CHAPTER 1

How to Write a Simple Makefile

The mechanics of programming usually follow a fairly simple routine of editing source files, compiling the source into an executable form, and debugging the result. Although transforming the source into an executable is considered routine, if done incorrectly a programmer can waste immense amounts of time tracking down the problem. Most developers have experienced the frustration of modifying a function and running the new code only to find that their change did not fix the bug. Later they discover that they were never executing their modified function because of some procedural error such as failing to recompile the source, relink the executable, or rebuild a jar. Moreover, as the program's complexity grows these mundane tasks can become increasingly error-prone as different versions of the program are developed, perhaps for other platforms or other versions of support libraries, etc.

The `make` program is intended to automate the mundane aspects of transforming source code into an executable. The advantages of `make` over scripts is that you can specify the relationships between the elements of your program to `make`, and it knows through these relationships and timestamps exactly what steps need to be redone to produce the desired program each time. Using this information, `make` can also optimize the build process avoiding unnecessary steps.

GNU `make` (and other variants of `make`) do precisely this. `make` defines a language for describing the relationships between source code, intermediate files, and executables. It also provides features to manage alternate configurations, implement reusable libraries of specifications, and parameterize processes with user-defined macros. In short, `make` can be considered the center of the development process by providing a roadmap of an application's components and how they fit together.

The specification that `make` uses is generally saved in a file named *makefile*. Here is a *makefile* to build the traditional "Hello, World" program:

```
hello: hello.c
        gcc hello.c -o hello
```

To build the program execute `make` by typing:

```
$ make
```

at the command prompt of your favorite shell. This will cause the make program to read the *makefile* and build the first target it finds there:

```
$ make
gcc hello.c -o hello
```

If a target is included as a command-line argument, that target is updated. If no command-line targets are given, then the first target in the file is used, called the *default goal*.

Typically the default goal in most *makefile*s is to build a program. This usually involves many steps. Often the source code for the program is incomplete and the source must be generated using utilities such as `flex` or `bison`. Next the source is compiled into binary object files (*.o* files for C/C++, *.class* files for Java, etc.). Then, for C/C++, the object files are bound together by a linker (usually invoked through the compiler, `gcc`) to form an executable program.

Modifying any of the source files and reinvoking `make` will cause some, but usually not all, of these commands to be repeated so the source code changes are properly incorporated into the executable. The specification file, or *makefile*, describes the relationship between the source, intermediate, and executable program files so that `make` can perform the minimum amount of work necessary to update the executable.

So the principle value of `make` comes from its ability to perform the complex series of commands necessary to build an application and to optimize these operations when possible to reduce the time taken by the edit-compile-debug cycle. Furthermore, `make` is flexible enough to be used anywhere one kind of file depends on another from traditional programming in C/C++ to Java, TEX, database management, and more.

Targets and Prerequisites

Essentially a *makefile* contains a set of rules used to build an application. The first rule seen by `make` is used as the *default rule*. A *rule* consists of three parts: the target, its prerequisites, and the command(s) to perform:

```
target: prereq1 prereq2
        commands
```

The *target* is the file or thing that must be made. The *prerequisites* or *dependents* are those files that must exist before the target can be successfully created. And the *commands* are those shell commands that will create the target from the prerequisites.

Here is a rule for compiling a C file, *foo.c*, into an object file, *foo.o*:

```
foo.o: foo.c foo.h
        gcc -c foo.c
```

The target file *foo.o* appears before the colon. The prerequisites *foo.c* and *foo.h* appear after the colon. The command script usually appears on the following lines and is preceded by a tab character.

When make is asked to evaluate a rule, it begins by finding the files indicated by the prerequisites and target. If any of the prerequisites has an associated rule, make attempts to update those first. Next, the target file is considered. If any prerequisite is newer than the target, the target is remade by executing the commands. Each command line is passed to the shell and is executed in its own subshell. If any of the commands generates an error, the building of the target is terminated and make exits. One file is considered newer than another if it has been modified more recently.

