6 September 1995
Tom Anastasio

Updated 11 September 2009
Dennis Frey

These notes have been excerpted from the GNU manual on
makefiles. To see the whole manual, enter emacs, then
type meta-x info
Follow the directions there.

What is "make"?

The "make" program is a Unix utility that helps manage large software projects by automating the compilation and linking of files to create an executable file. The "make" program uses the timestamps of your files to determine which files require
recompilation in order to "make" a target file. Only files which have been changed are recompiled. Files which have not been changed since the last time the target file was "made" are not recompiled, hence reducing the time to "make" the target file. This is very convenient when working on a small part
of a larger project (such as a function that you are debugging) and the rest of the
project is not being changed.

Compilation and linking commands are placed into a "makefile" which is read and interpreted by the make program, removing the need to retype these commands over and
over again.

Preparing and Running Make

To prepare to use `make', you must write a file called the "makefile" that describes the relationships among files in your program and provides commands for updating each file. In a program, typically, the executable file is updated from object files, which are in turn made by compiling source files.

Once a suitable makefile exists, each time you change some source files, this simple shell command:

```
make
```

suffices to perform all necessary recompilations. The "make" program uses the makefile data base and the last-modification times of the files to decide which of the files need to be updated. For each of those files, it issues the commands recorded in the data base.

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).

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.

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' (*note Phony Targets::.).

A "dependency" is a file that is used as input to create the target. A target often depends on several files, e.g. a .o file depends on .c and .h files; an executable depends on .o 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. *Note Writing Rules: Rules.

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.

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
        gcc -o edit main.o kbd.o command.o display.o \
                   insert.o search.o files.o utils.o

main.o : main.c defs.h
        gcc -c main.c
kbd.o : kbd.c defs.h command.h
        gcc -c kbd.c
command.o : command.c defs.h command.h
        gcc -c command.c
display.o : display.c defs.h buffer.h
        gcc -c display.c
insert.o : insert.c defs.h buffer.h
        gcc -c insert.c
search.o : search.c defs.h buffer.h
        gcc -c search.c
files.o : files.c defs.h buffer.h command.h
        gcc -c files.c
utils.o : utils.c defs.h
        gcc -c utils.c
clean :
        rm -f 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
```

or

```
make edit
```

To use this makefile to create the object file search.o, type

```
make search.o
```

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 `gcc -c main.c' and `gcc -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".

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.

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.

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 (based on the file's timestamp) 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'.

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
        gcc -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 or "macros". "Variables" allow a text string to be defined once and substituted in multiple places later. Variable are very similar to #defines in C/C++.

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. UPPERCASE variables are preferred.
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 `$(OJBECTS)'.

Here is how the 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)
        gcc -o edit $(OBJECTS)
main.o : main.c defs.h
        gcc -c -Wall main.c
kbd.o : kbd.c defs.h command.h
```

```
        gcc -c -Wall kbd.c
command.o : command.c defs.h command.h
        gcc -c -Wall command.c
display.o : display.c defs.h buffer.h
        gcc -c -Wall display.c
insert.o : insert.c defs.h buffer.h
        gcc -c -Wall insert.c
search.o : search.c defs.h buffer.h
        gcc -c -Wall search.c
files.o : files.c defs.h buffer.h command.h
        gcc -c -Wall files.c
utils.o : utils.c defs.h
        gcc -c -Wall utils.c

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

Common Makefile Variables
==========

Note that many of rules above have very similar commands -- they compile .c files.
It's therefore common to create the variable CC for the name of the compiler, so that if
you decide to change the compiler you're using, it needs only be changed in one place.
For example, we might define the name of the compiler as

```
CC=/usr/local/bin/gcc
```

Similarly, we often want to pass "flags" to the compiler for options such as the level of warnings and optimization. It's common to create the variable CFLAGS
which is list of compiler flags (run the command 'man gcc' for a full list of compiler
options.
We might define CFLAGS as

```
CFLAGS= -c -Wall
```

where -c means "just compile" and -Wall means "display all warnings".

Sometimes you must link C libraries (such as the math library) with your own .o files
to create your executable. It is therefore common to create a variable named LIBS or LIBRARIES which is a list of C libraries to be used when linking.
We would define LIBS as

```
LIBS = -lm
```

to include the math library (-l means "this is a library flag" and m refers to the
math library). If there are no libraries required, you can define the variable as

