Variables

Makefiles allow you to use variables, which are reminiscent of shell variables. A variable declaration can occur anywhere in a makefile, and is of the following form:

```
name = value
```

The value assigned to a variable extends to the end of the line, and may be a sequence of words. For example, the following variable declaration occurs often in makefiles used to compile C code:

```
objects = file1.o file2.o file3.o
```

To use a variable, you prepend *$* to the variable name in parentheses. The variable gets expanded by *make* prior to interpreting the makefile. For instance, if we assume the variable *objects* as defined above, we can imagine the rule:

```
myapp : $(objects)
    gcc -o myapp $(objects)
```

Other variables often occuring in makefiles include variables to hold program options, if they need to be changed as a whole. For example, compilation flags are often put in a variable, so that it is easy to recompile an application with debugging information by simply adding the appropriate command-line options to the variable instead of changing all the occurences of the compiler. Another typical use of variables is to abstract away from the name of commands to use. For example, a compilation makefile may contain rules such as:

```
file.o : file.c
    $(GCC) $(CFLAGS) -c file.c
```

where the variable *GCC* contains the name of the compiler to use, *gcc*, or *cc*, and *CFLAGS* contains the appropriate flags for that compiler.

A very interesting aspect of *make* is that it automatically defines a variable for every environment variable that exists. Hence, a makefile can refer to environment variables. Of course, you can override environment variables by simply declaring a makefile variable with the same name.

Phony targets

I haven't talked a lot about targets. In my discussions, I have assumed that there was an implicit *final target* in the makefile, for example corresponding to the compiled application in a compilation makefile. Time to be a bit more precise. When invoking *make*, you can specify what target to build:

`make` *`target`*

This will rebuild *target* if it needs to be updated according to the rules of the makefile. By default, if no target is specified, the target of the first rule of the makefile is assumed (which is what was happening until now).

Being able to specify a target when calling *make* enables the use of *phony targets* to do special maintenance jobs. For example, a phony target that often occurs in compilation makefiles is a target that cleans up the directory by removing the intermediary object files.

```
clean :
    rm -f $(objects)
```

(We use the variable *objects* to hold the names of the object files.) If a file named *clean* does not occur in the directory, then doing a *make clean* will automatically trigger this rule, since a rule is always triggered when its target does not exist. This will have the effect of removing all object files from the directory.

(To enforce to *make* that the target *clean* is not meant to correspond to a file in the directory, you can use a special declaration. The declaration

`.PHONY :` *`target target`* `...`

declares that the given targets are phony targets not corresponding to any files. This helps *make* not get confused if files of the same name occur in the directory.)

A trick: if a phony target depends on other phony targets, those targets are invoked as "subroutines" of the target, that is, they are triggered themselves before the commands of the target are executed. In general, it is a bad idea to have a real target depend on a phony target, as it will force the triggering of the rule corresponding to the real target.