DEFINING RULES IN MAKEFILE

The general syntax of a Makefile target rule is −

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

Arguments in brackets are optional, ellipsis means one or more. Note 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 pretty 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, the `make` executes the commands one at a time *aftermacrosubstitution*. If any dependents have to be made, that happens first *soyouhavearecursion*.

A `make` will terminate if any command returns a failure status. That's why you see rules like −

```
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 as it happens. 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, is reasonable to expect that the targets all *orjustmake*, 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. But watch out that things are installed in the right place for your system.
- **make clean** − It clean applications up, 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 *sourcefiles*.

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

alternatively

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