Instructor: Nicolas Savva

February 27, 2015

based on slides by Hussam Abu-Libdeh, Bruno Abrahao and David Slater over the years
(with additional material from software-carpentry)
Announcements

- A4 is due tomorrow night
see software-carpentry: make *
GNU Make

- manage compilation of programs written in languages like C
- used to automatically update any set of files that depend on another set of files
- Makefile describes how files depend on each other, and how to update out-of-date files
- makes use of patterns, rules, and variables to eliminate redundancy
- uses macros to control operation
myapp : file1.o file2.o
    gcc -o myapp file1.o file2.o

file1.o : file1.c macros.h
    gcc -c file1.c

file2.o : file2.c macros.h
    gcc -c file2.c

This Makefile describes the dependencies require to compile the C source file file1.c, file2.c (and header file macros.h) into the executable called myapp.
The basic syntax of a Makefile is

```
target:  dep dep dep
    [tab] cmd
    [tab] cmd
    [tab] ...
```

Such a rule builds the file `target` in the following way:

- Checks to see if `target` does not exists
- Checks to see if it is **older** than any of the dependency files `dep`
- If either of the above are true, the file `target` needs to be rebuilt and the commands are executed
- A command **must** start with a tab character!
Our example again

myapp : file1.o file2.o
       gcc -o myapp file1.o file2.o

file1.o : file1.c macros.h
         gcc -c file1.c

file2.o : file2.c macros.h
         gcc -c file2.c

This rebuilds the file myapp if either it does not exist or is older than either file1.o or file2.o. But make does more, also checks to see that file1.o and file2.o are newer than file1.c macros.h and file2.c macros.h respectively. If it is not it first compiles either of these files, then compiles myapp automatically!
The real advantage of `make` is it does as little work as possible. It only recompiles the files that need updating to insure the executable (or file) is update date.

To execute such a makefile, you simply call `make` in the directory of the makefile (assuming the makefile is called `makefile` or `Makefile`). If you want to specify a different filename run `make -f makefilename`.
You can also specify which target you wish to compile by typing

```make
make target
```

Make will execute the first target if none is specified. So in our example we could type

```make
make file2.o
```

To just recompile `file2.o` if we wanted to for some reason.
Variables in make

You can assign variables anywhere in a makefile by a declaration of the form

    name = value

The value assigned extends to the end of the line and may be a sequence of words. For example we could modify part of the script to

```
objects = file1.o file2.o file3.o
myapp : $(objects)
    gcc -o myapp $(objects)
```

- When referencing a variable you use the syntax $(name)
Other variables often occurring in makefiles include variables to hold program options as well as the programs themselves. This makes it easy to compile with debugging information by simply adding the appropriate command-line options to the variable instead of changing all the occurrences.

```make
CC = gcc
CFLAGS=-c
myapp : file1.o file2.o
        $(CC) $(CFLAGS) myapp file1.o file2.o

file1.o : file1.c macros.h
        $(CC) $(CFLAGS) file1.c

file2.o : file2.c macros.h
        $(CC) $(CFLAGS) file2.c
```
make also automatically defines a variable for every environment variable that exists. You can override these variables by simply declaring a makefile variable with the same name. This does not overwrite the environment variable in your shell.

If you use a variable that has not been defined, it is automatically assigned the empty string.
Phony targets

Suppose we want to have our makefile perform some tasks that do not create compiled applications or files when they finish. For example, suppose we want to remove the intermediate objects file1.o, file2.o and file3.o by using

```
clean :
    rm -f $(objects)
```

Then make clean will work perfectly unless there is a file called clean in the current directory. To enforce to make that the target clean is not meant to correspond to a file in the directory, you declare it as phony by typing

```
.PHONY : clean
```

clean :
```
    rm -f $(objects)
```
The commands we are can be any shell command. So for instance we could make a Makefile that is the following:

```bash
countfiles = file1 file2 file3
ioufiles = iou
BFLAGS =
.PHONY : all clean

all : lcount_index iou_index

lcount_index: lcount.sh $(countfiles)
    bash $(BFLAGS) lcount.sh $(countfiles)

iou_index: $(ioufiles) iouscript.sh
    bash $(BFLAGS) iouscript.sh $(ioufiles)

clean :
    rm -f lcont_index iou_index
```

So that make counts the lines in the files and updates the iou index whenever any of the files are updated or the iou file is updated (yes I know this is a silly example).
Many rules in makefiles are essentially the same except for the names of the target and dependencies. To compile a C file the rule is typically:

```
file.o : file.c
gcc -c file.c
```

Wouldn’t it be nice to be able to say something like ”for every target with a .o extension, it depends on the corresponding file with .c extension, and to build it, you invoke such and such?"
Well of course you can use **static pattern rules**! The general form is

```
targets :  patterntarget :  patterndep dep ...
  cmd
  cmd
...
```

Meaning: for all targets `targets`, if it matches `patterntarget`, then it depends on `patterndep` and possibly `dep` and other fixed dependencies, and to build it you execute the `cmds`. 
Some special make variables

- `@$` - current target
- `@<` - first dependency file
- `@^` - all dependency files
- `@*` - stem
- `@$` - all dependency files that are newer than target
- start a command with `@` to suppress output echoing
Here is an example for compiling a bunch of c files

\[ \text{objects} = \text{file1.o file2.o file3.o} \]

\$(\text{objects}) : %.o : %.c

\text{gcc -c $<}

- % matches any number of characters. Whatever matches % is called the stem
- $<$ is a makefile variable refering to the first dependency file
foo != bar
- checks if foo has been defined, if not it assign it the value bar

SOURCES=main.cpp hello.cpp factorial.cpp

OBJECTS=$(SOURCES:.cpp=.o)
- substitution reference, sets OBJECTS to be main.o hello.o factorial.o

You can use shell wildcards in target prerequisites and commands but be careful when defining variables. Thus

clean :
  rm *.o

does what you guess, but

objects = *.o

assigns *.o to objects (This would be ok if you you were going to do something like bash lcountr.sh $(objects) because here the shell will expand *.o.
Another Example

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

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
    $(CC) $(LDFLAGS) $(OBJECTS)

$(OBJECTS) : %.o : .cpp
    $(CC) $(CFLAGS) $(OBJECTS)

clean:
    rm -rf *.o
```
The following does the same thing:

```bash
.cpp.o :
   $(CC) $(CFLAGS) $<

%.cpp : %.o
   $(CC) $(CFLAGS) $<
```

The first is "old-fashioned" and obsolete (but you may still see it somewhere). Both compile all .cpp files into .o files.
There is a ton more that make can do with regards to implicit rules, expansion, static patterns and more. For a not so concise summary check out the make documentation

Overview: Version Control Systems (VCS)
Next Time