Makefile: The first Foot Step

By : Subrat Kumar Ojha

Sr. SW Eng., P & M

Global Edge Software Limited

CONTENTS

Ch 1: The World without Makefile

Before we move into Makefile. Let us first have a small look how the world would have been without make file.

Assume that we have 4 files :

a. file1.c
b. file2.c
c. file3.c
d. header1.h

To compile this we can use the below instruction

```
cc  file1.c  file2.c  file3.c  -I  header1.h  -o  example1
```

Here we have 3 c files and one header file. So it's easy to compile. Let us consider the below situation.

- Assume we have 300 c files an 400 header files.
- We have 20 directories that has all the files in previous points.

Possible cases :

1. we can use **cc *.c -I *.h -o example 2**. Well this is ok if we want to have the executable to compile with all c files an header files.
2. We have 2 projects that uses same 20 dirs but exclude certain files as per the need. In that its cumbersome to compile.
3. We need to compile all file but we have made change to one c file. Can we achieve this without make file???

Here comes the actual usage of the utility program "make file". It's worth to note that make file is not limited to c files.

Ch 2: The Makefile: Introduction

It's a Build Automation Tool. The **make** command allows you to manage large programs or groups of programs. As you begin to write large programs, you notice that re-compiling large programs takes longer time than re-compiling short programs. Moreover, you notice that you usually only work on a small section of the program such as a single function, and much of the remaining program is unchanged.

Author : Stuart Feldman
Year : 1977

Assume that we have 4 files:

e. file1.c
f. file2.c
g. file3.c
h. header1.h

Suppose we modified one file2.c. Here the make file will check the modification time of all the files used for compilation as against the target (output program).

Modification time: = m.time.

Makefile follows the below rule: Makefile derives the **list of dependent files** and then follow the below rules.

```
If (! file_newly_created  || !target_doesnot_exists)
{
   m.time (file) >= m.time (target)  ➔ Recompile  the file & recreate the target.
}
Elseif (file_newly_created || !target_doesnot_exists)
{
  ⇨   compile the new file & recreate the target.
}
Else
{
   // do nothing.
}
```

Kindly note that output program can be object file , library or executable file.

Ch 3: The Makefile: Assigning Values

Lazy Set

```
VARIABLE = value
```

Normal setting of a variable - values within it are recursively expanded when the variable is used, not when it's declared

Immediate Set

```
VARIABLE := value
```

Setting of a variable with simple expansion of the values inside - values within it are expanded at declaration time.

Set If Absent

```
VARIABLE ?= value
```

Setting of a variable only if it doesn't have a value

Append

```
VARIABLE += value
```

Appending the supplied value to the existing value (or setting to that value if the variable didn't exist)

Examples :

Lazy Set

```
VARIABLE = 4 5 6 7
```

Immediate Set

```
VARIABLE := 4 5 6
```

Set If Absent

```
VARIABLE ?= 4
```

Append

```
VARIABLE += 8
```

Difference between Lazy Set vs Immediate Set :

Using `=` causes the variable to be assigned a value. If the variable already had a value, it is replaced. This value will be expanded when it is used. For example:

Using `:=` is similar to using `=`. However, instead of the value being expanded when it is used, it is expanded during the assignment. For example:

```
HELLO = world
HELLO_WORLD := $(HELLO) world!

# This echoes "world world!"
echo $(HELLO_WORLD)

HELLO = hello

# Still echoes "world world!"
echo $(HELLO_WORLD)

HELLO_WORLD := $(HELLO) world!

# This echoes "hello world!"
echo $(HELLO_WORLD)
```

Ch 4: Writing My first Makefile

Basic syntax:
target : **pre_requisite**
<tab> **<basic set of instructions>**

Suppose we have 3 file
1. file1.c & file2.c
2. header1.h

Create a file either called it Makefile or makefile.

Content of Makefile :

```
exec :file1.o file2.o

        gcc -o exec file1.o file2.o

file1.o :file1.c

        gcc -o file1.o -c file1.c

file2.o : file2.c header1.h

        gcc -o file2.o -c file2.c -I .
```

How to Compile:

make –C "dir to makefile" –f <file> <target name>.

File:

Default name accepted by make utility are makefile or Makefile. If user provides makefile_own, he needs to use –f option.

Dir:

Default dir accepted by make is current dir ".". However if users have makefile in other dir use –C , to indicate the make that please enter to "dir" and then continue.

Target :

By Default target are either of all, phony etc. However user can give their target.
then use : make <user_target>
like in our case : make exec or make file1.o

Ch 5: Working of Makefile

Makefile cannot work on its own. It can however search the first target if user does not have pre defined target while invoking.

Like in our example if we say make. It will be same as make exec.

But if we give make file1.o , it will only run file1.o commands.

let us see how “make exec” works.

exec ➔ file1.o file2.o

file1.o ➔ file1.c

file2.o ➔ file2.c header1.h

Hence make will first search for file1.o . Once done with file1.o , it will move to file2.o. So make always create a stack of dependencies and then execute from last.

Ch 6: Magic Variables of Makefile

$@ The file name of the target.

$< The name of the first dependency.

$* The part of a filename which matched a suffix rule.

$? The names of all the dependencies newer than the target separated by spaces.

$^ The names of all the dependencies separated by spaces, but with duplicate names removed.

$+ The names of all the dependencies separated by spaces with duplicate names included and in the same order as in the rule.

