Defining Rules in Makefile

We will now learn the rules for Makefile.

The general syntax of a Makefile target rule is −

```
target [target...] : [dependent ....]
[ command ...]
```

In the above code, the arguments in brackets are optional and ellipsis means one or more. Here, note that the tab to preface each command is required.

A simple example is given below where you define a rule to make your target hello from three other files.

```
hello: main.o factorial.o hello.o
   $(CC) main.o factorial.o hello.o -o hello
```

NOTE − In this example, you would have to give rules to make all object files from the source files.

The semantics is very simple. When you say "make target", the **make** finds the target rule that applies; and, if any of the dependents are newer than the target, **make** executes the commands one at a time (after macro substitution). If any dependents have to be made, that happens first (so you have a recursion).

Make terminates if any command returns a failure status. The following rule will be shown in such case −

```
clean:
   -rm *.o *~ core paper
```

Make ignores the returned status on command lines that begin with a dash. For example, who cares if there is no core file?

Make echoes the commands, after macro substitution to show you what is happening. Sometimes you might want to turn that off. For example −

```
install:
   @echo You must be root to install
```

People have come to expect certain targets in Makefiles. You should always browse first. However, it is reasonable to expect that the targets all (or just make), install, and clean is found.

- **make all** − It compiles everything so that you can do local testing before installing applications.
- **make install** − It installs applications at right places.
- **make clean** − It cleans applications, gets rid of the executables, any temporary files, object files, etc.

Makefile Implicit Rules

The command is one that ought to work in all cases where we build an executable x out of the source code x.cpp. This can be stated as an implicit rule −

```
.cpp:
   $(CC) $(CFLAGS) $@.cpp $(LDFLAGS) -o $@
```

This implicit rule says how to make x out of x.c -- run cc on x.c and call the output x. The rule is implicit because no particular target is mentioned. It can be used in all cases.

Another common implicit rule is for the construction of .o (object) files out of .cpp (source files).

```
.cpp.o:
   $(CC) $(CFLAGS) -c $<

alternatively

.cpp.o:
   $(CC) $(CFLAGS) -c $*.cpp
```