Linux Makefiles (Basics)

To understand the concept of makefiles, one should have basic knowledge of the compilation process.

Do you know how to compile a code using gcc? For those who doesn't know, here is a quick description :

Here is a code of most simple 'C' program, hello1.c:

Code:

```
#include<stdio.h>

int main(void)
{
    printf("\n Hello World\n");
    return 0;
}
```

Now, to compile the above program, one needs 'gcc' compiler on Linux(which by default comes on most of Linux distros).
To compile the above code, issue the following command :

gcc -Wall hello1.c -o hello

The above line asks the gcc compiler to compile hello1.c, turn on all types of warnings(-Wall) and name the output executable as 'hello'. This is the most basic way in which one can compile the code.

Now suppose you have a project with multiple files, say hello1.c hello2.c, hello3.c, hello4.c. to compile multiple files like this one needs following command :

gcc -Wall hello1.c hello2.c hello3.c hello4.c -o hello

Seems doable task. Now suppose for some reason lets say, you want to compile hello2.c without warnings enabled while all the other source files with warning enabled. Then one needs to compile hello2.c first and then all the other files. So we see that with growing need, manual compilation can become and exhaustive and tedious task. So one feels the requirement of a tool through which one can manipulate the compilation needs easily.

Linux provides 'make' command for this purpose. The make command once issued searches for a file named 'Makefile' in the current directory and executes the commands in it.

if name of your Makefile is other than 'Makefile', then make utility accepts the name of your makefile (lets say 'my_makefile') by using flag -f.

make -f my_makefile

Understanding Makefiles

The basic makefile is composed of :

```
target: dependencies
[tab] system command
```

for example :

```
all:
gcc -Wall hello1.c -o hello
```

Now, to run the above makefile (assuming that its name is 'Makefile'), just type the 'make' command. The target 'all' is a standard default for Makefiles and thats the reason why we do not specify this target as this is the default target that make utility executes if we do not specify any other target to it.

For example lets suppose we do not specify 'all', instead we specify 'complete' :

```
complete:
gcc -Wall hello1.c -o hello
```

In the above case, we need to specify run 'make complete' as 'complete' is not a default like 'all' and hence needs to be mentioned with 'make' command.

Now, coming back to the basic structure of Makefiles, there was a category known as 'dependencies' [refer above]. Lets understand it using the following example :

```
all: hello1.o hello2.o hello3.o hello4.o
gcc -Wall hello1.o hello2.o hello3.o hello4.o -o hello
```

In the above example, we see that the target 'all' this time has some dependencies like hello1.o hello2.o etc. These dependencies tell make utility that the command under the target 'all' is dependent on these object files and hence these object files should be created before executing the command under 'all' target. So now make utility thinks of all these dependencies as targets and tries to find these targets in the makefile and execute the command under the respective targets.The complete makefile looks like :

```
all: hello

hello: hello1.o hello2.o hello3.o hello4.o
gcc -Wall hello1.o hello2.o hello3.o hello4.o -o hello

hello1.o: hello1.c
gcc -Wall -c hello1.c

hello2.o: hello2.c
gcc -Wall -c hello2.c

hello3.o: hello3.c
```

```
gcc -Wall -c hello3.c

hello4.o: hello4.c
gcc -Wall -c hello4.c

clean :
rm -rf *.o hello
```

In this example we see a target called clean. It is useful to have such target if you want to have a fast way to get rid of all the object files and executables.

One can use comments using '#' for each line fo comment.

```
# I am comment number 1
# I am comment number 2
all: hello

hello: hello1.o hello2.o hello3.o hello4.o
gcc -Wall hello1.o hello2.o hello3.o hello4.o -o hello

hello1.o: hello1.c
gcc -Wall -c hello1.c

hello2.o: hello2.c
gcc -Wall -c hello2.c

hello3.o: hello3.c
gcc -Wall -c hello3.c

hello4.o: hello4.c
gcc -Wall -c hello4.c

clean :
rm -rf *.o hello
```

Using Variables

One can use variables in Makefiles to make the maintainance of makefiles easy. Lets see the follwoing example :

```
# I am comment number 1
# I am comment number 2
CC=gcc
WLEVEL=-Wall
all: hello
hello: hello1.o hello2.o hello3.o hello4.o
```

```
$(CC) $(WLEVEL) hello1.o hello2.o hello3.o hello4.o -o hello

hello1.o:
$(CC) $(WLEVEL) -c hello1.c

hello2.o:
$(CC) $(WLEVEL) -c hello2.c

hello3.o:
$(CC) $(WLEVEL) -c hello3.c

hello4.o:
$(CC) $(WLEVEL) -c hello4.c

clean :
rm -rf *.o hello
```

We see in the above example, we used two varibales 'CC' and 'WLEVEL'. 'CC' represent the compiler we are using while 'WLEVEL' represents the warning level.
Hence we see that variables can be extremely useful as change in one variable can infuse the desired change in complete makefile.

Conclusion

To conclude, we saw in this tutorial that what powerful tool can the 'make' utility become if we use the concept of Makefiles. Makefiles are used in most of the professional projects on Linux/Unix and are a standard way of managing the compiling process.

Source: http://www.go4expert.com/articles/linux-makefiles-basics-t26805/