man make

A PRIMER ON THE MAKE UTILITY

In the modern world of Integrated Development Environments, we forget what really goes into compiling a large code project. This article should be a refresher on (or teach for the first time) the basics of makefiles, the most underrated part of any code project.

Adrian Hannah

In a compiled language, the makefile is arguably the most important part of any programming project. To compile your project, you first have to compile each source file into an object file, which in turn needs to be linked with system libraries into the final executable file. Each command can have a considerable number of arguments added in. That's a lot of typing and a lot of potential for mistakes. The more source files you have, the more complex the compilation process becomes, unless you use makefiles. Most Linux users have at least a cursory knowledge of make and makefiles (because that's how we build software packages for our systems), but not much more than that. Most developers probably don't have too much in-depth experience with makefiles, because most Integrated Development Environments (IDEs) have the capability of managing makefiles for them. Although this is convenient most of the time, knowing more about how make works and what goes into makefiles can help you troubleshoot compilation errors down the road.

According to make's man page, "The purpose of the make utility is to determine automatically which pieces of a large program need to be recompiled, and issue the commands to recompile them." Essentially, make is used to determine efficiently (and without user error) which portions of the source code have been updated since the last compilation and recompile them. It can be used for more than just compiling programs. Because it isn't limited to any particular language, you can use it for anything you can come up with that relates to the modified date of a group of files.

Running make is a straightforward

IMPORTANT:

Command lines must be indented with tab characters; spaces cause funky errors. This has been a design flaw in make for decades. Empty lines must still have a tab character or else make will throw a fit.

process. The more convoluted portion of using make is constructing the makefile. The makefile is a file that consists of a series of rules that define the dependencies of your project. These rules govern the behavior of make during execution.

Listing 1. Example Makefile

```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.cpp hello.cpp factorial.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)


$(EXECUTABLE): $(OBJECTS)
        $(CC) $(LDFLAGS) $(OBJECTS) -o $@

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

Rules and Targets

Each rule in the makefile is an independent series of commands that are executed in order to build a target. Make does not necessarily run each rule in order. Make will run through the rules recursively, building each target in turn, based on modification. Rules are formatted like this:

```
target: dependency list ...

        commands

        ...
```

The target is typically the name of a file, but it can be a phony target (discussed later in this article). The dependency list is a space-separated list of files that designate whether the

THE BASICS:

- Comments start with a pound sign (#).
- Continuation of a line is denoted by a back slash (\).
- Lines containing equal signs (=) are variable definitions.
- Each command line typically is executed in a separate Bourne shell—that is, sh1.
- To execute more than one command line in the same shell, type them on the same line, separated by semicolons. Use a \ to continue the line if necessary.

target needs to be rebuilt. The commands can be any shell command, so long as the target is up to date at the end of them. It is imperative that you indent the commands with a tab character and not spaces. This is a design flaw in make that has yet to be fixed, and it will cause some strange and obscure errors should you use spaces instead of tabs in your makefile.

When make encounters a rule, it first checks the files listed in the dependency list to ensure that they haven't changed. If one of them has, make looks through the makefile for the rule containing that file as the target. This recursion continues until a rule is found where all the dependencies are unchanged or rebuilt (or have no further dependencies), and then make executes the listed commands for that rule before returning to the previous rule, and so on, until the root rule has been satisfied and its commands run.

You may use pattern-matching characters to describe dependencies in the dependency list or in commands, but they may not be used in the target.

Phony Targets

Phony targets (also called dummy or pseudo-targets) are not real files; they simply are aliases within the makefile. As I mentioned before, you can specify targets from the command line, and this is precisely what phony targets are used for. If you're familiar with the process of using make to build applications on your system, you're familiar with `make install` (which installs the application after compiling the source) or `make clean` (which cleans up the temporary files created while compiling the source). These are two examples of phony targets. Obviously, there are no "install" or "clean" files in the project; they're just aliases to a set of commands set aside to complete some task not dependent on the modification time of any particular file in the project. Here is an example of using a "clean" phony target:

