2 An Introduction to Makefiles

You need a file called a *makefile* to tell `make` what to do. Most often, the makefile tells `make` how to compile and link a program.

In this chapter, we will discuss a simple makefile that describes how to compile and link a text editor which consists of eight C source files and three header files. The makefile can also tell `make` how to run miscellaneous commands when explicitly asked (for example, to remove certain files as a clean-up operation). To see a more complex example of a makefile, see Appendix B [Complex Makefile], page 149.

When `make` recompiles the editor, each changed C source file must be recompiled. If a header file has changed, each C source file that includes the header file must be recompiled to be safe. Each compilation produces an object file corresponding to the source file. Finally, if any source file has been recompiled, all the object files, whether newly made or saved from previous compilations, must be linked together to produce the new executable editor.

2.1 What a Rule Looks Like

A simple makefile consists of "rules" with the following shape:

```
target ... : dependencies ...
        command
        ...
        ...
```

A *target* is usually the name of a file that is generated by a program; examples of targets are executable or object files. A target can also be the name of an action to carry out, such as '`clean`' (see Section 4.4 [Phony Targets], page 28).

A *dependency* is a file that is used as input to create the target. A target often depends on several files.

A *command* is an action that `make` carries out. A rule may have more than one command, each on its own line. **Please note:** you need to put a tab character at the beginning of every command line! This is an obscurity that catches the unwary.

Usually a command is in a rule with dependencies and serves to create a target file if any of the dependencies change. However, the rule that specifies commands for the target need not have dependencies. For example, the rule containing the delete command associated with the target '`clean`' does not have dependencies.

A *rule*, then, explains how and when to remake certain files which are the targets of the particular rule. `make` carries out the commands on the dependencies to create or update the target. A rule can also explain how and when to carry out an action. See Chapter 4 [Writing Rules], page 19.

A makefile may contain other text besides rules, but a simple makefile need only contain rules. Rules may look somewhat more complicated than shown in this template, but all fit the pattern more or less.

2.2 A Simple Makefile

Here is a straightforward makefile that describes the way an executable file called `edit` depends on eight object files which, in turn, depend on eight C source and three header files.

In this example, all the C files include '`defs.h`', but only those defining editing commands include '`command.h`', and only low level files that change the editor buffer include '`buffer.h`'.

```
edit : main.o kbd.o command.o display.o \
       insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit main.o kbd.o command.o display.o \
           insert.o search.o files.o utils.o
```

We split each long line into two lines using backslash-newline; this is like using one long line, but is easier to read.

To use this makefile to create the executable file called '`edit`', type:

```
make
```

To use this makefile to delete the executable file and all the object files from the directory, type:

```
make clean
```

In the example makefile, the targets include the executable file '`edit`', and the object files '`main.o`' and '`kbd.o`'. The dependencies are files such as '`main.c`' and '`defs.h`'. In fact, each '`.o`' file is both a target and a dependency. Commands include '`cc -c main.c`' and '`cc -c kbd.c`'.

When a target is a file, it needs to be recompiled or relinked if any of its dependencies change. In addition, any dependencies that are themselves automatically generated should be updated first.

In this example, '`edit`' depends on each of the eight object files; the object file '`main.o`' depends on the source file '`main.c`' and on the header file '`defs.h`'.

A shell command follows each line that contains a target and dependencies. These shell commands say how to update the target file. A tab character must come at the beginning of every command line to distinguish commands lines from other lines in the makefile. (Bear in mind that `make` does not know anything about how the commands work. It is up to you to supply commands that will update the target file properly. All `make` does is execute the commands in the rule you have specified when the target file needs to be updated.)

The target '`clean`' is not a file, but merely the name of an action. Since you normally do not want to carry out the actions in this rule, '`clean`' is not a dependency of any other rule. Consequently, `make` never does anything with it unless you tell it specifically. Note that this rule not only is not a dependency, it also does not have any dependencies, so the only purpose of the rule is to run the specified commands. Targets that do not refer to files but are just actions are called *phony targets*. See Section 4.4 [Phony Targets], page 28, for information about this kind of target. See Section 5.4 [Errors in Commands], page 44, to see how to cause `make` to ignore errors from `rm` or any other command.

2.3 How `make` Processes a Makefile

By default, `make` starts with the first rule (not counting rules whose target names start with '`.`'). This is called the *default goal*. (*Goals* are the targets that `make` strives ultimately to update. See Section 9.2 [Arguments to Specify the Goals], page 90.)

In the simple example of the previous section, the default goal is to update the executable program '`edit`'; therefore, we put that rule first.

Thus, when you give the command:

```
make
```

`make` reads the makefile in the current directory and begins by processing the first rule. In the example, this rule is for relinking '`edit`'; but before `make` can fully process this rule, it must process the rules for the files that '`edit`' depends on, which in this case are the object files. Each of these files is processed according to its own rule. These rules say to update each '`.o`' file by compiling its source file. The recompilation must be done if the source file, or any of the header files named as dependencies, is more recent than the object file, or if the object file does not exist.

The other rules are processed because their targets appear as dependencies of the goal. If some other rule is not depended on by the goal (or anything it depends on, etc.), that rule is not processed, unless you tell `make` to do so (with a command such as `make clean`).

