Makefiles Revisited

UNIVERSITY OF MINNESOTA

PROFESSOR DANIEL F. KEEFE

With our recent upgrades to the Duck program, we now have a lot of files to compile!

- Duck.h/cpp
- DecoyDuck.h/cpp
- FlyBehavior.h/cpp
- FlyNoWay.h/cpp
- FlyWithWings.h/cpp
- MallardDuck.h/cpp
- MuteQuack.h/cpp
- Quack.h/cpp
- QuackBehavior.h/cpp
- RedheadDuck.h/cpp
- RubberDuck.h/cpp
- Squeak.h/cpp
- main.cpp

Intro to Makefiles and the Make Program

- If you have used Visual Studio or Eclipse or some other integrated development environment, you might be familiar with the idea of a Project File or Project Settings or something like this.
- Makefiles are sort of similar. They are like a cross between Visual Studio Project Files and a shell script.
- The most common use for Makefiles is to:
    - Maintain a list of all of the files in your project.
    - Define the appropriate commands to run in order to (re)compile each file and link all the files together to create your program.
- Makefiles are read by a command-line program called *make*.

A Bit More High-Level Background

- Make can be used for lots of things, not just compiling. Makefiles are sort of like really fancy shell scripts.
- However, one of the key distinctions between Makefiles and scripts is that Makefiles know about *dependencies*:
    - If MyProgram.exe depends upon MyClass.cpp, and you edited MyClass.cpp more recently than the last time that I built MyProgram.exe, then I know that MyProgram.exe is out of date and needs to be rebuilt.
    - You have to manually tell make (by writing a Makefile) what these dependencies are in your program, but once you do, make can look at the last modification date for each of your files in order to automatically handle the dependencies.

How does it work?

- At the command line type:
  - make all
  - make some-other-target
- When invoked, the *make* program looks for a file you have written with all your targets and commands in it, typically named *Makefile.*
- Make searches the Makefile for the target you specified on the command line and then runs the commands listed under that target.

Inside a Makefile: Rules

```
target: [dependencies]
[TAB][command1]
[TAB][command2]
…
```

Using a TAB is critical. This has caused me much pain over the years!
The most common mistake in Makefiles is to use spaces rather than a tab.

Use Case

Suppose you want to compile helloworld.
You typed `make all` into your shell.
What happens?

Makefile

```
all: helloworld # default target

helloworld: main.o # linking rule
    g++ -o helloworld main.o

main.o: main.cpp # compile rule
    g++ -c main.cpp

clean:
    rm helloworld main.o
```

How make works

- Execution is recursive
- If a target in the Makefile is not needed (it is not a dependency of the target you specified on the command line), its rule is not processed
- Some useful common targets
  - *all*
  - *test*
  - *clean*

Variables

- Some items get repeated
  - compiler, flags, object files...
- We can define them as variables

Variables

```
CXX=g++
CXXFLAGS=-O3
all: helloworld

helloworld: main.o
    $(CXX) $(CXXFLAGS) -o helloworld main.o

main.o: main.cpp
    $(CXX) $(CXXFLAGS) -c main.cpp

clean:
    rm helloworld main.o
```

Static Pattern Rules

- Rules that apply to multiple targets
- Use patterns to define the names of the dependencies from the name of the target.
- Very useful... e.g. the rule to compile .cpp files into .o files is the same for all .cpp files so it would be nice to only specify it once.