```
LIBS=
```

In order to use the gdb debugger with your executable, you must tell the compiler
and linker to save information about variable and function names that it would otherwise
discard. The -g compiler flag tells the compiler/linker to save this information.

You might include the -g as part of CFLAGS, or define a separate variable such as

```
DEBUG = -g
```

If you don't want debugging information, define DEBUG as

```
DEBUG=
```

Here is how the simple makefile looks when you use the macros described above

```
CC = /usr/local/bin/gcc
CFLAGS = -c -Wall
DEBUG  = -g
LIBS = -l
OBJECTS = main.o kbd.o command.o display.o \
          insert.o search.o files.o utils.o

edit : $(OBJECTS)
       $(CC) -o edit $(CFLAGS) $(OBJECTS) $(LIBS)
main.o : main.c defs.h
       $(CC) $(CFLAGS) main.c
kbd.o : kbd.c defs.h command.h
       $(CC) $(CFLAGS) kbd.c
command.o : command.c defs.h command.h
       $(CC) $(CFLAGS)  command.c
display.o : display.c defs.h buffer.h
       $(CC) $(CFLAGS) display.c
insert.o : insert.c defs.h buffer.h
       $(CC) $(CFLAGS) insert.c
search.o : search.c defs.h buffer.h
       $(CC) $(CFLAGS) search.c
files.o : files.c defs.h buffer.h command.h
       $(CC) $(CFLAGS) files.c
utils.o : utils.c defs.h
       $(CC) $(CFLAGS) utils.c

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

Implicit rules

The "make" command has some knowledge of file types and can figure out how to create some targets that are not explicitly listed in your makefile. For example, make "knows" that to create a .o file it must compile the corresponding .c file. In the abseenc of a rule, make will use the command "gcc -c" For example, it will use the command

```
`gcc -c main.c -o main.o'
```

to compile `main.c' into `main.o' if main.o is not found as a target in your makefile.

Note that using the implicit rule may not be what you want since no flags (e.g -Wall or -g) are passed to the compiler. see Advanced Topics below.

Rules for Other Tasks

==========

Compiling a program is not the only thing for which you might want to write rules. 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', or submitting your files for grading.

Here is how we could write a `make' rule for cleaning our directory which contains
our files for our editor example

```
clean:
        rm -f 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 -f 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'.

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'.

We can even include a rule to submit our files for grading. Again in this example,
our target is not a file, so it is labeled as PHONY. If our editor example was Project 5 for CMSC 313, we might write a rule such as this

```
.PHONY : submit
submit:
      submit cs313 Proj5 main.c kybd.c command.c display.c \
            insert.c search.c files.c utils.c \
            defs.h buffer.h command.h
```

Common Makefile Errors
==========

Probably the most common error when writing a makefile if failing to use a TAB at the beginning of the command line(s). Since TABs are hard to see, you must often
open your makefile in an editor, go to the command line and move the cursor to the
right to verify that there's a TAB and not spaces. If there is no TAB, make will
complain that there is no command to run.

Another common mistake is to inadvertently have TABs at the beginning of blank lines.
This mistake causes make to complain that there is a blank command.

Miscellaneous

1. Makefile may (and probably should) contain comments. The comment character for makefiles is '#'. All text from the # to the end of the line are ignored.

2. By default the "make" command looks for a file named "makefile".
If no such fileexists, "make" looks for a file named "Makefile".
If your makefile is named something else, you can specify the name of the makefile to the make command using the -f flag.
For example

```
make -f MyMakefile
```

3. To use the value of a variable (eg. CC), either $(CC) or ${CC} are allowed

4. A target may have multiple command lines listed one after the other, each indented
with a TAB. For example

```
project1.o : project1.c utility.h
      echo Compiling project1.c
      gcc -c -Wall project1.c
```

Advanced Topics

1. When there are many files, it may be inconvenient to write a rule for compiling each .c file to make a .o file. We can specify our own default rule for compiling .c files as shown below.

```
.c.o:
      $(CC) $(CFLAGS) $(DEBUG) $<
```

The first line -- .c.o -- tells make that we are define a default rule for making .o files from .c files. The second line is the command to execute to make the .o file. The "$<" is a special makefile symbol that is expanded to be the name of the .c file that caused the default rule to be used.