UNIVERSITY OF CALIFORNIA
Department of Electrical Engineering
and Computer Sciences
Computer Science Division

CS164
Spring 2009

P. N. Hilfinger

Basic Compilation Control with `Gmake`

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.

There are numerous dialects of `make`, both among UNIX installations and (under other names) in programming environments for personal computers. In this course, we will use a version known as `gmake`[1]. 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 `gmake`. Relatively complete documentation is available on-line in Emacs (use `C-h i` and select the "Gmake" menu item).

1 Basic Operation and Syntax

The following is a sample makefile[2] for compiling a simple editor program, `edit`, from eight `.cc` files and three header (`.h`) files.

[1]For "GNU `make`," GNU being an acronym for "GNU's Not Unix." `gmake` is "copylefted" (it has a license that *requires* unrestricted use of any product containing it, if distributed). It is also more powerful than the standard `make` utility.

[2]Adapted from "GNU Make: A Program for Directing Recompilation" by Richard Stallman and Roland McGrath, 1988.

```
# Makefile for simple editor

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

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

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 is, commands that you could type in response to the Unix prompt) 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 (so that one of its dependencies was changed after it was last updated), the actions of the rule are executed to create or update that target. The program will complain if any dependency does not exist and there is no rule for creating it. To start the process off, the user who executes the `gmake` 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 `edit.o`, in turn, requires first that `edit.cc` and `defs.h` be updated. Presumably, `edit.cc` is the source file that produces `edit.o` and `defs.h` is a header file that `edit.cc` includes. There are no rules targeting these files; therefore, they merely need to exist to be up-to-date. Now `edit.o` is up-to-date if it is younger than either `edit.cc` or `defs.h` (if it were older, it would mean that one of those files had been changed since the last compilation that produced `edit.o`). If `edit.o` is older than its dependencies, `gmake` executes the action "`gcc -g -c -Wall edit.cc`", producing a new `edit.o`. Once `edit.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 `gmake` for this example, one issues the command

`gmake -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. You may (and usually do) leave off `-f` *makefile-name*, in which case it defaults to 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 `gmake` 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.

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

edit.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 `gmake`. The rest of `gmake`'s features exist mostly to enhance the convenience of using it.

2 Variables

The dependencies of the target `edit` in §1 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 = edit.o kbd.o commands.o display.o \
       insert.o search.o files.o utils.o

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

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

You will sometimes find that you need a value that is just like that of some variable, with a certain systematic substitution. For example, given a variable listing the names of all source files, you might want to get the names of all resulting `.o` files. We can rewrite the definition of OBJS above to get this.

```
SRCS = edit.cc kbd.cc commands.cc display.cc \
       insert.cc search.cc files.cc utils.cc
OBJS = $(SRCS:.cc=.o)
```

The substitution suffix '`:.cc=.o`' specifies the desired substitution. We now have variables for both the names of all sources and the names of all object files without having to repeat a lot of file names (and possibly make a mistake).

You may set variables in the command line that invokes `gmake`. For example, if the makefile contains

```
edit.o: edit.cc
        gcc $(DEBUG) -c -Wall edit.cc
```

Then a command such as

```
gmake 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.

You may use UNIX environment variables for variables not set by either of these methods. Thus, the sequence of commands

```
setenv DEBUG -g
gmake ...
```

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

3 Implicit rules

In the example from §1, 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, `gmake` 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*`.cc`, and knows that the *F*`.o` file depends on the file *F*`.cc`. Specifically, `gmake` automatically introduces (in effect) the rule

```
F.o : F.cc
        $(CXX) -c -Wall $(CXXFLAGS) F.cc
```

when called upon to produce *F*`.o` when there is a C++ file *F*`.cc` present, but no explicitly specified actions exist for producing *F*`.o`. The use of the prefix "CXX" is a naming convention for variables that have to do with C++. It also creates the command

```
F : F.o
        $(CXX) $(LDFLAGS) F.o $(LOADLIBES) -o F
```

to tell how to create an executable file named *F* from *F*`.o`.