Here is a program to count the number of occurrences of the words "fee," "fie," "foe," and "fum" in its input. It uses a flex scanner driven by a simple main:

```
#include <stdio.h>

extern int fee_count, fie_count, foe_count, fum_count;
extern int yylex( void );

int main( int argc, char ** argv )
{
    yylex();
    printf( "%d %d %d %d\n", fee_count, fie_count, foe_count, fum_count );
    exit( 0 );
}
```

The scanner is very simple:

```
        int fee_count = 0;
        int fie_count = 0;
        int foe_count = 0;
        int fum_count = 0;
%%
fee     fee_count++;
fie     fie_count++;
foe     foe_count++;
fum     fum_count++;
```

The *makefile* for this program is also quite simple:

```
count_words: count_words.o lexer.o -lfl
        gcc count_words.o lexer.o -lfl -ocount_words

count_words.o: count_words.c
        gcc -c count_words.c

lexer.o: lexer.c
        gcc -c lexer.c

lexer.c: lexer.l
        flex -t lexer.l > lexer.c
```

When this *makefile* is executed for the first time, we see:

```
$ make
gcc -c count_words.c
flex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -lfl -ocount_words
```

We now have an executable program. Of course, real programs typically consist of more modules than this. Also, as you will see later, this *makefile* does not use most of the features of make so it's more verbose than necessary. Nevertheless, this is a functional and useful *makefile*. For instance, during the writing of this example, I executed the *makefile* several dozen times while experimenting with the program.

As you look at the *makefile* and sample execution, you may notice that the order in which commands are executed by make are nearly the opposite to the order they occur in the *makefile*. This *top-down* style is common in *makefile*s. Usually the most general form of target is specified first in the *makefile* and the details are left for later. The make program supports this style in many ways. Chief among these is make's two-phase execution model and recursive variables. We will discuss these in great detail in later chapters.

Dependency Checking

How did make decide what to do? Let's go over the previous execution in more detail to find out.

First make notices that the command line contains no targets so it decides to make the default goal, *count_words*. It checks for prerequisites and sees three: *count_words.o*, *lexer.o*, and -lfl. make now considers how to build *count_words.o* and sees a rule for it. Again, it checks the prerequisites, notices that *count_words.c* has no rules but that the file exists, so make executes the commands to transform *count_words.c* into *count_words.o* by executing the command:

```
gcc -c count_words.c
```

This "chaining" of targets to prerequisites to targets to prerequisites is typical of how make analyzes a *makefile* to decide the commands to be performed.

The next prerequisite make considers is *lexer.o*. Again the chain of rules leads to *lexer.c* but this time the file does not exist. make finds the rule for generating *lexer.c* from *lexer.l* so it runs the flex program. Now that *lexer.c* exists it can run the gcc command.

Finally, make examines -lfl. The -l option to gcc indicates a system library that must be linked into the application. The actual library name indicated by "fl" is *libfl.a*. GNU make includes special support for this syntax. When a prerequisite of the form -l<NAME> is seen, make searches for a file of the form *libNAME.so*; if no match is found, it then searches for *libNAME.a*. Here make finds */usr/lib/libfl.a* and proceeds with the final action, linking.

Minimizing Rebuilds

When we run our program, we discover that aside from printing fees, fies, foes, and fums, it also prints text from the input file. This is not what we want. The problem is that we have forgotten some rules in our lexical analyzer and `flex` is passing this unrecognized text to its output. To solve this problem we simply add an "any character" rule and a newline rule:

```
            int fee_count = 0;
            int fie_count = 0;
            int foe_count = 0;
            int fum_count = 0;
%%
fee     fee_count++;
fie     fie_count++;
foe     foe_count++;
fum     fum_count++;
.
\n
```

After editing this file we need to rebuild the application to test our fix:

```
$ make
flex -t lexer.l > lexer.c
gcc -c lexer.c
gcc count_words.o lexer.o -lfl -ocount_words
```