```
clean:
	-rm *.o my_bin_file
```

Special Targets

Some special targets are built in to make. These special targets hold special meaning, and they modify the way make behaves during execution:

.PHONY — this target signifies which other targets are phony targets. If a target is listed as a dependency of .PHONY, the check to ensure that the target file was updated is not performed. This is useful if at any time your project actually produces a file named the same as a phony target; this check always will fail when executing your phony target.

.SUFFIXES — the dependency list of this target is a list of the established file suffixes for this project. This is helpful when you are using suffix rules (discussed later in this article).

.DEFAULT — if you have a bunch of targets that use the same set of commands, you may consider using the .DEFAULT target. It is used to specify the commands to be executed when no rule is found for a target.

.PRECIOUS — all dependencies of the .PRECIOUS target are preserved should make be killed or interrupted.

.INTERMEDIATE — specifies which targets are intermediate, or temporary, files. Upon completion, make will delete all intermediate files before terminating.

.SECONDARY — this target is similar to .INTERMEDIATE, except that these files will not be deleted automatically upon completion. If no dependencies are specified, all files are considered secondary.

.SECONDEXPANSION — after the initial read-in phase, anything listed after this target will be expanded for a second time. So, for example:

```
.SECONDEXPANSION:
ONEVAR = onefile
TWOVAR = twofile
myfile: $(ONEVAR) $$(TWOVAR)
```

will expand to:

```
.SECONDEXPANSION:
ONEVAR = onefile
TWOVAR = twofile
myfile: onefile $(TWOVAR)
```

after the initial read-in phase, but because I specified .SECONDEXPANSION, it will expand everything following a second time:

```
.SECONDEXPANSION:
ONEVAR = onefile
TWOVAR = twofile
myfile: onefile twofile
```

I'm not going to elaborate on this here, because this is a rather complex subject and outside the scope of this article, but you can find all sorts of .SECONDEXPANSION goodness out there on the Internet and in the GNU manual.

.DELETE_ON_ERROR — this target will cause make to delete a target if it has changed and any of the associated commands exit with a nonzero status.

.IGNORE — if an error is encountered while building a target list as a dependency of .IGNORE, it is ignored. If there are no dependencies to .IGNORE, make will ignore errors for all targets.

.LOW_RESOLUTION_TIME — for some reason or another, if you have files that will have a low-resolution timestamp (missing the subsecond portion), this target allows you to designate those files. If a file is listed as a dependency of .LOW_RESOLUTION_TIME, make will compare times only to the nearest second between the target and its dependencies.

.SILENT — this is a legacy target that causes the command's output to be suppressed. It is suggested that you use Command Echoing (discussed in the Command Special Characters section) or by using the `-s` flag on the command line.

.EXPORT_ALL_VARIABLES — tells make to export all variables to any child processes created.

.NOTPARALLEL — although make can run simultaneous jobs in order to complete a task faster, specifying this target in the makefile will force make to run serially.

.ONESHELL — by default, make will invoke a new shell for each command it runs. This target causes make to use one shell per rule.

.POSIX — with this target, make is forced to conform to POSIX standards while running.

Variables

In other versions of make, variables are called macros, but in the GNU version (which is the version you likely are using), they are referred to as variables, which I personally feel is a more appropriate title. Nomenclature aside, variables are a convenient way to store information that may be used multiple times throughout the makefile. It becomes abundantly clear the first time you write a makefile and then realize that you forgot a command flag for your compiler in all 58 rules you wrote. If I had used variables to designate my compiler flags, I'd have had to change it only once instead of 58 times. Lesson learned. Set these at the beginning of your makefile before any rules. Simply use:

```
VARNAME = information stored in the variable
```

to set the variable, and do use `$(VARNAME)` to invoke it throughout the makefile. Any shell variables that

PREDEFINED VARIABLES

- `$?` — evaluates to the list of components that are younger than the current target. Can be used only in description file command lines.
- `$@` — evaluates to the current target name. Can be used only in description file command lines.
- `$$@` — also evaluates to the current target name. However, it can be used only on dependency lines.
- `$<` — the name of the related file that caused the action (the precursor to the target). This is only for suffix rules.
- `$*` — the shared prefix of the target and dependent—only for suffix rules.

COMMON VARIABLES FOR C++ PROGRAMMING

- `CC` — the name of the compiler.
- `DEBUG` — the debugging flag. This is -g in both g++ and cxx. The purpose of the flag is to include debugging information into the executable, so that you can use utilities like gdb to debug the code.
- `LFLAGS` — the flags used in linking. As it turns out, you don't need any special flags for linking. The option listed is `-Wall`, which tells the compiler to print all warnings. But, that's fine. We can use that.
- `CFLAGS` — the flags used in compiling and creating object files. This includes both `-Wall` and `-c`. The `-c` option is needed to create object files—that is, .o files.

existed prior to calling make will exist within make as variables and, thus, are invoked the same way as variables. You can specify a variable from the command line as well. Simply add it to the end of your make command, and it will be used within the make execution.

If, at some point, you need to alter the data stored in a variable temporarily, there is a very simple way to substitute in this new data without overwriting the variable. It's done using the following format:

