

Make Tool in Unix

Prof. P.C.P. Bhatt



Introduction

- *Make* is both a productivity enhancement utility as well as a program management tool..
- Individual as well as team *productivity enhancement* tool.
- *make* tool avoids wasteful re-compilations
- It is also useful in the context of *software installations*.



When to use make?

- Make can be used in a context where tasks can be expressed as (UNIX) shell commands which need to be executed in the context of certain forms of dependencies amongst file.
- Make also helps to determine what definitions the new software should assume during installation.



How make works?

- *Makefile structure*: the basic structure of makefile is a sequence of targets, dependencies, and commands.
- *Linking with libraries*: helps in creating a customized computing environment



Makefile Structure

```
-----  
| <TARGET> : SET of DEPENDENCIES /* Inline comments */  
| <TAB> command /* Note that command follows a tab */ |  
| <TAB> command /* There may be many command lines */ |  
| . |  
| . |  
| <TAB> command /* There may be many command lines */ |  
-----
```

```
-----  
| <TARGET> : SET of DEPENDENCIES /* Inline comments */  
| <TAB> command /* Note that command follows a tab */ |  
| <TAB> command /* There may be many command lines */ |  
| . |  
| . |  
| <TAB> command /* There may be many command lines */ |  
-----
```




Example - 1

We will demonstrate the use of a make file through this simple example.

Step1:

Create a program file helloWorld.c as shown below:

```
#include<stdio.h>
#include<ctype.h>
main
{
printf("HelloWorld \n");
}
```



Example – 1 continues

Step2: Prepare a file called “Makefile” as follows:

```
# This may be a comment  
hello: helloWorld.c  
cc -o hello helloWorld.c  
# this is all in this make file
```



Example - 1

Step3: Now give a command as follows:

`make`

Step4: Execute helloWorld to see the result.

To see the effect of make, first repeat the command make and note how make responds to indicate that files are up to date needing no re-compilation. Now modify the program.



Example 1 Continues

Let us change the statement in printf to:

`printf("helloWorld here I come ! \n")` Execute make again. One can also choose a name different from Makefile. However, in that case we use `-f` option:

`make -f given_file_name.`

To force make to re-compile a certain file one can simply update its time by a Unix touch command as given below:

`touch <filename> /* updates its modification time */`



Make File Options

Make command has the following other useful options:

-n option : With this option, make goes through all the commands in makefile without executing any of them. Such an option is useful to just check out the makefile itself.

-p option: With this option, make prints all the macros and targets as it progresses.

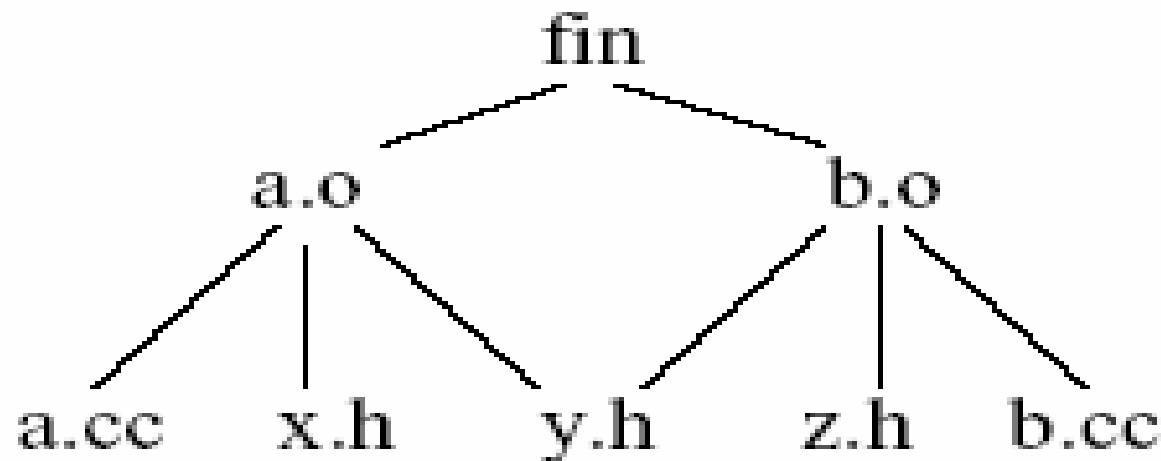


More on Make file options

- s *option*: With this option, make is silent. Essentially, it is the opposite of -p option.
- i *option* : With this option make ignores any errors or exceptions which are thrown up. This is indeed very helpful in a development environment. During development, sometimes one needs to focus on a certain part of a program. One may have to do this with several modules in place. In these situations, one often wishes to ignore some obvious exceptions.

