Source : Makefiles: 95% of what you need to know - Gabriel Parmer

1 Introduction

- It a build system.
  i) Automate the compilation and linking of source files into *exectutables*.
  ii) Recompilation on only the changed portion of source code and the portion dependant on the changed code.
  iii) Make maintaining the build system easier by avoiding redudant code in build system.
- Used by any compiled language.
  - C
  - C++
  - Fortran
  - Latex

1.1 Printing

- To print Text, one can use any of the following:

```
$(info This is an info text)
$(warning This is an warning text)
$(error This is an error text)
```

- Printing an `$(error)` will make the execution stop.

1.2 Variables

- Variables in MakeFile are defines as

```
VARNAME = VALUE
```

- They can be used with the help of dollar symbol ($).

```
PIVALUE = 3.14
$(info Checking the Variable $(PIVALUE))
```

1.3 Build Rule

- Every rule starts with a target (which is going to be built) followed by colon (:) followed by the dependencies the target require.
- The next line may or may not contain commands to exectue to get the dependencies.
- The dependencies may be
  - Files
  - Other Build Rules

1.3.1 Build rule depending on files

```
my_rule1 : source.c test.c
    gcc -c source.c -o source.o
```

- In this case, the *my_rule1* build rule will run only when *source.c* and *test.c* are available.
- Even if *test.c* is not used in the command, since it is present as dependencies, *test.c* must be available for *my_rule* to be executed.

1.3.2 Build rule depending on other build rules

```
my_rule1 : my_rule2

my_rule2 : my_rule3
    gcc source.o -o source.bin

my_rule3 : source.c
    gcc -c source.c -o soruce.o
```

- In this case, *my_rule1* depends on *my_rule2* which in turn depends on *my_rule3* which in turn depends on *source.c*.
- Here, only *my_rule2* and *my_rule3* has commands and those executed, when their correspoding dependancies are met.

1.4 Build Rule with *wildcards*

- Here, *wildcards* are denoted by *%*.
- *%* denotes to any text.

```
%.o : %.c
```

- **The above build rule indicates, to build anytext.o, then the dependency is the same-text.c**.
- This can be used to build any **.o*.

1.4.1 Make special varaibles

- Used inside the command below the build rule.
- @ takes the left hand side of build rule.
- ^ takes the right hand side of build rule.

```
    source.o : source.c
        gcc -c $^ -o $@
```

- The above lines are same as:

```
    source.o : source.c
        gcc -c source.c -o source.o
```

- **These special varaibles along with wildcards can be used to create a generic building of *.o files from *.c files**.

```
%.o : %.c
    gcc -c $^ -o $@
```

- But, they can run independently without a target, hence

```
OBJFILES = test.o source.o

all : $(OBJFILES)

%.o : %.c
    gcc -c $^ -o $@
```

- Above, for every object files needed by *all* build rule, the corresponding object files are created using *%.o* build rule.

1.5 Building

- Builing is done with *make*.
- This will run the first build rule and not *all* build rule.
- Only if the dependencies changes, the build rules changes and this is propagated to other build rules which uses the dependencies.
- **Generarlly, MakeFile doesn't look for program oriented dependencies,like header files.**
- **@ symbol can be used to not to print the command.**

2 Simple MakeFile Example

A sample example MakeFile used for building code inside *./Simple_MakeFileSystem* is shown below:

```
1  # A Simple MakeFile to build C programs
2
3  # C files to be compiled
4  CFILES = prog.c mod1.c mod2.C
5  OBJECTS = prog.o mod1.o mod2.o
6  EXEC = prog
7
8
9  # Compiler to use ; Can be changed for cross compilation
10 CC =  gcc
11 # Include directories to be used
12 INCDIRS = ./Include
13 # C Flags to be given for compilation
14 CFLAGS = -Wall -Wextra -I $(INCDIRS)
15
16 # make command will run the first build rule, so generally the first one is all
17 all : $(EXEC)
18
19 # prog is the final executable  to be created which isdepeneded on the object files
   ↪  to be available
20 $(EXEC) : $(OBJECTS)
21         $(CC) $^ -o $@
22
```

```
%.o : %.c
        $(CC) $(CFLAGS) -c $^ -o $@

# The @ symbol means while running 'make clean', the command won't be shown
clean:
        @rm -rf $(OBJECTS) $(EXEC)


# added test to run the program
test : all
        $(info Running the program ...)
        $(info )
        ./$(EXEC)
```

2.1 Disadvantages

I) The *make* command recompiles the code where there is a change in .c files as only those are the depeneded for creating .o files.

- But,when only the header content changes the command doesn't compile.

II) Also, the include directory is liminted to one.

III) In this, we have to give the .c and .o files.

