Build Automation using makefile and make

Objective

- To create and use a makefile to compile and generate an executable
- To substitute macros for directory, compiler options etc in makefile

Benefits of makefile

- Automate the build process
- Recompile only updated dependency sources (without recompiling all) and generate an executable

How to use makefile?

- Create a makefile
- Compile makefile using make and generate executable

How does make work?

- The make program uses the makefile data base and the last-modification times of the files to decide which of the files need to be updated. Hence only the changed source files are recompiled.

How to create makefile?

A simple makefile consists of "rules" with the following shape:
target ... : prerequisites ...
command
...
...

Writing a Makefile from scratch

Prerequisite:

1.Create the following file with contents as below.

a. module.c – with one function say sample_func() to display "hello "

b. module.h – prototype of function in module.c

c. myprogram.c – include module.h and invoke sample_func()

By convention, all variable names used in a Makefile are in upper-case. A common variable assignment in a Makefile is CC = gcc, which can then be used later on as ${CC} or $(CC). Makefiles use # as the comment-start marker, just like in shell scripts.

The general syntax of a Makefile rule is as follows:

```
target: dependency1 dependency2 ...
[TAB] action1
[TAB] action2
```

```
...
```

Let's take a look at a simple Makefile for our sample project to generate myprogram executable

```
all: myprogram.o module.o
    gcc main.o module.o -o myprogram
myprogram.o: myprogram.c module.h
    gcc -I./ -c myprogram.c
module.o: module.c module.h
    gcc -I./ -c module.c
clean:
    rm -f *.o
    rm myprogram
```

We have four targets in the Makefile:

- all is a special target that depends on myprogram.o and module.o, and has the command (from the "manual" steps earlier) to make GCC link the two object files into the final executable binary.
- main.o is a filename target that depends on myprogram.c and module.h, and has the command to compile myprogram.c to produce myprogram.o.
- module.o is a filename target that depends on module.c and module.h; it calls GCC to compile the module.c file to produce module.o.
- clean is a special target that has no dependencies, but specifies the commands to clean the compilation outputs from the project directories.

Running make:

In directory containing makefile, execute command below.

$ make

Will see output as below.

```
gcc -I . -c myprogram.c
gcc -I . -c module.c
gcc myprogram.o module.o -o myprogram
```

What has happened here?

When we ran make without specifying a target on the command line, it defaulted to the first target in our Makefile — that is, the target all.

If we immediately run make again, without changing any of the source files, we will see as below.

```
$make

Nothing to be done for `all'.
```

Now, we update module.c by adding a statement printf("\nfirst update"); inside the sample_func() function. We then run make again:

```
$ make
gcc -I . -c module.c
gcc myprogram.o module.o -o myprogram
```

We can explicitly invoke the clean target to clean up all the generated .o files and myprogram:

```
$ make clean
rm -f *.o
rm myprogram
```

How to run a makefile with a different name?

```
$ make -f <name of makefile>
```

Using Macros

You can define macros and use them for repeated words, like in above make file, for gcc we can define a macro CC as below .

```
CC=gcc
```

To substitute for gcc use $(CC). Similarly we can define and use macros for directory locations and for compiler/linker options as below.

```
#-I for include directory path
INC=../inc
OBJ=../obj
SRC=../src
BIN=../bin
CFLAGS=-c -g -Wall
IFLAGS=-I$(INC)

#update flag below to include  library if any

LFLAGS=-l
```

Create a project directory with sub-directories as below in your home directory and place the files (i.e *.c in prj/src and *.h in prj/inc).

```
prj/src

prj/obj

prj/bin
```

```
prj/inc

prj/make
```

Create and place a makefile in prj/make directory. With above macro substitution, the updated makefile is

```
# simple make file
CC=gcc


#-I for include directory path
INC=../inc
OBJ=../obj
SRC=../src
BIN=../bin
CFLAGS=-c -g -Wall
IFLAGS=-I$(INC)

#update flag below to include library if any
LFLAGS=-l


#all target generates the executable
all: $(OBJ)/myprogram.o $(OBJ)/module.o
	$(CC) $(LFLAGS) $(OBJ)/myprogram.o $(OBJ)/module.o -o $(BIN)/myprogram


$(OBJ)/myprogram.o: $(SRC)/myprogram.c $(INC)/module.h
	$(CC) $(IFLAGS) $(CFLAGS) $(SRC)/myprogram.c
$(OBJ)module.o:  $(SRC)/module.c  $(INC)/module.h
	$(CC) $(IFLAGS) $(CFLAGS) $(SRC)/module.c


#clean target removes the *.o and executable
clean:
	rm -f $(OBJ)/*.o
	rm $(BIN)/myprogram
```