Example - 2

This time around we shall consider a two-level target definition. Suppose our executable “fin” depends upon two object files (a.o) and (b.o) as shown in below figure. We may further have (a.o) depend upon (a.cc) and (x.h) and (y.h). Similarly, (b.o) may depend upon (b.cc, z.h) and (y.h).





Example - 2

These dependencies shown in the previous figure dictate the following:

- A change in x.h will result in re-compiling to get a new a.o and fin. A change in y.h will result in re-compiling to get a new a.o, b.o, and fin. A change in z.h will result in re-compiling to get a new b.o and fin.
- A change in a.cc will result in re-compiling to get a new a.o and fin.
- A change in b.cc will result in re-compiling to get a new b.o and fin.



Makefile for Example 2

```
fin : a.o b.o /* the top level of target */
g++ -o fin a.o b.o /* the action required */
a.o : a.c x.h y.h /* the next level of targets */
g++ -g -Wall -c a.cc -o a.o /* hard coded comd. line */
b.o : b.c z.h y.h /* the next level of targets */
g++ -g -Wall -c b.cc -o b.o
x.h : /* empty dependencies */
y.h :
z.h : /* these rules are not needed */
```



Linking with Libraries

Makefile

```
#
# fin depends on a.o and a library libthelib.a
#
fin : a.o libthelib.a
g++ -o fin a.o -L -lthelib
#
# a.o depends on three files a.c, x.h and y.h
# The Wall option is a very thorough checking option
# available under g++
a.o : a.cc x.h y.h
g++ -g -Wall -c a.cc
#
# Now the empty action lines.
#
x.h :
y.h :
thelib.h :
#
# Now the rules to rebuild the library libthelib.a
#
libthelib.a thelib.o
ar rv libthelib.a thelib.o
thelib.o: thelib.cc thelib.h
g++ -g -Wall thelib.cc
# end of make file
```

Commands to run the makefile

If we run this make file we should expect to see the following sequence of actions

1. g++ -c a.cc
2. cc -c thelib.cc
3. ar rv libthelib.a thelib.o
4. a - thelib.o
5. g++ -o fin a.o -L -lthelib



Macros - 1

Variables in make are defined and interpreted exactly as in shell scripts. For instance we could define a variable as follows:

```
CC = g++ /* this is a variable definition */  
# the definition extends up to new line character or  
# up the beginning of the inline comment
```

In fact almost all environments support CC macro which expands to appropriate compiler command, i.e., it expands to cc in Unix, cl in MS and bcc for Borland. A typical usage in a command is shown below:

```
$(CC) -o p p.c /* here p refers to an executable */
```




Macros - 2

Let us look at flags first. Suppose we have a set of flags for c compilation. We may define a new macro as shown below :

CFLAGS = -o -L -lthelib

These are now captured as follows:

\$(CC) \$(CFLAGS) p.c

Another typical usage is when we have targets that require many objects as in the example below:

t : p1.o p2.o p3.o / here t is the target and pi the program stems */*

We can now define macros as follows:

TARGET = t

OBJS = p1.o p2.o p3.o

\$(TARGET): \$(OBJS)



General Substitution Rules

So if we have a situation as follows:

target : several objects

cc cflags target target_stem.c

This can be encoded as follows :

`$(TARGET): $(OBJS)`

`$(CC) $(CFLAGS) @$ $*.c`



Inference Rules in Make

An inference rule begins with a (.) (period) symbol and establishes the relationship. So a .c.o means we need an a .c file for generating the a .o file and it may appear as follows:

.c.o :

```
$(CC) $(CFLAGS) $*.c
```

In fact because the name stems of .c and o. file is to be the same we can use a built-in macro (\$<) to code the above set of lines as follows:

.c.o:

```
$(CC) $(CFLAGS) $<
```



Some Additional Options

- **Imake** : program to generate platform specific make files.
- **Imake.tmpl** : set of general rules and configuration properly, it yields a suitable makefile for suitable platforms.