3 Solving 2.1 - I)

- The particular problem, of header file not being seen can be solved by generating files (.d) that encodes make rule for .h dependencies.
- This can be done by adding the flags to the GCC compiler

```
    -MP -MD
```

- Now, this will create .d files for the .c files which has dependend over header files so as to be used by make build system.
- We also need to clean the .d files.
- Now, the new MakeFile is shown below:

```
# A Simple MakeFile to build C programs

# C files to be compiled
CFILES = prog.c mod1.c mod2.C
OBJFILES = prog.o mod1.o mod2.o
DEPFILES = prog.d mod1.d mod2.d
EXEC = prog


# Compiler to use ; Can be changed for cross compilation
CC =  gcc
# Include directories to be used
INCDIRS = ./Include
# C Flags to be given for compilation
CFLAGS = -Wall -Wextra -I $(INCDIRS) -MD -MP

```

```
# make command will run the first build rule, so generally the first one is all
all : $(EXEC)

# prog is the final executable  to be created which isdepeneded on the object files
 ↪  to be available
$(EXEC) : $(OBJFILES)
        $(CC) $^ -o $@

%.o : %.c
        $(CC) $(CFLAGS) -c $^ -o $@

# The @ symbol means while running 'make clean', the command won't be shown
clean:
        @rm -rf $(OBJFILES) $(DEPFILES)

# The @ symbol means while running 'make clean all', the command won't be shown
cleanall: clean
        @rm -rf $(EXEC)

# added test to run the program
test : all
        $(info Running the program ...)
        $(info )
        ./$(EXEC)
```

4 Solving 2.1 - II)

- += is used to append values to a varaible.
- To solve the issue of multiple include directories, we use foreach loop whose syntax is:

```
$(foreach var, range, cmds)
```

- An example to print numbers from 1 to 4 using foreach loop:

```
range = 1 2 3 4
$(foreach i, $(range), $(info $(i)))
```

- Using the above information, now the make file can be made to have multiple include directories:

```
# A Simple MakeFile to build C programs

# C files to be compiled
CFILES = prog.c mod1.c mod2.C
OBJFILES = prog.o mod1.o mod2.o
DEPFILES = prog.d mod1.d mod2.d
EXEC = prog


# Compiler to use ; Can be changed for cross compilation
CC =  gcc
# Include directories to be used
INCDIRS += .
INCDIRS += ./Include
```

```
# C Flags to be given for compilation - (adding foreach for iterating through each
↪  include directory
CFLAGS = -Wall -Wextra  $(foreach includedir, $(INCDIRS), -I $(includedir)) -MD -MP

# make command will run the first build rule, so generally the first one is all
all : $(EXEC)

# prog is the final executable  to be created which isdepeneded on the object files
↪  to be available
$(EXEC) : $(OBJFILES)
        $(CC) $^ -o $@

%.o : %.c
        $(CC) $(CFLAGS) -c $^ -o $@

# The @ symbol means while running 'make clean', the command won't be shown
clean:
        @rm -rf $(OBJFILES) $(DEPFILES)

# The @ symbol means while running 'make clean all', the command won't be shown
cleanall: clean
        @rm -rf $(EXEC)

# added test to run the program
test : all
        $(info Running the program ...)
        $(info )
        ./$(EXEC)
```

5 Solving 2.1 - III)

- The problem of finding .c files in the directores can be solved intializing the C directories to look for.
- Then, using foreach loop and wildcards, the .c files can be found using

```
CDIRS += .
CDIRS += ./src

CFILES = $(foreach cdir, $(CDIRS), $(wildcard $(cdir)/*.c))

$(info $(CFILES))
```

- This will produce the output as:

```
./prog.c ./src/mod2.c ./src/mod1.c
```

- The object files to be created for each of the .c files in the same location so we use patsubst to get the .o files.

```
OBJFILES = $(patsubst %.c, %.o, $(CFILES))

$(info $(OBJFILES))
```

- This will produce the output as:

```
./prog.o ./src/mod2.o ./src/mod1.o
```

- Now, the complete Make file will all the Disadvantages solved is given below:

```
# A Simple MakeFile to build C programs


# Include directories to be used
INCDIRS += .
INCDIRS += ./Include

# C directories
CDIRS += .
CDIRS += ./src

# C files to be compiled is found by using wildcard to get the c files
CFILES = $(foreach cdir, $(CDIRS), $(wildcard $(cdir)/*.c))
# o files are obtained by substituting .c with .o using pathsubst in CFILES
OBJFILES = $(patsubst %.c, %.o, $(CFILES))
# d files are obtained by substituting .c with .d using pathsubst in CFILES
DEPFILES = $(patsubst %.c, %.d, $(CFILES))
EXEC = prog


# Compiler to use ; Can be changed for cross compilation
CC =  gcc
# C Flags to be given for compilation - (adding foreach for iterating through each
↪  include directory
CFLAGS = -Wall -Wextra  $(foreach includedir, $(INCDIRS), -I $(includedir)) -MD -MP

# make command will run the first build rule, so generally the first one is all
all : $(EXEC)

# prog is the final executable  to be created which isdepeneded on the object files
↪  to be available
$(EXEC) : $(OBJFILES)
        $(CC) $^ -o $@

%.o : %.c
        $(CC) $(CFLAGS) -c $^ -o $@

# The @ symbol means while running 'make clean', the command won't be shown
clean:
        @rm -rf $(OBJFILES) $(DEPFILES)

# The @ symbol means while running 'make clean all', the command won't be shown
cleanall: clean
        @rm -rf $(EXEC)

# added test to run the program
test : all
        $(info Running the program ...)
```

```
47          $(info )
48          ./$(EXEC)
```