School of Computer Science and Applied Mathematics
UNIVERSITY OF THE WITWATERSRAND, JOHANNESBURG

COMS4040ACOMS7045A: High Performance Computing & Scientific Data Management
Make tutorial

2020-2-7

The `make` utility automatically determines which pieces of of a large program need to be recompiled, and issues commands to recompile them. This tutorial presents a quick introduction to GNU `make`. For more details, you should consult the `make` documentation [1]. The examples show C programs, but you can use `make` with any programming language whose compiler can be run with a shell command.

1 Preparing and Running Make

To prepare to use `make`, you must write a file called '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. If you run `make` program, it will look for a file named 'makefile' in your directory, and then execute it. 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 recipes recorded in the data base.

2 An Introduction to Makefiles

You need a file called 'makefile' (or 'Makefile') to tell `make` what to do. Most often, the makefile tells `make` how to compile and link a program.

2.1 Rule Introduction

A simple makefile consists of "rules" with the following shape

```
target: prerequisites
  recipe
```

Target: The name of a file that is generated by a program, e.g. executables or object files. A target can also be the name of an action to carry out, such as `clean`.

Prerequisite: A file that is used as input to create the target. A target often depends on several files.

Recipe: A recipe is an action that `make` carries out. A recipe may have more than one command, either on the same line or each on its own line. **Please note:** you need to put a tab character at the beginning of every recipe line!

Rule: A rule, then, explains how and when to remake certain files which are the targets of the particular rule. `make` carries out the recipe on the prerequisites to create or update the target. A rule can also explain how and when to carry out an action.

A rule tells `make` two things: when the targets are out of date, and how to update them when necessary.

2.2 A Simple Makefile

```
edit : main.o kbd.o command.o display.o \
  insert.o search.o files.o utils.o
  cc -o edit main.o kbd.o command.o display.o \
    insert.o search.o files.o utils.o
main.o : main.c defs.h
  cc -c main.c
kbd.o : kbd.c defs.h command.h
  cc -c kbd.c
command.o : command.c defs.h command.h
  cc -c command.c
display.o : display.c defs.h buffer.h
  cc -c display.c
insert.o : insert.c defs.h buffer.h
  cc -c insert.c
search.o : search.c defs.h buffer.h
  cc -c search.c
files.o : files.c defs.h buffer.h command.h
  cc -c files.c
utils.o : utils.c defs.h
  cc -c utils.c
clean :
  rm 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. To use this makefile to create the executable file called *edit*, type

```
make
```

To use this makefile to delete the executable file and all the object files from the directory, type

```
make clean
```

3 Basics of Variable References

To substitute a variable's value, write a dollar sign followed by the name of the variable in parentheses or braces: either '$(foo)' or '$foo' is a valid reference to the variable foo.

```
objects = program.o foo.o utils.o
program : $(objects)
	cc -o program $(objects)

$(objects) : defs.h
```

The rule

```
foo = c
prog.o : prog.$(foo)
	$(foo)$(foo) -$(foo) prog.$(foo)
```

could be used to compile a C program *prog.c*.Since spaces before the variable value are ignored in variable assignments, the value of foo is precisely *c*.

3.1 Setting Variables

To set a variable from the makefile, write a line starting with the variable name followed by '=' ':=', or '::='. Whatever follows the '=', ':=', or '::=' on the line becomes the value. For example,

```
objects = main.o foo.o bar.o utils.o
```

defines a variable named *objects*. Whitespace around the variable name and immediately after the '=' is ignored.

3.2 Variables Make Makefiles Simpler

```
OBJECTS = main.o kbd.o command.o display.o \
  insert.o search.o files.o utils.o
CC = gcc
edit : $(OBJECTS)
  $(CC) -o edit $(Objects)
main.o : main.c defs.h
  $(CC) -c main.c
kbd.o : kbd.c defs.h command.h
  $(CC) -c kbd.c
command.o : command.c defs.h command.h
  $(CC) -c command.c
display.o : display.c defs.h buffer.h
  $(CC) -c display.c
insert.o : insert.c defs.h buffer.h
  $(CC) -c insert.c
search.o : search.c defs.h buffer.h
```

```
17    $(CC) -c search.c
18  files.o : files.c defs.h buffer.h command.h
19    $(CC) -c files.c
20  utils.o : utils.c defs.h
21    $(CC) -c utils.c
22  clean :
23    rm edit $(OBJECTS)
```

Exercises

1. In 'makeExamples' folder, four files for a program are given. To compile the files and obtain an executable, simply run

```
1 g++ main.cpp hello.cpp factorial.cpp -o hello
```

   (a) Write a simple Makefile without any dependencies to build the program.

   (b) Write a Makefile with dependencies to build the program

   (c) Write a Makefile using dependencies and variables to build the program.

References

[1] R. M. Stallman, R. McGrath, and P. D. Smith. Gnu make. https://www.gnu.org/software/make/. 2020-02-06.