by P.N. Hilfinger (U.C. Berkeley)
modified by M. Clancy (UCB) and C. Bono

Basic Compilation Control with Make

Even relatively small software systems can require rather involved, or at least tedious, sequences of instructions to translate them from source to executable forms. Furthermore, since translation takes time (more than it should) and systems generally come in separately-translatable parts, it is desirable to save time by updating only those portions whose source has changed since the last compilation. However, keeping track of and using such information is itself a tedious and error-prone task, if done by hand.

The UNIX `make` utility is a conceptually-simple and general solution to these problems. It accepts as input a description of the interdependencies of a set of source files and the commands necessary to compile them, known as a *makefile*; it examines the ages of the appropriate files; and it executes whatever commands are necessary, according to the description. For further convenience, it will supply certain standard actions and dependencies by default, making it unnecessary to state them explicitly.

Though conceptually simple, the `make` utility has accreted features with age and use, and is rather imposing in the glory of its full definition. This document describes only the simple use of `make`. It is applicable both to the original `make` facility and to the new `gmake` (GNU `make`) program, a ‘‘copylefted’’ and rather more powerful version.

Basic Operation and Syntax

The following is a sample makefile for compiling a simple editor program, `edit`. (Adapted from ‘‘GNU Make: A Program for Directing Recompilation’’ by Richard Stallman and Roland McGrath, 1990. This is available as part of the GNU source code.)

from eight `.c` files and three header (`.h`) files.

```
# Makefile for simple editor

edit : main.o kbd.o commands.o display.o \
       insert.o search.o files.o utils.o
        gcc -g -o edit main.o kbd.o commands.o display.o \
                   insert.o search.o files.o utils.o

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

```
        gcc -g -c utils.c
```

This file consists of a sequence of nine *rules*. Each rule consists of a line containing two lists of names separated by a colon, followed by one or more lines beginning with tab characters. Any line may be continued, as illustrated, by ending it with a backslash-newline combination, which essentially acts like a space, combining the line with its successor. The '#' character indicates the start of a comment that goes to the end of the line.

The names preceding the colons are known as *targets*; they are most often the names of files that are to be produced. The names following the colons are known as *dependencies* of the targets. They usually denote other files (generally, other targets) that must be present and up-to-date before the target can be processed. The lines starting with tabs that follow the first line of a rule we will call *actions*. They are shell commands that get executed in order to create or update the target of the rule (we'll use the generic term *update* for both).

Each rule says, in effect, that to update the targets, each of the dependencies must first be updated (recursively). Next, if a target does not exist (that is, if no file by that name exists) or if it does exist but is older than one of its dependencies, the actions of the rule are executed to create or update that target. The program will complain if any of the dependencies does not exist and there is no rule for creating it. To start the process off, the user who executes the `make` utility specifies one or more targets to be updated. The first target of the first rule in the file is the default.

In the example above, `edit` is the default target. The first step in updating it is to update all the object (`.o`) files listed as dependencies. To update `main.o`, in turn, requires first that `main.c` and `defs.h` be updated. Presumably, `main.c` is the source file that produces `main.o` and `defs.h` is a header file that `main.c` includes. There are no rules targeting these files; therefore, they merely need to exist to be up-to-date. Now `main.o` is up-to-date if it is younger than either `main.c` or `defs.h` (if it were older, it would mean that one of those files had been changed since the last compilation that produced `main.o`). If `main.o` is older than its dependencies, `make` executes the action "`gcc -g -c main.c`", producing a new `main.o`. Once `main.o` and all the other `.o` files are updated, they are combined by the action "`gcc -g -o edit . . .`" to produce the program `edit`, if either `edit` does not already exist or if any of the `.o` files are younger than the existing `edit` file.

To invoke the `make` for this example, one issues the command

`make -f` *makefile-name target-names*

where the *target-names* are the targets that you wish to update and the *makefile-name* given in the `-f` switch is the name of the makefile. By default, the target is that of the first rule in the file and the makefile name is either `makefile` or `Makefile`, whichever exists. It is typical to arrange that each directory contains the source code for a single principal program. By adopting the convention that the rule with that program as its target goes first, and that the makefile for the directory is named `makefile`, you can arrange that, by convention, issuing the command `make` with no arguments in any directory will update the principal program of that directory.

It is possible to have more than one rule with the same target, as long as no more than one rule for each target has an action. Thus, we can also write the latter part of the example above as follows.

```
main.o : main.c
        gcc -g -c main.c
```

```
kbd.o : kbd.c
        gcc -g -c kbd.c
commands.o : command.c
        gcc -g -c commands.c
display.o : display.c
        gcc -g -c display.c
insert.o : insert.c
        gcc -g -c insert.c
search.o : search.c
        gcc -g -c search.c
files.o : files.c
        gcc -g -c files.c
utils.o : utils.c
        gcc -g -c utils.c

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

The order in which these rules are written is irrelevant. Which order or grouping you choose is largely a matter of taste.

The example of this section illustrates the concepts underlying `make`. The rest of `make`'s features exist mostly to enhance the convenience of using it.