Static Pattern Rules

```
targets: target-pattern: dependency-pattern
[TAB]commands
```

A specific example:

```
# This is a space-separated list of object files that need to be
# compiled to create our program
objects = apple.o banana.o

# This is a pattern rule that applies to everything in the list of
# object files specified above.
$(objects): %.o: %.cpp
    $(CXX) $(CXXFLAGS) -c $<
```

Automatic Variables

- The example on the last slide contained an "automatic variable"
- These are variables that are updated each time the rule is executed. You can use them in your pattern rules:

```
$@ - filename of target
$< - first dependency
$^ - all dependencies
$(@D) - directory of the target file (no trailing slash)
```

More Advanced Use of Make: Conditionals

- Make supports *if*, *if-else*, *if-else if-else* statements

```
ifeq ($(CC),gcc)
 libs=$(libs_for_gcc)
else
 libs=$(normal_libs)
endif
```

More Advanced Use of Variables

- Remember, variables are essentially strings of data.
- We can set one variable to another, and there are a few options for how to do this:
    - A = ${B} will not fill A in with the value of B until A is needed by a rule. So, you could set B to something else later in your Makefile and A will be assigned the most recent value of B whenever A is needed.
    - A := ${B} will assign the contents of B to A right away.
    - A ?= ${B} will only change A if it does not already have a value.
    - A += ${B} will append B to the current value of A.

Summary

- make: flexible, powerful, *not limited to recompiling*
- Used by everybody on every architecture.
- Extensively documented (170 pages)
- http://www.gnu.org/software/make/manual/make.html

Examples and Activities

Our FlashPhoto Makefile

Now, let’s write some Makefiles…

How about hello world?

1. Create a new rule. When, I run “make hello”, it should print:
   - “Hello Professor Keefe!”
   - use the “echo” command inside make to print something, also try “@echo”

2. Then, on the next line, make it print:
   - It is now: Tue Apr 19 07:10:52 CDT 2016
   - Note: you can get the current date with the unix “date” command.

3. Then, instead of “Professor Keefe”, change the first line to say Hello to whatever name is specified in the file name.txt.
   - Use the unix command: cat name.txt

Typically, we use make to “make” new files.

- This is how we use it when we compile programs, we follow the rules specified in the Makefile in order to “make” a new file, our program.

- Let’s update our Hello World so that rather than printing to the screen it prints the same information to a file named hello.txt.

    1. Update the name of the rule to be hello.txt, so now it should run when you type “make hello.txt”.

    2. Use the unix shell redirect command “>” to redirect the echo output to hello.txt.

Let’s talk about dependencies.

- Does “hello.txt” depend upon anything?

- Add “name.txt” as a dependency for the “hello.txt” rule in your Makefile.
- Now, try running “make hello.txt” a few times. What happens? What happens if you edit name.txt then re-run make?

Writing more complex rules: tips to keep in mind.

- For each line of a rule, make will start a **separate** shell using the unix sh command.
- This means if you want to run two commands in the same shell you need to put both commands on the same line separated by semicolons.
- After each line is executed, make checks to see if there is a non-zero exit code (i.e., an error). If so, it will print out “make: *** [rulename] Error 1” and then stop.

Here’s another challenge:

- Create a new rule called “testfiles” that:
  - Creates a new file that contains one line of text that reads: “File #1”.
  - Creates a second new file that contains one line of text that reads: “File #2”.
  - Uses `diff --brief file1.txt file2.txt` to test to see if the two files differ.
  - If they differ, print out “They differ!” and if they are the same print out “They are the same!”

Now, a refresher on Makefile variables

- I’d like to be able to easily change the filenames for file1.txt and file2.txt.
- Define these as variables in the top portion of your makefile and insert a comment so other programmers who use this makefile will know these are the two variables to edit.

How about a rule to build a C++ library?

- Remember, a C++ library is just an archived collection of the .o files that were compiled from a bunch of .cpp files.
- Given a bunch of .o files, use the following command to archive them and save them in mylibrary.a:

```
ar rcs  mylibrary.a <list of .o files>
```

`ar` = the GNU ar program for working with archives

`r` = replace items that have the same name with the newest versions

c = create an archive

s = include an index that will make linking faster when the archive is used later to build a program.

Creative Uses of Makefiles: A Brainstorming Exercise

- Example 1: My moment of “Makefile enlightenment” during graduate school :)
- Your examples??