Notice this time the file *count_words.c* was not recompiled. When `make` analyzed the rule, it discovered that *count_words.o* existed and was newer than its prerequisite *count_words.c* so no action was necessary to bring the file up to date. While analyzing *lexer.c*, however, `make` saw that the prerequisite *lexer.l* was newer than its target *lexer.c* so `make` must update *lexer.c*. This, in turn, caused the update of *lexer.o* and then *count_words*. Now our word counting program is fixed:

```
$ count_words < lexer.l
3 3 3 3
```

Invoking make

The previous examples assume that:

- All the project source code and the `make` description file are stored in a single directory.
- The `make` description file is called *makefile*, *Makefile*, or *GNUMakefile*.
- The *makefile* resides in the user's current directory when executing the `make` command.

When make is invoked under these conditions, it automatically creates the first target it sees. To update a different target (or to update more than one target) include the target name on the command line:

```
$ make lexer.c
```

When make is executed, it will read the description file and identify the target to be updated. If the target or any of its prerequisite files are out of date (or missing) the shell commands in the rule's command script will be executed one at a time. After the commands are run make assumes the target is up to date and moves on to the next target or exits.

If the target you specify is already up to date, make will say so and immediately exit, doing nothing else:

```
$ make lexer.c
make: `lexer.c' is up to date.
```

If you specify a target that is not in the *makefile* and for which there is no implicit rule (discussed in Chapter 2), make will respond with:

```
$ make non-existent-target
make: *** No rule to make target `non-existent-target'.  Stop.
```

make has many command-line options. One of the most useful is `--just-print` (or `-n`) which tells make to display the commands it would execute for a particular target without actually executing them. This is particularly valuable while writing *makefile*s. It is also possible to set almost any *makefile* variable on the command line to override the default value or the value set in the *makefile*.

Basic Makefile Syntax

Now that you have a basic understanding of make you can almost write your own *makefile*s. Here we'll cover enough of the syntax and structure of a *makefile* for you to start using make.

*Makefile*s are usually structured top-down so that the most general target, often called `all`, is updated by default. More and more detailed targets follow with targets for program maintenance, such as a `clean` target to delete unwanted temporary files, coming last. As you can guess from these target names, targets do not have to be actual files, any name will do.

In the example above we saw a simplified form of a rule. The more complete (but still not quite complete) form of a rule is:

$$\begin{aligned} &target_1\ target_2\ target_3 : prerequisite_1\ prerequisite_2 \\ &\quad command_1 \\ &\quad command_2 \\ &\quad command_3 \end{aligned}$$

One or more targets appear to the left of the colon and zero or more prerequisites can appear to the right of the colon. If no prerequisites are listed to the right, then only the target(s) that do not exist are updated. The set of commands executed to update a target are sometimes called the *command script*, but most often just the *commands*.

Each command *must* begin with a tab character. This (obscure) syntax tells `make` that the characters that follow the tab are to be passed to a subshell for execution. If you accidentally insert a tab as the first character of a noncommand line, `make` will interpret the following text as a command under most circumstances. If you're lucky and your errant tab character is recognized as a syntax error you will receive the message:

```
$ make
Makefile:6: *** commands commence before first target.  Stop.
```

We'll discuss the complexities of the tab character in Chapter 2.

The comment character for `make` is the hash or pound sign, #. All text from the pound sign to the end of line is ignored. Comments can be indented and leading whitespace is ignored. The comment character # does not introduce a `make` comment in the text of commands. The entire line, including the # and subsequent characters, is passed to the shell for execution. How it is handled there depends on your shell.

Long lines can be continued using the standard Unix escape character backslash (\). It is common for commands to be continued in this way. It is also common for lists of prerequisites to be continued with backslash. Later we'll cover other ways of handling long prerequisite lists.

You now have enough background to write simple *makefile*s. Chapter 2 will cover rules in detail, followed by `make` variables in Chapter 3 and commands in Chapter 5. For now you should avoid the use of variables, macros, and multiline command sequences.