As a result, we may abbreviate the example as follows.

```
# Makefile for simple editor

SRCS = edit.cc kbd.cc commands.cc display.cc \
       insert.cc search.cc files.cc utils.cc

OBJS = $(SRCS:.cc=.o)

CXX = g++

CXXFLAGS = -g

LOADLIBES = -lm

edit : $(OBJS)
edit.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 `gmake`. The `-p` switch will cause `gmake` 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 `.cc` (C++) files. It is also possible to supply your own default rules and to suppress the standard rules; for details, see the full documentation.

3.1 Chains of rules

GMAKE works for any kind of files that are built in some systematic way from other files, not just C++. For example, you might want to generate a `parser.o` file from a file `parser.cc`, which in turn comes from `parser.y` using the program Bison. If you add `parser.cc` to the definition of `SRCS` in our running example, and add

```
parser.cc: parser.y
        bison -v -o parser.cc parser.y
```

to your makefile, you'll get exactly this effect. Bison figures out that to make `edit`, it needs `parser.o`, which causes it to create or update `parser.cc`, which causes it to run `bison`.

3.2 Phony targets

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 GMAKE works that the actions for that target will be executed each time GMAKE is applied to that target (because it will think the target is missing). 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.

```
.PHONY: clean

clean:
        rm -f *.class *~
```

Every time you issue the shell command "`gmake clean`," this action will execute, removing all `.class` files and Emacs old-version files.

The special `.PHONY` target tells GMAKE that `clean` is not a file, and is instead just the name of a target that is *always* out of date. Therefore, when you make the "`clean`" target, GMAKE will always execute the `rm` command, regardless of what files happen to be lying around. In effect, `.PHONY` tells GMAKE to treat `clean` as a command.

Another possible use is to provide a standard way to run a set of tests on your program—what are typically known as *regression tests*—to see that it is working and has not "regressed" as a result of some change you've made. For example, to cause the command

```
make check
```

to feed a test file through our editor program and check that it produces the right result, use:

```
.PHONY: check

check: edit
    rm -f test-file1
    java edit < test-commands1
    diff test-file1 expected-test-file1
```

where the test input file `test-commands1` presumably contains editor commands that are supposed to produce a file `test-file1`, and the file `expected-test-file1` contains what is supposed to be in `test-file1` after executing those commands. The first action line of the rule clears away any old copy of `test-file1`; the second runs the editor and feeds in `test-commands1` through the standard input, and the third compares the resulting file with its expected contents. If either the second or third action fails, GMAKE will report that it encountered an error.

4 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, most unfortunately, is more commonly used here). For the simple makefiles we are likely to use, this will gmake little difference, but be prepared for surprises if you get ambitious.

The `gmake` 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 edit ...
        @gcc -g -o edit $(OBJS)
        @echo Done
```

The result of these actions is that when `gmake` 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 `gmake` 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 `gmake` 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.

5 Creating makefiles

A good way to create makefiles is to have a template that you tailor to your particular program. Something like the example in Figure 1, for example. By simply copying this template and replacing the angle-bracketed portions with the particulars of your program (as in Figure 2, you get a working makefile. We will maintain a template like this in `$master/lib/Makefile.tmplt`.

As a final convenience, you may use the `-MM` option to `gcc` to create the dependencies automatically, and then add them into your modified makefile. This is the purpose of the `depend` special target in Figure 1.

```
PROGRAM = <REPLACE WITH PROGRAM NAME>

LOADLIBES = <EXTRA LOAD LIBRARIES>

CXX_SRCS = <C++ SOURCE FILE NAMES>

LDFLAGS = -g

CXX = g++

CXXFLAGS = -g -Wall

OBJS = $(CXX_SRCS:.cc=.o)

$(PROGRAM) : $(OBJS)
        $(CXX) $(LDFLAGS) $(OBJS) $(LOADLIBES) -o $(PROGRAM)

# Create the dependencies file, if it does not exist
dependencies:
        depend

clean:
        /bin/rm -f *.o $(PROGRAM) *~

depend:
        $(CXX) -MM $(CXX_SRCS) > dependencies

include dependencies
```

Figure 1: An example of a Makefile template that can be tailored to many simple C++ programs.

```
PROGRAM = edit

LOADLIBES = -lm

CXX_SRCS = edit.cc kbd.cc commands.cc display.cc \
           insert.cc search.cc files.cc utils.cc

LDFLAGS = -g

CXX = gcc

CXXFLAGS = -g -Wall

OBJS = $(CXX_SRCS:.cc=.o)

.PHONY: clean depend

$(PROGRAM) : $(OBJS)
        $(CXX) $(LDFLAGS) $(OBJS) $(LOADLIBES) -o $(PROGRAM)

# Create the dependencies file, if it does not exist
dependencies:
        depend

clean:
        /bin/rm -f *.o $(PROGRAM) *~

depend:
        $(CXX) -MM $(CXX_SRCS) > dependencies

include dependencies
```

The generated file 'dependencies':

```
edit.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
```

Figure 2: A Makefile created by modifying the template in Figure 1.