Ex of $@.

```
exec :file1.o file2.o
        gcc -o $@ file1.o file2.o
file1.o :file1.c
        gcc -o $@ -c file1.c
file2.o : file2.c header1.h
        gcc -o $@ -c file2.c -I .
```

Ex of $<.

```
exec :file1.o file2.o
        gcc -o $@ $< file2.o
```

Ex of $*.

exec :file1.o file2.o

gcc -o **$@ $*.o**

Ex of $?

exec :file1.o file2.o

gcc -o **$@ $? (Only take the updated pre-requisite compared to target).**

Ex of $^

exec :file1.o file2.o file1.o

gcc -o **$@ $^ (ignore the duplicates)**

Ex of $^

exec :file1.o file2.o file1.o

gcc -o **$@ $+ (Same as $^ but no ignore).**

Ch 7: Makefile Hacked

Suppose we have below example. How can we write our makefile.

Our_dir

|---dir1 → file1.c file2.c file3.c file4.c header1.h header2.h
|----dir2 → file5.c file6.c header3.h

Basic Makefile :

```
exec : file1.o file2.o file3.o file4.o file5.o file6.o
	gcc -o exec file1.o file2.o file3.o file4.o file5.o file6.o

file1.o : dir1/file1.c
	gcc -c dir1/file1.c -I dir1

file2.o : dir1/file2.c
	gcc -c dir1/file2.c -I dir1

file3.o : dir1/file3.c
	gcc -c dir1/file3.c -I dir1

file4.o : dir1/file4.c
	gcc -c dir1/file4.c -I dir1

file5.o : dir2/file5.c
	gcc -c dir2/file5.c -I dir2

file6.o : dir2/file6.c
	gcc -c dir2/file6.c -I dir2
```

Let us first create some macros to make it more readable and usable.

```
INC_DIR +=  dir1
INC_DIR +=  dir2
CC_OBJ := gcc -c
CC_EXE := gcc -o
TARGET := exec

OBJ_FILE += file1.o
OBJ_FILE += file2.o
OBJ_FILE += file3.o
OBJ_FILE += file4.o
OBJ_FILE += file5.o
OBJ_FILE += file6.o

SRC_DIR  =  dir1
SRC_FILE := $(SRC_DIR)/file1.c
SRC_FILE := $(SRC_FILE) $(SRC_DIR)/file2.c
SRC_FILE := $(SRC_FILE) $(SRC_DIR)/file3.c
SRC_FILE := $(SRC_FILE) $(SRC_DIR)/file4.c
SRC_DIR   =  dir2
SRC_FILE := $(SRC_FILE) $(SRC_DIR)/file5.c
SRC_FILE := $(SRC_FILE) $(SRC_DIR)/file6.c


$(TARGET) :  OBJ_TAR
      $(CC_EXE)  $(TARGET)  $(OBJ_FILE)
OBJ_TAR : $(SRC_FILE)
      $(CC_OBJ) $(SRC_FILE) -I $(INC_DIR)
```

Let us use the macros explained in Ch 6.

```
INC_DIR +=  dir2
INC_DIR +=  dir1
CC_OBJ := gcc -c
CC_EXE := gcc -o
TARGET := exec

OBJ_FILE += file1.o
OBJ_FILE += file2.o
OBJ_FILE += file3.o
OBJ_FILE += file4.o
OBJ_FILE += file5.o
OBJ_FILE += file6.o

SRC_DIR  =  dir1
SRC_FILE := $(SRC_DIR)/file1.c
SRC_FILE := $(SRC_FILE) $(SRC_DIR)/file2.c
SRC_FILE := $(SRC_FILE) $(SRC_DIR)/file3.c
SRC_FILE := $(SRC_FILE) $(SRC_DIR)/file4.c
SRC_DIR   =  dir2
SRC_FILE := $(SRC_FILE) $(SRC_DIR)/file5.c
SRC_FILE := $(SRC_FILE) $(SRC_DIR)/file6.c

$(TARGET) :  OBJ_TAR
     $(CC_EXE)  $^   $(OBJ_FILE)

OBJ_TAR : $(SRC_FILE)
     $(CC_OBJ)  $<    -I   $(INC_DIR)
```

Complicating the Makefile, little bit…. ☺.

```
INC_DIR :=  -I .
INC_DIR +=  dir1
INC_DIR +=  dir2
CC_OBJ := gcc -c
CC_EXE := gcc -o
TARGET := exec
REM := rm -rf

OBJ_FILE += file1.o
OBJ_FILE += file2.o
OBJ_FILE += file3.o
OBJ_FILE += file4.o
OBJ_FILE += file5.o
OBJ_FILE += file6.o

SRC_DIR  =  dir1
SRC_FILE := $(SRC_DIR)/file1.c
SRC_FILE += $(SRC_DIR)/file2.c
SRC_FILE += $(SRC_DIR)/file3.c
SRC_FILE += $(SRC_DIR)/file4.c
SRC_DIR   =  dir2
SRC_FILE += $(SRC_DIR)/file5.c
SRC_FILE += $(SRC_DIR)/file6.c

$(TARGET) : OBJ_TAR
      $(CC_EXE)  $(TARGET)  $(OBJ_FILE)


OBJ_TAR :
      $(foreach src,$(SRC_FILE), $(CC_OBJ) $(src) $(INC_DIR);)


clean :
      $(REM) $(OBJ_FILE) $(TARGET)
```