MAKEFILE - OTHER FEATURES

Recursive Use of Make

Recursive use of `make` means using `make` as a command in a makefile. This technique is useful when you want separate makefiles for various subsystems that compose a larger system. For example, suppose you have a subdirectory named `subdir' which has its own makefile, and you would like the containing directory's makefile to run `make` on the subdirectory. You can do it by writing this −

```
subsystem:
        cd subdir && $(MAKE)

or, equivalently –

subsystem:
        $(MAKE) -C subdir
```

You can write recursive `make` commands just by copying this example, but you need to know about how they work and why, and about how the sub-make relates to the top-level make.

Communicating Variables to a Sub-Make

Variable values of the top-level `make` can be passed to the sub-make through the environment by explicit request. These variables are defined in the sub-make as defaults. You can not override what is specified in the makefile used by the sub-make makefile unless you use the `-e' switch.

To pass down, or export, a variable, `make` adds the variable and its value to the environment for running each command. The sub-make, in turn, uses the environment to initialize its table of variable values.

The special variables SHELL and MAKEFLAGS are always exported *unlessyouunexportthem*. MAKEFILES is exported if you set it to anything.

If you want to export specific variables to a sub-make, use the export directive, as shown −

```
export variable ...
```

If you want to prevent a variable from being exported, use the unexport directive, as shown −

```
unexport variable ...
```

The Variable MAKEFILES

If the environment variable MAKEFILES is defined, `make` considers its value as a list of names *separatedbywhitespace* of additional makefiles to be read before the others. This works much like the include directive: various directories are searched for those files.

The main use of MAKEFILES is in communication between recursive invocations of the `make`.

Including Header file from Different Directories

If you have put the header files in different directories and you are running `make` in a different directory, then it is required to provide the path of header files. This can be done using -I option in makefile. Assuming that functions.h file is available in /home/tutorialspoint/header folder and rest of the files are available in /home/tutorialspoint/src/ folder, then the make file would be written as follows.

```
INCLUDES = -I "/home/tutorialspoint/header"
CC = gcc
```

```
LIBS =  -lm
CFLAGS = -g -Wall
OBJ =  main.o factorial.o hello.o

hello: ${OBJ}
   ${CC} ${CFLAGS} ${INCLUDES} -o $@ ${OBJS} ${LIBS}
.cpp.o:
   ${CC} ${CFLAGS} ${INCLUDES} -c $<
```

Appending More Text to Variables

Often it is useful to add more text to the value of a variable already defined. You do this with a line containing `+=', as shown:

```
objects += another.o
```

It takes the value of the variable objects, and adds the text `another.o' to it *precededbyasinglespace*. Thus −

```
objects = main.o hello.o factorial.o
objects += another.o
```

sets objects to `main.o hello.o factorial.o another.o'.

Using `+=' is similar to:

```
objects = main.o hello.o factorial.o
objects := $(objects) another.o
```

Continuation Line in Makefile

If you do not like too big lines in your Makefile, then you can break your line using a back-slash "\" as shown below −

```
OBJ =  main.o factorial.o \
 hello.o

is equivalent to

OBJ =  main.o factorial.o hello.o
```

Running Makefile from Command Prompt

If you have prepared the Makefile with name "Makefile", then simply write make at command prompt and it will run the Makefile file. But if you have given any other name to the Makefile, then use the following command −

```
make -f your-makefile-name
```