Before recompiling an object file, `make` considers updating its dependencies, the source file and header files. This makefile does not specify anything to be done for them—the '`.c`' and '`.h`' files are not the targets of any rules—so `make` does nothing for these files. But `make` would update automatically generated C programs, such as those made by Bison or Yacc, by their own rules at this time.

After recompiling whichever object files need it, `make` decides whether to relink '`edit`'. This must be done if the file '`edit`' does not exist, or if any of the object files are newer than it. If an object file was just recompiled, it is now newer than '`edit`', so '`edit`' is relinked.

Thus, if we change the file '`insert.c`' and run `make`, `make` will compile that file to update '`insert.o`', and then link '`edit`'. If we change the file '`command.h`' and run `make`, `make` will recompile the object files '`kbd.o`', '`command.o`' and '`files.o`' and then link the file '`edit`'.

2.4 Variables Make Makefiles Simpler

In our example, we had to list all the object files twice in the rule for '`edit`' (repeated here):

```
edit : main.o kbd.o command.o display.o \
               insert.o search.o files.o utils.o
        cc -o edit main.o kbd.o command.o display.o \
                    insert.o search.o files.o utils.o
```

Such duplication is error-prone; if a new object file is added to the system, we might add it to one list and forget the other. We can eliminate the risk and simplify the makefile by using a variable. *Variables* allow a text string to be defined once and substituted in multiple places later (see Chapter 6 [How to Use Variables], page 55).

It is standard practice for every makefile to have a variable named `objects`, `OBJECTS`, `objs`, `OBJS`, `obj`, or `OBJ` which is a list of all object file names. We would define such a variable `objects` with a line like this in the makefile:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o
```

Then, each place we want to put a list of the object file names, we can substitute the variable's value by writing '`$(objects)`' (see Chapter 6 [How to Use Variables], page 55).

Here is how the complete simple makefile looks when you use a variable for the object files:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)
main.o : main.c defs.h
        cc -c main.c
kbd.o : kbd.c defs.h command.h
        cc -c kbd.c
command.o : command.c defs.h command.h
        cc -c command.c
display.o : display.c defs.h buffer.h
        cc -c display.c
insert.o : insert.c defs.h buffer.h
        cc -c insert.c
search.o : search.c defs.h buffer.h
        cc -c search.c
files.o : files.c defs.h buffer.h command.h
        cc -c files.c
utils.o : utils.c defs.h
        cc -c utils.c
clean :
        rm edit $(objects)
```

2.5 Letting `make` Deduce the Commands

It is not necessary to spell out the commands for compiling the individual C source files, because `make` can figure them out: it has an *implicit rule* for updating a '`.o`' file from a correspondingly named '`.c`' file using a '`cc -c`' command. For example, it will use the command '`cc -c main.c -o main.o`' to compile '`main.c`' into '`main.o`'. We can therefore omit the commands from the rules for the object files. See Chapter 10 [Using Implicit Rules], page 101.

When a '`.c`' file is used automatically in this way, it is also automatically added to the list of dependencies. We can therefore omit the '`.c`' files from the dependencies, provided we omit the commands.

Here is the entire example, with both of these changes, and a variable `objects` as suggested above:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

main.o : defs.h
kbd.o : defs.h command.h
command.o : defs.h command.h
display.o : defs.h buffer.h
insert.o : defs.h buffer.h
search.o : defs.h buffer.h
files.o : defs.h buffer.h command.h
utils.o : defs.h

.PHONY : clean
clean :
        -rm edit $(objects)
```

This is how we would write the makefile in actual practice. (The complications associated with '`clean`' are described elsewhere. See Section 4.4 [Phony Targets], page 28, and Section 5.4 [Errors in Commands], page 44.)

Because implicit rules are so convenient, they are important. You will see them used frequently.

2.6 Another Style of Makefile

When the objects of a makefile are created only by implicit rules, an alternative style of makefile is possible. In this style of makefile, you group entries by their dependencies instead of by their targets. Here is what one looks like:

```
objects = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(objects)
        cc -o edit $(objects)

$(objects) : defs.h
kbd.o command.o files.o : command.h
display.o insert.o search.o files.o : buffer.h
```

Here 'defs.h' is given as a dependency of all the object files; 'command.h' and 'buffer.h' are dependencies of the specific object files listed for them.

Whether this is better is a matter of taste: it is more compact, but some people dislike it because they find it clearer to put all the information about each target in one place.

2.7 Rules for Cleaning the Directory

Compiling a program is not the only thing you might want to write rules for. Makefiles commonly tell how to do a few other things besides compiling a program: for example, how to delete all the object files and executables so that the directory is 'clean'.

Here is how we could write a make rule for cleaning our example editor:

```
clean:
        rm edit $(objects)
```

In practice, we might want to write the rule in a somewhat more complicated manner to handle unanticipated situations. We would do this:

```
.PHONY : clean
clean :
        -rm edit $(objects)
```

This prevents make from getting confused by an actual file called 'clean' and causes it to continue in spite of errors from rm. (See Section 4.4 [Phony Targets], page 28, and Section 5.4 [Errors in Commands], page 44.)

A rule such as this should not be placed at the beginning of the makefile, because we do not want it to run by default! Thus, in the example makefile, we want the rule for edit, which recompiles the editor, to remain the default goal.

Since clean is not a dependency of edit, this rule will not run at all if we give the command 'make' with no arguments. In order to make the rule run, we have to type 'make clean'. See Chapter 9 [How to Run make], page 89.