```
$(VARNAME:find=replace)
```

where `find` is the substring you are trying to find, and `replace` is the string to replace it with. So, for instance:

```
LETTERS = abcxyz xyzabc xyz
print:
echo $(LETTERS:xyz=def)
```

will produce the output `abcdef xyzabc def`.

Suffix Rules

In certain situations, you will find that the rules for a certain file type are identical except for the filename. For instance, a lot of times in a C project, you will see rules like this:

```
file.o: file.c
        cc -O -Wall file.c
```

because for every .c file, you need to make the intermediate .o file, so that the end binary then can be built. Suffix rules are a way of minimizing the amount of time you spend writing out rules and the number of rules in your makefile. In order to use suffix rules, you need to tell make which file suffixes are considered significant (suffix rules won't work unless the suffix is defined this way), then write the generic rule for the suffixes. In the case described above, you would do this:

```
.SUFFIXES: .o .c

.c.o:
        cc -O -Wall $<
```

You may note that in the case of suffix rules, the dependency suffix goes before the target suffix, which is a reversal from the normal order in a makefile. You also will see that you use `$<` in the command, which evaluates to the .c filename associated with the .o file that triggered the rule. There are a couple predefined variables like this that are used exclusively for suffix rules:

- `$<` — evaluates to the component that is being used to make the target—that is, file.c.

- `$*` — evaluates to the filename part (without any suffix) of the component that is being used to make the target—that is, file.

Note that the `$?` variable cannot occur in suffix rules, but the `$@` variable still will work.

Command Special Characters

Certain characters can be used in conjunction with commands to alter the behavior of make or the command. If you're familiar with shell scripting, you'll recognize that \ signifies a line continuation. That is to say, using \ means that the command isn't finished and continues on the next line. Nobody likes looking at a messy file, and using this character at the end of a line helps keep your makefile clean and pretty. If a rule has more than one command, use a semicolon to separate commands. You can start a command with a hyphen, and make will ignore any errors that occur from the command. If you want to suppress the output of a command during execution, start the command with an at sign (@).

Using these symbols will allow you to make a more usable and readable makefile.

Directives

Sometimes, you need more control over how the makefile is read and executed. Directives are designed exactly for that purpose.

From defining, overriding or exporting variables to importing other makefiles, these directives are what make a more robust makefile possible. The most useful of the directives are the conditional directives though.

Conditional directives allow you to define multiple versions of a command based on preexisting conditions. For

Conditional directives allow you to define multiple versions of a command based on preexisting conditions.

example, say you have a set of libraries you want included in your binary only if the compiler used is gcc:

```
libs_for_gcc = -lgnu
normal_libs =

foo: $(objects)
ifeq ($(CC),gcc)
        $(CC) -o foo $(objects) $(libs_for_gcc)
else
        $(CC) -o foo $(objects) $(normal_libs)
endif
```

In this example, you use `ifeq` to check if CC equals `gcc` and if it does, use the gcc libraries; otherwise, use the generic libraries.

This is just a small, basic sampling of the things you can do with make and makefiles. There are so many more complex and interesting things you can do, you just have to dig around to find them! ■

Adrian Hannah has spent the past 15 years bashing keyboards to make computers do what he tells them. He currently is working as a system administrator for the federal government. He is a jack of all trades and a master of none. He spends all his waking hours on the *Linux Journal* IRC channel, on Twitter (@codemoney2841) and talking to random chat bots on the Internet.

Resources

GNU make comes with most Linux distributions by default, but it can be found on the main GNU FTP server: **ftp.gnu.org/gnu/make** (via HTTP) and **ftp.gnu.org/gnu/make** (via FTP). It also can be found on the GNU mirrors at **www.gnu.org/prep/ftp.html**.

Documentation for make is available on-line at **www.gnu.org/software/make/manual**, as is documentation for most GNU software. You also can find more information about make by running `info make` or `man make`, or by looking at /usr/doc/make/, /usr/local/doc/make/ or similar directories on your system. A brief summary is available by running `make --help`.