.
- CFLAGS: flags to provide to the compiler.
- LDFLAGS: extra flags for the compiler, concerning the linker (e.g. link program with the math library).

Using flags and variables in rules (2)

Let’s use variables and flags to improve our Makefile. Starting with flags, we can set the following:

```
CC=gcc # use gcc to compile the program

CFLAGS= -g -Wall # various flags for the compiler, -g & -Wall should be used when developing your programs
```

Using flags and variables in rules (3)

Adding variables, we can simplify the rules we need for our program. Given our previous source files we can set the following variables:

```
PROGRAM = main # the target for the main executable
DIFFERENT = different_main # the target for the different_main executable

OBJS = main.o fun.o foo.o # objects prerequisites for main
DIFF_OBJS = different_main.o fun.o foo.o # objects prerequisites for different_main
```

Using flags and variables in rules (4)

The rules are no different than before! All we have to do to write them, is use the variables:

```
# rule for target main. To get the value of a variable use use its name inside this : $()
$(PROGRAM) : $(OBJS)
	$(CC) $(CFLAGS) $(OBJS) -o $(PROGRAM) # use tab NOT spaces for commands

# rule for target different_main
$(DIFFERENT) : $(DIFF_OBJS)
	$(CC) $(CFLAGS) $(DIFF_OBJS) -o $(DIFFERENT) # use tab NOT spaces for commands

# all rule
all : $(PROGRAM) $(DIFFERENT)

# clean rule
clean :
	rm -f $(PROGRAM) $(DIFFERENT) $(OBJS) $(DIFF_OBJS)
```

Our final Makefile

```
CC=gcc # use gcc to compile the program

CFLAGS= -g -Wall # various flags for the compiler

PROGRAM = main # the target for the main executable
DIFFERENT = different_main # the target for the different_main executable

OBJS = main.o fun.o foo.o # objects prerequisites for main
DIFF_OBJS = different_main.o fun.o foo.o # objects prerequisites for different_main

# rule for target main. To get the value of a variable use use its name inside this : $()
$(PROGRAM) : $(OBJS)
	$(CC) $(CFLAGS) $(OBJS) -o $(PROGRAM) # use tab NOT spaces for commands

# rule for target different_main
$(DIFFERENT) : $(DIFF_OBJS)
	$(CC) $(CFLAGS) $(DIFF_OBJS) -o $(DIFFERENT) # use tab NOT spaces for commands

# all rule
all : $(PROGRAM) $(DIFFERENT)

# clean rule
clean :
	rm -f $(PROGRAM) $(DIFFERENT) $(OBJS) $(DIFF_OBJS)
```

Makefile