Variables

The dependencies of the target `edit` in the last section are also the arguments to the command that links them. One can avoid this redundancy by defining a variable that contains the names of all object files.

```
# Makefile for simple editor

OBJS = main.o kbd.o commands.o display.o \
       insert.o search.o files.o utils.o

edit : $(OBJS)
        gcc -g -o $ $(OBJS)
```

The (continued) line beginning ''`OBJS =`'' defines the variable `OBJS`, which can later be referenced as ''$(OBJS)'' or ''${OBJS}''. These later references cause the definition of `OBJ` to be substituted verbatim before the rule is processed. The special variable '`$@`' is set in each rule to the target of that rule. It is somewhat unfortunate that both `make` and the shell use '$' to prefix variable references; `make` defines '$$' to be simply '$', thus allowing you to send '$'s to the shell, where needed.

Variables may also be set in the command line that invokes `make`. For example, if the makefile contains

```
main.o: main.c
        gcc $(DEBUG) -c main.c
```

Then a command such as

```
make DEBUG=-g ...
```

will cause the compilations to use the `-g` (add symbolic debugging information) switch, while leaving off the `DEBUG=-g` will not use the `-g` switch. Variable definitions in the command lines override those in the makefile, which allows the makefile to supply defaults.

Finally, variables not set by either of these methods may be set as UNIX environment variables. Thus, the sequence of commands

```
setenv DEBUG -g
make ...
```

for this last example will also use the `-g` switch during compilations.

Implicit rules

In the example from the first section, all of the compilations that produced `.o` files have the same form. It is tedious to have to duplicate them; it merely gives you the opportunity to type something wrong. Therefore, `make` can be told about---and for some standard cases, already knows about---the default files and actions needed to produce files having various extensions. For our purposes, the most important is that it knows how to produce a file *F*`.o` given a file of the form *F*`.c`, and knows that the *F*`.o` file depends on the file *F*`.c`. Specifically, `make` automatically introduces (in effect) the rule

F`.o : `*F*`.c`
`	$(CC) -c $(CFLAGS)` *F*`.c`

when called upon to produce *F*`.o` when there is a file *F*`.c` present, but no explicitly specified actions for producing *F*`.o`.

As a result, the example may be abbreviated as follows.

```
# Makefile for simple editor

OBJS = main.o kbd.o commands.o display.o \
       insert.o search.o files.o utils.o

CC = gcc

CFLAGS = -g

edit : $(OBJS)
        gcc -g -o $@ $(OBJS)

main.o : defs.h
kbd.o : defs.h command.h
commands.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
```

There are quite a few other such implicit rules built into `make`. The `-p` switch will cause `make` to list them somewhat cryptically, if you are at all curious. We are most likely to be using the rules for creating `.o` files from `.c` files or from `.s` (assembler) files. It is also possible to supply your own default rules and to suppress the standard rules; for details, see the full documentation.

Special actions

It is often useful to have targets for which there are never any corresponding files. If the actions for a target do not create a file by that name, it follows from the definition of how `make` works that the actions for that target will be executed each time `make` is applied to that target. A common use is to put a standard ''clean-up'' operation into each of your makefiles, specifying how to get rid of files that can be reconstructed, if necessary. For example, you will often see a rule like this in a makefile.

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

Every time you issue the shell command `make clean`, this action will execute, removing all `.o` files.

Details of actions

By default, each action line specified in a rule is executed by the Bourne shell (as opposed to the C-shell, which is more commonly used here). For the simple makefiles we are likely to use, this will make little difference, but be prepared for surprises if you get ambitious.

The `make` program usually prints each action as it is executed, but there are times when this is not desirable. Therefore, a '@' character at the beginning of an action suppresses the default printing. Here is an example of a common use.

```
edit : $(OBJS)
        @echo Linking $@...
        @gcc -g -o $@ $(OBJS)
        @echo Done
```

The result of these actions is that when `make` executes this final editing step for the `edit` program, the only thing you'll see printed is a line reading ''`Linking edit...`'' and, at the end of the step, a line reading ''`Done`''.

When `make` encounters an action that returns a non-zero exit code, the UNIX convention for indicating an error, its standard response is to end processing and exit. The error codes of action lines that begin with a '-' sign (possibly preceded by a '@') are ignored. Also, the `-k` switch to `make` will cause it to abandon processing only of the current rule (and any that depend on its target) upon encountering an error, allowing processing of ''sibling'' rules to proceed.

Using makefiles with C++

One of the differences between `make` and `gmake` relates to the variables used in the implicit rules for compiling C++ programs. What I will describe here only works with `gmake`.

If you use the suffix `.cc`, it will create a `.o` file with the same prefix using the compiler given by the variable `CXX`, and the flags given by the variable `CXXFLAGS`. The default value for `CXX` is `g++`. Since that is the compiler we want to use, we don't have to set this variable in our makefile. Thus `.cc`, `CXX` and `CXXFLAGS` for compiling C++ programs are analogous to `.c`, `CC`, and `CFLAGS` for compiling C programs (see earlier section on implicit rules).

Here is a simple makefile that creates the executable `fracts` from the source files `Fraction.cc`, `testfracts.cc`, and the header file `Fraction.h` (which is included in both of the source files). Remember, this will only work with `gmake`.

```
CXXFLAGS = -g -Wall

OBJS = Fraction.o testfracts.o

fracts: $(OBJS)
        $(CXX) $(CXXFLAGS) $(OBJS) -o fracts

$(OBJS): Fraction.h
```